0GiNr 技术论坛 ---零下安全·信心无限--- 计算机编程-计算机安全-反病毒-反木马-反 RootKit

www.0ginr.com

# 牛刀小试 VC++教程高清 PDF 版

本教程来自于 0GinNr 论坛,本人只是稍做整理,未做任何改动,在这里献给爱好 VC++、牛刀的 FANS 以及 0GiNr 论坛的忠实支持者…

制作:Mikko.Hu　Email:Mikko.Hu@msn.com　2008-09-12

**牛刀教学〇一一：串串糖葫芦**

呵呵，将教学移到新论坛来了，也不知根过来的人多不多，不过没关系，根过来的，估计肯定对编程还是趣兴趣的是不，所以牛刀这儿不求会员多少，只要有人，牛刀就有信心能将文章发下去。

请管理员还将我的《牛刀集》中帖子还给移过来，我就在这儿给大家开课了，呵呵

对了，这个主题改为 C/C++，我那《牛刀集》，呵呵，我专用了，可能和大家讨论一些我个人的想法，当然也有人会说：我文化程序不高啊什么的，有空还给大家补补文化，说不准能构思两个网络游戏，希望大家到时候来捧场啊，呵呵。

我们今天讲什么呢？我们今天将我们以前讲的数据类型给串串糖葫芦，呵呵，大家看过吧？不要买那东西吃，听说那东西不干净。（你瞧牛刀会管闲事吧？呵呵）

我们用什么串呢？当然用数据串，请看下面程序：

```
#include <stdio.h>

void main()
{
   int a,b,c,d,e,f,g;
   a=1;
   b=1;
   c=a+b;
   d=b+c;
   e=c+d;
   f=d+e;
   g=e+f;
   printf("%d\n%d\n%d\n%d\n%d\n%d\n%d\n",a,b,c,d,e,f,g);
}
```

可能看过以前的教程的朋友能看出来了，不就是我以前说的一个数列吗？

不错啊，可如果你看了这一段程序后，你可能觉得牛刀特冤，你看后面的形式几乎一样，却要定义那么多的变量，如果要你输出 1 0 0 个数构成的数列呢？岂不是要累死牛刀啊，是不？

于是，人们想出了一个方法，将这些同一种数据类的数据组成一组，称之为“数组”。

那么上面的程序就好弄了，看吧，数组加循环，牛刀的程序又变得简单了，呵呵：

```
#include <stdio.h>

void main()
{
   int a[7];
   a[0]=1;
```

```
    a[1]=1;
    printf("%d\n%d\n",a[0],a[1]);
    for(int i=2;i<7;i++)
    {
        a[i]=a[i-1]+a[i-2];
        printf("%d\n",a[i]);
    }
}
```

同样的功能，不同的程序，

大家很容易看出，后一种方式和以前我们讲循环一样，很容易改也 100

我就不为大家改了，大家改时注意啊，不要超范围啊，超过数据范围就可能出现负数，或出现其他现象了。我们不是知道，计算机中不同的数据有不同的范围，是吧？这其实是我们编程所必须要考虑的。

那么，数组怎么定义呢？

数据类型　数组名[数组长度];

其实就是这么简单，

当然我们上面的叫做一维数组，其实数组还可以有两维，三维……。

我们再教大家定义二维数：

数据类型　数组名[数组二维长度][数组一维长度];

什么意思呢？如：我们定义一个字符型数组：

```
char ch[5][5];
```

那么二维数组有什么优势呢？优势多了，举个例子吧，我们以前给大家做过的程序《拉灯》就是一个程序在二维数组上操作的例子，我们可以将二维数据看成一个面，我们可以在这个面上进行我们所需要的操作。

我们下面看一个例子：然后大家思考怎么操作，然后我们再讲如何操作。

有人发过一个“生命游戏，我在名人论坛上有软件，大家可以去下载。

我们下面来做一个。

规则：一个 5 0 × 5 0 的培养盘，原来里面充满着细胞（注：全部都有细胞）

细胞每次都会根据周围的细胞数量而出生或者死亡，如果周围有三个细胞，则本细胞生，周围有二个细胞则状态不变，其他情况细胞死亡。

我们下面以整数类型数组表示一个 5 0 × 5 0 的细胞盘。

数组 b[50][50]记录以后状态。

我们细胞将根据以上规则变化 2 8 次（注：我发现 2 8 次还比较好看，次数你可以自己改改看，我们用“*”打印出细胞图：

```
#include <stdio.h>

void main()
{
    int a[50][50],b[50][50],i,j,k,sum;
    for(i=0;i<50;i++)
        for(j=0;j<50;j++)
            a[i][j]=1;
```

```
    for(k=0;k<28;k++)
    {
        for(i=1;i<50;i++)
            for(j=1;j<50;j++)
            {
                sum=a[i-1][j-1]+a[i-1][j]+a[i-1][j+1]+
                    a[i][j-1]            +a[i][j+1]+
                    a[i+1][j-1]+a[i+1][j]+a[i+1][j+1];
                if(3==sum)
                    b[i][j]=1;
                else if(2==sum)
                    b[i][j]=a[i][j];
                else
                    b[i][j]=0;
            }
        for(i=1;i<49;i++)
            for(j=1;j<49;j++)
                a[i][j]=b[i][j];
    }
    for(i=0;i<50;i++)
    {
        for(j=0;j<50;j++)
            if(1==a[i][j])
                printf("*");
            else
                printf(" ");
        printf("\n");
    }
}
```

我们下面来看看分析一下程序：

int a[50][50];定义了一个数组，数组 b 也同样。

又定义了三个整型变量：i,j,k;

其中 k 规定了细胞变化的次数，而 i,j 定义了对数组的处理。

一般我们一开始最好对数组进行初始化，初始化成我们所需要的值，例如上面的被我们初始化成了 1：即细胞全活。

下面的 for(k=0;k<28;k++){...}语句定义了细胞变化的次数，注意我们 k 值在程序中根本没有用到。

下面几行则定义了我们的变化规则，我们看，我们在 a[i][j],中，可以在数组的每一维变量中使用表达式，以表达式的值来取得数组的相应位置的数据，这就是我们定义数组的本质意义了。

你看我们有了二维数组，数据在程序中就有了上下左右之分了是吧？

```
sum=a[i-1][j-1]+a[i-1][j]+a[i-1][j+1]+
    a[i][j-1]            +a[i][j+1]+
    a[i+1][j-1]+a[i+1][j]+a[i+1][j+1];
```

大家看，我们我们这是一行数据，但却分成了三行，这样一看，不就有了上下左右的感觉了吗？你弄那什么 i+1，还是 i-1 啊什么的，不就不容易错了吗？是不是？

上面的 sum 就是计算周围细胞的个数，下面就判断了，当它为 3 时，本细胞变活， 2 时不变，其它情况变为死（ 0 ）。

接下来的一个循环就是将 b[][]中的所有数据复制到 a[][]数组中去，再进行下一次变化。

而外面那个循环就是打印出相应的结果（用“*”打印出）

其实在Ｃ语言中也可以用画点的方法来打印，不过我们的教程中就不教大家了，因为我们以后会教大家更先进的编程技术－－ＡＰＩ，在Ｗｉｎｄｏｗｓ的窗体中绘制这些图像。

三维或三维以上的数组一样的操作，不过我们一般用得最多的就是三维，四维或以上的很少使用，但可以定义使用，但我想如果你定义一个四维数组，数组之间的关系，你构思可能也需要构思大半天了，也就是我们对种种算法的熟悉，才能使我们能够很好地操作数组。

我们下面就用数组给大家做个实用的东西：找素数：

素数，也叫质数，数学上的定义为：只有自己本身与 1 两个约数的数为素数。

在计算机中求素数的方法可能很多，我们既然用到数组了，我们就用数组来做一个筛，来筛出素数来。

当然我们的筛子就是一个数组：我们将之定为１００００即一万个数据，看哪些是素数。

我们用布尔型变量，true 表示此数据为素数，而 false 表示此数据不是素数。

我们只需要从 2 开始，将相应的素数的倍数给挖掉（设为 false）就行了，于是经过有限次变换，我们这里一万个数据，只需要进行 100 次变换，你想想 101 的倍数肯定到１００开外了，你可能说，那２０２不是１０１的倍数吗？那已经被我们在 2 的倍数中给挖了，所以小于１００的倍数即然都挖了，那只有大于１００的倍数了，那么１０１×１０１已经超过１０００００所以我们已经管不着那么多了，所以我们筛子挖的过程只要进行到１００就行了。

好了，下面准备挖了，大家看程序：

```
#include <stdio.h>
void main()
{
   bool ss[10000];
   int i,j;
   for(i=0;i<10000;i++)
       ss[10000]=true;//我们先假设全是素数
   ss[0]=false;
   ss[1]=false;//我们知道 0 与 1 肯定不是素数，先将它们除去
   printf("===========以下为小于 10000 的素数列表
==================\n");
   for(i=2;i<100;i++)
```

```
    {
        if(ss[i])//大家看，这儿的条件怎么是 ss[i]？因为 ss[i]本身就是布尔（逻辑）类型啊，是不是？所以我们直接用在这就行了。
        {
            for(j=i*i;j<10000;j+=i)//这是我们的筛子
            {
                ss[j]=false;//我们将其倍数筛去
            }
        }
    }
    for(i=0;i<10000;i++)
        if(ss[i])printf("%d\t",i);
    printf("\n");
}
```

怎么样？这是历史上著名的筛法求素数的方法。

下面讲什么呢？讲讲我们数组在内存中的存放。
我们一维数组是按顺序存的，而多维数组呢？按最后一维按顺序存放举个例子吧：
a[3][3]
那么存放顺序为：
a[0][0]
a[0][1]
a[0][2]
a[1][0]
a[1][1]
a[1][2]
a[2][0]
a[2][1]
a[2][2]
就是这个原则，先变最后一维，然后顺次向里面变。
可能你惊讶了，怎么只到 2？a[3][3]哪去了？
可能你一直纳闷这个问题吧？其实我们的定义：
int c[n];
它表示定义一个有 n 个数据构成的数组，而这个数组是从 0 开始表示的所以我们的范围就是：0 ~ n-1
所以上面，你只看到了 a[2][2]，而没有看到 a[2][3];
而我们程序中也绝啊，看：
int c[10];
for(int i=0; i<10;i++){...}
你看到了没，用了 i<10，不就表示到 9 为止吗？这就是我们操作数组的技巧，尽量不要用“<=”，就可以正常处理了。
我们为什么要说它在内存中的顺序呢？因为我们在合适的时候，我们可能会要用其它的办法使用它。（当然可能是……）不说了，我们教程序不能为大家教

得太深入，太深入可能也不一定是好事。（当然你也不要着急，我们早迟要说到它，然而不是现在）

行了，下面我们要讲什么呢？一种特殊的数组－－字符串：
怎么还要讲字符串啊？字符串不是讲过了吗？
当然，是讲过了，可讲过如何操作字符串了吗？
对喽，字符串的处理是我们编程中的一项重要内容啊。

下面我们先来讲，怎么样定义一个字符串：
```
char ch[20];
```
这么简单啊？这不是定义一个字符数组吗？你还真聪明，这就是字符串，字符的一串，呵呵。
看段程序：
```
#include <stdio.h>
void main()
{
   char ch[20]="abcdefg";
   printf("我们的字符串为：%s\n",ch);
}
```
当然，通过上面的程序，我们也知道了“%s”就是输出字符串吧？由于字符串总是一起被操作的，所以我们并不需要对字符中的数据进行单个输出，所以我们用了 ch，而没用 ch[参数]这种形式。因为我们的字符串太受用了，没必要对它进行单个操作是吧？当然如果想对它进行单个的操作也是可以的当然就可以用 ch[]将单个字符进行输出，是吧？
```
#include <stdio.h>
void main()
{
   char ch[20]="abcdefg";
   printf("我们的字符串为：%s\n",ch);
   printf("我们的第 4 个字符为：%c\n",ch[3]);//注意啊，第四个字符的下标为 3 啊，呵呵
}
```
当然，如果我们需要对字符进行处理，怎么办呢？
你们先看下面的程序能够运行吗？
```
#include <stdio.h>
void main()
{
   char ch[20];
   ch="abcdefg";
   printf("我们的字符串为：%s\n",ch);
   printf("我们的第 4 个字符为：%c\n",ch[3]);//注意啊，第四个字符的下标为 3 啊，呵呵
}
```
大家将上面的程序打进计算机，看看会出现什么？

编译不通过是吧？什么原因呢？
因为我们的“=”只能对数组中的数据进行操作，而不能对数组整体进行操作。
那你可能要问了，那为什么前一种就可能吗？
嗯，这的确是个问题，字符串在定义时可以为其进行赋值，并且数组在定义时也可对其进行符值，我们看下面的程序：

```
#include <stdio.h>
void main()
{
   int a[5]={1,3,6,7,9};
   for(int i=0;i<5;i++)
       printf("a[%d]=%d\n",i,a[i]);
}
```

其实就是这样，编译器允许数组在定义时对其进行赋值。
为什么可以这样呢？这样方便啊，你想想，程序中进行赋值，就必须要用到循环，而定义时是定义了一段内存，而在定义这段内存的同时将后面的数据也乘机放了进去，事实就是这样。编译器很好做，而你过后想对一个数组整体赋值，你的数据， C 语言怎么能知道你放在什么位置呢？是吧？
行了，正因如此，我们不能对数组进行这样的操作，所以我们有两种办法处理这个问题：
一种办法是对数组进行一个字符一个字符地赋值，例：

```
#include <stdio.h>
void main()
{
   char ch[20];
   ch[0]='C';
   ch[1]='h';
   ch[2]='i';
   ch[3]='n';
   ch[4]='a';
   printf("你输入的字符串为：%s\n",ch);
}
```

你的机器上输出的是什么情况呢？牛刀的计算机上输出了一串怪怪的字符：

```
你输入的字符串为：China烫烫烫烫烫烫烫汤
Press any key to continue
```

注：有些特殊字符复制不过来啊，大家将程序自己做上计算机看看。
为什么会出现这种情况呢？
因为我们的每个字符串在定义时，内存中可能就会有数据了，而我们没有为其赋值时，它的值就是内存中的数据（多么可怕啊，如果这一串数据被别人拿走了怎么办？如果这一串数据是一个程序怎么办？如果是病毒怎么办？？？）
那么为什么我们前面在定义处定义就不会出现这种情况呢？原来，我们的字符串中有一个字符，就是零字符，'\0'，也就是某一位为零，表示字符串的结束。我们也叫做“结束符”所以我们上面的程序只需要加一行，变成下面的形式：

```
#include <stdio.h>
void main()
{
   char ch[20];
   ch[0]='C';
   ch[1]='h';
   ch[2]='i';
   ch[3]='n';
   ch[4]='a';
   ch[5]='\0';
   printf("你输入的字符串为：%s\n",ch);
}
```

大家再运行看看，不会出现问题了吧？
所以我们如果要定义一个字符串，必须要比字符多定义一个空间专门放这个零，以表示字符的结束。
那我们想想，如果有人定义一个很巨大的数据，然后将其输出，不就可以将你内存的东西给拿去了吗？只要不定义结束符就行了呗。
所以说我们的计算机安全性，我们又多了一丝不安。

我们内存中，某一程序运行后，程序是没有了，但计算机并不清内存，所以如果我们运行某一程序：如：游戏程序，我们输入了密码，但我们运行某和序时，我们将这个游戏程序关了，你真的关了吗？没错，这段内存是用不着了，也没有任何程序去管它了，但计算机操作系统也懒得去理它，所以程序其实还在内存中如果没有重新分配，所以它还存在在那儿，而如果一旦某程序的字符串分配到了这段内存，呵呵，你的密码人家用一句printf("%s",ch);就可以打印出来了，恐怖吧？这在计算机中就叫做“内存泄露”，很让人头疼的。很可能被别人的程序通过合适的手段就给“偷”去了。你可要知道，你的密码在内存中绝对是没有加密的，所以你如果测试上一程序如果有可能，你发现自己的帐号或者密码，你也不要奇怪，不过这种程序是自己做的无所谓，如果是人家做的你就完蛋了，如果它被发送到人家邮箱中去了怎么办？

但大家也没必要怕，一是杀毒软件，二是上网不要乱开网页，乱下载东西，上网站就上牛刀的教学网，下载东西就下载牛刀做的软件，呵呵。

一个软件如果想多一个功能可绝对不是费事的，所以很多程序都被人们绑定了一种程序，人们叫他们做“木马”。

跟牛刀学编程的人做的程序中不允许绑定木马啊，没有规矩，是不成方圆的。

好了，还谈谈字符串的事吧。
第二个操作字符串的方法嘛，那就是：用函数。

```
#include <stdio.h>
#include <string.h>
void main()
{
   char ch[20];
   strcpy(ch,"我跟牛刀学编程");
```

```
    printf("你输入的字符串为：%s\n",ch);
}
```

大家看到了 string.h 了吧？这是牛刀教大家的第二个C库，它为我们做了一系列的字符串操作函数。
我们想对字符串进行操作，则必须要将这个库给包进来。

行了，这下牛刀只要本着学几个函数的精神，给大家介绍几个字符串函数就行了呗：
strcpy:拷贝字符串。
strlen:字符串的长度
看程序：

```
#include <stdio.h>
#include <string.h>
void main()
{
    char ch[20];
    strcpy(ch,"我跟牛刀学编程");
    printf("你输入的字符串为：%s\n 这个字符串长度为：
%d\n",ch,strlen(ch));
}
```

这是怎么回事啊，为什么“我跟牛刀学编程”，不只有七个字吗怎么结果是 14 呢？如果你将程序运行了，你一定会提出这样的疑问。
其实中文一个占两个字节,可程序在计算字长是按字节来计算的,所以多了一倍，明白了吧？

这下好了，我们用以前学过的加密技术，对这段文字加加密看看，牛刀开始教大家一些好玩的例子了，大家一定要上机啊，不上机后悔一辈子。

```
#include <stdio.h>
#include <string.h>
void main()
{
    char ch[20];
    unsigned int i;
    strcpy(ch,"我跟牛刀学编程");
    printf("没有加密之前字符串为：%s\n",ch);
    for(i=0;i<strlen(ch);i++)
        ch[i]^=15;
    printf("加密后，字符串变为  ：%s\n",ch);
    for(i=0;i<strlen(ch);i++)
        ch[i]^=15;
    printf("解密后，字符串还原为：%s\n",ch);
}
```

怎么样？看到什么了？
可以改变加密的密字，再看看，变成了什么了？

加密字串技术：

```
#include <stdio.h>
#include <string.h>
void main()
{
   char ch[20];
   char jm[20]="goodfinelistok";
   unsigned int i;
   strcpy(ch,"我跟牛刀学编程");
   printf("没有加密之前字符串为：%s\n",ch);
   for(i=0;i<strlen(ch);i++)
       ch[i]^=jm[i];
   printf("加密后，字符串变为  ：%s\n",ch);
   for(i=0;i<strlen(ch);i++)
       ch[i]^=jm[i];
   printf("解密后，字符串还原为：%s\n",ch);
}
```

这种技术叫做逐位加密，密字各位都不同，解密者只要得不到加密密字，密文是没办法破解下来的。

犬富傲玖儡玩析牌,肺

上面是密字，用的是上一种加密技术，大家有空将它解密（注：加密密字俺可不告诉你噢，因为告诉你就不好玩了。）大家将解码后的字符串发上来。让牛刀看看倒底我们的小“程序员”们水平怎么样，呵呵。

下面再给大家介绍几个字符串函数，千万要将头文件包进来噢。
我们不给大家举例子了，大家自己试着做一些例子，看能不能成功。并且测试自己做出来的例子，看和自己想的相不相同：

字符串连接函数 strcat（字符数组 1，字符数组 2）
连接两个字符数组中的字符串，把字符串 2 接到字符串 1 的后面，结果放在字符数组 1 中。
说明：
① 字符数组 1 的长度必须定义得足够大，以便能容纳连接后的新字符串。
② 在进行连接前，两个字符串的后面都有一个'\0'，连接时将字符串 1 后面的'\0'取消，只在新串的最后保留'\0'。

字符串比较函数 strcmp（字符串 1，字符串 2）
作用是比较字符串 1 和字符串 2。
说明：
比较的规则为:将字符串 1 与字符串 2 按从左到右的顺序逐个字符地进行比较(按 ASCII 码值大小)，直到出现不相同的字符或遇到'\0'为止。若全部字符都相同，则认为两个字符串相等；若出现不同的字符时，则以第一对不相同的字符的比较

结果为准。
该函数的返回值的含义如下：
① 字符串 1=字符串 2，返回值为 0
② 字符串 1>字符串 2，返回值为正整数
③ 字符串 1<字符串 2，返回值为负整数
注意：对两个字符串的比较，不能用关系运算符，如以下形式：
if(s1==s2) cout<< "s1=s2";
这是错误的，只能用
    if(strcmp(s1,s2)==0) cout<< "s1=s2";

以前我们没有这么详细地介绍一样东西，其实上面的也不是我介绍的，而是在别人那里复制过来的，目的就是详细地告诉大家所有的情况，让大家自己去构思属于自己的程序。

再说了，别人做的比牛刀好的，牛刀拿过来用，别人不如牛刀的，牛刀给大家做，这样一来，牛刀就一定是最好的，是吧？呵呵，牛刀开始有些自吹自擂了，不过也真是如此，我们学习的目的不是学习别人知识，而是为了使自己变得足够强大。我们在当学生的同时还要当老师，把我们自己的思想传播出去，你说是也不是？

好了，这节课就先讲到这儿了。

对了，再缀一句，我们使用的 string.h 库，其实也是别人编写好的函数，我们有空等函数与其他一些相关的知识教完了，你也可以编写出这些函数，甚至做得比他们做得还要好，功能还要强。（相信我，没错的，呵呵）

作业：
解密下面的密文，用的前一种加密方法，密文（保密）

洚髭滠杷疴怃鞊

呵呵，这节课比较刺激，你认为呢？一到新论坛，牛刀就给大家做好菜啊。

比较下面两个有何区别：
'a'
"a"

搞程序的人要特别地勤于思考，否则我们就只会照着别人的代码往电脑上打，自己一遇到某个任务，就很难把它做成程序了。也就是说：实践是学习编程的必要环节，我希望在这个新论坛上，不要出现朋友动不动来个“我不懂”，因为我们的论坛不会为了提高点击率而作论坛推广，希望大家都是想学习的好孩子。
提到数据类型，可能大家对基本的数据类型也掌握得差不多了，我们C语言其实是默认数据类型转换的，所以我们的数据类型的转换有默认转换与强制转换两种。

我们作为默认的转换，其实不说也罢，为什么，因为计算机都给我们完成了，我们去说它又干嘛吗？何况，即使是计算机可以默认转换了，我们也同样可以用

强制转换，所以我们没有必要学，但我还是提一下吧，是怎么个规则呢：

短数据向长数据转换，整型向实型转换，字符型向整型转换，单精度向双精度转换。

也就是默认的转换是尽量不损失原来的数值精度与数值本身的值。

举个例子吧：我们一个 1 字节的数据向 2 字节转换，只需跟据数值进行高位扩充就行，有两种扩充方式，分别争对无符号数与有符号数（补码），无符号数的扩充原则就是高位补零，而有符号数的扩充方式是高位所有位等于本位的最高位，怎么理解呢：

还是举－ 5 这个例子吧，如果是一个字节，那么－ 5 为：
１ １ １ １ １ ０ １ １，这是有符号数的补码。
那么如果扩充两字节时，高位就就与最高位一致：
看：

```
                １ １ １ １ １ ０ １ １
１ １ １ １ １ １ １ １ １ １ １ １ ０ １ １
```

怎么样？扩充以后还是－ 5 的补码，当然了，正数由于最高位为 0 ，所以高位扩充都填 0 了。

无符号数会怎么样，无符号数中有 ２ ５ ４ ，最高位为 1 如果按上面方式扩充，不就成了……（你自己去想吧，呵呵）

所以无符号数字高位就是以 0 的方式向高位扩充。

当然，从原理上，我们知道这些，做程序你更本没必要知道这些就例如单精度怎么向双精度扩充啊，ＶＣ的编译器都为我们做好了，根本不需要我们操心。

所以说，当然，我们下面要讲一种特殊的情况：条件。

大家以前看过我的教程，知道好多地方用到条件吧？

```
for(;条件;)
{
}
while(条件)
{
}
do
{
}while(条件);
if(条件)
{
}
else
{
}
...
```

当一个数据在这里时，即使其不是关系运算符，也会补自动转换成逻辑值，原则就是： 0 为假，非零为真。

好了，大家还知道我们以前做过计算从 1 到１ ０ ０ 之和的程序吗？我们用这种自

动转换再做一个看看：

```
#include &lt;stdio.h&gt;
void main()
{
    int sum,i;
    sum=0;
    for(i=100;i;i--)
       sum+=i;
    printf("从 1 加到 1 0 0 之和为：%d\n",sum);
}
```

瞧，它中间的条件竟然是 i，只要 i 等于 0 （假）了，循环自然停止运行。

再来一个，我们知道吗？字符串都是以 0 作为结束标志的，那么下面的加密程序就简单了：

```
#include &lt;stdio.h&gt;
void main()
{
    char ch[]="牛刀给大家来个比较长的字符串,看这个字符串会怎么样,你看看,前面的 ch[]中连一个界限符都没有，奇怪吧？它可以根据我们的字符串的长短，自动给一个长度作为下标，这样的好处是，分配的空间不浪费，好了，该字符串也可以结束了！ ";
    int i;
    printf("没有加密前，字符串为：    %s\n",ch);
    for(i=0;ch[i];i++)//奇怪吗？这种“条件”，哎，太方便了，你看，少了函数不说还不需要判断了，呵呵
       ch[i]^=24;
    printf("加密以后，字符串变为：    %s\n",ch);
    for(i=0;ch[i];i++)
       ch[i]^=24;
    printf("哎哟，都糊涂了，变过来吧：  %s\n",ch);
}
```

你看，这东西好，连 string.h 的头文件都不要包，问题就给处理了，呵呵。

当然，我们仅限于以上两种，大家可以用这种方法，尽量不要独出心裁，否则可能出现意想不到的情况。

举个例子吧，浮点数，理论上说有 0 但事实上可能没有，因为浮点数的运算精度不是太高，大家可以也注意了，10/3*3 都不能复原。

那么如果我们用 10-10/3*3 它的结果就不是零，而我们做程序时，可能正想到这种情况时结束，能结束得了吗？所以说，浮点数在这里就可能引起“死循环”。

那么我们一般处理浮点数怎么作为条件判断呢？

浮点数一般不用等号(==)作为判断条件，而一般改为<或<=构造条件。

什么意思呢？例如我们两个浮点数 a 与 b，我们原来想做一个条件：a==b，可一旦做此条件就可能出现永假的情况，所以我们可以改为：a-b<=0.00000001 这样一来，程序达到适合的精度就会使条件变成改变而退出我们此段程序。

那么自动类型转换电脑为我们做了，那么手动的类型转换呢？我们也称之为

强制类型转换，只要将类型用括号括上就行了，当然有以下几种形式：
(int)表达式
int(表达式)
(int)(表达式)
最后一种是我自己加的，其实我就是将上面两种情况整合了，这是强制转换成整型，其它类型依葫芦画瓢，你也可以写出来。

当然，一般来说，强制转换是可能会有精度损失的例如
0xabc5　　是一个两字节数据，那么我们作下面强制性转换呢？（注 0x 表示什么，还清楚吧？）
(char)0xabc5=0xc5
看，前面的 0xab 全没了，可悲啊。
所以说除非必要，我们的数制转换尽量留给计算机。

当然我们以后学到函数后，随着函数功能的不断加强，数制转换将会被很普遍地使用，由于暂时可能用不着，我们就不为大家举例子了。当然大家可以自己做些程序玩玩，还是很好玩的。

**牛刀教学〇一二：找到你的家**

牛刀在这个论坛上条二个教程了，不知大家上次那个字符串解密有没有解开？呵呵。（给大家提个醒啊，我们计算机是用作干什么的？能交给计算机的事就交给她做好了，她可是不知疲倦的老黄牛啊，呵呵〔我们经常说谁谁像老黄牛，我看谁都不像，电脑像，我弄个很简单的题目你就可能打退堂鼓，而人家电脑是兢兢业业，那才是真的老黄牛呢！〕）

行了，这节课我们找什么呢？我们找到你的家。怎么找呢？很简单：&

大家可能奇怪了，这不是“与”运算吗？不错，这个运算有两种使用方法，我们以前是在做双目（两个数之间的运算）运算，而我们这节讲的是单目运算。那么，什么是家呢？“地址”啊，找到你家的门牌地址，还不就是找到你的家了？
下面我们看一段程序：

```
#include &lt;stdio.h&gt;
void main()
{
  int a;
  printf("我是:%d,我家住在：%d\n",a,&a);
}
```

大家看，a，我们没为它赋值，可在我的电脑上却显示：
我是:-858993460,我家住在：1245052

这是怎么回事呢？这就是我们内存中的残留数据了，当然了，有些语言会在定义一个数据时自动为其清零，这种语言编出来的程序还是比较安全的（也就是说：想用那种语言做坏事就难了），可毕竟我们现在学的是C语言是吧？
但我们怎么知道后面的是他们的地址呢？
我们的原则就是：在一起定义的数据基本上是分配在一起的。
我们做个示例程序看看。

```
#include &lt;stdio.h&gt;
void main()
{
  int a;
  int b;
  int c;
  printf("%d\n%d\n%d\n",&a,&b,&c);
}
```

我的电脑输出的结果为：
1245052
1245048
1245044
大家看，前面是相同的，而后面５２，４８，４４，正好相差４，不就是整型的４个字节吗？

但大家可能纳闷，怎么会从大到小呢？这个地址（邻居是不错，但怎么住反了？）

这告诉我们，我们的数据是分配在栈上的，栈是一种先进后出的类型。也就是先进去的数，后出来，这就是栈，所以栈的地址是反的。那么我们再试试在后面加一个数组：

```
#include &lt;stdio.h&gt;
void main()
{
  int a;
  int b;
  int c;
  int d[10];
  printf("%d\n%d\n%d\n",&a,&b,&c);
  printf("下面放的是数组的地址……\n");
  for(int i=0;i&lt;10;i++)
      printf("%d\n",&d[i]);
}
```

大家可以看到，数组的存放顺序是顺序存放的，但也是存放在栈中。而且是同一个栈中。

当然，大家可以利用同样的技术，看看我们二维数组或者多维数组在我们内存中的存放情况。

当然了，既然能通过你找到你家，那能不能通过你家找到你呢？那就是另一个与取地址相反的运算——“*”。

这不是乘号么？当然也是，但在单目运算中是取得某个地址对应运算“&”相反的运算，找到一家的主人。

怎么回事呢？既然这两种运算是相反的运算（即为逆运算），那么组合在一起，运算不就抵消？真是这样吗？我们试试看吧：

```
#include &lt;stdio.h&gt;
void main()
{
  int a=34;
  printf("*&a=%d\n",*&a);
}
```

好了，既然*是取某地址相对应的数据，那么*后面一定是地址了是不？
所以我们可以用*来定义一个地址：

```
int *adda;
```

此时，我们的 adda 就是一个地址了，前面的数据类型说明了其为一个整型数据的地址。
于是我们在做程序中，可以用它指向一个数据的地址，一个数组的地址，甚至是一个函数的地址，这种运算我们会在后面详细系统地讲解，我们这一讲先给大家开个头而已。

这种定义地址的方式，在有些教材中会将他们定义为“指针”。

那么既然人家有这么个说法，我们就借用人家这个说法吧，不过大家知道喽，指针就是数据的地址，就这么简单。

可能你说，指针我们要知道他们有什么用呢？可在有些人那里，用处可就大了（呵呵）。

其实指针这东西就是好啊，好到什么程度呢？好到我们以后需要用专题对她来进行讨论。

那么我们这节要给大家说什么呢？我们给大家介绍一个函数：scanf

可能有些朋友如果看过C语言教程，可能就对牛刀不满了，人家早就讲了，你到现在才讲。

不错，可你能知道里面一些、东西的含义吗？

行了，简单介绍一下这个函数吧。

为什么说简单介绍呢？因为我们并不打算使你老在这种Ｄｏｓ环境下将程序做下去。我们以后真正编程时，我们根本不会做这种“控制台应用程序”的，这就是原因。我们现在为大家教做这种程序，完全是为了教学的需要，如果我直接叫大家怎么做出一个窗体来，大家绝对会很纳闷，到底是怎么回事啊，是不。

行了，闲言少叙，我们下面就先看一段程序：

```
#include &lt;stdio.h&gt;

void main()
{
  int a;
  printf("请你输入一个正方形的边长，我告诉你正方形的面积：");
  scanf("%d",&a);
  printf("你输入边长对应的正方形的面积为：%d\n",a*a);
}
```

为什么我们要放到这里面讲，就是这原因了，大家看，条二个参数为整数a的地址，为什么这里是地址呢？原来这个程序的功能是将我们在键盘上的数字存入对应的变量中的函数，既然你要将数据放到一个地方，你当然要告诉别人你放的这个地方啦是不是？（即，你要告诉人家你家的地址，人家才能送货上门啊，否则一切岂不是空中楼阁？

其他教程在前面一开始就讲到这个函数的直接后果，就是很多人做程序时容易将取于址符给丢了，为什么呢？因为他不知道这东西的作用啊，而我们这里如果丢了，程序一旦报错，我们也知道我们出错的原因，是不？

上面程序执行会停在某个字符串后，等待你的数据输入，如果你输入了后，

一按回车，电脑就会将结果反馈给你。

我们为什么要讲这个函数呢？因为这个函数可以使得我们做的一些程序具有通用性，我们做一些以前我们做过的程序，看看这些程序是怎样真正地为我们人类服务的吧：

```
#include &lt;stdio.h&gt;

void main()
{
  int a,b,c;
  int i,sum;
  printf("下面我们将给大家求你所定义的有限个数据之和：\n 请输入你要求的数据的最小值:");
  scanf("%d",&a);
  printf("最大的数呢（上限）?:");
  scanf("%d",&b);
  printf("步长为多大？（每次做一数据后将数据加多少再进行运算？):");
  scanf("%d",&c);
  sum=0;
  for(i=a;i&lt;=b;i+=c)
      sum+=i;
  printf("我算出来了，我厉害吧？结果是：%d\n",sum);
}
```

如果我们输入的为：1 100 1 那么结果为 5050

当然这个程序我们可以做任何等差数列之和了，只需要输入相应的上限、下限与步长（即公差），即可实现数列的求和运算。

像这样的一个通用性的软件，我们就可以将其编译对应的 exe 程序拿去发布了，软件的功能就是求任意等差数列之和，当然是在计算机可处理的范围之内。

当然，还可以一次输入多个数据：

```
#include &lt;stdio.h&gt;
int max(int a,int b)
{
  if(a&gt;b)
      return a;
  else
      return b;
}
void main()
{
  int a,b;
  printf("请输入两个数，我给你找出其中最大的数：");
  scanf("%d %d",&a,&b);
  printf("你输入的那个大数为：%d\n",max(a,b));
}
```

大家还记得求最大值的程序吧，这样一改，就又是一个软件了，呵呵，虽然软件的功能不是很强，但这种软件却是绝对的绿色环保啊。但可能这种软件你是赚不到￥的，呵呵。
上面的输入方式可以是两个数据用空格或用逗号隔开，其他形式也有，我们就不讲了，可能有些书上会说，怎么做一个程序是错误的例如：
scanf("%d,%d",&a,&b);
怎么输入是正确的，怎么输入是错误的。

我说实话也不解，教材怎么老教人家错误的知识啊，呵呵，教正确的，少一些也没关系，你说是吗？我们在这里就不告诉大家还有哪些情况了，当然我们以后如果介绍到互动程序，我们再给大家举一些正确的例子就行了。其实我们真正地做程序，我们还是比较规矩的是吧？错误的东西，那是考试时试卷想要叫大家判断的内容。

我真纳闷，考试为什么不告诉大家正确的知识，非要弄三个错误的知识给大家选择？

噢，说到选择，却是我们编程做出来的软件的强项啊，有空给大家做些软件，当然我只会给大家代码，或者告诉你怎么去构思一个软件，软件你自己去做吧，可以说，你如果能跟着牛刀走到今天，你已经有做软件的能力了，只不过是做出来软件可能功能并不是那么强罢了。

我以前做过一个叙古拉的作业，我们这节用 scanf 语句帮助大家做一个测试叙拉古猜想的程序：

```
#include &lt;stdio.h&gt;
void main()
{
  int i,k;
  printf("请输入你要测试的数据：");
  scanf("%d",&k);
  printf("你正在测试数据%d 的叙古拉猜想：\n%d",k,k);
  for(i=k;i!=1;)
  {
      if(i%2)          //注：为什么这儿可以不用判断，请大家思考
          i=i*3+1;
      else
          i/=2;     //什么意思？还记得吧？
      printf("-&gt;%d",i);
  }
  printf("\n");
}
```

这个程序的功能大家去测试测试，或许对你做以前的作业会有所帮助。

当然，这节课我们给大家讲了 scanf 语句，其实以后如果我们告诉了大家 cin 与 cout 语句，你可能觉得 scanf 语句与 printf 语句几乎就没用了。

但我们还要讲它，为什么呢？因为如果你以后接触到黑客的一些技术，则这些语句还会有用。

因为我们的 printf 语句与 scanf 语句有漏洞。

好，提到漏洞，我们下面再与大家讲一讲有关软件的漏洞吧。

牛刀一再强调，作为学习，我们必须介绍这些，但是我们使用时，则必须避开一些技术上的漏洞。而不要有意识地去使用他们。当然，关于上面两个函数有什么漏洞，牛刀不想讲，至少暂时不想讲。

牛刀要讲的就是我们字符的定义本身就的漏洞：

```
#include &lt;stdio.h&gt;
void main()
{
  char ch[11]="牛刀找漏洞";
  ch[10]=33;
  printf("我倒要看看会输出什么？      :%s\n",ch);
  printf("我再看看我能看到什么？%d\n",ch[14]);
}
```

牛刀原来以为程序对越界不会报错，但对于：

```
  char ch[2]="牛刀找漏洞";
```

来说，牛刀的编译器报错了，看样程序还是不允许字符越界的，所以这里没有漏洞。

牛刀继续向下找，牛刀将它打印出来，也没问题，牛刀不服，好，既然你有个 ch[10]，我也知道你是 0，为什么？字符结束符啊，呵呵，牛刀一气，我将你做一个值吧，等于 3 3 。

程序运行出来了，牛刀一喜，后面果然多出来了一些东西：

牛刀找漏洞!汤  ↕

牛刀好好兴奋噢，牛刀更不服，那么牛刀让你输出一个 ch[14]瞧瞧，结果，编译器竟然没有报错。牛刀兴奋哟，终于漏洞被牛刀给找到了。编译器竟然没有越界检查。

牛刀暂时就不用这个漏洞给大家做试验了，大家看看后面那些数字，多做几个例子，看看你能够想到什么？（当然喽，这不是牛刀的教学范围，别想牛刀给你启示。

牛刀为什么会将这个漏洞告诉大家，目的并不是想你们去利用它，而是为了让大家去避免它，你如果不用字符型，如果用实型数据，也同样可以越界访问，例如你定义了一个有 2 个参数的数组，结果却访问到了它的第三个数组值，会出现什么情况呢？

```
#include &lt;stdio.h&gt;
void main()
{
```

```
  int a=111111;
  int b=222222;
  int c=333333;
  int d[2]={444444,555555};
  printf("%d\n",d[3]);
}
```

程序有数组越界报错，但程序还可以运行，运行出来的结果嘛？？
大家自己去运行运行，看看结果，仔细分析分析，看看牛刀为什么会用这种111111,222222,333333...这些奇怪的数字。因为我们一旦明白了（注：如果不明白的话，去运行我前面讲找到你家时给大家做的一个程序，看看你家与邻居之间地址的关系，看看你找人是不是找到了别人的房间去了？找到了谁的房间去了呢？如果你看C语言的书，书上只会叫你做程序时不要越界，而牛刀既然教大家了，大家就要跟着牛刀分析看看越到哪一个邻居家中去了，这个邻居是不是友好呢？〔再说明一点，你对这个邻居感不感兴趣呢？如果你感兴趣的话……〕）

所以说，大家一定要学会分析程序，当然还要学会做这些分析程序，以便使自己对程序有一个全面的了解。你知道什么叫尽信书则不如无书？牛刀听有人说过：《尽信书》是一本好书，《无书》比《尽信书》还要好。

牛刀一直在强调实践就是这个原因，实践就是不要相信书本，或者说不要迷信书本，书本上只会说不能这样，而我们要知道的是：为什么不能这样，如果这样，那会怎么样？如果那样对我有好处的话……

当然，牛刀话大家一定要弄明白的原因就是你既然串到邻居家中了，你就不要对邻居进行……（例如进行更改、例如进行……），因为你的邻居很可能就是这个程序本身（因为程序也是在内存中执行的啊），如果你一旦改错了，那……你说你晕不晕？你的程序就变成了……

所以我们说过，这些更改必须要对内存中数据的一些情况有点了解，你知道为什么我们前面字符串后输出总是：
汤 ↕
吗？所以说，一定要动脑筋。

牛刀再强调一遍，这些东西不要改，至少在没有人指导情况下不要改因为如果一改，可能后果就不堪设想。（改那玩艺儿的确是一项技术，不说了，不能再说了……，这就是人们说的所谓的漏洞。

给大家讲故事吧。曾经有一个人进入了人家的系统，怎么进去的呢？因为人家系统的密码是允许输入８０位，你想谁会设定８０位的密码啊？那不是疯了？所以理所当然的第８１位开始后面的四位，就理所当然地定义了系统的转向地址。（转向的地址总得存在一个位置吧？对了，什么叫地址你还记得吧？那就是你的家啊，呵呵）
而这位可不，就是一个“疯子”，他啊，竟然输入了一个８４位的密码，呵

呵，前８０位可是随便输入的，可后四位却是精心设计的。

于是，程序判断密码不正确了，于是就想转到密码错误的处理程序去处理了吧，没想却转向了软件的运行部分，软件竟然被成功地启动了。这是为什么呢？留作作业吧，大家自己去想，你家的门牌号被别人给改了，呵呵，你自己还不知道吧？所以人家就进来了呗。

再讲一个故事，就是关系漏洞问题的，有一次，程序员给别人编写了软件，但老是被别人攻击，什么原因呢？一直都不知道，有人气了，是不是程序员的问题。但竟然真的有人一检查原代码，竟然在密码部分根据没有错误（我原来想说："一点错误都没有"的，但作为程序员做的程序，那是不现实的。呵呵）。那是什么原因呢？

原因当时的编译器ＴＣ２．０的编译器编译时，判断部分只要输入某个特殊的数值，判断的结果就总是为真，但后来一问做编译器的人员，人家自有道理啊，这是我们做软件测试时用的，你怎么办？于是，那一个程序在程序没有任何漏洞的情况下却因为编译器的漏洞而进入了"黑客"们的视线。

所以说，我们不要以为我们做程序一点漏洞都没有，说不准我们的编译器就有漏洞，我们的操作系统都有漏洞，没说嘛，人家微软系统中存在多少万的漏洞，人家自己都承认的，所以说，这些漏洞，为黑客的入侵埋下了伏笔。

学习编程我们为什么还经常要提到黑客呢？因为我们与黑客们使用的是相同的技术，因为黑客们都是一些高手。当然了，现在的黑客倒不一定是如此了，一些"黑客"（注：为什么我们还叫他们黑客呢？因为他们的确对别人造成伤害了）却只会使用软件。

一个黑客高手，软件应该都是自己编写的，说实话，那叫做得心应手。而使用软件就不同了，那仅仅是一种偷窃行为，而不是一种技术行为。

当然，现实可能很惨，黑客正常是得不到应有的惩罚的，好多黑客高手，一旦遇到公安部门的排查，都会遇到一些大公司会将这些黑客们给"保"下来。

当然也有故事，有些人将某个企业给搞垮了，成了公安部门的排查对象，可正在这时，却有一个大公司，愿意全额甚至带利息地赔偿这些公司的损失，唯一的要求就是这些人，我公司要了，需要原公司不追究此人的责任就行了，公安部门不逮捕就行了。这种示例好多好多啊！为什么？因为好多黑客正是一些猎头公司所猎取的对象啊。这些人一旦来到新公司，那就绝对是公司的一流高手。

所以说我们不能对黑客抱有成见，我们也可以当另一种人，中国人称这种人为－－红客。

当你遇到了别人技术上的漏洞，及时地告诉别人并且能够提出解决方案，这种技术人材，你说中国能不缺吗？

所以我们必须研究一些入门的技术，用我们的技术为人民造福（话有些大，有些空了不是？）

作业：

自己做一个程序，程序分两部分，定义几个数字，定义一个数组，使用数组对数字区发动攻击（注：牛刀没试，能不能"攻击"成功），看能否攻击成功。

这个作业牛刀也没做过啊，等会儿我自己也去做做看。

大家还玩过游戏吗？用过修改器吗？其实修改器我们就可以将它看作是一个黑客程序，通过搜索内存的方式搜到指定的字符串或数字串，然后对其进行攻击（修改）以达到修改游戏人物或者物品属性的目的，从而使游戏通关变得容易。像这个不会对别人电脑构成伤害的“黑客”程序，我们倒可以自己在自己的电脑中试试看，但建议最好对自己做出来的软件进行攻击，不然一旦遇到一些版权问题或者是系统问题你会不知道怎么回事或者是如何处理。大家千万要注意，这种修改行为可能使系统不稳定，所以我告诉大家一定要勤于思考。

其实说实话，C 语言的位操作功能十分强大，但也是说实话，牛刀很少使用，位操作一般会涉及到密码、加密、以及数字化压缩等方面的知识。

对了，大家都用过压缩软件吧？什么叫 zip 文件呢？再问大家什么叫 txt 文件呢？什么叫 exe 文件呢？

除了有限的是计算机设计人员规定的文件类型以外其它文件类型可以说基本上是我们自己规定的。

例如 exe 文件,com 文件,甚至现在的 txt 文件，都是系统规定好了的文件类型。

而为什么说其他的文件类型都是自己规定的呢？一个重要的特点就是要其他的相应的软件打开，从这一点上来说，txt 文件，doc 文件也是后规定的。

那么我们能自己规定一个文件吗？能，怎么规定呢？我们规定一种类型后，我们做相应的软件，与这种文件进行关联，或者说处理这种文件就行了。

当然喽，这一节牛刀不是教大家要做一个我们自己类型的文件，而是要和大家交流一下我们以后如果做文件中可能要用到的运算－－位操作。

再说一种情况了，如果我们要操作一个机器人，那么就要操作某一个电动机的运行，则也必须进行位操作，不过那个涉及到硬件的编程技术了，暂与牛刀的教程无关，咱们这里不做过多的讨论。

还有一咱情况，就是对网络数据的处理，也必须要用到位操作，但说实话，牛刀到现在还不知道怎么处理，为什么呢？因为没有资料啊，什么叫活到老学到老就是这样。

我们对网络的处理往往是用微软的Ｗｉｎｓｏｃｋ，而那是人家已经做好的，并且包括好多协议的一个接口啊。但却一定会有人学过后说，俺会做网络程序了，但你还不说，那基本也够用了，做个聊天程序啊什么的都能用了，所以说封装是一个好东西啊，它有利于知识产权的保护，但不利我们对语言的学习。行了，其实我们说，如果真正要自己从底层做一个网络程序，那还真不是一件很容易的事，至少我们要学习好多好多的知识。

对了，从学习知识的方面，我们做程序的就有优势了，可以这么说，你为一个专业的人士做程序，你对这个专业的学习一定会比学这个专业的士还要专业，你可能有些不信，但事实的确是如些，因为你想要计算机为你做事，做程序的人不了解的，计算机是肯定做不了的是不是？但为什么说比专业人士还要专业呢？既然他想你做程序他就是想偷懒，所以说好多的专业人士真的对这些东西如果一旦有了计算机了，他们有好多的知识就是在退化，没办法。

曾经有些程序说过，做过了医院ＣＴ方面的软件，她自己也可去当医生了，ＣＴ拿到手，她真的是比那些专家还要强啊。

所以我们既然不知道它的核心编程支持（但我们必须知道它的原理啊，这是不能不知道的。），我们暂时就不考虑它吧，我们还是来谈我们的位操作。

其实位操作我们以前也做过，不就早：&,|,^,!...吗？

还有几个，这就给大家讲：

<<,>>

这是什么呢？左移，右移，可能够用了，什么叫左移、右移呢？想想大家也知道啦

0 0 0 0 0 0 0 1 向左移：0 0 0 0 0 0 1 0 不就是乘 2 操作吗？原来 1，现在变成 2 了，是吗？

看程序：

```
#include &lt;stdio.h&gt;
void main()
{
  int a=1;
  a&lt;&lt;=1;
  printf("%d\n",a);
}
```

嗯，还真的不错，那么后面那 1 表示什么呢？表示移 1 位，当然，上面相当于：a=a<<1;当然你也可以改改看看，就相当于乘以 2 的 n 次方。

为了看起来舒服，可以都用十六进制的方式输入输出：

```
#include &lt;stdio.h&gt;
void main()
{
  int a=0xff;
  a&lt;&lt;=4;
  printf("%x\n",a);
}
```

这样，更容易发现移位的效果：

当然了，移到最高位的数据哪儿去了呢？丢掉了呗。当然如果你需要对移走数据进行操作，可以事先对其进行保存，那么怎么保存一位呢？

注：我们上面的没将最高位移啊，因为整型是四个字节，所以大家可以用 0xffffffff 去试试看。真的移走了，呵呵。

那下面我们就结合上面的运算给大家说说关于位的保存：

当然我们用 char 类型啊，一个字节，这样的话，比较好看。

我们随便做一个数据，例如 3 3 吧：

那么 3 3 的第四位是 1 还是 0 呢？

我们怎么知道呢？

我们只需要用一个知识：就是说在 if 语句中，条位为 0 则为假，否则为真，那么我们用下面的二进制：0 0 0 1 0 0 0 0，注意啊，我们这第四位是从第零位算起的，也就是说，上面的数据为：0x10,我们用她干什么呢？与 3 3 进行与运算啊，第四位是 1 如果与了还是真（不等于 0），那么第四位就是 1 了是不是？否则第四位就为 0 ）

看程序：

```
#include &lt;stdio.h&gt;
void main()
{
  char ch=33;
  if(ch & 0x10)
      printf("第四位数据为 1 ");
  else
      printf("条四位数据为 0 ");
}
```

我的程序打出的是：第四位数据是 0，我们试着用这种方法将这个数字转换成二进制看看啊（注：我们可合理地使用位移操作符：）

```
#include &lt;stdio.h&gt;
void main()
{
  char ch=33;
  char test=0x80;
  for(int i=0;i&lt;8;i++)
  {
      if(ch & test)
        printf("%d",1);
      else
        printf("%d",0);
      test&gt;&gt;=1;
  }
  printf("\n");
}
```

大家将程序做过去看看，牛刀用电脑上的计算机验证了一下，怪了，不正确，为什么呢？牛刀不服，做了个测试的程序（注：牛刀将自己的思考过程都写出来了，错误也写出来了，大家也要多思考，多做程序才是。

```
#include &lt;stdio.h&gt;
void main()
{
/*    char ch=33;
  char test=0x80;
  for(int i=0;i&lt;8;i++)
  {
      if(ch & test)
        printf("%d",1);
      else
        printf("%d",0);
      test&gt;&gt;=1;
  }
  printf("\n");*/
```

```
  char test=0x80;
  printf("%x\n",test);
}
```

一运行程序，噢，我们在做：

printf("%x\n",test)时，test 被强制转换成整型了，所以说高位的并且是有符号的整型，所以符号位都移进来了，所以不符合我们的要求了。这可怎么办？

对了，大家看到“/*  */”了吗？这就是 C 语言中的注释，就是传统的多行注释，我们在多行不想要时，这样做挺方便的，并且有些人会在这个做自己程序的版权信息，这多好？

以上程序告诉我们，对字符不能用 0x80 的方式赋值，怎么办呢？我们改了成下面的形式：

```
  unsigned char test=0x80;
```

终于成功了，原来 char 类型也有有符号型与无符号型啊，注意了，我以前真的没注意这一点，char 有符号型如果超界了，竟然会自动转换为整型，所以我们以后想在程序中做些手脚，必须注意这些了，不注意也不行是吧？所以牛刀将程序加了一点料：

```
#include &lt;stdio.h&gt;
void main()
{
  char ch=33;
  unsigned char test=0x80;
  for(int i=0;i&lt;8;i++)
  {
      if(ch & test)
        printf("%d",1);
      else
        printf("%d",0);
      test&gt;&gt;=1;
  }
  printf("\n");
}
```

一切 O K 啦。

我们竟然可以将其一位一位转换成二进制，我们还不会对位进行操作吗？不可能。

这下我们可以去找一些文件的格式，看看他们到底是怎么构成的什么压缩文件啊什么的，如果我们一旦知道了他们的构成方法，我们就可以自己来做软件解压缩了是吧？

好了，再告诉大家一些压缩的原理吧。其实压缩是建立在统计学的基础上的，例如假如一段文字如下：

abcdddcccbaaabdadadbcdbaddbcbdbaddbcdbbadbdbcbdbadbcdbadbcbdbadbaddc

大家看，以上只有 a,b,c,d 四种字母，那么我们保存她们就不需要一个八位的数据量，而只需要两位就够了是不？这样一来，我们设：

a=00
b=01
c=10
d=11

abcd
ddcc
cbaa
abda
dadb
cdba
ddbc
bdba
ddbc
dbba
dbdb
cbdb
adbc
dbad
bcbd
badb
addc

我们按上面规则进行替换：

00011011
11111010
10010000
00011100
11001101
10110100
11110110
01110100
11110110
11010100
11011101
10011101
00110110
11010011
01100111
01001101
00111110

于是，我们的 6 8 个字符就压缩成了 1 7 个字节，也就是相当于压缩率为 2 5 %

呢，压缩率高吧？呵呵。
当然其实对于字符的压缩率没有这么高，我们这是建立在一个我们自己做的代码的压缩上面才会出现这种情况的，其实对于真正的字符压缩是采用统计学原理，将字符变成不定长的二进制串，然后按 8 位一字节来压缩的。当然理论基础在这儿就不地讲解了。

还有一种就是代码替换假如我们做一个系统，系统只认得大写字符，怎么办呢？我们很简单的一个办法就是代码的替换。
举例吧，怎么替换呢？

我们假设 Z 为替换码，那么 Z 字符在这里就替换成 ZZ 其它不变，然后就可用 ZA,ZB,ZC,ZD...来替换各种的符号啊，字符啊等等当然这种技术也用在我们的 HTML 语言中，其实 HTML 并不是一个编程语言，它的结构就是使用<>来作为一些替换符，而有些论坛也用[]来作为替换符，例如我们的论坛就是这样，所以当我们前一讲用到数组时用来[ i ]时就出现了问题了，发上来的帖子后面全部变成了斜体，大家不信可以在论坛上试试，挺好玩。

其实在网络上，这种替换是缕见不鲜，可以就，没有替换，就没有我们今天的网络，因为我们网络是二进制的网络，网络上的各种技术还不很完善（真的不很完善，就连时钟同步的问题都解决不了）。所以我们就必须用到这种替换技术，来解决一些同步的问题，如果我们发现不满足替换规则，那么时钟就可能出问题了，于是就会发出请求信号要求时钟同步，当然了，详细内容与我们编程关系不密切，我们就不多讲了。其实很古老的电报技术也是一种替换技术，不过那时候没有计算机，电报员很是辛苦，需要背好多电报码。你想想字母文字只有廿六个字母，而中国的汉字则是成千上万，可苦了我们中国技术员了，背的量那个是大啊（打仗很新辛苦，可后面的技术员更辛苦〔当然，我怀疑当时是不是用的汉语拼音。〕）。

行了，位操作的技术就讲到这里了，希望大家多实践，多学习，程序之路很漫长。

**牛刀教学〇一三：两种运算符**

教学到此，C 部分的基础部分就要做总结了，不知大家学习的怎么样？这一讲我们介绍两种特殊的运算符

对了，和论坛管理员说一下，现在的论坛有些转换字符会与程序犯冲啊，可以改一改论坛的转换字符啊，不知道是否可以，例如上面可以大括号“{}”作为转换标识符，这样就不容易出现问题了。

我们这一节讲什么呢？还讲运算符，讲我们C语言中很特别的两种运算符：逗号运算符与三目运算符
逗号运算符：“,”，顾名思义，一个逗号，呵呵
三目运算符：“?:”顾名思义，可以对三个表达式进行运算。也呵呵（不要笑）

一个一个来讲，逗号运算符什么功能呢？它的功能就是返回最后一个逗号后面的表达式的值。所以它的表示形式为：
表达式 1 , 表达式 2 , 表达式 3 , . . . , 表达式 n
返回表达式 n 的值，就这么简单，举个例子吧：

```
#include <stdio.h>
void main()
{
    int a,b,c,d,e;
    a=(b=1,c=b+1,d=b+c,e=d*d);
    printf("a=%d\n",a);
}
```

大家先不运行，想看看它等于多少，还后再运行，看看自己想的对不对。
我们再重做一个，为大家分析。（注：大家也可将上面的括号去掉，看看能不能正确得出原来结果，然后再思考，我们用这些表达式是不是一定要用括号。

```
a=(b=c=d=3,e=(++b)*(c=b+d));
```

我们看看这些，可能你会问，这个运算表达式还正确吗？真的，连牛刀都怀疑了，你可以将它拿到ＶＣ中运行看看。
看，由这个逗号前第一个看，我们可以分解成以下三句：

```
a=3;
b=3;
c=3;
e=(++b)*(c=b+d);
```

前三句大家一看就懂，那后面最后一句我们可以分解成下面三句：

```
++b;//b 自加，那么之后的结果为 b=4;
c=b+d;//c=4+3,结果为 7
e=b*c=4*7=28;
```

大家去运行看看结果是不是 28
当然，我们看，这多好，五六句可以并成一句，这可有用啊，大家看下面的程序：

```
#include <stdio.h>
void main()
{
    for(int i=1,sum=0;i<=100;sum+=i,i++);
    printf("sum=%d\n",sum);
}
```

大家看，怎么样？我们将上面的程序给分来开看啊，其实就是两句：

```
#include <stdio.h>
void main()
{
    for(
        int i=1,sum=0;
        i<=100;
        sum+=i,i++
        );
    printf("sum=%d\n",sum);
}
```

这种形式我们以后会经常用到的，将一个括号中的内容全都分开注意我们不里不是函数，所以逗号“,”在这里的作用仅仅是逗号表达式，而在函数中就不对了，在函数中，逗号将被用着分界符，到时间大家注意就是了。

其实逗号运算可以用在任何的表达式的地方，有时很方便，或者就是编程的一些小技巧，我们再看看下面对字符串加密的例子吧：

```
#include <stdio.h>
void main()
{
    char ch[]="请看 for 语句中对逗号运算符的使用情况";
    for(int i=0;ch[i];ch[i]^=12,i++);
    printf("加密后的字符串为：%s\n",ch);
}
```

行了，不深讲了，建议大家使用这些技巧尽量少一些，因为逗号运算符有时会使得程序结构不清晰，程序阅读困难。有时程序的确编写挺方便，但不方便理解与阅读的程序，我们至少说这种程序的风格不好，你说是不？

咱们再来讲讲三目运算符“?:”

它的运算原则是：

条件?表达式 1 :表达式 2

它的运算规则为：条件为真则运算表达式 1 ，否则运算表达式 2

我们再来举个例子吧：

```
#include <stdio.h>
void main()
{
    int a,b;
    printf("请输入两个数字：（中间用空格隔开）:");
```

```
    scanf("%d %d",&a,&b);
    printf("你输入的较大的那个数字是：%d\n",a>b?a:b);
}
```
大家看，我们这里少了一个函数，这就是三目运算符的优势之一，呵呵，我们在这种以某种条件二选一的情况下，用这个运算符可是特方便哟。
我们看：
条件?表达式 1 :表达式 2
就相当于一个语句：
```
if(条件)
    表达式 1 ;
else
    表达式 2 ;
```
你看看哪个方便？

行了，大家将前面的知识尽量系统化，我们下一讲进入一个新的专题：函数
今天星期天，不布置作业了，大家作好知识的巩固就行了。
呵呵，其实根本谈不上技术，就谈谈汉字的表示问题吧：其实作为一个中国人，使用计算机,你不处理汉字,怎么行呢？至少说一点民族自豪感都找不到,是吧？
我们怎么将汉字真正地做出来呢？
我们也别看什么材料，我们自己用程序自己做看看：

我们曾经说过，汉字只能用字符串表示，那么几个字符构成的字符串就行了呢？
汉字本身两个字节，我们说过了，再加一上个结束符，当然是三个字节啦。我们想当然，“阿”这个字在汉字库中必然在前面，我们就先用这个字测试一下吧，呵呵
```
#include <stdio.h>
void main()
{
    char ch[3]="阿";
    printf("%d,%d\n",ch[0],ch[1]);
}
```
结果：
-80,-94
不行，我们不能要负数，于是，我们改程序（注：大家看牛刀做程序的思路啊）。
```
#include <stdio.h>
void main()
{
    unsigned char ch[3]="阿";
    printf("%d,%d\n",ch[0],ch[1]);
}
```
我改成了无符号型，出现下面的结果：
176,162
好，我们想想：

一：汉字必须与西文字符区分，所以汉字最高位应该为 1 ，这一点符合
二：汉字至少是在同一个地方做吧，所以我们想，上面的 1 7 6 绝对不应该是最起始位置，所以我们做程序应该从 1 6 2 测试，我们不要弄一个了，我们当然要弄一串，那样好看：

```
#include <stdio.h>
void main()
{
    unsigned char ch[3]="阿";
    for(int i=162;i<200;i++)
        for(int j=162;j<255;j++)
        {
            ch[0]=i;
            ch[1]=j;
            printf("%s",ch);
        }
}
```

我们外层循环比较少，因为我们是为了测试，所以不需要用那么多，否则显示时间有些长。于是，有了下面的结果：

ⅱⅲⅳⅴⅵⅶⅷⅸⅹ 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.(1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)(14)(15)(16)(17)(18)(19)(20)①②③④⑤⑥⑦⑧⑨⑩ ㈠㈡㈢㈣㈤㈥㈦㈧㈨㈩ Ⅰ……

怎么罗马数字从 2 开始了，前面应该还有一个字符，所以我们改为 1 6 1

```
#include <stdio.h>
void main()
{
    unsigned char ch[3]="阿";
    for(int i=161;i<200;i++)
        for(int j=161;j<255;j++)
        {
            ch[0]=i;
            ch[1]=j;
            printf("%s",ch);
        }
}
```

噢，对了，前面还有个“啊”，牛刀糊涂，或者说是记错了，但我们分析事物就应该有一个过程，有错误可以改嘛

挺好看，牛刀就不给大家贴上来了，大家自己运行看一看：我们再将程序换换行，改改程序：

```
#include <stdio.h>
void main()
{
    unsigned char ch[3]="阿";
    for(int i=161;i<180;i++)
```

```
    {
        for(int j=161;j<255;j++)
        {
            ch[0]=i;
            ch[1]=j;
            printf("%s",ch);
            if(j%10==0)printf("  j=%d\n",j);
        }
        printf("\n   i=%d\n",i);
    }
}
```

好家伙，牛刀发了，这么舒服啊，牛刀看到了，
在[161,161 ~255]位置，定义了一些标点符号、数学符号与一些特殊符号：
　、。·ˉˇ¨〃々—　j=170
～‖…‘’　“”〔〕〈　j=180
〉《》「」『』〖〗【　j=190
】±×÷∶∧∨∑∏∪　j=200
∩∈∷√⊥//∠⌒⊙∫　j=210
∮≡≌≈∽∝≠≮≯≤　j=220
≥∞∵∴♂♀°′″℃　j=230
$¤¢£‰§№☆★○　j=240
●◎◇◆□■△▲※→　j=250
←↑↓〓
　i=161
在[162,161 ~255]位置，定义了一些数字符号：
ⅰⅱⅲⅳⅴⅵⅶⅷⅸⅹ　j=170
　　1.2.3.4.　j=180
5.6.7.8.9.10.11.12.13.14.　j=190
15.16.17.18.19.20.(1)(2)(3)(4)　j=200
(5)(6)(7)(8)(9)(10)(11)(12)(13)(14)　j=210
(15)(16)(17)(18)(19)(20)①②③④　j=220
⑤⑥⑦⑧⑨⑩　㈠㈡　j=230
㈢㈣㈤㈥㈦㈧㈨㈩　j=240
ⅠⅡⅢⅣⅤⅥⅦⅧⅨⅩ　j=250
ⅪⅫ
　i=162
在[163,161 ~255]位置，定义了一些中文符号包括中文字母：
！＂＃￥％＆＇（）＊　j=170
＋，－．／０１２３４　j=180
５６７８９：；＜＝＞　j=190
？@ＡＢＣＤＥＦＧＨ　j=200
ＩＪＫＬＭＮＯＰＱＲ　j=210
ＳＴＵＶＷＸＹＺ［＼　j=220

] ^ _ ` a b c d e f  j=230
g h i j k l m n o p  j=240
q r s t u v w x y z  j=250
{ | } ~

i=163

在[164,161 ~255]位置，定义了一些日文平假名：

ぁ あ ぃ い ぅ う ぇ え ぉ お  j=170
か が き ぎ く ぐ け げ こ ご  j=180
さ ざ し じ す ず せ ぜ そ ぞ  j=190
た だ ち ぢ っ つ づ て で と  j=200
ど な に ぬ ね の は ば ぱ ひ  j=210
び ぴ ふ ぶ ぷ へ べ ぺ ほ ぼ  j=220
ぽ ま み む め も ゃ や ゅ ゆ  j=230
ょ よ ら り る れ ろ ゎ わ ゐ  j=240
ゑ を ん  j=250

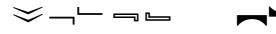
i=164

在[165,161 ~255]位置，定义了一些日文片假名：

ァ ア ィ イ ゥ ウ ェ エ ォ オ  j=170
カ ガ キ ギ ク グ ケ ゲ コ ゴ  j=180
サ ザ シ ジ ス ズ セ ゼ ソ ゾ  j=190
タ ダ チ ヂ ッ ツ ヅ テ デ ト  j=200
ド ナ ニ ヌ ネ ノ ハ バ パ ヒ  j=210
ビ ピ フ ブ プ ヘ ベ ペ ホ ボ  j=220
ポ マ ミ ム メ モ ャ ヤ ュ ユ  j=230
ョ ヨ ラ リ ル レ ロ ヮ ワ ヰ  j=240
ヱ ヲ ン ヴ ヵ ヶ  j=250

i=165

在[166,161 ~255]位置，定义了一些希腊字母与中文竖排版符号：

Α Β Γ Δ Ε Ζ Η Θ Ι Κ  j=170
Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ  j=180
Φ Χ Ψ Ω  j=190
α β γ δ ε ζ η θ  j=200
ι κ λ μ ν ξ ο π ρ σ  j=210
τ υ φ χ ψ ω  j=220
j=230
  j=240
j=250

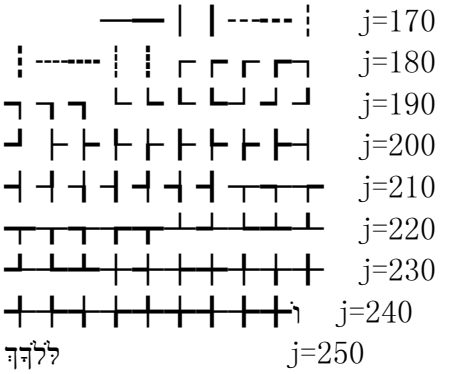
i=166

在[167,161 ~255]位置，定义了俄文字母：

А Б В Г Д Е Ё Ж З И  j=170
Й К Л М Н О П Р С Т  j=180

У Ф Х Ц Ч Ш Щ Ъ Ы Ь  j=190
Э Ю Я  j=200
а б  j=210
в г д е ё ж з и й к  j=220
л м н о п р с т у ф  j=230
х ц ч ш щ ъ ы ь э ю  j=240
я  j=250

i=167
在[168,161 ~255]位置，定义了一些中文汉语拼音符号与旧汉语拼音符号：
āáǎàēéěèīí  j=170
ǐìōóǒòūúǔù  j=180
ǖǘǚǜüêɑńňǹ  j=190
ng ㄅㄆㄇㄈ  j=200
ㄉㄊㄋㄌㄍㄎㄏㄐㄑㄒ  j=210
ㄓㄔㄕㄖㄗㄘㄙㄚㄛㄜ  j=220
ㄝㄞㄟㄠㄡㄢㄣㄤㄥㄦ  j=230
ㄧㄨㄩ  j=240
  j=250

i=168
在[169,161 ~255]定义了一些制表符号：

j=170
j=180
j=190
j=200
j=210
j=220
j=230
j=240
j=250

i=169
在[170 ~175,161 ~255]没有规定，我们可以制作自己的字库，以满足各种行业的需要
在[176 ~247,161 ~255]就是我们的汉字库了很壮观。
啊阿埃挨哎唉哀皑癌蔼  j=170
矮艾碍爱隘鞍氨安俺按  j=180
暗岸胺案肮昂盎凹敖熬  j=190
翱袄傲奥懊澳芭捌扒叭  j=200
吧笆八疤巴拔跋靶把耙  j=210
坝霸罢爸白柏百摆佰败  j=220
拜稗斑班搬扳般颁板版  j=230
扮拌伴瓣半办绊邦帮梆  j=240

榜膀绑棒磅蚌镑傍谤苞　j=250
胞包褒剥
　　i=176
薄雹保堡饱宝抱报暴豹　j=170
鲍爆杯碑悲卑北辈背贝　j=180
钡倍狈备惫焙被奔苯本　j=190
笨崩绷甭泵蹦迸逼鼻比　j=200
鄙笔彼碧蓖蔽毕毙毖币　j=210
庇痹闭敝弊必辟壁臂避　j=220
陛鞭边编贬扁便变卞辨　j=230
辩辫遍标彪膘表鳖憋别　j=240
瘪彬斌濒滨宾摈兵冰柄　j=250
丙秉饼炳
　　i=177
……
……
……
槟榇虚谊髌靓雩雳雯霆　j=170
雾霈霏霎霪霭霰霾龅龆　j=180
龅龆龇龈龋龌龊黾鼋鼍　j=190
隹隼隽雎雒瞿雠銮銮鎏　j=200
錾鏊鏊鎏鑝鑫鱿鲂鲅鲆　j=210
鲇鲈稣鲋鲎鲐鲑鲔鲑鲗　j=220
鲚鲛鲞鲟鲠鲴鲢鲣鲋鲦　j=230
鲧鲨鲿鲡鲎鲠鲫鲵鲳鲱　j=240
鲲鲶鲷鲴鲻鲼鲽鳄鳅鳆　j=250
鳇鳌鳊鳋
　　i=246
鳖鳍鳎鳏鳐鳓鳔鳕鳗鳘　j=170
鳙鳜鳝鳟鳢靼鞅鞑鞒鞔　j=180
鞯鞫鞣鞴鞘骶骰骷鹘骶　j=190
骺骼髁髀髅髂髋髌髑魅　j=200
魃魇魍魉魈魑飨餍餮饕　j=210
餼髟髡髦髯髫髻髭髹鬈　j=220
鬃鬓鬟鬣麽麾縻鹿麋麈　j=230
麋麒麈麝麟黛黜黝黠黟　j=240
黢黩黧黡黲黯鼢鼬鼯鼹　j=250
鼷鼽鼾齄
　　i=247
我们可以看到，前字的字库是按汉语拼音顺序排的，后面的字库是按部首排的，总字数应该是：９４×（２４７－１７５）＝６７６８个汉字，字库是够用的了，呵呵，一般情况下，不搞文字学，正常可以使用了。
[248 ~255,161 ~255]也没有定义，也可以作为造字字库使用。
　　你瞧，我们通过分析，我们就知道了汉字的分布、个数、功能、使用方法等

等一系列的东西，所以我们做程序一定要多动手，多动脑，我们的学习不一定要靠书本，那么我们一般使用的是十六进制表示，那大家只要将程序中的%d 换成%x 就可以看到结果了，汉字的区位码为：
[A1 ~FF,A1 ~FF]
功能大家可以象上面一样的去分析。

我们也可以打开造字程序，一看，我们分析的不错，造字程序就是从ＡＡＡ１开始制作造字程序的当然了，为什么从ＡＡＡ１而不从Ａ１Ａ１，大家可以自己去分析一下。

多动手，勤思考，这是做程序的唯一入门途径。否则，我们就只能够将别人的代码敲进电脑，我们只能有这种能力。（牛刀语）

牛刀老师 提一个问题哈
这个程序
[cpp]#include <stdio.h>
void main()
{
  char ch[]="请看 for 语句中对逗号运算符的使用情况";
  for(int i=0;ch;ch^=12,i++);
  printf("加密后的字符串为：%s\n",ch);
}
[/cpp]是不是就是相当于这个：
[cpp]#include <stdio.h>
void main()
{
  char ch[]="请看 for 语句中对逗号运算符的使用情况";
  for(int i=0;ch;i++);
  ch^=12;
  printf("加密后的字符串为：%s\n",ch);
}
[/cpp]
如果是的话 那么是不是又相当于这个：

[cpp]#include <stdio.h>
void main()
{
  char ch[]="请看 for 语句中对逗号运算符的使用情况";
  for(int i=0;ch;ch^=12,printf("加密后的字符串为：%s\n",ch),i++);
  ;
}
[/cpp]但是我运行第三个程序的时候 结果很杂乱 不知道是什么原因
还请牛刀老师指教 谢谢

另外 把第一个程序的 for 括号后面的分号去掉 也会出现很杂乱的的结果。。。
理解错误

请注意 for 语句后面不能有分号，语法应该是：
for(条件初始化;条件;参数条件设定)
语句
所以程序：

```
#include <stdio.h>
void main()
{
   char ch[]="请看 for 语句中对逗号运算符的使用情况";
   for(int i=0;ch[i];ch[i]^=12,i++);        //请注意，由于这里有了分号，所以后面是一个空语句
   printf("加密后的字符串为：%s\n",ch);
}
```

因为 for 后面是一个分号，是一个空语句，所以不能用以下的方式拿出来：

```
   for(int i=0;ch[i];i++);
   ch[i]^=12;
```

这是为什么呢？因为 for 后面还是一个分号，所以下面的
ch[i]^=12;
这句只能被执行一次，并且当执行到这一句时，数组已经越界，正确的改正方法是去掉后面的分号：

```
   for(int i=0;ch[i];i++)
        ch[i]^=12;
```

也就是说，你下面的那个代码也是错误的了：
因为

```
   for(int i=0;ch[i];ch[i]^=12,i++);
   printf("加密后的字符串为：%s\n",ch);
```

上面其实是两个语句，而不是一个语句，所以后面的 printf 不能放进循环，如果你放进去了，那么这个函数就会被执行若干次，于是出现了你那结果很乱的情况。

学习一定不能一知半解，一定要把定义吃透，你出现的这种错误其实就是没有吃透循环语句的定义所造成的，当然还有一种情况，就是你做程序时，有些“想当然”，这是要不得的。

再说，建议大家做程序时还是不要将所有的东东都放进 for 后面那括号中，虽然运行效果一样，但对于读程序的人来说，理解上并不是一个好事。

当然，习惯了好多算法后，好多精简的代码用起来就会显得格外方便了，例如我们上面那句：

```
   for(int i=0;ch[i];ch[i]^=12,i++);
```

这个算法就十分经典。
恍然大悟 恍然大悟！
for（）后面加了一个； 这个分号代表的就是空语句
把 printf 加进去就会执行很多遍
太大意了 太大意了,谢谢牛刀老师指点

## 牛刀教学〇一四：新型的输出

如果你看到这一章，我们将要讲到一个新的课题－－函数，在讲函数之前，我们将向大家介绍一种新型的输出，看程序：

```
#include <iostream.h>
void ask(char *s)
{
    cout << "我是"<<s<<endl;
}
void main()
{
    cout << "大家好！"<<endl;  // 输出字符串 "大家好！"
    ask("牛刀");              //调用 ask 函数
}
```

大家将程序做进电脑看看，出现了什么？

可以大家有些奇怪，怎么会这样呢？

这就是我们说的一种新型的输出方法：流输出，或者叫做输出流。由此，我们将进入一个全新的课题－－对象编程。

cout 我们可以看作是一种运算，后面的“<<”是它的运算符，由于我们这里面的运算符是重载的（什么叫重载，我们以后会详细说到）。所以我们在用 cout 输出时，根本不需要考虑后面字符的数据类型，数据格式，举个例子吧：

```
#include <iostream.h>
void main()
{
    int i=29;
    char ch[]="请问你几岁了？";
    float pi=3.141592654;
    cout<<i<<ch<<pi<<endl;
}
```

大家看，完全不需要考虑类型问题，（记得我们以前输出前要使用：%d,%o,%s,%x...）吗？那里我们必须控制格式的输出，而在这里呢？我们根本不需要做任何事，可以这样说：cout 对于任何标准数据类型，都是通吃。

当然你可能看到了一个特殊的符号：endl：这表示换行，简单吧？当然上面的输出可以这样：

```
cout<<i<<endl<<ch<<pi<<endl;
```

你运行看看，会出现什么？当然，为了程序看起来舒服，我们也会加一些空格：

```
cout<<i <<endl  <<ch <<endl  <<pi <<endl;
```

甚至可以向下面一样：

```
cout<<i<<endl
    <<ch<<endl
    <<pi<<endl;
```

就是说你怎么用怎么舒服。

当然，同理，我们再次引进一个新的输入符：cin

它用的是输入流符号：“>>”，大家看符号的方向，输入，当然是往数据中输入，如：cin>>ch;而输出当然是从数据中输出：cout<<ch;
你瞧，输入竟然连取地址符号都可以不要，当然你要也可以，这也是归功于重载的技术，也就是说，用这两个语句，特点是就：舒服，怎么用怎么舒服。

但好象你也注意到了，要包的头文件变了，变成了 iostream.h，这就是我们 cin 与 cout 所需要包含的头文件。

这下好了，我们将以前的一些例子做进来看看吧：

```
#include <iostream.h>
void main()
{
    int i,sum;
    sum=0;
    for(i=1;i<=100;i++)
        sum+=i;
    cout<<"1+2+3+...+99+100="<<sum<<endl;
}
```

怎么样？大家可以回去将你以前的作业所有的 printf()与 scanf()都改成上面的形式，简单啊，方便啊。

我们第二个讲的是什么呢？讲的是函数（当然先给大家提一下，算是个入门吧）

```
#include <iostream.h>
void ask(char *s)
{
    cout << "我是"<<s<<endl;
}
void main()
{
    cout << "大家好！"<<endl;
    ask("牛刀");
}
```

上面的 ask 就是一个函数，当然喽，函数是在前面，是吧？大家还记得什么叫函数原型吗？对喽，可以将函数拿到后面：

```
#include <iostream.h>
void ask(char*);//函数原型中可以只用类型，不用参数，大家还记得 char 后面加一个*表示什么吗？表示后面定义的是一个地址，详细情况我们以后再帮大家分析。
void main()
{
    cout << "大家好！"<<endl;
    ask("牛刀");
}
void ask(char *s)
```

```
{
    cout << "我是"<<s<<endl;
}
```
函数原型难道只像我们以前说的，是一面小镜子的作用吗？
不，函数原形还可以将我们的程序放在同一个文件中，而实现放在另一个实现的文件中，注意了，跟牛刀继续走啊：
我们为程序再加上两个文件：
ask.h 与 ask.cpp 文件：
我们还在上面的工程中做这件事啊，如果你有了上面的程序了，可以继续走：
文件/新建/[文件]/"C/C++ Header File"
文件中输入:ask
按“确定”，程序为我们新建了一个“ask.h”文件
同样：
文件/新建/[文件]/"C++ Source File"
也输入：ask
按“确定”
注意了，必须两个文件名称相同，则一个自动加了.h，一个自动加了.cpp
我们分别叫他们为“头文件”与“程序文件”〔牛刀叫法〕
我们在.h 文件中可以输入函数原形（注：我们以后程序中会用注释给大家指明目前的文件是在头文件还是在源文件，在哪个文件，大家如果工程中没有这个文件，就自己按上面的方法加进来。）
//ask.h  我们以后就用这表示下面文件在 ask.h 文件中：以后相同：
注：如果当前不是指定文件，可使用 Ctrl+Tab 来更改当前的文件。
当然我们也可以打开“工作区”中的“FileView”选项卡，我们会看到如下内容：（注：按前面的加号可以展开）
```
Workspace'MyFirst':1 progect(s)
---MyFirst file
  ---Source File
  | ---ask.cpp
  | ---MyFirst.cpp
  ---Header File
  | ---ask.h
...
```
我们可以双击其中的文件名打来即可。
当然如果我们要删某个文件选择它，按“Del”键就行了。

我们下面看程序啊：
```
//ask.h
void ask(char*);
//ask.cpp
#include <iostream.h>
#include "ask.h"
void ask(char *s)
{
```

```
    cout << "我是"<<s<<endl;
}

//MyFirst.cpp
#include <iostream.h>
#include "ask.h"
void main()
{
    cout << "大家好！"<<endl;
    ask("牛刀");
}
```

大家再运行运行，思考，为什么在 ask.cpp 文件中也要包含 iostream.h 文件
我们看，当文件 MyFirst.cpp 中遇到了#include "ask.h"时，她就寻找 ask.h 文件，在编译时，将这段代码插进来，即定义了一个函数原型，于是，我们的主程序就可以识别 ask 函数了，于是在程序中可以调用这个函数，而后，遇到了函数后，就去找函数原型，在 ask.h 中，找到了，可函数的定义在哪儿呢？编译器就会自动寻找到与.h 对应的.cpp 文件，所以我们前面说过这两个文件名要相同，就是这个文件。

在 ask.cpp 中，由于要用到对应的函数原型，所以说，也必须使用“ ask.h”文件，当然，为什么使用了 iostream.h 文件，我们作为思考题，大家看可以将这个去掉吗？去掉运行看看会出现什么情况。

于是，又找到了相应的程序，我们的程序就被编译器给“找全了，于是，程序就可以运行起来。

我们为什么要介绍这些呢？因为我们如果做一个大的工程，可能就不是一个人搞的了是不是，于是，我想做一个大型的工程，可以将其中的某个函数交能某个人去实现等他实现了这项功能以后，我们只需要将他的文件加进我们的工程，我们的程序就会有这种功能了是吧？

那么怎么加入一个文件呢？
工程/添加工程/Files...
然后选择相应的文件就行了，可以同时选择多个文件。

这种方法还有一个好处就是：我们可以将工程的各个部分在编译时，只编译一次，形成了一个整体的文件，我们到相应的 Debug 目录下，我们将看到：

子目录下多了一个文件：ask.obj 这就是我们程序所形成二进制的文件。

而这种文件是不可以运行的，所以当我们的工程中如果函数特多，函数就会按自己的规则，各自放在各自的文件中，特别的好管理。

所以说，为了将这些二进制文件能够使用，我们ＶＣ中在做 exe 应用程序时，还有一个重要的过程就是“连编”。所以我们说，高级程序的编译是分好多过程的，所以高级程序的反编译不太现实。

再有一个好处，就是你以后学到了 dll 动态链接库时就会发现，这些程序代码可以隐藏起来，不让人看见，实现知识产权的有效保护，所以我们如果为名人或者为步步高等电子产品编程时，需要好多的参数设置，并且我们看不到他们的

原代码，就是这个原因。他们只需要将我们要使用的函数接口告诉我们就行了。

对了，大家还刻前面的:

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

吗？这也是微软做好的，由于他们被统一放在了同一个目录中，并且编译器知道了这个目录，所以我们用“<>”括号来括起来，编译器找相应文件时，只需要到那个目录下去找就行了。

而我们的 ask.h 文件由于不在那一个目录下，所以我们要用“” ””分号做的括号括，编译器在系统目录找不到时就进入当前目录寻找。

当然我们也可以将我们的 ask.h 文件与实现文件在在系统的 Include 目录下，然后在程序中使用：

```
#include <ask.h>
```

也是一样的，但如果我们自己做程序一般不那样做罢了。

当然，如果你的水平高了，你也可以在系统下做出我们自己的一套编程系统来，一样的，你只要对外提供一些函数就行了。牛刀做过，如果那样的话，那我们的编程只要适合自己，可以将程序做得特别得简单，有时简单得让人瞠目，但为什么这种编程会很简单，就是因为复杂的事我们可能在后面的文件中给你做了。

好了，不信的话，我们就再给大家介绍一个头文件，用头文件中的函数求一个浮点数的平方根，它的函数为：sqrt();头文件为：math.h（大家一看就懂吧？呵呵：

```
#include <iostream.h>
#include <math.h>
void main()
{
    float a;
    for(a=1;a<10;a++)
        cout<<a<<"的平方根为："<<sqrt(a)<<endl;
}
```

你瞧，程序做起来格外地简单吧？这就是我们所说的“函数”的作用。

Ｃ、Ｃ＋＋给我们定义了很多很多的函数，而如果我们用到了ＭＦＣ，里面竟然有三千多个函数，说真的，记都够我们记好一阵子，不过就因为这些函数的存在，使得我们的编程变得分外地简单。

于是，一个新的问题就出现了，我们可以做这么多的函数，那怎么管理呢？一个 ask 文件，好弄啊，如果我有十个八个文件怎么办啊？这些做 VC 的程序的人早就为我们想过了。

大家再在工作区上单击“ClassView”选项卡（注：为什么叫 Class，我们以后就会清楚了，现在大家只需要知道我们可以在这儿看东西就行。

```
---MyFirst classes
  ---Globals
    ---ask(char *s)
    ---main()
```

...
看，怎么样？全都在这里呢，我们只要双击某个函数，就可以对她进行操作。
我们可以双击这儿迅速地定位到我们想要的函数。
当然，我们也可以在工具栏上右击，选择“WizardBar”，在这个工具栏上，也可以对所有的函数进行操作。
行了，这节课就算是结束了。

作业：
做一个求最大值与最小值的程序，要求分别建立在不同的文件中，然后在主函数中调用最大值与最小值函数分别求两个数的最大值与最小值。
将牛刀的程序加入到工程中实现“叙古拉猜想”

呜，这个名字好恐怖哟，“刀死”，牛刀要归位了，呵呵。

幸亏如今用的都是Ｗｉｎｄｏｗｓ，而不是Ｄｏｓ，否则牛刀这个名子不敢要了，所以说，牛刀不可能带你“深入”Ｄｏｓ时代，只是“进入”，呵呵。

按“开始/运行”，输入：cmd，按回车，即可进入我们的Ｗｉｎｄｏｗｓ下的Ｄｏｓ窗口模式：
当然你可以从“程序/附牛/命令提示符”中运行它。

运行后，我们输入
D:
注：输入D:再加回车，我们每一行都要回车的啊，我以后就不说了，大家按我们的一行一行去做就行了。
没有特殊的情况，我们将看到如下形式：
D:\>
上面的“>”是一个间隔符，它后面是我们输入命令的地方，而前面表示当前目录。什么叫当前目录呢？当前目录就是我们目前处在的目录，我们可以用命令提示符“CD”来改变当前目录。例如，我的程序中的所有自己做的程序都是放在ＶＣ目录中的，我就可以用来面的命令来改变到ＶＣ目录中去：
D:\>cd vc

D:\VC>
大家看，当前目录变成了：“D:\VC”，其中的“D:”表示当前的盘符，而“VC”在我们这里属于当前的节点，在Ｗｉｎｄｏｗｓ系统中叫做文件夹，有一个文件夹的图标，很形象。而我们如果没有ＶＣ这个文件夹怎么办呢？建啊，建文件夹的命令为：“MD”。

如：我们在当前目录中建一个“CCC”的文件夹（当然了，文件夹可以自己任意起名），看：
D:\VC>md ccc

D:\VC>

好像没有反应，那我们怎么看我们有没有建好这个文件夹呢？可以用“dir”命令来看我们建好的文件夹和文件：

```
D:\VC>dir
驱动器 D 中的卷没有标签。
卷的序列号是 74C6-A1D1

D:\VC 的目录

2006-05-28  08:58    <DIR>          .
2006-05-28  08:58    <DIR>          ..
2006-06-01  15:57    <DIR>          MyMirGui
2006-06-01  16:08    <DIR>          MirGui
2006-06-17  22:34    <DIR>          BaseWindow
2006-06-23  01:54    <DIR>          DrawTest
2006-06-24  00:07    <DIR>          fractal
2006-06-24  01:24    <DIR>          LSystem
2006-06-25  00:28    <DIR>          MyFont
2006-06-25  15:39    <DIR>          MyKey
2006-06-25  17:27    <DIR>          MyMouse
2006-06-29  00:30    <DIR>          MyMenu
2006-06-30  20:36    <DIR>          SuperMf
2006-07-01  01:42    <DIR>          Menu
2006-07-01  02:50    <DIR>          MyBitmap
2006-07-02  02:20    <DIR>          MyDlgBox
2006-07-03  13:04    <DIR>          DoubleDrog
2006-07-17  18:37    <DIR>          MidiTest
2006-07-23  13:59    <DIR>          Life
2006-07-24  14:17    <DIR>          MyFirst
2006-07-29  17:04    <DIR>          Study
2006-07-31  15:27    <DIR>          ccc
               0 个文件                0 字节
              22 个目录     516,210,688 可用字节

D:\VC>
```

大家看，上面的最后一个，后面的名称是不是就是“ccc”？，这就是我们刚建的，其它的嘛，我们看后面只要有<DIR>的，就是文件夹。

那么，下面我们用“rd”将目录删了，再看一下：

```
D:\VC>rd ccc

D:\VC>dir
驱动器 D 中的卷没有标签。
卷的序列号是 74C6-A1D1
```

```
D:\VC 的目录

2006-05-28  08:58    <DIR>          .
2006-05-28  08:58    <DIR>          ..
2006-06-01  15:57    <DIR>          MyMirGui
2006-06-01  16:08    <DIR>          MirGui
2006-06-17  22:34    <DIR>          BaseWindow
2006-06-23  01:54    <DIR>          DrawTest
2006-06-24  00:07    <DIR>          fractal
2006-06-24  01:24    <DIR>          LSystem
2006-06-25  00:28    <DIR>          MyFont
2006-06-25  15:39    <DIR>          MyKey
2006-06-25  17:27    <DIR>          MyMouse
2006-06-29  00:30    <DIR>          MyMenu
2006-06-30  20:36    <DIR>          SuperMf
2006-07-01  01:42    <DIR>          Menu
2006-07-01  02:50    <DIR>          MyBitmap
2006-07-02  02:20    <DIR>          MyDlgBox
2006-07-03  13:04    <DIR>          DoubleDrog
2006-07-17  18:37    <DIR>          MidiTest
2006-07-23  13:59    <DIR>          Life
2006-07-24  14:17    <DIR>          MyFirst
2006-07-29  17:04    <DIR>          Study
               0 个文件              0 字节
              21 个目录     516,177,920 可用字节

D:\VC>
```

看，怎么样？没有了，下面我们再看，我们第一次课教大家建的目录：MyFirst，在倒数第二个，大家找到了吗？我们进去：

```
D:\VC>cd myfirst

D:\VC\MyFirst>
```

再看看前面的当前目录，怎么样？进来了。

当然我们可以再看看当前的一些文件与目录：

这次我们用“dir /w”

```
D:\VC\MyFirst>dir/w
驱动器 D 中的卷没有标签。
卷的序列号是 74C6-A1D1

D:\VC\MyFirst 的目录

[.]             [..]            ask.asp         MyFirst.opt     ask.h
ask.cpp         Xgl.cpp         Xgl.h           MyFirst.cpp     MyFirst.ncb
[Debug]         MyFirst.plg     MyFirst.dsp     MyFirst.dsw
```

```
              11 个文件            100,335 字节
               3 个目录      516,096,000 可用字节

D:\VC\MyFirst>
```

这个命令是什么意思呢？原来啊，Dos 中的命令有几个特点：

一、大小写不敏感，这一点大家也知道了，在Ｃ中“dir”与“DIR”绝对不是一回事，而在 Dos 中，只要字母一样就行，随便你是大写小写还是大小写都有，如果大家学过ＶＢ，也会知道，ＶＢ对大小写也是不敏感的，便为了好看，ＶＢ中每一行输入后，都会给大小写作相应的转换，因为那样好看啊，大家还记得我们怎么用程序将大小写转换吗？

二、命令基本上都可以带参数，如：上面的“/w”，什么意思呢？我们可以用“/?”来查一查：

```
D:\VC\MyFirst>dir /?
```

显示目录中的文件和子目录列表。

```
DIR [drive:][path][filename] [/A[[:]attributes]]  [/C] [/D] [/L]
  [/O[[:]sortorder]] [/P] [/Q] [/S] [/T[[:]timefield]] [/W] [/X] [/4

  [drive:][path][filename]
              指定要列出的驱动器、目录和/或文件。

  /A          显示具有指定属性的文件。
  attributes   D  目录                R  只读文件
               H  隐藏文件            A  准备存档的文件
               S  系统文件            -  表示“否”的前缀
  /B          使用空格式(没有标题信息或摘要)。
  /C          在文件大小中显示千位数分隔符。这是默认值。用 /-C 来
              停用分隔符显示。
  /D          跟宽式相同，但文件是按栏分类列出的。
  /L          用小写。
  /N          新的长列表格式，其中文件名在最右边。
  /O          用分类顺序列出文件。
  sortorder    N  按名称(字母顺序)      S  按大小(从小到大)
               E  按扩展名(字母顺序)    D  按日期/时间(从先到后)
               G  组目录优先            -  颠倒顺序的前缀
  /P          在每个信息屏幕后暂停。
  /Q          显示文件所有者。
  /S          显示指定目录和所有子目录中的文件。
  /T          控制显示或用来分类的时间字符域。
  timefield    C  创建时间
               A  上次访问时间
               W  上次写入的时间
  /W          用宽列表格式。
  /X          显示为非 8dot3 文件名产生的短名称。格式是 /N 的格式，
```

```
                短名称插在长名称前面。如果没有短名称，在其位置则
                显示空白。
  /4            用四位数字显示年
```

可以在 DIRCMD 环境变量中预先设定开关。通过添加前缀 -（破折号）来替代预先设定的开关。例如，/-W。

```
D:\VC\MyFirst>
```

现在的 Dos 就是好啊，全是中文了。我们看其中的/w 解释为：用宽列表格式。就是一行几个吧，那么这里是怎么区分目录和文件的呢？原来啊，目录外面都有方括号“[]”。

其他的自己看，以后遇到命令不会用啊，都可以用 /? 来求得帮助，这就是 Dos 时代的帮助系统啊，呵呵。

当然了，我们第一次使用的 D:那是改变驱动器目录啊，大家知道喽。

我们再看看进入我们的 Debug 目录：

```
D:\VC\MyFirst>cd debug

D:\VC\MyFirst\Debug>dir
驱动器 D 中的卷没有标签。
卷的序列号是 74C6-A1D1

D:\VC\MyFirst\Debug 的目录

2006-07-24  14:17    <DIR>          .
2006-07-24  14:17    <DIR>          ..
2006-07-31  14:57            50,176 vc60.idb
2006-07-31  14:57            61,440 vc60.pdb
2006-07-24  14:17           186,924 MyFirst.pch
2006-07-31  14:57             8,587 MyFirst.obj
2006-07-31  14:57           240,816 MyFirst.ilk
2006-07-30  18:50             6,643 Xgl.obj
2006-07-31  14:57           188,464 MyFirst.exe
2006-07-31  14:57           508,928 MyFirst.pdb
2006-07-30  17:50             4,720 ask.obj
               9 个文件      1,256,698 字节
               2 个目录    515,899,392 可用字节

D:\VC\MyFirst\Debug>
```

大家看前面有两个文件夹：

一个是“.”，另一个是“..”如果你在Ｗｉｎｄｏｗｓ中，找不到这两个文件夹，那是什么呢？在Ｄｏｓ中，“.”就表示当前目录，也就是说，你输入下面的命令没有用（还在当前目录下）：

```
D:\VC\MyFirst\Debug>cd .
```

D:\VC\MyFirst\Debug>
而输入：cd..　　则进入上一节目录。大家看着，然后思考啊：（当然你也可以输入：cd /?　来查看帮助）
D:\VC\MyFirst\Debug>cd ..

D:\VC\MyFirst>cd /?
显示当前目录名或改变当前目录。

CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..]

  ..　　指定要改成父目录。

键入 CD 驱动器: 显示指定驱动器中的当前目录。
不带参数只键入 CD，则显示当前驱动器和目录。

使用 /D 命令行开关，除了改变驱动器的当前目录之外，
还可改变当前驱动器。

如果扩展命令名被启用，CHDIR 会如下改变:

当前的目录字符串会被转换成使用磁盘名上的大小写。所以，
如果磁盘上的大小写如此，CD　C : \TEMP 会将当前目录设为
C:\Temp。

CHDIR 命令不把空格当作分隔符，因此有可能将目录名改为一个
带有空格但不带有引号的子目录名。例如:

    cd \winnt\profiles\username\programs\start menu

与下列相同:

    cd "\winnt\profiles\username\programs\start menu"

在扩展功能停用的情况下，您必须键入以上命令。

D:\VC\MyFirst>

当然，我们要进入根目录，我们可以使用“cd \”，后面什么参数都不带（注根目录就是驱动器目录啊，大家看……）
D:\VC\MyFirst>cd \
D:\>

当然如果我们对于我们的路径非常熟悉，我们可以直接输入路径，进入我们所要进入的目录：

```
D:\>cd vc\myfirst\debug

D:\VC\MyFirst\Debug>
```

如果我们觉得太乱了，我们可以用命令“cls”来清屏：

```
D:\VC\MyFirst\Debug>cls
```

我们这下知道我们正常用 Dos 的一些方法了，我们怎么运行我们的程序呢？牛刀的程序名为 MyFirst.exe，我们可以直接输入前面的名字就行了，看：

```
D:\VC\MyFirst\Debug>myfirst
```

牛刀算命程式正在为你算命，请你脑中任想一数字（0 到 255 之间）：
不要告诉我噢!
但我通过几次测试，我就能知道你在想什么，你信不？
不信咱们来啊，看：我准备出数字了：
为了大家估计程序的结构，我们没有做手脚，即没有打乱，也没有变成其他的符号：

```
128  129  130  131  132  133  134  135  136  137  138  139  140  141
 142  143
144  145  146  147  148  149  150  151  152  153  154  155  156  157
 158  159
160  161  162  163  164  165  166  167  168  169  170  171  172  173
 174  175
176  177  178  179  180  181  182  183  184  185  186  187  188  189
 190  191
192  193  194  195  196  197  198  199  200  201  202  203  204  205
 206  207
208  209  210  211  212  213  214  215  216  217  218  219  220  221
 222  223
224  225  226  227  228  229  230  231  232  233  234  235  236  237
 238  239
240  241  242  243  244  245  246  247  248  249  250  251  252  253
 254  255
```

请问你想的数在里面吗？（你要如实回答），在按“y”或“Y”否则按“n”或者“N”:\
这是牛刀做的一个算命的程序，对了，说一下，如果大家运行这个程序的话，可能有时程序的数字会分两行显，大家一定要看清啊，因为牛刀的程序没有做优化，只是为大家提供一个思索的机会罢了。

在这里运行程序有什么好处呢？

一、可以正常运行Ｄｏｓ程序，运行后不自动退出，可以查看结果。

二、这种方法与Ｄｏｓ模式适应，比较适合运行Ｄｏｓ程序。

三、这种方法运行Ｄｏｓ程序，可以代入参数，我们下一节课会为大家做一些例子，大家运行时注意了，我会在例子中说“请大家在Ｄｏｓ模式下运行这个程序”，方法就是我们上面讲解的方法，我会教大家怎么输入，好了，我们将程序改改，再运行看看，我们用一个空函数啊：

```
void main()
{
}
```

上面这个程序，我们只有在ＶＣ环境中运行。如果在Ｗｉｎｄｏｗｓ环境下双击，会一闪而过，大家知道为什么在Ｗｉｎｄｏｗｓ中一定要加：“Press any key to continue”了吧？否则你看不到结果，事实上也没有结果，那我们在Ｄｏｓ模式下运行这个程序：

```
D:\VC\MyFirst\Debug>myfirst

D:\VC\MyFirst\Debug>
```

看，中央只有一空行，也没有结果，那我们加点内容：

```
#include <iostream.h>
void main()
{
    cout<<"我根牛刀学 Dos"<<endl;
}
```

编译一下，然后再在Ｄｏｓ方式下运行程序：

```
D:\VC\MyFirst\Debug>myfirst
我根牛刀学 Dos

D:\VC\MyFirst\Debug>
```

怎么样？这就是我们做的第一个Ｄｏｓ程序，并且在Ｄｏｓ下运行的程序，而我们前面的所有程序，都可以在Ｄｏｓ下运行这些程序，于是，我们给这种程序起了一个响亮的名字－－软件。

不要以为这样说比较可笑，想当年，软件都是以这种方式发行的。

好了，牛刀的“Ｄｏｓ”之旅，暂时结果，希望大家勤上机勤动手，我们真正的软件技术还在后面呢。

请问牛刀老师
强制转化类型来输出 cout 能做到吗？
就是说要以 int 的形式输出 char 型变量里的数据
是不是这个时候还是得用 printf 里的%d？

另外 如果想要输出的是字符串的话比如"牛刀编程" 就要两边加双引号 结果是:
牛刀编程
这个结果并没有加双引号
但是我想让这个结果也有双引号怎么办呢？
我一开始想用这种 ""牛刀编程"" 可是出错了
不知道有没有别的方法可以实现
谢谢老师!

我记得 cout 的类型是自动转换的。
要想输出双引号可以用 \"，比如 “\"牛刀编程\"”

## 牛刀教学〇一五：我爱我的家

呵呵，这个题目可能很特别，我们必须放到这里来讲，因为我们已经开始和大家讲函数了，好了，我们再来回顾一下我们以前的函数的定义：

```
函数类型  函数名(参数列表)
{
    语句
}
```

那我们第一个使用的函数是什么函数呢？当然是 main 函数，我们也称之为“入口函数”。

可能 main 函数也被重载过（什么是重载，就是定义多个同名的函数。），所以我们的函数不仅仅是下面一种形式：

```
void main()
{
}
```

当然,我们说一个函数如果函数的类型不是 void 类型的话,我们可以使用 return 来返回函数的值。如：

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

如果不用 return 语句，编译器会报错（至少我的编译器报错了。）

甚至我们可以使用下面的形式：

```
char main()
{
    return 'a';
}
bool main()
{
    return true;
}
```

我上网去查了查，有些文章说：void main  是错误的，不要说我们后两种情况了，然而我们说，我们要重视是实践，虽然说错误，不符合规定，那么在ＶＣ中却通过了,所以我们说不要认为规定是一成不变,那仅仅是人们的一种规定而已。我们何必拘泥于规定呢，至少这种规定在我们的ＶＣ６．０中不起着用，因为我们的程序编译通过了，是不是？

那么,我们也本着与规定相符的精神,我们以后给大家做程序就使用 int main(); 形式，当然我们学做程序不能看什么东西是什么形式，不能拘泥于书本，书上说 char ch[10];只能保存１０个字符，可我们却可以将 ch[15]给打印出来，这怎么说呢？谁敢说我这种用法就是错误的？考试当然不能用,编程还不能用吗？编程是一门实践的学问，而不是一门理论上的过程。为什么我们一看书说这么说，我们就说：void main();是错误的呢？错在哪里了？当然如果你参加考试，在试

卷上，我们不能这样想，因为那关系到了我们的分数。

行了，不说了，继续给大家讲讲主函数吧－－那是我们的老家啊，没有它，我们的一切都玩转不了。所以我们要做这么激烈的讨论，就是要告诉大家，我们是在电脑上做程序的，而不是在试卷上做程序的，所以我们的教程更有权威性。

我们再来看看我们可爱的家吧：

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

我们的程序返回的零值去干什么呢？当然是让Ｄｏｓ下的一些调用 main 的程序使用的是不？那我们就需要进入Ｄｏｓ模式，给大家运行运行这个程序，看看我们的程序在做什么，当然，为了使我们的程序有些生气，我们还要加点料，以表示我们的程序运行了。

```
#include <iostream.h>
int main()
{
    cout<<"我跟牛刀试试我们的主函数……"<<endl;
    return 0;
}
```

在Ｄｏｓ下运行此程序：

```
D:\VC\MyFirst\Debug>myfirst
我跟牛刀试试我们的主函数……

D:\VC\MyFirst\Debug>
```

过后我们输入命令：echo %ERRORLEVEL%

```
D:\VC\MyFirst\Debug>echo %ERRORLEVEL%
0

D:\VC\MyFirst\Debug>
```

看，怎么样？为什么是这个命令呢？

echo 是我们 Dos 下的显示命令，

而后面的%ERRORLEVEL%表示我们显示的是错误符，也就是说，如果我们的程序返回的是零，就表示我们程序是对的，那么我们的想法是不是正确的呢？我们再改改程序：

```
#include <iostream.h>
int main()
{
    cout<<"我跟牛刀试试我们的主函数……"<<endl;
    return 555;//看，牛刀将其改成了 555，看会出现什么情况？
}
```

继续在Ｄｏｓ下运行这个程序：

```
D:\VC\MyFirst\Debug>myfirst
```

我跟牛刀试试我们的主函数……

```
D:\VC\MyFirst\Debug>echo %ERRORLEVEL%
555

D:\VC\MyFirst\Debug>
```

怎么样？我们返回的值：555 出现在这儿了，那么有什么作用呢？这个值就返回给了我们系统调用（如Ｄｏｓ系统的调用），或者其他程序的调用（注：厉害吧？程序之间竟然还可以相互调用的。）

但我们一般没有特殊的情况，我们在程序中的返回值设置为零。

当然如果你做了一个很大规模的软件，其中有很多错误处理，并且在某些程序或Ｄｏｓ程序中要处理这些错误数据，我们就必须注意返回值的定义与规定了，那是一个软件系统的问题了。

行了，关于函数的返回值部分我们先讲到这儿，我们再讲讲我们家中的几个成员：

```
int main(int argc,char *argv[],char *env[])
{
    return 0;
}
```

你有些晕吧？如果是那样，我告诉你，下面几种函数形式都是正确的

```
int main(int argc)
{
    return 0;
}
int main(int argc,char *argv[])
{
    return 0;
}
int main(int argc,char *argv[],char *env[])
{
    return 0;
}
```

三种定义全是正确的，这还是符合我们的函数定义：

```
函数类型　函数名(参数列表)
{
    语句
{
```

是也不是？

行了，上面第一种情况，只有一个参数的，我们不经常使用，一般第三种也正常不使用，为了使大家明白第三个参数的使用，我们给大家做个程序，大家上机运行看看，第三个参数也就这样使用就行了：

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

```
int main(int argc,char *argv[],char *env[])
{
    for(int i=0;env[i]!=NULL;i++)cout<<env[i]<<endl;
    return 0;
}
```

这个函数在Ｄｏｓ下运行的结果很长(因为我们现在用的是Ｗｉｎｄｏｗｓ了，所以很长）它是打印出了我们的系统配置。同时也是为了保密我系统的一些密秘，所以中央用省略号省去了（有些人总对别人系统感兴趣，牛刀无语了，呵呵）

```
D:\VC\MyFirst\Debug>myfirst
ALLUSERSPROFILE=C:\Documents and Settings\All Users
……
windir=C:\WINDOWS

D:\VC\MyFirst\Debug>
```

行了，第三个参数就是这样，显示电脑系统的信息，如果你的程序有必要，可以查看别人的系统配置，再根据别人的配置来按不同的情况执行我们程序（这叫“看机下菜喋”，呵呵）。当然如果你真对别人电脑感兴趣，这个文件可以做成一个文件……

行了，我们看有两个参数的吧：

```
#include <iostream.h>
int main(int argc,char *argv[])
{
    cout<<argv[0]<<endl;
    return 0;
}
```

我们的 argc 是一个整型，它告诉我们后面*argv[]中有多少个字符串当然了，这个数字最小为１，表示我们最少有一个字符串，就是：

*argv[0],

我们上面的程序就是输出这个字符串。

而我们其他的字符串是哪儿来的呢？我们做个程序来看看吧,(注：由于 argv[0] 我们知道了，所以我们输出就从 argv[1]开始输出）：

```
#include <iostream.h>
int main(int argc,char *argv[])
{
    for(int i=1;i<argc;i++)
        cout<<argv[i]<<endl;
    return 0;
}
```

这个程序，我们在ＶＣ的环境下，什么都不输出，呵呵，怎么回事呢？没有字符串啊。那么，这个字符串是哪里来的呢？这，就是我们程序的参数（大家还知道我们前一节阅读材料中告诉大家　Ｄｏｓ程序都可以带参数的吗？我们进入Ｄ

ｏｓ，带进参数，对这个程序进行运行：

```
D:\VC\MyFirst\Debug>myfirst aaa bbb ccc ddd eee fff ggg
aaa
bbb
ccc
ddd
eee
fff
ggg

D:\VC\MyFirst\Debug>
```

怎么样？我们的参数（其中的 aaa,bbb,...）在程序中被处理了，我们最后再将 argc 输出来看看：

```
#include <iostream.h>
int main(int argc,char *argv[])
{
    for(int i=1;i<argc;i++)
        cout<<argv[i]<<endl;
    cout<<"你输入了"<<argc-1<<"个字符！";
    return 0;
}
```

注：大家看结果，一定要配合我们的程序看啊，不然会不知其所以然的，下面我们看看 argc 在这儿到底有什么用：（你瞧瞧我们的 cout 多好，表达式在其中都能使用:argc-1）

```
D:\VC\MyFirst\Debug>myfirst good fine list find ok
good
fine
list
find
ok
你输入了 5 个字符！
D:\VC\MyFirst\Debug>
```

当然我们的程序可以做得很完美，可以对字符进行字符统计，字符排序等等各种你能够想得出来的操作。

当然我们的程序也可以根据我们后面的参数做出不同的数据处理，很容易噢，作为一种好玩的东西，牛刀不能独享，但也不能影响大家的思路，牛刀就不能大家做例子了，大家自己看能够做出什么样的好玩的软件出来。

大家也知道，我们做的软件是按空格来分割后面的参数的，能不能是其他的分割符呢？大家试试好了，好玩的都留给大家自己去玩。

牛刀声明一下，以后牛刀不带领大家做这种可以将程序做得很强大的程序了，牛刀还是用下面的形式：

```
int main()
{
    return 0;
```

}
来做牛刀的程序。
注意后面的 return 0;不要忘记啊，因为我们的返回类型变了。我们说过，大家以计算机为准，不要以任何教材为准，包括牛刀的教材。

这一讲就不布置作业了，main 函数的确是一个有争议的函数，但我们还是要说，你怎么做程序舒服，你怎么做，不要认为某一个教材上说这样不能，那样不能，果然不能吗？当然还是那一句话，想知道梨子的味道，自己亲自咬一口。求 2 的１００次方，怎么来？这个想法我在玩小霸王时就做过，没想竟然被我在小霸王上给弄出来了，但说实话，小霸王当时是八位机，并且ＣＰＵ的速度特慢，运行了好一阵子，我等啊等啊……　当然结果是等出来了，于是就经常想，这种算法如果出现在电脑上……　今天，我们有机会实现了，于是，我在帮大家做教程时，给大家做一个求 2 的１０００次方的标准程序。　可能你会说，用计算机中的计算器啊，不到一秒钟，结果就出来了，不错啊，那一个是别人做的程序，二一个，你用她算一个 2 的１００００次方试试？三百多位哪。　行了，我们下面进行我们有关大数的编程之旅吧。　当然，我们还进行我们的正常思维，做出如下的程序：#include <iostream.h>int main(){　int i,sum=1;//注意，在加法中，累加器我们初始化为 0，而在乘法中，我们累乘器初始化为 1 大家想想为什么　for(i=0;i<100;i++)　sum*=2;　cout<<sum<<endl;　return 0;}　大家做出来的结果是什么呢？牛刀的结果为 0，算到二十几次方时，还能出结果，到４０次方就完蛋了，呵呵。　什么原因呢？数据溢出了呗，我们做得溢得也太多了，甚至超过了整数负的表示范围了，所以，计算机可能真懒得理，所以干脆给你清 0 了，具体是不是这情况，大家可在循环中，每一步都打出来看看：#include <iostream.h>int main(int argc,char *argv[]){　int i,sum=1;//注意，在加法中，累加器我们初始化为 0，而在乘法中，我们累乘器初始化为 1 大家想想为什么　for(i=0;i<100;i++)　{　sum*=2;　cout<<i+1<<"\t"<<sum<<endl;//这里也可用转义字符啊？不错，当然 endl 也可用"\n"来代替　}　return 0;}　结果是：２的３０次方还能出正确的值，２的３１次方已经是负的了，３２次方，电脑就拒绝运算，给清零了。其实中整数中，是溢出了。清 0 可能是编译器处理的一种办法。　好了，既然电脑不能胜任，那就看牛刀教大家的－－数组法。大家记得小学老师教我们用笔在稿纸上算加减乘除吗？牛刀就用电脑来模拟这个笔算的过程。当然了，我们没有必要让电脑去记住乘法口决，加法口决，我们就用电脑的一些基本的计算功能来达到我们的目的吧。我们假设一个数组，有１０００个元素，当然，如果大家要求１００００次方，数组就需要３０００多个元素，不要舍不得噢，有了电脑，内存是不花钱的。int a[100];大家这里用 int，当然你可以用 char 类型，但注意噢，cout<<可是不需要我们用类型定义的噢，所以如果用 char 类型的话，输出的就是定是字符，当然了，你可以用函数转换，cout 中定义了相当丰富的函数，可以转换这些，一样是一样的，但我们目前还没有必要。我们假设 a[0]存放数据的位数，这是一个技巧，我们为什么要同时处理 100 位呢？没有必要嘛，我们只要知道我们的位数，有多少位，咱们处理多少位，不就行了吗？　所以我们一开始要对数组进行赋初值：a[0]=1;//表示 1 位 a[1]=1;//表示赋初值 1，我们以后以数字小为低位，数字大为高位，所以我们以后输出时要注意了，要从高位向低位输起啊 for(i=2;i<100;i++)　a[i]=0;由于我们处理时，a[0]是表示位数的，所以我们真正处理数据时，只需要从 1 循环到 a[0]就行了。for(i=1;i<=a[0]-1;i++){...}这就是技巧:可是我们为什么没有循环到 a[0]，而是循环

到 a[0]-1 呢？原来啊，牛刀想将最高位拿到循环外面来做在外面，我们可以这样做：a[a[0]]*=2;if(a[a[0]]>=10){ a[a[0]]%=10; a[0]++; a[a[0]]=1;}这一段程序能看懂吗？原来啊，由于 a[0]是位数，所以 a[a[0]]就是当前的最高位，我们为了保证每位是一位数字，所以牛刀在这里先将她乘以 2，然后判断它是否大于 1 0，如果不大，没有进位，不必处理，但如果有了进位了，就进入了处理了。我们先将最高位十位上的数字去掉，就是对 1 0 进行取模：a[a[0]]%=10;而后我们将 a[0]自加，表示最高位已经多了一位，此时最高位我们并没处理过，而又因为低位有了进位（注：进位肯定不会超过 1 哟，我们自己去算算看。 于是，这种算法就成功了。 大家看牛刀上面的想法，有一个致命的错误，大家不知能否看得出来，试试看，看牛刀考虑问题时少考虑了什么？我们在以后的程序中会给大家以暗示，看程序：#include <iostream.h>int main(){ int a[100],i,j,jw;//牛刀的 jw 表示本位的进位啊，牛刀在上面与大家说思路时没有考虑，做程序时，我们就不能不考虑了，否则程序就会出错 jw=0;//初始进位为 0，我们的目的是从一开始就进行进位的处理。当然最高位的进位处理，也可根据进位的数据情况拿到外面，但为了上面的思路，牛刀没这么做。程序大家可以自己改改啊，程序是人做的，所以我们每个人都有资格在程序上加上自己的思
想 a[0]=1; a[1]=1; for(i=2;i<100;i++) a[i]=0; for(i=0;i<100;i++) { for(j=1;j<a[0];j++)//看牛刀这儿为什么可以这样变？ { //牛刀要对不是最高位的数据处理
了 a[j]=a[j]*2+jw; if(a[j]>=10) { a[j]%=10; jw=1; } else { jw=0; } }

a[a[0]]=a[a[0]]*2+jw; if(a[a[0]]>=10) { a[a[0]]%=10; a[0]++; a[a[0]]=1; } jw=0; } for(i=a[0];i>0;i--)//为什么我们这儿从高向低？ cout<<a[i]; cout<<endl;//这句话已经不在循环中了。 return 0;}怎么样？牛刀电脑一闪就出来了，呵呵，牛刀电脑出来的结果是：
126765060022822940149670320537 6 大家再按相同的方法求 2 的 1 0 0 0 次方看看当然，我们可以改程序中的数据了，这次，牛刀要教大家一个新的东西－－宏什么是宏呢？我们先来看下面的程序#include <iostream.h>#define MAX 100int main(){ int a[MAX],i,j,jw;//牛刀的 jw 表示本位的进位啊，牛刀在上面与大家说思路时没有考虑，做程序时，我们就不能不考虑了，否则程序就会出错 jw=0;//初始进位为 0，我们的目的是从一开始就进行进位的处理。当然最高位的进位处理，也可根据进位的数据情况拿到外面，但为了上面的思路，牛刀没这么做。程序大家可以自己改改啊，程序是人做的，所以我们每个人都有资格在程序上加上自己的思
想 a[0]=1; a[1]=1; for(i=2;i<MAX;i++) a[i]=0; for(i=0;i<MAX;i++) { for(j=1;j<a[0];j++)//看牛刀这儿为什么可以这样变？ { //牛刀要对不是最高位的数据处理
了 a[j]=a[j]*2+jw; if(a[j]>=10) { a[j]%=10; jw=1; } else { jw=0; } }

a[a[0]]=a[a[0]]*2+jw; if(a[a[0]]>=10) { a[a[0]]%=10; a[0]++; a[a[0]]=1; } jw=0; } for(i=a[0];i>0;i--)//为什么我们这儿从高向低？ cout<<a[i]; cout<<endl;//这句话已经不在循环中了。 return

0;}大家看，上面的：#define MAX 100 就是一个宏，她有什么用呢？你看它一个是未尾没有分号第二是它不是程序，根本不运行，而是在整个程序中仅仅用宏后面的定义来代替前面的符号，举个例子吧：#define MAX 100int a[MAX];for(i=2;i<MAX;i++)for(i=0;i<MAX;i++)我们将宏与用到宏的部分都拿出来了，宏的定义为：#define 宏标识符　宏替代符什么意思呢？当我们编译系统编译时，先遇到了宏，与是它先读一遍程序，并且将程序中的宏标识符都给换后，然后再编译，于是，上面几句就变成了下面几句：int a[100];for(i=2;i<100;i++)for(i=0;i<100;i++)可能你认为，不是没变么？还和我们以前的程序一样啊！是啊，的确没变，看上去这种宏定义的意义一点都没有了，可如果我们将宏中的数据改一下看看#define MAX 1000int a[MAX];for(i=2;i<MAX;i++)for(i=0;i<MAX;i++)这样一来，程序就变成了：int a[1000];for(i=2;i<1000,i++)for(i=0;i<1000,i++)怎么样？一下子程序都变了，可以算 2 的 1 0 0 次方的程序一下子就变成了可以算 2 的 1 0 0 0 次方的程序了，这可是一个质的飞跃啊。我的计算结果为：
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376 怎么样？我相信我的计算机。你不信，再去算算 2 的 1 0 0 0 0 次方瞧瞧我的电脑运算的效率还是挺高，
19950631168807583848837421626835850838234968318861924548520089498529438830221946631919961684036194597899331129423209124271556491349413781117593785932096323957855730046793794526765246551266059895520550086918193311542508608460618104685509074866089624888090489894838009253941633257850621568309473902556912388065225096643874441046759871626985453222868538161694315775629640762836880760732228535091641476183956381458969463899410840960536267821064621427333394036525565649530603142680234969400335934316651459297773279665775606172582031407994198179607378245683762280037302885487251900834464581454650557929601414833921615734588139257095379769119277800826957735674444123062018757836325502728323789270710373802866393031428133241401624195671690574061419654342324638801248856147305207431992259611796250130992860241708340807605932320161268492288496255841312844061536738951487114256315111089745514203313820202931640957596464756010405845841566072044962867016515061920631004186422275908670900574606417856951911456055068251250406007519842261898059237118054444788072906395242548339221982707404473162376760846613033778706039803413197133493654622700563169937455508241780972810983291314403571877524768509857276937926433221599399876886660808368837838027643282775172273657572744784112294389733810861607423253291974813120197604178281965697475898164531258434135959862784130128185406283476649088690521047580882615823961985770122407044330583075869039319604603404973156583208672105913300903752823415539745394397715257455290510212310947321610753474825740775273986348298498340756937955646638621874569499279016572103701364433135817214311791398222983845847334440270964182851005072927748364 5

505786345011008529878123894739286995408343461588070439591189858151457
791771436196987281314594837832020814749821718580113890712282509058268
174362205774759214176537156877256149045829049924610286300815355833081
301019876758562343435389554091756234008448875261626435686488335194637
203772932400944562469232543504006780272738377553764067268986362410374
914109667185570507590981002467898801782719259533812824219540283027594
084489550146766683896979968862416363133763939033734558014076367418777
110553842257394991101864682196965816514851304942223699477147630691554
682176828762003627772577237813653316111968112807926694818872012986436
607685516398605346022978715575179473852463694469230878942659482170080
511203223654962881690357391213683383935917564187338505109702716139154
395909915981546544173363116569360311222499379699992267817323580231118
626445752991357581750081998392362846152498810889602322443621737716180
863570154684840586223297928538756234865564405369626220189635710288123
615675125433383032700290976686505685571575055167275188991941297113376
901499161813151715440077286505731895574509203301853048471138183154073
240533190384620840364217637039115506397890007428536721962809034779745
333204683687958685802379522186291200807428195513179481576244482985184
615097048880272747215746881315947504097321150804981904558034168269497
871413160632106863915116817743047925967093760 怎么样？这可以三千多位的数啊，你想想，我们电脑做这些东西的功能是多么强大，这么一串数，让你看都看晕了是不？　　只要你愿意，再大的数，都可以帮你求出。当然，只要程序改一改，就可以做任意数的任意乘方，大家可以去试试，甚至做任意大的数的阶乘，那用电脑来做简单就是太厉害了。我们说电脑的最强大的功能就是计算，我们从今天的阅读材料，大家应该可以看出我们电脑的计算能力了吧？　　当然，用上面的方法，空间的浪费太大了，例如求一个 2 的 １０００次方吧，才用了３００多位，而我们定义了一个１０００个空间的数组，求１００００时，竟然浪费了六千多个整型的空间。　　这个问题等我们学到链表后，自然可以解决这种问题。我们还是为我们的计算机的计算机能力而感到自豪吧，原来我们的电脑不光能打游戏，还能做这么厉害的运算（废话，计算机本来就是用来计算的嘛）。　　当然你可以用一个整型数部分来处理 4 个或者更多位的数据，那样做显然比牛刀做的程序要快，大家可以改出来试试，再看看做出来的结果是不是和牛刀的一样？

牛刀老师
你在上面举的那个例子我没大看懂

```
#include <iostream.h>
int main(int argc,char *argv[])
{
   for(int i=1;i<argc;i++)
        cout<<argv<<endl;
   return 0;
}
```

[/cpp]在 dos 下调用这个程序的时候在后面加上了参数
aaaa bbbb cccc

问题 1
您可不可以详细说明一下后面输入的几个参数是怎么被程序使用的？
实在是有点糊涂

问题 2
为了搞清楚程序是怎么使用参数的 我写了一个程序
#include <iostream.h>

int main(int a,int b)
{
cout<<a<<"和"<<b<<endl;
}
然后在命令提示符下 输入文件名 戴上了参数 参数分别是 1 和 2
可是程序运行后结果却是
3 和 4394656

搞得我头晕 这是怎么回事
还请牛刀老师指教 谢谢

**引用:**

原帖由 *BlakeLv* 于 2008-3-29 16:03 发表
牛刀老师
你在上面举的那个例子我没大看懂
[cpp]#include
int main(int argc,char *argv[])
{
for(int i=1;i<argc;i++)
cout
第一个是参数数量，至于为什么是 3，我觉得可能是因为你给的文件名里有空格，或者确实给了 3 个文件名。
第二个很好解释，那是指针，指向了存有参数的内存地址。

我的疑问是：
问题 1：能不能详细说明一下 两个参数的被调用过程 就是他们被使用的过程是怎么样的

问题 2：

关于我写的那个程序 我跟的参数是 1 和 2
所以应该是 a=1 b=2 啊 怎么就成了 a=3 b=4394656 呢？
郁闷

还请各位高手哥哥们指点一下 谢谢

是这样的，我们的系统并不是直接调用 main 函数的，如果是那样的话，那调用就应该是：

```
main(1,2);
```

了，而事实上，我们的 DOS 是字符系统，其调用不是按函数调用规则去调用的，而是将我们输入的字符，按空格分割成若干部分，多么部分呢？是从 argc 传递过来的，分别是哪几部分呢？是从 argv 部分传递过来的。

还举以上的例子吧：

```
#include <iostream.h>
int main(int argc,char *argv[])
{
   for(int i=1;i<argc;i++)
       cout<<argv[i]<<endl;
   return 0;
}
```

如果我们将文件保存为 aaa.exe （当然是编译过后的文件了），那么我们如果调用：

aaa bbb ccc

请你看上面，一共三个字符串：

aaa
bbb
ccc

所以 argc=3，而

```
argv[0]="aaa";
argv[1]="bbb";
argv[2]="ccc";
```

也就是说，第一个参数是表明，我们传进了几个参数，（注：包括文件名），而第二个参数，就分别表示传进来的几个数据。

以后学到 WinMain 也会遇到类似的问题，这种东西是交与系统调用的，而非函数调用，所以我们不能按函数调用的思维去考虑问题，否则就会百思不得其解。

main 的参数和其他函数的参数的传递是不一样的 这下明白了

## 牛刀教学〇一六：家有好邻居

如果说Ｃ＋＋的库文件是远亲的话，那么你做的函数就是近邻。

远新我们可以通过电话线“<>”来与他们进行交流，而近邻只需要用“″″″”四只小脚走过去就行了（当然，不是四只脚，是两只脚，一双手，呵呵）

```
#include &amp;lt;iostream.h&amp;gt;

#include ″min.h″
```

不同的是什么呢？上面的是Ｃ＋＋的库函数，而下面是我们做的，将不同的函数做在不同的文件中，我们前面已经讲过了。

当然我们更熟悉的是一套四合院，好多的邻居就住在一起，那更好了，见面只要打招呼就行了，我们这节课研究的邻居，就属于这种邻居。当然了，这是我们教学的需要喽，大家在做程序时，需要看程序的情况而定，如果是小函数，则完全可以住在一起（当然，不能同居啊），Ｃ语言与Ｃ＋＋语言是不允许同居的，但有些语言允许。

什么叫同居呢？就是函数里面还有函数，可能有些语言会允许这种情况存在吧？但Ｃ语言绝对不允许，法律挺严格。

例如：

```
int fun1(void)
{
...
   int fun2(void)
   {
   ...
   }
...
}
```

这种形式是错误的，也就是说，我们前面的循环、条件语句都允许嵌套使用，便函数不允许。我们只能用下面的形式来表示这两个函数：

```
int fun1(void)
{
...
}
int fun2(void)
{
...
}
```

当然喽，在这里，fun1 是不允许调用 fun2 的，如果想调用的话，则必须在前面加一面小镜子：

```
int fun2(void);
int fun1(void)
{
...
   fun2();
...
}
int fun2(void)
{
...
}
```

当然喽，由于 fun2 在 fun1 后，所以在 fun2 中是允许调用 fun1 的，而在上面，由于 fun2 在 fun1 前面有个小镜子，所以这两个函数之间就可以相互调用了。

当然如果 fun1 又调用 fun2，fun2 又同时调用 fun1，这种调用叫用嵌套调用，也是递归调用的一种形式，如果的确两种函数都同时对对方进行调用的话，那千万要注意，不要引起“死循环”，我们就不举例子了，大家做程序时，总之要注意就行了。

那么我们这节课讲什么内容呢？

```
#include &lt;iostream.h&gt;

int main();
int fun()
{
   cout&lt;&lt;"Test"&lt;&lt;endl;
   main();
   return 0;
}
int main()
{
   fun();
   return 1;
}
```

这个程序你想不到吧？牛刀也想不到，牛刀只是试试看，竟然也能够运行，并且进入了死循环。

注意哟，不要将 main 放在前面，因为那样的话，一旦执行到 main()函数中的 return 语句，就返回系统了。

如果不试一下，我们就做教材，那么我的结论肯定是 main()函数不能被调用，事实却并非如此，所以我们当看到书上有什么介绍时，一定不要全信，尽量抱着怀疑的态度去电脑上试试，或者你可以怀疑，这是不是一个漏洞呢？那是黑

客的事情，我们不讨论那个。

行了，有关递归调用到此结束。
对了，我听过一个递归故事，不知大家听没听过：

从前有座山，山里有个庙，庙里有个老和尚，老和尚给小和尚讲故事，讲了一个什么故事呢？讲的是下面的故事：
从前有座山，山里有个庙，庙里有个老和尚，老和尚给小和尚讲故事，讲了一个什么故事呢？讲的是下面的故事：
从前有座山，山里有个庙，庙里有个老和尚，老和尚给小和尚讲故事，讲了一个什么故事呢？讲的是下面的故事：
……
……
……

呵呵，这个故事可以让你听一辈子。

行了，再讲讲我们的邻居吧，邻居的故事多着咧。

```
#include &lt;iostream.h&gt;
int main()
{
   double PI=3.1415926;
   double radius=5.0;
   double area=PI*radius*radius;
   cout&lt;&lt;"半径为"&lt;&lt;radius&lt;&lt;"的圆的面积为：
"&lt;&lt;area&lt;&lt;endl;

   return 1;
}
```

邻居不能带家来，家中人却可以出去做邻居，呵呵，上面的程序大家一看就明白吧，什么意思呢？求一个圆的面积呗。一个半径为 5 的圆的面积。而大家知道，求圆的面积的方法的固定的，是吧，所以我们做程序的人就喜欢将这些东西给从程序中“拿”出来，做邻居，呵呵，方便啊：

```
#include &lt;iostream.h&gt;
void CalArea()
{
   double PI=3.1415926;
   double radius=5.0;
   double area=PI*radius*radius;
   cout&lt;&lt;"半径为"&lt;&lt;radius&lt;&lt;"的圆的面积为：
"&lt;&lt;area&lt;&lt;endl;
}
int main()
{
```

```
   CalArea();
   return 1;
}
```
当然喽，我们喜欢跟邻居借东西，而不喜欢家中准备东西，因为那样方便啊，主要是省事，那么，怎么借呢？加个 return 语句给返回就行了，那返回的值就是邻居“借”给你的了，呵呵。

但在程序中有个好处是：借了不必还，哇，怪不着程序们那么喜欢用这种邻居：

```
#include &lt;iostream.h&gt;
double CalArea()
{
   double PI=3.1415926;
   double radius=5.0;
   double area=PI*radius*radius;
   return area;
}
int main()
{
   double area=CalArea();
   cout&lt;&lt;"半径为"&lt;&lt;5&lt;&lt;"的圆的面积为：
"&lt;&lt;area&lt;&lt;endl;
   return 1;
}
```
当然，这还不死心，竟然向邻居借东西还要提条件。（提条件的方法是通过参数列表给传过去：

```
#include &lt;iostream.h&gt;
double CalArea(double radius)
{
   double PI=3.1415926;
   double area=PI*radius*radius;
   return area;
}
int main()
{
   double radius=5.0;
   double area=CalArea(radius);
   cout&lt;&lt;"半径为"&lt;&lt;radius&lt;&lt;"的圆的面积为：
"&lt;&lt;area&lt;&lt;endl;
   return 1;
}
```
好了，这就是我们的函数可以为我们做的事情。

告诉大家一个秘密，函数中的数据与程序中的形参不一定要与调用它传入的参数标识符一致，可以不一样，看：

```
#include &lt;iostream.h&gt;
double CalArea(double r)
{
   double PI=3.1415926;
   double mj=PI*r*r;
   return mj;
}
int main()
{
   double radius=5.0;
   double area=CalArea(radius);
   cout&lt;&lt;"半径为"&lt;&lt;radius&lt;&lt;"的圆的面积为：
"&lt;&lt;area&lt;&lt;endl;
   return 1;
}
```

大家看，函数中的数据都被我们换了，结果还是一样。

这也是有好处的，我们函数与文件不一定在一起是吧？人家的东西我们怎么会知道呢？其实我们没必要知道，我们只要顺序提条件就行了，而邻居会按顺序给你记住，然后回去给你准备，准备好的结果送给你就行了。

那么我们就感到怪怪的了，那么这个参数是怎么传过去的呢？我们以后再说，我们先说说我们函数的定义吧，牛刀就是厉害，先给大家做些程序，然后再说定义，真想不通，连牛刀自己都想不通，甚至不知道这样别人能不能理解，理解当然有困难了，能不能上机做试验。

函数的格式为：

```
数据类型  函数名(形式参数表)
{
   语句
}
```

很简单吧，大括号像复合语句，函数有些像一个大的复合语句，只不过是多传进了几个参数而已，而复合语句不具备这种功能而已。当然正因为如此，这我们就不叫复合语句了，而叫“函数体”，想想这些名字起得怪怪的，会用就行了，有什么必要知道这些啊，哎～～～

这里的函数体我们再强调一下，数据类型如果为 void，则不需要 return 语句，否则必须用 return 语句返回一个函数值，并且数据类型最好与函数的数据类型一致为好（能不能不一致，大家可以去试试，估计应该没事，因为Ｃ与Ｃ＋＋是默认数据类型自动转换的。）

对了，大家在同一程序中不要定义两个或多个 main 函数噢，牛刀刚才试了试，编译通不过，呵呵，牛刀有些无聊，但实践嘛，就是这么回事，为什么牛刀说自己的教材是权威呢？厉害吧？

形式参数表：

就是一串用逗号分开的参数，这里要注意两点：

一、形式参数不是逗号表达式，所以这里的逗号不具有计算功能。
二、开式参数每个定义都必须自己定义自己的类型，不能同时用一个类型，还是举例子吧：

```
f(int i,int j,float k)    //这是正确的
f(int i,j,float k)    //这是错误的
```

即，每个变量都要自己定义。
注意了，这里不同与参数的定义，如果我们在标识符的定义中用了：

```
int i,int j;
```

那么就是错误的了，等等牛刀试看看，报不报错。
所以说，大家知道了，特殊情况，特殊处理，函数就有函数自己的特色，谁叫你喜欢邻居呢？
对了，牛刀试过刚才那种情况了，报错了的啊。也就是不能那样定义。
　　当然参数可以是不带，可以带一个，可以带多个，甚至……
　　不过我们注意，即使是不带参数，后面的小括号可不能省，你去做个程序省省看，看计算机报不报错？

```
void fun()
{
}
```

大家看，小括号一省：

```
void fun
{
}
```

上面变了一个定义数据的类型了是不？（你不要说 void 是无类型，在Ｃ与Ｃ＋＋可以表示任何类型，当然，这我们以后讲到指针再讲）
　　于是，程序发现定义的数据后面没有了“;”，却多了一个“{”，你说她不是莫名其妙吗？
　　所以就报错了呗。

　　行了，我们下一讲就帮大家介绍我们邻居的种种特点，我们彻底地了解好我们的好邻居，真正能够将我们想做的事放心地交给邻居去做，是不是？

作业：
　　定义一个函数，求两个数之和
　　定义一个函数，求一个小于１０的数的阶乘（不许用递归）
　　定义一个函数，判断一个数是不是质数（只能被自己与 1 整除的自然数，要求对于错误输入〔如输入零或负数〕要报错，并拒绝执行）
　　定义一个函数，将小写字母转换成大写字母（要求：判断是否为小写，不是小写不处理。
大家好，大家知道函数 sin x　从 0 到 π 之间与 x 轴围成的面积有多少吗？我告诉你吧，答案能将你惊得目瞪口呆。结果是：２。
　　这么简单啊，我怎么没想到呢？不说你没想到啊，谁都不容易想到竟然有什么简单的题目是吧？竟然和１＋１的结果一样。

　　再问你 y=x*x 与 x 轴与线 x=1 围成的面积有多少呢？是 1/3，不信吗？我们

今天就来用程序说话，我们做做这个面积，看看到底我们的想法有错没有？

```
#include &lt;iostream.h&gt;
#define STEP 0.01
int main()
{
   double x,y;
   double sum=0.0;
   for(x=0;x&lt;1;x+=STEP)
   {
       y=x*x;
       sum=sum+STEP*y;
   }
   cout&lt;&lt;"函数 x*x 从 0 到 1 部分与 x 轴与 x=1 构成的面积为："&lt;&lt;sum&lt;&lt;endl;
   return 0;
}
```

我来解释一下程序啊，我们显然是不能用很有效的数学手段去将这一段面积给求出来，但我们可以求出她的进似解啊，怎么办呢？我们从 0 到 1 分成若干等分，每等分的小步长用 SETP 来表示，我比较喜欢用宏来定义，因为这样一来，想改动比较方便（也的确比较方便）

那么每个等分点求出来的 y=x*x 的值作为一个长度，而 SETP 作为宽度，就可以作一个长方形，我们就用这长方形之和：sum+=STEP*y 来近似这个函数的面积。

当然，上面求出来的结果为：

```
函数 x*x 从 0 到 1 部分与 x 轴与 x=1 构成的面积为：0.32835
```

好像与我们想像的结果不符啊，当然，我们分的长方形的步长太大，也就是说上面与函数图形还有很大的空间没有被包上，于是，我们将步长改小：

```
#include &lt;iostream.h&gt;
#define STEP 0.0001
int main()
{
   double x,y;
   double sum=0.0;
   for(x=0;x&lt;1;x+=STEP)
   {
       y=x*x;
       sum+=STEP*y;
   }
   cout&lt;&lt;"函数 x*x 从 0 到 1 部分与 x 轴与 x=1 构成的面积为："&lt;&lt;sum&lt;&lt;endl;
   return 0;
}
```

```
函数 x*x 从 0 到 1 部分与 x 轴与 x=1 构成的面积为：0.333383
```

这个结果怎么样？还可以改小：
#define STEP 0.000001
这下行了：
函数 x*x 从 0 到 1 部分与 x 轴与 x=1 构成的面积为：0.333333
怎么样？一个小程序，就可以将面积求出来了吧？我们加入一个头文件 math.h，然后求 sin(x)从 0 到 π 看看：

```
#include &lt;iostream.h&gt;
#include &lt;math.h&gt;
#define STEP 0.000001
#define PI 3.141592654
int main()
{
   double x,y;
   double sum=0.0;
   for(x=0;x&lt;PI;x+=STEP)
   {
       y=sin(x);
       sum+=STEP*y;
   }
    cout&lt;&lt;" 函 数  sin(x) 从  0  到  π  部 分 与  x  轴 构 成 的 面 积 为 ："&lt;&lt;sum&lt;&lt;endl;
   return 0;
}
```

结果：
函数 sin(x)从 0 到 π 部分与 x 轴构成的面积为：2
你瞧，计算机给我们的结果多么干脆？直接就是 2，这下你信了吧？当然如果你上面改了 PI/2，那么给出的结果特别完美，就是 1 。

行了，可能大家认为好玩，其实这就是我们计算机做的积分，可能有好多朋友微积分还没有学到，但学过的朋友可能知道要背好多好多的公式，计算机不要，她就只会傻傻地算，然后给出结果就行，你看上面的程序多么通用？只要你给出函数，马上就可以将积分的结果给你硬“算”出来。

没学过微积分的同学们也应该知道我们的计算机在做什么，如果不知道的话，建议你画个图，试试就知道了，总之，你想做一个数字的积分，那很简单。

有没有朋友将这个程序简单改改，求一个圆的面积，然后用面积比半径，将圆周率 π 结算出来瞧瞧，这可完全是计算机的方法啊。

计算机就是这样，勤勤恳恳，兢兢业业，从来都是任劳任怨，为我们人类做自己的事，你发现了吗？我说计算机，用的都是“她”，因为我发现计算机太伟大了，虽然只是人们的工具，可你想过我们的天气预报是怎么来的么？你想过卫星是怎么上天的么？你想过利伟是怎么返回地球的么？你想知道你是如何玩游戏的么？……

行行业业，都有计算机的身影，她默默地为我们人类的社会而奔碌着，操劳着，一旦计算机被人们称着了“电脑”人们的脑袋就彻底变懒，总想着计算机帮我们搞定一切。于是，人们就想到了－－人工智能。

## 牛刀教学〇一七：墙上小抽屉

与邻居家打交道，可不像家里一样，有钱自己拿，有饭自己吃，那必须经过人家主人的同意是吧？基本礼貌。于是，邻居就会准备一个小本本，你叫他干啥，他就记下去，为你做，回来将结果告诉你罢了，而下面的一些东西就无能为力了。

```
#include <iostream.h>

void swap(int i,int j)
{
    cout<<"进入了交换函数。（邻居部分）"<<endl;

    cout<<"刚进来，我们 i 的值为："<<i<<"    j 的值为："<<j<<endl;
    int t;//我们准备一个中变量，准备交换一下 i 与 j,大家看我们是怎么做的。
    t=i;    //先将 i 的值保存好
    i=j;    //然后将 j 的值给 i,此时，i 的值在 t 中了
    j=t;    //所以我们再将 t 的值给 j 于是，i 与 j 的值便交换了
    cout<<"我们交换后,i 的值为："<<i<<"    j 的值为："<<j<<endl;
    cout<<"准备退出函数（邻居部分）"<<endl;
}
int main()
{
    cout<<"进入主函数"<<endl;
    int a,b;
    a=4;
    b=7;
    cout<<"进入交换函数前，a="<<a<<"    b="<<b<<endl;

    swap(a,b);

    cout<<"交换以后，a="<<a<<"    b="<<b<<endl;

    return 0;
}
```

我们做了一个交换函数，但退出函数一看，我们的目的却没有达到，哎，没办法，谁让人家是邻居呢？怎么办，这么点简单的问题都解决不了（啊，竟然抱怨起邻居来了，呵呵）。

C＋＋的编译者们也知道我们邻居关系不好处啊，他们给大家与邻居的墙上做了一个小“抽屉”－－引用（&）

可能你一看，这不是“与”操作符么？还可能取地址，难道，难道还能？？？

是了，大家看程序：

```
#include <iostream.h>

void swap(int &i,int &j)
{
    cout<<"进入了交换函数。（邻居部分）"<<endl;
    int t;
    t=i;
    i=j;
    j=t;
    cout<<"我们交换后,i 的值为："<<i<<"    j 的值为："<<j<<endl;
    cout<<"准备退出函数（邻居部分）"<<endl;
}
int main()
{
    cout<<"进入主函数"<<endl;
    int a,b;
    a=4;
    b=7;
    cout<<"进入交换函数前，a="<<a<<"    b="<<b<<endl;

    swap(a,b);

    cout<<"交换以后，a="<<a<<"    b="<<b<<endl;

    return 0;
}
```

怎么回事？不就是函数的参数表中参数前多了一个取地址符么？住！C＋＋设计者可没这么说，他们美其名曰：引用。

于是，上面的程序，我们简化为下面的形式：

```
#include <iostream.h>

void swap(int &i,int &j)
{
    int t;
    t=i;    i=j;    j=t;
}
int main()
{
    int a=4,b=7;
    a=4;
    b=7;
    cout<<"进入交换函数前，a="<<a<<"    b="<<b<<endl;
```

```
    swap(a,b);

    cout<<"交换以后，a="<<a<<"    b="<<b<<endl;

    return 0;
}
```

呵呵，说两个：
一个：我们的“引用”其实就是这么简单，如果你想要对别人的数据进行操作，前面加个“&”就一切ＯＫ了（当然，我们以后讲过指针还会教大家其他的方法，不过你可能也舍不得用，还是这种方法简单啊。
二一个：像交换数据这样经典的程序，我们竟然可以写在一行上面

```
    t=i;    i=j;    j=t;
```

如果按我们一般的想象，这不符合我们编程风格的要求啊？呵呵，这种程序太经典了，只要是学过程序的，到这儿眼一刷就知道这一段程序的功能了，所以我们不必拘泥于我们形式，一行就实现一个功能，这么好？

但不是经典的程序，建议大家还是分开的好，只不过是长了点而已。

行了，一旦我们明白了“引用”这个概念后，好多的编程问题几乎是迎刃而解，但编程也要注意了，因为这种函数对我们原有数据是有破坏性的。我们举一个例子吧，当然这里是我们有意调用这个函数的：

```
#include <iostream.h>
void test(int &i)
{
    ++i;
}
int main()
{
    int a=4,b=7;
    test(a);
    test(b);
    cout<<a+b<<endl;
    return 0;
}
```

我们当然这个程序是有意做的喽，我们想要４＋７的值等于１１但结果却被函数给改了。这是我们一般做函数所不需要的。当然，如果不是有必要的话，引用请不要过多地用，因为外部函数使用人员根本不会知道你这个程序哪一个参数被引用了，所以我哪个数字被改了，我自己都会莫名其妙，特别是搞多人开发时，可能很容易出现所有人的程序都正确，但互相一拼就错了，并且我们的函数应该有自维护性的，即为：我们有可能以后想为某一个函数增加一个功能，但参数也不需要变，返回值也没变，但如果一旦参数中有引用，你在程序中也必需要对这个参数操作，但一旦你发现误操作以后，再来检查函数，可能你根本不知道你在

哪儿改的了，那么可能在不断的改动中你的一个很好的函数就被这些小小的参数组毁了，我不是说过嘛，我宁可写一个一万行的程序，而不去查一个一万行的程序（当然是指没有原则性错误的，有些错误计算机就报错了不是？）当然，调试也是一种技术。

那么我们的引用正常使用在哪里呢？一是程序很小，二是真需要这种功能，例如上面我们的程序目的就是交换两个数，那你去用吧。人家编译器支持你不就是让你使用的吗？

如果一个大程序真的需要这种改变数据的情况，一般使用我们以后讲的指针来使用，如果你真的进入了ＡＰＩ与ＭＦＣ的Ｗｉｎｄｏｗｓ编程，你将会看到指针是铺天盖地。

行了，作业不布置了，俺的贴子也没人看，做得那么全，有必要吗？哎～～～
可能大家也看过我以前的关于筛法求素数的程序，虽然用那种方法求素数很简单，但是，如果给你一个数，它是合数还是质数呢？
这的确是一个问题，可以说，一个比较小的数，还真没说的，只要从 2 到这个数的根号的所有数，做一遍求余判断，就行了：

```
#include <iostream.h>
#include <math.h>
int main()
{
    int i,j;
    cout<<"请输入一个数：";
    cin>>j;
    for(i=2;i<=sqrt(j);i++)
        if(j%i==0)
        {
            cout<<"\n 这个数不是一个素数，是一个合数！"<<endl;
            return 0;
        }
    cout<<"\n 这个数是质数"<<endl;
    return 0;
}
```

这个程序大家想一想为什么只要测试到 sqrt(j)就行了？
再说一个话，大家看，这里出现了两个 return 语句，这两个 return 并不矛盾，如果你输入了一个是合数，那么会从上面一个 return 跳出，而如果是一个质数，则上面一个 return 语句被跳过不执行，而执行下一个 return 语句。所以说我们的程序可以很完美地执行。

以前有很多人提到过函数只能有一个出口与一个入口的问题，什么意思呢？是指我们做函数的话，函数只能从一个地方结束，否则的话不符合程序设计的一些原则，也就是说，我们设计程序应该是单入口，单出口的。

而我们上面的设计，显然有两个出口，但程序也一样地可以运行，但这种运行必须将不同的出口放在不同的条件判断之中，使我们程序每次执行时，总只能从一个出口出去，而如果我们编写了以下的程序：

```
for(int i=0;i<10;i++)
{
    return 0;
    ...
}
```

大家看，上面的循环一点意义都没有，因为程序在第一次循环就已经出去了。根本不执行循环体的其他部分，再看：

```
int fun()
{
...
return 0;
return 1;
return 2;
return 3;
...
}
```

这种多出口也一点意义都没有，因为第一次执行就返回了，并且返回的值肯定是0

告诉大家一个秘密，程序中有一个语句，叫做 goto 语句，现在不建议使用了，其实这个语句配合 if 条件语句，几乎可以替代任何的循环，与一些特殊的需要，我们还是举一个 1 加到 1 0 0 的例子吧：

```
#include <iostream.h>
int main()
{
    int i,sum;
    i=1;
    sum=0;
label:
    sum+=i;
    i++;
    if(i<=100)goto label;
    cout<<"1+2+...+99+100="<<sum<<endl;
    return 0;
}
```

我们上面的 label:是一个标签，标签必须为冒号（:）结尾，于是，每做一次都判断一下，是否条件成立，若成立，则回到标签处继续执行，说实说，这种程序编出来，叫你看起来很舒服，意义又很明了，几乎可以替代任何循环与分支，甚至可以实现一些循环与分支不能实现的东西，如我们上面所说的 break;等功能，

很灵活。

再有一个特方便的应用，一段程序我们不需要了，可以弄个标签，做个 goto，很方便地跳过去，能从很深的循环内跳出，或者跳进循环……

这么好的东西，现在的程序都不建议大家使用了，那些人的理由就是，会破坏程序的结构化，可能引起程序有从个出口，于是，人们发明了好多替代的东西，如 for 语句，while 语句，switch 语句……

说实话，一下子将程序弄得复杂了，可是，你还别说，程序容易发生的问题却是真的少了，并且再配以现在要求的编程风格的问题，使得我们的编程走向了正轨，并且有了一定的原则。

goto 语句，我们在程序中可以使用，可以灵活的使用，牛刀可不限制大家这些，但一定要注意，这个语句的使用，很容易出现我们意想不到的情况。

行了，我们还看看质数问题吧，我们的程序是没错误，可惜根本没有寻找到一个真正有效的方法来判断一个大数是否是质数，虽然我们程序中使用了根号，但还是不行，为什么呢？这里我们就要讲到计算机的效率问题了：

一个好的程序，不但要可以计算出结果，而且要效率高，虽然现在说每多少年，计算机的计算机效率翻多少翻，但一个大数的质数测试仍然是个问题，为什么呢？

因为我们的程序是要一定的时间的，这个程序执行的时间分几种：一种是程序的执行时间，一种是指令执行的时间，一种是时钟周期。

也就是说，电脑中有一个时钟，可能它一秒钟要变动几亿次甚至更高，但每个指令需要好多个时钟周期（至少一个），而每个程序又有好多指令，再加上循环的出现，我们就要考虑这个问题了，看一个实例：

```
for(int i=1;i<100;i++)
    for(int j=1;j<100;j++)
        for(int k=1;k<100;k++)
            for(int l=1;l<100;l++)
            {
            ...
            }
```

其实这个程序已经很复杂了，我原来为什么叫大家循环嵌套层数不要太多，就是这个原因，

我们假设这个循环体好多指令要执行 1 毫秒（假设有好多程序），那么我们这个四重循环就要执行（100000000 毫秒）注：由于循环需要时间，不过我们忽略了它，其实应该考虑的。

有多长时间呢？100000 秒（我们将单位转换过来了）那是多长时间，什么概念呢？我们再转化一下：为 27.777 个小时，那么我们上面的程序改一下，会出现什么情况呢？

```
for(int i=1;i<1000;i++)
    for(int j=1;j<1000;j++)
```

```
for(int k=1;k<1000;k++)
    for(int l=1;l<1000;l++)
    {
    ...
    }
```

我们每位多替了一个零，于是就变成了：27777.7778 个小时，也就是：３．１７５年。

什么概念？原来需要一天多的程序，多了四个零，就需要三年多才能完成。我们称这种增加速度为级数增长，一个程序的随着复杂着增加时，线性增加最好，但如果不是线性的，后果就很恐怖了，有些程序，再好的计算机，也没有用，我们的大质数判断就是如此。

我们计算机的速度增加了，一个大数可以判断是否为质数了，可能我只要在质数上加上几位，可能计算机又不能胜任了，或者一加就需要再来十几年，所以说，现在为止，大型数分解质因数与判断是否为质数的问题还没有被解决，也正因为如此，质数常常被人们用着电脑上的加密解密工作。这种加密技术，暴力破解是不现实的。还有一种技术叫做公开秘钥技术，就是这种加密方法加密后，用这种官密钥没办法解密，而必须是一种密秘的密钥才可以解密，这样一来，我们的数据传输就特安全，因为数据只要被加密，就是安全的，就连加密者都没有办法去对数据进行解密。你想想，你的一个很大的文件，在网上传输，如果用了这种技术，别人就是把这段数据给偷去了，也只不过是废数据一堆而已。

假如你能够用程序解决这个问题，你就真够厉害的了，那么一旦解决，那么以前的所有密码学在你面前就要府首称臣了。

牛刀老师看我这样理解对不对：
假如主函数中两个变量(a,b) 传递到了子函数 xxx(int i,int j)里面
那么传递的只是 a 和 b 的值，内存中临时开启了 i 和 j 两个空间来存放 a 和 b 的值
子函数对这两个存放在 i 和 j 里面的数值操作完毕后这两个临时的空间消失
所以对主函数中的 a 和 b 没有影响。
但是如果传递的形式是由主函数传递给 xxx(int &i,int &j)的话
传递的就不是 a 和 b 的值了 而是传递 a 和 b 的地址
这样 子函数操作的 i 和 j 其实是在操作 a 和 b 的物理值
返回主函数后 a 和 b 的数值改变

我一开始是这样理解的 可是后来发现不对

因为前面学到的 scanf 这个函数中 &是这样用的
scanf("%d%d",&i,&j);
接受的地方是 i 和 j 的空间
&打开了 i 和 j 的房间门 让数据进房间

问题就在这里

主函数中两个变量(a,b) 往子函数 xxx()里面传
这时候使用引用 以 xxx(int &i,int &j)的形式接受参数
&打开了 i 和 j 的门 使 a 和 b 的数据进去
那么子函数操作的还是 i 和 j 啊 并没有操作 a 和 b
可是 a 和 b 的值还是改变了

这是为什么呢？这样理解错在什么地方呢？

非也非也，事实是这样的：

我们的参数只能有两种传递方法：值传递与引用传递。
值传递
```
int fun(int a,int b);
```

引用传递
```
int fun(int &a,int &b);
```
那么我们就需要知道什么叫引用：
```
#include <stdio.h>
void main()
{
        int a=3;
        int &b=a;          //请注意，这是引用，不是取地址
        printf("改变值前：a=%d,b=%d\n",a,b);
        //下面我们对 b  值进行改变
        b=5;
        printf("改变值后：a=%d,b=%d\n",a,b);
}
```
上面的运行结果为：
改变值前：a=3,b=3
改变值后：a=5,b=5
显然上面表示，我们 a 与 b 其实是同一个数据，好，为了理解，我们下面来看两种不同的定义：

* 取某个地址地方的值，所以其后面应该是个地址，所以我们可以用 int *a; 去定义一个地址，也就是指针，而下面我们看：
& 取某个值的地址，所以我们后面定义的应该是一个值，什么位置的值呢？是 a 处的值，即：
int &b=a;

也正因为如此，我们这种引用的定义只能在定义的时候进行，而不能用以下程序：

**代码:**

```
    int &b;
    b=a;
```

上面的显然是错的。

也就是说，这种 & 的引用定义，必须是定义的时候就立刻给其赋一个值，而赋给 b 的并不是 a 的地址，而是将 b 与 a 指向同一个地址。

也就是说，这其实是：一个地址，两个变量，就好像一个人的两个名子一样，不管这人称之为“张三”，还是其他名子，只要说出他的某个名子，就都是这个人，从这个角度来说，其实就是一个人本身。

从这一点上来说，引用传参其实也是一样的：
*复制内容到剪贴板*

**代码:**

```
int fun(int &a,int &b);

//    那么我们传进值
int x=3;
int y=5;

//    那么我们如果调用函数：
fun(x,y);

//    那么这里的 a 其实就是 x ，而 b 就是 y 了，并且因为是引用，所以其实是一个地址，也就是说，我们对 a,b 进行操作其实就是直接对 x,y 进行操作，那么我们就实现了对函数外部数据的改变。
```

但是，地址传输就不同了，地址传输其实还是值传递：

```
int fun(int *a,int *b);
//    因为前面是 * 定义了地址，所以其实函数中的 a,b 其实不是数据，而是地址，传递进来的还是 a 与 b 这两个地址，所以我们要对 a b 处的值进行处理，那么我们就必须使用 * 取值符取得这个地址处的值。
```

下面我们就来比较一下两种传参方式的不同函数：

```
//地址传递
void swap( int *a,int *b)
{
   int c;      //我们定义一个中间变量
   c=*a;      //看，由于 a 其实是一个地址，所以必须用 * 取得地址处的值，而地址其实也是另一个数据，其本质还是值传递，只不过传进来的是地址值而已，我们改变的是这个地址地方的数据，而如果两个地址一样，那么这两个地
```

址指向的数据也必然一样，这就是原理

```
   *a=*b;
     *b=c;
}

//  下面我们看看引用传递

swap(int &a,int b)
{  //用了引用后， a 就是传递进来的参数本身，一个参数两个名子
  int c;
   c=a;   //因为  a  就是传递进来的参数，所以直接就可以改变参数据了
   a=b;
   b=c;
}
```

不知够详细不？如果还不明了的话，牛刀开专讲。

牛刀老师看看这样对不对

假设主函数中的 i 传了进来
1.
如果传给了*a
对于*a 来说 a 里面放的是 i 的地址 *a 就是指向 i 的地址的指针
对*a 的操作就是间接对 i 的操作
2.
假如传给了$a
对于&a 来说 &已经不是取地址符了 而是引用符
通过&a 这样引用后
a 这个变量其实就是 i 这个变量
对 a 的操作就是直接对 i 的操作
只不过两个名字 其实是一个人

不知这样理解对不对

大体意思弄得差不多了 但是如果牛刀老师有空的话还是希望能讲的详细一点
因为我觉得搞清楚原理比搞清楚操作更有价值 呵呵
谢谢牛刀老师

**牛刀教学〇一八：共用小厨房**

上一讲我们讲了小抽屉，那只是C＋＋的一种技术而已，也是人们对一些东西抽像出来的一种对于数据使用的方式，同时，这种引用又被人们作为一种新的定义：别名。

可能你有小名吧？你在外面别人叫你大名，在家里大人叫你小名，还不是同一个人，所以说，引用也被有些人称为“新起的一个名子”。

看程序：

```
#include &lt;iostream.h&gt;
int main()
{
   int i;//
   int &p=i;//呵呵，为 i 定义了一个新的名子：
   i=3;    //看，现在 i=3 啊，我们下面输出 p
   cout&lt;&lt;"p="&lt;&lt;p&lt;&lt;endl;
   p=10;    //看，下面我们再输出 i
   cout&lt;&lt;"i="&lt;&lt;i&lt;&lt;endl;

   return 0;
}
```

通过这个例子，大家明白了吧？i 就是 p，p 就是 i，同一个变量，不同的名子而已，所以大家看上节课我们做程序时，不过是在参数列表中为原来的变量起一个新名子吧？变量却被别人拿去了，呵呵，计算机中竟然也有“偷梁换柱”，阴险啊阴险。所以说，我们上一节课说的引用，其实就是一种偷梁换柱技术的具体应用而已，希望大家能够掌握。

而这节课我们说的是什么呢？共用小厨房，怎么回事呢？这涉及到计算机变量作用域问题：

看程序：

```
#include &lt;iostream.h&gt;

int i;

void seti()
{
   i=100;
}


int main()
{
```

```
   seti();
   cout&lt;&lt;i&lt;&lt;endl;

   return 0;
}
```

瞧瞧，你瞧瞧，这个 i 的定义怎么不在任何函数里面呢？怎么会在函数外面呢？

这个好啊，在外面定义，所以任何函数都可以使用它啊，所以我们调用 seti 函数后，i 被赋值为 100

而 i 在 seti 函数中，在 main 函数中，都可以使用，所以我们说它是共用的，所以说这是“公用小厨房”。

当然喽，我们再定义两个数字，各定义各的：

```
#include &lt;iostream.h&gt;

int i;

void seti()
{
   i=100;
   int j;    //你瞧瞧，变量的定义放在哪里都是可以的。
   j=50;
   cout&lt;&lt;"在 seti 函数中:j="&lt;&lt;j&lt;&lt;endl;
}


int main()
{
   int j;
   j=20;
   cout&lt;&lt;"调用 seti 之前，j="&lt;&lt;j&lt;&lt;endl;
   seti();
   cout&lt;&lt;i&lt;&lt;endl;

   cout&lt;&lt;"调用 seti 后，j="&lt;&lt;j&lt;&lt;endl;

   return 0;
}
```

你瞧瞧，瞧瞧，j 是在函数中各自定义的，所以没有变啊，所以我们说这个函数中的 i 为全局变量，而 j 为局部变量，局部变量，各部分的定义不影响其他地方局部变量的定义。

这有什么好处呢？你想想，你做一个函数用到了 i，别人的函数也用到了 i，

如果不是各自独立的，那么如果函数在一起运行，那就会出现问题了是不？所以在编程语言中，为各自不同的变量分配了不同的内存地址：我们知道：（&）是取地址符号，是不？我们试看看：

```
#include &lt;iostream.h&gt;

int i;

void seti()
{
   i=100;
   int j;    //你瞧瞧，变量的定义放在哪里都是可以的。
   j=50;
   cout&lt;&lt;"在 seti 函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
       &lt;&lt;"              j 的地址为："&lt;&lt;&j&lt;&lt;endl;
}



int main()
{
   int j;
   j=20;
   seti();
   cout&lt;&lt;"在主函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
       &lt;&lt;"            j 的地址为："&lt;&lt;&j&lt;&lt;endl;
   return 0;
}
```

我的计算机运行的结果为：

```
在 seti 函数中，i 的地址为：0x004299B8
             j 的地址为：0x0012FF24
在主函数中，i 的地址为：0x004299B8
           j 的地址为：0x0012FF7C
```

大家看，i 的地址是一样的，而 j 的地址却不一样，这就告诉我们，i 为全局变量，而 j 是局部变量，局部变量的定义相互是不影响的。

好了，那么我们在局部可不可以定义一个和全局变量相同的变量呢？

```
#include &lt;iostream.h&gt;

int i;

void seti()
{
   i=100;
   int j;    //你瞧瞧，变量的定义放在哪里都是可以的。
```

```
    j=50;
    cout&lt;&lt;"在 seti 函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
        &lt;&lt;"              j 的地址为："&lt;&lt;&j&lt;&lt;endl;
}


int main()
{
    int j;
    int i=30;
    j=20;
    seti();
    cout&lt;&lt;"在主函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
        &lt;&lt;"             j 的地址为："&lt;&lt;&j&lt;&lt;endl;
    cout&lt;&lt;"在主函数中,i="&lt;&lt;i&lt;&lt;endl;
    return 0;
}
```

这里会出现什么情况呢？我们又将 i 的结果打印出来了，作为对比试验：结果为：

```
在 seti 函数中，i 的地址为：0x004299B8
              j 的地址为：0x0012FF20
在主函数中，i 的地址为：0x0012FF78
            j 的地址为：0x0012FF7C
在主函数中,i=30
```

大家看，我们在 seti 中，地址没有变，而在主函数中，i 的地址变了，也就是说，在函数中，全局变量被局部变量给“覆盖”了，正是这种覆盖技术，保证了函数中的各种定义不受外界的影响，否则我们想如果我们函数先做出来，可由于定义了一个全局变量，函数就不能用了，岂不是一件很悲伤的事？于是，有了这种覆盖技术，即：局部变量无条件覆盖全局变量。

当然在变量中，还有一种文件变量，即在文件最上面使用:extern 来声明其变量，但说实话，很难用到，大家自己去试试看，牛刀这儿就不给大家试了，你想想，一个个模块是我们基本结构，如果让一个变量来左右我们编程思路，那么我们是不是太累了？所以说，这种东西，C＋＋是给我们了，但却说实话，不太常用，如果真想用，一般用其他的技术来使某个变量在各种程序中通用（例如上面的引用啊，后面的指针啊等等）。就好像 goto 语句一样，我们要讲，但正常不用。

行了，那么我们如果想在程序中有与全局变量相犯冲的情况下，我们如何使用外部变量呢？C＋＋给我们提供了一种运算符：全局变量运算符（::）两个冒号，呵呵，我们看程序：

```
#include &lt;iostream.h&gt;

int i;
```

```
void seti()
{
   i=100;
   int j;    //你瞧瞧，变量的定义放在哪里都是可以的。
   j=50;
   cout&lt;&lt;"在 seti 函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
       &lt;&lt;"              j 的地址为："&lt;&lt;&j&lt;&lt;endl;
}


int main()
{
   int j;
   int i=30;
   j=20;
   seti();
   cout&lt;&lt;"在主函数中，i 的地址为："&lt;&lt;&i&lt;&lt;endl
       &lt;&lt;"            j 的地址为："&lt;&lt;&j&lt;&lt;endl;
   cout&lt;&lt;"在主函数中,i="&lt;&lt;i&lt;&lt;endl;
   cout&lt;&lt;"全局变量 i 的地址为："&lt;&lt;&::i&lt;&lt;endl
       &lt;&lt;"全局变量 i 的值为：  "&lt;&lt;::i&lt;&lt;endl;
   return 0;
}
```

看结果：

```
在 seti 函数中，i 的地址为：0x004299B8
             j 的地址为：0x0012FF20
在主函数中，i 的地址为：0x0012FF78
           j 的地址为：0x0012FF7C
在主函数中,i=30
全局变量 i 的地址为：0x004299B8
全局变量 i 的值为：  100
```

怎么样？全局变量又回来了。事实就是如此，（::）这个运算符很有用，它为我们进“公共小厨房”提供了方便。

行了，那么我们局部变量什么时候结束呢？这就是我们变量的生存周期的问题。我们文件级变量虽然不常用（我们也称为：外部变量）但它的生存周期几乎与我们软件的生存周期一样，我们软件不结束，变量也一直存在着。

而外部变量则在文件内部，所有函数中都有效，如果被局部变量覆盖的话，可以用全局变量运算符（::）给找回来

而局部变量则由定义处存在，到相应的花括号结束处结束。我们再来看一个例子吧：

```
#include &lt;iostream.h&gt;
int j=10;
int main()
{
   int j=20;
   cout&lt;&lt;"j="&lt;&lt;j&lt;&lt;endl;

   cout&lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl;

   for(int i=0;i&lt;1;i++)
   {
       int j=30;
       cout&lt;&lt;"在 for 语句中：j="&lt;&lt;j&lt;&lt;endl;
       cout&lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl;
       do
       {
           cout&lt;&lt;"在 while 语句中：j="&lt;&lt;j&lt;&lt;endl
               &lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl
               &lt;&lt;"注：我们在 do~while 语句中可没有定义 j 噢
"&lt;&lt;endl;
           int j=40;
           cout&lt;&lt;"定义了 j 以后,j="&lt;&lt;j&lt;&lt;endl
               &lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl;
       }
       while(0);
       cout&lt;&lt;"出了 while 语句，又时入了 for 循
环:j="&lt;&lt;j&lt;&lt;endl
           &lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl;
   }
   cout&lt;&lt;"出了 for 循环，进入了主程序：j="&lt;&lt;j&lt;&lt;endl
       &lt;&lt;"::j="&lt;&lt;::j&lt;&lt;endl;
   return 0;
}
```

大家把程序做进计算机,仔细体会着局部变量的生存周围与 :: 运算符的使用。看看我们在程序中定义了多少次 j。
在一个在方可以将 j 定义两次吗？我们在程序中将 j 定义了几次？为什么可以定义这么多次？仔细地去体会，我们就会知道，变量定义其实在每一个花括号中都可以不考虑其他地方是否有相应的定义，其中的定义与外面的定义没有关系。

当然，如果大家运行程序了，大家会发现，在内层大括号中是没有办法使用上一级大括号中定义的同名变量的，所以我们定义变量也不要太滥，否则我们真的想用某个变量可能就会陷入了很尴尬的境地，我们定义了变量，却没办法使用他们，这是我们所有愿意看到的

行了，关于公用厨房的函数部分，我们就先讲到这里，我们知道我们怎么定义变量，定义方法有什么用就行了，其实说实话，不要说全局变量这种数据的定义了，当然定义一个全局变量可能有些程序可能变得很简洁例如我们在所有的函数中都想改变某个变量，或作为某个程序的同步变量，都可以使用全局变量，但人们在使用在越来越发现这种程序的数据使用依赖性很强，使得程序的通用性得不到保证，往往一个应用就要做一个软件，这使得我们的编程有很大的，很多的重复劳动，很不划算，所以有人构思着将程序与数据给分开，将数据从程序中分离开来，而程序只管程序，而数据存在一个单独的文件中，供程序使用，人们后来称这种系统为：数据库系统，现在数据库使用得十分广泛，甚至已经深入到了生活的方方面面。我们说这什么意思呢？还是那句话，没有必要，少用全局变量，这一样是我们上面所说的，我们介绍，但不建议过多地使用。

与邻居共用小厨房，难免锅碗碰瓢盆，所以说，我们还是自己有自己的厨房为更好。

再有一个就不是厨房了，而是与邻居共用的仓库了，我们下节再讲。

这个技术应用的多了，可以是铺天盖地，我们只要用电脑，几乎不可能不用这个技术。

那么什么叫排序呢？就是将一串字符或数字排某个顺序排列。

字符串与数据有什么不同呢，大家看：

12
2

可能大家都知道 12>2,

而下面两个呢？
"12"
"1"
可能结果会出乎你的意料："12"<"2"
我们为什么要说这个呢？因为如果我们看电脑中的排序，都是按照后一种方式排序的，可能大家如果不知道的话就会纳闷，我的一个文件夹中，明明我是按文件名顺序排的啊，可为什么还会出现下面的情况：
1.txt
11.txt
112.txt
12.txt
...
19.txt

2.txt
24.txt
3.txt
...
我们举个例子啊，原来啊，我们在文件名中的文件名不会被看成一个数字，而被看成一个字符串，字符串只比较最高位的，如果最高位一样才比较第二位，最高位不一样就按最高位来排序，所以会出现："12"<"2"的情况。

行了，我们这一个阅读材料不与大家讨论字符串，我们就以数字为对象来研究研究排序。

大家还知道两个函数吗？
void swap(int &a,int &b);

什么叫排序呢？就是要在这个过程中不断地看两个数的大小，然后判断需要不需要交换，从而进行交换的行为。当然我们只是为大家说一个思想，其实排序的办法有很多，我们这讲只讲一种排序，告诉大家，用计算机来做这些事情很方便就行了。我们先任意准备五个数据吧：

123,34,56,21,77

能说明问题就行，呵呵，我们要从小到大排列，那么我们就将第一个数字和后面所有的数字比较，找到最小的数字，从而进行交换就行了：

看，我们先比较第一个数：
[123,34]我们发现 34 较小，所以我们交换他们：
34,123,　56,21,77
然后再与第三个数字比较
[34,56]
不需要交换
[34,21]
交换
21,123,56,34,  77
然后再比较:
[21,77]
不需要交换。

看，怎么样？第一个数字变成最小的数字了，就这样，再拿第二个数字来比较，一直到最后，我们的顺序就确定了。

我们再介绍一个更简单的，我们假如我们有十个数字，写在十个卡片上，然后我们做一个小盒子，先放入第一个数字，然后每个数字与这个数字比较，比较到最后一个数字，盒子里就只剩我们最小的数字了，然后将最小的数字与第一个数字交换（如果不同的话），然后再从第二个开始与盒子中比较，再找出最小的，放

在第二个中，依次类推，我们就可以做出以下的程序：

```
#include &lt;iostream.h&gt;

void swap(int &a,int &b)
{
   int t;
   t=a;
   a=b;
   b=t;
}
int main()
{
   int a[10]={122,345,343,13,65,23,456,533,742,43};//先准备十个数，一个数组；
   int b,i,j;//定义一个小盒子
   cout&lt;&lt;"排序之前，数组的顺序为：";
   for(i=0;i&lt;10;i++)
        cout&lt;&lt;a&lt;&lt;",";
   for(i=0;i&lt;10;i++)
   {
        b=i;
        for(j=i;j&lt;10;j++)
        {
             if(a&gt;=a[j])
             {
                  b=j;
             }
        }
        swap(a,a);
   }
   cout&lt;&lt;endl&lt;&lt;"排序之后，数组的顺序为：";
   for(i=0;i&lt;10;i++)
        cout&lt;&lt;a&lt;&lt;",";

   cout&lt;&lt;endl;
   return 0;
}
```

怎么样？大家将程序做进电脑看看，像这种排序的程序早就被变成了一种简单的函数，人们只需要调用就行了，其实告诉你，用什么函数，牛刀也不知道因为牛刀没用过，如果想用，牛刀宁可自己去做一个，这种程序属于小程序啊。

我在没有定义 swap 函数时，我输入了函数，还发现库中的这个函数呢。
我找了一下：竟然被做成了模块，什么叫模块？其实那是一种特殊的函数，如果有机会，牛刀再详细与大家说。

其实就是这样，我们做程序时，就必须对程序中的各种情况都有所了解，而我们正真做程序时，可能好多的事情就不需要我们自己做了只要用别人做好的就行了，这就是我们的编程，学习时我们必须学习各种知识，而做程序时，我们就可以使用别人已经做好的东西，其实原因很简单，我们都已经掌握了，并且谁做的程序都是一个样，那么我们为何不能在将这些东西做成一个模块，需要时，我们只需要像语句一样，用一下就行了。

牛刀老师 您说的这句话好像和我所运行的结果不太一样啊呵呵
在内层大括号中是没有办法使用上一级大括号中定义的同名变量的
程序：

```
[cpp]#include <iostream.h>
int j=10;
int main()
{
   int j=20;
   cout<<"j="<<j<<endl;

   cout<<"::j="<<::j<<endl;

   for(int i=0;i<1;i++)
   {
       int j=30;
       cout<<"在 for 语句中：j="<<j<<endl;
       cout<<"::j="<<::j<<endl;
       do
       {
          cout<<"在 while 语句中：j="<<j<<endl
              <<"::j="<<::j<<endl
              <<"注：我们在 do~while 语句中可没有定义 j 噢"<<endl;
          int j=40;
          cout<<"定义了 j 以后,j="<<j<<endl
              <<"::j="<<::j<<endl;
       }
       while(0);
       cout<<"出了 while 语句，又时入了 for 循环:j="<<j<<endl
          <<"::j="<<::j<<endl;
   }
   cout<<"出了 for 循环，进入了主程序：j="<<j<<endl
       <<"::j="<<::j<<endl;
   return 0;
}
```

运行后我得到的结果是：

j=20
::j=10
在 for 语句中：j=30
::j=10
在 while 语句中：j=30
::j=10
注：我们在 do~while 语句中可没有定义 j 噢
定义了 j 以后,j=40
::j=10
出了 while 语句，又时入了 for 循环:j=30
::j=10
出了 for 循环，进入了主程序：j=20
::j=10[/cpp]

在这个程序中 while 是在 for 里面的
所以就可以说 for 是 while 的上一级
但是从结果的第三句和第五句来看
while 里面所用的 j 的值是和 for 里的值是一样的
也就是说 在 for 里面定义的 j 现在在 while 里面还可以用

也就是说在上一级大括号中所定义的变量还是可以直接被使用到下一级的

这是怎么回事啊？

同名，什么叫同名，首先是有了相同的名称，才叫同名是也不是？

在 while 中，我们先并没有定义 j ，所以在没有定义 j 的时候，就是上面刚进入 while 的时候，因为在 while 中，并没有 j 变量，所以不存在“同名”一说，那么也就是说，在没有在本层括号之内定义一个变量的时候，我们的外层定义变量对括号是穿透的，也就是说，这时候，我们引用的变量就是外层变量，没有歧义。

这就是第五句的 j=30; 的情况。

当然，一旦定义了一个 j 后，就出现了第八行的 j=40; ，而我们再也拿不到 for 中的 j 了，也正因为这一点，我们上面的结论：
在内层大括号中是没有办法使用上一级大括号中定义的同名变量的
还是成立的。
当然，一旦定义了后，就对外层括号的中的变量形成了同名，于是进行了覆盖。

**牛刀教学〇一九：共用大仓库**

牛刀无奈，可能大家曾经说过：牛刀文章看不懂，我还是要顶。
这下好了，牛刀的帖子无人问津了，呵呵。
虽然你看不懂，但我还是要发（牛刀可和你们大伙儿犟上了）。

其实你不要说，发贴子，大家看不看倒不要紧，主要是牛刀水平蒸蒸日上，真的,做一个教程绝对对别人对自己都有好处,所以我想有一些朋友不要气妥啊，我发现　燃灯古佛　不见了，哎～～～。

还算牛刀脸皮厚，竟然无所谓，呵呵。是啊，做什么事都要脸皮打得厚一点，不然说真的很容易半途而废。

这一讲，我们要讲到新的内容了，其实如果你看过以前的文章，这个内容也不新，就是一个好地方－－公用大仓库。什么意思？牛刀总是瞎造一些词，但还是自觉自美。

地址啊，大家还记得吗？我们电脑内存的存取是需要通过地址来进行的，内存是共用的，所以我们叫她“共用的　大仓库”啊。

其实这个仓库，我们可以隔成一间间小屋子，既然是共用的嘛，那么，我们总要知道哪儿是门，哪儿有窗是不？这就涉及到了我们的地址运算符：&和取数据运算符：*

我们知道了，既然（*）是取一个地址对应的数据，那们由它定义的那个变量我们就叫做：地址变量：也就做指针：如

```
int *p;
```

那么这个 p 是一个地址变量。
哎，可惜啊可惜，指针只准我们赋一种值，那就是空值（NULL），其实就是 0，用一个新的字符代替了吧，但其他的值不能赋了，看例子：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=NULL;
}
```

上面的程序编译通过，但如果你给她赋其他地址，那不报错才怪呢？其实 NULL 是一个零地址，但在指针中，零地址就表示空，即什么也没有，固定了的，大家知道啊，地址不能是负的啊，所以说空就只能用零来表示喽。

但如果有其他的赋值就错（理论上可以赋，但编译器总是报错你有什么办法？）那么我们如何找到其他的地址呢？用自加或自减运算赋来做吧：（当然，可以将一个整型值的地址赋给她还是可以的，也就是说，她接受的必须是一个地址，而

不能是一个具体的数值：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   int i=20;
   p=&i;    //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
   cout&lt;&lt;p&lt;&lt;endl;    //当然我们可以输出 p 以来查看 a 的地址
   cout&lt;&lt;*p&lt;&lt;endl;    //*这个符号为求地址对应的数值，所以它就应该是 a 的值，那么我们看他们的结果。
}
```

0x0012FF78
20
大家看，第一行地址。而第二行则为这个地址上存放的值，不就是 i 的值（20）吗？
当然，我们可以看看 2 0 前面是一个什么数据：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   int i=20;
   p=&i;    //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
   p--;
   cout&lt;&lt;p&lt;&lt;endl;    //当然我们可以输出 p 以来查看 a 的地址
   cout&lt;&lt;*p&lt;&lt;endl;    //*这个符号为求地址对应的数值，所以它就应该是 a 的值，那么我们看他们的结果。
}
```

我的计算机中为：
0x0012FF74
-858993460
你瞧，是一个负数，但我们看地址啊，我们明明是“--p”啊，为什么地址少了四个呢？原来啊，指针变量的自加自减是以它所指向的类型来移动相应多个指针的，因为 int 是四个字节型，所以“--p”向前移动了四个字节，当然如果是“++p”那么指针就会向后移动四个字节，这就是说我们指针每次都可以指向一个正确的数值，而这种指向，指针是自动做的，而不需要我们考虑指针的数据类型。这下大家知道为什么我们要给指针定义一个数据类型了吧？

当然，我们可以看看后面十个位置的值是什么：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
```

```
    int i=20;
    p=&i;     //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
    for(int k=0;k&lt;10;k++)
    {
        cout&lt;&lt;p&lt;&lt;"  "&lt;&lt;*p&lt;&lt;endl;
        ++p;
    }
}
```

我电脑中的结果为：

```
0x0012FF78  20
0x0012FF7C  1245052
0x0012FF80  1245120
0x0012FF84  4208281
0x0012FF88  1
0x0012FF8C  4394640
0x0012FF90  4394448
0x0012FF94  34603777
0x0012FF98  34603536
0x0012FF9C  2147348480
```

怎么样？这样一来，内存中的数据不就又成了我们囊中之物了？

当然我们可以再来看看以前的程序，再看看指针能发现我们定义的数据区的什么规律？

```
#include &lt;iostream.h&gt;
void main()
{
    int *h;
    int i=20;
    int j=30;
    int k=40;
    int l=50;
    int m=60;
    int n=70;
    int o=80;
    int p=90;
    int q=100;
    int r=110;
    h=&r;     //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
    for(int t=0;t&lt;10;t++)
    {
        cout&lt;&lt;h&lt;&lt;"  "&lt;&lt;*h&lt;&lt;endl;
        ++h;
    }
```

```
}
```
大家将程序输进电脑看看，结果会出现什么情况，明白数据在内存中的存放方式：
```
#include &lt;iostream.h&gt;
void main()
{
   int *h;
   int i=20;
   int j=30;
   int k=40;
   int l=50;
   int m=60;
   int n=70;
   int o=80;
   int p=90;
   int q=100;
   int r=110;
   h=&i;    //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
   for(int t=0;t&lt;10;t++)
   {
       cout&lt;&lt;h&lt;&lt;"  "&lt;&lt;*h&lt;&lt;endl;
       --h;
   }
}
```
再将上面程序做进电脑，巩固学习内容，看看与自己的想像是不是一样？
```
#include &lt;iostream.h&gt;
void test()
{    int i=20;
   int j=30;
   int k=40;
   int l=50;

}
void main()
{
   int *h;
   int m=60;
   int n=70;
   int o=80;
   int p=90;
   int q=100;
   int r=110;
   h=&r;    //看，&是取地址符，所以此句的函义为，指针 p 指向变量 a 的地址。
```

```
    for(int t=0;t&lt;10;t++)
    {
        cout&lt;&lt;h&lt;&lt;"  "&lt;&lt;*h&lt;&lt;endl;
        ++h;
    }
}
```

又出现什么情况了？大家能不能想办法将函数中的地址也找出来，看看他们之间有什么关系，函数是怎么调用的，调用的入口在哪儿？（注：我们上面的程序中的数据定义没有按照一种很规矩的方式定义，大家不要用这种方式在程序中做，但做一些测试程序（如上面的程序）则无所谓。

那么我们再定义一个数组再瞧瞧，和上面的什么不同：

```
#include &lt;iostream.h&gt;
void main()
{
    int a[100];
    for(int i=0;i&lt;100;i++)
        a=i*10;
    int *p;
    p=a;
    for(i=0;i&lt;100;i++)
    {
        cout&lt;&lt;p&lt;&lt;"  "&lt;&lt;*p&lt;&lt;endl;
        ++p;
    }
}
```

大家看，我们没有用：p=&a;

大家使用 p=&a 看看，报错的啊，这是为什么呢？

原来啊，数组变量中的数组名竟然就是一个指针，只不过它是一个固定的指针，它只指向 a[0]，不信我们可以将 a[0]的地址再打印出来看看结果，是不是与数组的指针的位置一样？

也就是说，数组是一个地址不能变动的指针，那么说来数组后面的[]就是指针运算符了，其实这是真的这个中括号“[]”就是一种指针运算，所以说我们也可以这样做：

```
#include &lt;iostream.h&gt;
void main()
{
    int a[10];
    int i;
    for(i=0;i&lt;10;i++)
        a=i;
    int *p;
```

```
    p=a;
    for(i=0;i&lt;10;i++)
        cout&lt;&lt;p&lt;&lt;endl;
}
```

怎么样？还真能用啊，但就是有一点是：++p 是正确的，而++a 是错误的。不信你去电脑上做做看。

那么就存在着另一个问题了，二维数组与多维数组是什么情况呢？大家先在电脑上自己做一些程序试试，看看二维数组的地址顺序，二维数组的指针表示，指针的自加运算，等等，先留给大家思考

我们已经学到最关键的地方了，所以我们的每一讲的速度会明显放慢，供大家自己慢慢理解，牛刀不布置作业了，如果你真的会去上机的话，牛刀布置不布置作业是一样的。

数据类型，到现在为止，我们学过了很多，什么整型，浮点型，字符型……

但是，如果我们看到了一串定义，是不是有些头疼？

unsigned int

这一串数字，也够你输入一小阵子的啊，呵呵，习惯就好，但是……

懒是人的通病，如果不是人的通病的话，也没有我们这一讲，有人就给了我们自己定义一个类型的机会，也就是说，我们以前的类型，可以换个名子，如：

typedef unsigned int UINT;

好了，以后就可以用 UINT 去定义一些无符号整型的类型了，那么我们再看它的形式：
typedef 数据类型　自定义数据类型;

你将前面的 typedef 拿掉，看是不是很像定义一个变量？其实就是这样，有了前面那个标识符，我们后面就是定义了一个标识符了。这样做有好处啊：

```
#include &lt;iostream.h&gt;
void main()
{
   typedef unsigned int HEIGHT;
   typedef unsigned int WIDTH;
   HEIGHT height;
   WIDTH width;
   height=10;
   width=15;
   cout&lt;&lt;height*width&lt;&lt;endl;
```

}

有这种必要吗？牛刀故弄玄虚。

有这种必要啊，大家可知道，我们的定义往往是与程序分开的，定义往往一些头文件中，如果不为那些定义符做些手脚，我们怎么能够在做一个大型的程序能看到一些东西，立刻明白一些他的意义的呢？举个例子吧：

大家按下面的过程做一个程序：
文件/新建/[工程]/Win32Appllication
在工程中填入“WinTest”
按“确定”、“完成”、“确定”。
呵呵，牛刀帮大家做Ｗｉｎｄｏｗｓ程序了。

再按下面步骤进行：
文件/新建.../[文件]/"C++ Source File"
在“文件”下面输入框中填入“WinTest”，然后按“确定”
大家注意了，以后我们这叫做：建立一个窗体程序“WinTest”（注：牛刀叫法）

行了，我们输入下面一段的内容：

```
#include &lt;windows.h&gt;

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{
   return 0;
}
```

当然喽，这一个程序正如我们前面的程序：

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

一样，什么事情都不做，所以你编译的话，什么东西都不出来（可能有人不服，上面不是出来一个黑屏，并且上面有一串字：“Press any key to continue”吗？，呵呵，那是为了让你看的，其实就是什么功能都没有）。

而为什么 WinMain 函数什么都不出来呢？因为用不着，如果你想出来一样东西的话，请输入下面的一段程序：

```
#include &lt;windows.h&gt;

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{
   MessageBox(NULL,"我跟牛刀学编程","牛刀对话框",MB_OK);
   return 0;
}
```

这一段程序就能出结果了，输出一个对话框，当然，功能很简单，我们再来分析

程序吧，主要分析定义：

我们先看 windows.h，这个头文件是一个基本的 Windows 程序的头文件，而下面一行
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
就是我们在Ｗｉｎｄｏｗｓ编程中的入口函数。
我们知道，返回类型为 int 类型，所以我们在程序中必须有“return 0”语句，功能和 main 中的一样。
而后面 WINAPI 表示这个函数是一个 Win 下的 API 函数关于ＡＰＩ函数，我们以后详细地加以讨论。
而大家看我们后面的定义就好玩了：
HINSTANCE
LPSTR
...
什么意思呢？现在告诉你你可能真的不知道，但你可以将鼠标放在上面，按右键，选择：Go to Definition Of HINSTANCE
于是，可能我们看到：
DECLARE_HANDLE(HINSTANCE);
我们看不懂，没关系，这我们以后再看，再看下一个：
LPSTR，他会是怎么的情况？
typedef CHAR *LPSTR, *PSTR;
这个我们就似乎是懂了，我们再看看 CHAR 为什么不是 char 呢？
typedef char CHAR;
噢，原来 CHAR 与 char 是一样的，那么上面的 LPSTR 就好理解了，就是一个字符指针呗。
其实就是这样，基于 HINSTANCE 吧，可能有些看不懂，
其实它是一个宏，就表示我们的 HINSTANCE 是一个无符号的整数，以后再说吧
再来看一个函数：
LRESULT CALLBACK WndProc( HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
　　大家看，这里又有了一些新的符号：
LRESULT
HWND
UINT
WPARAM
LPARAM
好象只有 UINT 在哪里见过，其他都没有见过。
没关系，只要我们以后学到那儿，你会发现这种定义数据类型的方式特别地方便，
于是乎，人们又有一些不成文的规定，大家都习惯了的，如：
LP 代表是指针类型。
STR 代表是字符串类型。
INT 代表是整型。
U 代表是无符号型。

……

于是，有了这些规定后我们定义东西就更简单了，看
UINT 不就是无符号整型吗？
那么说：
UCHAR 就应该是无符号字符型喽
LPSTR 就是指针型字符串了吧

而大家看前面有 H 的，那就更有用了，那表示这个类型是一种句柄，而在Ｗｉｎｄｏｗｓ中，句柄的概念是铺天盖地，到时间再给大家讲吧。

也就是说，一个 typedef 给我们带来的是很多很多的方便，使得我们编程过程更加理性，有了一些人为的规定，我们读别人的程序，别人读我们的程序都将会变得方便，一看就懂，一看就明白，这也是我们所说的编程风格的另一个方面。

随着我们以后数据类型的增加，大家还会看到其他形式的数据类型，正是由于我们可以自己来定义这些，所以我们的编程的过程会变得相当灵活，相当方便，有时甚至使得我们的编程过程是一个享受，有人还提出一些新的编程思想，以后编程可能像搭积木一样，把东西拿来搭一搭，就是一个相当不错的程序了。
当然我们期待着那一天的到来，但跟着牛刀学到这儿，你会发现，那些程序做的的确是很优秀了。

对了，还有一个函数大家没弄明白吧？
MessageBox 功能就是弹出一个小窗口，它的句柄为一个参数，我们设为ＮＵＬＬ因为我们程序中没有设备描述表句柄，而第二个是显示内容，第三个是标题，第四个是表明我们要显示一个什么样的窗口，详细内容我们以后再说。

牛刀老师 提两个问题

问题一

**引用:**

大家能不能想办法将函数中的地址也找出来，看看他们之间有什么关系，函数是怎么调用的，调用的入口在哪儿
这句话没看懂 呵呵 不知道牛刀老师让我们找什么

问题二
记得牛刀老师原来说过数组的存放方式

[cpp]int a[5]={0,1,2,3,4};[/cpp]
如果这样定义数组的话
那么数据在栈中的存放方式是这样的

0　　栈顶
1
2
3
4　　栈底

也就是说 先放入 4 接着 3 再是 2 然后 1 最后 0

但是如果我的数组是这样定义的呢
for(i=0;i<5;i++)
a=i;
那这样就是 a[0]先得到数据 a[4]最后得到数据
压栈的话也是先压 a[0]最后是 a[4]啊

这样 a[0]最先得到数据 数据为 0
先压入栈 0 进入栈底
然后 i 变成 1
那么 a[1]=1 压栈
.....
最后 a[4]
那么 4 就跑到了栈顶 成了这样
4　　栈顶
3
2
1
0　　栈底

这就反过来了 怎么回事啊
问题一：涉及破解方面内容了，只可意会，不可言传，嘻嘻，我今天将上面的 test 程序又输入电脑运行一下，牛刀也发现我自己也弄不明白到底哪儿是函数入口了，呵呵。

问题二：

关于这个问题，我们可以定义一些程序去做一些测试：

```
#include <iostream.h>
void main()
{
        int a=10;
        int b=20;
        int c=30;
        int d=40;
        int f[]={50,51,52,53,54,55};
```

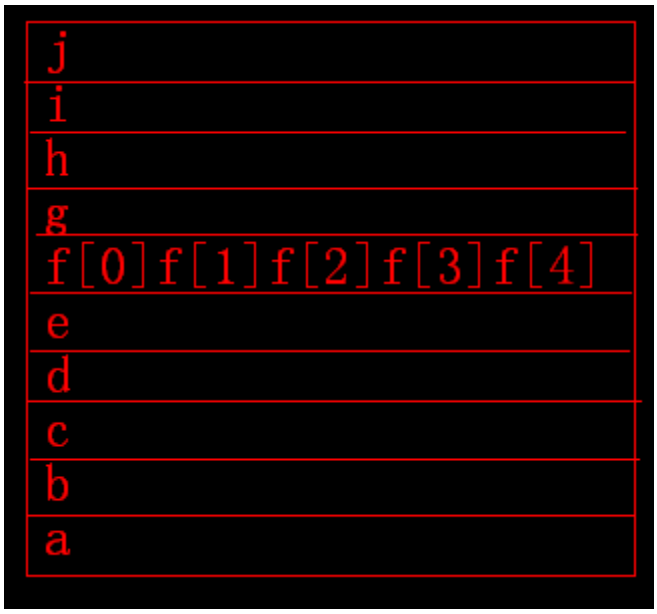
```
        int g=60;
        int h=70;
        int i=80;
        int j=90;
        int *x=&j;

        for(int y=0;y<14;y++)
        {
                cout<<"x="<<x<<" "<<"*x="<<*x<<endl;
                x++;
        }
}
```

看结果：

```
x=0x0012FF48 *x=90
x=0x0012FF4C *x=80
x=0x0012FF50 *x=70
x=0x0012FF54 *x=60
x=0x0012FF58 *x=50
x=0x0012FF5C *x=51
x=0x0012FF60 *x=52
x=0x0012FF64 *x=53
x=0x0012FF68 *x=54
x=0x0012FF6C *x=55
x=0x0012FF70 *x=40
x=0x0012FF74 *x=30
x=0x0012FF78 *x=20
x=0x0012FF7C *x=10
```

你发现什么了？是不是数组部分是顺序存放，而其他部分是逆序存放的？

原来啊，数组在定义之时，就一下子为其分配了一段空间，而这一段空间是顺序存放的，而不是逆序存放的，也就是说，我们的数据其实是分配在堆中的，而堆的每次分配是从高字节向低字节分配的，而一旦定义一个数组，那么系统就直接为这整个数组分配一块空间，而在这块空间中，是顺序存放的：



如上图，其实做一个程序试试，就什么都明白了。

问题一：
这个这个。。只是不知道牛刀老师让找什么东西
函数地址是不是指子函数的地址？
实在有些头晕 呵呵

问题二：
原来在定义的时候
数组的顺序就已经给安排好了
往里面赋数值的顺序不影响数组在内存中的位置

搞明白了

谢谢牛刀老师 嘿嘿

牛刀老师 追加一个问题
看到课外阅读里面有一个程序
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
return 0;
}
我把它改了一下 把那个字符指针改成了整形的 然后指向 nCmdShow
[cpp]#include <windows.h>
#include <iostream.h>

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPINT

```
lpCmdLine,int nCmdShow)
{
        nCmdShow=1;
    lpCmdLine=&nCmdShow;
        cout<<"变量的值="<<*lpCmdLine<<endl;
    cout<<"变量的地址="<<lpCmdLine<<endl;
        return 0;
}
```

这时候发现报错了

```
--------------------Configuration:       WinTest       -       Win32
Debug--------------------
Compiling...
WinTest.cpp
F:\我的程序\WinTest\WinTest.cpp(5) : error C2731: 'WinMain' : function
cannot be overloaded
        F:\我的程序\WinTest\WinTest.cpp(4) : see declaration of
'WinMain'
Error executing cl.exe.

WinTest.obj - 1 error(s), 0 warning(s)
```

这是怎么回事啊

难道整形指针不能用？

错误的原因我看不懂 还请牛刀老师指教

嘻嘻，还没学到哪儿了，就想改人家的入口函数啦，呵呵。看牛刀教程的人脑袋就是比看其他书籍的朋友要先进，呵呵。

原来啊，我们的入口函数，是由操作系统调用的，而操作系统调用的函数，是按函数指针的格式调用的，也就是说，系统是用一个指针来调用我们的函数的，而像这种入口函数的函数名系统为我们定义好了，一定要叫 WinMain ，而参数的名称可以改变，但类型与数据的个数一定不能改变，这就是原因。

也就是说，系统找不到她自己需要调用的函数了，也就是说，没有入口函数了，那么当然要报错啦，就是这个原因，看程序：

```
max(int a,int b,int c)
{
...
}
void main()
{
   max(2,3);
}
```

你看，我们定义了一个三个参数的 max 函数，而你要调用一个两个参数的函数，那么系统当然找不到相应的函数，就必定要报错喽，就是这个原因。

你将人家函数中的参数类型给换了，人家操作系统当然就找不到要调用的函数了，于是就报错，这也是人之常情啊。

总之，我们对于这些系统调用的函数，有一定的原则，如果你看到函数指针的那一章，有些问题就会明白很多，但其中的参数名称是可以改变的，那无所谓，因为我们的函数原型就可以这样定义：

```
main(int,int,int);
```

当然，有些具体的问题，自己慢慢去琢磨，有些事情是不能想当然的，WinMain函数中的四个参数，都是有各自的用途的，虽然现在好多参数都没用了，但既然有了这些参数，就会有其存在的道理。

提问:

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

void main()
{
   int ch[3];
   ch[0]=1;
   ch[1]=2;
           ch[2]=3;
           ch[3]=4;
   for(int i=0;i<3;i++)
   cout<<ch<<endl;
}
```

这个程序 运行后会说 Unhandled exception in text.exe ：0xc0000005 Access Violation?

这是什么意思呢？经过试验发现 把 ch[3]=4 这句注释掉后 程序就正常了？可是为什么不能定义ch【3】呢？

请牛老师 解惑

呵呵，显然是数组越界了啊，看样你的编译器检查得还挺严格，不知用的什么编译器。

我们定义一个数组后，下标最大值总比定义值要小一个，否则越界，这也是新手常会犯的毛病。

当然喽，上面我看你老喜欢——顶我的贴子，当然，有问题欢迎提出来，顶就没有必要了，你没看到这个子版块的贴子全是我的吗？呵呵。

为牛刀开专版，还得感谢管理员，一般没有人在牛刀这个板块中灌水，再说管理员也不让啊，呵呵，放心好了，只看不支声，有问题，当然欢迎交流。（其实不是说不能顶，一般有新回贴牛刀就会认为有人有问题了，就会进来看，帮助别人解决解决问题，如果看到的全是顶啊顶的，那多没意思，呵呵）

## 牛刀教学〇二〇：新的与旧的

牛刀的脸皮好厚哟，都第廿讲啦，呵呵。

人们常说，新的不来，旧的不去，所以我们这一讲就讲一讲这个新（new）与旧。

当然我们前面课程中说过，一个指针只能指向一个存在的数，那么是不是我们一定要定义这个数呢？呵呵，说起来还真有些复杂，理论上说是必须定义的，然而，我们也可以通过一些技术手段，来自己做一个（做一个新的啊，呵呵，用人家旧的有什么好？）

看程序：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=new int;
   *p=2;
   cout&lt;&lt;*p&lt;&lt;endl;
}
```

看：new 给了我们新找一个内存，当然这个内存是系统给我们分好的，其实还属于那一个公用的大仓库，只不过那地址不是我们指过去的，而是专门为这个指针新分配的了，那么它与下面的有什么不同呢？

```
int i=2;
p=&i;
```

如果现在说起来，可能真的没什么不同，不同就是一个有个变量 i，而另一个只能通过指针指过去呗，当然，如果你再 new 一个，会出现什么问题？

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=new int;
   *p=2;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   p=new int;
   *p=3;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
}
```

我的电脑中显示出来的是：

```
0x00431E30 2
0x00431BD0 3
```

大家想，如果我们还想指回去，指回 2 去，指不回去了是不是？
当然你可能会说(E30-BD0)/4 不就等于它的指针偏移量吗？加回去不就行了吗？

真的有这种规律吗？我们再做一个：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=new int;
   *p=2;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   p=new int;
   *p=3;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   p=new int;
   *p=4;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
}
```

我的结果是：
0x00431E30 2
0x00431BD0 3
0x00431BA0 4
大家看还能加加减减吗？
原来啊，new 新分出来的区间是随机分配的，根本没有规律，所以说你这次可以能够这样用了，可程序运行时间或地点或一移到其他的机上，你的程序根本没有通用性。

于是，原来的指针变量就像断了线的风筝，你根本控制不了它，于是，这些指针被称之为“野指针”，一个程序中如果出现了大量的野指针，肯定会浪费空间吧？会通过野指针泄密吗？牛刀不知道，牛刀反正知道野指针的存在肯定不好，或许什么时候一旦别的什么程序发现你的程序中有这种野指针，可能能够使用吧，但怪怪的是，你自己的程序却根本没办法使用。所以我们当一个指针变量不需要用时，一定要使用 delete 语句，将指针变量给删除，内存空间给收回(注：这一点 Java 语言做得就很好，Java 指针使用过后可以不用管她，你如果在没有空间时，或程序退出时，有系统统一地进行处理，叫做垃圾处理机制〔牛刀语〕，而ＶＣ却没有这种机制，那么是不是程序运行完后，这些指针还在内存中呢？或者操作系统会不会清理呢？是不是一定要机器关了这些指针才没有呢？一系列问题，说实话，验证都没办法做程序验证，只有朋友们如果在其他资料在看完了长这种见识了。如果一旦发现其中的漏洞，那就太恐怖了，所以我们要回收空间)。
看程序：

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
```

```
   p=new int;
   *p=2;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   delete p;
   p=new int;
   *p=3;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   delete p;
   p=new int;
   *p=4;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
}
```

大家看：

```
delete p;
```

在程序中似乎没有什么着用，没有她程序也运行，有她程序也不报错，那么如此说来，我们怎么知道我们的 p 值被删了呢？

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=new int;
   *p=2;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
   delete p;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
}
```

```
0x00431E30 2
0x00431E30 -572662307
```

看，我们的数据 2 没有了，代之的是一个负数，为什么是这个数呢？你问牛刀，牛刀还真的不知道，有空你自己多删几个或者用十六进制打印出来看看能不能看看它有什么规律？

当然了，有一个缺点，我们发现虽然 p 被删了，但指针还在，还有值，哇，又是一个漏洞，（注：牛刀可不是存心想找漏洞的，可没办法，new 一个，就出了一个漏洞你叫我有啥办法，

那么删去不是没有什么用？我们再给它赋个值看看出现什么情况？

```
#include &lt;iostream.h&gt;
void main()
{
   int *p;
   p=new int;
   *p=2;
   cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
```

```
    delete p;
    cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
    *p=5;
    cout&lt;&lt;p&lt;&lt;" "&lt;&lt;*p&lt;&lt;endl;
}
```

呵呵，竟然还真能再赋值，没办法，可能做ＶＣ的人不会想到牛刀敢这样做，所以说我们的指针变量大家用完后，删完后，最好能够执行一下：

p=NULL;

将原来分配的那部分彻底掩盖，当然，可能牛刀也不会这样做的，何况我们以后再使用 p 的话，我们还会使 p 重新指向另外的地方的，所以说，我们的担心可能也是多余。

于是，delete 果真面临着一层神秘的面纱，于是，我们只有用书本上的一套规定，在 new 一个数据时，不用后，请使用 delete 删除它（当然牛刀也认为还是删除了的好，养成习惯，因为我们以后可能会用它来做链表，如果你没有不使用就删除的习惯的话，那么在链表中就可能出问题，而我们这里一个半个的数据定义，可能删不删真的无所谓。这些我们以后到链表那一节再与大家讲吧。

总之，想要新的，就用 new 来一个（之前则需要用 delete 删去旧的〔当然没有旧的就不必要删了〕），所以这就叫“旧的不去，新的不来”。

行了，关于新的与旧的，我们就先讲到这儿，我们再来讲一讲数组：（注：大家看我写程序又不由自主地用 void main 了是不是？没办法，这个用起来舒服啊，其实我说过，怎么舒服怎么来，但……我下面还是改过来吧。）

```
#include &lt;iostream.h&gt;
int main()
{
    int a[4][4];
    int i,j;
    for(i=0;i&lt;4;i++)
        for(j=0;j&lt;4;j++)
            a[j]=i*10+j;
    int *p;
    p=a;
    for(i=0;i&lt;16;i++)
    {
        cout&lt;&lt;*p&lt;&lt;endl;
        p++;
    }
    return 0;

}
```

上面的程序做上电脑会报错的，为什么呢？大家思考，我们下面改一下再运行吧：

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   int *p;
   p=*a;
   for(i=0;i&lt;16;i++)
   {
       cout&lt;&lt;*p&lt;&lt;endl;
       p++;
   }
   return 0;

}
```

为什么会这样？因为我们曾经说过，数组中的[]相当于是指针的运算，也就是取地址，是不？所以说我们一个二维数组则被取了两次地址了，所以我们在定义一个指针时，就必须在数组名上使用一次取地址中的数据的符号，以表明 p 是一个整型数据。

再告诉大家一个事实，我们上面的的程序中，我们为 a,a[0],a[0][0]我们求一次大小看看：

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   cout&lt;&lt;"sizeof(a)="&lt;&lt;sizeof(a)&lt;&lt;endl
       &lt;&lt;"sizeof(a[0])="&lt;&lt;sizeof(a[0])&lt;&lt;endl
       &lt;&lt;"sizeof(a[0][0])="&lt;&lt;sizeof(a[0][0])&lt;&lt;endl;
       return 0;

}
```

我们程序运行的结果是什么呢？

```
sizeof(a)=64
sizeof(a[0])=16
```

sizeof(a[0][0])=4

在这里牛刀不要大家来分析了，牛刀为大家分析了，数组 a 是一个 64 字节的数据，也就是指向的是一个 6 4 字节的整体，而 a[0]是一个 1 6 字节的整体 a[0][0]才是一个 4 字节的整体，也就是说什么意思呢？就是说 a 是一个以 4 个 1 6 字节构成的数组而 a[0]是以 4 个 4 字节整型构成的数组，所以说以下的程序是正确的，但不一定结果是正确的。

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   int *p;
   p=a[1];
   for(i=0;i&lt;16;i++)
   {
       cout&lt;&lt;*p&lt;&lt;endl;
       ++p;
   }
       return 0;

}
```

下面结果，大家自己看看，自己去分析分析吧：

```
10
11
12
13
20
21
22
23
30
31
32
33
1245120
4208569
1
4394640
```

（我为什么每行每列要用这么特殊的数据呢？大家自己思考，想法自己再做一些其他的例子）

于是，我们知道，上面的：

p=*a;
与
p=a[0];
的效果是等同的。
那么我们再定义一个指针的数组与数组的指针看看有什么不同：

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   int *p[4];
   cout&lt;&lt;"sizeof(p)="&lt;&lt;sizeof(p)&lt;&lt;endl;
       return 0;

}
```

结果为：

```
sizeof(p)=16
```

看，也是１６，那么是不是它就能指向数组名了呢？

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   int *p[4];
   p=a;
       return 0;

}
```

结果我们发现编译通不过，于是，我们需要对每个指针进行赋值：注：我们这里有*p[0],*p[1],*p[2],*p[3]四个指针呢。

```
#include &lt;iostream.h&gt;
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
```

```
    int *p[4];
    p[0]=a[0];
    p[1]=a[1];
    p[2]=a[2];
    p[3]=a[3];



        return 0;

}
```

当然，我们这是为了方便，你可以用一个循环来处理这种问题。
那么就没有别的办法了吗？看：下面是数组的指针：

```
#include &lt;iostream.h&gt;
int main()
{
    int a[4][4];
    int i,j;
    for(i=0;i&lt;4;i++)
        for(j=0;j&lt;4;j++)
            a[j]=i*10+j;
    int (*p)[4];
    cout&lt;&lt;"sizeof(p)="&lt;&lt;sizeof(p)&lt;&lt;endl;
        return 0;

}
```

我们 p 的字节还是有四个字节，但我们一共有 4 个这种字节吧，不是 16 个吗？
所以：

```
#include &lt;iostream.h&gt;
int main()
{
    int a[4][4];
    int i,j;
    for(i=0;i&lt;4;i++)
        for(j=0;j&lt;4;j++)
            a[j]=i*10+j;
    int (*p)[4];
    p=a;

    return 0;
}
```

大家看还报错吗？这就叫做数组的指针，与上面指针的数组可是两回事。
我们可以用*p 来引用其中每个数据：

```
#include &lt;iostream.h&gt;
```

```
int main()
{
   int a[4][4];
   int i,j;
   for(i=0;i&lt;4;i++)
       for(j=0;j&lt;4;j++)
           a[j]=i*10+j;
   int (*p)[4];
   p=a;
   for(i=0;i&lt;4;i++)
   {
       cout&lt;&lt;*p&lt;&lt;endl;
   }
   return 0;
}
```

大家上机运行一下看看，打印出的结果，看看用数组指针打印出的结果成指针数组有什么不用，想想其用法，我们留给大家一点思考的空间。
如果我们想要用数组指针来打印出二维数组中的全部内容，我们应该怎么办呢？请大家思考。

大家再思考三维以及高维数组如何利用指针，我们下面再讲一讲一个与数组密切相关的，字符串的处理与函数中的数组

大家还知道我们前面的定义字符串的方法吧？

```
char ch[]="......";
```

是吧，那么既然我们知道，数组名不就是指针吗？那么我们字符串显然可以定义为：

```
char *ch="......";
```

能不能呢？我们做个实验：

```
#include &lt;iostream.h&gt;
int main()
{
   char *ch="我跟牛刀学指针";
   cout&lt;&lt;ch&lt;&lt;endl;
   return 0;
}
```

大家看，多方便。对了，大家记得我们以前的一些程序了吗？我们告诉大家一些东西需要以后学到的呢？

```
#include &lt;iostream.h&gt;
void ask(char *s)
{
   cout &lt;&lt; "我是"&lt;&lt;s&lt;&lt;endl;
}
void main()
{
```

```
    cout &lt;&lt; "大家好！"&lt;&lt;endl;
    ask("牛刀");
}
```

大家看：ask 函数中传递的不就是一个指针数组么？这种数组传递的好处是：不需要指定大小并且直接传递的是地址。

所以我们以后再看到一个函数中如果传递了一个数组过去，一定注意了，她不是值传递，而是地址传递，所以我们在函数中尽量不要对数组中的内容做变化，我们看一个例子：

```
#include &lt;iostream.h&gt;
void test(int *b)
{
    b[9]=100;
    b[8]=99;
    b[7]=98;
    b[6]=97;
    b[5]=95;
}
int  main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,0};
    test(a);
    for(int i=0;i&lt;10;i++)
        cout&lt;&lt;"a["&lt;&lt;i&lt;&lt;"]="&lt;&lt;a&lt;&lt;endl;

    return 0;
}
```

大家看，怎么样？所以以后做程序时千万注意不要对数组的数据进行更改，因为那不是值传递。

当然也可以改成下面的形式：

```
#include &lt;iostream.h&gt;
void test(int b[])
{
    b[9]=100;
    b[8]=99;
    b[7]=98;
    b[6]=97;
    b[5]=95;
}
int  main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,0};
    test(a);
    for(int i=0;i&lt;10;i++)
        cout&lt;&lt;"a["&lt;&lt;i&lt;&lt;"]="&lt;&lt;a&lt;&lt;endl;
```

```
   return 0;
}
```
但一定不能只写成：
```
void test(int b)
```
那就不是传递数组了，而仅仅是传递一个参数了，而且是值传递，那么我们上面的 test 函数中的语句则全部错误。
当然了，传递一个数组，通常都副带传进一处整型变量，说明数组的长度，经便函数对其进行统一的处理：如：
```
void test(int i,int a[])
```
当然，我们也可以做其他的事情，那是我们怎么想做什么样的程序了。
大家分析下面的定义，定义了什么他们有什么意义，我们怎么使用它，它是表示多维数组吗？如果是，怎么表示？如果不是，那么如何使用，试用电脑将自己的想法验证出来，以便以后编程过程中可以得心应手地使用：
```
int **p;
char ****q;
UINT ********t;
```
说实话，上面前面加好多好多这种符号的，我们就是告诉大家，可以这样做，但真的编程序时，最多加两个，不然你自己就被搞糊涂了，当然，计算机不会糊涂，你有本事做出相应的程序来，那算你狠，牛刀没那个本事。
看完了别人的类型，定义了别人类型的别名，下面我们试着自己也定义自己的类型看看，Ｃ＋＋给我们一个工具：enum
　　先来看个程序，我们用这种方式来定义一个我们自己的类型：
```
#include &lt;iostream.h&gt;
enum weekday{sun,mon,tue,wed,thu,fri,sat};
int  main()
{
   weekday myweek;
   myweek=sun;
   return 0;
}
```
大家看，我们定义了一个 weekday,他的数据为：sun,mon,tue...
　　那么，这些数据到底是什么呢？
```
#include &lt;iostream.h&gt;
enum weekday{sun,mon,tue,wed,thu,fri,sat};
int  main()
{
   weekday myweek;
   myweek=sun;

   cout&lt;&lt;sun&lt;&lt;endl
        &lt;&lt;mon&lt;&lt;endl
        &lt;&lt;tue&lt;&lt;endl
        &lt;&lt;wed&lt;&lt;endl
```

```
        &lt;&lt;thu&lt;&lt;endl
        &lt;&lt;fri&lt;&lt;endl
        &lt;&lt;sat&lt;&lt;endl;
    return 0;
}
```

原来如此，就是一串数字啊，其实事实上就是这样，当然我们还可以使其数据变化：

```
#include &lt;iostream.h&gt;
enum weekday{sun=5,mon,tue,wed,thu,fri,sat};
int  main()
{
    weekday myweek;
    myweek=sun;

    cout&lt;&lt;sun&lt;&lt;endl
        &lt;&lt;mon&lt;&lt;endl
        &lt;&lt;tue&lt;&lt;endl
        &lt;&lt;wed&lt;&lt;endl
        &lt;&lt;thu&lt;&lt;endl
        &lt;&lt;fri&lt;&lt;endl
        &lt;&lt;sat&lt;&lt;endl;
    return 0;
}
```

```
#include &lt;iostream.h&gt;
enum weekday{sun=5,mon=7,tue,wed,thu,fri,sat};
int  main()
{
    weekday myweek;
    myweek=sun;

    cout&lt;&lt;sun&lt;&lt;endl
        &lt;&lt;mon&lt;&lt;endl
        &lt;&lt;tue&lt;&lt;endl
        &lt;&lt;wed&lt;&lt;endl
        &lt;&lt;thu&lt;&lt;endl
        &lt;&lt;fri&lt;&lt;endl
        &lt;&lt;sat&lt;&lt;endl;
    return 0;
}
```

甚至其中有些东西还可以重复（当然可能有人会说不可以，但牛刀在电脑上试过，你不信去试呗。只不过一般不那么用罢了。

好了，我们看一下定义啊：
enum 枚举类型名
{
  枚举元素 1 [=整型常量 1 ],
  枚举元素 2 [=整型常量 2 ],
  ...
  枚举元素 n [=整型常量 n ]
};

就这样的定义很简单，像上面的定义我们可以在程序中判断，如

if(myweek==sun)
{
}
else
{
}
方便吧？例程就不为大家做了，其实说实话，这种定义用得比较少。

这一章实在深奥 看得我头快爆炸了
牛刀老师啊 那个 p=*a 实在是看不懂为什么这样写
a 是一个地址啊 *a 就是一个指针啊
p 也是一个地址啊
那 p=*a 就是一个指针赋给了一个地址可这不对啊
我的思维现在很混乱。。。还望牛刀老师指点

这是二重指针，我们曾经说过：
int a[2]; 其实就相当于定义一个指针，只不过这个指针是固定的，而
int a[3][4]; 则相当于定义了一个二重指针，相当于如下的定义：
int **a;
　　而 p 是这样的定义的：
int *p;
　　所以根据以上的规则来看：**a 是一个数据，而 *a 就是一个指针，而 **a 是一个指向指针的一个指针，所以说：
p 是一个指针
*a 也是一个指针
　　那么 p=*a; 不就理所当然地可以用了吗？
　　请注意一点： * 是取值符号，所以在定义之后，* 后面的肯定是一个地址，不要忘记了这一点，这就应该很容易明白，当然，如果还有疑问告诉我，我再做个教程，详细地给你解释。

## 牛刀教学〇二一：邻居与仓库

指针是指针（仓库），邻居是邻居（函数），但我们这一节却要来讨论讨论这指针与函数。

一个就是指针作为参数来用，其实我们前面也说过，不过这节我们举一个不是数组与字符串的例子吧：

```
#include &lt;iostream.h&gt;
void display(int *,int *);
int  main()
{
   int i,j;
   i=5;
   j=10;
   int *a=&i;
   int *b=&j;//这两句能够懂吧？上一句将 i 的地址送给 a，这一句嘛……
   display(a,b);
   return 0;
}
void display(int *num1,int *num2)
{
   cout&lt;&lt;"进来的第一个数为："&lt;&lt;*num1&lt;&lt;endl
       &lt;&lt;"进来的第二个数为："&lt;&lt;*num2&lt;&lt;endl;//大家看了，我们用了指针运算符*来取得指针指向的值
}
```

怎么样？大家将程序弄过去看看，结果怎么样？

当然我们也可以将程序的调用部分改一下，为什么一定要用两个中央变量 a 与 b 呢？

```
#include &lt;iostream.h&gt;
void display(int *,int *);
int  main()
{
   int i,j;
   i=5;
   j=10;
   display(&i,&j);
   return 0;
}
void display(int *num1,int *num2)
{
   cout&lt;&lt;"进来的第一个数为："&lt;&lt;*num1&lt;&lt;endl
       &lt;&lt;"进来的第二个数为："&lt;&lt;*num2&lt;&lt;endl;//大家看了，我们用了指针运算符*来取得指针指向的值
}
```

当然，这里我们看函数中，*num1 显然是 i 噢，因为它是 num1 是 i 的地址嘛，而*num1 又是一个取地址值的运算，所以说*num1 表示的就是 i,而 i 就是*num。纳闷，不这和引用一样？

做这个指针时，的确与引用一个样，但引用是起别名，引用中的指针是固定了，一个指向 i 的引用只能指向 i，而指针就不同了，指针是很灵活的，其至可以改变甚至交换 i 与 j 的值，我们下面再做一个程序，试试用指针改变或交换数据的值：

```
#include &lt;iostream.h&gt;
void change(int *a)
{
   ++*a;
}
int  main()
{
   int i;
   i=5;
   change(&i);
   cout&lt;&lt;"i="&lt;&lt;i&lt;&lt;endl;
   return 0;
}
```

对了，大家能够知道++*a 与*++a 的区别吗？做个程序试试。

我们再用指针做一下数据交换的例子：

```
#include &lt;iostream.h&gt;
void swap(int *n1,int *n2)
{
   int a;
   a=*n1;
   *n1=*n2;
   *n2=a;
}
int  main()
{
   int i,j;
   i=5;
   j=7;
   swap(&i,&j);
   cout&lt;&lt;"交换后，
i="&lt;&lt;i&lt;&lt;"  j="&lt;&lt;j&lt;&lt;endl;
   return 0;
}
```

大家看，一定要数据作交换啊，如：

```
*n1=*n2;
```

而不要指针作交换啊：

```
n1=n2;
```

如果我们一用上面的形式，就可能使得 n2 原来的指针找不回来，而变成了野指针了。
再说，如果我们即使定义了一个指针：
int *a;
来交换指针，那么他们将没有实际的意义。
对了，我发现一个现象，不知大家发现了没有？其实定义：
int *a;
如果我们将*a 看成一个整体，就是定义一个变量*a，也就是说*a 就是变量，而什么指针啊，可能是我们编译器给我们的一种运算是吧，我想这样一理解，容易使得你能够真确地操纵指针，例如：
int *******b;
...
cout<<*******b;
那么这个一串星加个 b 不就是这个数么？虽然不可能这样用，但这至少对我们理解指针是有好处的。拿掉一个*就是求一级地址是不？那么上面的形式的 b 就是指针的指针的指针……的指针，是不是？有多少个指针，你自己去想。
行了，既然能改指针对应的值，那么我们做一个大写字母变小写的程序：
```
#include &lt;iostream.h&gt;
void lower(char *c)
{
   if(*c&gt;=&#39;A&#39; && *c&lt;=&#39;Z&#39;)
       *c+=32;//大家知道为什么是 32 吗？当然也可用 0x20
}
int  main()
{
   char str;
   cout&lt;&lt;"请输入一个字母："; 
   cin&gt;&gt;str;
   lower(&str);
   cout&lt;&lt;"你输入的字母对应的小写母为："&lt;&lt;str&lt;&lt;endl;

   return 0;
}
```
这个程序运行起来很方便，功能也可以但就是没有实用性。
　当然，函数也可返回一个指针，我们举一个例子吧（当然是一个极不实用的例子，但却可以告诉我们一个极有用的事情，怎么返回指针：
```
#include &lt;iostream.h&gt;
char * lower(char c)
{
   if(c&gt;=&#39;A&#39; && c&lt;=&#39;Z&#39;)
       c+=32;
   return &c;
}
int  main()
```

```
{
   char str;
   cout&lt;&lt;"请输入一个字母："; 
   cin&gt;&gt;str;
   cout&lt;&lt;"你输入的字母对应的小写母为：
"&lt;&lt;*lower(str)&lt;&lt;endl;

   return 0;
}
```

怎么样？其实有好多的函数是返回指针的，使得我们使用时能够得心应手，如果有例子，我们再说吧，我们再与大家讨论一个特殊的指针：
void 指针，看程序：

```
#include &lt;iostream.h&gt;
void * lower(char c)
{
   if(c&gt;=&#39;A&#39; && c&lt;=&#39;Z&#39;)
       c+=32;
   return &c;
}
int  main()
{
   char str;
   cout&lt;&lt;"请输入一个字母："; 
   cin&gt;&gt;str;
   cout&lt;&lt;"你输入的字母对应的小写母为：
"&lt;&lt;*lower(str)&lt;&lt;endl;

   return 0;
}
```

出了问题了，我们改改程序（这里要用到强制类型转换了，呵呵，看啊，慢慢理解）

```
#include &lt;iostream.h&gt;
void * lower(char c)
{
   if(c&gt;=&#39;A&#39; && c&lt;=&#39;Z&#39;)
       c+=32;
   return (void*)(&c);
}
int  main()
{
   char str;
   cout&lt;&lt;"请输入一个字母："; 
   cin&gt;&gt;str;
```

```
   cout&lt;&lt;"你输入的字母对应的小写母为：
"&lt;&lt;*(char*)lower(str)&lt;&lt;endl;
   return 0;
}
```

这一段程序必需细嚼慢咽，我也做了好久才做出来。

由于指针是一个固定的四字节长度的空间，所以他指向的数据可以指向任意的类型，所以Ｃ＋＋就为她定义了一种指针类型 void*，它不表示没有类型，而是表示任意类型，当然这种类型为了在应用时能够使程序知道它的应用，我们必须使用强制性转换，这就是我们要强制性转换的原因。当我们做ＡＰＩ编程时，这种强制性转换用得特别的多，我们现在就要知道其所以然，以后才能用好它，它甚至可以转换成我们下面要说的函数的指针：

大家还记得牛刀在名人编程教学中使用过的：

```
(FPCALLBACK)((MView*)hWnd)-&gt;fpHandleEvent =
(FPCALLBACK)WindowHandleEvent;
```

吗？这就是一个强制性转换成函数的例子，大家看，这里有多少的转换？

（注:[->]小箭头不要看，我们没讲到，我们只看我们看得懂的，如：

```
(MVie*)
(FPCALLBACK)
```

这些表示什么呢？上面一个表示转换成名人认可的 MView 类型，那是名人定义的类型了，而下面一个就是指向函数的指针，我们下面就讲一讲函数的指针：大家可知道，函数也是放在内存中的，所以她也有首地址，即：指针，但参数我们怎么传进去呢？于是，我们就定义了：函数的指针:

```
#include &lt;iostream.h&gt;
int (*p)(int,int);
int max(int a,int b)
{
   return (a&gt;b)?a:b;//呵呵，简洁吧？
}
int min(int a,int b)
{
   return (a&lt;b)?a:b;
}
int  main()
{
   int a=3,b=5;
   p=&max;
   cout&lt;&lt;"a 与 b 中最大值为："&lt;&lt;p(a,b)&lt;&lt;endl;
   p=&min;
   cout&lt;&lt;"a 与 b 中最小值为："&lt;&lt;p(a,b)&lt;&lt;endl;
   return 0;
}
```

怎么样？简洁吧？无论你想求大小，一个函数指针就解决了，其实我们真的做程序时不会这样，我们会把指针封装在函数中，于是乎：

```
#include &lt;iostream.h&gt;
```

```
int (*p)(int,int);
int max(int,int);
int min(int,int);
int f(char c,int a,int b)
{
   if(c==&#39;a&#39;)
   {
       p=&max;
       cout&lt;&lt;"a 与 b 中最大值为："&lt;&lt;p(a,b)&lt;&lt;endl;
   }
   if(c==&#39;i&#39;)
   {
       p=&min;
       cout&lt;&lt;"a 与 b 中最小值为："&lt;&lt;p(a,b)&lt;&lt;endl;
   }
   return p(a,b);
}
int max(int a,int b)
{
   return (a&gt;b)?a:b;//呵呵，简洁吧？
}
int min(int a,int b)
{
   return (a&lt;b)?a:b;
}
int  main()
{
   int a=3,b=5;
   f(&#39;a&#39;,a,b);
   f(&#39;i&#39;,a,b);
   return 0;
}
```

怎么样？程序变成了只需要一个参数，就可以控制我们函数 f 自动求最大值或最小值了，是不是很神？其实这种程序在ＡＰＩ中比比皆是，大家还记得我们前面的一个例子吗？

```
MessageBox(NULL,"我跟牛刀学编程","牛刀对话框",MB_OK);
```

后面的 MB_OK 就是一个参数，但可能并不是调用不同的函数，但这种技术使用的是相当普及，普及到我们并不知道它们在背后用的是什么技术实现的，因为实现这些功能的技术太多了，行了，我们将这些数据列一下，大家看看啊

```
#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
```

#define MB_RETRYCANCEL 0x00000005L

其实就是这样，这些数据，在背后可能就是一些宏，然后程序根据这些宏的值，看是调用哪些程序，还是作出哪些判断，然后再作一定的处理。

当然，有关函数与指针的部分我们就讲到此结束了，我们指针部分就暂告一段落，不懂的大家可以发贴子上来讨论。

曾有人说过，学C语言不懂指针，相当于没有学过C语言。还有人做过一个试验，将一个有指针的程序全部换用没有指针来实现，程序的代码量增加了3倍。

我们不管这些说法真假，如果你指针部分没有学好，那么说实话，还不如去学Ｂａｓｉｃ好玩，我们当时学Ｂａｓｉｃ时，只有１７条语句，没了，呵呵，当然那是在小霸王学习机上。（但那可是入门啊，我很自豪的啊）。所以说，我们指针只是暂告一段落，我们以后还会千方百计地将指针拎出来给大家讲讲，大家等着牛刀以后的教材啊，牛刀脸皮可够厚的。云深不知处。

注：这是在名人上进论坛的，贴了过来，多数内容没改。

牛刀不知是要继续走下去，还是洗手不干了，有人对牛刀意见了，呵呵，这节我们先教大家编一个游戏吧，以缓和缓和气氛，不管游戏是否好，但牛刀总要尽力啊是不？（注：游戏中的rand()函数为求随机数的函数其他的牛刀都讲过，相信大家应该都能够看懂，牛刀在程序中做些注解就行了。）

```
#include &lt;iostream.h&gt;
#include &lt;stdlib.h&gt;
//需要用到 rand()随机函数时需要用到头文件 stdlib.h
enum Guess
{jian,baoz,chui};//这个是自定义的类型
int  main()
{
  cout&lt;&lt;"我猜！我猜！我猜猜猜"&lt;&lt;endl
      &lt;&lt;"********************"&lt;&lt;endl
      &lt;&lt;"**牛 刀 猜 拳 游 戏*"&lt;&lt;endl
      &lt;&lt;"********************"&lt;&lt;endl;
  int myyxd=100,coyxd=100;//我与电脑的游戏点，原来想用钱的，那不就是赌博了？呵呵
  Guess myguess,coguess;
  char ch;
  do
  {
      coguess=(Guess)(rand()%3);//这里用到了随机函数 rand()，功能为随机出一个数，强制类型转换，转换成牛刀自定义的类型
      cout&lt;&lt;"请你出拳：注：剪子出（j 或 J），包子出（b 或 B），锤子出（c 或 C）"&lt;&lt;endl;
      cin&gt;&gt;ch;
      switch(ch)
      {
      case &#39;j&#39;:
      case &#39;J&#39;:
          myguess=jian;
```

```
            break;     //注意这一句千万不要丢了
        case &#39;b&#39;:
        case &#39;B&#39;:
            myguess=baoz;
            break;
        case &#39;c&#39;:
        case &#39;C&#39;:
            myguess=chui;
            break;
        default:
            cout&lt;&lt;"你怎么瞎出啊？不玩了"&lt;&lt;endl;
            return 0;
            break;     //这句其实有没有根本没什么，但我们为了好看，放在里面，
呵呵
        }
        if(((myguess==jian) && (coguess==baoz)) || ((myguess==baoz) &&
(coguess==chui)) ||((myguess==chui) && (coguess==jian)))
        {
            cout&lt;&lt;"呵呵，不错，你赢了"&lt;&lt;endl;
            myyxd+=10;
            coyxd-=10;
        }
        else
        {
            cout&lt;&lt;"呜～～牛刀赢啊，呵呵"&lt;&lt;endl;
            myyxd-=10;
            coyxd+=10;
        }
        switch (myguess)
        {
        case jian:
            cout&lt;&lt;"你出的是剪刀"&lt;&lt;endl;
            break;
        case baoz:
            cout&lt;&lt;"你出的是包子（布）"&lt;&lt;endl;
            break;
        case chui:
            cout&lt;&lt;"你出的是锤子"&lt;&lt;endl;
            break;
        default:
            break;//呵呵，这里我们什么都不做，但是处理一下
        }
        switch (coguess)
        {
```

```
        case jian:
            cout&lt;&lt;"牛刀出的是剪刀"&lt;&lt;endl;
            break;
        case baoz:
            cout&lt;&lt;"牛刀出的是包子（布）"&lt;&lt;endl;
            break;
        case chui:
            cout&lt;&lt;"牛刀出的是锤子"&lt;&lt;endl;
            break;
        default:
            break;//呵呵，这里我们什么都不做，但是处理一下
        }
        cout&lt;&lt;"你的游戏点："&lt;&lt;myyxd&lt;&lt;"      牛刀的游戏点：
"&lt;&lt;coyxd&lt;&lt;endl;//现在的游戏挺喜欢用游戏点，牛刀这里也用。
    }
    while(myyxd!=0 && coyxd!=0);
    if(myyxd==0)
        cout&lt;&lt;"呵呵你输了！ "&lt;&lt;endl;
    if(coyxd==0)
      cout&lt;&lt;"呵呵，牛刀赢了！"&lt;&lt;endl;//牛刀耍赖哟，呵呵，大家不生气
    return 0;
}
```

这个程序是基本框架，可能不太好玩，但牛刀教的好多的东西都用到了，大家可以自己再去改改啊，原来用 co...想表示计算机的，后来被我全改成“牛刀”了，呵呵。

当然，这个程序没有用到函数和指针，如果有朋友有兴趣，可以将函数与指针都加进去，看看能将程序缩短么？能缩成什么样？

这里有些东西牛刀没的按照严格的风格来写，否则像上面的 jian,baoz,chui，最好用大写：JIAN,BAOZ,CHUI，当然，程序中的不足之处，肯定还有很多，而且有好多功能没有实现，如：更改游戏币下注的部份，游戏是否结束的询问等等。

总之，给大家作一个抛砖引玉的作用吧，当然，你有空也可以将这个程序改成名人机的程序，如果那样的话，可能像 cout<<;等等这些语句都不能用了，也不见得，大家用看看吧，如果能用，大家交流交流。

**牛刀教学〇二二：牛刀隐江湖**

呵呵，有些人对牛刀已经比较烦了，好了，牛刀其实说实在的也累了（真累）。

那么，这一节，我们先对我们近来的教程序做一个总结与未来的展望吧。

其实如果说C部分，已经大部分结果了，我们今天就讲最后一个内容：结构、联合与类

什么叫结构呢？自己定义的一个数据类型，它的形式如下：

```
#include &lt;iostream.h&gt;
struct student
{
   char ch[8];
   int number;
   int math;
   int English;
   union sss
   {
       int a;
       int b;
       int c;
       char str;
   };
};
class Point
{
   int x;
   int y;
   Point();
};

Point:Point()
{
}
int main()
{
   student st;
   st.English=80;
   st.math=90;
   return 0;
}
```

结构就是上面的struct后面定义的一系列的数据一起称为一个结构，它很有用。

联合呢？就是unio后定义的那个部分，而类，就是class定义的那一部分内容。

联合即为：

union 联合名

```
{
    数据类型 1   数据 1;
    数据类型 2   数据 2;
    数据类型 3   数据 3;
    ……
    数据类型 n   数据 n;
}
```

而结构与类则差不多：

```
class|struct 类(结构)名
{
[public:|private:|protected:]
    数据类型 1 数据 1;
    数据类型 2 数据 2;
    ……
    数据类型 3 数据 3;
    函数 1;
    函数 2;
    函数 3;
    ……
    函数 n;
}
```

结构与数据的使用很灵活，结构原来是C中的东西，原来不能包函数，并且也没有限定原外关系的符号，而在C＋＋中加入了这些，从而使得结构与类的功能可以变得很强很强，并且也是面向对象的基础。

于是乎，类被弄得很复杂，原来牛刀准备放来后面大头子讲那东西的，可是……

可是我告诉大家几乎C部分已经全部介绍完了，大家已经可以正常的搞自己的一些软件开发了，我上次做的一个猜拳游戏就是一个典型的C游戏开必，只不过没有增加图形部分，如果加进了图形元素，就完全可以当作一个成品的软件来发布了，呵呵。

C＋＋/C之路，其实还很艰难，还有什么类啊，继承啊，重载啊，多态啊……好多好多内容呢，牛刀先告退，看看牛刀的贴子点击率如何，如果点击率太低，那牛刀就与大家拜拜了，如果大家还很喜欢件刀，那牛刀会和大家再见。

对了，我们还有一种内联函数没介绍，其实那东西介绍不介绍也没什么，牛刀这廿多天来，算是给大家提供了一个入门级的教材吧，有人想叫我将 txt 文件一下子发上来，当然如果你是真想看的话，你慢慢看，如果你不想看的话，牛刀将 txt 文件发上来你也不可能看，是也不是？

下面这个软件是牛刀这次的封笔之作吧，因为有人已经有些讨厌牛刀了。

牛刀原来想把它定为:牛刀算命　软件的,但由于软件具有的强烈的迷惑性，所以牛刀没有那么做，为了破除迷信，牛刀将她定为了：牛刀魔术。

其实魔术也是骗人的，并且牛刀公布了原代码，所以说软件的设置与技巧，大家仔细地读程序，应该可以发现这其中为什么软件那么神，当然喽，你每次不能选错，如果你选错了一次，可能结果就会大大地出乎你的意料之外，我们的软

件就不灵了。

大家相信算命吗？如果用电脑给你以这种方法算，那么电脑是一百个灵，因为它的整数计算几乎是不太可能出现错误。

```
#include &lt;iostream.h&gt;
int main()
{
  unsigned char ch[3]="dd",c;
  int i,j,k,sum=0,t;
  cout&lt;&lt;"******************"&lt;&lt;endl
      &lt;&lt;"**牛　刀　魔　术**"&lt;&lt;endl
      &lt;&lt;"******************"&lt;&lt;endl;
  for(k=0;k&lt;6;k++)
  {
      ch[0]=0XB0;
      ch[1]=0XC0;
      t=0x1&lt;&lt;k;
      cout&lt;&lt;"******************";
      j=0;
      for(i=0;i&lt;128;i++)
      {
          ch[1]=0xC0 | i;
          if(ch[1] & t)
          {
          if(j%8==0)cout&lt;&lt;"*"&lt;&lt;endl&lt;&lt;"*";
              cout&lt;&lt;ch;
              j++;
          }
      }
      cout&lt;&lt;endl&lt;&lt;"******************"&lt;&lt;endl;
      if(k==0)cout&lt;&lt;"请你看上面的六十几个字，记住其中的一个，好了请按“Y 或 y”"&lt;&lt;endl;
      else    cout&lt;&lt;"请问你记往的字在这里吗？（细心看，仔细选择，在按“Y 或 y”，不在按“N 或 n”）"&lt;&lt;endl;
      cout&lt;&lt;"请选择：";
      cin&gt;&gt;c;
      if(c==&#39;y&#39; || c==&#39;Y&#39;) sum+=0x1&lt;&lt;k;
  }
  ch[1]=0xc0 | sum;
  cout&lt;&lt;"你选择的字是："&lt;&lt;ch&lt;&lt;endl;
  return 0;
}
```

**牛刀教学〇二三：自己的类型**

论坛的人气逐步有些提上来了，牛刀继续以前的教程，给大家做点教程吧，

上一讲，我们讲了，C 部分几乎结束了，但其实还有什么结构体啊，联合体啊什么的，牛刀给大家慢慢讲。

其实有些朋友问了一些问题，如：

MWindow* hWnd;//这是定义什么?!

MRect rect;//这个又是定义什么?!

UINT32 dwStyle, dwResult;//UNIT32 是不是定义无符号占 4 字节的整数!?

这些问题，的确如果没学过的话，让大家觉得很难理解，我们将一些程序定义部分弄过来让大家看看啊：

```
typedef struct tagWindow {
   MWinView    winview;        // MWinView 结构 [父类]
   UINT32      dwWinStyle;     // 窗口样式
   UINT8       *pHelpID;       // 帮助 ID 号
   UINT8       *pTitleText;    // Title 文本指针
   UINT8       *pMenuBar;      // menubar 指针
   MToolBar    *pToolBar;      // 功能区指针
   UINT        uToolNum;
   UINT*       pToolPos;
   BOOL        bToolSel;
   UINT        uToolIndex;
} MWindow;

typedef struct tagMRect{
   UINT16  x;
   UINT16  y;
   UINT16  width;
   UINT16  height;
}MRect;
```

这是牛刀在名人工具中找到的，什么意思呢？是两个结构体，我们下面就先讲讲结构体：

结构体的定义为：

```
struct 结构名
{
   数据类型 1 数据 1;
   数据类型 2 数据 2;
   ……
   数据类型 n 数据 n;
}
```

这就是结构体，大家知道 typedef 什么意思吧？如果看过牛刀的教程，就知道，其实是在为结构体起一个别名，如上面的 tagWindow 的别名为：MWindow，而 tagMrect 的别名就为： MRect。好用吧？

所以说，MWindow* hWnd;是定义了一个指针 hWnd ，而 MRect rect;就是定义了一个结构体 rect;
当然喽，这两个结构体什么意思，我们下面给大家讲解：

```
#include "osuser.h"
#include "project.h"
void HelloDrawCallback(MDC* hDC)
{
   GraphDrawText(hDC, 20, 20, 0, 0, NULL, "Hello World!");
}

APP_DLL UINT32 GuiMain(UINT32 hInstance, UINT32 hPrev, UINT8* pCmdLine,
INT nCmdShow)
{
   MWindow* hWnd;
   MRect    rect;
   UINT32   dwStyle, dwResult;

   GuiGetUserFullRect(&rect);
   dwStyle = WS_APWINDOW;
   hWnd = WindowCreate(NULL, NULL, 0, &rect, dwStyle, NULL, NULL, NULL,
NULL);

   (FPCALLBACK)((MView*)hWnd)-&gt;fpHandleEvent =
(FPCALLBACK)WindowHandleEvent;
   ((MView*)hWnd)-&gt;fpDrawBackGround = HelloDrawCallback;

   dwResult = ApRun((MWinView *)hWnd);
   GuiDestroy((MView *)hWnd);
   return dwResult;
}
```

上面一段是我从名人中“Hello.c”源程序拷贝过来的，是一个比较小的可运行程序。
MWindow* hWnd; 为窗口句柄，什么意思呢？就是我们使用的名人，上面的窗口，就用 hWnd 来表示，这样一来，我们就可以在窗口上操作了。
也就是说，我们想操作窗口，则使用 hWnd 在程序语句中就行了。
MRect rect; 呢？也是一个结构体，表示一个矩形，其中 x,y 表示了一个左上角坐标，而 width，height 则表示矩形的宽度与高度。
有些程序需要对窗体中的一个矩形区域进行操作，那么就需要用 rect 来使用，就行了。
当然，我们可以将这些通过函数传过去用，例如：
GuiGetUserFullRect(&rect);
什么意思呢？这个函数给我们的 rect 中放进了值，呵呵，就这么简单，只要这一个函数一调用，我们的 rect 就有了值了，简单吧

当然我们也可用下面办法赋值：

```
rect.x=0;
rect.y=0;
rect.height=240;
rect.width=320;
```

这样一来，我们的 rect 就表示一个实实在在的矩形了，所以说我们真正做程序时，显然是调用一个函数比较方便。

```
dwStyle = WS_APWINDOW;
```

这一句什么意思呢？赋值语句啊，后面的 WS_APWINDOW 肯定是一个自定义的值，或是一个宏：

事实上是什么呢？

我们可以在程序中找到：

```
#define    WS_APWINDOW     0x00000001          // 是否为 APWINDOW
```

这就是了，其实上面的赋值语句与下面语句效果一样：

```
dwStyle=0x00000001;
```

但就是我们用名称方便，并且一看就知道程序有什么用，所以说，我们一般使用上面一种方法赋值。当然喽，dwStyle 是多少位，对我们来说并不重要，但从 UINT32 看来，我们就知道它占有四个字节

下面就是 hWnd=WindowCreate(NULL, NULL, 0, &rect, dwStyle, NULL, NULL, NULL, NULL);

显然是一个函数，它用到了 rect（注意啊，是带取地址符的，那么就是说我们程序中可以对 rect 进行更改喽，看过牛刀以前教程的，这一点一定会明白）
dwStyle 是一个 0x00000001，而已，其实都是默认值（什么叫 NULL，其实就是 0，但我们不这样说而已，以后有空再跟大伙说吧）

```
(FPCALLBACK)((MView*)hWnd)-&gt;fpHandleEvent =
(FPCALLBACK)WindowHandleEvent;
```

我们可以知道（从 oshelp 中），WindowHandleEvent 是系统默认的一个函数 它的功能就是将用户处理完后，交给系统处理，大家用时，只要调用一下就行了，如果不求甚解的话，上面一句只要放到程序中就行了，当然你也可以将它删去，再编译运行看看，会出现什么情况？学编程就是要不断地实践啊。

我们再看看：(FPCALLBACK)是一个强制类型转换，表示我们调用的是一个回调函数，而(MView*)hWnd，前面也是强制类型转换，“->”是表示结构体，或者函数指针指向的某个值或函数。我们下面再看看后一句吧：

```
((MView*)hWnd)-&gt;fpDrawBackGround = HelloDrawCallback;
```

这一句也是一个回调函数， ((MView*)hWnd)和上面一样，我们以后再为大家做教程时详细介绍。
它的意思，我们也看得出，绘背景的函数，那么我们通过上面的方法，就告诉系统，我们做的背景函数的函数名是什么，系统在合适的时候就可以调用背景刷新

程序，对背景进行刷新，那么显然，后面的程序可以换成我们自己定义的名子。

那么这个函数是什么呢？它的形式呢？就是下面的：

```
void HelloDrawCallback(MDC* hDC)
{
   GraphDrawText(hDC, 20, 20, 0, 0, NULL, "Hello World!");
}
```

只要返回值为空，参数为：MDC＊hDC,就行了。当然里面的还可以变，名子无所谓的，如：

```
void MyDrawBack(MDC * myDC)
{
}
```

怎么样？不明白吗？如果不明白的话，你先这样记着，呵呵，学习的最高境界就是“不求甚解”。特别是做程序，你想一直到底层硬件都明白，那可能是几年以后的事情。

好了，下面的几行：

```
   dwResult = ApRun((MWinView *)hWnd);
   GuiDestroy((MView *)hWnd);

   return dwResult;
```

则为运行一个进程，释放一个进程序，与返回一个数值而已。

行了，我们这节就先讲到这儿吧，大家仔细去消化消化，不多讲了，这一讲特有深度，大家有问题就问啊，不过不能理解也没关系，下一讲我们就再次脱离名人讲程序，努力帮大家理解好名人的编程技术，当然，牛刀也会给大家做些电脑程序的，希望大家喜欢。

**指针实践〇四八：指针的表白**

其实很想下大家继续讨论我们的汇编技术，但如果你不想当黑客的话，那种技术可能真的过时了，当然，我们以后在适合的时候还会与大家慢慢讨论，这一讲开始，我们就讲一些很现实的东西：指针

菜新老想 DELPHI VS VC 其实没意思，但如果 C 语言中如果没有指针，那么我们的所有的语言其实是一样的，难怪有人说：没有学好 C 语言中的指针，就等于没有学会 C 语言。

其实说实话，我们所有的语言中，几乎都有打印输出、输入语句，条件语句与分支语句，可以说所有的语言都是一样的。

但指针使得 C 语言与其他的语言不一样了。

那么什么叫指针呢？其实指针就是地址。

```
#include <stdio.h>
void main()
{
        printf("sizeof(int *)    =%d\n",sizeof(int *));
        printf("sizeof(char *)   =%d\n",sizeof(char *));
        printf("sizeof(long *)   =%d\n",sizeof(long *));
        printf("sizeof(float *)  =%d\n",sizeof(float *));
        printf("sizeof(double *) =%d\n",sizeof(double *));
        printf("sizeof(short *)  =%d\n",sizeof(short *));
        printf("sizeof(unsigned int *)=%d\n",sizeof(unsigned int *));
}
```

大家还记得 sizeof 吗？这其实不是个函数，大家注意喽。

其实上面也就是一些没有实际用途的代码，虽然没用但其显示的结果可以告诉我们一些信息：

```
sizeof(int *)    =4
sizeof(char *)   =4
sizeof(long *)   =4
sizeof(float *)  =4
sizeof(double *) =4
sizeof(short *)  =4
sizeof(unsigned int *)=4
```

大家看，结果全是 4 为什么呢？因为这是地址的定义，地址就是占四个字节，如果你看过牛刀的《汇编实践》，你可能不明白，地址不是 20 位吗？于是出现了分段的概念，不错，正因为如此，我们说我们如果再教大家汇编有一些不合时宜了，我们现在用的计算机都是 3 2 位机，什么叫 3 2 位呢？也就是四个

字节，所以上面的结果都是 4 ，因为这是一个指针。

好了,以下我们就通过实践,来看一看我们的指针是如何定义与使用的:(注:我们以下都是以字符 char 指针为例，与大家讨论问题，其他一样)

```
#include <stdio.h>
void main()
{
        char *pch,ch;
        printf("sizeof(pch)=%d\nsizeof(ch)=%d\n",sizeof(pch),sizeof(ch));
}
```

我们也是通过结果来认识我们的指针：

```
sizeof(pch)=4
sizeof(ch)=1
```

事实告诉我们，pch 占四个字节，这当然在上面我们就知道了，而下面的字符则占一个字节。

那么我们如何使用指针呢？有两个运算符，我们称之为“指针运算符”： & *

我们以后的所有指针操作，将都与这两个运算符有联系。甚至我们上面定义时，就用到了指针运算符： *

如上面的 char *pch;

也正因为这是一种运算符，所以根本没有定义的作用，所以我们像上面的两个应该这样写：

```
char *pch,ch;
```

请大家不要写成如下那样：

```
char* pch,ch;
```

其实效果一样的，但下一种表示方法会发生歧义，让我们感觉上好像是定义了两个指针，一个指针为 pch ，另一个指针为 ch，其实不是那么一回事。当然如果我们要定义两个指针,就一定要每一个定义前面都使用指针运算符 *ch，如：

```
char *pch,*pstr;
```

这样一来，我们就定义了两个指针 pch 与 pstr 了。

行了，我们接下来，就打印出这些指针看看，指针给了我们什么吧：

```
#include <stdio.h>
void main()
```

```
{
        char *pch;
        printf("pch=%d\n",pch);
}
```

什么结果呢？结果让人莫名其妙：

```
pch=-858993460
```

当然你的运行结果可能与牛刀运行的结果不同，不过这也没什么，正因为这种结果是一种随机的，可能是内存中存在的一些无效的数据，所以说，我们为了好使用这些指针，我们可以在定义时，给指针赋个空值： NULL

```
#include <stdio.h>
void main()
{
        char *pch=NULL;
        printf("pch=%d\n",pch);
}
```

这下结果又如何了呢？结果变成了我们可以接受的了，大家计算机上的结果应该是一样的：

```
pch=0
```

使用指针：

定义一个指针是为了使用的，如果我们不使用，那么这个指针的定义一点意义都没有，你说是不是？

当然我们说过，指针只是一种运算，其实我们对任何一个数据都可以使用“&”去取得他们的地址，那其实就是指针，我们再来看一个例子：

```
#include <stdio.h>
void main()
{
        char ch;
        printf("&ch=%d\n",&ch);
}
```

当然上面我们没有定义指针，但我们也使用了指针运算符：&

其实 & 就是取地址运算符，那么结果是什么呢？

```
&ch=1245052
```

牛刀的结果如上，你如果测试的话，结果可能与牛刀的不同，如果不信，我们可做如下的实验：

```
#include <stdio.h>
void main()
{
```

```
        char ch1,ch2,ch3,ch4,ch5;
        printf("&ch1=%d\n&ch2=%d\n&ch3=%d\n&ch4=%d\n&ch5=%d\n",&ch1,&ch2,&ch3,&ch4,&ch5);
}
```

结果可能让你大惊：

```
&ch1=1245052
&ch2=1245048
&ch3=1245044
&ch4=1245040
&ch5=1245036
```

大家思考，我们的 char 不是一个字节吗？那么他们的地址之间为什么相互之间都相差 4 呢？

原来，我们的系统是 32 位的系统，小于 32 个字节的数据，都被扩充存储成了 32 位的喽，当然，大于 32 位的都被扩充成 64 位去存储，因为这样方便，因为我们 32 位系统一次就只能读出 32 位数据，不能一次只读出 8 位或者 16 位数据。

那么如果我们按字节顺序存放这些数据的话，那么，每次读出一个 32 位数据，都要将其中的一些数据都“拣”出来，那多繁啊？所以，我们的计算机使用的技术是 32 位的对齐技术。

当然如果你还不明白的话，你就知道字符数据不是按字节存放的就行了，当然如果你想用“溢出”的方法去干坏事的话，上面的内容你就必须要理解，呵呵。

当然我们看上面的规律，是前面的字符存放在高字节，后面的存放在低字节。我们一般在显示地址时，使用十六进制的表示方式，程序改为：

```
#include <stdio.h>
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        printf("&ch1=0x%x\n&ch2=0x%x\n&ch3=0x%x\n&ch4=0x%x\n&ch5=0x%x\n",&ch1,&ch2,&ch3,&ch4,&ch5);
}
```

结果为：

```
&ch1=0x12ff7c
&ch2=0x12ff78
&ch3=0x12ff74
&ch4=0x12ff70
&ch5=0x12ff6c
```

从上面的程序我们明白，我们所定义的数据一定是从后向前存放的。这样一来，我们就很容易用这种技术来干一些坏事：

```
#include <stdio.h>
```

```
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';

        printf("*pch=%c\n",*pch);
}
```

大家想，我们的结果是什么呢？这里我们告诉大家： * 与 & 正好是逆运算，“*”是取地址地方的数据。

结果可想而知：

```
*pch=c
```

干坏事，怎么干呢？改改指针吧：

```
#include <stdio.h>
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';

        printf("*pch=%c\n",*pch);

        pch+=4;

        printf("*pch=%c\n",*pch);

}
```

大家知道为什么我们指针的数据要加 4 吧？结果是什么呢？

```
*pch=c
*pch=b
```

结果如我们所料，再来：

```
#include <stdio.h>
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';

        printf("*pch=%c\n",*pch);

        pch+=4;

        printf("*pch=%c\n",*pch);

        pch+=4;

        printf("*pch=%c\n",*pch);

}
```

什么结果？

```
*pch=c
*pch=b
*pch=a
```

看，我们的指针从指向 ch3，被我们经过两次的运算，指向了 ch1 ，这就是指针，结束了吗？牛刀要干一种比较恶心的事了，大家看：

```
#include <stdio.h>
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';
        printf("*pch=%c\n",*pch);
        pch+=4;
        printf("*pch=%c\n",*pch);
```

```
        pch+=4;
        printf("*pch=%c\n",*pch);
        pch+=4;
        printf("*pch=%c\n",*pch);
}
```

大家思考，什么结果呢？

```
*pch=c
*pch=b
*pch=a
*pch=?
```

下面那一个问号是什么意思？结果出问题了吗？我们将之数据显示出来：

```
#include <stdio.h>
void main()
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';
        printf("*pch=%c\n",*pch);
        pch+=4;
        printf("*pch=%c\n",*pch);
        pch+=4;
        printf("*pch=%c\n",*pch);
        pch+=4;
        printf("*pch=%c\n",*pch);
        printf("*pch=%d\n",*pch);

}
```

结果出来了：

```
*pch=c
*pch=b
*pch=a
*pch=?
*pch=-64
```

那么-64 是什么呢？是内存的一个数据呗，于是，我们可以使用一个循环来显示出一些我们想要的东西：

```
#include <stdio.h>
void main()
{
```

```
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';

        for(int i=0;i<20;i++)
        {
                printf("*pch=%x\n",*pch);
                pch++;
        }

}
```

怎么样？结果出来了吗？什么结果？

```
*pch=63
*pch=ffffffcc
*pch=ffffffcc
*pch=ffffffcc
*pch=62
*pch=ffffffcc
*pch=ffffffcc
*pch=ffffffcc
*pch=61
*pch=ffffffcc
*pch=ffffffcc
*pch=ffffffcc
*pch=ffffffc0
*pch=ffffffff
*pch=12
*pch=0
*pch=59
*pch=13
*pch=40
*pch=0
```

大家看，内存的数据全都暴露出来了，不是好事啊，呵呵，所以说，学习指针，让人喜欢让人忧。大家玩过游戏吗？用过游戏修改器吗？游戏修改器用的也就是这个原理，用了指针，很简单吧？你用 DELPHI 来做一个试试（开玩笑，别介意啊，呵呵）。当然，如果我们真的想内存情况全部暴露的话，我们还可以改：

```
#include <stdio.h>
void main()
```

```
{
        char ch1,ch2,ch3,ch4,ch5;
        char *pch=&ch3;
        ch1='a';
        ch2='b';
        ch3='c';
        ch4='d';
        ch5='e';

        for(int i=0;i<20;i++)
        {
                printf("*0x%x=0x%08x\n",pch,*pch);
                pch++;
        }
}
```

结果：

```
*0x12ff74=0x00000063
*0x12ff75=0xffffffcc
*0x12ff76=0xffffffcc
*0x12ff77=0xffffffcc
*0x12ff78=0x00000062
*0x12ff79=0xffffffcc
*0x12ff7a=0xffffffcc
*0x12ff7b=0xffffffcc
*0x12ff7c=0x00000061
*0x12ff7d=0xffffffcc
*0x12ff7e=0xffffffcc
*0x12ff7f=0xffffffcc
*0x12ff80=0xffffffc0
*0x12ff81=0xffffffff
*0x12ff82=0x00000012
*0x12ff83=0x00000000
*0x12ff84=0x00000059
*0x12ff85=0x00000013
*0x12ff86=0x00000040
*0x12ff87=0x00000000
```

行了吧，这一讲就到这儿吧，这一讲我们学习了两个内容，一个就是取地址运算符：“&”，一个就是指针运算符“*”，好戏还在后头呢，如果你真的想学习C语言的话，就与牛刀来慢慢磨吧，呵呵。

**指针实践〇四九：指针的使用**

指针，作为 C 语言的一大特色，怎么能让你给比下去？其实我们使用每一种语言时，都有其很明显的特色，BASIC 语言，牛刀学她时，就是简单，说实话你可能不太相信，牛刀学 BASIC 时，BASIC 的语句只有 17 句，翻来覆去没变化，变化的就是我们的思维。因为 17 句够用了，当然，面对现在的面向对象与图形操作系统来说，肯定是远远不够的，于是，BASIC 引入了面向对象的概念，并且支持了所有的 API 函数，一下子使得 VB 学起来不是那么容易。

但是如果说编程的话，谁能不说 VB 来得更方便？因为 VB 是完全面向对象的。

当然也正因为 VB 是完全面向对象的，所以无形之中就失去了其灵活性与方便性，如果你是一个 VB 爱好者，你一定会为 VB 而苦恼，因为 VB 发行时，一定要将 VB 库也附带着一同发行。一个很简单的 VB 程序小则三四兆，做起来十分痛苦，在使用软件的年代（1.44M 一张），一个 VB 程序就只能分成三张软盘发行。

想当年，牛刀看着别人的程序只有十几 k 甚至几 k，牛刀就是满脑子纳闷，人家不也是编程的么，怎么人家的软件就能做得那么小，甚至不需要安装。

一切都成为了历史，我们在比较一些语言时，不能按我们的喜好来，还是那一句话，各有千秋，百家争鸣。

随着我们学习的深入，我们会发现我们编程越来越不是那么回事。为什么？

想当年，我们那十七句，够用了，什么软件都是由那十几句组成的， BASIC 语言甚至根本就没有函数的概念，在那里的程序就是直接写的，特方便。当然是有子程序的概念的，与我们函数一样。

但 C 语言就不同了，C 语言就必须有函数，至少有一个入口函数 main() 函数。

其实使用 VB 时，我们所编的程序就全是子程序了，我们在使用 VB 时，甚至根本不知道我们的程序是怎么执行起来的，但运行效果却特棒。

我们在学 VB 时，我们知道， VB 在后台给我们好多的东西全都做了，事实也真是如此，当你使用 MFC 时，你也会有同样的感觉，但学习 MFC 却不是一件很容易的事，我们必须从现在开始抓基础。因为在那里，我们的好多事情，系统都给我们做了，甚至你在其中根本找不到入口函数。

学习 C 语言还有一个优势就是灵活，大家用 VC 做过控制台应用程序吗？大家编写过面向过程的应用程序吗？大家用过面向对象的程序吗？ VC 通吃一切。

我发觉我与大家说有废话有点多了，呵呵，但每次都能得到菜新版主的回贴，牛刀其实很高兴，因为至少我们文章你是看了，并且你对 VC 的编程很感兴趣，呵呵。

闲言少叙，我们言归正传，还来谈谈指针。

大家一定明白，什么叫指针，指针就是一个地址，在 3 2 位系统中，就是一个 32 位的地址，当然如果我们以后用到 64 位的系统的，就一定是一个 64 位的地址，就这很简单，牛刀一般写文章都是想将事情说得简单，如果我们好像说天书一样，让你越看越不懂，那又有什么意义呢？

当然也正因为任何指针都是一样的，那么既然一样，我们又要定义各种类型的指针干什么呢？我们还是使用程序来说话，什么叫——实践？

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=&ch[5];
        int *pint=(int *)&ch[5];
        printf("pch =%x  pint=%x\n",pch,pint);
}
```

在上面的程序中，牛刀因为想使(int*)型指针指向一个字符型数据，牛刀使用了强制类型转换，为什么这样，也是很简单，牛刀为了使两个指针的位置一样，那么下面的结果我们也应该理解得了：

```
pch =12ff69  pint=12ff69
```

也就是说，两个指针指向同一个地方呗，当然我们也可以显示出相应的数据喽，我们的目的先是看地址的变化，那么我们怎么让其变化呢？继续看程序：

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=&ch[5];
        int *pint=(int *)&ch[5];
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
}
```

大家思考上面结果会出现什么？牛刀总爱让大家思考，其实所谓实践，就是需要思考，我们动了脑筋了，就可能一切都明白了，而牛刀不会像某些书上似的，总是会提出一些繁琐枯燥的概念：

```
pch =12ff69  pint=12ff69
pch =12ff6a  pint=12ff6d
```

从理论上来说，下面一行的结果也应该是一样的，是不？因为每个指针都只进行了自加（++）的操作啊，看不明白吗？牛刀再多弄点，反正不就是复制/粘

贴吗？呵呵。

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=&ch[5];
        int *pint=(int *)&ch[5];
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
        pch++;
        pint++;
        printf("pch =%x  pint=%x\n",pch,pint);
}
```

如果我们再将结果贴出来：

```
pch =12ff69  pint=12ff69
pch =12ff6a  pint=12ff6d
pch =12ff6b  pint=12ff71
pch =12ff6c  pint=12ff75
pch =12ff6d  pint=12ff79
pch =12ff6e  pint=12ff7d
pch =12ff6f  pint=12ff81
pch =12ff70  pint=12ff85
```

大家看出规律来了吗？ pch 是一个一个加的，而 pint 是四个四个加的，为什么同样的操作，会出现不同的结果？自加运算是自加 1 啊，怎么这里自加 4 了？

这就是我们为什么要定义指针类型的原因。假如我们所有的指针作自加运算

时，都是自加 1 的话，那么指针类型的定义就一点意义都没有了，原来指针的类型告诉了系统，我们按怎么一个规则实现指针的自加运算，以使得每次指针运算会有意义。

什么叫有意义呢？我们还是来实践一下：

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=(char*)&ch[5];
        int *pint=(int*)&ch[5];
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch++;
        pint++;
        printf("*pch=%s  *pint=%s\n",pch,pint);
}
```

大家看，我们对这两个数据进行自加运算，结果是怎么样呢？我们看：

```
*pch=fghijklmnopqrstuvwxyz  *pint=fghijklmnopqrstuvwxyz
*pch=ghijklmnopqrstuvwxyz  *pint=jklmnopqrstuvwxyz
```

自加运算嘛，向后移一个字母，但大家看 pint 指针却移动了 4 个字节，跳开了 fghi，从 j 开始。为了看得更清楚，我们再多做几次：

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=(char*)&ch[5];
        int *pint=(int*)&ch[5];
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch++;
        pint++;
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch++;
        pint++;
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch++;
        pint++;
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch++;
        pint++;
        printf("*pch=%s  *pint=%s\n",pch,pint);
}
```

大家思考，结果会怎么样呢？

```
*pch=fghijklmnopqrstuvwxyz  *pint=fghijklmnopqrstuvwxyz
*pch=ghijklmnopqrstuvwxyz  *pint=jklmnopqrstuvwxyz
*pch=hijklmnopqrstuvwxyz  *pint=nopqrstuvwxyz
*pch=ijklmnopqrstuvwxyz  *pint=rstuvwxyz
*pch=jklmnopqrstuvwxyz  *pint=vwxyz
```

上面的结论告诉我们，我们为什么要定义指针类型呢？其原因就是我们的类型告诉我们的系统我们怎么处理当前的指针，如此而已。

好了，事实让我们知道了我们我们定义一个指针，必须按照我们的需求来定义，并且我们自加运算与自减运算与我们的类型有关，而不是我们以前所想像的自加一，自减一，那么我们再对其程序做一些修改，看看加四与减四会出现什么情况：

```
#include <stdio.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=(char*)&ch[5];
        int *pint=(int*)&ch[5];
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch+=4;
        pint+=4;
        printf("*pch=%s  *pint=%s\n",pch,pint);
}
```

结果：

```
*pch=fghijklmnopqrstuvwxyz  *pint=fghijklmnopqrstuvwxyz
*pch=jklmnopqrstuvwxyz  *pint=vwxyz
```

结论告诉我们，对于指针的操作不是简单地加加减减，而是对于加加减减的数据乘以我们的指针类型所表示数据的类型大小，什么意思呢也就是说移动位移为：移动长度×sizeof(类型)

明白了上面的道理后，我们知道，当我们定义一个类型的话，就可以很容易地使用指针来指向我们的类型，而不会发生错误了。

那么我们如果指针没有这个能力，会出现什么情况呢？那我们用强制类型转换的办法来试试看：

```
#include <stdio.h>
void main()
{
        char ch[]="abc,def,ghi,jkl,mno,pqr,stu,vwx,yza,";
        char *pch=(char*)&ch[7];
```

```
        int *pint=(int*)&ch[7];
        printf("*pch=%s  *pint=%s\n",pch,pint);
        pch+=5;
        pint+=5;
        printf("*pch=%s  *pint=%s\n",pch,pint);
}
```

上面我们加了一些逗号，并且我们第一次指向的数据，第一个为逗号，我们假设我们定义的是一个整型数指，那么每次移动一次，移动四个字节，那么每次不管怎么移动，第一个字符必然是逗号。那么结果呢？

*pch=,ghi,jkl,mno,pqr,stu,vwx,yza,  *pint=,ghi,jkl,mno,pqr,stu,vwx,yza,

*pch=jkl,mno,pqr,stu,vwx,yza,  *pint=,vwx,yza,

结果正如我们所料，pint 指针做到了这一点，而 pch 指针没有做到。也就是说，如果我们的指针运算的加减运算如果没有与类型相乘的话，那么指针一移动，我们的结果就会出问题，正因如此，所以我们必须理解我们的指针是如何工作的，我们指针的工作方式使得我们对指针的操作不会出现任何问题。

正因为定义了一个指针的类型有了这种好处，所以我们使用指针就可以自由而安全地使用了。

大家如果看过牛刀前面关于函数部分的话，一定会记得一个关键字 ：void 吧？

void 原来的没有任何类型的意思，但一旦后面加上了指针运算符，其就变成了一个万能指针：

```
void *pvd;
```

什么意思呢？意思就是这个指针可以指向任何类型（当然也就是说不需要类型转换喽，看程序）：

```
#include <stdio.h>
void main()
{
        char ch[]="abc,def,ghi,jkl,mno";
        char *pch=(char*)&ch[7];
        int *pint=(int*)&ch[7];
        void *pvd=&ch[7];
        printf("*pch=%s  *pint=%s *pvd=%s\n",pch,pint,pvd);
        pch++;
        pint++;
        pvd++;
        printf("*pch=%s  *pint=%s *pvd=%s\n",pch,pint,pvd);
}
```

牛刀的原意是想看看，pvd 倒底是按什么方式移动指针的，但结果却出现了报错：
error C2036: 'void *' : unknown size

想必大家一看就知道什么意思吧？不知道类型长度，不知道就不知道吧，至少可以证明我们的万能指针是不能移动的，至少是不能在不知情的情况下移动的。

也就是说，我们的万能指针只有指向作用，而不能对万能指针进行移动操作。

好了，这一讲我们讲了两个方面内容，一个就是我们为什么要定义某种类型的指针，另一个就是万能指针可以指向任何类型，但却不能对万能指针进行移动操作，否则系统会报错。

至于我们怎么使用这些指针呢？我们下一讲再作详解。

**指针实践〇五〇：固定的指针**

牛刀在　名人论坛　上为他们做了一个《数独》游戏，好久没到这个论坛来了，呵呵，时间耽搁不少啊。

其实《数独》牛刀早就玩过了，也早就想将之搬上名上，可惜有关的数独生成的算法不太好弄啊，呵呵，后来有人发一个程序，可好了，算法拿过来就行了，所以我们说，其实算法在我们程序中占有重要的着用，说实话，我们所有人学习的 C 语言都一样，不一样的就是每个人掌握的算法的多少不同，其实牛刀也想将《黑白棋》拿上电脑，可惜也是缺算法，有谁有相应的算法给牛刀发一份，呵呵。

还是那句话，拿到C语言，如果对指针没有灵活的运用，那与 DELPHI 与 Basic　等语言就没有什么区别，所以我们要学就要对有区别的部分了如指掌。

这一讲我们还来谈一种特殊的指针——数组。

我们不和大家说数组是怎么定义的，怎么运用，那没意思，想大家也必定了解了，会用了，并且能够使用，我们想知道的是什么是数组，牛刀给出的概念与书上绝对不同，其实数组就是指针，但是一种指针常量，就好像“5”是一个常量一样，也就是说，数组的指针是一个固定的指针，不可移动的，但也反过来说，这种不可移动是指一旦程序运行起来，其是不可移动的，但如果没有运行，这个数组在什么位置，是什么那谁也说不准，既然我们要实践，那么我们还是上机做程序吧：

```
#include <stdio.h>
void main()
{
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        printf("%c\n",ch[4]);
}
```

如果你学过 C 语言，上面的程序你一看就可以知道输出的结果为：e

当然作为实践，我们也必定知道这一点，但牛刀告诉你，上面的还不能称之为“字符串”，因为其中的十个字符不假，但没有串结尾标志“\0”，那么如果我们想将其作为字符串输出，会出现什么情况呢？

```
#include <stdio.h>
void main()
{
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        printf("%s\n",ch);
}
```

当然你肯定明白 %s　就是将后面的数据以字符串的形式输出，结果是什么呢？

**引用:**

abcdefghij 烫?↕

好像多了一点东西，可能你的运行结果也是这样，有些老师会称之为垃圾数据，但作为一个爱实践的你，一定不会认为那是一个垃圾数据的，如果你那样认为，那你作为一个黑客的资格都不够。

其实牛刀不想将程序搞得多么复杂，弄出地址来让大伙瞧吧，那多累啊，毕竟我们对数据的敏感性还没到炉火纯青的地步，是也不是，我们给上下加一点东西吧：

```
#include <stdio.h>
void main()
{
        char cha[]={'A','B','C','D','E','F','G','H','I','J'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        printf("%s\n",ch);
}
```

大家还知道我们如果在后面大括号中如果直接赋值时，可以不加数组大小吧，自动加的，呵呵，所以直接加一个方括号就行了。上面的 cha[]与 chn[] 都是这样定义的，当然我们也可以定义不是这样的数据。

那么上面的运行结果会是什么样子呢？我们可以将运行结果显示一下：

**引用:**

abcdefghij 烫 ABCDEFGHIJ 烫?↕

上面的结果很出我们的意外，我们至少可以得出如下结论，每一个数据组后面都根着一个“烫”，那是数据啊，呵呵，另一个结论就是：程序最好以 ?↕ 结束我们可以想到， ?↕ 后面必然是一个“\0”。

行了，我们改一改数组的大小看看我们上面的讨论结论是否还能成立：

```
#include <stdio.h>
void main()
{
                                                        char
cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        printf("%s\n",ch);
}
```

显示结果：

**引用:**

abcdefghij 烫 ABCDEFGHIJKLMN 烫?↕

好了，通过上面的数据我们知道，那个“烫”，必然是“C”语言生成的来表示数据分配结束标志的，（注：牛刀结论，成立不成立还需大量实践）。

再一个结论就是我们程序数据部的的放置是从后向前放置的，也就是说，先定义的数据放在后面，为什么我们要这样去“琢磨”呢？因为我们为了让大家相信什么是指针？为什么牛刀说数组就是指针，牛刀可以做出像下面一样变态的程序来（想必大家在任何书上都不会看到这程序，因为这程序是牛刀琢磨出来的，呵呵。

```
#include <stdio.h>
void main()
{                                                                    char
cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        printf("%c\n",ch[-4]);
}
```

你敢想像这个程序吗？好多都是老师会说：C 语言是不会做数组的边界检查的，并且会给出下面的例子：

```
int num[10]={1,2,3,4,5,6,7,8,9,0};
```

然后会给出下面的程序：

```
  printf("%d\n",num[11]);
```

运行的结果，出现了“随机”的一个数字，于是老师会说这是垃圾数据，我们学到此就为止了。

可是我们上面的程序至少证明二点：

一、ch[10]不是数组，是指针，如果是数组，承认 ch[14] 是不做越界检查了，那么 ch[-4] 怎么解释呢？

二、ch[10]中 ch 是一个固定的指针，的确是一个指针，但说固定，是说其相当于一个常量，所以我们不能移动她，例如 ch++ 这样的运算肯定是错误的，但一定可以使用 ch+4 这样的运算，也就是说 ch 不可改变，但的确可以作为指针一样使用。（注：我们先解释完上面的程序，再来做这方面的实践。

我们再看上面的数据，输出 ch[-4]的值，结果是： 9 为什么会出现什么情况呢？因为 char 是一个字节，而大家看上面那个“烫”可是两个字节啊，我们再将 chn 输出来，看一看其中的有什么猫腻：

```
#include <stdio.h>
void main()
{                                                                    char
cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
```

```
        printf("%s\n",chn);
}
```

结果是显然的：

**引用：**

1234567890 烫 abcdefghij 烫 ABCDEFGHIJKLMN 烫?↕

我们不要再说其中的“烫”是垃圾数据喽，大家可以多弄点其他类型的再试试看，那才叫实践。

大家知道 ch 指向 a，那么“烫”跳过两个，即 ch[-3]为 0 ,那么 ch[-4]不就为 9 吗？事实上就是如此的。

于是，我们知道，了另一种定义指针的方法，使用指针的方法即另一种指针运算符 []

那的确是个指针运算符，但其定义的只不过是一个固定的指针而已，并且在数组的运算中比指针运算符：“& *”来得还要方便。

为了证明其为指针，我们可以使用后面定义的数组名来直接赋值给指针，那是一样的，就好像 x=5; 一样是合法的：

```
#include <stdio.h>
void main()
{
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        char *pch=ch;
        printf("%s\n",pch);
}
```

运行结果我就不往这里写了，大家自己上机去“玩”，呵呵，和固定的数组不同，作为一个指针变量，pch 是可以自由移动的，这一点来说，ch 就不行。看：

```
#include <stdio.h>
void main()
{
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        char *pch=ch;
        pch+=5;
        printf("%s\n",pch);
}
```

运行结果：

**引用:**

fghij 烫 ABCDEFGHIJKLMN 烫?↕

大家看，pch 指向了的数据变成了 f ，大家好理解吧？其实就是这样，为了再次证明 [] 这种运算就是一种指针运算，我们仍然可以使用这种运算符去指针上试试：

```
#include <stdio.h>
void main()
{
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        char *pch=chn;
        pch+=5;
        printf("%c\n",pch[3]);
}
```

为了让大家一看就知道 pch 指向哪，这次牛刀将指针指向了 chn，这样一来，执行了 ch+=5 后，大家可想而知指针指向了 6 是不是？（注：第一个数据为 1 啊，别忘了，那 pch[3] 是什么呢？

结果是：9

事实就是如此，这种事实也告诉我们，[] 运算符，的确是一个指针运算符，并且在指针运算中通用。

牛刀做的这些程序一点实用价值都没有，只不过是为了让大家明白什么叫指针，什么叫数组，我们应该怎么样来使用这些东西而已。

于是乎，我们的程序一点安全感都没有了。只要有一个指针跑一跑，我们的内存中的数据就全都泄露出去了。

我们说过，指针其实就是一个地址，仅仅是一个地址而已，所以说，我们做一个指针，就很自然地知道这是一个地址的引用，而我们要找到相应地址对应的内容，我们就需要使用 * 运算符，如果想找到某个数据的地址，就使用 & 运算符。

于是，我们很自然地想到一点，我们可以很简单地将一个指针指向一个数组中的任一位置：

```
pch=&ch[4];
```

大家知道，ch[4]是一个数据，大家肯定不会否认，于是，&ch[4]就是一个地址，这一点看上去挺绕人，也有好多人会被绕在这里，感觉不好理解，我们既然大家明白 ch 是一个地址，而后面的 [] 仅仅是一个运算符，就是取得一个位置 ch+4 处地址处的数值，于是乎，我们的 ch[4] 与 ch+4 就是一样的了。因为 ch 是一个地址，所以 ch+4 是一个地址 ch-4 也是一个地址， 所

以说 ch[-4] 在程序中也不会报错，如果说越界，那么 ch[14] 越不越界还与数组的大小有关，那么 ch[-4] 肯定是越界的了。

那么越界会给我们带来什么呢？至少有一点是肯定的，不安全。

看一个程序吧：

```
#include <stdio.h>
void main()
{
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        ch[-6]='X';
        printf("%s\n",chn);
}
```

结果想大家都会知道了：

123456X890 烫 abcdefghij 烫 ABCDEFGHIJKLMN 烫?↕

我们通过对数组 ch 的一个越界操作就将 chn 中的数据给改了，你想如果 chn 是一个重要的数据，那么会出现什么情况呢？ chn 肯定是要哭了。

我们再假设一个特殊的情况，大家看，我们假设 ch 与 chn 是相互不相识的邻居，而 ch 家中的东西与 chn 家中的东西各不相同，那么这时， ch 很想知道 chn 家有什么东西，又不想惊动人家，那怎么办呢？

```
#include <stdio.h>
void main()
{
        int i=0;
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
        char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        for(i=0;i<10;i++)
                ch[i]=ch[i-12];
        printf("%s\n",chn);
}
```

什么结果呢？

**引用:**

1234567890 烫 1234567890 烫 ABCDEFGHIJKLMN 烫

你看看，chn 的内容全过来了，可我们并没有对 chn 做任何操作啊，假如 chn 中是用户的密码，那结果可悲啊，我们的密码不就很容易地通过这种数据的操作，很容易地被 ch 给“偷”过去了吗？ 所以中国有句古话说得好，害人之心不可有，防人之心不可无啊。

但事实告诉我们，想要防住别人，很难，屋里说话窗外有人偷听，大路上话说草窠有人偷听，防是防不住的，这一点上来说，我们不得不说其他的语言做得比 C 语言要好，但也正因为 C 语言中的这种现象，给了黑客们很大的发挥空间：

```
#include <stdio.h>
void main()
{
        int i=0;
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
char ch[10]={'a','b','c','d','e','f','g','h','i','j','k','l','m','n'};
        char chn[]={'1','2','3','4','5','6','7','8','9','0'};
        printf("%s\n",chn);
}
```

大家注意了，ch[10]  我们定义了十个字符型数据，可我们却给出了十几个数据，会出现什么结果呢？看结果：

结果很无奈，对于这么“傻”的错误编译系统竟然认出来了，所以说，我们在真正做去做一个这种示例的东西时，应该放到函数中去实现。至于怎么实现，那牛刀在这儿就不详细讲解了（这再讲了似乎就很像《黑客教程》了，而且还是很基础的，呵呵）。

当然，以后讲到指针作参数时还会提及，不过提示大家，学了东西不要拿去干坏事，这是牛刀做教程的一条原则（鼓掌），呵呵。

上面的程序，牛刀拿到  tc2.0  中去也运行了一遍，结果全不是这么回事，当然如果你有兴趣就自己去做做这方面的实践好了，我们做程序是面向我们当前的环境的，不可能面面俱到，特别是这种实践性很强的教程，不可能给大家在任务编译环境下都讲全，那也不现实啊，你说是不？说不准牛刀做了一个编译器，再编译你的程序就不是那么回事了，呵呵。

我们通过了上面的一大套的实践证明了我们以前所以认为的运算符  [] 其实是指针运算符，那么既然是那样，我们以后做程序中，就数组指针不分家了，再来一点程序：

有人可能说，实践是检验真理的唯一标准，于是牛刀想了一个问题，我们上面的指针如果改成都是 8 个元素的数组，会出现什么情况呢？

```
#include <stdio.h>
void main()
{
        int i=0;
        char cha[]={'A','B','C','D','E','F','G','H'};
        char ch[8]={'a','b','c','d','e','f','g','h'};
        char chn[]={'1','2','3','4','5','6','7','8'};
        printf("%s\n",chn);
```

}

不信吗？结果很出乎我们的意料：

**引用:**

12345678abcdefghABCDEFGH

中间的“烫”哪去了？最后那些怪怪的数哪去了，大家思考，我们下一讲再与大家详细讨论。

牛刀哥哥.我这样做[cpp]#include <stdio.h>
void main()
{
char
cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
char chn[]={'1','2','3','4','5','6','7','8','9','0'};
printf("%c\n",ch);
}[/cpp]
打印出的 ch 为什么是 d 呢..?

**引用:**

原帖由 *冥冥* 于 2008-5-26 16:57 发表
牛刀哥哥.我这样做[cpp]#include
void main()
{
char cha[]={'A','B','C','D','E','F','G','H','I','J','K','L','M','N'};
char ch[10]={'a','b','c','d','e','f','g','h','i','j'};
char ...
冥冥看看是不是这个问题：
首先你在输出的时候用的是%c 型输出 %c 型输出是输出 char 型数据 输出大小仅为一字节
而一个地址是四字节的数据 所以%c 仅仅取这个地址的最低端的一个字节 然后将它输出
为了看清我们这个地址的原本面目 我们将%c 输出改为%d 输出 这样就可以将该地址数据取完整
那么最后一句就改成了 printf("%d\n",ch);
在你的机器上输出结果应该是 1245028 我的机器上也是一样的
也就是说这个地址是 1245028
但是为什么老是出现 d 呢？
我们将 1245028 转化成二进制 得出的数据是 10010111111101100100
%c 仅取最后一字节 即仅去其中的最后八位：01100100

而 01100100 这个数据是什么呢 正好是小写字母 d 的 ASCII 码 自然输出了 d

不知这样讲对不对 请各位指正

突然发现一个很纳闷的地方

```
#include <stdio.h>
void main()
{
        int i=0;
        char cha[]={'A','B','C','D','E','F','G','H'};
        char ch[8]={'a','b','c','d','e','f','g','h'};
        char chn[]={'1','2','3','4','5','6','7','8'};
        printf("%s\n",chn);
}
```

从这个程序里看最后一句 printf("%s\n",chn);
是以字符串的形式输出 chn 而 chn 只是一个地址啊
要想输出 chn 里面的内容必须要加一个*啊
难道%s 具有*的功能?

Google 了一天 终于发现
原来%s 果真就带有*号的取数据功能
只要给他一个字符串的首地址 他就可以将一串字符取出来 直到撞到"\0"

**指针实践〇五一：指针与数组**

可能你纳闷，上一讲不就是说的指针与数组吗？其实我们上一讲只是告诉大家，其实我们的数组就是指针，但这一讲我们倒不是想与大家讲有关数组与指针，而是指针与数组。

行了，我们再接着上一讲做个小结，再来与大家讨论今天的实践。

还记得我们上一讲中的“烫”了吗？为什么会出现这种怪怪的现象呢？我们还是用实践来说话吧：

```
#include <stdio.h>

void main()
{
        char ch[30];
        printf("%s",ch);
}
```

大家知道结果是什么吗？

**引用:**

*烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫?↕*

怎么样？事实上，字符的默认值就是这个“烫”字，呵呵。

但为什么我们当使用 ch[10] 时有这个“烫”，而使用 ch[8] 却没有呢？如果以前看过牛刀教材的话，你一定会明白，其实我们的系统是 32 位的系统，系统一次只能读 32 位数据，而不能读 8 位，你想啊，我们用的可是 32 位的地址线啊，其实牛刀也说过了，其实我们系统的目的就是兼容啊，没办法，你想想，如果我们一次处理 3 2 位数据，而系统只能处理 3 2 数据，所以我们现在要使用 8 位的 char 类型，那真是秀才遇见兵，有理说不清啊，大家想兼容还是想创新啊？而事实上是兼容限制了人们的创新，没办法，谁让我们 8 位机时会出现那么多优秀的软件呢？

事实也正是如此，即使我们现在的 3 2 位变成了真正的 3 2 位系统了，那如果以后再升成 6 4 位的呢？怎么办？事实上就是如此，我们的目标是好的，但是总是受到各种现实条件的限制。大家知道我们网络有七层模型吗？多美好？可惜我们事实的网络却是一个四层的网络，大家知道我们多么杀望 IPV6，甚至 IPV8 但事实上 IPV6 也只是处于试验阶段。

正因如此，世界上才会出现种种标准与协议的制定机构，也正因为有了这些机构，我们才能在网络上这样自由地交流。真得感谢这些标准，虽然是一种限制但却是一种自由。

所以当我们定义一个 ch[10]时，其实系统却不能真正地给我们定义一个

ch[10] 而为我们定义的其实是 ch[12] ，而我们为了不“超界”，正常却不用那两个而已，也就是说，我们定义一个字符型数据，必然会出现一些空间的浪费。而我们必须要知道，那么我们作如下的定义：

```
char cha;
char chb;
char chc;
char chd;
```

大家想想，在内存中会占几个字节呢？如果你挺用心一想的话，其实这里定义了四个字符，但在分配内存时，却分配了１６个字节，无形之中就造成了很大的浪费，再一个现实就是这种地方很容易给病毒的生存空间。

以前的 DOS 时代，人们可以通过看一种文件的大小，或软件的大小来判断是否感染了病毒，而在 Ｗｉｎｄｏｗｓ 时代，这种现象就完全变了，当我们定义一个 char cha; 我们用了一个空间，而给病毒的生存留下了三个的空间，你想想，你一个巨大的软件给病毒创造出了多少生存的机会？

当然，这不能怪谁，但我们的确可以通过一个运动的指针立刻找到那些空间，例子就不举了，反正如果一但感染了这种的病毒，你说好不好，对你软件的运行一点影响都没有，却可以足以藏身。关于怎么藏身，我们这里就不深究了。

上面是对以前的一些遗留问题的解释，这一讲的题目为——指针与数组，其实上一讲也可为——数组与指针，感觉上概念上好像有点不同，怎么个不同呢？给大家举例子吧，我们不得不承认，数组毕竟是数组，她肯定有比指针优秀的地方，例如：

```
char *pch[4];
```

大家会想到什么呢？是四个指针么？还是一个指向每组有四个元素的若干组数据？这是怎么回事呢？这就不得不讲讲我们的一个新的话题——高维数组：

说起高维数组，想必大家一定不陌生，不管你是学什么语言，数组一定是编程中的一个重要的部分，而 C 语言经过上面我们的一介绍，你可能真有点丈二和尚，摸不着头脑了。

不管怎么样，我们还是先实践吧：

```
#include <stdio.h>

void main()
{
        char cha='a';
        char chb='b';
        char chc='c';
        char chd='d';
        char *pch[4];
        pch[0]=&cha;
```

```
        pch[1]=&chb;
        pch[2]=&chc;
        pch[3]=&chd;
        int i;
        for(i=0;i<4;i++)
                printf("%c\n",*pch[i]);

        printf("\n");
        for(i=0;i<4;i++)
                printf("%p=%c\n",pch[i],*pch[i]);
}
```

看结果再说：

**引用:**

a
b
c
d

0012FF7C=a
0012FF78=b
0012FF74=c
0012FF70=d

事实告诉我们 *pch[4]; 的确给我们申请了四个字符指针，并且可以和数组一样地自由使用。

我们再改一改程序：

```
#include <stdio.h>

void main()
{
        char cha='a';
        char chb='b';
        char chc='c';
        char chd='d';
        char *pch[4];
        pch[0]=&cha;
        pch[1]=&chb;
        pch[2]=&chc;
        pch[3]=&chd;
        int i;
        for(i=0;i<4;i++)
                printf("%c\n",*pch[i]);
```

```
        for(i=0;i<4;i++)
                *pch[i]++;
        printf("\n");
        for(i=0;i<4;i++)
                printf("%p=%c\n",pch[i],*pch[i]);
}
```

看结果

**引用:**

a
b
c
d

0012FF7D=?
0012FF79=?
0012FF75=?
0012FF71=?

大家看，首先看，我们指针指向的数据变了，再看地址，也变了，而且是变一位，为什么这一位不是下一个数据呢？为什么只移动一个地址单元呢？

为了与 *pch[4]; 相区别,有好事者做了一个与之相区别的定义：(*pch)[4];

于是，大家知道了 *pch[4]; 的话，就应该知道其是四个指针，而我们就好理解一下 (*pch)[4]; 我们只要知道其是一个指针就行了,那怎么使用呢？这就涉及到高维数组了。

```
#include <stdio.h>
void main()
{
        char ch[][8]=
        {
                {'A','B','C','D','E','F','G','H'},
                {'a','b','c','d','e','f','g','h'},
                {'1','2','3','4','5','6','7','8'}
        };
        char (*pch)[8];
        pch=ch;
        printf("%s\n",pch);
}
```

结果是什么呢？

**引用:**

ABCDEFGHabcdefgh12345678?↕

行，其实我们看出了一些端倪，高维数组在内存在是怎么样存放的。高维数组就是从低字节开始存放的（注：看到了吗？牛刀现在有意地使用 8 这个存储空间了，呵呵。

再看一段程序：

```
#include <stdio.h>
void main()
{
        char ch[][8]=
        {
                {'A','B','C','D','E','F','G','H'},
                {'a','b','c','d','e','f','g','h'},
                {'1','2','3','4','5','6','7','8'}
        };

        char (*pch)[8];
        pch=ch;

        printf("%s\n",pch);
        *pch++;
        printf("%s\n",pch);
}
```

我们增加了两行，大家看完结果后再想想其中的奥妙：

**引用:**

ABCDEFGHabcdefgh12345678?↕
abcdefgh12345678?↕

大家看，当 pch++ 时，其实他移动了八个字节的位置，其实这正是数组的指针的作用，数组的指针是一个指针，而方括号中的数据只是表示指向数组的大小，目的是为了好移动我们的指针。而对这种指针的移动，那花样就多了例如 *(*pch+1)+2 那就相当于是 ch[1][2] ，这些东西，我们必须经过大量的实践才能做到了如指掌。

多于二维的数组，一般我们也用不了，其说实话，也没有必要掌握，想必你不会用一个五维的数组，五级指针去做一个软件的，如果是那样，那只是为自己找麻烦，我们做程序目的是为了方便，实用，一看就懂，如果我们做一个程序，自己想半天可能也想不懂，弄不明白，那就不是我们想要见到的结果了。

给大家提供一些形式，大家自己上机去试试，并且拿到地址来瞧瞧，分析分

析，其实牛刀做的很多程序大家如果仔细想想，没一个实用的，其实就是为了我们好理解一些东西，大家也需要这样做，我们不能去背一些书上的概念，因为那些东西可能不仅仅是晦涩难懂，有些东西还容易引起误解。

y/y[0],*(y+0),y/y+1/y[1],*(y+1)/

y[1]+2,*(y+1)+2,&y[1][2]/*(y[1]+2),*(*(y+1)+2),y[1][2]

在这里需要与大家说明的，不管是多少维的数组，多少级的指针，都是一个地址，如果是一个二级的指针，如：

char *pch[4];

char **pcha;

可能这些指针只是一个地址的地址，也可能是一个数组的地址，让我们在使用时灵活运用，切不可使用我们不明白的，其实指针与数组之间的关系相当复杂，并不是牛刀几个程序就可以说得了的，当然如果不是实践，可能会有些书会给出一些总结性的条条来，不过大家明白一点，一般就可以通吃一切：几级指针，加几个“*”就是数组，其他的都是地址。但表示的含义可能不同，数组与指针的关系我们暂告一段落，关于多级指针，我们再做一个程序给大家看看：

```
#include <stdio.h>
void main()
{
        char ch='a';
        char *pch=&ch;
        char **ppch=&pch;
        printf("ppch=%p\n",ppch);
        printf("*ppch=%p\n",*ppch);
        printf("**ppch=%c\n",**ppch);
        printf("\npch=%p\n",pch);
}
```

结论：

**引用:**

ppch=0012FF78

*ppch=0012FF7C

**ppch=a

pch=0012FF7C

大家看，*ppch 其实就是 pch ，他们都是表示一个地址，而 *pch 指向的是一个数据 'a' ，而 *ppch 却是指向一个地址，并没有指向一个字符型的数据，这就是最大的区别。

如果有必要，我们以后再与大家讨论这种怪怪的多维指针，一般我们使用的情况下，最多二维也就够用了，如果你用一个三级指针，四级指针，那你那不是在做程序了，似乎你要出一个很困难的题目，而即使是出题目，这种绕人的题目意义也不大。

使用指针的目的原来是为了方便，迅速，而如果我们的程序中的多级指针倒使得我们做程序不那么方便了，那不是与我们的目标背道而驰了么？

指针的基础部分我们就先讲到这里了，我们以后再讲，那就是指针的运用了，我们的指针将会出现在我们程序的每个角落，我们以后讲函数，讲结构，讲类，讲到我们面向对象，我们都要涉及到指针，那时，我们一定会发现，我们指针的运用是多么的方便，多么的好用，而我们现在为大家讲的实践，大家一定要仔细上机分析，一定要搞透彻，这样我们以后的课程才能顺利地开展与展开。

最后再提一下，大家千万要注意了，其实指针就是一个地址，而隐藏在这地址背后的东西，可能就是千奇百怪，我们只有遇到了这些东西，才能明白其中的奥妙。

在这一讲的最后，我们再提一下两个非常好用的东西——  new 与  delete

这其实完全可以一提而过，我们上面都是让指针指向一个数据，其实这种指针使用空间非常有限，而一旦有了  new 与  delete ，我们的指针则给我们的程序增添了好多美妙的生机。

这两个运算符（我们不称之为函数，关键字，而也称之为运算符，是因为牛刀感觉她真的与以前我们想像中的一些东西完全不同，似乎她给了我们什么，给了我们一种原来并不存在的东西），先看程序：

```
#include <stdio.h>
void main()
{
        char *pch;
        printf("%c",*pch);
}
```

上面的程序一定会报错，因为我们申请的 *pch  并没有指向任何一个数据，或者干脆说， pch  这个指针其实不存在，只不过是定义过了。当然我们可以随便使其指向一个地方：

```
#include <stdio.h>
void main()
{
        char *pch=NULL;          //为了表示不存在，建议大家在定义时，如果不指向数据，将其定义为空指针。空指针，其实就是  0
        pch=(char*)0x10000;
        printf("%c\n",*pch);
}
```

当然可以像上面那样做，弄一个强制转换，可那样挺危险，因为那个空间可能是系统的空间。而不是系统分配的空间，而  new 却可以让系统为我们分配临时空间：

```
#include <stdio.h>
void main()
```

```
{
        char *pch=NULL;
        pch=new char;
        printf("%c\n",*pch);
}
```

大家看，上面的 new char 是否很像是一个常量，或者是一种运算呢？所以感觉上 new 更像是一个运算符,就像负号,取地址等运算符一样,其实 sizeof 牛刀也喜欢看成是一种运算符。

我们理解那个 new ，我们可以暂时理解成是“无中生有”,但是我们申请一个空间，这个空间就会给我们的指针专用，于是，下面的程序是相当危险的：

```
#include <stdio.h>
void main()
{
        char *pch=NULL;
        pch=new char;
        pch=new char;
        pch=new char;
        pch=new char;
        pch=new char;
        pch=new char;
        printf("%c\n",*pch);
}
```

上面的程序不会出现任何问题，当然如果你想试试，那你不可能做成像牛刀这样的多少行，你用循环试试，于是，这段程序就会出现了你申请了好多的空间，但却没有释放这些空间，于是我们假设这些空间中有数据，这些数据会怎么样呢？

理论上来说，这些数据全没了，因为我们没有合适的指针指过去了，于是我们永远都拿不到这些数据。那么这些数据无形之中“丢失了”。

真的丢失了吗？牛刀也做了一些程序测试了一下：

```
#include <stdio.h>
void main()
{
        char *pch=NULL;
        int i;
        for(i=0;i<10;i++)
        {
                pch=new char;
                printf("pch=%p\n",pch);
        }
}
```

结论如下：

**引用:**

pch=00431D20
pch=00431CF0
pch=00431CC0
pch=00431C90
pch=00431C60
pch=00431C30
pch=00431C00
pch=00431BD0
pch=00431BA0
pch=00431B70

如果你看一下，上面的地址真的没任何的规律耶。

不对，好像还有一点，你自己去琢磨吧，呵呵，但这不代表在任何程序中都是一样啊，我们再加个 delete 看看：

```
#include <stdio.h>
void main()
{
        char *pch=NULL;
        int i;
        for(i=0;i<10;i++)
        {
                pch=new char;
                printf("pch=%p\n",pch);
                delete pch;
        }
}
```

结果：

**引用:**

pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20
pch=00431D20

为什么一加 delete 结果就一样了？就像牛刀这样实践，你自己定义两个

甚至多个指针看看，牛刀建议大家没必要去寻找其规律，当然如果你一心想当黑客的除外，呵呵。

但从上面的加了 delete 的程序来说，我们的空间真的被系统收回了，因为可以重复利用了，所以我们从理论上来说，请大伙一定将 new 与 delete 配对使用，并且不多不少，也就是我们在程序中一旦写了一个 new ,就立刻顺手将 delete 部分也写了，这样一来，我们的程序就不容易出现问题了。至于什么问题，我们以后慢慢讲，如果我们假设内存是一个资源（其实本来就是一个资源，呵呵），那么就存在着一个资源释放的问题，大伙玩游戏，特别是大型的游戏，是不是会遇到一个游戏玩久了，电脑就提示“空间不足”或“虚拟内存不足”等等问题啊，其实就是编程者在程序制作中肯定有地方忘了释放资源了，当然我们也可以做出一个这样的程序：

```
#include <stdio.h>
void main()
{
        char *pch=NULL;
        int i;
        for(i=0;i<100000000;i++)
        {
                pch=new char;
                printf("pch=%p\n",pch);
        }
}
```

当然，像上面的程序，牛刀是等不了啦，如果你想快，可以将 printf 语句拿了。你慢慢等结果好了，牛刀是等不了了，如果你的电脑能力还是很强的话，那你就来个二重循环，不行来个三重四重，看你的电脑能顶多久，呵呵。

**指针实践〇五二：指针与函数**

首先感谢管理员，每次都是管理员帮我修改发贴，使我的贴子更加好看，牛刀甚是感激，说实话，教材写完了，真是懒得再去加那些转换符，哎～～～

经过上面的讲解，大家应该明白什么是指针了，其实指针就是内存的地址，也就是说，只要在内存的东西，不管是什么，都有一个地址，也就是说都有一个指针，那么我们的函数不也是在内存中吗？既然如此，那么我们就有理由可以定义一个指向函数的指针，看程序：

```
#include <stdio.h>
int max(int x,int y)
{
        return x>y?x:y;
}
int min(int x,int y)
{
        return x<y?x:y;
}
int mid(int x,int y)
{
        return (x+y)/2;
}
void main()
{
        int (*p)(int x,int y);
        p=max;
        printf("3 和 5 中的最大的数为:%d\n",p(3,5));
        p=min;
        printf("3 和 5 中的最小的数为:%d\n",p(3,5));
        p=mid;
        printf("3 和 5 中间的数为:    %d\n",p(3,5));
}
```

这有什么用啊？可能你会这样问，其实也不奇怪，果真没什么用，但如果你继续跟着牛刀向后看，你会发现，这种东西的作用还真的不可或缺。大家记得牛刀以前教程中有一些程序吗？如果记不得的话，请将你的 MJ6000 的环境找出来，里面有例程“Hello”，我们可以看到一些程序：

```
#include "osuser.h"
#include "project.h"
void HelloDrawCallback(MDC* hDC)
{
    GraphDrawText(hDC, 20, 20, 0, 0, NULL, "Hello World!");
}

APP_DLL UINT32 GuiMain(UINT32 hInstance, UINT32 hPrev, UINT8* pCmdLine,
```

```
INT nCmdShow)
{
    MWindow* hWnd;
    MRect    rect;
    UINT32   dwStyle, dwResult;

    GuiGetUserFullRect(&rect);
    dwStyle = WS_APWINDOW;
    hWnd = WindowCreate(NULL, NULL, 0, &rect, dwStyle, NULL, NULL, NULL,
NULL);

                    (FPCALLBACK)((MView*)hWnd)->fpHandleEvent          =
(FPCALLBACK)WindowHandleEvent;
    ((MView*)hWnd)->fpDrawBackGround = HelloDrawCallback;

    dwResult = ApRun((MWinView *)hWnd);
    GuiDestroy((MView *)hWnd);

    return dwResult;
}
```

大家看到了吗？这一句：((MView*)hWnd)->fpDrawBackGround = HelloDrawCallback;

我们只需要将我们程序的名称（其实说实话，函数名，我们也可认为是一个指针，只不过与数组一样，是一个固定的指针而已）赋值给某一个指针函数就一切 OK 了，而我们的相应的功能的函数则放在相应的函数 void HelloDrawCallback(MDC* hDC) 中，那么调用这个函数的工作就由系统给我们做了，什么时候做呢？屏幕刷新时或者是接收到相应的消息时。

对了，说到消息这儿说一句，其实消息与我们的死循环一个样，只不过在那种消息循环中，就是通过这种指针调用相应函数的方式来达到传送消息的目的的。

为了大家好理解这一点，并且知道我们以后将会与大家讲解的 API 的原理，下面我们就来模拟这种消息循环的处理过程，想必我们以后会碰到，以便大家碰到了不陌生。

```
#include <stdio.h>
int k(void);
void s(int i);

void main()
{
        int i;
        while((i=k())!=0)
        {
```

```
                s(i);
        }
        printf("你输入为 0，退出系统 OK\n");
}

int k(void)
{
        int i;
        printf("请输入数据（0~9）\n");
        scanf("%d",&i);
        return i;
}
void s(int i)
{
        printf("你输入的数据为:%d\n",i);
}
```

大家看，上面的程序和指针没有关系，只是模拟了消息循环的过程，因为我们只是演示的目的，所以我们假设系统只使用 0~9 9 个数字作为按键消息，按 0 退出系统，而按其他数字则进入消息循环。

于是大家看，我们通过

```
while(...)
{
...
}
```

这种结构，就很顺利地将消息循环做进了系统，而我们主要的编程过程则被分配到了不同的函数：k() 与 s(int i) 大家看，这样一来，我们的程序似乎显得更专业了，k()专门处理按键消息，而 s(int i) 则专门处理显示消息，两部分被有效地分开了。

那这与我们函数与我们指针又有什么关系呢？其实很简单，我们只需要在外面定义一个函数原型的指针，大家看

```
#include <stdio.h>

int (*pfun)(int x,int y);

int k(void);
void s(int i);

void main()
{
        int i;
        while((i=k())!=0)
        {
```

```
                s(i);
        }
        printf("你输入为 0，退出系统 OK\n");
}

int k(void)
{
        int i;
        printf("请输入数据（0~9）\n");
        scanf("%d",&i);
        return i;
}
void s(int i)
{
        printf("你输入的数据为:%d\n",i);
}
```

大家看见上面的指针了吗？这一部分与 main 函数可以放到一个单独的文件中隐藏起来，或者放到一个头文件中去，甚至放到 .DLL 二进制库中去，这样一来，一下子，我们的程序可能就找不到主函数了，而我们要做的就是定义对按键处理的函数，定义显示的函数，定义我们自己需要的函数，定义好后，将函数名赋给一个指针就一切 OK 了，这就是上面我们对名人机那一段程序的理解，如果大家还不能理解的话，没关系，我们以后再遇到再讲，如果我们这一讲与大家铺开做一个操作系统的话，可能与我们讲函数与指针的关系就背道而驰了。

上面讲的是函数的指针，其最主要的原则就是，函数与函数指针的返回值要相同，参数个数要相同，每个参数的类型要相同，如果达到这三个目的，随时随地都可以将某一个指针指向这个函数，从而实现对这个函数的调用，基于什么时候调用，如何调用，那则不是我们的事了，那要看主函数是怎么设计的了。

顺便提一下，如果你以后跟牛刀学 API 则还可以看到一个主函数 WinMain 而如果你跟牛刀学到了 MFC ，则什么主函数都看不到了，其实也就是这个原因，因为 MFC 使用了函数指针的技术，当然喽， MFC 使用了一个巨大的 MFC 类库，所以其编译出来的程序不能直接发行，必须带着 MFC 类库一起发行。当然如果我们原意，可以做自己的类库，不过那是以后的事情了，我们这一讲就不扯远了。

下面我们看如何在函数中使用指针：

其实大家只要明白指针就是一个地址，那么一切都好办，下面就举一个函数与指针中最典型的例子，牛刀不得不举这个例子，因为没有再比这个例子再好的例子了（交换两数据）

```
#include <stdio.h>
void s(int x,int y)
{
        int t;
```

```
        printf("==========================\n");
        printf("进入了交换函数后，x=%d,y=%d\n",x,y);
        t=x;
        x=y;
        y=t;
        printf("退出交换函数之前，x=%d,y=%d\n",x,y);
        printf("==========================\n");
}

void main()
{
        int a=3,b=5;
        printf("进入交换函数之前，a=%d,b=%d\n",a,b);
        s(a,b);
        printf("退出交换函数之后，a=%d,b=%d\n",a,b);
}
```

如果你看有一些书，会使用 swap ，而这个函数却是 VC6.0 系统已经定义了，所以牛刀改成了 s(int x,int y) 功能是一样的，也正因为我们常用的函数，系统已经定义了，所以其实我们只要包含相应的头文件，我们其实使用这些函数就行了，如果不是牛刀为了讲解的需要，这种常用的函数根本没必要自己去做，系统为我们都已经做好了，行了，言归正传，看结果：

**引用:**

进入交换函数之前，a=3,b=5
==========
进入了交换函数后，x=3,y=5
退出交换函数之前，x=5,y=3
==========
退出交换函数之后，a=3,b=5

结果显然是数据没有交换过来，为什么，因为我们传进去的是数值，也就是相当于考试作弊，有两种作弊方法：一种办法是，别人成绩好，我们只需要将他的考试卷上的东西抄过来，交上去，我们就可以得１００分了，而地址传递则比较恶心，人家做好了，你哗，将人家卷子拿过来，名子一改，交给老师，那也就是１００分了。同样１００分，来源不一样。

而函数却没有我们聪明，函数是如何作弊的呢？一看人家的，想抄，只不过是想抄，但却忘了抄了，所以原来的数据根本没有换过来，所以说，电脑没有我们聪明，呵呵，所以说，想让电脑作弊。唯一的办法就是，改姓名，地址传递。

但电脑更恶心，不需要改名子，将名子先写好喽，卷子送给人家，人家函数就帮你做完了，并且还不需要你交，他帮你交，等卷子发下来，你就１００分了，厉害吧，言归正传，我们不作弊了，看程序：

```
#include <stdio.h>
```

```
void s(int *x,int *y)  //修改
{
        int t;
        printf("============================\n");
        printf("进入了交换函数后，*x=%d,*y=%d\n",*x,*y);  //修改
        t=*x;         //修改
        *x=*y;         //修改
        *y=t;         //修改
        printf("退出交换函数之前,*x=%d,*y=%d\n",*x,*y);         //修改
        printf("============================\n");
}

void main()
{
        int a=3,b=5;
        printf("进入交换函数之前，a=%d,b=%d\n",a,b);
        s(&a,&b);         //修改
        printf("退出交换函数之后，a=%d,b=%d\n",a,b);
}
```

对上面修改部分，牛刀全都作了注释，如果看不懂的话，去看牛刀前面的教材，反正你千万明确一点就行，指针是一个地址，想得到数据，就必须使用地址运算符 “*”，想得到一个数据的地址则必须使用取地址运算符 “&”。

再有一个就是返回值是一个地址（注：这里说一下啊，大家还记得吗？牛刀说过，数组就是一个固定地址的指针，所以一切数组传递，都是地址传递，如果有兴趣，你去做几个程序实践一下，牛刀在这里就不给大家实践了。）：

```
#include <stdio.h>
int i;
int *max(int x,int y)
{
        i=x>y? x : y;
        return &i;
}
void main()
{
        int *p;
        p=max(3,5);
        printf("*p=%d\n",*p);
        printf("i=%d\n",i);
        printf("p=%p\n",p);
        printf("&i=%p\n",&i);
}
```

其实这是一个例子，例子有点牵强，但如果你能读明白，你也就对函数与指针的关系可以说是比较了解了，注意了，函数中千万不要返回一个函数中定义数

据的指针，因为我们的函数在内存中调用过后就没有了，那么这个指针也就消失了，这涉及到作用域的问题，为什么牛刀定义了一个全局变量 i 就是这个原因。

当然我们说函数结束了，并不是说局部变量就没有了，很可能我们还能读到那个变量，但如果真的那样，会出现什么情况呢？

```
#include <stdio.h>
int *max(int x,int y)
{
        int i;
        i=x>y? x : y;
        return &i;
}
void main()
{
        int *p,*pn;
        p=max(3,5);
        printf("*p=%d\n",*p);
        pn=max(6,7);
        printf("*p=%d\n",*p);
}
```

大家看，虽然我们第二次并没有对指针 p 进行操作，可出现了什么结果呢：

**引用:**

*p=5
*p=7

你忍心看着你没有动某个数据，而这个数据却被改变吗？这就是“垃圾数据”的问题，虽然我们知道其实这个数据并不是垃圾数据，但事实上我们需要这样认为，还是那句话，除非你真的想当黑客，才会去研究这些怪怪的东西，呵呵。

再讲一个问题，那就是“野指针”的问题，看过这个名词，没见过具体的程序，所以牛刀做的这个程序仅仅说明这个一个意思而已，如果与你学习的不符，还望见教，但牛刀这里就是这个意思了。

为了真能将指针放跑，牛刀这里的例子不是函数的例子，而用了 new 却没用 delete ：

```
#include <stdio.h>
void main()
{
        int *p;
        p=new int ;
        *p=3;
        p=new int ;
```

```
        *p=5;
        p=new int ;
        *p=7;
        p=new int ;
        *p=9;

        printf("*p=%d\n",*p);
}
```

结果很显然，不上机大家也知道最后的结果是：

*p=9;

可上面的 3 5 7 哪去了？这些指针突然被我们放跑了，并且告诉你，如果你不想当黑客的话，你这些指针永远也找不回来，所以我们称之为“野指针”，如果你在函数中返回一个局部变量的指针，那么这种指针跑野的现象则会更常见，并且告诉你一个事实，这种程序，你想调试还没门，因为你根本不可能想到，一个函数的局部变量，还能够改变另一个函数的局部变量，甚至用函数将另一个函数的局部分量给覆盖。

可在黑客那儿就完全不同了，黑客们巴不得你这种野指针越多越好，这种指针在黑客那，就被称之为后门，假设你的帐号与密码的指针跑野了，那么黑客们根本不需要拿到你软件去详细分析其中数据结构，只需要拿到指针，你的软件安全性就一切完蛋了。

原因简单啊，你的密码帐号可能不同，但地址不可能变的啊，事实就是不变，没办法。

当然一个优秀的软件，在帐号与密码这部分的投入也是最多的，但如果我们一开始学习这方面知识时，就打下良好的基础，让我们的程序少出问题，那不是更好么？

但话又说回来，想让我们的程序一点问题都没有，那几乎是不可能的，这就需要我们做程序时养成良好的习惯，一个很好的习惯就是我们申请了一块内存后及时销毁 delete 。

最后再说一个题外话，因为我们使用的是 VC6.0 是兼容 C++ 的，所以我们不能“死脑筋”，因为我们的 C++ 的确比 C 要优秀多了。在 C++ 中，给我们数据的参数传送定义了第三套方案：重命名法。

其实也是地址传送，但传过去的名称却不表示地址，而就是值，这样一来，我们函数中的参数与调用的原数据是同一数据（建议大家不要理解成同一地址，因为这里并不保存地址，也就是说，并没有一个指针指向一个实在的地址，先看程序：

```
#include <stdio.h>
void s(int &x,int &y)        //修改
{
```

```
        int t;
        printf("==========================\n");
        printf("进入了交换函数后， x=%d,y=%d\n",x,y);
        t=x;
        x=y;
        y=t;
        printf("退出交换函数之前， x=%d,y=%d\n",x,y);
        printf("==========================\n");
}

void main()
{
        int a=3,b=5;
        printf("进入交换函数之前， a=%d,b=%d\n",a,b);
        s(a,b);
        printf("退出交换函数之后， a=%d,b=%d\n",a,b);
}
```

大家看，这一程序与最上面那程序只在参数部分改了，由 void s(int x,int y); 变成了 void s(int &x,int &y); 就这一点区别，大家看就像定义一个地址，而不是定义一个指针似的，牛刀也就不清楚，但大伙明白一点就行了，参数中只要加了 “&”（请注意，这儿不要理解成“取地址”，仅仅是C++的一种规定）。那么函数参数中相应的数据与调用中的数据就是同一数据，也就相当于考试时抄都懒得抄，一张试卷上，两个名子就行了，而老师也是一个糊涂老师，看有几个名子，这一个分数就是这两个人的。

再举一个简单的例子，假如你叫大毛，而在家家中的人就叫你二狗子，那么家就相当于函数，而出来就相当于不在函数里面了，虽然函数中与函数外对你的叫法不同，但却是同一个人，事实就是这样。

值传递 地址传递 重命名 这三种传递方式各有千秋，大家在使用中灵活运用好了，因为我们还没有学到 结构 联合 类 等一系列的知识，我们的指针功能目前看来还有点显得苍白无力，不过也快了，我们对指针的专业讨论到此就结束了，指针部分还将会在以后的教程中不断出现，原因还是很简单，指针太好用了。

**结构实践〇五三：初识结构体**

指针完了吗？没有，指针才刚开始，如果没有我们以下的知识，那么我们的指针的使用将大打折扣。

其实我们这一讲开始，应该是结构体实践才是，呵呵，那么怎么定义一个结构呢：

```
struct 结构体名
{
  数据表定义
};
```

和你们看的 C 语言材料相同吗？不管是否相同，上面就算是牛刀定义法啦，呵呵，其实牛刀顺便说一下联合，我们不特意再讲联合了：

```
union 联合体名
{
  数据表定义
};
```

什么是联合呢？就是好多数据占同一空间，如果我们用到再与大家说，我们先举个结构的例子啊，什么例子呢？

```
struct POINT
{
  int x;
  int y;
};
#include <stdio.h>
struct POINT
{
        int x;
        int y;
};

void main()
{
        POINT pt;
        pt.x=3;
        pt.y=5;
        printf("pt.x=%d,pt.y=%d\n",pt.x,pt.y);
}
```

上面这是一个典型的结构体啦，一旦你进入了 Window  的世界，上面这种结构体将是最基本的结构体。千万要注意了，最后的分号一定不能忘了啊，如果你看其他的 C 言书的话，可能会需要这样定义：

```
  struct POINT pt;
```

所以说定义时，可以使用  typedef 来定义，如：

```
typedef struct POINT CPoint;
```

```
CPoint pt;
```

当然，定义与后面的重命名的定义与前面结构的定义也可以放在一起：

```
typedef struct POINT
{
  int x;
  int y;
} CPoint;
```

当然，甚至我们定义中的名称都可以不要了：

```
typedef struct
{
  int x;
  int y;
} CPoint;
```

但对于我们的 VC6.0 系统来说，其实上面的种种定义是一样的。

既然是一样，我们就随便选一种来来继续我们的结构体之旅吧：

```
#include <stdio.h>
typedef struct
{
        int x;
        int y;
} CPoint;

void main()
{
        CPoint point,*ppoint=&point;
        ppoint->x;
        ppoint->y;
        printf("pt.x=%d,pt.y=%d\n",pt.x,pt.y);
}
```

总结上面的一些程序，我们知道一些最基本的东西：

·结构体中可以定义一些基本的数据类型
·引用结构体中的数据使用一个小点“.”就一切 OK
·引用结构体指针中的数据使用箭头，其实是一个减号加一个大于号：“->”。

也就是说 -> 是结构体中的指针运算符。 到现在为止，我们学过了四种指针运算符了：

**引用:**

& 取地址
* 取数据（指针引用的数据）

[]取数据（固定指针[数组]引用的数据）
->取数据（结构体中的元素）

你说 VC 做得就是好啊，当我们在相应的结构体或指针中使用“.”或“->”时，后面就会给我们列出所有的元素，多好啊，一按相应的字母，一个空格，就可以立刻将相应的元素上屏了，我们做个程序试试：

```
#include <stdio.h>
struct        DATE//日期型数据
{
        int yea;        //年
        int mou;        //月
        int day;        //日
};
void main()
{
        DATE date;
        date.day=24;
        date.mou=6;
        date.yea=2007;
}
```

日期数据，又是一个典型的数据，大家一定上机试看看，牛刀做的每一个数据后面都有一个注释，大家一试就知，当我们选择了相应的元素时，注释也乘机就显示出来了，这一点与上面的一些提示输入让我们编程显得分外地方便，我们可以随心所欲地定义一些数据，不管多长，我们只需要一个或几个字母，一加空格，就一切 OK，这要算人家 VC 环境做得好，当然牛刀下面的例子不会做得太复杂的，牛刀这里实践的含义之一也是要熟悉我们编程的环境，在这一点上，牛刀走过不少的弯路，也正因为牛刀走过好多的弯路，才让牛刀在网上发这一套还算比较系统的教程，其实牛刀并不想写一本书，而只是想带领大家来实践，并且告诉你原因，原理，这也是牛刀写这套教程的目的。

说些题外话吧，如果你直接去学 VC 相关方面的书，可能你很容易做出了一个很好的软件，可你并不知道这个软件是怎么来的，因为你做软件的过程只不过是将书上的一些程序段搬上了电脑，这一点如果你学过 MFC 你可以很容易地感觉出来，当然牛刀也不是夸大什么，你想学 MFC 还真不如慢慢地看看牛刀的教材，但一定每个程序都上机实践。

当然喽，牛刀实践过程中的一切错误也都放进教材中，并且告诉你为什么会出现这些错误，甚至有时候做的程序就是一个错误的程序，并且来分析，这都是有可能的，但还是那句话，我们的教程是基于 VC6.0 的，我们也从原理慢慢来，当然大家不能心急，俗话说心急吃不了热豆腐，再说牛刀的教程可是牛刀在电脑上一个字一个字的敲进去的啊，你放心，牛刀教程一般不可能使用复制一段代码来充数的，那没意思，所以错误肯定是再所难免。

行了，言归正传吧，我们上面的程序其实一点功能都没有，只是演示给你看，你一上机就知道是怎么回事了，就这种作用。

我们的结构甚至可以嵌套使用，一样的，大家看：

```
#include <stdio.h>
struct        DATE//日期型数据
{
        int yea;        //年
        int mou;        //月
        int day;        //日
        struct TIME
        {
                int hou;        //时
                int mun;        //分
                int sec;        //秒
        };
};
void main()
{
        DATE date;
        date.day=24;
        date.mou=6;
        date.yea=2007;
}
```

那我们怎么调用中间的数据呢？其实一样的使用“.”或者是“->” 当然这里要说两方面内容：

·在外层的结构体外不可以直接定义 TIME 结构体（内层结构体）

·调用内层结构体方式不变。

大家可以看到我们提示中的 TIME 前会出现一个图标与其他不同，大家自己仔细体会体会，有些图标表示结构体，有些表示元素，玩玩自然就会了，懂了。

其实内层结构体的名子对我们来说根本没有任何的作用，你说是不？所以我们一般内层结构体的名子可以省略，而引用结构体一样：

```
#include <stdio.h>
struct        DATE//日期型数据
{
        int yea;        //年
        int mou;        //月
        int day;        //日
        struct
        {
                int hou;        //时
                int mun;        //分
                int sec;        //秒
        };
```

```
};
void main()
{
        DATE date;
        date.day=24;
        date.mou=6;
        date.yea=2007;
        date.hou=20;
        date.mou=31;
        date.sec=25;
}
```

大家看内层的结构体好像没有意义啊，但如果内层是联合就很有意义了，看：

```
#include <stdio.h>
struct        MESSAGE//日期型数据
{
        int yea;        //年
        int mou;        //月
        int day;        //日
        struct
        {
                int hou;        //时
                int mun;        //分
                int sec;        //秒
        };
        union
        {
                char teachername[10];        //老师名称
                int studentage;        //学生年龄
        };
};
void main()
{
        MESSAGE message;
...
}
```

上面的联合做得有点牵强，其实意思到了，也就行了，如果我们要定义一个人的信息的话，那么这个人要么是老师，要么是学生，所以联合的意思就是其中的元素只能是一种属性，这些东西具体用到了大家自然就明白了，联合的使用需要学的东西不太多，大家上机一试便知，如果有必要，以后给大家展开讲，或者大家发贴询问时，给大家详细解释。

那么结构体的定义到底有什么用呢？那好处大大的啊，大家知道，指针就是一个三十二位的数据，所以说，我们假如有一个函数返回了一个结构体的地址，那么无形中就返回了一大堆的数据啊：

```
#include <stdio.h>
struct POINT
{
        int x;
        int y;
};

POINT *SetPoint()
{
        POINT pt;
        pt.x=3;
        pt.y=5;
        return &pt;
}
void main()
{
        POINT *ppt;
        ppt=SetPoint();
        printf("x 坐标为：%d  ， y 的坐标为:%d \n",ppt->x,ppt->y);
}
```

但如果看过牛刀上一讲教材的应该知道，其实这种返回值是很危险的，我们一般在函数中也是定义一个指针，而临时 new 出一段内存空间，然后将相应的数据返回：

```
#include <stdio.h>
struct POINT
{
        int x;
        int y;
};

POINT *SetPoint()
{
        POINT *pt;
        pt= new POINT;
        pt->x=3;
        pt->y=5;
        return pt;
}
void main()
{
        POINT *ppt;
        ppt=SetPoint();
        printf("x 坐标为：%d  ， y 的坐标为:%d \n",ppt->x,ppt->y);
}
```

是不是这样做就很安全呢？我们在函数中打印出地址来看看啊：

```
#include <stdio.h>

struct POINT
{
        int x;
        int y;
};

POINT *SetPoint()
{
        POINT *pt;
        pt= new POINT;
        printf("pt 的地址为：%p\n",pt);
        pt->x=3;
        pt->y=5;
        return pt;
}
void main()
{
        POINT *ppt;
        ppt=SetPoint();
        ppt=SetPoint();
        ppt=SetPoint();
}
```

我们看结果：

**引用:**

pt 的地址为：00431D10
pt 的地址为：00431CD0
pt 的地址为：00431C90

不错，不是同一个地址，所以就，如果看过牛刀的《防杀衣》那篇文章的话，应该对 FILE 结构体有印象。

但这里同样存在着“野指针”与“资源释放”的问题。

所以我们当定义了这种没有 delete 的函数后，必需要定义一个含有 delete 的函数，以便系统释放资源，并且不要用同一指针作多次调用，因为那样一来，好多的指针就一下子变成了“野指针”，你“抓”不回头了。

所以说，大家应该明白，为什么有 fopen 函数后，还要调用 fclose 函数吧？ fclose 函数就是要释放我们的指针，并且做好指针释放后的一些善后工作。

大家看上面的程序，我们定义了一个 POINT 指针，当我们返回这个指针时，就乘机返回了两个元素，我们可以安全地拿到这两个元素，这是一个优势吧？再来：

·指针实现了封装

封装是一个很美妙的概念，就好像数组使得我们的可以很容易地用一个循环来很容易地处理好多数据一样，封装使得我们可以有效地隔离一些不必要的数据之间的联系，看程序：

```
#include <stdio.h>
struct POINT
{
        int x;
        int y;
};

void main()
{
        POINT pt1,pt2;
        pt1.x=3;
        pt2.x=4;
        pt1.y=5;
        pt2.y=6;
        printf("pt1.x=%d\npt1.y=%d\npt2.x=%d\npt2.y=%d\n",pt1.x,pt1.y
,pt2.x,pt2.y);
}
```

大家看，如果我们不用结构体定义，则是不是要定义四个变量：

```
int x1,x2,y1,y2;
```

并且我们还要时时刻刻注意哪个变量是什么作用？而一旦用了结构体，这种关系自然地就被封装到了结构体之中。

C++中甚至还允许将函数封装进结构体，那才是真正的封装，看程序：

```
#include <stdio.h>
struct POINT
{
        int x;
        int y;
        void print()
        {
                printf("这是结构体中的函数，其中的 x 坐标为：%d\ty 坐标
为：%d\n",x,y);
        }
};
```

```
void main()
{
        POINT pt1,pt2;
        pt1.x=3;
        pt2.x=4;
        pt1.y=5;
        pt2.y=6;

        pt1.print();
        pt2.print();
}
```

执行结果如下：

**引用:**

这是结构体中的函数，其中的 x 坐标为：3　y 坐标为：5
这是结构体中的函数，其中的 x 坐标为：4　y 坐标为：6

当然这种情况我们放在以后的类与面向对象中进行详细讨论，总之，封装使得我们的结构体变成了一个整体。

·再次就是结构体给我们组织数据提供了方便。

牛刀为名人做过不少程序，而用结构体与不用结构体，组织思路上会出现很微妙的变化。

变化之一：定义数据减少
变化之二：往往功能可以直接通过结构体中的变量去定义，实现参数同步传送与变化
变化之三：思路清晰，这一点不可言寓，大家经常做程序才能体会到

下面再讲讲结构体的另一种功能——数据结构

如果与你详细地讲数据结构，本身就可以写出一本书，但牛刀的教材还是那句话，是重于实践的，也就是说，我们的结构体中也可以定义指针，甚至可以定义指向自身结构体的指针（注：这与函数中的递归是两码事，别搞混了）

```
#include <stdio.h>
struct A
{
        int i;
        A *a;
};
void main()
{
```

```
        A *pt;
        pt=new A;
        pt->i=1;
        pt->a=new A;
        pt->a->i=2;
        printf("pt->i=%d\npt->a->i=%d\n",pt->i,pt->a->i);
}
```

其实 C++中已经允许这样定义，如果你看其他书，可能有些版本上需要提前定义，就像函数原型一样先定义一个：

```
struct A;
```

然后才可以像上面一样的定义，但我们还是那句话，我们用的是 VC6.0 而这个版本允许我们不作如前面一样的定义，但如果两个结构体的递归定义可能如果你面的要引用后面的，就需要作如上处理了，但我们这是一个，我们如果想那样做的话，知道应该怎么做就行了。

注意两点：

一、上面的程序可以 new 一个，两个，甚至可以一直 new 下去，从而形成一个长长的链，我们称之为“链表”，没有结构体就没有链表。

二、上面的程序不完善，没有 delete 所以说上面的程序仅供演示参考。

既然提到了链表，我们下面就给大家开个头，我们的链表处理一般是用相应的函数来处理，而不是直接在程序中处理，再一个就是，我们的链表一般都定义一个链表头，如果你有兴趣，或者做双向链表，还要定义一个链表尾，并且还要有两个指针去处理链表中的数据以及链表的结构，也就是说，我们一般需要有三到四个指针，甚至更多，看程序：（注：我们的解释做成注释放在程序中，大家千万要读懂下面的程序：）

```
#include <stdio.h>
struct A        //链表的定义
{
        char ch;        //链表的数据，这里被牛刀改成了 char 类型
        A *a;        //链表的指针，必须，没有这个指针，形成不了下一个链节点
};

A *Cre()        //创建链表，返回链表头
{
        A *pt;
        pt=new A;        //新建一个链接点，为了以后处理链表的方便，我们一般不用头接点，但如果你使用了，那以后链表的处理就需要单独处理头接点
        pt->ch='\0';        //因为不用，所以我们赋值为 0
        return pt;        //返回链表头指针，这个链表就建立了，是一个空链表
}
A* Add(A *a,char ch)        //向链表尾增加一个数据 ch
```

```
{
        A *p=new A;
        p->ch=ch;          //注意前面的 ch 与后面的  ch 是不同的，一个是结构体中的，一个是参数传递过来的。
        p->a=NULL;
        a->a=p;                    //这里的程序有点糊涂了吧，仔细想，慢慢想，要得好画画图比较好，不过这工作让给大家了，牛刀不给做了，你自己想出来印象深刻，呵呵，以后再介绍
        return p;
}

void Lis(A *a)          //显示链表中的数据：
{
        A *p=a->a;          //请注意了，我们将头接点跳过了。
        for( ;p->a!=NULL;p=p->a)
                printf("%c->",p->ch);
        printf("%c\n",p->ch);
}
void main()
{
        A *head,*tail;          //我们为了方便，使用了尾指针
        tail=head=Cre();                    //怎么样，只需要一个函数，就定义了一个链表
        tail=Add(tail,'a');
        tail=Add(tail,'z');
        tail=Add(tail,'b');
        tail=Add(tail,'2');
        Lis(head);
}
```

**结构实践〇五四：漫游结构体**

这一讲我们展开一个全新的课题——API 编程。

怎么，不是结构体吗？是啊，我们学习是为了使用，而既然使用，我们就用最新的，最好的，还要把理说透了，你说是不？

在这里牛刀将课程计划调整一下，什么原因也很简单，链表的使用与数据结构如果我们想说的话，本身就可以写一本书，并且实践性不是很强，并且是计算机底层的东西，如果管理员有兴趣的话，专门开一个《算法》版块，专门讨论算法，我想讨论起来可能会比较火，呵呵，对了，突然发现我牛刀也是这一版块的版主了，呵呵，谢谢管理员啊，再一个就是发现牛刀变成了特邀嘉宾了，是不是权限比超级版主要大？呵呵，再次感谢管理员。

行了，我们说为什么要教大家 API 吧，因为这里就有好多的结构体，并且我们早迟要学，那为什么不现在就学呢？并且各个结构体的类型还比较全，行了，这一讲我们作为一个入门，先带领大家将我们的程序先做进电脑吧，我们接下来的工作可不是控制台应用程序了，而要专门编写 Windows 应用程序了，所以这一讲先讲一讲如何开始制作一个 Windows 程序：

**引用:**

·打开 VC6.0
·文件/新建/[工程]/Win32 Application
·工程下面的输入框中输入工程名，我的这一工程名为 WinTest
·下面一个框选择位置，就是我们的工程需要放置的位置。
·按“确定”
·在下面的选项中选择第一个，我们创建一个空的工程，点“完成”。
·再点“确定”
通过以上的步骤，我们就制作了一个空的 Windows 工程的框架，当然一定要这样做，否则可能我们做了相应的程序也会运行不起来，这是因为我们这里与控制台应用程序的主入口函数根本不同，并且一些后台的设计不同，至于哪些不同，我们暂时不管，大家以后按步就班就行了。

行了，我们再加入我们的代码编写窗口，还是按下面步骤：

**引用:**

·文件/新建/[文件]/C++ Source File
·在左边“文件”下的输入框中输入文件名，我们这里也输入：WinTest
·按“确定”，建立一个 WinTest.cpp 文件，这里的扩展名 cpp 就是指 C++ 文件，我们以前为名人做程序时，需要做 C 文件，就必须附加一个“.c”，的扩展名，这样做出的文件是 C 文件。

那么这两种文件有什么不同呢？不同点多喽，C++中加入了很多新东西，像什么重载啊，什么类啊，多喽，那我们以后逐步会与大家说到，我们今天的目的

是与大家说结构，所以不涉及 C++，所以像今天的这一课题，你建立一个 C 文件效果一样，Windows 编程不是 C++的专利，API 更不是 C++的专利，原本就是 C 语言的东西，只不过 C++也拿过来用了。

行了，接下来，我们要包含进一些东西：

包含什么呢：windows.h 头文件，这是 Windows 的基础啊，呵呵。

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
	return 0;
}
```

上面的程序已经可以顺利运行了，不过运行的结果是一点反应都没有，甚至像控制台应用程序那样的提示都没有，没办法，谁让我们现在用的是 Windows 呢？

对了，这里说一下，以后我们大家看见 main 主函数作为入口函数，就应该知道是控制台应用程序，看到 WinMain 作为入口函数，就知道是 Windows 程序，文件的制作过程就是如今天教大家所示，以后正常情况下，我们的这种制作过程就不说了，大家多上机练几遍熟悉熟悉。

行了，上面的几行代码一点意义都没有，因为程序一点反应都没有，那么怎么让我们的程序有反应呢？这就需要应用到我们的巨大的函数库——API 函数。

大家别紧张，一点一点来，我们先给大家做一个让大家看看：

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
	 MessageBox(NULL,"我跟牛刀学程序之 Windows 篇","提示信息",MB_OK);
	return 0;
}
```

上面的程序已经可以非常完美地运行了。

在这里，我们先与大家说一下，入口函数中的参数，大家先别管，写程序时照抄就行，甚至复制过去都无所谓，关于其中的内容我们到需要用到什么再给大家讲，不过大家先知道就行，HINSTANCE 是句柄，上面定义了两个句柄；LPSTR 是字符串指针，int 大家不会不知道吧？

什么叫字符串指针呢？我们看系统是怎么定义的就知道了：

```
typedef CHAR *LPSTR, *PSTR;
```

typedef 看过牛刀教材的人应该知道吧？可能你想， CHAR 是什么？也没看过啊，好，就让你看看：

```
typedef char CHAR;
```

看见了吧，和小写的一样，当然我们也可以用：

```
typedef int INT;
```

来定义一个 INT 的数据，用

```
typedef bool BOOL;
```

来定义一个布尔型数据，是一样的，这种东西在 Windows 中多了，Windows 中有一套命名法叫“匈牙利命名法”肯定是匈牙利人发明的，就是使用 LP 作为一个指针啊，还有 sz 作为字符啊等等，我们提到时再说。

我们再说说 MessageBox 函数，以后会常用，第一个参数是一个设备描述表，不懂没关系，在这里我们用 NULL 空值代替，一样的，因为我们这说实话还不是真实的 Windows 编程，呵呵。

第二个参数是显示的数据，而第三个参数其实是标题，第四个参数其实一般是用 #define 定义的一些宏，这里的 MB_OK 就表示只有一个确定钮，仅此而已。

上面这个函数在调试程序时或者在编程过程中用得特别得频繁，也是我们学的第一个 API 函数，我们通常称之为信息框，或消息框，一样的。至于有哪些按键定义，我们下面给大家一个表：

```
#define         MB_OK
       0x00000000L
#define         MB_OKCANCEL
       0x00000001L
#define         MB_ABORTRETRYIGNORE
       0x00000002L
#define         MB_YESNOCANCEL
       0x00000003L
#define         MB_YESNO
       0x00000004L
#define         MB_RETRYCANCEL
       0x00000005L
```

大家通过后面的单字一定能够知道什么意义吧，牛刀不多说了，自己去玩玩就一切明了，这些去说多没意思，呵呵。

当然还可以指定一些外观：

```
#define                   MB_ICONHAND
          0x00000010L
#define         MB_ICONQUESTION                                   0
x00000020L
#define         MB_ICONEXCLAMATION                                0
x00000030L
#define         MB_ICONASTERISK
```

0x00000040L

这些东西相互组合可以用“|”，即“或”来组合使用，很自由的，呵呵。

当然关于详细的内容，我们以后有空再与大伙讨论，我们先看看我们今天的几个结构吧：

```
HWND hwnd;          //注意，这不是结构，是句柄，说明了是一个三十二位的数据而已
MSG Msg;
WNDCLASS wndclass;
```

下面两个是结构体，看定义：

```
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
#ifdef _MAC
    DWORD       lPrivate;
#endif
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;

typedef WNDCLASSA WNDCLASS;

typedef struct tagWNDCLASSA {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCSTR      lpszMenuName;
    LPCSTR      lpszClassName;
} WNDCLASSA, *PWNDCLASSA, NEAR *NPWNDCLASSA, FAR *LPWNDCLASSA;
```

可能你会问，怎么搞得那么复杂，定义那么多个？还有什么不同的？

说实话，说来话长，不得不如此，就包括上面的 MessageBox 函数也不是一个，为什么呢？为了兼容，不但是向前兼容，而且需要向后兼容。

大家知道我们在西文程序时，字符大小写 52 个就够了，再加上符号与其他的格式定义，也就 128 个，后来又扩展了一些表格符号，最后成了 256 个字符，对于美国佬来说，够用了。

而一旦用到了其他的国家，特别是中国，这种八字节的数据就远远不够了，于是形成了汉字国标码，现在进入了 XP 时代，又形成了 Unicode 码，这种种码，使得原来的函数没办法处理了，你想处理一个西文字符行啊，那你如何显示一个汉字呢？当然有两套方案，一套方案就是重新定义一套处理汉字的函数，当然这套方案显然是不合理的，于是第二套方案，就是兼容函数，兼容结构，兼容……

什么都是兼容的，怎么办呢？函数或相应的名称后面加一个“A”，来表示宽字符，也就是这个函数同时处理两个字节。

问题又来了，那么既然一个函数又能处理一个字节，又能处理两个字节，那么我们编程岂不是很不方便，于是这个任务交给了系统，于是我们看这些原代码时，一些定义显得千奇百怪，不过既然我们知道是千奇百怪了，也就不奇怪了，下面我们就一个一个结构体来看：

```
char lpszClassName[]="窗口";
char lpszTitle[]="My_Windows";
wndclass.style=0;
wndclass.lpfnWndProc=WndProc;
wndclass.cbClsExtra=0;
wndclass.cbWndExtra=0;
wndclass.hInstance=hInstance;
wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
wndclass.hbrBackground=(HBRUSH)GetStockObject(BLACK_BRUSH);
wndclass.lpszMenuName=NULL;
wndclass.lpszClassName=lpszClassName;
```

上面其实就是对结构体中的一些数据的赋值，如果明白这一点，上面的东西就一目了然了，那么我们要说的是什么呢？首先是 UINT 其实这就是无符号整型，一定是用 typedef 重新定义过的，Windows 就喜欢玩这种花活，呵呵，int 类型的那两个数据一般赋值为 0，暂时我们用不到。大家看前面带 H 的，都是一些句柄，其中不同的句柄用了不同的赋值方法， hInstance 我们称之为 实例句柄，这个句柄表明了我们当前的程序，所以当我们使用 WinMain 函数时，函数中就传了进来了，我们直接将那个值赋给结构体就一切 OK，而 HICON 是图标句柄，这里用 LoadIcon 函数调用了一个系统的图标，同理，光标也用 LoadCursor 调用了系统的一个光标 HBRUSH 是一个画刷，在这里是背景画刷，牛刀这里用了 BLACK_BRUSH 如果运行起来你一定感觉别扭，背景是黑的，下面给大家完整程序时，我们改成白的，注意了，画刷用的函数前面一定要加强制类型转换：(HBRUSH) 原因是 GetStockObject 这个函数太通用了，关于这些函数，我们以后在给大家讲绘图时再详细讲解。

大家看，LPCSTR 是一个字符指针，如果你很有心，你可能纳闷，字符串不是不能直接赋值的吗？也就是说必须要用函数啊!!! 是的啊，但那是 C 语言，我们有这规定，但在 C ＋＋中，一切都变了，因为 C ＋＋中有函数的重载，运算符的重载，一旦重载了，这功能就显得分外地强劲，至于怎么个强劲法，我们给大家聊到那再说，也就是说，在这里，我们也可以用等号，没问题是了。（注：

以后学到 MFC，我们字符串的连接，可以直接用加号“＋”连接，十分方便，当然那是后话，我们暂且不表）

这里最值得一提的就是 WNDPROC 这就是一个函数的指针，我们赋过去的就是函数名，当然，我们下面就必须为这个函数写代码，如果你看了牛刀上面《指针与函数》你就会觉得这种引用分外地亲切。好了，我们下面就说说这个函数的结构吧：

LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam);

函数必须是这种结构，名称可以变，其中变量名称也可以变，但不能变的就是每一个变量的变量类型。这就是函数指针所必须的要求，当然喽，这个函数我们等一下再详细讨论。我们再继续看上面的结构体：

菜单指向 NULL 没有菜单，什么叫菜单大家不陌生吧？都用计算机这么长时间了，呵呵。行了，我们继续下面的程序:

```
if(!RegisterClass(&wndclass))
{
        MessageBeep(0);
        return FALSE;
}
```

这里的 RegisterClass 是向系统注册窗体，什么原因呢？因为我们的系统与 DOS 系统不同，我们用的系统为多任务系统，所以我们在写东西时可以同时听音乐，我们上面与大家说过，我们有一些数据定义中有 H 为句柄，为什么要这么多句柄呢？其实每一个句柄都是一个资源，都需要系统进行合理的分配的，其为一个 3 2 位数据，也就是说，我们 Windows 系统能管理的资源量最多是 2^32=4294967296 种资源，很有限吧？不过绝对够用了，呵呵，远远用不完，所以说，我们每一个句柄，都需要让系统知道有这个句柄，那系统怎么知道呢？就需要我们程序在运行之前注册，当然喽，这里的注册一般是不会失败的，但我们为了程序的强壮性，我们有必要将程序写成上面的形式，中间一个函数 MessageBeep 我想大伙一定能够明白什么意思吧？报警，不详细讨论了，你一定明白了。

```
hwnd=CreateWindow(lpszClassName,lpszTitle,WS_OVERLAPPEDWINDOW,0,0,960,704,NULL,NULL,hInstance,NULL);
        ShowWindow(hwnd,nCmdShow);
        UpdateWindow(hwnd);
```

上面三个函数，第一个函数比较复杂，返回一个句柄，其实是得到一个窗口句柄，什么意思呢？我们上面的赋值，仅仅将我们的想法告诉了一个结构体，这个结构体本身就是一堆的数据而已，而注意结构体，是告诉系统，我需要用这一堆数据来产生一个窗体，而 CreateWindow 则是产生这个窗体，我们看其中的参数吧：

第一个参数 窗口类名，就是我们上面那个名子，为什么要这个名子呢？因为这个名子的窗口，我们已经向系统注册了，系统中已经有这个名子的窗口存在

了，所以我们产生的窗口就必须是这个名字的窗口，当然你可能会想，不是有结构体吗？还要名子干嘛？是有结构体啊，那个结构体一旦注册了窗口，她的使命也完成了啊，结构体是一种数据结构，而窗口是一种系统的资源，两码事。行了我们下面再看：

第二个参数　窗口标题名　知道吧，不知道等我们程序能运行了，一运行就知道了，我就不信，能在电脑上看牛刀教程的人，什么叫窗口，什么叫窗口标题会不知道，呵呵。

第三个参数　窗口样式，就是有几个按钮啊，（例如有没有最大化按钮啊，有没有系统图标啊，等等，举些常用的样式给大家吧：

**引用:**

WS_BORDER
WS_CAPTION
WS_VICROLL
WS_MAXIMIZEBOX
WS_MAXIMIZE
WS_MINIMIZEBOX
WS_MINIMIZE
WS_OVERLAPPED
WS_OVERLAPPEDWINDOW
WS_POPUP
WS_POPUPWINDOW
WS_SYSMENU
WS_HSCROLL

上面的是牛刀一个一个敲进电脑的，不保证正确性，如果有错误，大家就去按　F1，找帮助吧，呵呵，学 VC 一定要装帮助噢。

再下面就是窗口左上角坐标与窗口的窗度与高度，当然如果你不想自己定义也可以使用　CW_USEDEFAULT　来取系统默认值，一样的。

再下面就是窗口句柄啊，菜单句柄啊，实例句柄啊，等等，一般情况下，大家就直接堵：NULL,NULL,hInstance,NULL　就行了，牛刀不着过多的解释，因为你以后真正的编程，绝对不会这样一个一个地将这些东西敲进电脑的，一般都是复制/粘贴，呵呵，真的，如果你以后编程序达到一定的水准了，你会发现你其实并不是在编程序，而是一直在“改”程序。

显示窗口：

```
ShowWindow(hwnd,nCmdshow);
```

前面也是窗口句柄，而后面是窗口的状态，如：

**引用:**

SW_HIDE

SW_SHOWNORMAL
SW_SHOWMINIMIZE
SW_SHOWMAXMIZE
SW_SHOWNOACTIVE
SW_RESTORE
而 UpdateWindow 听说可以发出 WM_PAINT 消息，重画显示区域，大家这一讲学好后，你可以将这一句删了看有什么不同，(其实每一句你都可以删了试试，实践嘛，就是应该将每一句都搞明白)。

```
while (GetMessage(&Msg,NULL,0,0))
{
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
}
```

消息循环，记得牛刀前面给大家写过类似于：

```
while((i=k())!=0)
{
        s(i);
}
```

这样的函数吗？

其实是一样，但大家知道为什么上面有两个函数呢？不，三个，括号中还有一个，呵呵，我们一个一个来看啊：

GetMessage 显然好像是得到消息，得到的消息就放在 Msg 中，我们知道 Msg 是一个结构体，其实其值就通过这个函数给传了出来，然后再传进下面的函数中，既然拿到了 Msg,就可以再次使用了。

TranslateMessage 对比我们下面的一段程序，这一函数好像是多出来的，为什么要有这一函数呢？因为我们下面一个函数：DispatchMessage 是分发消息的，而我们分发的消息中却没有字符消息，什么意思呢？我们来个慢镜头啊：

当你轻轻地按下键盘上的一个“A”键，电脑会通过上面的 GetMessage 得到一个 WM_KEYDOWN 消息，当你抬起这个键时，则产生一个 WM_KEYUP 消息，你是不是按过了这一个键了？

不错，那么你的操作仅仅是按了一下又松开，这两个操作，但消息中却没有字符消息 WM_CHAR 而这一个消息是怎么来的呢？就是通过 TranslateMessage 来的，我们就先称之为翻译消息吧。

当然同理，在这里也可以加进我们自己的消息处理，一样的，我们可以自己定义一个消息，通过这儿再定义一个什么函数发出去，当然你要定义一个发送消息的函数，则必须再定义一个接收的函数，不然这种消息没用的，举个例子，你打手机给别人，但你打的那一边必须有人是不？你老拨空号什么意思呢？当然这里的改进，如果你想玩，你就自己去做吧，但牛刀要说的，这里的函数告诉我们，这些消息处理其实是系统给我们做了，所以以后大家调试程序时，一定不要在这

里设断点，因为这是一个“死循环”，并且不会有任何反应，因为这一切，都是系统给我们完成了，至于怎么完成的，那是操作系统的事了，呵呵。

最后返回的是消息中的一个数据：
return Msg.wParam;
什么意思，我们下面再讲。

讲到这里，这个程序已经算是完了，但程序一调试就报错，为什么呢？因为我们指针指向的一个函数我们还没定义呢？注意，如果这个函数定义在后面，则前面一定要跟函数原型，否则程序就找不到函数，我们接下来看这一段消息处理函数：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
	switch(message)
	{
	case WM_DESTROY:
		PostQuitMessage(0);
	default:
		return DefWindowProc(hwnd,message,wParam,lParam);
	}
	return (0);
}
```

相比而言，这一段函数就显得短小多了，但内容却并不短小，我们以后的编程任务，主要就是这一段函数，所以牛刀准备将这一段函数挪到下一讲，大家按照一讲，一个一个输上电脑，已经是一个非常优秀的程序了，可以给你画出一个窗口，大家试着玩玩，这一讲着重讲了一个结构体 WNDCLASS ，我们下一讲开始，我们还会遇到更多的结构体，也正因为有了这些结构体，我们的软件才能够被做得活灵活现，就先讲到这儿了，上面的程序，请大家不要用复制/粘贴的方法，一个字符一个字符地敲进电脑，牛刀也是这样一个一个地敲出来的，希望大家别偷懒。

## 结构实践〇五五：常用结构体

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{
        HWND hwnd;
        MSG Msg;
        WNDCLASS wndclass;
        char lpszClassName[]="窗口";
        char lpszTitle[]="牛刀教我做自己的窗口";
        wndclass.style=0;
        wndclass.lpfnWndProc=WndProc;
        wndclass.cbClsExtra=0;
        wndclass.cbWndExtra=0;
        wndclass.hInstance=hInstance;
        wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION);
        wndclass.hCursor=LoadCursor(NULL,IDC_ARROW);
        wndclass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName=NULL;
        wndclass.lpszClassName=lpszClassName;
        if(!RegisterClass(&wndclass))
        {
                MessageBeep(0);
                return FALSE;
        }
        hwnd=CreateWindow(lpszClassName,lpszTitle,WS_OVERLAPPEDWINDOW
,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,hI
nstance,NULL);
        ShowWindow(hwnd,nCmdShow);
        UpdateWindow(hwnd);
        while (GetMessage(&Msg,NULL,0,0))
        {
                TranslateMessage(&Msg);
                DispatchMessage(&Msg);
        }
        return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        switch(message)
```

```
        {
        case WM_DESTROY:
                PostQuitMessage(0);
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

上面的代码有些被牛刀改过了，大家做程序时，一般正常的情况下，都是将上面的代码复制过去就一切 OK 喽，事实也是如此，可以说以上的代码一点用都没有，也可以说是鸡肋，说其没有用，并不是说没必要，没有上面的程序可以说是什么也玩不转，可是有了上面的程序吧，从来没有哪个程序员会去敲进电脑，基本上都是复制/粘贴过去了，而说实话也好玩，基本上我们等到程序做到一定时候之后，我们基本上是在“改程序”，而不是在“写程序”，所以牛刀对大家有以下几点建议：

一、写程序时，合理使用空格，空行，如：

```
for( int i=0 ; i < 100 ; i++)...
```

二、写程序时，以下结构一定要使用如下结构（大括号）

```
for(int i=0; i<100;i++)
{
}
if(...)
{
}

if(...)
{
}
else
{
}
```

为什么牛刀对大家作如上的“规定”呢？这是牛刀对编程环境的一种总结，牛刀使用的是 VC6.0，而 VC6.0 做得相当优秀，当一写一个程序时，基本上可以同时对程序进行缩进，但可惜的就是当没有大括号时，对于一行的缩进会出现一些异常，举个例子吧：

```
for(i=0;i<9;i++)
        for(j=0;j<9;j++)
                printf("...");
        ...
```

上面的缩进大家看有问题吗？如果没有问题的话，应该是如下的缩进：

```
for(i=0;i<9;i++)
        for(j=0;j<9;j++)
                printf("...");
```

...

但由于我们是编程环境制作时没有考虑到这种特殊的情况吧，于是形成了这种缩进不如我想像的情况，而如果使用下面结构就不会出现问题了：

```
for(i=0;i<9;i++)
{
        for(j=0;j<9;j++)
        {
                printf("...");
        }
}
```

...

牛刀为什么讲这些细节的东西呢？因为我们的系统有一个相当优秀的功能 Alt+F8　可以自动排版，但如果我们不用大括号的话，自动排版的结果就是上面一种结果，这种程序我们看起来很不舒服，并且在读程序时容易引起歧义，我们原来做程序同一缩进的内容，我们可能感觉上是一个程序块，而一旦不按我们想像那样的话，那么很容易让我们看起来误解，你说上面的缩进方式，我们会认为后面的程序是上面 for 循环的一部分吧，这些大家自己编程时细细地去体会体会就知道了。

三、大括号的左括号与右括号必须同时敲出也就是说，我们写如下程序时：

```
if(a>b)
{
        t=a;
        a=b;
        b=t;
}
```

我们敲时，应该先这样：

```
if(a>b)
{
}
```

然后按向上键，将其余代码敲进去。

这种顺序与我们写作顺序不一样，因为我们在写东西时，是自上而下，特别如果我们在纸上写东西时，不可能将某一部分先写下来，然后将纸拉长，那不现实，但在电脑上写就不同了，电脑允许我们后换行。

为什么牛刀对大伙作上面的规定呢？因为如果我们的程序很长，很大，那很容易出现大括号不配对的情况，而如果按牛刀这种写法，永远不可以出现问题。

事实就是这样，大家到任何一本书上，都不会给你作出上面种种规定，但如果我们想我们的程序结构好的话，建议大家还是遵循上面原则，并且上面的原则也符合我们上面提到过的，我们做程序基本上是“改程序”的规律，当然程序的主要功能还是我们做出来的喽。

所以牛刀以前曾经告诉过大家，面对一个程序，千万不要将一个程序一行一行，一字不漏地敲进电脑，而是要将程序理解，消化，做程序时，则完全是按上面的原则，来一段一段的代码去改写。

大家学过 C 语言的都知道，有空函数，空语句吧？为什么要有这些语句与函数呢？其实这也告诉我们，其实我们做的每一个语句的开始都是空语句，每一个函数的开始其实都是空函数，而我们做程序的过程，就是去一步一步地改写，一步一步地实现我们程序的功能。

四、合理的语句注释也函数注释。

关于注释，牛刀不想多说了，一个程序如果是写给人看的，就必要要有注释。给大家一段程序（这段程序有点恶心了，呵呵）：

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow){ HWND hwnd; MSG Msg; WNDCLASS wndclass; char lpszClassName[]="窗口"; char lpszTitle[]="牛刀教我做自己的窗口"; wndclass.style=0; wndclass.lpfnWndProc=WndProc; wndclass.cbClsExtra=0; wndclass.cbWndExtra=0; wndclass.hInstance=hInstance; wndclass.hIcon=LoadIcon(NULL,IDI_APPLICATION); wndclass.hCursor=LoadCursor(NULL,IDC_ARROW); wndclass.hbrBackground=(HBRUSH)GetStockObject(WHITE_BRUSH); wndclass.lpszMenuName=NULL; wndclass.lpszClassName=lpszClassName; if(!RegisterClass(&wndclass)) { MessageBeep(0); return FALSE; } hwnd=CreateWindow(lpszClassName,lpszTitle,WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,hInstance,NULL); ShowWindow(hwnd,nCmdShow); UpdateWindow(hwnd); while(GetMessage(&Msg,NULL,0,0)) { TranslateMessage(&Msg); DispatchMessage(&Msg); } return Msg.wParam;}LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam){ switch(message) { case WM_DESTROY: PostQuitMessage(0); default: return DefWindowProc(hwnd,message,wParam,lParam); } return (0);}
```

上面的程序与前面程序一模一样，但没有换行，没有注释（当然牛刀没有做更恶心的事，大家可以试试再将空格全拿了，看你还有精神去读这一段程序吗？

但告诉大家一个事实，这一段程序，拿到电脑中，一样的运行，也不报错，结果全都一样，就是我们看上去不舒服了，仅此而已。

所以牛刀以上的一些实践事实告诉我们，我们其实可以编像上面那样，连个换行都没有的程序，可是计算机不累，人累，你也累，看的人更累。

而牛刀看有人为名人做的例程中的注释就让人刮目相看：

```
/*############################################################################
######
  Copyright (c) 2004, MEIJIN COMPUTER TECH LTD.
  All rights reserved.

  File name     :Diary.c
  Description   :日记本

  Version       :1.00
  Author        :黄猛
  Complete Date :2004-08-XX

  Function List :
  History       :
  ------------------------------------------------------------------------
______
  DATE          AUTHOR    VERSION    DESCRIBE
  ------------------------------------------------------------------------
------
  2004-08-15    黄  猛    Ver1.00    Create
  ------------------------------------------------------------------------
------
  2004-09-21    黄  猛    Ver1.01    怎加了日记查找的内容查找方式，改进了心
                                      跳回忆的部分算法。
  ------------------------------------------------------------------------
------
############################################################################
####*/



/*初始化##########################################################################初始化*/
#include        "Diary.h"
/*程序模式定义*/
#define AP_MODE_CURVE           (EV_APPLICATION+11)                 /*心跳回忆模式*/

/*日记本的相关常量*/
```

```
#define                 INDEX_NUM               1               /*数
据库的索引数*/
#define                 FIELD_NUM               4               /*多
域编辑器域数，数据库的字段数*/
#define                 KEYFIELD                0               /*数
据库的关键字段号*/
#define                 ROWSPACE                2               /*多
域编辑器的行间距*/
#define                 INDENT                          0
 /*多域编辑器缩进大小*/
#define                 SPIRIT_NUM                5               /*
心情域的选项数*/
#define                 WEATHER_NUM               5               /*
天气域的选项数**/

#define                 MAX_SEARCHLEN       64              /*按内
容查找时查找缓冲区长度*/
```

...

不复制过多的过来了，大家看这段程序舒服吧？牛刀感觉也舒服，这是日记本中的注释，人家每个函数前面都有详细的注释，特别适合于学习。

当然，这也是我们要学习的。

行了，这一讲是不是大家会嫌我有点啰唆？其实我们都在成长，我们学习的只是书本知识，而我们实践了，并且掌握了，那才是我们的财富。

再说我们上面的程序吧，我们看到了 WinMain 函数，而我们却从来不写 WinMain 函数，所以说，MFC 将这个入口函数都封装了，看不到了，封装的对啊，我们以后做程序也是将上面一段程序一复制，到我们的环境中一编译，没问题了，下面才开始我们的编程之旅，那么第一站是哪里呢？

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        switch(message)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

上面这个程序，没必要分开来讲了，这里告诉大家，WndProc 名称可以改

变，只需要与 WinMain 中结构体中相应的赋值对应就行，要讲的是里的几个参数：
HWND hwnd　这是句柄，也就是说，参数带过来了一个句柄，也叫窗口句柄，因为我们以下的操作都是要对这个窗口进行操作的，所以我们必须要有这个句柄，否则，程序怎么知道我们下面要对什么地方进行操作呢？你说是吧？

UINT message 一看定义就知道，这是一个无符号整数，看参数知道，传进的是消息，我们在这个函数中，主要是对消息进行处理，处理的过程在 WinMain 中的消息循环中，而处理的消息就由我们这一函数来承担。

WPARAM wParam,LPARAM lParam　这两个参数被弄得哭笑不得，这也是兼容带来的毛病，其实一个是十六位参数，一个是三十二位参数，所以一个叫前面附加一个长（L）与 Word(W)类型，不过现在都是三十二位了，但还是为了兼容，或者是这两个参数太有名了吧，流传下来了，由于现在系统都承认了，所以我们不这样写也不行。

可不要小看这两个参数噢，这两个参数可是神通广大，我们接下面的讨论主要是对这几个参数而言，当然，我们要讲好多，也就是说，不是一句两句就讲得完的，所以说，还请静静听来：

关于前面的　LRESULT CALLBACK ，大家忘文生意吧，总之只需要这样写就行了，我们不需要什么东西的来龙去脉都搞得十分清楚，如果我们等什么都学完了，再来编程，那我们要等到什么时候再去写　main　函数啊？大家说是吧？

基于程序中的就好理解了，一个 switch 分支结构，主要就是处理消息，大家看，其中有一个 default:，下面是　return
DefWindowProc(hwnd,message,wParam,lParam);
参数不用说了，写我们做的函数一模一样，也就是说，我们不处理的消息都送给这个函数去处理，至于怎么处理，那我们就管不着了。

我们再来看我们要处理的消息。

我们上面只处理了一个消息：WM_DESTORY　，这其实是一个当我们按下关闭时，系统需要捕捉的一个消息，其实如果没有这一消息，系统也一样地关闭，但为什么要有呢？大家用记事本时，如果内容改了，一定会出现一个对话框吧，问要不要存盘？哎，就是这个消息要管的事，这个消息可以截获我们的关闭信息，也就是说，如果有这个消息，就可以实现不关闭，当然我们可以做个试验，还在其中做个消息框吧：（注意啊，大家看这里没有　break　啊，其实应该有的，但一般正常情况下一个窗口既然关了，下面也就什么都运行不到了，但作为一个好习惯，建议大家还是加上　break;　，不要使用一些技巧去精简我们的代码，否则一旦出问题，我们都不知道出在哪里的，呵呵）
```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
```

```
        switch(message)
        {
        case WM_DESTROY:
                MessageBox(hwnd,                    //注意，这里有句柄了，
就要用啊，呵呵
                        "牛刀不让你关",
                        ":'(",
                        MB_OK);
                break;
//                PostQuitMessage(0);
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

大家去实践一下，发现牛刀上面的解释错了，这个 WM_DESTORY 是窗体关闭之后，大家再去试试 WM_CLOSE 吧，自己去试，牛刀在这不做例子了，反正这些东西自己试一下就知道是怎么回事了，就像上面一样，牛刀说真的还真不知道 WM_DESTORY 消息是在 WM_CLOSE 消息之后发出的，但 PostQuitMessage(0) 的确是发出一条 WM_QUIT 消息，大家也可以去捕捉这个消息试试，这些讨论不在我们这一讲的范围之内，这一讲主要讲一个消息：WM_PAINT 消息：

对了，说一下，上面经牛刀实践，顺序应该是 WM_CLOSE WM_QUIT WM_DESTROY 而 PostQuitMessage(0) 是发出的 WM_QUIT 消息，所以放在 WM_DESTROY 中处理是没有任何意义的，但如果你看有些书上，甚至电脑为我们自动生成的程序中也是这么使用的，如果让牛刀来说，那是错的，所以说，尽信书则不如无书，就是这个道理，但我们一般程序中不将其作为一个错误，大家也大可不必去深究这些东西，因为有些东西已经是被实践证明了是无所谓的，何必深究？

这一样原来是要讲常用结构体的，没想这事情要放在最后来交待，Windows 就是烦，没办法。谁让我们要去实践的呢？

好了，加一个消息处理 WM_PAINT 请注意，这个消息其实是“画图”消息，就是在窗体上画东西，那么既然要得到得体的一些信息，我们就需要知道一些结构体：

**引用:**

PAINTSTRUCT
POINT
RECT

有时候忘文生意还就是好，如果你在 VC6.0 中，那么你就可以去查查这些结构体是怎么定义的，方法是在相应的结构体上单击鼠标右键，选择： Go To Definition of ...

一般情况下，就可以找到相应的定义，牛刀对一些结构体定义部分，也是通

过这种方法找给大家的。

有一句话叫：授之渔而不授之鱼，其实就是这个道理，牛刀将一些实践的方法告诉大家可能比一个一个地东西介绍给大家要好得多。

好了，给大家一段程序，大家将之敲进电脑，到下一讲，我们将详细地向大家介绍并带领大家实践一下“消息循环”。

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        RECT rt;
        POINT pt;
        switch(message)
        {
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);          //这一句必须，得到设备描述表
                {                    //这一个括号是牛刀加的，牛刀喜欢，呵呵
                         GetClientRect(hwnd,&rt);          //这一句这一讲暂时用不到，但先放在这里，这可是一个结构体哟，当然这里也可省去，一样的。
                        pt.x=20;
                        pt.y=20;
                        TextOut(hdc,pt.x,pt.y,"我跟牛刀学做 Windows 程序",23);
                }                    //这一个括号也是牛刀加的，牛刀喜欢，呵呵
                EndPaint(hwnd,&ps);                                  //这一句必须，释放设备描述表
                break;
        case WM_CLOSE:
                PostQuitMessage(0);
                break;          //注意，牛刀将这一句加了上来，大家以后最好也加
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

注意，上面的例子有点牵强，但也很能说明问题，牛刀做的这程序与其他不同之处在 BeginePaint 与 EndPaint 之间加了一个花括号，有没有都无所谓，大家喜欢就加，不喜欢不加也没关系，呵呵，

GetClientRect 是返回窗体的区域（矩形区域），而下面 pt.x pt.y 分别赋了

值，然后使用，虽然有点脱库放屁的意味，但也是为了教大家如何运用吧，但一旦我们以后与下面的鼠标啊什么的结合起来，这就不是多此一举了，不信咱们以一个例子作为结尾（注：不做那么多注释了）

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
	HDC hdc;
	PAINTSTRUCT ps;
	RECT rt;
	static POINT pt;
	switch(message)
	{
	case WM_CREATE:
		pt.x=20;
		pt.y=20;
		break;
	case WM_MOUSEMOVE:
		pt.x=LOWORD(lParam);
		pt.y=HIWORD(lParam);
		InvalidateRect(hwnd,NULL,TRUE);
		break;
	case WM_PAINT:
		hdc=BeginPaint(hwnd,&ps);
		{
			GetClientRect(hwnd,&rt);
			TextOut(hdc,pt.x,pt.y,"我跟牛刀学做 Windows 程序",23);
		}
		EndPaint(hwnd,&ps);
		break;
	case WM_CLOSE:
		PostQuitMessage(0);
		break;
	default:
		return DefWindowProc(hwnd,message,wParam,lParam);
	}
	return (0);
}
```

大家自己去理解理解，我们一下讲再为大家详细地讲解。一下子抛出这么多的消息，大家承受得了吧？能够望文生意吗？下讲见，呵呵。

**结构实践〇五六：结构与循环**

这一讲一开始，我们又要做控制台程序了，跳入跳出，大家要习惯啊，我们说过，DOS 与 WDINWOS 其实各有千秋，我们大家的思路也要跟着跳入跳出啊：

```
#include <stdio.h>

int k(void);
void s(int i);
void main()
{
        int i;
        while((i=k())!=0)
        {
                s(i);
        }
        printf("你输入为 0，退出系统 OK\n");
}

int k(void)
{
        int i;
        printf("请输入数据（0~9）\n");
        scanf("%d",&i);
        return i;
}
void s(int i)
{
        printf("你输入的数据为:%d\n",i);
}
```

上面的程序大家知道吧，其实上面的就是一个消息循环，这是以前牛刀为大家做过的一个程序，然而我们还需要将其进行一些扩展，让大家真正地明白消息循环中的真正意义，大家还知道我们说我们消息最后使用一个函数：DefWindowProc 吗？我们也来做一个：

```
#include <stdio.h>
int k(void);
void s(int i);
void defs(int i);
void main()
{
        int i;
        while((i=k())!=0)
        {
                s(i);
        }
```

```
        printf("你输入为 0，退出系统 OK\n");
}
int k(void)
{
        int i;
        printf("请输入数据（0~9）\n");
        scanf("%d",&i);
        return i;
}
void s(int i)
{
        switch(i)
        {
        case 1:
                printf("用户调用第  %d 号消息\n",i);
                break;
        case 5:
                printf("用户调用第  %d 号消息\n",i);
                break;
                //用户只要调用，则阻止了系统调用这一消息，而只要用户没有调用，消息则会自动由系统处理。
        default:
                defs(i);//系统处理函数
        }
}

void defs(int i)
{
        switch(i)
        {
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
                printf("系统默认调用第  %d 号消息\n",i);
        }
}
```

我们系统函数中，对所有的消息都进行了处理（defs 函数），而在我们消息处理函数中（s 函数），只要我们函数处理了的消息，系统就不处理了，而我们

一旦处理某一个消息，则系统就没有权处理这种消息了，什么意思呢？也就是说，我们函数中没必要对所有消息进行处理，我们在消息循环中，只处理我们感兴趣的消息就一切 OK 了。

行了，还到我们的 Windows 函数中来吧，上一讲我们讲了三个消息：WM_CLOSE WM_QUIT WM_DESTROY ，这三个消息是按顺序进行的，但 WM_QUIT 这一个消息不是由系统发出的，是由函数：PostQuitMessage(0) 发出的，也就是说，如果我们用户没有对 WM_CLOSE 函数作处理，则系统自动调用 PostQuitMessage(0) 函数，则我们的系统就发出一条消息 WM_QUIT ，而大家还记得我们前面的消息循环吗？

```
while (GetMessage(&Msg,NULL,0,0))
{
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
}
```

上面的消息循环中，GetMessage 函数我们说是一个“死循环”，其实不是，这一函数的返回值就在接收到 WM_QUIT 消息时返回 0 值，其他消息均不返回零值，所以说，我们在程序中，必须有 WM_QUIT 的消息发出来，也就是说，如果我们一旦由用户将 WM_CLOSE 消息处理了，那很有可能我们程序就关不了了：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        switch(message)
        {
        case WM_CLOSE:
                MessageBox(hwnd,"牛刀不让你关，呵呵","请注意，关不掉，用资源管理器关吧!!! ",MB_OK);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

牛刀 WinMain 函数没必要给出了，都一样，你想改标题或者是什么，自己改啊，呵呵。

一旦上面的程序大家在电脑上实现了，程序就真的关不了了，必须使用资源管理器，去结束进程，建议大家不要做这种程序，一定要让你的程序能够结束，还是那句话，让我们的程序是“伪死循环”，而不能真的成了一个“死循环”。

行了，WM_CLOSE WM_QUIT 消息，我们说过了，我们再说说 WM_DESTROY 与 WM_CREATE 消息。这两个消息很特殊，每一个消息在每一个窗体运行中，每个消息只产生一次，WM_CREATE 消息为窗口还没产生，

即创建窗口时的消息，而 WM_DESTROY 为窗口结束了，关闭后的消息，所以牛刀说过，……

对了，牛刀曾经将 WM_DESTROY 中的 PostQuitMessage 消息给拿掉过啊，结果是什么呢？结果是窗体的确不见了，消息循环却没退出，也就是说，如果你有心的话，去资源管理器中找找，还可以找到相应的进程，也就是说，WM_CLOSE 消息是关闭窗体的消息，但没有退出循环，如果一旦执行了 WM_CLOSE 就准备关闭窗体，但窗体能不能完，应该是 WM_DESTROY 的事，而退出消息循环，应该是 WM_QUIT 的事，牛刀在这里也就是猜想啊，大家可以将这三个消息捕获（也就是写出来，我们说过，消息写进代码，就由我们程序执行了，而如果我们没处理相应的消息，则相应消息由系统统一处理。）

这些东西必须慢慢品味，细细咀嚼。才会越来越有味道，呵呵。

当然我们说过，这些消息其实和牛刀一开始给大伙的那段模拟的程序一样，只不过其将数据改成了“宏”，我们可以看一些定义：

```
#define WM_SETFOCUS                     0x0007
#define WM_KILLFOCUS                    0x0008
#define WM_ENABLE                       0x000A
#define WM_SETREDRAW                    0x000B
#define WM_SETTEXT                      0x000C
#define WM_GETTEXT                      0x000D
#define WM_GETTEXTLENGTH                0x000E
#define WM_PAINT                        0x000F
#define WM_CLOSE                        0x0010
#define WM_QUERYENDSESSION              0x0011
#define WM_QUIT                         0x0012
#define WM_QUERYOPEN                    0x0013
#define WM_ERASEBKGND                   0x0014
#define WM_SYSCOLORCHANGE               0x0015
#define WM_ENDSESSION                   0x0016
#define WM_SHOWWINDOW                   0x0018
#define WM_WININICHANGE                 0x001A
```

是吧，仅仅是三十二位的无符号数据而已，大家看程序千万不要将程序看得神秘了，一旦你将其看神秘了，有好多的东西你就不敢碰，或者说不敢有自己的想法和看法，真的，牛刀也看过关于一些消息的书本解释，复制一段给大伙看看吧：

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

WM_DESTROY 讯息

WM_DESTROY 讯息是另一个重要讯息。这一个讯息指示，Windows 正在根据使用者的指示关闭视窗。该讯息是使用者单击 Close 按钮或者在程式的系统功能表上选择 Close 时发生的（在本章的後面，我们将详细讨论 WM_DESTROY 讯息是如何生效的）。

HELLOWIN 通过呼叫 PostQuitMessage 以标准方式回应 WM_DESTROY 讯息：

PostQuitMessage (0) ;
该函式在程式的讯息伫列中插入一个 WM_QUIT 讯息。前面提到过，GetMessage 对於除了 WM_QUIT 之外的从讯息伫列中取出的所有讯息都传回非 0 值。而当 GetMessage 得到一个 WM_QUIT 讯息时，它传回 0。这将导致 WinMain 退出讯息回圈，并终止程式。然後程式执行下面的叙述：

return msg.wParam ;
结构的 wParam 栏位是传递给 PostQuitMessage 函式的值（通常是 0）。然後 return 叙述将退出 WinMain 并终止程式。
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

你看，人家称消息为“讯信”，这本书可能不是大陆版的，有好多翻译得牛刀感觉还不太适应，呵呵，其实看这些，还真不如动手去实践，实践后有自己的理解，那就行了，我们的目的是做程序，而不是去详细地了解一切，再说我们也了解不了，说实话，API 函数就是三千多个，你能去将其全背下来吗？参数能都不用错吗？显然不会，当然我们在用这些东西时，最好能有一本参考书之类的东西，不知道的东西查一查就行了呗，事实就是如此，因为别人为我们做了一个很强大的“库”，而我们的目的就是“合理使用”。

牛刀当然不服输，做了几个程序继续测试：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        switch(message)
        {
        case WM_QUIT:
                MessageBox(hwnd,"我要退出了","退出测试",MB_OK);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

上面的程序运行后，一点反应都没有，事实证明，WM_QUIT 消息一旦发出，窗体是肯定没有了，但进程还在资源管理器中。同样的如果我们对 WM_CLOSE 消息捕捉，会发现窗体跟本关不了，证明排序来说 WM_CLOSE 肯定在 WM_QUIT 之前，我们再看看 WM_DESTROY 其中的消息框还可以显示，如此我们可以确定一个事实，这三个消息的顺序为：
WM_CLOSE WM_DESTROY WM_QUIT
这证明人家原来书上的一些结论没有错，是牛刀错了
再一个就是说，WM_QUIT 消息没办法捕捉，必须由系统调用，功能是结束进程，也就是说，我们在编程中不可以对 WM_QUIT 消息编程，当然如果我们知道其执行机理了，也可以进行编程了，事情不能一概而论。

第三就是 WM_CLOSE 消息完了后，WM_DESTROY 之前就销毁窗体，如果我们要在窗体销毁之前做事情，就需要响应 WM_CLOSE 消息，如果我们要在窗体销毁之后做事，就去响应 WM_DESTROY 消息，呵呵，其实 destroy 就是销毁的意思嘛，有时候忘文生意更好，但我们要知道，一旦有了这个消息发出了，窗体已经不存在了，所以一般情况下，这里的处理基本上都是调用 PostQuitMessage(0)。

这儿牛刀给大家做个思考题，是不是在 WM_CLOSE 中调用 PostQuitMessage(0)，WM_QUIT 消息就会跑到 WM_DESTROY 之前产生呢？注：牛刀也没实践过，大家自己慢慢去琢磨吧，呵呵。

下面就上面的三个消息，我们做一个程序，当我们关闭记事本时，是不是会出现一个对话框（假如里面内容被我们改动了）
文件...已经改变，想保存吗，下面出现三个按钮，如果我们按是，则保存文件退出，如果我们按否，则不保存文件退出，如果我们按取消，则没有反应，怎么做呢？如果我们只考虑现象的话，这种程序做起来格外地简单。

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        int i;
        switch(message)
        {
        case WM_CLOSE:
                i=MessageBox(hwnd,"我要退出了,你想保存文件吗？","退出测试",MB_YESNOCANCEL);
                if(i==2)
                {
                        MessageBox(hwnd,"你按了取消键,取消退出","退出提示",MB_OK);
                }
                if(i==6)
                {
                        MessageBox(hwnd,"你按了“是”键，确认退出，再见了， 8 8","退出提示",MB_OK);
                        PostQuitMessage(0);
                }
                if(i==7)
                {
                        MessageBox(hwnd,"你按了“否”键,我就不退出了,谢谢","退出提示",MB_OK);
                }
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
```

```
        }
        return (0);
}
```

当然牛刀这里仅仅是给出一些提示，而真在做程序时，这里就应该是后台程序了，关于后台，我们以后慢慢说，说实话，真正学习编程后，我们主要的任务就是后台编程，也就是话，我们的程序根本就是看不见，摸不着的。而像一些前台的显示，有时候反而不是我们编程需要做的事了，别说给你不信，我们的窗体基本上是“画”出来的，而不是程序做出来的，我们的菜单基本上是敲进去的，而不是程序生成的，那难道就叫面像对象吗？你承认，牛刀还不承认，呵呵，行了，关于三个退出的消息，到此为止了，下面就讲讲一个主要的消息 WM_PAINT 消息，如果没有这一个消息，那我们的程序就仅仅是一个窗口，什么能力都没有。

大家知道，人是需要反馈的动物，假如我们面对的东西是死气沉沉，什么反应都没有，你会有什么感觉呢？你是不是想把你的电脑给砸了？呵呵。

下面就是使得你的程序“活”起来的时候了：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        switch(message)
        {
        case WM_PAINT:
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

牛刀喜欢这样做程序，这样做可以避免好多的错误，真的。

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        switch(message)
        {
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                {
                }
                EndPaint(hwnd,&ps);
                break;
...
...
```

像上面这种情况，我们以后会这样说，在函数中定义一个句柄 hdc 与一个

结构体 ps ，然后在消息 WM_PAINT 中写如下程序：

```
...
hdc=BeginPaint(hwnd,&ps)
{
}
EndPaint(hwnd,&ps);
break;
...
```

我们以后不给大家复制整段的程序了，没意思，下面我们就具体就一说结构体 PAINTSTRUCT 与 句柄 HWND:

为什么我们要用 BeginPaint 与 EndPaint 呢，一个是取得设备描述表 HDC ，一个是销毁设备描述表，其实什么叫设备描述表呢？这还得从我们的系统说起，还是为了兼容啊，大家看，我们有 15'显示器，还有 17'的显示器，有大有，小的，黑白的，彩色的， 4 色的， 2 5 6 色的，真色彩的，带亮度的，带贴图的，等等，多了去了，怎么办？如果按我们以前的方法去编程的话，那得首先知道你是使用什么显示器，否则这程序做得再好，来一个与显示器不兼容，还是显示不出来，什么 Windows 啊，什么 DOS 啊，一切都完，是不？

于是，人们就想到了一个折中的办法，定义了一个设备描述表，也就是，我们的程序其实不是在屏幕上做图，而是在设备描述表上做图，而设备描述表是一个接口，并且规定了一套 API 函数，我们马上要用到，而这些函数为什么能在不同的显示器上显示出来呢？这就是说，每个显示器虽然不同，但显示器的驱动与我们设备描述表的接口对应，现在的显示器一般不需要驱动了，但以前的电脑，每添一个新的设备就必须安装一个相对应的驱动程序，就是这个道理。

也就是说，我们用一个 API 向设备描述表中写了一样东西，这设备描述表就根据表中的内容，寻找相应的驱动程序，就可以在显示器上显示东西了。

再明说一点，我们中国人任一外国人交流，有语言障碍，这就需要翻译，例如一个葡萄牙人，那我们必须找又懂中文，又懂葡萄牙文的翻译么？不，一般我们找一个懂英文的中国人与懂英文的葡萄牙人就一切 OK 了，也就是说，英语相当于一个设备描述表，而基于中文要翻译成哪国文，那就随便了，只要找相应的懂英文的本国人就一切 OK，那人就是我们这里所说的驱动程序。

明白了吧，BeginPaint 是产生一个设备描述表，而 EndPaint 是注销设备描述表，而一旦我们拿到设备描述表，就可以用我们的与设备描述表相关的 API 函数直接向屏幕上写东西了。

这样一来，我们就不需要明白显示器这个硬件，那是显示器厂家的事，如果显示器不支持我们的设备描述表的话，唯一的缺点就是那款显示器肯定卖不出去。

既然厂家与程序员都愿意遵守这一些规则，那么我们还有什么放心自由使用的呢？

当然，设备描述表 HDC 只不过是一个三十二位的数据，而这一数据后面却跟着一串东西呢，什么东西呢？ ps

```
hdc=BeginPaint(hwnd,&ps);
{
        TextOut(hdc,20,20,"汉字显示测试",12);
}
EndPaint(hwnd,&ps);
```

大家看清楚了吗？画图（注意啊，这里的汉显是画图，谁让我们使用的是图形系统呢，呵呵）函数中必须要使用设备描述表 hdc ，目的就是让我们的程序知道是在哪里画图，画在什么地方。我们上面的字符就是显示在窗体的 20,20 这个坐标位置（注意：计算机中的坐标系统与我们直角坐标系不同，直角坐标系是 x 轴向右，y 轴向上的，而计算机中却是 x 轴向右，y 轴向下，坐标原点为右上角）。

当然喽，设备描述表什么作用呢？属性，修改属性的 API 函数也是使用这个设备描述表，看：

```
hdc=BeginPaint(hwnd,&ps);
{
        SetTextColor(hdc,RGB(255,0,0));
                TextOut(hdc,20,20,"汉字显示测试",12);
}
EndPaint(hwnd,&ps);
```

其实我们不是更改了汉字的属性，而是用 SetTextColor 改变了设备描述表的属性，我们汉字的显示就相应地不同了，那到底是哪儿不同了呢？当然是设备描述表所指向的某一项数据改变了，至于哪项数据改变了，我们暂时先不讨论，因为这些东西的确很复杂。

我们这要说的是 RGB(255,0,0) 并不是一个函数，而是一个宏，其中的每个参数分别表示红（R）绿（G）蓝（B）。

当然我们这里的画图属性一般都有默认值，以后我们学多了，会知道，可以用好多的函数来改变这些属性，甚至改变画笔，改变画刷，等等。

这一讲好像又过多地描述好像与结构无关的东西了，但没办法啊，这些都是结构实践的预备知识，因为我们与大家说过，其实 Windows 因为是图形环境，所以对一些结构的表现力特强，并且还有好的消息，再教大家两个函数，我们用一个结构体来结束我们这一讲的内容吧：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        static RECT rt,rtWindow;
        switch(message)
```

```
        {
        case WM_CREATE:
                rt.left=20;
                rt.right=200;
                rt.top=20;
                rt.bottom=200;                                  //上面
几句定义了一个矩形框
                SetTimer(hwnd,0,100,NULL);          //这一句定义了一个
计时器，一秒钟变十次，详细内容我们以后再讲
                GetClientRect(hwnd,&rtWindow);
                break;
        case WM_TIMER:                                          //这儿
是响应定时器消息
                rt.left=rand()%rtWindow.right;
                rt.right=rand()%rtWindow.right;
                rt.top=rand()%rtWindow.bottom;
                rt.bottom=rand()%rtWindow.bottom;
                InvalidateRect(hwnd,NULL,TRUE);         //这一句是发出
WM_PAINT 消息，没有这一句，则不会产生动态效果
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                {
                        Rectangle(hdc,rt.left,rt.top,rt.right,rt.bott
om);
                }
                EndPaint(hwnd,&ps);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

上面的内容，我们起个名子，就叫做《跳动的矩形》吧，大家仔细地体会 WM_TIMER 消息，再体会 RECT 结构前面为什么要用 static ，如果取消可以吗？在家仔细实践实践，有不懂的地方，一定记得发贴问牛刀。

**结构实践〇五七：结构与消息**

其实说实话，如果单单讲结构，可以还真没什么可讲的，也就是一些数据的一个集合而已，并且结构元素用“.”分隔，而结构指针用“->”来指向其中的元素。就这么多的东西，我们还没有会使用这些结构么，事实就是如果，我们都是学习 C 或者 C++这么长时间，而且都是很聪明，是不？呵呵。

上一讲讲的是循环，其实主要是与大家讨论一下 WinMain 中的那一个循环体，向大家展示了消息循环的原理，而我们这一讲就需要着重讲一讲消息。

什么是消息？其实就是一个一个的三十二位数据，用宏定义一下而已，也就是仅此而已。

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        switch(message)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

对了，牛刀说一下，上面程序中的 WM_DESTROY 消息必须响应，好像默认函数中没有对其的响应，因为上一讲好多的程序，牛刀做上机以后，发现虽然窗体结束了，但资源管理器中的进程还在，必须结束进程，所以说，PostQuitMessage(0);是结束进程用的，上一讲管理员说 WM_QUIT 消息不能被拦截，牛刀想应该不能拦截，因为一旦发出这消息，应该就是退出循环了，如果想捕捉，应该这样改一下程序（理论上可行）：

```
        ...
        while (GetMessage(&Msg,NULL,0,0))
        {
                TranslateMessage(&Msg);
                DispatchMessage(&Msg);
        }
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
        return Msg.wParam;
}
```

上面是 WinMain 函数中，如果像上面那样改了，就应该可以拦截了吧，呵呵，牛刀试试啊。

呵呵，不行，为什么不行很可以后台程序中也是一旦收到 WM_QUIT 消息也释放了对我们函数的调用，所以我们这个函数在内存中已经不存在了。

但是下面的一定行：

```
...
while (GetMessage(&Msg,NULL,0,0))
{
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
}
if(Msg.message==WM_QUIT)
{
          MessageBox(NULL,"我捕捉了 WM_QUIT 消息","消息提示
",MB_OK);
}
return Msg.wParam;
}
```

于是，我们看到了一个结构体有机地与我们的消息联系了起来：MSG

而事实也证明了这一点，上面的三个消息处理函数可以有效地处理消息，所以说，我们可以很容易地在这消息循环中去拦截甚至更改消息，一旦我们更改了某一消息，这消息可能就可以被程序拦截了，我们试试看，我们将 WM_CLOSE 消息拦截后，更改为 WM_QUIT 消息，看会出现什么情况，当然因为是中途拦截，所以不会影响系统发出的 WM_QUIT 消息关闭我们的窗口。

牛刀又做了一个程序：

```
...
while (GetMessage(&Msg,NULL,0,0))
{
        if(Msg.message==WM_CLOSE)
                Msg.message=WM_QUIT;
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
}
...
```

想必牛刀的目的大家都知道，调试了一下，但发现得到的 Msg 根本捕捉不到 WM_CLOSE 消息，不知道是什么原因，大家自己还可以试着用其他的例子去实践，牛刀也十分想与大家讨论，其实牛刀还是不服的，又想到了一个怪怪的程序：

```
...
while (GetMessage(&Msg,NULL,0,0))
{
        Msg.message++;
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
```

```
    }
    ...
```

这样一来，程序相当好玩，当然喽，因为我们根本没对任何消息进行响应，也就是默认的相应，牛刀的窗体可以随着鼠标的移动而改变大小，当然这一定是将 WM_MOUSEMOV 的消息变成了 WM_LBUTTONDOWN 消息，我们可以去看看这两个消息是不是一前一后，看：

```
#define WM_MOUSEMOVE                    0x0200
#define WM_LBUTTONDOWN                  0x0201
```

牛刀的猜想果然不假，很不成就感哟，但为什么 WM_CLOSE 消息捕捉不到，还真的请大伙儿仔细讨论。

·结论：我们可以在消息循环中捕捉消息，更改消息，甚至发假消息，呜，感觉有点像黑客们干的事了，不过一般黑客们都通过 HOOK 函数干这些事情，为什么他们必须通过那个函数呢？

原理很简单啊，MFC 对 WinMain 函数是封装的，所以根本不能够半路截获消息，但微软的程序员们为了测试使用，而定义了 HOOK 函数，不知什么时候，这个函数被泄密出来了，从而变成了黑客们的常用工具。

当然我们的技术与 HOOK 函数不能同日而语，HOOK 是计算机底层的东西，既然已经说到这儿了，牛刀也说一说吧，大家别用这些技术去干坏事：

我们消息虽然大家感觉上是通过上面的循环进行的，其实不是，为什么牛刀没有捕捉到 WM_CLOSE 消息可能也是这方面原因，原来啊，我们 Windows 消息不是发一个消息，计算机就响应一个消息的，而是将我们的消息加进了一个队列，我们称之为消息队列，什么叫队列呢？我们讲《初识结构体》时，曾经想详细地讲这些内容，但由于实践性不强，而改与大家讲 API 函数了。队列就像人们排队一样，消息是排除等待的，假如消息队列可以容易二十条消息（注：多少条，牛刀也不知道，呵呵，上面数据是瞎说噢），那么我们发一条消息就放在了消息队列中等待，等到前面的消息都处理完了才响应这一条消息，所以说，我们发出的消息都不能得到及时的响应，但由于响应时间相当地短，所以我们根本感觉不到，也不可能感觉得到，并且告诉大家一个事实，假如你的鼠标一移动，就会不断地发 WM_MOUSEMOVE 消息的，相当频繁，所以说消息的处理其实是不断进行的，也就是说，不断地有消息插入队列，当然也有消息被执行，当然这一点我们一般感觉不到，然而当我们的电脑相当卡，或我们打网络游戏时，网速相当慢，我们就会感觉出来了，按一个按钮可能半天都没反应，明明打着怪物了，再看看发现我们自己的血没了，人回城了，是很常见的事，大家有没有同感，呵呵，好多朋友就大骂 GM ，我想我们会做程序的人一定不会骂的，网络就这样，流量大了必然慢，瓶颈效应嘛，不过说实话，游戏时被卡回了，心里毕竟不舒服，有些言语还真是可以理解的，大家说是不是？

感觉岔得太远了，呵呵，所以说，我们上面试图捕获 WM_CLOSE 消息还没发出时，可能系统已经提前捕获到了 WM_QUIT 消息了，什么意思呢？我们知道， GetMessage 函数遇到 WM_QUIT 消息，就退出了啊，所以当我们将 WM_CLOSE 消息变成了 WM_QUIT 消息时，WM_QUIT 消息被加进了消息队列，但并不能得到及时的执行，我们的消息处理函数还在处理其他的消息，

所以我们发出的 WM_QUIT 消息，已经在队列中，所以以后必然被 GetMessage 函数捕获，而根本来不及由消息处理函数去处理，从而因为 GetMessage 函数捕捉到了 WM_QUIT 消息而返回值为零了，于是退出了消息循环，于是，我们的消息处理函数也就不工作了，于是这个消息队列中的 WM_QUIT 消息连同其他一些没来得及处理的消息被一同放弃执行了。

不知牛刀这样说大家是否能够弄懂，所以说，我们在这消息循环中，可以发送消息，可以更改消息，但要注意两点：
一、消息是不能及时得到执行的
二、消息可以被捕获与更改。

好，再来谈 HOOK 函数（注：不讨论，但要懂原理，呵呵）：

HOOK 函数不是拦截我们发出的消息，而是拦截消息队列中的消息，更改消息队列中的消息，向消息队列中发出一些假消息，从而实现一些“恶心”的目的，呵呵，到些打住，不着深入讨论。

不忍心再说两句，你想想，我们一个程序的消息队列如果被人拿到，还有个好啊？什么东西拿不出来啊？你可知道有些消息的功能真的是很强的，你不得不服，你信吗？大家都有向一个文本框中写入信息，但看到的都是一串“*”的经历吧，我们说那叫密码框？真是是密码框吗？框人行，你框得住发送消息吗？不说了，再说有点……

打住，打住，不着深入讨论。

其实事实上也是这样，学 VB 的人有这个经验吧，TEXT 框，可以用星号，但 TEXT 属性不是星号啊，如果那也是一串星，那么这密码就没有意义了，也就是说，这种密码只是对人，对偷看的人有效，而 TEXT 框的 TEXT 属性还是明文，事实就是如此，所以我们前面在《汇编实践》中讲的那一些技术，那真就算是高档的了，其实事实上根本用不着那些技术。那些技术说实话，破解软件都好使（注：牛刀没破解过软件啊，再说牛刀的那个《汇编实践》为什么中止了，一个是主要内容大体结束了，二一个就是如果再写，就变成了培养黑客的好教材了，划不来，呵呵，再说了，电脑上的黑客有几个是真正的黑客啊？真正的黑客还能被公安机关去破案啊，做梦去吧。你见过几个真正的黑客用别人的黑客软件的？）

牛刀在防杀衣那一篇文章中也曾经讲过，有些软件是绝对不能商品化的，一旦商品化的软件，千万不要用来干违法的事，否则后果不堪设想。说真的，即使想学黑客，也必须将功底打扎实了，计算机的底层的东西了解全面了，我们才有资格当真正的——“红客”，牛刀不允许看牛刀教材的朋友去当黑客啊，这是宗旨。

行了，不多聊了，发现一讲到关键的地方就容易跑题，既然系统是一种消息系统，那是否可以定义我们自己的消息呢？做梦都想着做自己的消息，甚至做自己的系统。

当然喽，自己的消息也是一个数据，我们就先定义这个数据吧：

在程序的一开始，定义：

```
#define WM_MYMESSAGE 0x1234
```

在程序中合使用消息：

```
        case WM_MYMESSAGE:
MessageBox(hwnd,"我捕捉了自己做的消息 WM_MYMESSAGE，如果捕捉成功，则会出现这一对话框","消息捕捉提示",MB_OK);
                break;
```

在合理的地方发送消息：

在这里牛刀说一下，消息进入消息队列，有 GetMessage 函数来接收，但也必须有发送这个消息地方，大家想，假如我们定义了一个消息，但系统根本没有地方调用这个消息，那么我们这个消息做得再好，根本也得不到执行是不？所以我们就必须用到函数 SendMessage 来发送我们的消息，在哪儿发送呢？哪里都可以，在我们想使用此消息的地方最好。

作为测试用，我们在 WM_LBUTTONDOWN 消息中发送吧，鼠标按键一按下，就发送这个消息，这不是挺容易看到反应，呵呵。

```
        case WM_LBUTTONDOWN:
                SendMessage(hwnd,WM_MYMESSAGE,wParam,lParam);
                break;
```

大家看，发送消息的结构与我们上面的参数几乎一一对应，是吧，其实就是这么回事，当然牛刀在这里省事了，其实 wParam lParam 是可以由我们自己自由规定的，自己的消息，自己的函数，自己的响应参数，一切都是自己的。

当然喽，事实上，我们上面的定义 0x1234 大家觉得很大吧，你知道系统用了多少消息呢？所以这种定义方法很容易与系统消息重名，所以我们必须改一下定义：

```
#define WM_MYMESSAGE WM_USER+1
```

很显然，上面的 WM_USER 就是消息的上限，后面加一个合适的数据来定义我们的消息，这个消息就不会与系统犯冲了。

好了，以上使用了消息，响应了消息，我们也定义了自己的消息，想必大家应该对消息有一个全面的了解了吧？

乘机在这地方与大家讲一讲消息的类型，消息基本上有系统消息、用户消息、用户自定义消息这样几种。用户自定义消息我想大家不用说了吧？其实系统消息与用户消息又会有一些分类，我们就不具体分了，没意思：

什么叫系统消息呢？由系统发出的消息呗，像我们前面的 WM_TIMER 消息，其实就是当我们用 SetTimer 函数定义一个计时器后，系统就按规定时间就发出一个 WM_TIMER 消息，这就是系统的事情，也是一个例子吧。

再一个就是用户消息，例如我们按下鼠标的一个键或者是键盘上一个按键，就是一个用户消息了。

当然我们很想系统假如能够发出其他的消息可能就更方便了，大家想没想过

假如我们的端口如果能发消息与接收消息，那有多方便啊，那我们做网络程序就就需要响应几个消息就一切 OK 了，然而我们的 Windows 系统可能在初期时候就只是一个单机系统，而没有考虑网络问题，也是说没有考虑接口问题，可能也正因为如此，所以我们现在每安装一个新的设备就需要安装相应的驱动程序。当然这种情况在 WindowsXP 中要好多了，当然系统也要大多了，当然现在的一些机构也正在做各种标准的制定，像蓝牙技术啊等等。

当然还有什么 IPV6 啊，技术多喽，牛刀也有很多技术，没钱申请专利啊，呵呵，没办法。哎～～～
有空在论坛上发出来，呵呵。

好了，我们既然知道了这些消息的各种形态与发送接收的方法，以及消息的处理，那么我们就使用这些技术，结合结构体来与大家做个程序：作为这一讲的结束。

其实嘛，我们学习本来就是要做程序的。如果我们学习了，光停在理论上，结构体再好，对于我们又有什么用呢？是吧？

那我们这一讲的最好给大家做什么程序呢？用到什么结构呢？画图吧，大家知道画图程序吗？选中一个颜色，选笔，在屏幕上一拖，就会出现一条轨迹，大家是不是常做这样的事？其实也是的，很方便的画图软件，当然如果让我们去做一个画图程序，也没必要，暂时水平也还不够，我们就做一个画线程序吧，我们用鼠标左键一拖，出现一道红线，右键一拖出现绿线，怎么样？

当然喽，这里就要用到结构体 POINT ，但那是系统定义的，也很简单：

```
struct POINT
{
        int x;
        int y;
};
```

我们不用系统定义的，我们自己定义一个结构体 P ，我们用四条属性，虽然大家知道真正做程序没有必要，但我们是给大家看看，结构体是如何封装数据的。

```
struct P
{
        int x;
        int y;
        BOOL bdfla;         //是否画图
        BOOL bcfla;         //画什么颜色的图
};
```

定义这一个结构体，其中的 x,y 表示了当前的坐标，而 bdfla 表示是否画图，什么意思呢？当我们的鼠标按键放开后 WM_LBUTTONUP WM_RBUTTONUP ，我们就不需要画图了，将这个置为 FALSE ，如果我们鼠标键按下 WM_LBUTTONDOWN WM_RBUTTONDOWN，则我们需要画图，

置为 TRUE

同理，当是左键操作，我们将 bcfla 置为 TRUE 否则置为 FALSE 过程大家明白了吗？如果明白了，建议大家先不要看下面的程序，自己做一个试试，然后再看牛刀给出的程序（对了，提示一下，因为我们在整个过程中要跟踪这个坐标，所以我们有两套定义方案，一套是将其声明为全局变量，也就是不在函数中定义，而在 WinMain 函数之前就定义好了，第二套方案在前面加上 static 定义为局部固定的变量，我们采用第二套方案：

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        static P pt;          //定义一个变量，注意前面定义方式
        HDC hdc;          //想画图，这一定义与下一定义是必须有的。
        PAINTSTRUCT ps;
        switch(message)
        {
        case WM_CREATE:                    //窗建窗体的消息，我们一般在这里对数据进行初始化
                pt.bcfla=FALSE;                    //初始化画红线
                pt.bdfla=FALSE;                    //初始化为不画线
        case WM_MOUSEMOVE:          //一旦有鼠标的移动，我们就需要捕捉位置
                pt.x=LOWORD(lParam);
                pt.y=HIWORD(lParam);
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_LBUTTONDOWN:          //当鼠标左键按下，我们就需要画图，所以需要将相应的数据设计，其他也一样，不多说了。
                pt.bcfla=TRUE;
                pt.bdfla=TRUE;
                break;
        case WM_RBUTTONDOWN:
                pt.bcfla=FALSE;
                pt.bdfla=TRUE;
                break;
        case WM_LBUTTONUP:
                pt.bdfla=FALSE;
                break;
        case WM_RBUTTONUP:
                pt.bdfla=FALSE;
                break;
        case WM_PAINT:          //由于上面的设计，这里程序做起来就方便了，按照上面的设计去画图就一切 OK 了
                hdc=BeginPaint(hwnd,&ps);
                {
```

```
                        if(pt.bdfla)          //对是否画图进行判断
                        {
                                LineTo(hdc,pt.x,pt.y);
                        }
                }
                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

注：我们想得挺好，事实上很让我们失望是不是？牛刀也有这个感觉，可为什么会出现以上的情况呢？什么情况大家知道吧，只可意会不可言传，如果你看牛刀的教材不上机实践，那不关牛刀的事，牛刀有时候做东西是不给结果的，逼着你去实践，呵呵。

那就还回到我们的消息循环上来吧，原来啊，程序每进行一次循环，刷新一个消息，如上面的 WM_PAINT 消息，好多东西是复位的，什么意思呢？也就是说每一次进入我们的消息函数，首先位置信息就复位到了左上角，即原点(0,0)，所以我们看到，总是从左上角到鼠标处冒出一条线来，可为什么仅仅是一条呢？因为我们的 WM_PAINT 消息是刷新屏幕的，如果大家想不刷新屏幕，可以将 WM_MOUSEMOVE 消息中的

```
InvalidateRect(hwnd,NULL,TRUE);
```

改为：

```
hdc=GetDC(hwnd);
{
        LineTo(hdc,pt.x,pt.y);
}
ReleaseDC(hwnd,hdc);
```

大家仔细体会，细细分析，牛刀在这里不做过多地解释了。其实好多东西是需要思考的。

行了，既然事不如不原，那么有两种解决办法，一种就是我们不要 WM_PAINT 消息来画图了，直接使用 WM_MOUSEMOVE 消息，拿到一个设备描述表去画，请注意，在 WM_PAINT 消息中是用 BeginPaint 函数拿到设备描述表的，而在这个消息之外，则需要用 GetDC 函数拿到设备描述表，为什么呢？大家看 BeginPaint 函数拿到描述表的同时又拿到了 ps 结构体嘛？谁让那是专业处理画图的消息呢，没办法，人家享有特权啊，大公章在人家手里哪，呵呵。

当然不要画图函数中处理，我们就要再保存一份当前的指针，所以我们需要

再定义一个结构，当然我们可以再用 POINT 来定义一个试试：

```
        ...
        case WM_CREATE:                    //窗建窗体的消息，我们一般在这
里对数据进行初始化
                pt.bcfla=FALSE;                    //初始化画红线
                pt.bdfla=FALSE;                    //初始化为不画线
                pt.x=200;
                pt.y=200;
        case WM_MOUSEMOVE:          //一旦有鼠标的移动，我们就需要捕捉位
置
                point.x=pt.x;
                point.y=pt.y;
                pt.x=LOWORD(lParam);
                pt.y=HIWORD(lParam);
//                InvalidateRect(hwnd,NULL,TRUE);
                hdc=GetDC(hwnd);
                {
                        if(pt.bdfla)
                        {
                                MoveToEx(hdc,point.x,point.y,NULL);
                                LineTo(hdc,pt.x,pt.y);
                        }
                }
                ReleaseDC(hwnd,hdc);
                break;
        ...
```

上面的程序效果就相当好了，但是不能画颜色，怎么加呢？那就需要用到另一个结构体——画笔了：（HPEN）

注意，我们由于在 WM_MOUSEMOVE 中做方便，如果大家要在 WM_PAINT 中做相同的画图，就需要定义一个点（POINT 或我们自己定义的 P ）的数组了，做得大大的，并且需要每次处理多少数据都要心知肚明，那样的程序就让大家自己去练手吧，其实作为聪明的你，做这种程序是很容易的，一定是很容易的，我们这里暂时就不对那里进行编程了，下面用一个画笔程序来结束本章，关于相关内容，我们到下一讲再详细与大家探讨：

```
        case WM_MOUSEMOVE:          //一旦有鼠标的移动，我们就需要捕捉位置
                point.x=pt.x;
                point.y=pt.y;
                pt.x=LOWORD(lParam);
                pt.y=HIWORD(lParam);
//                InvalidateRect(hwnd,NULL,TRUE);
                hdc=GetDC(hwnd);
                {
                        if(pt.bdfla)
                        {
```

```
                    hPen=CreatePen(1,1,pt.bcfla?RGB(255,0,0):RGB(0,255,0));        //这一句我创建了一个画笔，根据我们的颜色设计
    SelectObject(hdc,(HPEN)hPen);
                    //大家看，我们选入一个画笔时，同样是用了  hdc  就是说，设备描述表，所以大家也可以管中窥豹了吧，呵呵。这一句是将一个画笔选入设备描述表。

                        MoveToEx(hdc,point.x,point.y,NULL);
                        LineTo(hdc,pt.x,pt.y);
                }
            }
            ReleaseDC(hwnd,hdc);
            break;
```

经过上面的改动，程序效果就相当地好了，大家拿着鼠标，去我们自己做的画图程序上写几个字“牛刀教我做画图”，呵呵，其实说实话，我们做程序的过程其实就是这样的简单，一旦我们有什么构思了，用一些办法去看看怎么能够实现她。

**类的实践〇五八：类与结构体**

这一讲开始类的实践了，那么结构结束了吗？指针结束了吗？没，远远没有结束，类是结构的延续，指针却将类的功能发挥得淋漓尽致。

那么什么是类呢？与结构就是定义的关键字不同罢了，结构用的是 struct，而类的关键是 class 例如：

```
struct POINT
{
        int x;
        int y;
};
class POINT
{
        int x;
        int y;
};
```

以上两个一样吗？果真一样吗？当然不一样，如果一样的话，那么也就没什么意义了，我们在结构体的定义中，可以使用其中的元素，但类中却不能。

为什么不能呢？说来话长，在类中定义了三种访问方式：

**引用：**

public:
private:
protected:

注意后面，需要冒号“:”。

**引用：**

public: 公共
private: 私用
protected: 保护

什么叫公用的呢？就是大家都能用的呗。

什么叫私用的呢？就是并不是大家都能使用的。

保护的呢？这就需要我们以后慢慢与大伙讨论了，这一讲我们就讲公用与私用的两种：

而 struct 与 class 的唯一不同点就是默认的类型不同，那么：

```
class POINT
{
pbulic:
        int x;
        int y;
```

```
};
```
就与下面的结构体相同喽：
```
struct POINT
{
        int x;
        int y;
};
```
而结构体：
```
struct POINT
{
private:
        int x;
        int y;
}
```
则与下面的类相同：
```
class POINT
{
        int x;
        int y;
};
```
通过上面的对比，我们知道了，结构体的默认类型为公共类型，而 class 的默认类型则为私有类型。

其实我们前面说的结构体，则全是公有类型的，所以说我们根本就是使用一个点“.”或者用箭头“->”，来引用结构变量或结构指针中的元素数据，而类也是一样，但原则是，私有的不能引用，而公用的可以引用。这样说起来是不是很枯燥啊？我们还是老一套，来做程序吧。当然，这一讲开始，我们还是做控制台应用程序。

怎么又不做 API ，又不做 Windows 了呢？原因很简单，我们一开始类就与 API 套在一起讲，以后讲到面象对象的时候就很容易将类、对象、与 API，窗体，等东西搞混了，说实话，学 VB 的人，基本上就不知道什么叫过程，什么叫对象了，VB 号称是面向对象，可在牛刀这儿说来吧，面向对象根本不存在，是不是大家觉得有点吃惊？事实就是如此，这取决于我们电脑的原理与结构，我们的电脑是程序控制，也就是说程序驱动，并且程序之间是线性关系的，也就是说，我们的电脑只能处理过程信息的，我们看到的所谓的面向对象，其实就是一种表面现象的模拟，就好像人们老说的人工智能，真的能实现人工智能吗？就现代的电脑的结构，也是不可能实现的，甚至我们说的所谓的 3D 技能也是不现实的，都是一种表面现象。当然一些表面现象能做到如今这样，也算是很辉煌了。

什么意思再说得简单一点吧，大家看过牛刀的《汇编实践》吧？当然我们那是十六位汇编，或者说是古典式的汇编，现在奔腾的芯片出来了，就引入了三十二位的系统，也就是说现在的系统支持的是三十二位的汇编系统，但毕竟还是汇编，所以说，我们不管是 VB，还是 VC，或者是 DELPHI，编译的结果都是三十

二位的汇编指令，而这些指令与我们以前学的那些指令也没有什么区别，只不过是多了几个寄存器，每个寄存器都能支持三十二位数据而已，你想汇编就是一串代码而已，什么叫面向对象？你从哪方面说这是面向对象的？

再说人工智能，其实就是一种数据处理而已，只不过是一些数据选得比较绝，并且符合一些特征，用电脑来演算而已。但得出的一些结论可不能小瞧。

再说 3D 技术，说实话，如果让电脑去做，2D 都做不了，但为什么我们的屏幕能显示 2D 的东西呢？那是显示器与显卡在发挥作用，在这牛刀也不想具体谈显卡的结构或者显示器的原理，总之我们显示器其实不是 2D 的，什么叫 2D 我们应该有概念，牛刀概念就是横轴上无限延伸，可无限分割，竖轴上也同样无限延伸并且无限分割，那才叫 2D，而我们显示器去是每行多少像素，有多少行，都是有数的，所以说，在显卡中的显存其实也是一维的结构，只不过我们在显示器上看到的反而是一种假象，从这个方面的原理来说，3D 技术就显得更假了。

但我们不得不说 3D 技术中的数据结构是一个相当优秀的数据结构，并且这种结构的理论基础相当扎实，当我们作出一个 3D 的模型，加上光照模型，再加上返射模型，真实感模型，终合起来生成的一些东西看起来就像照片一样效果很棒的，当然我们不再聊开去了，这样聊牛刀发现已经有些跑题了，呵呵。

我们说结构其实是定义了一段空间，只不过这个空间中的哪个地方，什么数据类型都定义好了，类也是一样，甚至类中的某一段空间的数据是不允许外面访问的。还是来实践一段程序吧：

```
#include <stdio.h>

class POINT
{
        int x;
        int y;
};

void main()
{
        POINT pt;
        pt.x=20;
        pt.y=20;
}
```

这段程序其实没什么意义，并且一运行就报错，什么原因呢？因为我们说过了，我们说类的默认类型是私有类型的，外部数据不能访问的，那么我们想在外部访问，就需要在前面加上 pbulic: 就一切 OK 了。

```
class POINT
{
public:
        int x;
```

```
        int y;
};
```

当然，这种定义的确可以这样，但如果我们就这样来使用的话，那不是和结构一样了吗？我们既然学“类”，我们就要使用类的特点，所以我们还将 x,y 设为私有的（也就是说前面什么都不加，或者加 private: ）

```
class POINT
{
        int x;
        int y;
};
```

如果真的这样定义，那我们这个类就什么用都没有，所以类使用的一个原则就是：必须有公用接口，也就是说，必须有 pbulic: 如果没有这个，那我们定义类干什么呢？有什么用呢？大家知道，程序不是摆设，是需要运行的，你做一个非常优秀的类，却没办法使用，那这个类再好也是废代码一堆，你说是不？

也正因为如此，我们所有类中都有关键字 public: ，甚至一般我们在一开始就加上这个关键字，从这一点上来说，我们的类与结构其实也没什么区别，所以说，我们从这一讲开始讲的类，其实就是结构的延续，大家别以为我们的讨论与结构没关系了，从理论上来讲，一个一开始就定义 public: 的类，就是结构，一开始就定义 private: 的结构，也就是类，所以说，以后所讲的一切，在结构体中都好使，当然这也是 C++同时也扩充了结构的功能了，如果是做一个 C 程序，如为名人掌上电脑做程序，我们就不能这样用了，因为那是 C 程序。

好了，我们还来定义下面的类：

```
class POINT
{
        int x;
        int y;
public:
};
```

像上面的结构体，因为一进来默认是私有的，x 和 y 外部根本不能访问，也就是说不能像下面一样：

```
POINT pt;
pt.x=20;        //错
pt.y=20;        //错
```

那么这两个数据我们怎么能够使用呢？原来类中又加进了一些新的内容，我们的类可以封装函数，我们称这为公共接口，通过这些公共接口，我们就可以使用类中的成员变量了，大家看：

```
#include <stdio.h>
class POINT
{
        int x;
        int y;
public:
```

```
        void Setx(int a)
        {
                x=a;
        }
        void Sety(int b)
        {
                y=b;
        }
        void print()
        {
                printf("x=%d\ny=%d\n",x,y);
        }
};

void main()
{
        POINT pt;
        pt.Setx(4);
        pt.Sety(5);
        pt.print();
}
```

这个程序又是一个什么用处都没有的程序，但还是那句话，牛刀的实践是为了理解，而不是为了使用。

大家想到了一个问题没有？我们的类做得很小，并且函数个数也不多，每个函数只有一句,这看起来感觉还可以,可真正如果我们做一个功能十分强劲的类，那么就函数这一块要有多长啊？那么函数岂不会显得十分臃肿？于是,我们的类允许我们在类中只定义函数原型，而对类中函数的定义拿到函数外面来进行，也就是说在函数中定义一面“小镜子”，呵呵，大家还记得小镜子的事吧？

```
class POINT
{
        int x;
        int y;
public:
        void Setx(int a);
        void Sety(int b);
        void print();
};
```

这样一来，我们的类就会显得很有条理了，那么，这里是有原型了，函数怎么定义呢？这就要用到作用域标识符“::”，其实就是两个冒号。当两个冒号前没有什么数据时，这表示是一个全局变量，如：

```
#include <stdio.h>
int i=20;         //定义全局变量  i=20
void main()
```

```
{
        int i=5;        //又定义了一个与全局变量一样的变量，i=5
        printf("i=%d\n",i);        //大家思考，这里会打印出什么呢？
        printf("::i=%d\n",::i);        //大家再想想，这里又会打印出什么呢？
}
```

又是一个没用的程序，大家仔细体会 :: 这一作用域符的用途，而这前面可以是一个类（当然也可是结构啦，一样的）。

```
void POINT::Setx(int a)
{
        x=a;
}
void POINT::Sety(int b);
{
        y=b;
}
void POINT::print()
{
        printf("x=%d\ny=%d\n",x,y);
}
```

大家看，主函数不要变，只需要做这样的定义就一切 OK 喽。

于是，我们顺利地踏进了类的大门，函数，为我们的类铺就了一条功能的大道，使得我们的类可能会具有很强大的功能，强大到什么程度呢？

可能我们在主函数中，只定义了一个类，赋了一个值，并且运行了类中的一个函数，而全部过程，则都被移到了类中进行，我们就用这种技术来做一个例程序吧。

一般教程会做类似于下面的程序：

```
void DOG::say()
{
        printf("汪汪汪");
}
```

好玩是好玩，我们不做这个，我们做一个历史名题——叙古拉猜想：

也有书上称之为叙拉古的，到底是什么，牛刀也弄糊涂了，不管是什么吧，猜想很简单：

一个任意自然数，按如下规则进行运算：

·如果这个数是奇数，则×３＋１

·如果这个数是偶数，则÷2

·猜想结论：任一自然数，经过如上运算，最终都会回到１并且按１、４、２、１循环。

请注意，这个猜想目前还只是一个猜想，没有得到证明，而我们计算机可以很快地验证，当然你可以做一个程序去验证好多数据，但结果都回到１，不信你

跟着牛刀来做这个“类”。

```
#include <stdio.h>
class Xgl
{
        int ibeg;        //初始数据
public:
        void Init(int a);        //初始化类
        void print();                //打印数据
};

void main()
{
        Xgl xgl;
        xgl.Init(37);
        xgl.print();
}
```

大家看，这样一来，类显得格外地简单，而主函数也简单到了极底，这就是一个非常优秀的程序了，可主要代码呢？我们还没编呢，难怪的人说这就是面向对象，不错，我们面向的的确不是一串串的代码了，但代码却做到了一个一个的小程序中了，下面我们就要做这一部分代码了：

```
void Xgl::Init(int a)
{
        ibeg=a;
}
void Xgl::print()
{
        int sum=ibeg;
        printf("%d->",sum);
        for(  ;sum!=1;  )
        {
                if(sum%2)
                {
                        sum=sum*3+1;
                }
                else
                {
                        sum/=2;
                }
                printf("%d->",sum);
        }
        printf("4\n");
}
```

这样很容易地一来，我们的程序就可以顺利运行了，并且运行的很好，我们运行程序时，如果想改变我们测试的数据，只需要改变：

xgl.Init(...) 中的数据就一切 OK 了，听说 27 这个数据十分诡，大家去测试看看：

**引用:**

27->82->41->124->62->31->94->47->142->71->214->107->322->161->484->242->121->364
->182->91->274->137->412->206->103->310->155->466->233->700->350->175->526->263-
>790->395->1186->593->1780->890->445->1336->668->334->167->502->251->754->377->1
132->566->283->850->425->1276->638->319->958->479->1438->719->2158->1079->3238->
1619->4858->2429->7288->3644->1822->911->2734->1367->4102->2051->6154->3077->923
2->4616->2308->1154->577->1732->866->433->1300->650->325->976->488->244->122->61
->184->92->46->23->70->35->106->53->160->80->40->20->10->5->16->8->4->2->1->4

如果你遇到什么数据代进去循环结束不了的，可要告诉牛刀哟，那一定是一项新发现，呵呵。

就上面的程序，我们是不是很容易将其改变变通的程序，大家看，我们将 xgl.Init 用函数名 xglInit 替代了，xgl.print 用 xglprint 替代了，而其中的变量也用相应的名称替代了，大家再来看程序：

```
#include <stdio.h>
int xglibeg;
void xglInit(int a);
void xglprint();
void main()
{
        xglInit(27);
        xglprint();
}
void xglInit(int a)
{
        xglibeg=a;
}
void xglprint()
{
        int sum=xglibeg;
        printf("%d->",sum);
        for(  ;sum!=1;  )
        {
```

```
                if(sum%2)
                {
                        sum=sum*3+1;
                }
                else
                {
                        sum/=2;
                }
                printf("%d->",sum);
        }
        printf("4\n");
}
```

一样是一款非常优秀的程序，从这个意义上来说，大家明白为什么牛刀就牛刀不承认面向对象啊，什么类啊，是什么原因了吧？其实说真的，C++的编译系统只不过是按照另一种方式来改变我们函数的高手，为什么这么样说，等到我们以后说到什么重载啊，什么继承啊，等等一系列的东西时，再详细讨论。

但我们不得不说 VC6.0 环境中的提示功能做得相当的好，虽然我们知道不可能实现面向对象，但我们还是愿意这样认为的。方便啊。

我们看上面我们很容易将我们的对象变成了普通的函数，其实编译系统做的也是同样的工作，按照一定的规则，将我们的所有函数全变了，真的是所有的函数全变了，如果不想变的话，你反而要在前面加上特殊的标志：

extern "C"

为什么要定义这个呢？兼容，还是兼容惹的祸，有好多的 C 代码太优秀了，不忍放弃，怎么办呢？前面加上上面一句，一旦 C++看到上面一句了，就按 C 语言的规则去编译这一个文件的程序了。以后提到再讲。

事实上就是如此，其实类、结构就是一种提法，或者是一种数据的组织方法，而实现这些方法的是编译器，而不是电脑，一旦这些东西变成了机器的二进制的东西，也就是说变成了汇编代码，那也只不过是一套原来就存在的汇编程序而已，仅此而已。

还是那句话，大家也可以提出自己的设想与构思，去做自己的编译器，一样的，只要我们的想法合理，容易实现，那么我们的编译器做起来相应还是比较简单的。

当然本讲牛刀不带大家做一款自己的编译器，如果那样的话就严重跑题，呵呵。

对了，既然我们的“类”已经开了头了，我们就讲一些实在的东西吧，或者说对我们来说是有用的东西，类中有两个函数特别有趣，这两个函数，一个与类名相同，一个是类名前面加上一个“~”，这一符号，好事者将其中同名函数称之为构造函数，而前面加 ~ 符号的称之为析构函数，那么这两个函数有什么用呢？

大家知道，任何数据都有生命周期，是不是？当一个函数中定义的数据，函数结束了，数据的生命周期也就结束了，大括号中定义的数据，遇到相应的右大括号，生命周期也就结束了，但那是数据的定义，一个类也同样有生命周期，而上面两个函数就是生命周期的一始一终，所形成的那一段空间中可以对此类做相应的调用，但是一旦类被定义，构造函数却是自动调用，不需要显式调用，而析构函数也正好是在生命周期结束时被自动调用，看程序，看结果，仔细体会，有些东西真的要体会的，不然概念太抽象，我们定义一个类“A”，看看调用过程：

构造函数是肯定没有返回值的，所以说，构造函数的类型说明都可以不要，看程序：

```
#include <stdio.h>
class A
{
public:
        A();
        ~A();
};

void main()
{
        printf("进入主函数\n");
        A a;
        printf("退出主函数\n");
}

A::A()
{
        printf("进入类的构造函数\n");
}
A::~A()
{
        printf("进入类的析构函数\n");
}
```

还是那句话，牛刀做的这些程序是什么功能都没有，看结果：

```
进入主函数
进入类的构造函数
退出主函数
进入类的析构函数
```

大家看，我们一进入主函数时，就有相应的提示了，而我们定义类时，并没有调用相应的构造函数啊，但这个函数的确被调用了。

再看退出主函数后，我们类的使命也就结束了，于是又自动调用了析构函数。

明白了以上的过程，我们就自然会思考，为什么要定义这两种函数呢？

人懒啊，没办法，大家还记得我们以前给大家讲过，使用 new 定义的东西一定要用 delete 释放吗？大家看，我们将上面的程序改一改：

```
#include <stdio.h>
class A
{
        char *ch;
public:
        A();
        ~A();
};

void main()
{
        A a;
}

A::A()
{
        ch=new char[10];
}
A::~A()
{
        delete []ch;
}
```

又是一个严重没用的程序，这个程序只是在构造函数中 new 了一个指针数组，在析构时释放了。事实还果真如此，我们一般都是在构造函数中作类的初始化工作，而在析构函数中做类的善后工作，也就是说，我们一般做一个类，都是先将这两个东西组织好，而我们一旦做程序时，则不需要考虑好多好多的问题，方便啊。

当然这种方便目前我们还真体会不到，牛刀也很难用一个合适的例程来体现这种方便。只有大家在学习过这些内容后，自己去仔细地琢磨。

这一讲的结束，与前面不同，这一讲我们不给程序给大家了，这一讲的结束，牛刀要与大家讨论一个问题——类给了我们什么？

类给了我们一种面向对象的思想与思路，并且这种思想与思路做得相当的完美。

类给了我们一种整体的思想与思维，我们做程序根本不需要像以前那样，去设计每一个细节，我们只需要考虑我们需要哪些功能。

类给了我们一种自顶向下的编程方案，我们的程序通过类被分解成了一个一个的功能区，我们的目的就是实现每一个功能，用我们的脑袋。

类给了我们一种程序结构，使得我们的程序结构严谨层次分明。

类给了我们一种符合我们思维的思考方案，使得我们做程序不是面像机器，

而是面向人

类给了我们一种……

……

类的优点的太太太太太太太太太多了。

假如你来设计编程的语言，你会想到做一个类吗？类是人类思想的结晶，永远都是电脑编程史上不可抹煞的一笔。

好像牛刀写文章不容易这样大发感慨哟，呵呵，但类做得的确完美，大家还记得前面有一行代码吗？

```
wndclass.lpszClassName=lpszClassName;
```

我们说过，字符串是不能用等号赋值的，但这里为什么就能了呢？这是类的符号重载。

类还有好多优秀的特性，像什么多态性啊，好多，甚至可以做出一些你以前想都不敢想的事例如

```
CRect rt1,rt2;
...
...
rt2=rt1;
...
```

什么意思呢？就是一个结构或者说一个类，我们一个等号甚至可以将好多好多的数据都赋值过去，而这在以前的结构中是绝对不允许的，但现在我们在结构中也可以了，我们说过 C++中的结构功能已经作了扩展，只要我们用的是 C++，我们就可以按 C++的规则来办。

**类的实践〇五九：相同的函数**

其实这一篇这应该拿到这儿来写的，应该放在后面，但在论坛上已经有人讨论到了，牛刀在些就先拿到前面来讨论一下，以解大家心中疑云。

呵呵，说实话也惭愧，看人家的贴子点击率那么高，回复率那么高，可牛刀的贴子却一直默默无闻，只不过有时管理员还回复一下，相对于那些热门的帖子来说，牛刀真是自愧不如啊。

好在我们有个好的管理员，也真感谢管理员，牛刀的帖子老被加入精华。

行了，闲言少叙，我们书归正传。这一讲我们就讲一讲好多函数重名的问题，我们还是从　jeyya　给出的一个类与一个定义开始吧：

```
int staff;
class staff
{};
void main()
{
}
```

为什么这里类与一个全局变量重名，却不报错？真是搞不懂？咱们来做一个函数原型，但我们不定义这个函数，编译，再看看会出现什么情况：

```
int max(int a,int b);
void main()
{
        int a=3,b=5;
        int c;
        c=max(a,b);
}
```

这一段程序再简单不过了，千万别认为牛刀忘了写函数了，牛刀是有意不写的，一运行，当然报错：

**引用:**

Compiling...
Test.cpp
Linking...
Test.obj : error LNK2001: unresolved external symbol "int __cdecl max(int,int)" (?max@@YAHHH@Z)
Debug/Test.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.

大家看，程序编译是通过了，并且还进行了链接，一切 OK，但就是在链接时出现了一个问题，括号中出现了一串怪怪的东西：max@@YAHHH@Z　，这是什么？没看过啊。

的确没看过，可事实上就是如此，程序在编译过后，悄悄地将我们函数的名称给“变”了。至于怎么变，那是编译系统的事的，我们讲原理就不考虑那一串的东西了，假如我们现在手工来编译，我们给出下面的函数名改变规则：（注：其实上面的函数名也是按某一规则变换过来的）

·规则：

**函数名＋返回值类型＋参数个数＋参数类型。**

什么意思呢？上面是牛刀给出的规则，我们假设编译器就是按上面的规则来编译的，如果函数返回值与参数是 int 型，就在后面加上 i，而如果是 float 型，我们就加上 f ，如果是 double 型，我们就加上 d，还是举例子吧：

```
int max(int a,int b);
```

如果编译的话，就变成： maxi2ii （大家对照规则来看）

```
float max(float a,float b);
```

如果编译的话，就变成： maxf2ff

```
double max( double a,double b);
```

如果编译的话，就变成： maxd2dd

事实上编译系统就是这样做的，只不过是编译的规则不同，那么这样做有什么好处呢？

```
void main()
{
        int a=3;
        int b=4;
        int c=max(a,b);
}
```

这我们看上去也挺正常，可事实上不是，当出现一个调用函数的地方，编译系统就会去分析这些参数的数据类型，怎么分析？ sizeof 啊，呵呵。

然后根据返回值的类型与参数类型作同样的改变，所以说如果一旦上面的程序经过编译，就出被编译成：

```
        int c=maxi2ii(a,b);
```

当然，这些过程都是编译器自动完成了，或者说对我们编程人员来说是透明的，所以我们根本不知道这一点，有些书上就会对你说，C++支持多态型，函数可以重载，其实经过上面一分析，其实远不是那么回事，只不过是编译器在编译过程中耍了一套把戏而已，仅此而已。

但这种性质，我们还是要提，什么原因？方便啊。

·牛刀规则二：

**函数中如果有默认值，则默认值一定要是后几个，函数则变成相应的只含有没有默认值的函数。**还是来举例子吧：

```
int max(int a,int b,int c=4,int d=5);
```

像上面的函数就会被牛刀编译系统变成以下三个函数：

```
int maxi4iiii(int a,int b,int c,int d);
int maxi3iii(int a,int b,int c)
{
        int d=5;
        maxi4iiii(a,b,c,d);
}

int maxi2ii(int a,int b)
{
        int c=4;
        maxi3ii(a,b,c);
}
```

大家看，上面三个函数各不相同，所以我们的函数中可以随便调用几个参数的函数，如：

```
void main()
{
        int a=7;
        int b=2;
        int c=4;
        int t=max(a,b,c);
}
```

如果上面的调用经过编译，会编译成哪个函数相必大家一定会知道吧？

所以说，不是函数可以重载，而是函数不会重名，当然，我们说，即使 C++，函数也不能重名的，以至于下面两个函数编译就一定会出现重名错误：

```
int max(int a,int b,int c);
int max(int a,int b,int c,int d=5);
```

大家自己去分析为什么上面两个函数不能同时存在，是怎么重名的？

上面的分析也就是我们所说的 C++中的重载技术，C++中，每个类，每个函数都按照某种规则进行了重载，只不过我们不知道罢了，但我们知道了，上面的一段程序就显而易见了：

```
int staff;
class staff
{};
void main()
{
}
```

我们还假设类名的重载后面加了一个 c，那么上面的程序就一定被编译成了如下的程序：

```
int staff;
class staffc
{
};
void main()
{
```

}

请问，上面的程序有问题吗？显然没有问题。类与定义本来就不是一个名称啊，能有什么问题？但至于在定义中不能定义相应的类，那是编译器做得不完善，或者说不完美，不支持这种使用罢了，但我们做程序正常情况下也不可能这样使用，你们说是吧，但我们一定要知道，为什么编译器不报错，那就是函数与类，其实不是原来那个函数与那个类了，事实就是如此。

所以我们说这种现象，我们也必须说一说其规则，也就是重载规则，我们不知道哪些编译器可以重载，哪些不能，那就必须按别人做好的编译器的规则办事：

·函数参数个数不同可以重载

·函数参数与返回值的数据类型有不同可以重载

·函数可以定义默认值，但必须是参数中从后向前定义默认值，有默认值参数全拿掉后形成的同名函数不能重载。（这一个规则牛刀定义得有此咬口是不是？绝对不是书上的定义，呵呵，牛刀想当然的，也就是说，我们在调用函数时不能发生歧义）

看程序：

```
int max(int a,int b,int c=5);          //第一个函数
int max(int a,int b);                  //第二个函数
void main()
{
        int a=3,b=5;
        int c=max(a,b);                    //这里你是让程序调用第一个函数，还是第二个函数呢？不要说编译系统，你知道吗？
}
```

还有一条牛刀规则，或者叫 jeyya 规则吧，这是一条实践规则：

**函数与数据定义可以重载，但不能使用。**//VC6.0 实践规则

其实所谓的规则就是这样的，名人上有一个叫 MJ_BASIC 的 BASIC 模拟器，其实早就有人叫牛刀开始 BASIC 模拟器了，牛刀没做，为什么？事多啊，动不动就有人叫改规则，动不动就有人叫改规则，不信你们去名人论坛上去看，事实就是如此，特烦，再说那种东西只能模拟，又不能做成可执行文件，说实话（再说一点吧，有人好久前就叫牛刀做《魔塔》了，那种提议也是牛刀想看看论坛上人对这种游戏的热衷程度的而已，说实话，《魔塔 5 0 关》我自己都没玩过去，你说怎么做游戏啊？不是无米之炊嘛？呵呵，再说一句实话，那游戏也真懒得玩，真的。）。

行了，关于 C++ 的多态性与重载，我们暂时就讨论这么多，我们学习是为了什么？用啊，实践，不实践，这些理论干什么的？我们下面就来使用这种种特性，来为大家做程序：

告诉大家一个好事，构造函数允许重载。看程序：

```
#include <stdio.h>
class A
```

```
{
        int x,y,z;
public:
        A()
        {
                x=3;
                y=4;
                z=5;
        }
        A(int a)
        {
                x=a;
                y=6;
                z=7;
        }
        A(int a,int b)
        {
                x=a;
                y=b;
                z=8;
        }
        A(int a,int b,int c)
        {
                x=a;
                y=b;
                z=c;
        }
        void print()
        {
                printf("x=%d\ny=%d\nz=%d\n",x,y,z);
        }
};
void main()
{
        A a,b(11),c(12,13),d(14,15,16);
        a.print();
        b.print();
        c.print();
        d.print();
}
```

牛刀沉默，大家看结果：

**引用:**

x=3
y=4
z=5
x=11
y=6
z=7
x=12
y=13
z=8
x=14
y=15
z=16

注意结果与类的对应关系，大家慢慢去分析两个问题：
一、构造函数是如何调用的？
二、a,b,c,d 四个定义分别调用的是哪一个构造函数？

请大家注意，构造函数是可以通过后面加括号来调用的，而析构函数却没有任何地方可以调用，所以析构函数就也有两个特点：
一、析构函数没有参数与返回值（有也没意义）
二、析构函数不允许重载 （大家可以去做试验，如果能重载，告诉牛刀一声，这一条规则不是实践规则，是牛刀瞎想的，呵呵）

再说析构函数果真做出来了，能在程序中调用么？大家也去试验：

```
~A(int a)
{
        printf("fine\n");
}
```

事实就是如此，上面程序编译通不过。

但析构函数是可以在程序中被调用的，可以这样定义：

```
A a;
a.~A()
```

结果就是调用了析构函数，但我们一般不这样做，这也是我们的约定俗成，不需要理由，呵呵，当然就像我们以前说过 main 函数不能被其他函数调用，但我们以前也实践过，其实这一条规则也是约定俗成，真正做起程序来， main 函数一样可以被调用，甚至可以自己一个函数在那儿递归，大家还记得有关死循环的问题吗？

感觉好像讲完了，呵呵，但还有一个问题就是运算符的重载问题，那个问题我们以后单独列为一讲，这一讲就是由于论坛上有讨论，我们才提前讲的，我们下面与大家做几个实践，来看看类的运行过程：

```
#include <stdio.h>
```

```
class A
{
        int x;
public:
        A(int i)
        {
                printf("进入了类%d 的构造函数\n",i);
                x=i;
        }
        ~A()
        {
                printf("准备退出类%d 的析构函数\n",x);
        }
};
void main()
{
        printf("===============进入了程序的主函数==========\n");
        A a(1),b(2),c(3),d(4),e(5);
        printf("---------------准备退出程序的主函数---------\n");
}
```

下面是程序的运行结果：

**引用:**

```
===============进入了程序的主函数==========
进入了类 1 的构造函数
进入了类 2 的构造函数
进入了类 3 的构造函数
进入了类 4 的构造函数
进入了类 5 的构造函数
---------------准备退出程序的主函数---------
准备退出类 5 的析构函数
准备退出类 4 的析构函数
准备退出类 3 的析构函数
准备退出类 2 的析构函数
准备退出类 1 的析构函数
```

下面我们再来改一改主程序：

```
void main()
{
        printf("===============进入了程序的主函数==========\n");
        {
                A a(1),b(2),c(3),d(4),e(5);
        }
        printf("---------------准备退出程序的主函数---------\n");
```

}

再看一看改动后的运行结果：

**引用:**

==========进入了程序的主函数==========
进入了类 1 的构造函数
进入了类 2 的构造函数
进入了类 3 的构造函数
进入了类 4 的构造函数
进入了类 5 的构造函数
准备退出类 5 的析构函数
准备退出类 4 的析构函数
准备退出类 3 的析构函数
准备退出类 2 的析构函数
准备退出类 1 的析构函数
----------准备退出程序的主函数---------

从上面的两个程序的不同，大家明白了几样东西了吗？
一、结构函数什么时候调用？析构函数什么时候调用？
二、构造函数的调用顺序是什么，析构函数的调用顺序是什么？
三、花括号在程序中有什么作用？仅仅是定义一个程序块吗？对函数或者数据或者类与结构的生命周围有没有影响，有什么影响？

呵呵，牛刀好玩吧，可能在某些书上，这些东西是结论，可到牛刀这，就变成了问题了。

事实也是如此，我们实践的目的就是要弄懂这些东西。

最后，我们再来讲一讲文件结构，这里的文件结构不是说文件有什么结构，而是我们怎么来规划与定义我们的一个文件，还是以上的老方法，大家新建文件总没问题吧，呵呵，这里就不详细说了，我们还是用 //----***.h---- 这种结构来说明文件名，也就是大家要定义相应的文件啊，方法还是新建啊，呵呵。

```
//--------------a.h------------
class A
{
        int x;
public:
        A(int i);
        ~A();
};

//----------------test.cpp---------------
#include <stdio.h>
#include "a.h"
```

```
void main()
{
        printf("==============进入了程序的主函数==========\n");
        {
                A a(1),b(2),c(3),d(4),e(5);
        }
        printf("--------------准备退出程序的主函数---------\n");
}

//----------------------a.cpp-----------------------
#include "a.h"
A::A(int i)
        {
                printf("进入了类%d 的构造函数\n",i);
                x=i;
        }
A::~A()
{
                printf("准备退出类%d 的析构函数\n",x);
}
```

大家看，上面的程序被牛刀分成了三个文件，一分以后呢，test.cpp 文件一遇到 #include "a.h" 就会将 a.h 包含进来，也就是说，这一个类就被包含进来了，于是，当一旦使用到了这个类，我们的程序就知道了有这个类，并且还知道这个类就定义在 a.h 这个头文件中，于是，需要执行某个函数时，就自动地到 a.cpp 或者 a.dll 文件中去寻找相应的类函数。这样一来，一个优秀的程序就被不由地分成了三个文件。

理论上来说，我们定义几个类，我们就需要定义相应个数的这样的文件，并且在其中作相应的定义，什么定义呢？最基本的当然是构造函数与析构函数喽。

也正因为这种定义太太太太太太太常用，所以 VC6.0 给我们提供了一个相当优秀并且方便的工具：跟着牛刀来实践——

·按 插入/新建类...
·出现一个对话框，

其他我们先不管，我们先想好我们想定义类的名称，我们假设我们就先定义一个 Point 类吧，于是在 Name: 后的名称中填入：Point

大家一定发现，我们在填入 Point 时，下面的 File name: 同时就为我们生成了 Point.cpp ，可以改变，一般我们不改，直接按 OK

于是，电脑自动为我们生成了一个类： Point，怎么看呢？点击“工作区”中的 ClassView ，我们就可以看到这里有我们定义的类了：

大家看到了吗？我们定义的类 A 也在这里，我们可以点击前面的加号展开：

一旦展开，大家会看到里面的函数，数据定义，全在里面，当然我们关心的不类 A ，而是 Point，我们可以直接双击类，就会看到定义的文件：

```
// Point.h: interface for the Point class.
//
//////////////////////////////////////////////////////////////////////
/
#if !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLU
DED_)
#define AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
class Point
{
public:
        Point();
        virtual ~Point();

};

#endif
// !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUD
ED_)
```

当然，也可以双击里面的函数，来看看函数的定义：

```
// Point.cpp: implementation of the Point class.
//
//////////////////////////////////////////////////////////////////////
/

#include "Point.h"

//////////////////////////////////////////////////////////////////////
/
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
/

Point::Point()
{
}
```

```
Point::~Point()
{

}
```

大家看是不是比我们定义的要复杂多了？我们先来看 Point.cpp 吧，这一个简单，我们将注释全去除，说实话，多的是注释，至少在 cpp 文件中是这样：

```
#include "Point.h"
Point::Point()
{

}

Point::~Point()
{

}
```

好了，这个与我们前面讲的差不多了你看多好，程序直接就给我们的构造函数与析构函数定义好了，这下大家明白为什么牛刀老说我们以后做程序基本上改程序，什么原因了吧，呵呵。 cpp 文件好看，但再回过头来，将我们 .h 文件中的注释也去掉，我们捞干的。

```
#if !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_)
#define AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif

class Point
{
public:
	Point();
	virtual ~Point();

};

#endif
```

大家先看类，类中有一个变化，就是在析构函数前加了一个 virtual ，有关这一点，我们以后在继承那一章再详细讨论，大家明白至少在我们这里有没有无所谓。好了，我们再将类也去了：

```
#if !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLU
```

```
DED_)
#define      AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLU
DED_
#if _MSC_VER > 1000
#pragma once
#endif
#endif
```

这一串代码是干什么用的？预编译，牛刀以前在 infoheader 发过有关宏与预编译的东西时，想详细讲解的，一直没写教程，今天一并与大家讨论了，其中的一段：

```
#if _MSC_VER > 1000
#pragma once
#endif
```

目前为止，牛刀也不明白这一句什么用，有谁知道告诉牛刀一下，但肯定是平台编译遇到什么问题怎么样处理罢了，我们就看剩下的。

```
#if !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLU
DED_)
#define      AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLU
DED_
        ...
#endif
```

显然第一行与第二行的后面一大串东西是相同的，为什么要这样做？下面牛刀与大家讨论一个问题，就是关于文件的问题：

```
class A
{
};

class A
{
}

void main()
{
}
```

虽然允许重载，但像上面那样同名的类，程序中显然不会允许，但如果一旦用多文件，用头包含，以上的情况果真就会发生了：

```
//--a.h--
class A
{
};
//--main--
#include "a.h"
```

大家想，包含什么意思呢？就相当于：

```
//--main--
```

```
class A
{
};
```
一个类不会出现问题，但大家看：
```
//-----b.h----
#include "a.h"
...
//----c.h----
#include "a.h"
...
```
上面都不会出现问题，但一旦我们主函数这样一来，就必然会出现问题了：
```
//----main----
#include "b.h"
#include "c.h"
```
上面的程序就相当于什么呢，我们看：
```
//----main----
#include "a.h"
...
#include "a.h"
...
```
结果，我们包含了两次 a.h 这个头文件，一次是包含 b.h 文件时包含进来的，一次是包含文件 c.h 时，包含进来的。

于是，就出现了我们上面的情况：
```
//-----main-----
class A
{
};
...
class A
{
};
```
我们当然要阻止上面这种情况的发生，那么怎么阻止呢？我们下一讲再与大家详细讨论。

**类的实践〇六〇：合理的使用**

这一讲我们先接着上一讲来讲一讲条件编译，当然，我们不是讲全部的条件编译，我们讲这一讲我们使用得到的，那么什么叫条件编译呢？那就是当我们在不同的平台或不同的文件中，一个程序中的有些东西不同了，怎么办？

举个例子吧：我们暂时不说不同的机器啊，就是不同的平台，就是不同，我们 C、C++中的字符串的结尾是以'\0'作为结尾的，我们称之为 0 串，或者 Z 串，那么在汇编语言中，字串的结尾标志则为'$'，我们称之为$串，那么我们开发一个程序，如果是要对　Z 串处理，很显然用下面的程序就一切 OK:

```
for(...;ch[i]!='\0';...)
{
        ...
}
```

而如果我们在$串中呢？当然也可以这样做程序：

```
for(...;ch[i]!='$';...)
{
        ...
}
```

那么我们看这两个程序之间显然唯一不同的就是后面的那个结束符不同。

我们一句可以像这样地改，我们编写 C 程序处理 Z 串显然没什么问题，可如果我们的程序这样编写是不是更好？

```
#define STREND '\0'
for(...;ch[i]!=STREND;...)
{
        ...
}
```

我们显然知道，这种判断串尾的肯定在一个程序中不是一处，如果是多处，那么我们的程序就很方便了，如果要变成处理$串的程序就只需要改变一下定义宏：

```
#define STREND '$'
```

重新编译就一切 OK 了，有些人不重视宏的使用，其实不对的，我们做一个宏很容易实现一个程序的跨平台，跨系统，甚至运行在不同类型的计算机上，事实上就是如此。

于是，像上面定义宏一样，有好多的这样的处理指令，我们称之为预编译指令，专门处理这些事情的，举几个例子吧，或者是我们这一讲要用到的例子：

**引用:**

#if
#else
#elif

#endif

是不是大家觉得很面熟呢？在哪儿见过？其实就是这样的，但预编译指令不是指令，所以最后不要加分号这些指令都是给编译器看的，大家如果看过牛刀前面的《汇编实践》，大家一定知道在汇编中有“伪指令”这个词吧？其实意思差不多，我们以前学的C的指令都是编译成机器指令的，而预编译指令就是让机器知道我们的意图，从而在编译时有选择地编译，而不是将程序全编译了，说起来很冤，做一大气程序可能绝大多数没有用，谁让我们的计算机不是世界统一的呢？没办法。

上面的几个可能你一看就懂，下面的呢？

**引用:**

#ifdef
#ifndef

我们这一讲就着重讲这两个，其他还有好多的预处理，我们不教大家，至少暂时用不到：

大家知道为什么会有 #endif 吗？我们说过了，预处理的指令不是我们通常用的指令，所以没有语句行的概念，更没有语句块的概念，也就是说大括号不好使，那么这样一来，就必须有替代大括号作用的语句于是就有了 #endif 目的是用来结束上面的 #if 语句条件判断，这一点如果你是学 VB 过来的，那就好理解，容易理解了。

于是，我们举个例子，假如我们（我们说假如，可能系统并不是这样处理，而且肯定不是这样的，因为不是牛刀想当然的，呵呵）定义编译为 c++ 处理时，我们系统定义了一个宏 CPP 而处理 $串则定义了一个 ASM 宏，那么我们上面的程序就可以这样改动：

```
#ifdef CPP
        #define STREND '\0'
#endif
#ifdef ASM
        #define STREND '$'
#endif
```

这样一来，程序会根据我们前面是否定义了相应的宏而自动作出选择，也就是说，我们以后编译时，只需要在编译前，对相应的平台宏定义一下就一切 OK 了。

当然上面的条件编译其实是有问题的，如果系统对 CPP 与 ASM 都作了定义了，那么第一个宏定义与第二个宏定义就犯冲了，系统就会报错。但意思就是这个意思，有关宏与预编译的设计，相当麻烦，甚至有些代码你根本就不知道怎么回事，像前一讲我们讲的:

```
#if _MSC_VER > 1000
#pragma once
```

```
#endif // _MSC_VER > 1000
```

说实话，知道那么一点意思，但你不是系统程序员，就根本不知道这一句的功能。

行了，我们再来看看上一讲的遗留问题吧：

```
...
...
class A()
{
        ...
}
...
...
class A()
{
        ...
}
...
...
```

我们说过，类和函数是允许重载的，但并不是说真能同名，像上面这种同名，编译器显然要报错，那么这种程序我们在单文件的程序中显然不会做出这种类似的程序来，但在多文件的程序中，事实就不同了，一个巨大的程序可能分派给不同的部门，不同的程序员去完成，甚至是一个巨大的群体去完成，你知道人家文件中包含有哪些文件，有没有一个文件在不同的文件中多次被包含，显然是不可能做到不出任何问题的，于是，就必须用了我们这一讲的预编译指令了，我们给类前后加一些预编译的指令，大家看程序：

```
...
...
#ifndef CLASS_A
        #define CLASS_A 12345
        class A()
        {
                ...
        }
#endif
...
...

#ifndef CLASS_A
        #define CLASS_A 12345
        class A()
        {
                ...
        }
```

```
#endif
...
...
```

这样一来，有什么好处呢？大家看，当编译器第一次看到 #ifndef CLASS_A 时，是不是这样来，是不是没有定义 CLASS_A 呢？显然是，既然是，就进入了我们的预编译后的第一个类前面的那个 #define CLASS_A 12345 去执行了，是不？当然这一次一执行过了，这个宏就被定义了，并同时将 class A; 这个类的定义包含了进来。

可当第二次遇到 #ifndef CLASS_A 时，已经被定义了啊，这个宏，所以第二个类的定义在预编译时，就被编译器自动不去处理了，当然后面不管包含多少次，这个类就只处理一次，这就是预编译指令的好处。

当然，这样一来，只要有了这两个预编译，不管我们一个类的文件被包含多少次，处理其实只处理一次，不就无形中实现了我们多文件类与函数的处理了吗？

显然这样一来，会有一个问题，假如我们这个类 CLASS_A 被我们在其他地方不小心定义了怎么办？这个类不就永远别想被包含进来了吗？于是，我们有了一些不成文的规则，大家都根据这个规则来做类，做其他的程序，一般就不会出现问题了：

规则：两个下划线＋文件名＋_H＋两个下划线

其实这个规则牛刀没看有任何书，看有些人这样用过的，像上面 class A ，一般我们类的头文件名称为 a.h 那么定义时，就可以作如下定义：

```
#ifndef __A_H__
#define __A_H__
...
#endif
```

说实话，牛刀没见过任何书上说有这样一个规则啊，但一般我们定义宏不太可能用两个下划线开始去定义，所以如果大家都按这个规则，一般不会出问题。

但 VC6.0 做得就更绝了，大家看：

```
#if !defined(AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_)
#define AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_
...
#endif
```

大家看到 AFX_POINT_H__A70DC0BF_F0E6_4CFB_B110_CEF41DBE0C88__INCLUDED_ 了吧？你有本事做一个宏与之重名？当然这是系统自动生成的，所以我们接上面一讲，我们一般做一个类，都用系统做好的这一个工具，一般不会出问题，并且非常好用。

对了，给大家说一下啊，大家看

```
#define __A_H__
```

怎么没有定义宏的数值啊？其实这是一个空宏（“牛刀定义”，书上可能找不到），我们根本用不到这个数值，所以根本没必要定义数值，当然你定义一个一点关系都没有，但没有任何实际意义。这个宏的定义就是为了预处理，预编译，使得我们的程序编译后不会出现编译错误。仅此而已。

也就是说，一旦我们用系统的方法加进一个类，我们没有任何的后顾之忧。下面再看看，我们在 VC6.0 中，怎么对这个类进行操作。

怎么？类还有什么操作？原来 VC6.0 做得相当成熟，尤其对函数、类的操作，格外地方便，下面跟着牛刀，一步一步地来做：

任务一、给类定义一个变量：

大家打开“工作区”，在其中的 ClassView 选项卡上双击 Point 就可以看到右面给我们列出了如我们上一讲的定义部分，其实定义变量很简单，在类的定义中敲就一切 OK 了。

但牛刀与大家来个绝的，也是常用的：

右击 Point 类，弹出一个菜单，选择：Add Member Variable...

系统会弹出一个对话框：

大家看，挺好玩，我们可以在 Variable Type: 下面的文本框中输入 int 而在 Variable Name: 下面文本框中输入： x ，然后选择下面的 Private ，按“OK”

你瞧瞧，系统自动给我们增加了一个：

```
private:
    int x;
```

算了吧牛刀，这不是脱裤放屁——多此一举吗？

还真别说，果然是多此一举，但有些人就在这上出了点问题，特别是在 MFC 中，有些人根本不敢去改系统的任何一个数据或者变量，弄得太小心了，MFC 书看多了可能有些人就不太敢编程了，事实真的会如此，所以牛刀和大家说，这个功能，建议大家不要用，还不如直接在类中输入：

```
    int y;
```

你看，多方便？

但大家想过没有，你如果真在做一个很大的程序，又想加一个类的全局变量，你舍不得离开自己做程序的地方啊，于是想用了，还是会不由地右击一下，加个想要的变量里面去就一切 OK 了，用这个右键的好处就是，根本不需要去类中找地方插入代码，随时随地都可以添加任一变量，实话说，你如果是学习的话，果真是脱裤放屁，但你真的做一个大型的程序，你就会体会到用这些工具有多么的方便了。

大家可知道，我们对类的编程不是在同一个文件中进行的啊，一个是头文件，一个是类文件，如果我们想要加入一个变量就要打开一个文件你说烦不烦？当然这里空口说可能没有说服力，大家在做程序的过程中细细体会吧。

任务二、增加一个函数：

同样，右键类图标，按 Add Member Function...

也是弹出一个对话框：

大家看结构与上面几乎如出一撤：
在 Function Type: 下面输入 void
在 Function Declaration: 下面输入 print()
选择 Public 点 “OK”

这里面说一下，下面多了一个 Static 和 Virtual ，一个是定义为固定函数，一个是定义为虚函数，这些我们以后再说。

增加函数，我们一般不手工添加，为什么？这个工具方便啊，函数原型与函数调用一并给我们加进来了，并且给我们的光标直接放到函数编写的代码中，我们只需要定义一下，顺便就可以直接敲进代码了。

```
void Point::print()
{
        printf("x=%d\ny=%d",x,y);
}
```

多省事？事实就是如此，我们一般做一个类都是使用这两样工具，而不是自己去一个一个地敲：

```
class A
{
public:
        ...
}

A::A()
{
        ...
}
...
```

等我们程序做多了，我们可以很自然地想到 VC 的好处的。

什么是类？什么是对象，什么是结构？什么是指针？好多问题，一并向我们涌来，这一讲我们不打算再讲新内容了，这一讲给大家理理头绪。

类，就是由 class 定义的一个特殊的结构体，这个结构体可以实现函数的封装，默认的结构类型为 private: 但我们使用时，一般一开始都加上一个 public: 所以说，我们的类与结构体，理论上说没有区别，就是一堆函数与数据的组合。

什么是对象？这一点在 VB 那里说得尤其得玄，什么说 VB 不是面向过程了，是面向对象了，我们做程序就是需要对一些对象设计一些属性，然后捕捉一些事件，使用一些方法而已，是那样吗？

其实 VB 中的属性就是我们类中定义的数据，而 VB 中的方法就是我们类中定义的函数，而 VB 中的事件就是我们调用这个类的一些条件。

我们在一个控制台程序中，调用类，当然是随时随地，但如果在 Windows 程序中，则一般是在一些消息捕捉中去调用某个类，于是，VB 就称是什么事件，遇到 WM_MOUSEMOVE 消息，VB 中就结出了一个事件：

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)

End Sub
```

遇到了 WM_KEYDOWN 消息，就给出一个事件：

```
Private Sub Form_KeyDown(KeyCode As Integer, Shift As Integer)

End Sub
```

事实就是如此，我们在这些事件中处理一些信息与在相应的消息处理函数中处理有什么不同呢？唯一的不同就是，我们在那里面对的是 WPARAM LPARAM 两个晦涩的变量，而 VB 中做得则特别得人性化。

到此，大家明白牛刀为什么不承认有“面向对象”这个概念了吧？但我们在编程过程中还必须得提，那么什么是对象呢？类不是对象，但一旦定义了一个类，我们就说我们定义了一个对象，从这一点上来说，C++中的对象比 VB 中要丰富了许多：

```
class A
{
        ...
}

void main()
{
        A a;        //在这里，我们说我们建立了一个对象 a，而创建这个对象的过程，我们称之为对象的实例化。
        ...
}
```

那么我们为什么要实例化一个对象呢？因为我们需要用到类中的函数与数据，就必须要实例化一个对象，使得我们的类能够变成可以操作的东西。

什么意思呢？我们要盖一坐房子，就必须有一个图纸，那个图纸，就好像我们这里的类，而那个房子，则像我们这里的对象，图子的功能越全，那么房子的功能就越多，但图纸却不能住人，道理就是这样的。

所以我们定义任何一个类，任何一个结构体，在定义中，永远不能变成现实，必须将其实例化。

前面教大家 API 编程时，大家还记得么？我们定义并赋值了一个结构：WNDCLASS ，但那并不是窗体，我们必须调用函数去将这个窗体变成一个现实的窗体，那也就叫实例化，每实例化一个窗体，就需要使用一个实例句柄 HINSTANCE ，这是为什么，现在可以与大家挑明了，其实就是说，系统有这么多资源，你每次是对什么资源进行操作，必须要让系统知道，就是这个意思，大家看我们好多 API 函数都要用到窗口句柄 HDC 就是要让系统明白，我们要画的图，要画在哪一个窗口上，否则不乱套了吗？

扯得有些远了。下面我们就讲讲类的指针的问题，我们可以定义一个类的指针，与结构指针一模一样，所以说，我们可以很容易地使用类去实现链表操作树操作，甚至图操作（如果你不懂没关系，我们在此课程中，暂时也不讲，叫管理员开个《算法》版块，管理员也不开，呵呵），而这些操作却可以直接封装在类里面，对外提供一些接口函数就一切 OK 了，于是，我们创建一个链表，可能只需要定义一个类，然后用类中的函数就可以自由地实现插入与删除操作，而对这些操作是怎么实现的，那都是由类去自己完成了。多方便啊，我们看一段 VB 的程序：

```
Private Sub Form_Load()

    List1.AddItem ("牛刀测试一")
    List1.AddItem ("牛刀测试二")
    List1.AddItem ("牛刀测试三")
    List1.AddItem ("牛刀测试四")
    List1.AddItem ("牛刀测试五")
    List1.AddItem ("牛刀测试六")
    List1.AddItem ("牛刀测试七")
    List1.AddItem ("牛刀测试八")
    List1.AddItem ("牛刀测试九")
    List1.AddItem ("牛刀测试十")

End Sub
```

为什么好多人都说 VB 好学， VB 方便，果真方便啊，就是这个原因，你看上面区区几行的代码，在 VC 中你一个类不就得做半大天啊？并且还不能保证你类中没有错误。

而 VB 中也是使用的类，但人家不那样说，人家说那是对象，而 AddItem 那是方法，你怎么说？事实上，VB 中的好多东西都是在后台被封装了，不能我们看不见就说人家这个方便，你知道牛刀以前做 VB 时多辛苦吗？大家知道，

VB 程序是不能直接运行的，必须运行的电脑上要有 VB 运行库，而一旦这些东西打包，那至少是 3M 多的空间啊。

牛刀那个时候就满腹的疑惑，怎么同样的程序，我做就是 3M 多，而人家却能够区区几 K 的程序也能运行得很好呢？

DELPHI 不知怎么样？菜新版主也老长时间不上这个版块上来了

事实就是如此，所以我们以后也会与大家做我们自己的对象，而且我们以后的思路最好能够适应这种面象对象的思想。为什么牛刀要一讲一讲地与大家讲基础，这一讲者六十讲了，就是要让大家成为两栖程序员，过程程序我也会，面向对象我也行，什么都可以赶行势，赶潮流。

好多程序员面对这种面向对象犯愁的，也有好多新程序员遇到为名人机编程那样的面向过程不习惯，这些都一点不奇怪，这就需要大家一定要打好基础，有了扎实的基本功后，再去用新眼光去看面向对向。

最后再说一句，我们可以像上面那样去定义一个类，但是，我们如果要用到这个类，只需要用 #define 语句将这个类的头文件包含进来就一切 OK，甚至我们只要愿意，包含多少次都无所谓，如果你知道了为什么，那么我们这一讲的目的也就达到了。

从此，我们也可以面向对象了，面对对向，我们能够做什么呢？

我们可以定义这个对象的属性，我们可以定义对象的功能，下面，我们就从一个对象的开始，为大家做一个没用的程序吧：

```
#include <stdio.h>

class A
{
};

void main()
{
        A a;
}
```

事实就是如此，我们以后做的程序中 main 主函数可能显得异常的简单，就是定义一个对象，然后调用几个函数，就一切 OK 了，而对于程序功能的组织，则全部在对象中就完成了。

那么，我们问一个问题，是不是说，这个类就空空如也了呢？其实不然，类中还有两个函数，一个是构造函数，一个是析构函数，只不过这两个函数也都是空函数而已，而这两个函数的确存在。

对类有一个默认规定，就是默认有一个构造函数和一个析构函数，如果用户定义了这两个函数，就使用用户的，而用户如果不定义这两个函数，那么这两个函数就是空函数，可能你看源代码看不到，但编译器在编译的时候给自动加上，所以我们不要认为我们定义了一个空类，我们是不可能定义一个空类的。

关于结构体，是不是也有默认的构造函数与析构函数，那牛刀就不知道了，真的不知道，如果有谁见到相关的资料告诉牛刀一声。

好了，上面就是一个程序，虽然什么功能都没有，但这个程序的确可以运行了，虽然没有任务反应，但至少有三个函数，一个主函数，一个类的构造函数与一个类的析构函数。

也正因为如此，我们用 VC6.0 的添加类的功能添加了类时，同时系统就给定了这两个默认的函数，用不着可以不用去管她，如果用得着，你可以可以去改变她，甚至你可以继续去重载她，一样的。

行了，这一讲我们就要结束了，但在结束之前我们说一声，我们以后为大家做的类都是单文件的类，因为我们讲原理没必要弄那么多的文件，再用包含给头文件包含进来，我们以后主要是讲原理，其实我们这一讲以及前面也是一样，没有多少实用的程序，但正是这些没用的程序，帮我们理解了好多的知识点，并且我们以后的程序，也都要从这些没用的程序开始。我们做一个程序，就是需要这个程序实现好多好多的功能，但一旦我们要做这个程序了，我们总要有主函数，并且在主函数中要干什么我们总要知道是吧？

所以在以后的教程中，我们会就面向对象与大家重新讨论一些编程的思考，希望大家做好迎接面向对象的准备。

最后说说为什么我们要面向对象，其实微软做这个事情就是想要使得程序的代码可以重用，也就是说，我们做一个代辛辛苦苦，不要浪费了，如果以后我们再需要做相同程序时，顺手拿过来使用就一切 OK 喽。例如，我们定义了一个类：

```
//-----------circle.h--
class Circle
{
        float r;
public:
        CirCle(float a)
        {
                r=a;
        }
        float GetArea()
        {
                return 3.1415926*r*r;
        }
}
```

这是定义一个圆的类，那么我们以后只要用到圆，我们就随手就可以将这个类包含进来，就可以顺手得到面积，当然你也可以做一个求周长的函数，那么我们以后只要需要求一个圆的面积，我们就可以很容易地写出简练的代码：

```
#include <stdio.h>
#include "circle.h"

void main()
{
        Circle cir(2.4);
        printf("半径 2.4 的圆的面积为:%f\n",cir.GetArea());
}
```

你瞧，我们的代码多简练，当然喽，我们可以顺手做一个库，将圆、矩形、三角形、梯形等等好多东西的面积都包含进去，做成一个很大的库，使得我们以后想求什么面积时，做一个对象，这个对象就可以告诉你这些图形的一些参数了，多方便，

类与对象的技术，可以很容易地将我们学到过的微积分、图形技术、图像技术……很容易地做成一个很大的库，而且有好多库是做好的，技术成熟的，大家知道 OpenGL 吗？大家知道 DirectX 吗？是 3D 库是吧，玩过游戏的朋友都知道，好多的游戏都需要有这些库的支持是不是？因为好多游戏的代码就是用这些库中提供的函数编写的，而这些函数就存在那些库中。

岔开了，其实我们以后用到的更多的库就是 MFC 类库，我们等一些基础打扎实了，我们做程序几乎都是在那里完成。

好了，这一讲就到此结束了，希望大家在以后做程序时，要明白我们的程序是想用面向对象的思路还是面向过程的思路去做我们的程序，千万不要搞混了。

**类的实践〇六一：奇妙的父子**

父子关系，大家都不陌生吧？想必大家不陌生（那是一定不陌生，呵呵），计算机界中好像有点男权思想，全是男性，几乎不用女性，不知道那父子的子是怎么来的。

不管怎么来的吧，我们类中的关系就称之为父子关系，当然，你也可以称之为母女关系，一样的，大家明白意思就行，牛刀这儿什么都有，不拘泥于书本的称法。

那么怎么定义父子关系呢？一个冒号“:”，看程序：

```
#include <stdio.h>
class A
{
};
class B:public A
{
};

void main()
{
        B b;
}
```

这个程序会报错，不过这种是可容忍的错误，也就是说，编译器没有发现 B 中有任何的东西，其实也就是这种类为空类，系统还是提出来了，但不代表提出来了就是错误的，系统在编译后仍然为我们加了构造函数与析构函数，只不过是系统感觉有些莫名其妙，这个类给我干嘛？什么东西都没有？就是这个意思。

大家看，我们说上面的程序就是类 B 继承了类 A，很简单吧？其实就是很简单，我们明白了，知道了就行，别把书上的一些定义理解得多透彻，有时候我们知道一些东西就行，至于书上怎么说的，说实话有些真的是文绉绉的，什么类的继承啊，然后给出一套定义，再讲一套规则，其实没必要。大家看上面，我们在继承 A 时，前面加了 public，意思是公共继承，什么意思呢？我们以后再说，暂时我们只讨论公共继承。

那么这种继承会有什么结果呢？我们将其构造函数与析构函数都写出来，一运行就明白了：

```
#include <stdio.h>
class A
{
public:
        A()
        {
                printf("进入了 A 类的构造函数\n");
```

```
        }
        ~A()
        {
                printf("进入了 A 类的析构函数\n");
        }
};
class B:public A
{
public:
        B()
        {
                printf("进入了 B 类的构造函数\n");
        }
        ~B()
        {
                printf("进入了 B 类的析构函数\n");
        }

};

void main()
{
        B b;
}
```

其实相应的构造函数与析构函数中只是提示，不过提示就可以帮助我们理解，其实这就是实践，比去啃书上的概念要舒服多了，你说是吗？看结果：

**引用:**

进入了 A 类的构造函数
进入了 B 类的构造函数
进入了 B 类的析构函数
进入了 A 类的析构函数

感觉上一个函数都没调用，没想四个函数全调用了，大家看，先是进入了 A 的构造函数，后进入 B 的构造函数，析构时正好相反。什么意思呢？也就是说，这种继承时，我们想要定义一个子类，就必须先构造出父类，再根据父类构造出子类，为什么会这样呢？原来啊，这种类之间的关系还真有生物学上的遗传特性，真因为需要遗传，所以就是必须要先将父类有了，才可以将其特性遗传给子类。还是举例子吧，我们在父类中定义一个函数啊：

```
#include <stdio.h>

class A
{
public:
```

```
        void print()
        {
                printf("这是A类（父类）的print函数\n");
        }
};
class B:public A
{
public:
        B()
        {
        }
        ~B()
        {
        }

};

void main()
{
        B b;
        b.print();
}
```

大家看，子类 B 类并没有 print 函数，可以我们的确用了，那么这个函数是哪里来的？父类的，也就是说，是继承过来的，这就是继承的由来。于是，我们可以想当然，结果是：

这是 A 类（父类）的 print 函数

事实就是如此，为什么微软做 C++的类时，要这样做呢？一切都是为了代码重用。

不用说了，父类（A）的代码“print 函数”，在子类 B 那儿得到了重用。事实这样的确挺好，你想想啊，做一个很大的软件，一旦我们的代码用一次就没用了，多可惜啊，你想一个优秀的代码至少是以万行为单位的啊，顺手一丢就没用了，你觉得不可惜吗？于是，继承就很好地解决了这个问题，当然喽，有继承就有变异（很像生物学中的遗传与变异似得吧，呵呵）。如果我们觉得父类中的某一个函数做得不好，我们可以在子类中重做一个一模一样的函数，千万注意，这就是同名函数，而不是重载，在 C++中规定，在子类中可以做与父类一样的同名函数，如果某一函数子类中如果有了，就不再调用父类中的函数了。这是不是很像变异？一猪生九仔，连母十个样啊，呵呵，我们试试：

```
#include <stdio.h>
class A
{
public:
```

```
        void print()
        {
                printf("这是 A 类（父类）的 print 函数\n");
        }
};
class B:public A
{
public:
        void print()
        {
                printf("这是 B 类（子类）的 print 函数\n");
        }
};

void main()
{
        B b;
        b.print();
}
```

结果也可以想当然地知道：

这是 B 类（子类）的 print 函数

这种变异使得我们有改变父类功能的机会，这一点其实特别有用，特别是当父类被封装成了 .dll 文件时，我们在子类中几乎是为所欲为。父类好的，我们就用父类的了，父类不好的，顺手就给改了，所以说 C++的这一特点就使得我们的程序总是最好的，最棒的。

你信吗？如果别人最好的你都有了，你还有比别人强的地方，那你就是最棒的，真的，这一规定使得我们 C++的程序几乎可以做得相当得完美。

也正因为如此，才使得我们用别人的 MFC 时可以做到想怎么改就怎么改，如果这一技术能用得恰到好处，那么我们做 MFC 几乎是没什么真正的技术难题了，当然我们不举 MFC 的类子，我们还举个 控制台中的例子吧：看看我们是怎么去修改一个程序以达到我们的完美的。

```
#include <stdio.h>
//我们假设下面的类在一个 .dll 文件中，我们不能够动她：
class A
{
public:
        void test()
        {
                printf("DLL 文件中的测试函数\n");
        }
```

```
};

//我们假设下面的主函数在一个  cpp  文件中，当然真正做程序时，需要头包含，我们在这就不定义了：

void main()
{
        A a;
        a.test();          //我们调用 a 的某一个测试工能，来完成我们需要类 A 做的工作
}
```

我们假设我们对 A 中的内容不能更改了，（事实上你以后真正做程序时好多的东西就是如此），但我们又对 test 程序不满意，我们怎么办呢？假如你遇到这种情况时怎么办？

我们加一个类 B ，让其继承 A ，然后在 B 类中再重新定义一个 test 函数即可改变 A 中原有的 test 功能，看程序：

```
#include <stdio.h>
//我们假设下面的类在一个  .dll  文件中，我们不能够动她：
class A
{
public:
        void test()
        {
                printf("DLL 文件中的测试函数\n");
        }
};

//我们加进一个类  B  ，使得类  B 继承  A  的所有函数，有时候当别人的儿子真好，呵呵（开玩笑，别当真）

class B:public A
{
public:
         void test()                    //我们再在  B  中重新定义一个 test  函数
        {
                printf("这是我们更改后的测试函数\n");
        }
};

void main()
{
//        A a;
```

```
        B a;            //当然这里也要改了，但注意：变量 a  不能改，我们举例时，只用了 a  中的一个函数，但如果我们真正做软件就不是这样了。
        a.test();       //我们调用 a 的某一个测试工能，来完成我们需要类 A 做的工作
}
```

以上的东西，我们真的可以称之为“技术”，至少说明我们的继承真的十分有用。曾经看过人家将 MFC 生成的程序中的按钮显示的一些特性给改了，改得十分可爱，用的其实就是这种技术，如果不懂得继承，直接去看别人的代码，说实话，真的会十分难懂，甚至你根本不知道别人在干什么，给你代码你也不明白，可能每一句都看得懂，但确不明白这段程序要干什么。

说实话，也正因为有了继承， MFC 中几乎什么东西都能改，你瞧瞧，就这样一个很简单的技术，几乎可以颠覆现有的已经做得很成功的 MFC 类库，你说用这种 C++ 技术去做软件，是不是一定是最优秀的呢？

从此以后，我们做软件几乎没有门槛，看到好东西就拿过来，见到不满意的就是一个字——改，管她是在代码中，还是已经变成了二进制文件 .dll 文件了呢？

行了，好处说过了，C++的这种继承技术也不是全是好处的，有时也会出问题，什么问题呢？乱伦：

```
#include <stdio.h>

class G
{
public:
        G()
        {
                printf("我是爷爷我怕谁？\n");
        }
        ~G()
        {
                printf("爷爷的骨头敲鼓了。\n");
        }
};
class F:public G
{
public:
        F()
        {
                printf("我是爸爸我怕谁？\n");
        }
        ~F()
        {
```

```
                printf("爸爸翻白眼了。\n");
        }
};
class M:public G
{
public:
        M()
        {
                printf("我是妈妈我怕谁？\n");
        }
        ~M()
        {
                printf("妈妈的腿伸直了。\n");
        }
};
class X:public F,public M
{
public:
        X()
        {
                printf("我是小明我怕谁？\n");
        }
        ~X()
        {
                printf("哟，谁都看不见谁啦！\n");
        }
};
void main()
{
        X x;
}
```

以上一下子给出了四个类，大家不会迷糊吧？其实挺简单，上面的程序来自于一个笑话：

**引用:**

大年三十晚上，小明与爷爷、奶奶、爸爸、妈妈五个人坐在一个桌子上吃饭，（注意啊，我们讲的古典笑话，那时候还没有春节联欢晚会呢。）吃肉，当然要吐骨头的啦，于是，爷爷顺手将一块骨头往地上一扔，没想正好扔在一面鼓上，“咚”，于是小明就说：“呕，爷爷的骨头敲鼓喽”，爸爸当然很生气喽，于是用眼睛瞪了小明一眼，小明连忙又说：“爸爸翻白眼了”，当然有人更生气，妈妈于是在桌子下面狠狠地用脚蹬了小明一下，意思很明显，你瞎说什么？可就明毕竟是小明，语出惊人：“妈妈腿伸直喽”，奶奶在一边忍不住了，一笑，将桌上的蜡烛吹灭了。于是小明又叫了起来：“哟，谁都看不见谁啦！”。

当然，要是有电灯就吹不灭啦，谁让这事总是让小明一家在大年三十的晚上赶上了呢，呵呵。笑话归笑话，我们来看看运行的结果：

**引用:**

我是爷爷我怕谁?
我是爸爸我怕谁?
我是爷爷我怕谁?
我是妈妈我怕谁?
我是小明我怕谁?
哟，谁都看不见谁啦!
妈妈的腿伸直了。
爷爷的骨头敲鼓了。
爸爸翻白眼了。
爷爷的骨头敲鼓了。

其实我们最关心的就是结果了，大伙说是不是一个爷爷，一个爸爸，一个妈妈，一个小明，你看爷爷与爸爸、爷爷与妈妈，都是父子关系，而爸爸妈妈又与小明是父子关系，是不是有点“乱伦”了？

在结果中，我们也发现，爷爷这个类被构造了两次，又敲了两次的鼓。

这就是我们在 C++ 继承中最著名的钻石结构，这种结构给程序带来了很大的歧义，看程序：

```
#include <stdio.h>

class G
{
public:
        void test()
        {
                printf("这是爷爷的  print  函数\n");
        }
};
class F:public G
{
public:
        F()
        {
        }
        ~F()
        {
        }
};
class M:public G
```

```
{
public:
        M()
        {
        }
        ~M()
        {
        }
};
class X:public F,public M
{
public:
        X()
        {
        }
        ~X()
        {
        }
};
void main()
{
        X x;

        x.test();
}
```

这个程序一运行就报错，为什么？因为编译器根本不知道这个 test 函数是从爸爸那儿继承爷爷过来的，还是从妈妈那儿继承爷爷过来的，所以就报错了呗。

对于这结构，其实我们定义 x 时并没问题，就是不能调用 test 函数，为了说明问题，我们将例子简化，再给大家做一个程序：

```
#include <stdio.h>
class A
{
public:
        test()
        {
                printf("A::test();\n");
        }
};
class B
{
public:
        test()
```

```
        {
                printf("B::test();\n");
        }
};
class C:public A,public B
{
public:
};

void main()
{
        C c;
        c.test();
}
```

这个程序比上面的结构要明显多了，就是继承了两个父类，但两个父例中却都有 test 函数，你叫人家 c 调用哪一个？如果是你去调用，你会调用哪一个呢？

这种问题解决有两个办法：

一、显式调用：

```
void main()
{
        C c;
        c.A::test();
}
```

显然这种调用方式挺别扭，呵呵，但没有办法，谁让其有同名函数呢？只有将其类也带上了呗。

二、在子类中定义一个同名函数，由函数来决定调用哪一个：

```
class C:public A,public B
{
public:
        test()
        {
                B::test();
        }
};
```

根据同名覆盖的原则，子类如果有同名函数，则系统就不会去调用父类的相应函数了，所以说，这样一来就不存在那种相同函数继承的问题了。

但好玩的是，上面的程序，我们编译器是报错的，但这种错误是编译器可以容忍的错误，毕竟你是真的同名了嘛。

行了，我们再讲一个问题，让大伙来看看我们编译器是有多么的愚蠢：

```
#include <stdio.h>
class A
{
private:
        int i;
public:
        test()
        {
                printf("A::test()\n");
        }
        test(int a)
        {
                i=a;
                printf("A::test(int a)\n");
        }
};

class B:public A
{
public:
        test()
        {
                printf("B::test()\n");
        }
};

void main()
{
        B b;
        b.test(5);
}
```

上面一程序，编译器报错，说找不到相应的其中含有一个参数的函数，为什么会出现这种情况呢？我们在父类 A 中，不是对 test 函数进行了 test(int a); 重载了吗？为什么没有呢？

原来啊，编译器其实是一个笨家伙，也是一个懒家伙，当调用 b.test(5); 时，我们说过，子类中如果有同名，就同名覆盖父类吗？其实哪是那么回事啊，其实编译器一旦在子类中发现相应的函数，就不再往父类中找相应的函数了。

什么意思呢？子类中的了 test 函数啦，于是父类在这个函数中就好像不存在一样，编译器根本理都不理，好，那就是说，要在子类中找 test(int a); 函数，哪里去找得到啊。

所以我们说，这种同名覆盖一下子就将父类中的好多重载的东西给覆盖了，

使得我们想使用父类的也没办法使用了。怎么办？

办法有二个：

一、在子类中定义一个同名重载函数。

在这就不举例了，想必大家代码都写得出来。

二、使用关键字 using

说实话，这种技术也是在网上刚看过不久，这叫现炒现卖，呵呵：

```
class B:public A
{
public:
        using A::test;
        test()
        {
                printf("B::test()\n");
        }
};
```

以上一经定义，其实就是告诉了编译器，如果在本子类中找不到相应的函数或函数重载，请到父类中再找找看，别忙着报错。其实事实上就是这样。

**引用:**

重载，让我欢喜让我忧。
同名，让我担心让我怕。
说实话，挺复杂。
让人恼，让人繁。
出了一个爱报错的编译器。
你说怎么能够快快乐乐，
安安心心，
从从容容，
踏踏实实，
舒舒服服地坐在电脑前？

学习本是苦差事，
知识好比摩天塔。
根基若要扎不稳，
完完。

水滴石穿非一日，
冰天雪地三月寒。
哪能一口吃个胖？
总得消化小半年。

若要编译不报错，
努力学习与实践。
各种情况我知道，
那么编程有何难？

多问几个为什么？
多多坐在电脑前。
打开Ｖ６编译器，
玩。

说实话，中国古代有一句话，叫做——尽信书不如无书。其实也是这样，我们做程序更是这样，我们的知识其实是实践得来的，而书本上只不过是前人的总结，至于总结的对于错，那只能说是书本的一面之词。（有人问，什么叫尽信书不如无书呢？另一个回答：《尽信书》是一本好书，但还是没有《无书》这本书好）

当我们实践了，我们就会知道哪些是我们需要的，哪些是错误的，哪些是允许的，我们心里明镜似的。牛刀其实早就说过，有些书本，那其实就是理论形态，而不是实践形态，给我们规定了一套的规则，要我们去遵守。

其实如果我们真的去遵守那一个个的规则，那我们都成了木头人了，如果我们都遵守了那一套规则，那黑客就不可能存在了，如果我们都遵守那一套规则，那么就不会存在网络安全问题与病毒问题，你们说是么？

书上规定，数组不能越界，可现实却证明，数组越界可以实现我们某种不可告人的目的，那么你就很可能去干一些坏事。

对了，听过单田芳的评书《封神演义》吗？　×××拿了一个葫芦，嘴里念动五字真言，哪五个字呢？我不能告诉你，告诉你你拿个葫芦出去干坏事咋办？.....

可事实上，有些人干的坏事却比那个葫芦要可怕多了。你知道一个病毒做起来不费劲，可一旦上网上传播一下，全世界要损失多少个亿吗？（可惜啊可惜，牛刀没做过病毒，也没人给牛刀多少个亿，哎～～～）

最手的时间用类给大家做一个游戏，其实什么叫游戏呢？其实很简单，就是我们游戏的电脑翻版，我们有时候的游戏规则是很简单的，我们做个什么游戏呢？猜拳吧，大家知道呢？剪子——包子——锤，或者又叫剪子——锤子——布，其实一样的，就是剪子剪布，布包锤子，锤子砸剪子吧。

既然要用到这三个东西，我们定义一下他们之间的关系与类型：

```
#include <stdio.h>
#include <stdlib.h>
enum CQ {JD,CZ,BU};
```

```
class Db        //赌博
{
private:
        CQ cqc;         //猜拳计算机
        CQ cqm;         //猜拳人
        int Gc,Gm;//计算机与人所有的金币数目
        int Jy;         //每次交易的数目
public:
        Db(int a,int b)
        {
                cqc=(enum CQ)(rand()%3);
                Gc=a;
                Gm=a;
                Jy=b;
        }
        int Bj()        //比较胜负
        {
                  if((cqc==JD && cqm==BU) || (cqc==BU && cqm==CZ) ||
(cqc==CZ && cqm==JD))return 1;
                  if((cqc==JD && cqm==JD) || (cqc==BU && cqm==BU) ||
(cqc==CZ && cqm==CZ))return 0;

                return -1;
        }
        void Setcqm(int a)
        {
                cqc=(enum CQ)(rand()%3);
                cqm=(enum CQ)a;
        }
        int print()
        {
                int i;
                printf("计算机现在拥有%d 金币，你现在拥有%d 金币，每次
赌金为:%d\n",Gc,Gm,Jy);
                switch(cqc)
                {
                case JD:
                        printf("计算机出的是剪刀，");
                        break;
                case CZ:
                        printf("计算机出的是锤子，");
                        break;
                case BU:
```

```
        printf("计算机出的是布，");
        break;
}
switch(cqm)
{
case JD:
        printf("你出的是剪刀。\n");
        break;
case CZ:
        printf("你出的是锤子。\n");
        break;
case BU:
        printf("你出的是布。\n");
        break;
default:
        printf("你出错了，重出。\n");
        printf("请你出拳（1--剪刀 2--锤子 3--布)：");
        scanf("%d",&i);
        Setcqm(i-1);
        return 2;
}
if(Bj()==1)
{
        printf("计算机赢了\n");
        Gc+=Jy;
        Gm-=Jy;
}
else if(Bj()==-1)
{
        printf("你赢了\n");
        Gc-=Jy;
        Gm+=Jy;
}
else
{
        printf("你们平手，再来。\n");
}
if(Gc<Jy)
{
        printf("计算机的钱不够了，你赢了!!! 恭喜\n");
        return 0;
}
if(Gm<Jy)
{
```

```
                printf("你带的钱不足，回家再练练吧，再见\n");
                return 0;
            }
            printf("请你出拳（1--剪刀 2--锤子 3--布)：");
            scanf("%d",&i);
            Setcqm(i-1);
            return 1;
        }
};
void main()
{
        int iz,im,i;
        printf("请输入你们要赌博的总钱数：");
        scanf("%d",&iz);
        printf("请输入你们每次要赌博的钱数：");
        scanf("%d",&im);
        Db db(iz,im);
        printf("请你出拳（1--剪刀 2--锤子 3--布)：");
        scanf("%d",&i);
        db.Setcqm(i-1);
        while(0!=db.print());
}
```

上面做的类中的 print 函数很像一个消息循环，不断地执行，直到返回值为 0，即没钱了，停止。

当然我们还可以多定义一些函数，使得计算机出拳的整个过程都有相应的函数控制，甚至可以控制每次出钱的次数，那就是我们编程者是怎么样构思的了，牛刀这里用了好多汉语拼音的头字母，你也可以改成英文头字母，一样的，只要有利于我们编程与思考，怎么来都可以，甚至你所有的函数都统一地编成 fun1 fun2 fun3...都没关系，只怕如果那样一来，计算机运行起来没问题，你自己把自己搞糊涂了，呵呵。

行了吧，其实这个游戏再做一些很简单的修改，就可以拿到名人机上去吹了，不又是一个名人机的小游戏，呵呵，牛刀近来为名人机做的程序少了，倒是 MJ_BASIC 的天下了，哎～～～

**类的实践〇六二：假假的函数**

假不假，白玉为堂金作马……

不对，应该是：假做真来真亦假……

也不像，应该叫做：一把辛酸泪……

嘿，没文化就别装着有文化，那哪叫，假做真来真亦假，那应该叫真做假来假亦真……

哪儿挨哪儿啊？牛刀也弄糊涂了，反正我们这里决定不用“假”了，我们决定用“虚”，看你说什么？

虚函数也是C++中一道靓丽的风景线，其实大家还记得我们上一讲讲的有关“钻石结构”的类定义吗？其实哪是钻石啊，纯粹是“乱伦”，那些写书的人真能美化事实，还“钻石结构”，亏得他们想得出来。

那种结构的确让人繁，后来有人想到了“虚函数”，可惜也没有什么大用处，但却给我们一个做程序的新思路。

大家以后做程序不是万不得以千万别弄出那种结构来，因为有好多的东西你真的不容易处理得好，你想想如果谁一下子出来两个爸爸，那还好得了吗？可惜在电脑里，却变成了现实。

我们一般在做类时，做成树型结构，当然有两种树，一种叫正树，一种叫倒树，正树的话就是一个子类有多个父类的情况，我们这种思想可以为我们将一个很巨大的程序构思逐步简化，变成一个一个小的模块去实现，其实牛刀估计（注意是估计啊，没有任何现实依据，不负法律责任）：VB 可能就是这种结构，将什么东西的运行过程都做好了，你看什么父亲要变动，你去动吧，一动，儿子的相应的属性就变了，多可怕啊，父亲一旦变成蓝眼珠，儿子立刻就变了，比现实中的遗传规律要强了很多，呵呵。

另一种叫倒树，就是一个父类派生出若干个儿子，一个儿子又派生出若干的孙子，子子孙孙无穷尽也（——摘自《愚公移山》），牛刀猜想（注意是猜想啊，没有连带责任）：ＶＢ可能是这种结构吧？一个父子统率一帮的儿子，父亲发指令给儿子去做，当皇帝的滋味真好受（哪儿又冒出一个皇帝出来啊，哎～～～）。

牛刀你忽悠谁呢？怎么 VB 一会儿正树一会儿倒树啊？谁知道呢？应该是一种倒树吧。（什么叫应该，准确点）

冷静，冷静一点，牛刀也被搞糊涂了，什么倒树正树，还要“乱伦”什么玩艺啊，哎～～～。

呵呵，其实 VB 也是一个类派生出来的，应该是一种倒树结构，毕竟 VB 也

是微软出品，其基本思路与 VC 相差也不会太远的，而我们的 VC 中的 MFC 的结构，给我们展现了一个很壮观的大树：

**引用:**

CObject 树根，节点举例：
CCmdTraget CFile CDC CArray .....
CCmdTraget 节点举例：
CWndThread CWnd .....
CWnd 节点举例：
CFrameWnd CDialog CView CAnimateCtrl .....
.....

不举例了，没意思，再举将我们这一章全举了也有什么意思？意思到了就行了。既然这一讲我们要讲“虚函数”，我们就是建立在这种倒树结构或者是一种直线结构的基础上的，由于树型结构过于复杂，所以我们讲解时，只使用直线结构的直接继承来与大家讨论一下“虚函数”的作用。

原来啊，“虚函数”是专门为“指针”设计一种特殊的函数，我们看程序：

```
#include <stdio.h>
class A
{
public:
        A()
        {
        }
        ~A()
        {
        }
};

class B:public A
{
public:
        B()
        {
        }
        ~B()
        {
        }
};
class C:public A
{
public:
        C()
```

```
        {
        }
        ~C()
        {
        }
};
void main()
{
        B b;
        C c;
        B *pb=&b;
}
```

以上我们定义了三个类，一个父例有两个子类，说开去了点，其实这种一父二子的结构比较好，中国的计划生育的政策使得好多的家庭断子绝孙（虽然这样说好像不太好，但这是一个事实，在中国悠久的历史文化中，一个家庭是必须有一个男孩的，中国人称之为：香火。而说实话现在人们无所谓了，但在传统的意义上说，一旦生女孩其实就是断子绝孙）。

好多家庭真的是一个，可能是宝贝男孩，但很可以一场车祸或意外也会使这个家庭断子绝孙。（其实这一点说起来很残忍，也可能有些人会用这些话来当作骂人调侃，但事实上却是每天都在各个不同的地方上演着相同的惨剧，虽然我们这样说不太好，但尊重事实吧，这的确是事实，任何人都改变不了的。）我就有一个亲戚的亲戚，孩子都高三了，家庭中的独苗子，成绩都很好，突然来了一个白血病。咋办？（注：OVER 了）

再谈谈这独生子女的事吧，如果你仔细去算来，用不了多少代，中国就会没人了，我在《算法》版块中讲了《浅议递归与数学归纳法》中讲了几个故事大家如果看了应该知道，这 2 倍会使得数据迅速增加，而 1 / 2 肯定会使得总体数据迅速减少。一开始对人口的增长的确有效，因为青壮年实在太多了，即使计划生育，人口增长的速度还是高于死亡的速度。大家知道现在中国的人口老龄化现象严重吧，若干年后，中国的人口必然会迅速减少（什么意思想必大家都明白，人死了呗，说人不死，那倒真是骂人）。

你知道吗？那时候再叫一对夫妇多生几个孩子，那可就难了，除非国家的教育费用、医疗费用等若干费用全免，想必到这个论坛上来看帖子的人层次都很高，你也一定知道一个家庭培养成功一个孩子要多大的代价。

现在有好多上了年纪的人都说了（注：老人的话往往具有代表性，出自肺腑，真实性较高，呵呵）：现在的年青人啊，你叫她生二胎也不生了。

事实也真是如此，现在的年青夫妇即使生到女的也无所谓，时代不同了，男女都一样（怎么像计划生育的口号，呵呵）。

岔开去了，上面提出车祸的事，其实如果是我们学过计算机的人，解决起这种问题来其实显得格外地简单，只需要使用计算机运算的高速度，几乎可以杜绝任何车祸的发生。

一个汽车在驾驶员驾驶行为出现异常的情况时，自动接管驾驶就任务就一切OK了，其实用计算机去开车远比驾驶员去开车要安全得多，你听说过几驾客机从天上掉下来的事？有是有，毕竟拿车祸来比要少得多了，什么原因，就是因为飞机其实完全是自动导航。

其实牛刀也构思也设计过高速公路的自动导航系统，几乎可以替代现有的红绿灯与立交桥，但那只是设想啊。

如果一旦牛刀构思的这套高速路系统用起来，可能汽车之间就可以自由通信了，并且与红绿灯进行通信，那时候汽车可以提速了时速２５０公里/时，交叉路口如果遇到虚拟的红灯信号，只需要将车速降至２００公里，前面就可以让出足够安全的间距让给其他线２５０公里时速的汽车顺利通过了。牛刀称之为限速红灯（其实我们现有的红灯可以理解为0速红灯，也就是说速度降为零了，以便让出一段距离，其实如果一旦我们所有的汽车上用上计算机，这远远没有必要，难道你还不相信计算机的判断速度与判断力么？大家听说了么？现在的汽车发动机的喷油嘴喷油的时间是由计算机控制的，这样一来，同样的汽车要省好多油。为什么这种计算机的技术就没人考虑用来机动车辆的安全行驶上呢？

其实你如果来论坛并且能够看到牛刀这儿你就应该知道，这其实很简单：

```
if(前方有障碍物)
{
        停车;
}
```

不就行了吗？呵呵，事实就是如此，再来：

```
if(右方有障碍物)
{
        向左拐弯;
}
```

想发笑是吗？事实就是如此，很简单，如果汽车厂商做汽车时，能将方向盘，刹车，机动，加油，等东西做成一个 API 函数库，我想编程根本不是一件难事，甚至可以使得汽车完全实现无人驾驶。

题外话太多大家不会介意吧？牛刀感觉有点一发不可收拾了，大家知道现在培养一个驾驶员多长时间吧？20 天到三个月，知道以前培养一个驾驶员需要多长时间吗？至少三年。牛刀只能用一个字——可悲（买一送一，呵呵）。

好了，不聊太多了，能够让我们做程序的发挥的余地多着呢，我们还来看上面的程序吧：

```
void main()
{
        B b;
        C c;
        B *pb=&b;
}
```

这里我们定义了一个指针 B，指向 B，这一点对于我们来说一点都没问题，再做向个实在的程序吧：

```
...
class B:public A
{
public:
        void test()
        {
                printf("这是 B 的 test 函数\n");
        }
};
class C:public A
{
public:
        void test()
        {
                printf("这是 C 的 test 函数\n");
        }
};
void main()
{
        B b;
        C c;
        B *pb=&b;
        pb->test();
}
```

这个程序想必大家读起来不费劲吧？结果想必也不用说：

**引用:**

这是 B 的 test 函数

但牛刀突发奇想，能不能用这个指针去调用类 C 中的 test 函数呢？痴人说梦，呵呵，你一定会有这种反应，那牛刀不服，用强制类型转换呢？(C*)pb=c;呵呵，别痴心妄想，是两个不相干的东西，你怎么能够去强制呢？事实也真的如此。（注意啊，牛刀前面那个代码(C*)pb=c;这样的强制类型转换是错误的啊，我们不能强制左值。什么原因大家上机去报个错再仔细去体会吧。）

说实在的，不甘心，真的不甘心，同一个父亲怎么差别就那么大呢？对了，微软人员突发奇想，指向父亲的指针能不能指向儿子呢？

```
void main()
{
        B b;
        C c;
        A *pa=&c;
```

```
        pa->test();
}
```

哎，还真的可以，但一旦调用其中的 test 函数，父亲犯了难，向儿子请教吧，那不行，我是爸爸，我是老子，咋办，做父亲的不甘示弱，我也来定义一个 test 函数吧，我也有，看你咋办？

```
...
class A
{
public:
        void test()
        {
                printf("这是父亲 A 类的 test 函数  \n");
        }
};

...

void main()
{
        B b;
        C c;
        A *pa=&c;
        pa->test();
        pa=&b;
        pa->test();
}
```

做儿子的还真不买帐，你做父亲的不是有么？有你调用你自己的好喽，我的就是不给，于是，两次调用，就出现了如下的结果：

**引用:**

这是父亲 A 类的 test 函数
这是父亲 A 类的 test 函数

父亲毕竟是父亲，开始买糖（virtual）来哄儿子啦，儿子乖，儿子好，儿子……

```
class A
{
public:
        virtual void test()
        {
                printf("这是父亲 A 类的 test 函数  \n");
        }
};
```

这回儿子的确被哄住了，乖乖地给出了函数：

**引用:**

这是 C 的 test 函数
这是 B 的 test 函数
父亲这种哄孩子的伎俩我们就称之为“虚函数”。

父类如果想要调用子类的某个函数，则必须要使用虚函数，也就是说父子的虚函数被子类的函数给覆盖了，这样一来，与我们前面的子类覆盖父类的原则就有机地统一了起来。

大家思考过吗？父亲为什么要使用虚函数，原因就是我们需要一个能够指向任意一个子类的指针只有有了这样的指针，我们做程序才会得心应手。

下面与大家讲讲为什么要这样做吧，因为父类作为一个指针来说的话，其实指针指向的是父类的类，而子类是继承了父类的所有函数，也就是在父类的基础上再加一块而已，所以说，指向父类的指针，如果指向了子类，那么指针中前面的内容是一模一样，所以说，能够指过去，并且可以顺利地调用子类继承父类的所有的函数，当然也正因为父类中的相应函数都有，所以这个指针也就是父类的指针，调用的所有内容其实还是父类中的内容，那么指向子类有什么意思，不还是和指向父类一个样吗？
所以说，为了解决这个问题，给相应的函数加上了一个虚函数的标志：virtual，其实是标识符，就是告诉了指针，假如子类中有相同的函数，指针就指向子类的函数，这样一来，就解决了使用指针中的继承与覆盖的问题。

我们详细的例子下面再举，再讲一个事实，就是说，析构函数能不能也像其他函数那样，可以用指针指过去呢？感觉上显然不能，因为不同的类，析构函数肯定不会相同，于是，C++作了特殊的规定，析构函数也可以做成虚函数，并由父类指针调用，为了实现这个目的，一般在析构函数前也加上一个 virtual 使之变成虚类，一般我们析构函数都做成虚类，所以说，我们当用 VC6.0 中的类生成工具生成一个类时，析构函数都是虚函数。

C++规定，基类中如果定义一个虚函数，不管子类中是不是定义了虚函数（也就是前面加不加 virtual ），同名函数都是虚函数。但一般我们都加上虚函数标志，以便人们看起来知道这是一个虚函数。

大家说为什么要用虚函数呢？我们子类不就可以继承父类吗？定义一个类，就不会有了？哪里是那样，这样一定义好玩啊：

```
#include <stdio.h>
class A
{
public:
        virtual void Say()
        {
        }
```

```
};

class F:public A
{
public:
        void Say()
        {
                printf("我是爸爸我怕谁？\n");
        }
};

class M:public A
{
public:
        void Say()
        {
                printf("我是妈妈我怕谁？\n");
        }
};

class T:public A
{
public:
        void Say()
        {
                printf("我是第三者我怕谁？\n");
        }
};
//不再多举例了，三个够了，呵呵：

void main()
{
        F f;
        M m;
        T t;
        A *pa=&m;
        pa->Say();
        pa=&f;
        pa->Say();
        pa=&t;
        pa->Say();          //牛刀这里做的是不是有点恶心，呵呵
};
```

大家看，父类中的 Say 函数，就是一个空架子，而我们只需要定义一个子针，就可以指向任意一个派生类，这是多么美妙的事情啊，上面的结果想必大

家都知道了吧？

**引用:**

我是妈妈我怕谁？
我是爸爸我怕谁？
我是第三者我怕谁？

再与大家说说这种虚函数的好处，那就是间接使用，我们还是来举一个例子吧：（请注意我们以前的约定）

```
#include <stdio.h>
//注：由于以下程序被假设分成了多个文件，所以在适当的位置应当用包含将某一文件包含进来
//但由于我们是在一个文件中模拟这种结构，所以头包含我们用注释注起来，大家明白就行
//------------a.h-------------
class A
{
public:
        virtual void test();
};

//-------------a.DLL--------------
//#include "a.h"
void A::test()
{
        printf("A::test();\n");
}
//-----------B.h----------
//#include "a.h"
class B:public A
{
public:
        void test();
};
//----------b.dll---------
//#include "b.h"
void B::test()
{
        printf("B::test();\n");
}
//---------c.h------------
//#include "b.h"
class C
{
```

```
public:
        A* Getap();         //这是对外的接口，我们假设 C 函数就是接口函数，任何一个系统都要有对外接口
                                //的 API 函数的，否则这个系统就是一个封闭的系统，根本不可能更改
};
//------------test.dll--------
//#include "c.h"
void main()
{
        C c;
        A *ap=c.Getap();
        ap->test();
}
//请注意，假设我们上面的东西全部封装在一个 DLL 中，即做成了二进制文件
//要求，不改变 DLL 文件，改变输出结果

//以下是提供对外的接口类 C 类
//------------c.cpp----------
//#include "c.h"

A* C::Getap()
{
        A* ap=new A;
        return ap;
}
```

上面的程序结构比较复杂，好像程序感觉也比较大，但事实上我们以后要面对的类要比这个大得多，大家仔细去阅读，理解清楚之间的层次与关系。

大家看，我们想将主程序中的 test 函数改变，还有机会么？对于上面对外提供的接口，我们知道，只需要做如下的修改就一切 OK：

```
A* C::Getap()
{
        A* ap=new B;        //注：这里新建的 A 被牛刀改成了 B 了，就这么简单
        return ap;
}
```

于是乎，我们一运行程序。结果就变了，这也正是虚函数强势所在，可能通过一个很简单的外部接口，定义一个函数为虚函数，就可以随时将一个事情降级给儿子或者孙子去处理 MFC 中这种技术用了老多了，虽然可能用的并不是相同的结果，但技术角度来讲，就是将基类中需要降级处理的函数全部定义成虚函数，而其中的调用则使用指针调用，通过一定的技术手段让其他函数或子函数来改变我们的指针，大家想，上面的类 B 是不是被封装成了 DLL 了？我们

再编译时，这个 DLL 我们可以不用啊，我们再重新做一个，我们不就可以任意地改变我们程序的功能了？

这样一来，我们的程序中可能入口函数都找不到了，而我们做的事情就是定义子类、为子类定义所需要的函数，重定义父类的虚函数等等，就这样的事。

于是，为了方便，MFC 又使用了一种新技术——消息映射技术，使得我们的消息处理全部变成了函数处理，并且为此定义了一个强势工具——建立类向导...

按 查看 菜单，下面就有，当然我们也可以按 Ctrl+W 调出，这个向导，使得我们的类、对像、图标、对话框、菜单、以及各种其他的资源有机地结合了起来。

当然为了让 VC6.0 能够自动管理这些类，在程序中用了新的注释：

```
//{{
...
//}}
```

这种注释就是让编译器看的，我们使用类向导时使用的，并且类向导是自动管理的，所以我们自己做 MFC 程序时，我们的代码千万别放在这两个注释之间，因为如果一旦放上， 类向导在再次运行时，找不到相应的对象或资源，就无情地会将你的代码给删了，好多人会弄得莫名其妙。

所以我们说，类向导其实也有 BUG ，而且不小，大家使用中，突然发现一个类不见了，或者突然发现工作巨中找不到相应类的提示了，其实是一个挺麻烦的事，以后大家如果遇到这种情况，退出 VC6.0 以后，将相关文件下的 .clw 文件删了，再重新打开程序的话，重建一个，一般就可以有所改观，如果还不行的话，朋友们，认命吧，重做一个新的工程吧，或许那样会对你有所帮助。

事实其实就是如此，VC6.0 还并不完美，牛刀以前说过，程序是人做的，就不可能做到尽善尽美，肯定有 BUG，所以说，我们的电脑与网络是不安全的。

行了，以一个故事结束我们这一讲吧，事实上这是真事，很久很久以前，当人们还用 TC 系列的编译器，做完了一个软件，使用后，却经常受到黑客的攻击，也不知道是什么手段，好像系统的管理人员不管怎么设密码，都会被别人猜出来似的，这是怎么回事。

于是一帮程序员就回去到相关代码上去找错误，当然这种错误我们一定知道啦，例如：

```
if(a=5)
{
        ...
}
```

这种代码肯定是不论 a 是任何值条件都成立是吧？这种错误也可以找到，可结果却令人失望，密码处理部分没有任何问题，加个密吧：

```
b=(...+a...^a...|a......)
if(b==(.....))
```

```
{
        ....
}
```

上面的过程是牛刀瞎想的，但也可能是真的被牛刀瞎猫撞上死耗子了，呵呵，可还不顶用。

辛苦的程序员，一天天地坚持着，没有发现任何毛病（其实夸张了，做程序能没毛病吗，呵呵）。

怎么回事？你想不通吧，如果告诉你一个事实，会让你大吃一惊，程序员做的程序果真没有任何毛病，毛病出在编译器上。这又是怎么回事？原来做 TC 编译器的人啊，为了调试编译器方便，为其设计了一个密码，只要输出这一个密码，所有的 if 判断语句后的条件则无条件地设置为真。为什么会这样呢？因为做编译器人为了调试自己的程序用的，不过由于程序做得太大，后来发行时，忘了改了，于是乎，那一个密码就成了这种编译器编译出来的程序的万能密码，不管什么软件，只要输入这一个密码，什么密码都不好使，什么加密技术都不好使。大家还记得 API 中有钩子函数吗？其实也是微软内部工作人员调用系统用的，后来不知怎么泄了密了，使得这个函数一下子成为系统控制的最高指挥。不由得你不信，用钩子，没几行的代码就可以使得你的电脑完全死机（其实不是死机，而是装一个全局的鼠标钩子与键盘钩子，去拦截你所有键盘与鼠标的消息并且吃掉，于是，你整个屏幕上的任何软件都接收不到键盘与鼠标的消息了，所以你看到的现象就是鼠标失灵了，键盘失灵了，请问，你鼠标没法用了，键盘没法用了，那和死机有什么区别？也就是说，我们电脑由于对这些消息的拦截而果真进入了“死循环”。还有更为恶心的人，在别人的电脑上下完这种钩子后，去修改注册表，将这个软件在电脑启动后自动运行，那样，你的电脑就彻底完蛋了。牛刀还是那句话，学了技术不要用去干坏事。本以为类是花哨不实用的东西，特别是抽象类。现在看来作用还是很大的。呵呵 MFC 可全靠着她呢，其实 MFC 中好多是通过一个结构体在子函数中重载一个虚函数处理过去的，

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        if( !CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs
        cs.cx=200;
        cs.cy=200;
        return TRUE;
}
```

你瞧这个多方便，我们想改变窗体的大小，只要在相应的函数中改变结构体的相应属性，再一编译就一切 OK 了。

**类的实践〇六三：特殊的指针**

类中有一个特殊的指针，不需要定义也不需要说明，甚至有没有都无所谓，但这个指针的确存在，这个指针就是——　this 指针。

说实话，这个指针什么意思呢？就是指的类本身，很有用的一个指针，就是指向类自己，请大家注意这是一个指针，而不要将其认为是一个关键字，但 VC6.0 的显示效果是一个关键字的效果，我们可以按固定指针的方式来使用这个指针，这个指针不能移动，但的确表明了一个身份，并且还很有用，看程序：

```
#include <stdio.h>
class A
{
private:          //其实这一句没有必要，但我们习惯上还是加上。
        int x,y;          //大家可以在右方的工作区中打开 A 这一个类，会看见这两个变量前面图标上加了一把锁，表明这两个变量是私有变量（被锁住了，呵呵）
public:
        A(int a,int b);
        void print()
        {
                printf("x=%d,y=%d\n",x,y);
        }
};

void main()
{
        A a(12,34);
        a.print();
}

A::A(int a,int b)
{
        x=a;
        y=b;
}
```

这个程序想必大伙都写得出来，我们单看构造函数，x=a;y=b;　其实没什么吧？但如果我们如下定义呢？

```
#include <stdio.h>
class A
{
private:
        int x,y;
```

```
public:
        A(int x,int y);
        void print()
        {
                printf("x=%d,y=%d\n",x,y);
        }
};

void main()
{
        A a(12,34);
        a.print();
}

A::A(int x,int y)
{
        //???
}
```

重名了吗？没重名，作用域不同，私有成员中的 x,y 属于类的，或者说在函数 A(int x,int y);中来说属于外部变量，而参数表中的 x,y 属于内部变量，（哎～～～，函数就是有点吃里扒外，没办法）所以说，这并没有重名，那么我们显然可以用作用域定义符来做到这种参数赋给类变量的方法来做程序：

```
A::A(int x,int y)
{
        A::x=x;
        A::y=y;
}
```

这一点问题都没有，事实上作用域定义可以很好地解决这种外部变量重名问题，可一旦我们用了 this 指针就显得更方便，或者说方便多了：

```
A::A(int x,int y)
{
        this->x=x;
        this->y=y;
}
```

感觉好像比上面的要复杂了是吧？呵呵，其实不复杂，为什么大家看牛刀这个例子觉得复杂了呢？因为：

一、牛刀定义的这个类的定义符太少了，一个字母 A ，这里仅仅是为了教学的目的定义的，而现实中你不可能使用这么短的名称。
二、当你定义好多类时，如果一下子做多个类的程序，你很可能会搞糊涂的，而 this 指针永远不会糊涂。
三、this 指针几乎可以作为一个很好的提示来用，这一点我们在 MFC 编程中如果用起来就很方便了，只需要一打 this->马上后面就会列出本例中所有的函数与父类中所有的函数，往往我们根本不需要去有意地记住或去查找某些函数

（这一点要在以后编程过程中才能真实地体会出来。
四、由于 this 指针根本不需要定义，所以说我们根本不用担心有没有这个指针的问题，并且任尔千变万化，指针形成特固定，就是 this->。用多了，习惯了，你其实随手就会用上。

不说过多的理由了再多的理由大家自己在实践中去慢慢地体会吧，大家在 MFC 中其实根本用不着去记忆什么函数，想调用什么，随时随地调用一个 this 指针就好像有一个说明书了，并且你打了前几个字母，函数自动上屏，那种感觉不必说，舒服。

说实话，如果谈 this 指针，说到这儿其实也就结束了，好多的东西真的需要自己去细细体会的，没有编程经验，就去记忆书本上的概念是没有意义的。

```
#include <stdio.h>
class A
{
private:
        int x,y;
public:
        A(int x,int y)
        {
                this->x=x;
                this->y=y;
        }
        void print()
        {
                printf("x=%d\ny=%d\n",x,y);
        }
};

void main()
{
        A a(2,5);
        a.print();
}
```

其实上面的函数我们就像上面那样定义一下就行了，用不着拿到类外面，所以像上面那样的忒也小的函数有时候我们前面也加上一个关键字 inline ，也就是将函数表示为内联函数，那么什么叫内联函数呢？也就是这个函数其实在程序运行时是不执行的，而是如果遇到这样的调用，在编译器编译时，就将这个函数直接加到代码后面去了。

有些书上会有一些规则，说什么情况下要加，什么情况下不要加，其实没必要知道，内联其实是一种编译器的行为，由编译器把握，如果条件不允许，就是你加内联关键字也没有用的，其实内联就是要提高程序运行速度而已，大家如果做实时控制系统，就必须要对这些知识有详细地了解，并且要符合内联规则。

再说了，如果果真做一个实时系统，谁还用 VC6.0 啊，高手一般用汇编，那东西快啊，你知道你做一个很简单的程序：

```
y=a+b;
```

会被汇编成什么代码吗？而如果用汇编：

```
MOV AH,b
ADD AH,a
MOV y,AH
```

事实上任何一个编译器都不可能编译出这么高效率的代码出来，何况一旦我们 y 只是中间变量时，在汇编中几乎就直接用 AH 作为中间变量的寄存就一切 OK 了，根本不需要 y 这个变量。

做任何语言程序的朋友都不要与汇编程序员去比效率，你肯定不是对方的对手。

所以说，一般我们短小的程序如果你觉得有必要，前面就加上 inline 好了，至于能不能达到目标，那是编译器的事，我们用不着管。

我们再加一行代码啊：

```
void main()
{
        A a(2,5);
        a.print();
        A b(a);
        b.print();
}
```

大家想看看程序运行的结果吗？

**引用:**

```
x=2
y=5
x=2
y=5
```

怪事怪事，出怪事了，为了让大家看明白点，牛刀再将程序改一改：

```
void main()
{
        printf("定义一个A的类 A a(2,5);\n");
        A a(2,5);
        a.print();
        printf("定义一个A的类 A b(a);\n");
        A b(a);
        b.print();
}
```

结果如下：

**引用:**

定义一个A的类 A a(2,5);
x=2
y=5
定义一个A的类 A b(a);
x=2
y=5

怪事啊怪事，怎么C++中尽出怪事？其实啊，不奇怪，因为我们类中还有一个默认的函数，我们没有讲，下面我们将其补上：

```
class A
{
private:
        int x,y;
public:
        A(int x,int y)
        {
                this->x=x;
                this->y=y;
        }
        inline void print()
        {
                printf("x=%d\ny=%d\n",x,y);
        }
        A(A &a)
        {
                x=a.x;
                y=a.y;
        }
};
```

大家看到这种函数了吗： A(A &a); 这种函数，我们称之为拷贝构造函数，这种技术在程序中用得相当普遍，甚至我们不定义这个函数，系统也自动有了这个函数，系统为我们自动定义的就是利用参数中的类来为自己的类成员所有变量赋值，所以如果我们不需要改变一些东西，我们可以不重新定义这个函数，当然也可以去实现我们一些“恶心”的目的，看程序：

```
...
        A(A &a)
        {
                x=a.x+3;
                y=a.y+5;
        }
...
```

这下大家再运行程序，再看，什么结果了？

**引用:**

定义一个A的类 A a(2,5);
x=2
y=5
定义一个A的类 A b(a);
x=5
y=10
事实就是如此，记得噢，上面那个 & 大家不要当成是一个取地址符哟，如果那么认为就错了，那是一个拷贝对像，或者说是一个变量有两个名子，也就是我们前面所说的引用既然说到这是引用，牛刀再来做些“恶心”的事，来验证一下：

```
...
        A(A &a)
        {
                x=a.x+3;
                y=a.y+5;

                a.x++;          //请注意这两行代码
                a.y++;
        }
};

void main()
{
        printf("定义一个A的类 A a(2,5);\n");
        A a(2,5);
        a.print();
        printf("定义一个A的类 A b(a);\n");
        A b(a);
        b.print();
        printf("我们再调用一下 a.print();看看 a 的结果是不是被我们改了？\n");
        a.print();
}
```

大家看结果，果然不错这是一个引用，看结果：

**引用:**

定义一个A的类 A a(2,5);
x=2
y=5
定义一个A的类 A b(a);
x=5

y=10
我们再调用一下 a.print();看看 a 的结果是不是被我们改了？

**引用:**

x=3
y=6

我们必须这样做一下才会让大家知道，什么叫构造函数，其实就是引用另一个函数来为自己赋值，这种技术在 C++ 中到处可以得到体现，其实我们可以不用 “=”为一个变量赋值，看程序：

```
void main()
{
        int i(5);
        printf("i=%d\n",i);
}
```

程序是不是看起来有些惨不忍睹，其实用这种办法来做，那是一个字，方便啊：（怎么买一送二了，呜，牛刀要亏本了）。

```
        A(int a,int b):x(a),y(b)
        {
        }
```

这个程序看起来晕吧，呵呵，我们程序中什么都没有，就靠这种拷贝函数就给变量给赋了值了。遇到这种这么舒服的事，还有多少人乐意去写那函数体呢？根本没必要。

既然如此，我们现在就定义一个拷贝函数，来看看函数的调用情况，（大家注意，随着我们编程水平的逐步提高，我们要学会去读别人的代码，看懂别人在干什么：

```
#include <stdio.h>
int i=1;
class A
{
private:
        int x,y;
public:
        A(int a,int b):x(a),y(b)
        {
        }
        inline void print()
        {
                printf("x=%d\ny=%d\n",x,y);
        }
        A(A &a)
        {
                x=a.x;
```

```
                y=a.y;
                printf("本拷贝函数被调用了 %d  次。\n",i++);
        }
};
void fun1(A &a)
{
        printf("进入了函数 void fun1(A &a)\n");
        a.print();
        printf("退出了函数 void fun1(A &a)\n");
}

A fun2()
{
        A a(6,7);
        return a;
}


void main()
{
        A a(2,5);
        a.print();
        A b(a);
        fun1(b);
        a=fun2();
        a.print();
        fun1(fun2());         //这种情况看过吗？呵呵
}
```

我们看结果：（注：仔细地体会这种拷贝函数是在哪些地方被调用了，是怎么被调用的。)

**引用:**

```
x=2
y=5
本拷贝函数被调用了 1  次。
进入了函数 void fun1(A &a)
x=2
y=5
退出了函数 void fun1(A &a)
本拷贝函数被调用了 2  次。
x=6
y=7
本拷贝函数被调用了 3  次。
```

进入了函数 void fun1(A &a)
x=6
y=7
退出了函数 void fun1(A &a)

大家千万别一看眼一刷就过去，说实话，牛刀做的这些程序虽然没用，但很典型，说不准是有什么机会就可能会用得上。

说了这么多，大家是不是累了？聊聊天吧，以上的知识点呢，说实话，挺碎的，但挺有用，恨不得不讲这些，感觉有些鸡肋，但如果我们真得能够感觉得到她的存在，其实用起来还真是挺方便的。

你就像那种拷贝构造吧，大家想过没有，看程序：

```
struct Rec
{
        int x,y,w,h
}

void main()
{
        Rec rect;
        rect.x=20;
        rect.y=20;
        rect.w=50;
        rect.h=50;
}
```

而如果我们还需要定义一个呢？

```
void main()
{
        Rec rect;
        rect.x=20;
        rect.y=20;
        rect.w=50;
        rect.h=50;
        Rec rec;
        rec.x=rect.x;
        rec.y=rect.y;
        rec.w=rect.w;
        rec.h=rect.h;
}
```

多少行代码？是不是做这些程序很累呢？但如果我们将之改成类，使用拷贝构造再来看一下：

```
class Rec
{
private:
```

```
        int x,y,w,h
public:
        Rec(int a,int b,int c,int d):x(a),y(b),w(c),h(d)        //是不是这样感觉挺舒服，挺方便？:)
        {
        }
}

void main()
{
        Rec rect(20,20,50,50);
        Rec rec(rect);
}
```

这种程序的精简大家还受得了吗？是不是一下子感觉我们编程要简洁多了，我们的程序清爽了？

我曾在 VB 那个版块看到有些人认为 VC 没必要，VB 多简单啊，是 VB 简单吗？还是 VC 简洁？

甚至一般我们写像上面那样的类的话，后面的括号有时也放在一行，那种感觉就像这种不是函数一样，看：

```
class Rec
{
private:
        int x,y,w,h
public:
        Rec(int a,int b,int c,int d):x(a),y(b),w(c),h(d){}
}
```

天啊，这还是类吗？函数的实现哪儿去了，呵呵，是不是看见一行没有分号结尾很不习惯？这哪是语句啊，这是整个的函数啊，呵呵。

其实事实就是这样，最后我们再讲一个特殊的类成员变量 静态变量（static），类中也可以定义一个静态变量，什么是静态变量呢？其实大家在函数部分就知道，其实这种变量就是整个程序中，只产生一次，其生命周期与整个程序的结束周期相同。那么这么一来，我们这种静态的变量就不可能属于哪一个对象，只能属于类，也就是说，这个类在定义时，就存在这个变量了，而与其他的类的对象没有关系，为了说明其不同，我们还是来看程序：

```
#include <stdio.h>
class A
{
private:
        int x,y;
        static int num;
public:
```

```
        A(int a,int b):x(a),y(b){}
};

int A::num=0;          //请大家注意，num 属于类，但只属于类，并且类定义了，这个变量就存在了
                       //也就是说，这个变量与类其实没有关系，所以说，我们只能在类外面进行初始化，
                       //而且必须进行初始化，也就是说，这相当于是个外部变量，但又不是，属于类
                       //所以说，我们调用这个数据时，只能在类中进行调用，而我们这仅仅是定义

void main()
{
        A a(2,5);
}
```

上面的程序我们只告诉大家应该怎么使用这个变量，为了说明其静态的性质，我们来为其做一些程序，请注意，num 这个变量绝对是属于类，并且是私有的，所以在程序中，根本不允许类外进行访问，那么我们怎么看这个变量呢？我们将其功能作为计数功能吧，我们用她来看我们对象调用了这个类的 print 函数多少次了。看程序：

```
#include <stdio.h>
class A
{
private:
        int x,y;
        static int num;
public:
        A(int a,int b):x(a),y(b)
        {         //为了我们明白各种调用过程，我们还是在这里做些提示信息吧：
                printf("这是类中函数  A::A(int x,int y)  的一次调用，这次传进来的参数为：%d,%d\n",x,y);
        }
        void print()
        {
                printf("这个类的 print  函数被各个不同的对象调用了%d次。\n",num++);
                printf("这次调用此函数时，相应的值 x=%d,y=%d\n",x,y);
        }
};
int A::num=1;
void main()
{
```

```
        A a(2,5);
        a.print();
        A b(3,4);
        b.print();
        A c(6,7);
        c.print();
        A d(9,0);
        d.print();
}
```

我们还是先来看结果（大家注意分析）：

**引用:**

这是类中函数　A::A(int x,int y)　的一次调用，这次传进来的参数为：2,5
这个类的 print　函数被各个不同的对象调用了 1 次。
这次调用此函数时，相应的值 x=2,y=5
这是类中函数　A::A(int x,int y)　的一次调用，这次传进来的参数为：3,4
这个类的 print　函数被各个不同的对象调用了 2 次。
这次调用此函数时，相应的值 x=3,y=4
这是类中函数　A::A(int x,int y)　的一次调用，这次传进来的参数为：6,7
这个类的 print　函数被各个不同的对象调用了 3 次。
这次调用此函数时，相应的值 x=6,y=7
这是类中函数　A::A(int x,int y)　的一次调用，这次传进来的参数为：9,0
这个类的 print　函数被各个不同的对象调用了 4 次。
这次调用此函数时，相应的值 x=9,y=0

大家看到了吧，不论定义了多少个对象，但这个静态成员还就是不理不睬，我行我素。

事实上，我们还可以定义一个静态函数，想必大家可能都会定义了吧，呵呵，一样的，不过静态函数也必须在类外定义，其实就相当于一个全局函数，但不同的就是其他的地方不能调用这个函数而已，所谓静态，其实也就是这么回事。

总而言之，大家明白一点，这个关键字（static），不论用在哪儿，用到哪儿哪儿的那样东西在整个程序中就只定义一次，其实就是这样，不要看书上又要考虑什么作用域啊，又要考虑什么定义域啊，其实完全没这种必要，我们要知道一点就是这种静态的东西在哪儿定义的，其作用域就是那个范围，但不同的是，整个程序中，只定义一次，就是这个道理。

显然大家明白，一旦我们上面的对象被析构了，那么类中的变量 num　还存在不存在呢？显然是存在，但我们调用不到了，如果想要用，就再产生一个对象才能用。

这样一来，真好啊，静态变量就像同一个类之间的通讯员，可以使得我们相同对象之间实现通信，通信大家不会不知道吧，呵呵，打电话，上网，等等等等……

如果我们这个类一旦被继承，那么派生出的功能就非常得强大了，有时候强大的你不可想像，但一旦这个技术与继承或者虚函数等技术一旦结合起来，其关

系的复杂程度想必大家一定会吃惊的，其实为了实现类中成员的保护，继承被分成了三种：公有继承、私有继承与保护继承。

以前学继承时牛刀没有说，因为那时候我们的确抽不出空来进行这么详细的分类啊，那么什么叫公有属性呢？公有属性其实就是对外公开的。大家知道收音机吧，内部电路多复杂，那是私有的，你不能去动，但调台旋钮是公有的，你可以改变，其实意思也是一样。那么什么叫保护类型呢？那就是对子类来说的，保护类型对外和私有类型一样，外部函数不能调用，但对儿子就不同了，对儿子来说是公有的，儿子可以随便调用。那么这么一样我们就知道，公有是公开的对外对内全一样，保护是对内的，儿子家长都一样，而私有则是父亲的一点小秘密，儿子也不行，儿子也拿不到。于是，大家以后定义一个类就要注意我们是不是愿意继承者能得到这个类中哪些函数的调用，我们需要设置成保护的，而不让调用的设置成私有的，其他函数全部设计成公有的。　好了，还有一个问题就是，为什么继承还分公有继承、保护继承与私有继承呢？有什么联系与区别么？那是当儿子的孝顺，有时候老子给什么就要什么，不给就不要，那叫公有继承，当然有时候儿子一想，这老子太大方了，肥水不流外人田，我凭什么让你外人占有啊，于是儿子耍了点小心眼，就将其改变为保护继承，也就是父亲只要公有的东西，也全都不对外了，那么孙子呢？孙子还可以，哪叫那是爷爷呢？你说是吧？
最自私的就是第三种了，也不知是儿子孝顺还是什么原因，一概变成私有继承，也就是不管你保护不保护，公有不公有，儿子也不要了，外人也不想给，于是就来了个私有继承.至于私有继承有什么用，牛刀也不明白，既然老子的东西儿子都不要了，还要继承干嘛，干脆独立门户吧。这一点牛刀讲的也不知对不对，还请高手指点。也就是说，果真有没有用私有继承来做程序的，能做哪些程序，私有继承与不继承与区别吗？父类指针能不指向私有继承的儿子的虚函数等等，牛刀还真是不知道，也懒得上机实践了，真累!

说实话，这一讲讲的知识点比较散，因为牛刀看看这些知识点都不足以单独列章，对于大伙来说，举太多太大的例子意义又不大，所以牛刀将其整合成一单独的一讲，希望这一讲大伙能够展开讨论，毕竟有些东西牛刀也不能说得太明白，只不过是实践过了拿过来与大伙分享一下，希望大家以后程序中能够用到这一讲的知识。如果说类的知识，说实话还真的就不太多了，零碎的在以后有必要时再拾起来与大家慢慢讲，大家学完知识要多做程序，多在程序中去理解，去运用，说实话，知识是死的，但编译器是活的，她能够有效地编译我们的知识并给我们打分，牛刀就这种观念，学编程不需要老师的打分，不需要考试，每次我们按编译时，都是对我们所掌握知识的一次检验，考试你很容易通过，你能够通过编译器这一关吗？只有在编译器的这张试卷中，我们才能拿满分。对了，还有一些零碎的知识，像什么类的模块啊，函数模板啊，牛刀就不打算讲了，那东西说实话，一看就会，一听就懂，一扔就忘，说实话，大伙在自己的几个程序中真正用过模板啊，说实话你真可以用也想不起来，如果要用到，去翻翻书或者到电脑上去查一下就一切 OK 了。

## 类的实践〇六四：特别的函数

这一讲是牛刀基础教程的最后一讲了，类中有许多概念，什么抽象基类啊，什么友元函数啊，牛刀都没有作详细地讲解，说实话，如果大家能够看到这，样必那些东西只是心有灵犀的知识点了，什么叫友元，朋友关系呗，比父子关系要明朗多了，朋友之间有些东西就是对朋友公共呗，当然，我们这里的朋友一定是知心朋友，酒肉朋友免谈啊。

其实说实话，微软搞这些东西其实有时候也是被“逼”出来的，向往过去那种 BASIC 啊，一共十七条语句，怎么来怎么去都是那十七条翻来覆去就一切 OK 了。

大家知道在 BASIC 中（注：不是 VB）怎么放音乐吗？

```
PLAY("CDEFGAB")
```

舒服吧？注意哟，BASIC 中没有封装哟，就是这样的，简单，怎么说都是简单，甚至 BASIC 中没有函数的概念，一到编程环境中直接输入程序，当然可以定义类似于 C 语言中的函数的东西，在 BASIC 中被叫做：子程序，其实那时子程序的目的就是将重复的东西做成子程序，以减少代码量，仅此而已，哪像我们 C 语言中，离开了函数就不行啊，呵呵。

后来学 PASCAL，说实话，那时候牛刀还不知道电脑是啥玩艺呢，不过上面程序倒被牛刀一个一个地搬到了“小霸王电脑学习机”上，呵呵。

说实话，后来由于学过 BASIC 从而就学 VB 了，牛刀学 C 语言很枯燥，根本没有环境的，电脑买回来也不知道怎么向系统上装 C 语言系统。就不要说什么 C++，VC 了，再说一句实话，C++那书买回来，牛刀就好像读天书一样，为什么牛刀要写这本实践教材，其实也就是这个目的，帮助大家少走弯路而已，其实说实话，谁不会买一本书回去读？谁稀罕牛刀？所以说，也正因为如此，牛刀这一讲就准备结束这本教材的基础部分，因为如果再讲那些细枝末节的东西可能就会惹得大家嫌烦了，真的，不说牛刀也知道，牛刀贴子想来是没人问的，呵呵。牛刀这就剃头的挑子——一头热。（人家没说出来罢了）

好了，这一讲废话有点多，这一讲讲什么呢？一种特殊的函数：运算符函数。

有些书上叫做运算符重载，为什么叫运算符重载呢？因为这种运算符，系统就有，就默认，你再做一个，不就重载了吗？其实可能也真是如此，我们做一个程序来看看：

```
#include <stdio.h>
class A
{
private:
        int x,y,z;
```

```
public:
        A(int a,int b,int c):x(a),y(b),z(c)
        {
        }
        void print()
        {
                printf("x=%d,y=%d,z=%d\n",x,y,z);
        }
};

void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A b=a;
        printf("A      b=a:\t");
        b.print();
}
```

大家感觉奇怪吗？怎么能用 b=a; 这样的方式来赋值呢？这可是一个类啊，是喽，可以，在类中就可以（注：在结构中也可以，不信将上面的 class 给改了试试，但一定要注意，是 C++的编译环境哟，并且建议最好后缀名用 .cpp 后缀名用 .c 可不可以，牛刀不知道，大家去做试验。

事情一下子变得简单了，简单到什么程度了呢？我们似乎只需要定义一个类，类之间就可以用一些运算符了，是这样吗？我们试试：

```
void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A b=a;
        printf("A      b=a:\t");
        b.print();
        A c=a+b;
        //...
}
```

大家看到 c=a+b; 了吗？程序牛刀就不往下写了，一运行就报错，为什么呢？因为对于运算符“+”，系统没有默认定义，所以我们想要使用这个运算符，就必须使用一种特殊的运算符函数，有些书上叫运算符重载，大家注意了，我们并没有重载啊，系统本来就没有，我们定义了，这叫什么重载？说得过去吗？

当然喽，由于我们根本不能真的将运算符作为一个函数名，所以 C++中定义了一个特殊的关键字 operator 看程序，我们对等号进行重载：

```
#include <stdio.h>
class A
{
private:
        int x,y,z;
public:
        A(int a,int b,int c):x(a),y(b),z(c)
        {
        }
        void print()
        {
                printf("x=%d,y=%d,z=%d\n",x,y,z);
        }
        A operator+(A &a)
        {
                int i=x+a.x;
                int j=y+a.y;
                int k=z+a.z;
                return A(i,j,k);
        }
};

void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A b=a;
        printf("A      b=a:\t");
        b.print();
        A c=a+b;
        printf("A    c=a+b:\t");
        c.print();
}
```

运行结果：

**引用:**

A a(2,3,4):  x=2,y=3,z=4
A  b=a:  x=2,y=3,z=4
A  c=a+b:  x=4,y=6,z=8

结果很理想，似乎我们的目的达到了，于是乎：

A operator+(A &a);

| *---|---*(....);
| 大家看这
| 一部分，
| 是不是很想一个函数名啊？
|
而这一部分很想一个返回值

括号里很像一个参数列表？对，其实就是一个函数，标准的函数，只不过函数名不是我们自己定义的，好是系统就这样默认的，我们想做这些函数或者重载这些函数，就必须要这样做。

可以说，什么叫运算符的重载（既然好多书上都这样说了，我们也附和着说吧，但其实不是这么回事，大家注意了）：也就是一个特殊名称的函数，而这个名称是固定的，必须的。

做个恶心的事，大家看，我们将对加号“+”的重载函数改一改：

```
A operator+(A &a)
{
        int i=x*a.x;
        int j=y*a.y;
        int k=z*a.z;
        return A(i,j,k);
}
```

我们将其改成了乘号，会出现什么结果呢？

**引用:**

A a(2,3,4):  x=2,y=3,z=4
A  b=a:  x=2,y=3,z=4
A  c=a+b:  x=4,y=9,z=16

事实真如我们想像的，每一个数变成了上面两数的乘积了，可以我们怎么也理解不了，我们加了两个数字，怎么结果却都乘了起来呢？所以说，我们做一个类，有一个原则就是，不要更改我们原有的定义习惯，一旦你改了，那么程序你自己明白什么意思，那么看你程序的人就不知所以然了。

所以说，我们当重载一个函数时，千万注意其功用，不要将一个重载函数的功能瞎定义，否则会令使用你的类的人哭笑不得，真的，别做这些捉弄人的事。

在我们的类中，所有的符号都可以重载，但不允许我们自己定义特殊的符号举个例子，大家不要弄个这样的函数出来 A operator\(A &a); ，编译器不认的，我们重载的函数必须是系统有的符号。从这一点上来说，又极大地限制了我们做程序的构思。

不过说实话，够了，绝对是够了。只要大家合理使用，系统符号是不会不够

我们用的。

大家还记得牛刀以前说过吗？字符串的连接问题：

```
Cstring str="牛刀教我 C++";
Cstring st="我学符号重载";
```

怎么来连接呢：

```
Cstring s=str+s;
```

就这么简单，为什么？因为在 Cstring 中，对等号“=”与加号“+”都重载过了，重载过了就可以用。

最典型的是对“<<”，与“>>”这两个符号，那才真叫：重载，如果大家使用 iostream.h 库的话，可以使用：

```
cin>>...
```

或

```
cout<<...
```

说实话，那才真叫重载呢？什么意思，后面不管什么类型，都有不同的函数去重载。所以我们用起来特别得方便并且可以连着用，这一点性质被 C++的设计者们称之为“流”，或者叫“流对象”，其实 cin、cout 就是一个对象，这两个对象对 cin 、 cout 进行了疯狂得重载，几乎所有的形式都被包进去了，形成了一个强大的库，C++这种技术用过了以后，以 VC6.0 为基础的功能库有好多好多，甚至我们一些常用算法、链表、树、队列、栈、串、图 以及排序、排序树、字符串处理、常用图像处理、三维技术、OpenGL，等等等等，一系列的大型的数据结构都被做成了一个一个的巨大的库，你学去吧，累死你，呵呵，学不全的，那是一定的，牛刀看了一会 OpenGL，但说实话，指望牛刀用 OpenGL 做出什么像样的程序来，那真是做梦，呵呵。

一些大型的库，几乎每一个库都能写出一本书来，当然我们自己也可以利用这些技术做出一些相应的库来，做成 DLL 文件，再做一个头文件，我们这些库就可以发布了，有了相应的功能，我们称这些东西为组件，（当然你的库必须有详细的说明书了，不然别人知道怎么用啊，呵呵），传说以后我们做程序就像搭积木那样简单，只要我们将一些东西拿过来，顺手搭搭而已。

哪能那样啊，暂时可能只是一个梦吧，我在网上看过有些公司出《易语言》看了一看，说实话，如果学过 C++，看那东西真是简单，几乎一个种头都不要就可以做出很像样的程序来，呵呵，还玩过《燃烧积木》，大家玩过吗？都是一大堆，一大堆的控件，美着你去吧，真的鼠标拖拖就成了。

难道真的鼠标拖拖就好吗？不，还是要做程序，只不过那种程序啊，说实话，没学过程序的越看越爱，学过程序的你越看越无聊。

不过说真的，这种组件技术其实真的很有用，有专门做组件的公司的，举个例子吧，如果我们想做个软件，其中需要用到一个地图，并且需要返回一些地图信息，那么你是自己从 API 起去做一个函数库呢？还是乐意去直接构买一个地

图的组件来直接向我们代码中一插呢？

其实说起来你可能也不信，C++中，默认的这些组件有好多好多，有空你可以去玩玩，例如我们要做一个快捷菜单吧，当然我们编程可以做，但你一插入一个组件，立刻这个菜单就可以在你的程序中使用了，你的任务就是将相应的菜单添加好，并且相应相应的函数就一切 OK 了，想要为我们的软件加一个封面吗？插入一个组件，将时间一设，封面图像一变，就一切 OK 了。

当然，那得需要我们的编程技术达到熟练级才能使用这些技术以便使得我们的软件更好看，更美观。

行了，我们还来说说上面的程序吧：

c=a+b，其实我们可以这样理解： c=a.operator+(b);

是不是就是调用了一个函数了？其实在 C++中好多的东西我们都可以这样类比一下就知道是怎么回事了。

那么这是一种双目运算符，那么对于单目的呢？例如对于负号“-”，如：c=-a ，怎么办？其实就相当于： c=a.operator-();就是这样，我们做个函数给大家试试：

```
...
        A operator-()
        {
                int i=-x;
                int j=-y;
                int k=x+y+z;          //这儿牛刀来点恶心的，这种现象马上就要体现出来，呵呵
                return A(i,j,k);
        }
};

void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A c=-a;
        printf("A    c=-a:\t");
        c.print();
}
```

结果：

**引用:**

A a(2,3,4):  x=2,y=3,z=4
A  c=-a:  x=-2,y=-3,z=9
大家看，是不是我们对这些运算符重载中想怎么干就怎么干？事实就是如此。

```
...
        void operator=(A &a)
        {
                x=a.x+5;          //注意牛刀又要干不法的事情了。
                y=a.y+4;
                z=a.z+3;
        };
};

void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A c(5,6,7);
        c=a;
        printf("A    c=a:\t");
        c.print();
}
```

大家知道结果会是什么吗？

**引用:**

A a(2,3,4):  x=2,y=3,z=4
A  c=a:  x=7,y=7,z=7
但怪的是，如果牛刀将程序变一下：

```
void main()
{
        A a(2,3,4);
        printf("A a(2,3,4):\t");
        a.print();
        A c=a;
        printf("A    c=a:\t");
        c.print();
}
```

结果却变成了：

**引用:**

A a(2,3,4):　　x=2,y=3,z=4
A　　c=a:　　x=2,y=3,z=4
　　显然在定义时如果调用等于函数，则使用了类默认的函数，为什么会这样？牛刀也说不清楚，可能是编译器的 BUG 吧。大家试看看，你的编译器上是不是这个情况，是否你能解释得通？

　　不管怎么样说来，同一程序出了两种不同的结果这一点还是会让我们以后的编程陷入一种难以想像的地步的，为什么，因为我们的目的达不到，你做了一个程序，结果目的达不到不是失败了么？

　　如果你不能给出合适的解释那么牛刀就又要来定义这种实践原则了：
牛刀实践原则：
　　如果在程序中定义了自己的等号运算符重载函数，则在定义此类时不能同时为类赋值，除非你的等号重载与系统默认定义相同（当然，如果相同又可以不用定义了是不，呵呵）。

　　其实有时候实践原则比书上概念还要有用啊，毕竟我们是在一个系统中来学习的，而不能光学理论。理论上　A c=a;就应该调用我们做的重载函数为什么没有调用呢？这的确是个问题。呵呵，牛刀又想再来干点事了：

```
void main()
{
        A a(2,3,4),b(4,5,6),c(7,8,9);
        c=b=a;
        a.print();
        b.print();
        c.print();
}
```

　　大家不上机估计会出现什么结果呢？

　　很可惜牛刀的电脑报错了，为什么会出现这种情况呢？因为牛刀定义的等于函数返回的是　void　类型，没返回值啊，所以为了让上面的程序也能够执行，于是，我们一般在定义等号函数时，使用下面的方式定义：

```
        A & operator=(A &a)
        {
                x=a.x+5;          //注意牛刀又要干不法的事情了。
                y=a.y+4;
                z=a.z+3;
                return *this;
        };
```

　　大家明白什么意思吧？就是说，我们改变自身的同时，又将我们本身的值送了出去，大家还记得 this　指针吧，这是一个指针，所以要返回一个值，必须使用取值符号“*”，其实就是这样，而返回的类型是一个引用类型，表明返回的值

与我们的是不同名而已。

也就是说什么意思呢？意思就是说： b=a; 这个运算是有值的，值就是 b ，而不是 a 或者其他，也就是说，当 b 取到值后又将自己送了出去，就这个意思。

好多好多的重载我们都可以这样定义，这样一定义有个什么好处呢？大家看：

```
cout<<"a="<<a<<endl<<"b="<<b<<endl;
```

如果你真正拿过一本 C++的书读过，那么你对这一段代码一定很熟悉，可为什么会连续使用呢？就是用的这种重载的技术。

再来一段代码：

```
...
        A & operator++()
        {
                x--;        //牛刀又要恶心了，呵呵
                y--;
                z--;
                return *this;
        }
};

void main()
{
        A a(2,3,4);
        a.print();
        ++a;
        a.print();

}
```

牛刀的恶心很快就会在程序中体现出来的，呵呵：

**引用:**

```
x=2,y=3,z=4
x=1,y=2,z=3
```

大家看，感觉比较好是不？可你不能使用 a++; 不信你用，你一用就报错，呵呵。

为了解决这个问题，C++特别规则如果用后置的单目运算符，则在括号中使用参数 int ，在这里这个参数大家不要理解成整型的意思，就是这样的一个规定。（呵呵，牛刀刚刚在网上查到的，这叫现炒现卖。）

```
...
```

```
        A & operator++(int)
        {
                x--;        //牛刀又要恶心了，呵呵
                y--;
                z--;
                return *this;
        }
};

void main()
{
        A a(2,3,4);
        a.print();
        a++;
        a.print();

}
```

网络是个好东西啊，用得好对我们的学习很有帮助，我们既然赶上这种时代了，就需要适应这个时代。

行了，我们再来，我们返回类型真的一定要是原类吗？不一定，我们看：

```
...
        int operator[](int i)
        {
                if(i==0)return x;
                if(i==1)return y;
                if(i==2)return z;
                return -1;
        }
};

void main()
{
        A a(2,3,4);
        a.print();
        int i,j,k,l;
        i=a[0];
        j=a[1];
        k=a[2];
        l=a[5];
        printf("====================\ni=%d,j=%d,k=%d,l=%d\n",i,j,k,l
);
}
```

什么结果呢？

**引用:**

x=2,y=3,z=4
==========
i=2,j=3,k=4,l=-1

大家看，感觉像数组，却又不是，挺别扭是不是，其实这就告诉我们，我们运算符重载是对于所有运算符的，而并不是和哪一个运算符有什么关系。

至少这个运算符的功能，那完全在于我们想叫这个运算符有什么功能。

```
...
        int operator()()
        {
                return x+y+z;
        }
};

void main()
{
        A a(2,3,4);
        a.print();
        int i;
        i=a();
        printf("==========\ni=%d\n",i);
}
```

看，甚至能对括号进行运算符重载，什么意思呢？函数调用，大家看 i=a()，是不是很像调用一个函数呢？这种事情可以越做越恶心，呵呵，我们再来：

```
...
        int operator()(int a,int b)
        {
                return a+b;
        }
};

void main()
{
        A add(2,3,4);
        add.print();
        int i;
        i=add(5,6);
        printf("==========\ni=%d\n",i);
}
```

结果：

**引用:**

x=2,y=3,z=4
==========
i=11

是不是这些感觉有点怪怪的呢？怎么这个类很像很像一个函数啊，而不像是一个类啊，呵呵，这就是运算符重载的特别魅力。

从上面的一些实践来看，我们做运算符重载，几乎可以重载任何一个运算符，以来达到我们的目的，下面我们就用上面这些知识，来为大家做一个类，复数类。

什么叫做复数呢？复数的运算规则是什么呢？我想大家都懂，我们的复数类我们取最前面一个字母 C 作为类名，来正式做一个可以做复数的加减乘除的复数运算：

```
#include <stdio.h>
#include <math.h>
class C
{
        float r,i;
public:
        C():r(0.0),i(0.0)
        {
        }
        C(float a,float b):r(a),i(b)          //注：这种编程方法大家要习惯哟，方便啊，呵呵
        {
        }
        void print()
        {
                if(i>=0.0)printf("%f+%fi",r,i);
                else printf("%f-%fi",r,-i);
        }
        C & operator=(C &c)
        {
                r=c.r;
                i=c.i;
                return *this;
        }
        C operator+(C &c)
        {
                float x=r+c.r;
                float y=i+c.i;
                return C(x,y);
        }
```

```
        C operator-(C &c)
        {
                float x=r-c.r;
                float y=i-c.i;
                return C(x,y);
        }
        C operator*(C &c)
        {
                float x=r*c.r-i*c.i;
                float y=r*c.i+i*c.r;
                return C(x,y);
        }
        C operator/(C &c)
        {
                float t=c.i*c.i+c.r*c.r;
                float x=r*c.r+i*c.i;
                float y=c.r*i-c.i*r;
                x=x/t;
                y=y/t;
                return C(x,y);
        }
        float operator[](C &c)
        {        //牛刀有点恶心了，用[]当作求模，呵呵
                return (float)sqrt(c.r*c.r+c.i*c.i);
        }
        //*/
};

void main()
{

        C a(1.0,1.0),b(2.5,-2.5),c;
        a.print();
        printf("\n");
        b.print();

        printf("\n");
        c.print();

        printf("\nc=a+b\n");
        c=a+b;
        c.print();
        printf("\na[c]=%f\n",a[c]);
}
```

上面的程序中对“[]”的重载有些牵强，但目的就是让大家知道可以这样用，我们重载运算符可以随心所欲，就是这个意思。当然上面的程序还没完，大家还可以定义一些运算符或者用一些函数去求复角啊，等等，想要怎么做就可以怎么做，多方便啊，大家看，我们只需要做一个类，就可以随意地对复数进行加减乘除，以及一些相应种种运算，上面的类的功能我们可以随意地扩展，我们想要什么样的功能就往里面加就行了，这就是 C++中的类。和 VB 中的对象不同的就是，那个对象是人家做好的，方法就那几个，想新的没有了，而我们 C++中却可以随心所欲，说句难听一点的就是可以为所欲为。

也正因为这种种的特点让我们的 C++语言成为多数朋友们的首选，编程之路还有很长，牛刀的基础教程部分就至此为止，告一段落了，希望大家上机，实践，实践再实践，只有实践中才能够发现问题，才能够解决问题，才能够使得书本知识与牛刀教程转化为我们自己的能力，做程序的武器。

行了，牛刀也继续冲冲电了，估计在短期内牛刀先不出教程了，前面的基础让大家消化消化，并且希望大家有问题提出来，大家来讨论，毕竟牛刀也还有好多不懂的，看许多高手们讨论却没有牛刀的份，其实挺不是滋味，呵呵，学无止境啊!!!

**对象实践〇六五：什么是对象**

这个问题有些怪异，学 VB 的人，学 DELPHI 的人可能认为自己在“面向对向”，学 JAVA 学 C++的人可能也有能力去实现一些“面向对象”的技术，可到底什么是对象？是一个类吗？是一个按钮？还是一个窗体？或者是一个图片或声音文件？

这些都不是，对象是定义了的一个数据，在内存中占据一定的位置：

```
void main()
{
        int i;
        i=4;
}
```

上面的 i  其实就是对象，只不过我们不这样说罢了，并且我们对其的使用也仅限于一些运算或者是中间变量。

```
struct A
{
        int x;
        int y;
};

void main()
{
        A a;
        a.x=5;
        a.y=6;
}
```

上面的 a 也是一个对象，这个对象有了一定的结构，我们经常将这个对象一起使用，只不过其实这种对象与上面的 i  其实没什么区别，看程序：

```
void main()
{
        int ax;
        int ay;
        ax=5;
        ay=6;
}
```

这样的程序与上面的结构体是不是一样的呢？其实是一样的，只不过当我们使用了结构体，重名的机率就少了，定义的数据量也少了，什么意思？我们还是

来看程序：

```
struct A
{
        int x;
        int y;
        int z;
};

void main()
{
        A a,b,c,d;
        ...
}
```

上面我们定义了多少的变量呢？四个结构体变量，并且使用很方便，而如果我们不这样做，直接定义变量要达到相同的效果就要定义 12 个变量，使用结构体的好处也仅此而已。

当然我们不能将其说得一无是处了，结构体的还有一个好处就是，可以通过结构体指针将一个结构体中的数据全部由函数的返回值带回，这一点可真是定义十二个变量做不到的。

所以我们说，定义出的 a,b,c,d 也是一个对象，但这个对象的功能相比上面的 i 要强了许多。

而我们正常认为的对象是由类定义的变量，看程序：

```
class A
{
        int x,y,z;
pbulic:
        A(int a,int b,int c):x(a),y(b),z(c)
        {
        }
        A():x(0),y(0),z(0)
        {
        }
        print()
        {
                printf("x=%d\ny=%d\nz=%d\n",x,y,z);
        }
}

void main()
{
```

```
        A a;
        a.print();
}
```

这绝对是一个值的飞跃，我们定义的数据突然有了功能，上面的定义后的功能即同时为类中的私有成员变量 x,y,z 赋了值，而 print() 成员函数则可以显示出私有成员的值，我们的“对象”一下子“活”了起来，似乎有了生命，所以我们一般所说的对象不特指的话就是指的由“类”定义的数据：

请注意牛刀的说法，上面的类 A 不是对象，而由 A 定义的数据 a 被牛刀称之为了对象。

和大家的传统观念是不是有些不同呢？特别是学 VB 的朋友是不是觉得特别的别扭，其实 VB 中称这种数据为“属性”，而定义的函数称之为“方法”，而调用这些函数与数据的触发条件称之为“事件”。

属性与方法好像还比较好理解，那么什么叫事件，可能就不那么明了了。那是消息响应的过程，想必看过牛刀前面教程的朋友对这种消息机制应该有一个相对全面的了解，而事件就是相应的消息触发条件，例如我们按下了鼠标按键，就触发了一个鼠标按下的事件，而抬起时则触发了一个抬起事件与单击事件。当然由于前面介绍的也比较详细，这一讲我们就借助一些程序来理解我们是怎么从什么都没有开始来创建一个对象的吧。

再声明一遍，对象与我们的窗体、按钮没有任何关系，如果说有关系的话，那是人与衣服的关系，对象仅仅是一件很漂亮的外套而已，而并是我们这个人。

所以说，我们做一个控制台应用程序，也一样可以做一个对象，或者说像我们上面说得那样，i,a,等等，都是对象，但我们与大家讲解对象时，还是用 API 函数部分做的程序来与大家讲解，毕竟这部分内容太经典了，并且我们做出的外套几乎可以变成一个一个的 DLL 文件，使得我们形成一个很巨大的对象库，从而使得我们做程序显得更加得方便。还是老一套，与大家做一个窗体程序吧：

```
#include <windows.h>
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{
        HWND hwnd;
        MSG Msg;
        WNDCLASS wndclass;
        char lpszClassName[]="窗口";
        char lpszTitle[]="牛刀教我做自己的窗口";
        wndclass.style=0;
        wndclass.lpfnWndProc=WndProc;
……
```

```
        return Msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        switch(message)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hwnd,message,wParam,lParam);
        }
        return (0);
}
```

中间好多的东西用不着给大家列出来了吧？复制一段代码，贴到我们程序中来就行了，下面给大家说说牛刀以后需要用到的几个描述方式：

• 增加全局变量……：意思为增加一个全局变量，即在 WinMain 函数之前增加

• 增加一个类……：建议大家使用专用工具增加，一般在增加后，需要用 #include "类名.h" 将我们的类包含进来，也在 WinMain 函数之前增加

• 增加消息处理函数变量……：即在 WndProc 函数之中，switch 消息循环之前增加。

• 增加消息处理……：即增加相应的消息，即在 switch 消息循环中增加一个 case WM_……: 消息了，每增加一个消息别忘了增加一个 break; 否则就会顺带执行了其他的消息处理。

• 在消息处理函数中增加变量……：明白吧？在消息处理函数中增加一个变量定义，一般不这样使用，但注意了，如果增加，一定要在适合的时候将其析构掉，也就是说注意其生命周期。

行了，估计暂时能够使用的就这么多吧，下面我们还是从 API 的基础讲起，为什么要讲基础呢？因为那是 Windows 编程的基础，我们知道一点，Windows 与控制台程序远远的不同了，有什么不同？

• 控制台程序是单任务操作系统，每次只能运行一个程序，即使在某些做得好的控制台程序在打字时可以听歌，但那并不是运行了两个程序，可能是一个程序的两种功能而已。而 Winodws 程序是一个多任务的运行环境，所以我们每做一个程序，必须让系统知道我们所处理的哪个窗口的哪个地方，也就是说，在多任务系统中，系统必须为每一个窗口发放一个唯一的三十二位常量，使得我们在用程序对窗口操作时，系统知道我们这个程序是对哪个窗个进行操作的，于是，我们称这个唯一的数据称之为“句柄”，一般使用大写的“H”开头来定义，句柄

一般我们不能定义，意思也就是说，这必然是一个变量，而不可能是一个常量，所以我们需要得到她，那么有几种获得方法呢：

一、参数传递：

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow);
```

大家看上面的入口函数中就有了两个句柄，并且都是 HINSTANCE ，我们称之为：“实例句柄”，或者叫“程序句柄”吧，总之，你爱怎么叫就怎么叫，我们不是要知道怎么个称呼，而是要知道其功能，这个句柄的功能就是代表着我们这一个程序。所以说，我们一旦需要用到这个程序的一些东东了，那么就必须得到这个程序的一个信息啊，不过暂时用处还不是太大，好像就是注册类时用到了一下，我们以后慢慢聊。

二、函数返回：

当然是 API 函数返回了,那些 API 其实是系统定义的函数啊,所以说，一个好的函数就可以给我们返回一个句柄，例如：

WM_PAINT: 消息中的 hdc=BeginPaint(hwnd,&ps);,在其他地方也可以使用 hdc=GetDC(hwnd); 来返回一个,返回的是什么呢？我们称之为设备描述表，好像以前与大家说过了，为什么我们要得到这个句柄，在这个设备描述表上作图呢？因为这是一个接口，我们这样做出程序来就不需要考虑设备的驱动了，也就是说，任何设备在我们面前没有差异了。窗体、屏幕、打印机、绘图仪，都是一样，都是这一段代码，而至于在什么地方出什么图形，那是设备驱动的事了，当然这个设备描述表的句柄也是 Windows 系统要管理的句柄，当然其中还有一个句柄呢：看、hwnd ，这个句柄哪里来的呢？原来啊，这个句柄是消息处理函数中传进来的，也就是上一个来源：参数传递。

当然,还有例如 画笔句柄啊 HPEN 画刷句柄啊 HBRUSH 也都是系统要管理的唯一的一个三十二位值，所以说，当我们用完这些句柄时，一定要记得释放此句柄。

如果你容易忘，那么，这一点就是类的一个强项了，我们试着为我们的程序做一个类吧，我们就测试着玩玩用，那么我们这个类的名称就叫做： test 类吧，那么我们怎么做这个类，大家还知道吧？

• 插入/新建类...

然后在类名（Name）中填入：test，点击“OK”就一切完毕，其实就是这样简单。

当然做完这个类的同时，我们就同时在我们程序前面加上：

#include "test.h"
    并且做一个定义：（全局变量）
test t;

    对了，在这里与大家说一下，一般我们做一个类的话，开头字母用大写，或者全大写，但事实上大写小写一样，那只不过是人们的习惯问题，但一般我们还是尊重大伙的习惯，当然在这里我们就不强调那么多了，为什么呢？因为我们是来“实践”的啊，而不是来强调什么。

    当然如果你还没有定义的话，你就将第一个改大写好了，但事实上无所谓，我们要的是知识，而不是习惯，呵呵，但有一个好习惯也是好啊。

    你可能有些纳闷，怎么才写一个  test  再按一个“OK”，我们却立刻又定义上了啊？

    其实不奇怪啊，我们的程序已经为我们做好这个类了，下面就是代码，这段代码相当优秀：

```
// test.h: interface for the test class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_TEST_H__879AB00D_64F1_4E9C_9B25_2C0DE7712309__INCLUDED_)
#define AFX_TEST_H__879AB00D_64F1_4E9C_9B25_2C0DE7712309__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class test
{
public:
        test();
        virtual ~test();

};

#endif
// !defined(AFX_TEST_H__879AB00D_64F1_4E9C_9B25_2C0DE7712309__INCLUDED_)
```

    大家看到了吧，这就是一个最简单的类，其实的定义都是自动为我们写好了

的，下面我们为这个类增加一些东西吧，当然，我们在做这个类时，这个类并不认识我们的 HWND 、HDC ……这些东西，想要让这个类也认识怎么办呢？这就需要在前面同样地加上：

```
#include <windows.h>
```

是的，必须要加上，否则你如果用到这些句柄就肯定要报错。

下面我们就定义一些私有成员变量：什么叫私有大家知道吧：

```
class test
{
private:
        HWND hwnd;
        HDC hdc;
……
```

注意哟，前面一定要将 windows.h 头文件包含进来，否则这儿就准备出错了。

既然我们有了这个设备描述表句柄，我们就试图首先得到她，那么怎么办呢？当然需要有一个函数来得到了，再加入一个函数：

```
void test::Init(HWND hwnd,HDC hdc)
{
        this->hwnd=hwnd;
        this->hdc=hdc;
}
```

这里牛刀使用了 this 指针，大家还记得吧？这就是牛刀为什么向大家强调一些基础，就是这个原因，这些东西用起来方便啊。

对了，你加函数是手动加的，还是使用系统工具的？反正牛刀是用工具的，工具放在那，不用白不用，呵呵。

好了，我们这里有得到句柄的了，那么我们在做得到句柄的程序的同时就要想到一个问题，那就是：释放句柄，这当然是要在析构函数中完成的事喽：

```
test::~test()
{
        ReleaseDC(hwnd,hdc);
}
```

这里牛刀其实做得有点牵强，不过只不过是说明这么一个意思吧，这样的程序有什么好处呢？我们随时随地就可以在屏幕上画图了，当然上面的 Init 函数必须要有地方调用，并且只能被调用一次，从这一点上面来看，牛刀这个类做的相当不成熟，假如有人调用了第二次呢？所以一般这种工作放在构造函数中去完成。

那么既然我们定义了，我们就调用一次吧，在哪儿调用呢？当然需要响应一个 WM_CREATE 消息，这种消息，在 VB 中就被称之为事件：

```
Private Sub Form_Load()

End Sub
```

说实话，还真很向往那种事件编程，真的很方便，我们并不是说 VC 好，VB、DELPHI 就一无是处，真的是各有千秋的，人家 VB 做起来就是方便：

```
……
        case WM_CREATE:
                t.Init(hwnd,GetDC(hwnd));
                break;
……
```

怎么样，还习惯吧，大家看，我们将 API 函数 GetDC 直接放到了函数调用中，少定义了一个句柄，其实说实话，我们这样做的目的也就是为了实现这种类的封装，有时候真的很方便哟。

当然我们还对鼠标位置比较感兴趣怎么办呢？定义两个坐标（当然是私有成员喽，）来接收鼠标位置信息呗：

```
……

private:
        HWND hwnd;
        HDC hdc;
        int x,y;
……
```

所以我们现在很想在鼠标移动时，接收到鼠标坐标，那么我们就需要定义一个函数来接收，当然这个接收的事件部分就在响应鼠标移动消息就一切 OK 喽：

```
void test::MouseMove(int x, int y)
{
        this->x=x;
        this->y=y;
        Ellipse(hdc,x-20,y-20,x+20,y+20);          //为了看我们鼠标移动的效果，大家看，我们一旦将句柄变成了类成员，方便啊，呵呵
}
```

下面我们就要响应鼠标事移动事件：

```
……
        case WM_MOUSEMOVE:
```

```
                t.MouseMove(LOWORD(lParam),HIWORD(lParam));
                break;
……
```

大家看看，我们鼠标的响应又是一行代码就一切 OK，而其他一切的工作都被我们拿到了类中去进行了，这就是对象的魅力，我们的程序其实已经相当好玩了，大家编译一下，移动鼠标，就可以看到效果了，呵呵。

当然，我们上面的 Ellipse 函数其实仅仅是做测试用的，一般如果我们真的做程序，可以将那一行代码注释掉。

同理，我们可以很容易地在我们 test 类中实现鼠标的按下、抬起，键盘的按下等等，每一个响应，我们就在类中为之定义一个函数，我们只需要将我们需要做的函数做进类中相应的函数中就一切 OK 了。

大家感觉是不是很像 VB ？怎么我们的程序也一下子就变得只需要去响应我们关心的“事件”就一切 OK 了呢？

是啊，VB 是什么，那只不过是我们定义好了的一个类，那个类可以将窗体、按钮、文件框…… 全都定义好了而已，而我们目前面临的困难就是没有现成的定义而已。

而一旦我们也能做这些定义了，大家想想看，我们的类是不是随着我们增加的函数与成员变量的增多，我们的类会变得越来越强大呢？

事实也是如此，我们只要有那种必要，我们可以做一个像模像样的类库。为了证明我们的观点。我们在以后的教程中，使用我们的类来为大伙构造一台电脑，

当然是一台能使用的电脑喽，我们通过这个类的构造达到两个目的：
一、掌握电脑与接口的一些基本知识
二、掌握面向对象的实质。

说实话，如果用类去封装一个按钮或者菜单滚动条之类的东西，反而更加方法，那些东西，大伙随便找一些书来看看就自然能够上手了，所以我们还有一个目的就是教会大家面向对象的思想。

好了，先问问大家，当我们买回一台电脑时，我们买回了几台呢？可能你说，不就是一台吗？不对，其实我们一下子就买回了好几台，只不过有几台电脑我们称之为专业电脑或专职电脑。

什么意思呢？意思就是，我们的显卡其实就是一台独立的电脑，声卡也是。不信你将显示器关了，其实电脑还在运行，你把音箱拿了，我们的电脑还是可以顺利地运行。
你不要和牛刀抬杠，那么我将显卡成声卡拨了呢？电脑还能运行吗？不能运行了。

为什么呢？因为现代的电脑系统已经将显卡与声卡给管理起来了，不要说显卡与声卡，有些电脑你将鼠标拨了都会出问题，难道不是吗？

四几年的大三间房子的巨兽，那么大有家伙，没有显示器，没有键盘，那种日子也过来了，好像那时连汇编都没有，你想那些人是怎么样过来的？呵呵。

现代的显卡越来越专业了，你听说过显卡支持 3D 吗？你听说过显卡支持 OpenGL 吗？是的，那都是显卡的功能，而我们好多 3D 游戏的显示其实并不是在内存中来处理的，而是在显存中。为什么会这样呢？

这还是因为我们显卡是一台独立的专职电脑，而显存与内存之间的通迅，就好像是我们现代的网络通讯一样，我们来举一个例子：

我们要在显示器上的(100,100)到(300,300)画一条线，那么我们有两种方案，一种方案就是，我们在内存中处理线上的每一个点，这样形成近 200 个坐标的数据，一个坐标假如是四个字节，那么就需要 2 0 0 × 4 ＝ 8 0 0 字节的通讯量，但如果我们只传送过去二个端点，八个字节，假如再告诉显卡是用这两个端点干什么，假如也要二个字节，那么数据通讯量就是 1 0 个字节。也就是内存法生成的图像要比显存法生成图像的数据传送量多 7 9 倍的数据传送量，而大家想，(0,0)到(400,400)，我们显存法还是 1 0 个字节，而内存法一下子是显存法的 1 6 0 倍，这种数据传输量是吓人的。

所以我们当做 3D 技术时（我们有空给大家模拟），用一个打点的 API 函数完成可以胜任，但效果的不同就是我们做出来的图像会不断地闪烁，而用 OpenGL 提供的函数来做这些东西则一点问题都不会有。并且运行很流畅，这是为什么，因为你的显卡支持了 OpenGL，因为你的显卡就是一台计算机。

行了，最后我们给大家介绍一个 API 函数（虽然好多人都学过 API，但牛刀还是要介绍的，毕竟并非所有看牛刀教材的都已经吃透了每一个环节，何况我们的论坛还会不断地有新人前来光顾）：SetPixel

我们响应一下 WM_PAINT 消息，并且假设类 test 中已经有了一个函数 Draw(HDC hdc); 这其实就是 C 语言中的空函数技术，大家不要以为这不是技术，呵呵，我们以后的主要精力就放在去实现这个函数了。

虽然这个函数不存在，但我们提前使用，然后再去增加这个函数，这是我们做程序的基本思想，我们可以在做这个函数之前提前规划好我们的程序，如果我们忘了，大不了编译时程序报错，一报错，我们不就知道我们还有什么函数没有实现了吗？

其实也是这么回事，有时候错误是一种好事，怕就怕有错误了，编译器还发现不了，那种程序最烦了，唯一的手段就是——调试，相当复杂。

……

```
        case WM_PAINT:
```

```
                hdc=BeginPaint(hwnd,&ps);
                t.Draw(hdc);
                EndPaint(hwnd,&ps);
                break;
```
……

请大家别忘了在消息函数中增加两个变量：

```
HDC hdc;
PAINTSTRUCT ps;
```

然后我们就要去实现 test 类中的 Draw(HDC hdc); 函数了，在类中增加相应的函数：

```
void test:Draw(HDC hdc)
{

}
```

好了，这样一来，我们的程序就可以顺利运行了，编译一下试试，然后我们再增加一个函数：

```
void test:Draw(HDC hdc)
{
        SetPixel(hdc,20,20,RGB(255,0,0));
}
```

再运行，就会发现，我们的屏幕的 200,200 这个位置被置成了红色点，我们下面就来修改这个函数，用打点的办法来显示一幅图片，下一讲我们再讲讲对这些函数的封装与处理。（注意哟，我们没有用到任何图片处理函数，只是打点与文件读写，我们实践并不是要实践别人已经做好的东西，而要来实践实践，我们如果想做这些东西，应该怎么做）

好了，不多聊了，下面我们来将这一个函数做全，我们只用打字点办法来在我们的窗体上显示一幅图片：

```
void test:Draw(HDC hdc)
{
        unsigned int len,hei,wid;
        unsigned char *ch,*pch;
        unsigned int i,j;
        FILE *fp=fopen("test.bmp","rb");        //由于这里要用来文件操作，所以需要将 stdio.h 头文件在这个 cpp 文件中包含进来，否则报错。
        //请大家复制一份图片，并将其名称改为 test.bmp 复制到相应的程序目录下
        fseek(fp,0x02,SEEK_SET);        //这里涉及到具体的文件结构，不详细讨论了，我们下一讲与大家详细讨论
        fread(&len,4,1,fp);
        fseek(fp,0x12,SEEK_SET);
```

```
        fread(&wid,4,1,fp);
        fseek(fp,0x16,SEEK_SET);
        fread(&hei,4,1,fp);

        ch=new unsigned char[len];

        fseek(fp,0,SEEK_SET);
        fread((void*)ch,len,1,fp);
        fclose(fp);
        pch=&ch[0x36];
        wid=wid*3;

        if((wid%4)!=0)
        {
                wid=wid/4;
                wid++;
                wid=wid*4;
        }
        for(i=0;i<hei;i++)
        {
                pch=&ch[0x36+wid*i];
                for(j=0;j<wid;j++)
                {
                        if(!(j%3))SetPixel(hdc,j/3,hei-i,RGB(pch[2],p
ch[1],pch[0]));
                        pch++;
                }
        }
        int test=0;
}
```

大家仔细体会我们的函数，我们除了一个打点的 API 函数，还有就是文件操作，已经可以很优秀（牛刀自己这么认为，呵呵）地来显示一幅图片，唯一的缺憾就是速度太慢，我们可以明显地感觉出来程序是在画图，但请大家注意啊，这程序是我们自己做出来的啊，可不是显卡的功能啊，所以说，如果学过相应图片处理的函数的朋友不要过多地指责，因为你们那些函数可能是由硬件来处理了，而牛刀的程序却是每一个点都是实实在在的一个点，我们的程序是用点画出来的，下面几讲我们重点讲一讲这种图像处理技术，并且对图像处理进行类的封装，看看我们的类到底能做到多么地强大。

当然喽，提示一下，上面的程序讲入的图片必须是真色彩， 2 4 位的，我们对其他情况没有做任何处理，这一讲肯定是来不及了，呵呵。

## 微软积木〇六六：初试身手

大家过年好，牛刀又来喽，呵呵，论坛一直老上不上，哎，不知什么原因，牛刀这下给大家拜个晚年吧。

现在既然能上了，我们就如期地给大家做新东东吧，呵呵。

看一段代码，那是一段代码；
学一段代码，那是一段知识；
品一段代码，那是一段思维。
有代码的人，可以将代码变为软件；
有知识的人，可以将软件变成代码；
有思维的人，可以将思维变成代码。

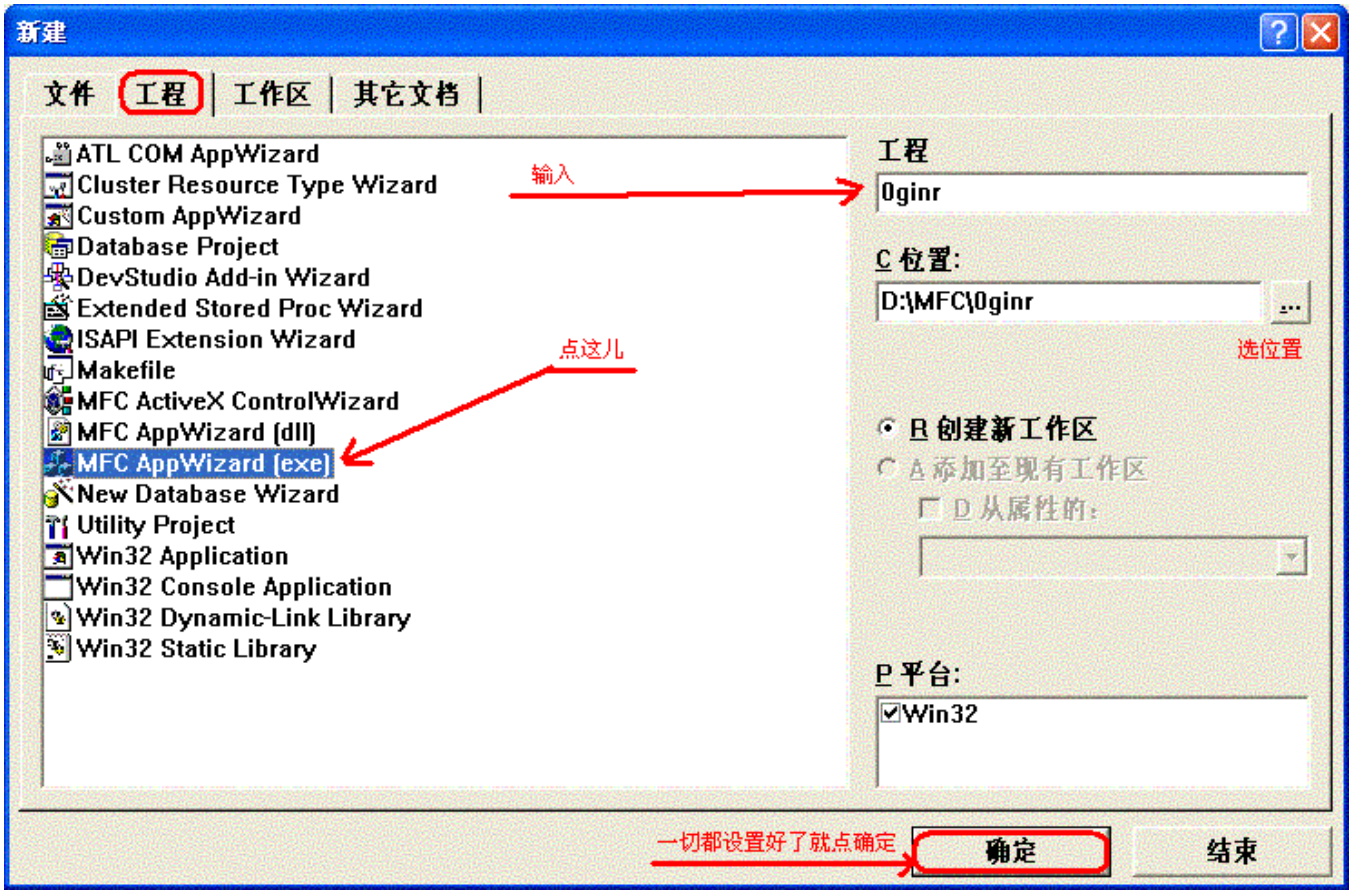
——牛刀新年寄语　二○○八年春

看过微软的ＭＦＣ图标的朋友一定有一种感觉，怎么那么像积木啊？

不错，微软的 MFC 其实就是一个大型的积木，做程序就好像搭积木一样简单。下面就随着牛刀来做一款真正的“软件”。

第一步，当然是打开ＶＣ（建议使用ＶＣ６．０中文版，虽然翻译不怎么样，但感觉毕竟要舒服得多，呵呵）。

第二步，打开/新建/[工程]/...（注：下面的设置看图片，这叫“看图说话”，呵呵，感觉好不好？其实不能直接上传网页，



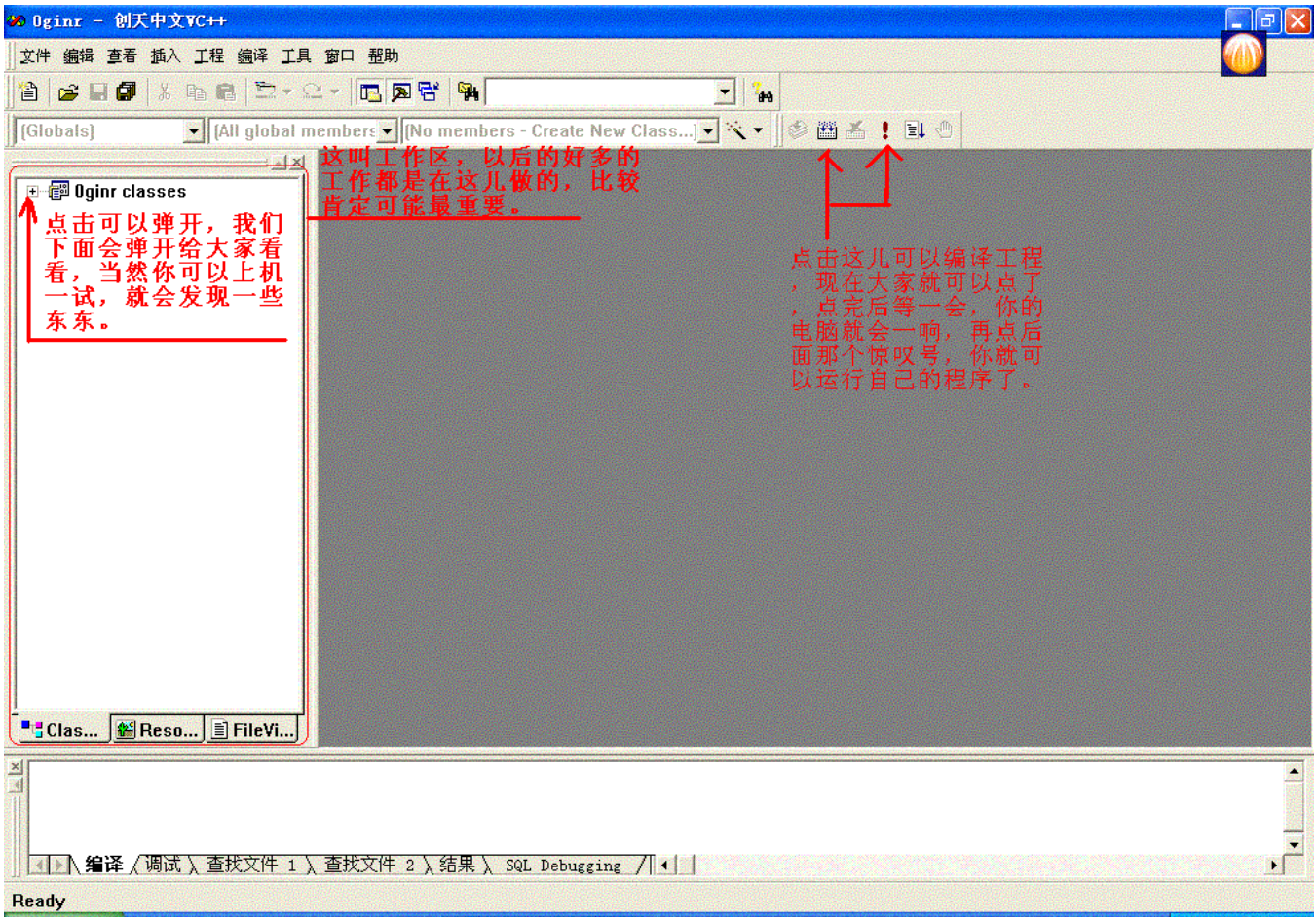
否则感觉还要好）

设置好了就按确定，进入下面：

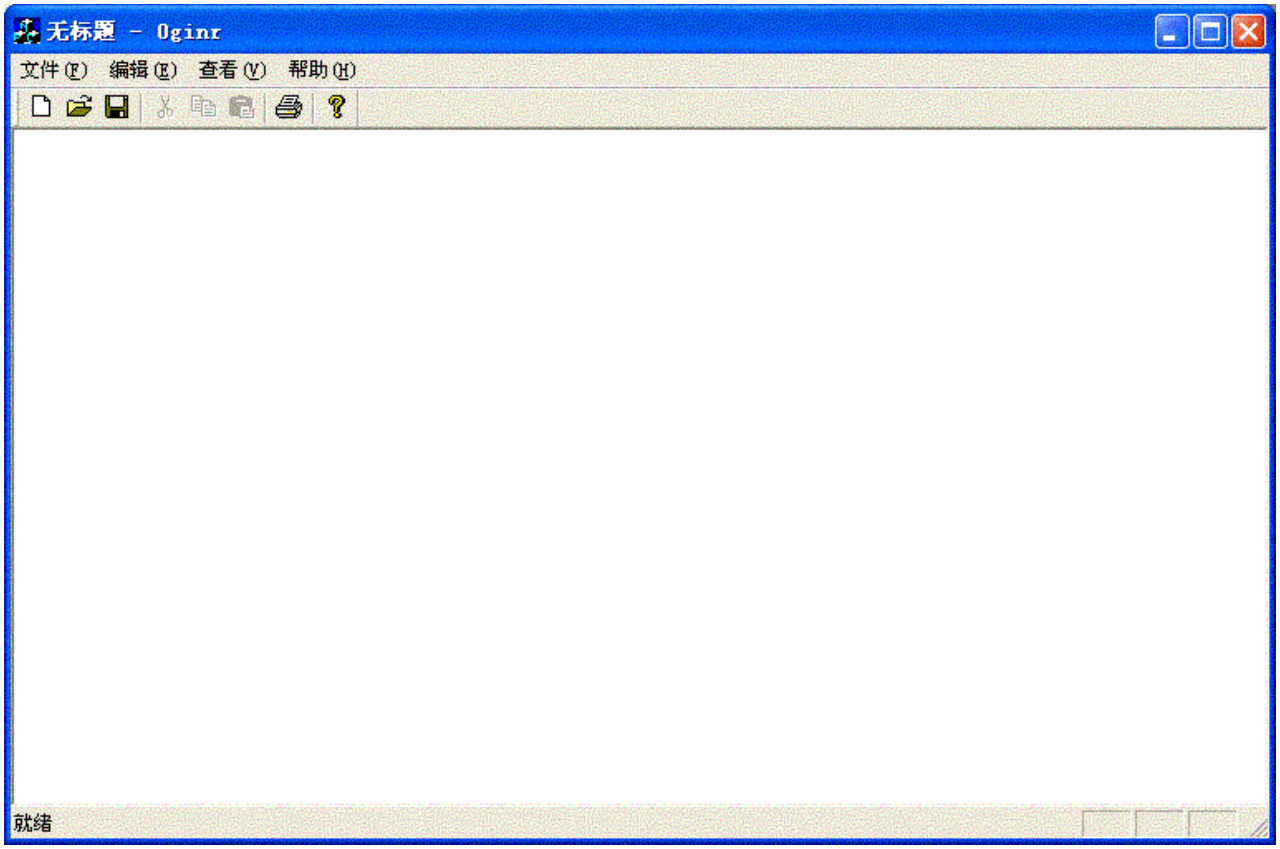
第三步：选择“单个文档”，然后点“完成”，出现一个新的对话档，直接点“确定”，就建立了一个新的软件界面。

牛刀说得简单吧？基于一般的书本肯定会将好多的东东作一些介绍，有那必要吗？如果你学过了，你可能不需要看牛刀的这些东东了是不？如果你还没学，介绍给你你看得懂吗？（看得懂才怪，哼）（对了，以后上面的步骤我们统称之为“建立一个单文档程序”）

好了，大家看下面的图：




编译程序，点惊叹号，就会出现下面的软件界面，是不是感觉太太太太太太好了？

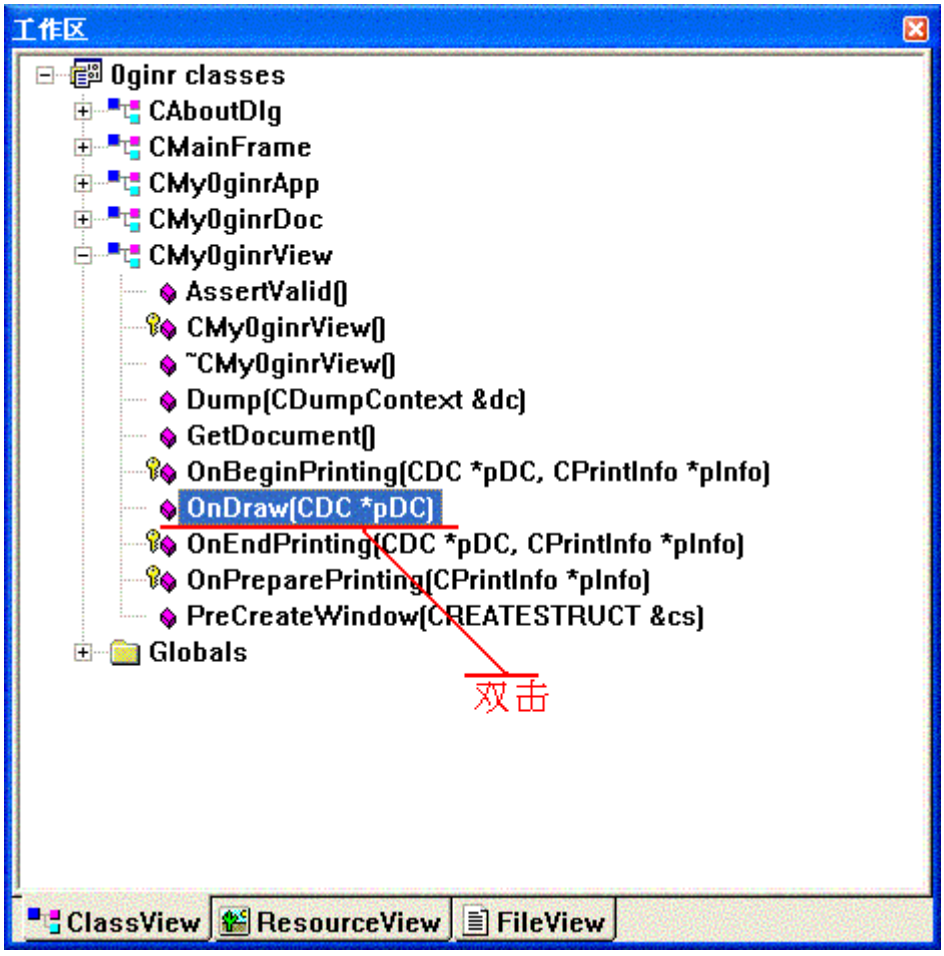


你瞧瞧，你瞧瞧，我们一行代码没有敲，可出来的东东却是“麻雀虽小，五脏俱全”啊，我们通常所说的软件的全部都具有了，有菜单，有工具栏，状态栏，还有标题栏，并且菜单上的东东都是可以点的，点点看看，是不是与我们通常的软件就基本一致了？

不错，是一致了，我们还没有做任何事件，我们的软件就已经基本成形了。

可这软件是怎么成形的呢？如果你看一些 MFC 书籍，那是八仙过海，各显神通，都有其独到的解释，当然，牛刀的解释不能与那些书一样，如果一样，那大家岂不去买本书看看就一切 OK 了？

好了，大家看见边上的工作区了吗？点开她：




我们看到有几样东东，我们称之为“类”

CAboutDlg

CMainFrame

CMy0GinrApp

CMy0GinrDoc

CMy0GinrView

一共有五个类，其中上面两个是所有用这种办法生成的软件中固定的，由于我们自己的名称的第一个字符是数字的零“0”，所以系统自动给我们在前面加上了 CMy，后面来看，有三种，一种为 App 一种为 Doc 一种为 View

好了，我们双击 CMy0GinrView ，然后双击其中的 OnDraw 函数，就进入了下面的代码编写状态：

```
void CMy0ginrView::OnDraw(CDC* pDC)
{
        CMy0ginrDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
}
```

我们这一讲与大家不会讲太多的实质内容，所以这里的代码部分大家可以放心地删去：

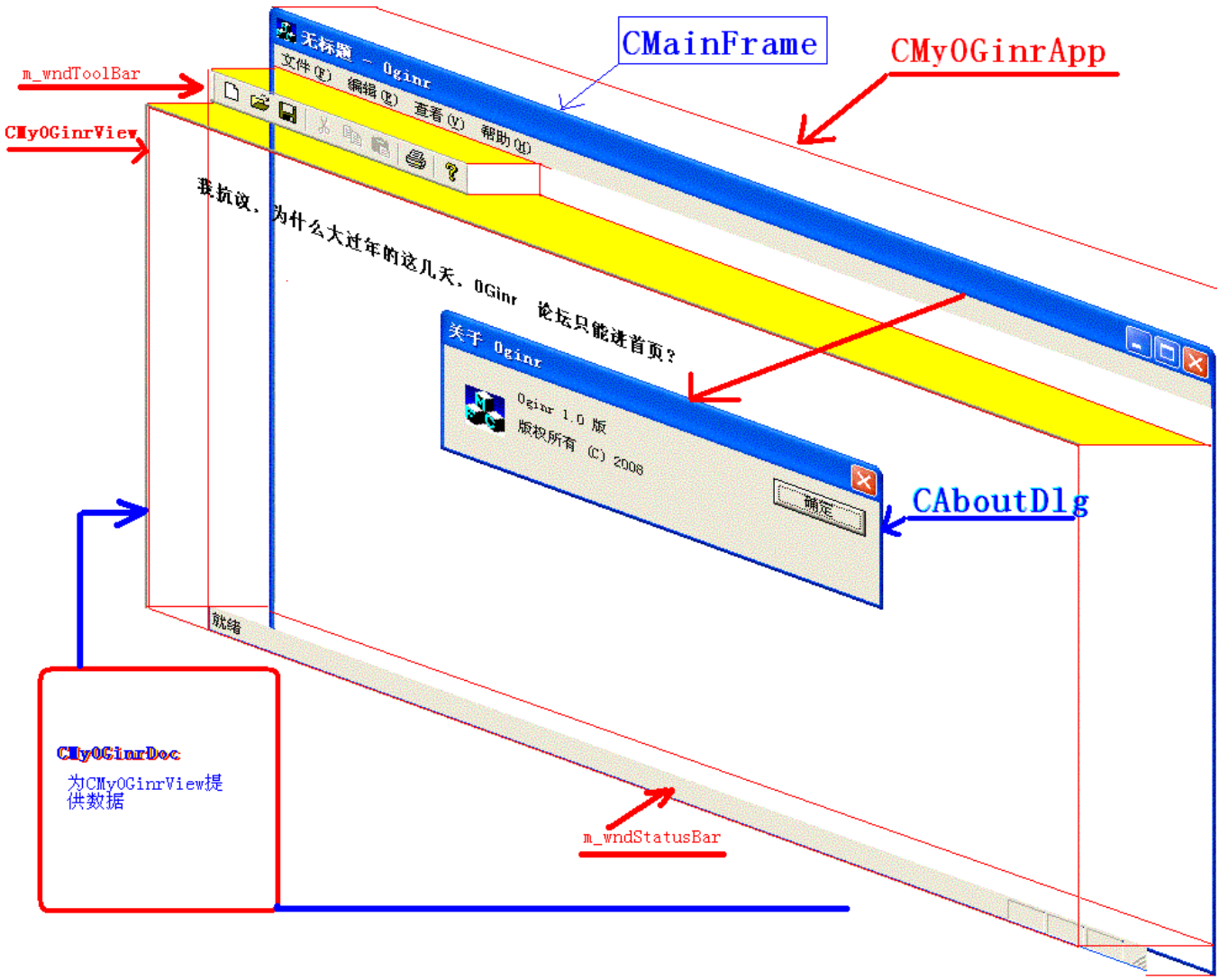
```
void CMy0ginrView::OnDraw(CDC* pDC)
{

}
```

大家看，参数中有一个 CDC 指针，那就是代表着我们生成程序中的一些东东（什么东东我们以后再讲）。我们顺手拿来就用，敲入下面的代码：

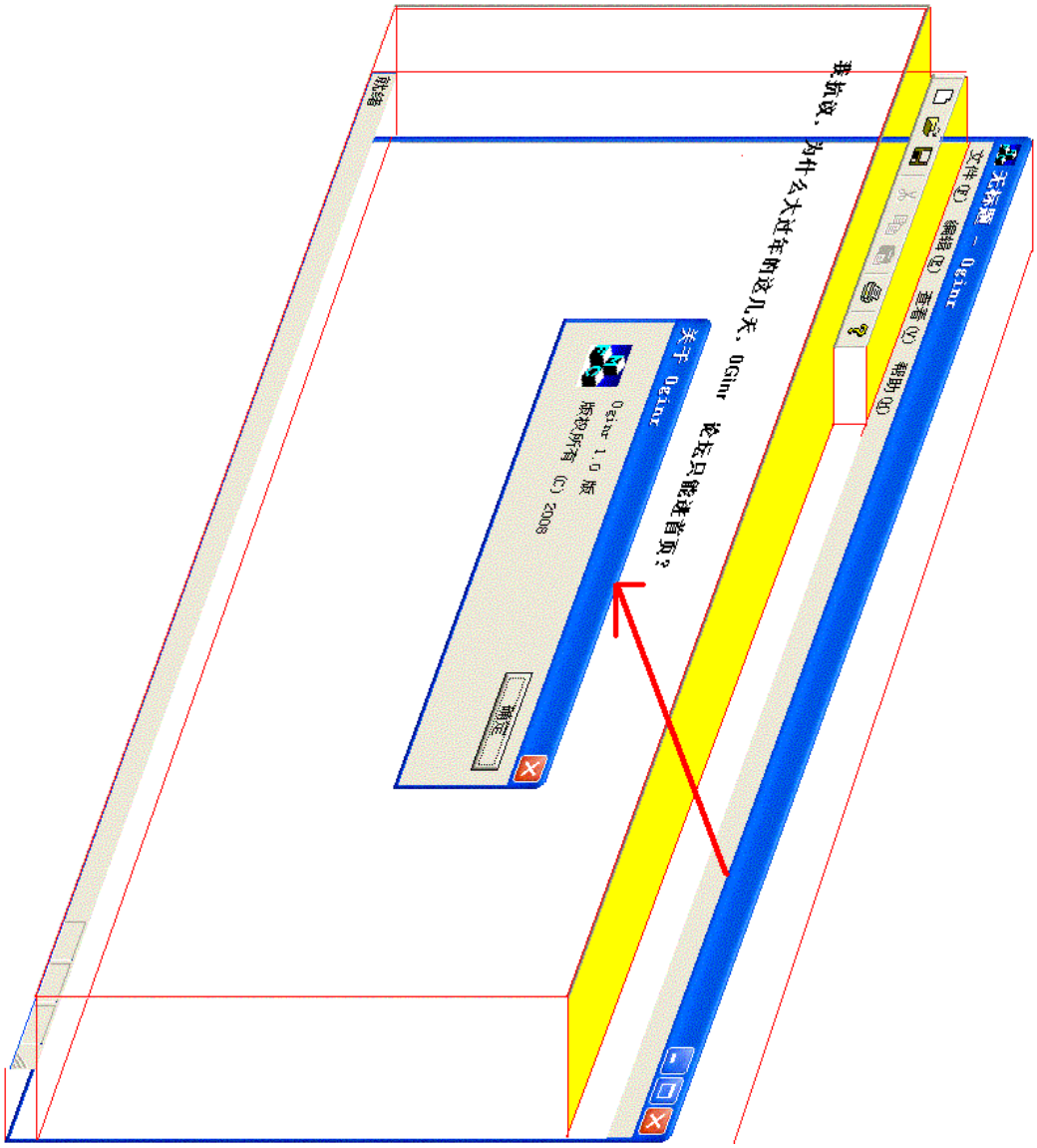
```
void CMy0ginrView::OnDraw(CDC* pDC)
{
        pDC->TextOut(40,40,"我抗议，为什么大过年的这几天，0Ginr 论坛只能进首页？");
}
```

大家再将软件编译一下，就可以看到新的结果了：（注：下面进入实践阶段后，主要的实践过程牛刀就不给图片了，全是改改代码啦，都是小菜一碟，你做时，随时都可以编译，看结果）

好了，下面大家看下幅图，是被牛刀给“肢解”了的软件，中间给大家一些软件与类之间的对应关系：

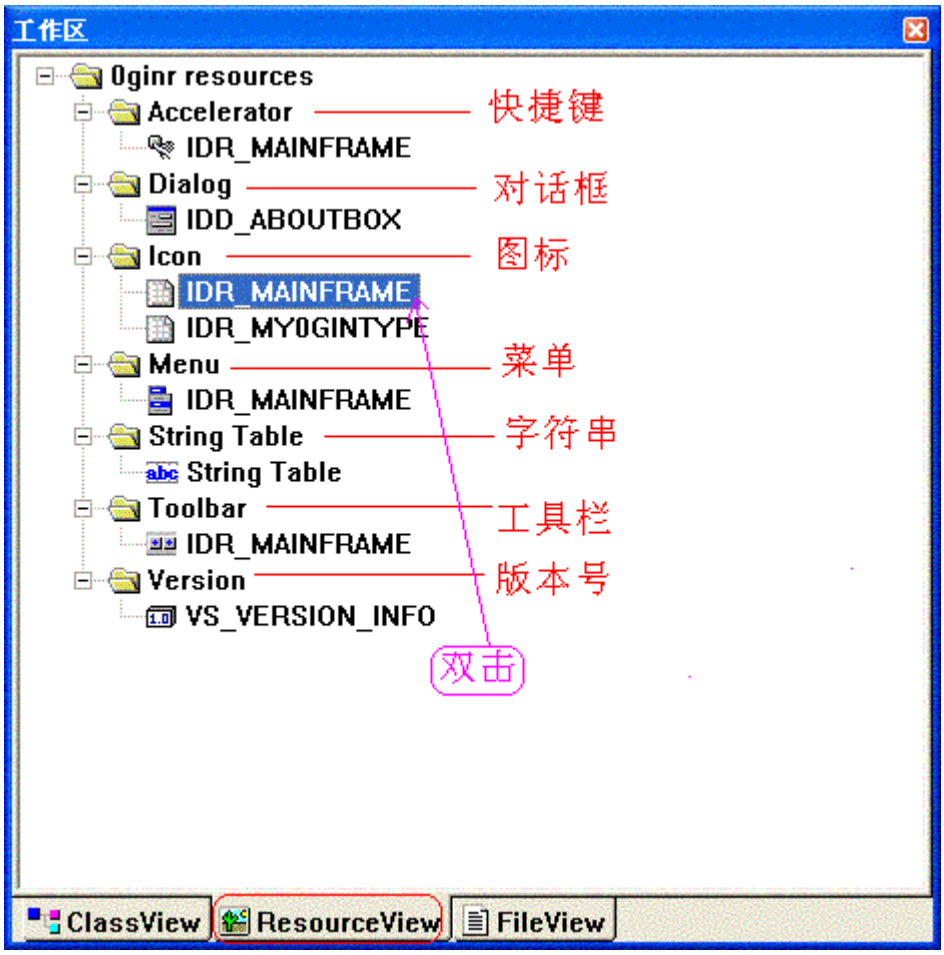



当然，我们上面表示的类其实与上面图片中的一些类没有必然的联系，类好像就是为这些窗体做了一件功能强大的外套，那我们以后再说，看这幅图有点感觉了吧，呵呵，如果说还有那么一点意思，那么再看下幅图：

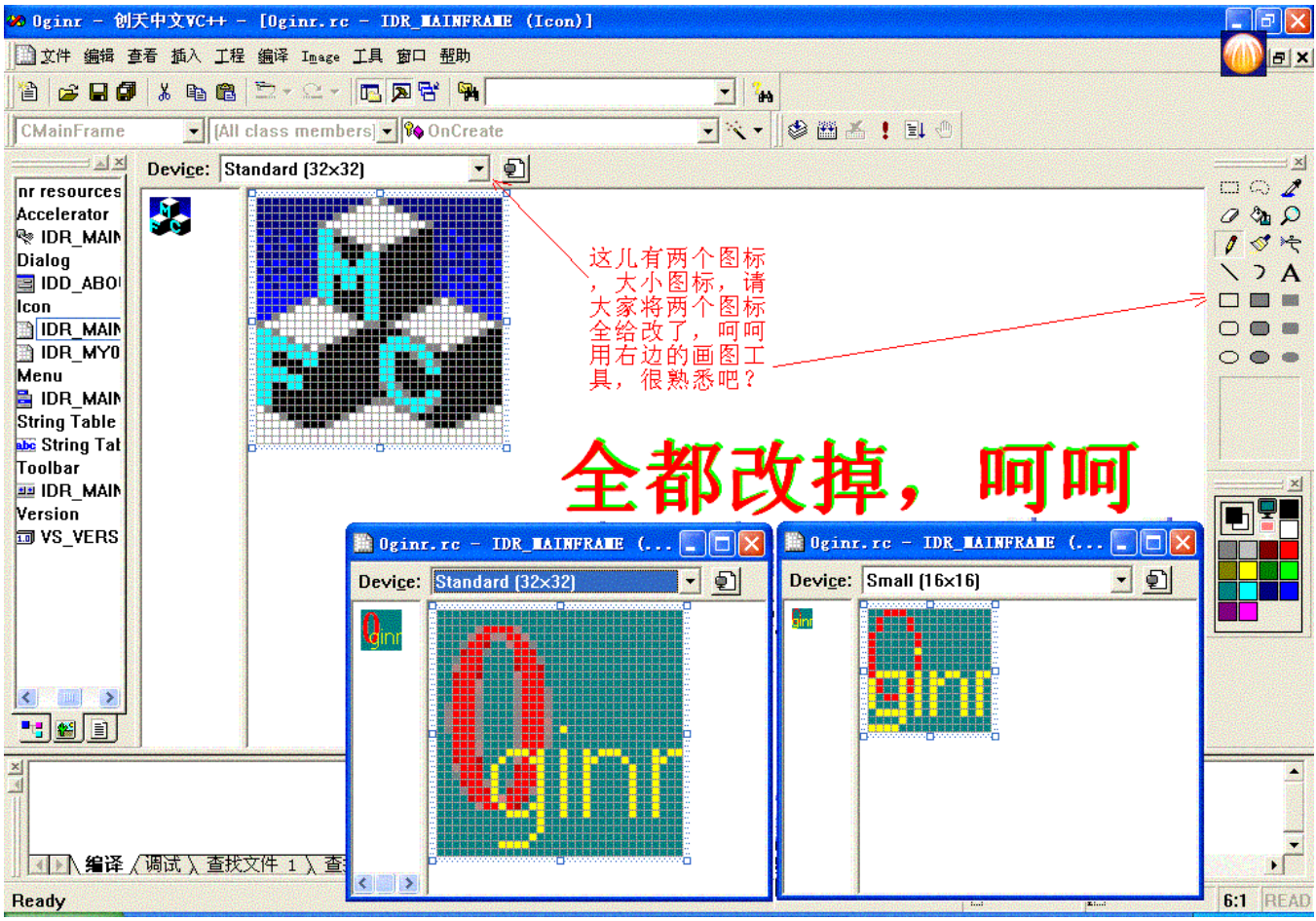



是不是果然有点像积木了？事实就是如此，我们“肢解”开来的一块一块的东东，都是可以去通过一番改造，从而满足我们自己的一些奇奇怪怪的想法。（当然这些想法，我们这节就先不详细讲解了。）

下面我们双击工作区下面中间一个选项卡“ResourceView”，这也被ＶＣ编程的人们称之为“资源选项卡”，说实话，没有这个选项卡时，我们的资源都是在程序中用　#define 去定义的，或者根据一定的规则去定义一定的结构体，而在ＶＣ系列中，编程环境帮我们将这些后台的一些东东全部在后台做了，从而使得我们做程序变得分外地简单。（当然如何简单，我们这一讲也不可能给大伙讲，等以后慢慢来吧，毕竟一口不能吃一个胖，是也不是？）




好，我们双击其中的 Icon 下的 IDR_MAINFRAME，（注意了，其他好多的资源下，都有 IDR_MAINFRAME 这样的资源名称，不要点错了，为什么都一样，我们以后再讲）




看见上面的图标了吗？全部改掉，然后再编译，运行，大家看到什么了？图标变了是也不是？

不错，如果你大小图标全改了，那么你的程序保存的图标也就变了，变得很有自己的特色了，是不？（注：建议论坛做出自己特色的图标来，大家做软件统一图标，这样是不是更标准些。）呵呵，改图标简单吧？改其他的东东其实都很简单，你可以尽情地去修改一些东东，当然，如果你的功底很深厚了，那这里几乎任何东东都可以直接修改。当然你现在就只改改图片，改一改对话框等看得见的东东就罢了，当然菜单也可以自由地改，工具栏也可以自由地改，当然这些改动并不会增加一些新的功能，因为我们还没有对自己的软件进行任何编程，如果想做系统的修改，那么就跟着牛刀一步一步地来。

好了，这一讲不着过多地讲述了，大家还记得上面我们与大家讲过怎么建立一个单文档程序吗？下面我们再建一个，不过这次大家选择“单个文档”后，先不要点“完成”。按下面步骤来：

·新建一个 MFC AppWizard(exe) 程序，程序名设定为 “MyIE”，选择一个位置，点“确定”
·在第一步（Step 1）中，选择“单个文档”
·连续点击“下一个”，到第六步（Step 6 of 6），大家看到下面有“Base class:”吗？点击下面的下拉列表，选择“CHtmlView”，也就是一个网页的类，点“完成”，再按“确定”

点开我们的类，还是那五个吧，不过这次由于第一个字母不是数字了，系统没有给我们前面加“My”（注：你看到的那个 MyIE 是我们自己的命名），这次我们双击“CMyIEView”下的“OnInitialUpdate()”，进入这个函数之中，大家是否看到其中有一句：

```
Navigate2(_T("http://www.microsoft.com/visualc/"),NULL,NULL);
```

改为：

```
Navigate2(_T("http://0ginr.com/bbs/"),NULL,NULL);
```

编译，运行，怎么样？一款我们自己论坛的“专业”浏览工具这就做好了。我们当然可以很容易地增加一些属于我们自己论坛的专业的功能，当然这些功能可能需要论坛的支持，但做一个浏览器也的确就这么简单，你想专业就往上加功能呗，你增加的一些辅助功能越多，你的浏览器就显得越专业，例如你可以增加一个菜单，名称叫做“灌水机”，然后在相应的相应函数中加上一个自动发贴一万条的函数，当然大家即使有那种能力了，也最好不要干这种事好，论坛管理本身就不是一件很容易的事，如果你用那东东与管理员对抗的话，那么管理员会累坏的，如果你的那种灌水能力太专业了，足够专业了，小心警察敲你家的门（嘻嘻）。

（当然喽，上面是与大家开个玩笑，我们学习编程的目的是为了做软件，做好事，而不是去干坏事。做程序的人需要有做程序人的基本素质，你说是吗？）

当然，这一讲，我们是大伙“管中窥豹——略见一斑”，想要系统地了解、理解我们的 MFC，那么就需要跟着牛刀一步一步地从基础走来。

[思考]

一个程序的运行（注：我们以 void main() 函数为入口的函数去思考），是不是一定会先运行 void main() 入口函数？

[提示]请大家按 C++ 的思路去考虑问题，如果不可能，说明理由，如果可以请给出程序，这个思考题的结论可能要过几讲才与大伙讨论，下一讲开始与大伙讲一些基础的理论与基本的 API 函数，那些东东是基础。

**微软积木〇六七：聚沙成塔**

当我们初窥了 MFC 的种种方便，你是不是有些动心了，巴不得立刻上手，去做一款自己的软件呢？打住，我们那只是一种初始的演练，而如果我们要真正地明白其中的道道，那就必须得一步一个脚印地走来。当然跟着牛刀这么走，没什么可怕的，呵呵，相信牛刀，没错的。

好，我们先还是来讲讲我们三种编程环境：

·第一种是控制台程序，以 main 为入口函数，运行窗口一般情况下是一个黑黑的 DOS 窗口。（Win32 Console Application）
·第二种是窗口应用程序，以 WinMain 为入口函数，运行窗口为一个真正的窗口。（Win32 Application）
·第三种是 MFC 应和程序，似乎找不到 WinMain 函数，主要程序被封装起来了，但出现的结果特舒服（MFC AppWizard(exe)）

当然，你如果没看过牛刀以前的教材，可能会有些不太明白，但上一节我们做的就是第三种应用程序，这一讲我们看上面的第一种与第二种窗口的建立方法：

先建立一个 控制台应用程序 吧，（注：记住喽，以后我们就不详细讲解建立过程了）
·打开ＶＣ，文件/新建../[工程]/Win32 Console Application
·在“工程”下面输入文件名，如 ：Test ，点确定后，再点击“完成”（建立一个空文档，当然下面几种你自己也可以点看看）。
·继续 文件/新建../[文件]/C++ Source File
·在“文件”下面添加自己的文件名称，可以与上面名称相同，当然也可以有自己的名称，如 main
·这样，我们就建立了一个 main.cpp 文件，我们可以在里面输入程序：

```
void main()
{
}
```

运行看看，如果能运行了，我们就成功了，当然有些朋友会说 main 函数不是必须有返回值吗？那要看是什么标准了，至少牛刀标准中可以没有，完全可以。当然你也可以用下面的函数

```
int main()
{
    return 1234;
}
```

呵呵，上面返回的 1234 有什么意思呢？
当然，在这种一运行就出来一个黑窗的东东上，可以运行 Windows 应用

程序吗？事实告诉我们是：可以，但必须包含相应的头文件，看程序：

```
#include <windows.h>
void  main()
{
        MessageBox(NULL,"这是一个控制台应用程序，点我干嘛？","我有些“狗拿耗子”的感觉",MB_OK);
}
```

怎么样，你运行一下程序，看到出现了一个黑黑的窗口之外，还出现了一个确认对话框，是也不是？

不错，理论上来说，我们的 DOS 程序与 WINDOWS 程序是“无界的”。（只需要加入相应的头文件就一切 OK 了。）

但要创建一个真正的窗口，那就必须要系统为我们分配一个唯一的整数，我们称之为“句柄”。

为什么要有“句柄”这种概念呢？没办法，这是因为我们的 Windows 操作系统是一个多任务操作系统。

在 DOS 操作系统中，任务是一个一个运行的，也就是说你在编辑一个文件时，不能够再开一个软件来听歌，而在 WINDOWS 系统则可以。也就是说 DOS 是一个一个软件执行完毕后，再执行其他的软件，所以在那里就根本没必要为软件分配什么空间啊，资源啊，等等这些问题，而在 WINDOWS 中，名义上是多任务执行的，但大家想如果我们的内存是线性的结构，CPU 是只有一个 IP ，那么是怎么可能同时执行好多程序呢？当然是不能，所以所谓的多任务操作系统其实也就是一个假象而已，其实还是一个一个程序执行的，而操作系统则为我们的程序分配了不同的时间片，因为电脑运行速度实在是太快，所以我们感觉好像是在同时执行多个任务。甚至现在的技术可以使得一台电脑拖多个键盘，多个显示器，多个鼠标，严然就是好多的电脑，但其实是不可能出现这多个电脑，多个任务的，那么，我们的操作系统是如何管理这些任务的呢？

那就是操作系统的一些分配原则，既然要给不同的程序分配不同的时间，不同的程序就必须有一个可以标识这个程序的东东，那个东东，我们就称之为“句柄”，如果你还不懂的话，那先跟着牛刀来做一个 Windows 程序吧：

·打开 VC ，单击 文件/新建../[工程]/Win32 Application
·选择一个文件夹，在工程下面那个框内题上名称 WinTest
·确认 完成 确定
·再来， 文件/新建../[文件]/C++ Source File

是不是和上面有点像相？不错，是建立一个 cpp 文件，cpp 文件是我们的程序文件， 文件下面输入 WinMain ，确定，我们就建立好了一个 Windows 的编程环境了。当然，我们先来看看我们要输入的入口函数：

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
	return 0;
}
```

请大家注意了，这里是一定要包含头文件的，不然编译通不过。

当然，我们看其中的内容是大大的丰富了，有四个参数，其中的第一个参数是系统为我们程序分配的实例句柄，这就是句柄了，而第二个参数是先前的一个实例句柄，听说这个参数不用了，总是 0，第三个参数是我们传递进来的参数，我们的程序是可以带参数的，例如大家可以点击“开始/运行”在相应对话框中键入： notepad.exe 1.txt ，点“确定”，看见什么了，程序说没有 1.txt 要你新建，是不？那里的参数就是从这里传递进来的。

第四个参数是程序的状态，当然这个状态是系统调动时规定的，我们没办法去更改之，但我们可以使用，下面到使用时我们再与大伙详细讲解。

好了，还来上面我们曾经做过的程序吧：

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
	MessageBox(NULL,"这才是真正的 windows 环境，呵呵","环境确认",MB_OK);
	return 0;
}
```

这个程序已经是可以完美地运行了，但大家看我们并没有用到实例句柄，是也不是？

不错，那么我们就一步一步地来制作一个窗口吧。

那么窗口是什么呢？其实就是一个“结构体”，请大家注意，我们这里先不要考虑 C++ ，我们现在其实就是 C 的编程。好，我们下面就来定义一个窗口结构，结构体的名称叫做 WNDCLASS

```
WNDCLASS ws;		//注：以下程序都在 WinMain 函数中去实现。

	ws.cbClsExtra=0;
	ws.cbWndExtra=0;
```

上面这两个结构变量至少在我们这没什么用，但必须赋值，而且一个也不能

少，赋什么呢？都赋 0 吧。

```
ws.hbrBackground=(HBRUSH)GetStockObject(GRAY_BRUSH);
ws.hCursor=LoadCursor(NULL,IDC_ARROW);
ws.hIcon=LoadIcon(NULL,IDI_APPLICATION);
```

上面三个是什么呢？第一个是给我们窗体准备了一个画刷，第二个是为我们窗体准备了一个光标，（就是我们的鼠标光标），第三个为我们窗体准入了一个图标，就是我们要显示给大家的，至于上面三个后面都有一个函数，这些函数怎么使用，我们以后会给大家详细介绍，不相关的东东我们照猫画虎就一切 OK 了，呵呵。（其实不是不想介绍，我们是想在这一讲让大家的窗体能够在自己的电脑上画出来，其他细枝末节的东东我们以后一块一块地攻破，何况还有好多知识都是需要我们去学习的。）

```
ws.hInstance=hInstance;
ws.lpfnWndProc=MyProc;
```

实例号被我们用上了，这是必须的因为操作系统只能通过这一个实例号来管理我们的程序，所以这里根本就没有必要改变，当然我们看下一个就是为我们的程序定义了一个回调函数，将函数名传递给操作系统，操作系统就会根据我们的回调函数名去调用我们编写的回调函数，从而进入消息循环了（什么是消息循环？不懂没关系，以后我们再讲，我们继续我们下面的编程过程）：

```
ws.lpszClassName="MyWindow";
ws.lpszMenuName=NULL;
ws.style=CS_HREDRAW | CS_VREDRAW;
```

上面是为我们的结构命一个名称，当然什么名称都可以，马上我们注册窗体以后，我们操作系统中就有了一个以我们这个结构体名称命名的窗体了，而下一个菜单，我们这里没有，接下来是窗体的类型，我们的类型是横向与纵向重画的。好了，下面我们就要进行窗体的注册：

```
RegisterClass(&ws);
```

不错，就是一个 API 函数，就可以进行注册了，注册过后，我们操作系统中就有了一个以 MyWindow 为名称的窗体，不错，有了，有了以后，我们就需要将这些窗口建出来，上面的赋值好像是在画图纸，后面的工作就像盖房子了，我们用注释的办法与大家来分配盖房的乐趣。

```
HWND hwnd;          //这是我们自己建的一个窗口句柄（注，与实例句柄两码事，我们以后详细说明）
hwnd=CreateWindow(
        "MyWindow",         //窗体类型名称，必须与上面一致
        "牛刀制作的窗口",//将在标题栏中显示的名称
        WS_OVERLAPPEDWINDOW,//窗口的类型
        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,//这四个参数是窗口的位置与大小，这里是个宏，可以用具体数字
```

NULL,NULL,hInstance,NULL);//这里是父窗体啊，菜单啊，实例句柄啊，等等，除了实例句柄外，我们程序中统统地没有。

上面一个函数好像是我们这个程序中最长的函数了，当然，就这一个函数，就可以产生出我们需要的窗口。返回一个窗口句柄，又来了，那么这个句柄是什么呢？这个句柄代表着我们的窗口，以后我们要在窗口上干事，就少不了这个句柄了，详细的以后我们再讲吧，接下来干什么？显示与刷新我们的窗口。

```
ShowWindow(hwnd,SW_SHOWNORMAL);
UpdateWindow(hwnd);
```

后面一个函数竟然可以不要，呵呵，但一般我们都留着。

好了，窗口设置好了，接下来，我们就需要进入了我们的消息循环：

```
MSG msg;
while(GetMessage(&msg,NULL,0,0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

消息循环就这么简单，但绝对不能没用，消息每循环一次，就将消息发送给我们在类中定义那个函数，什么函数呢？大家还记得上面我们定义的：
ws.lpfnWndProc=MyProc;

不错，就是要将消息发送给这个函数，于是，我们就需要建立这个函数，在 WinMain 函数之前建立（当然也可以在函数前面先建立一个函数原型，而真正的函数放后面，我们一般都这样做）

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);
```

请大家注意了，函数名可以改，但必须与自己定义的名称相同，但参数不可以改，什么原因我们以后慢慢给大家讲解。好了，下面我们来这个函数：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        return 0;
}
```

如果你那样定义的话，你几乎看不到窗口，为什么呢？因为我们的程序退出了（因为我们返回了 0）。

那么不想让其退出，就必须处理消息循环，有一个默认的处理消息循环的函数，我们返回就一切 OK 了。

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
	return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
```

上面的程序就可以一切正常的运行了。至于如何进行消息循环，我们下一节与大家专题地开一讲，这一讲就到些结束了，下面是代码：

```
#include <windows.h>
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
	WNDCLASS ws;

	ws.cbClsExtra=0;
	ws.cbWndExtra=0;

	ws.hbrBackground=(HBRUSH)GetStockObject(GRAY_BRUSH);
	ws.hCursor=LoadCursor(NULL,IDC_ARROW);
	ws.hIcon=LoadIcon(NULL,IDI_APPLICATION);

	ws.hInstance=hInstance;
	ws.lpfnWndProc=MyProc;

	ws.lpszClassName="MyWindow";
	ws.lpszMenuName=NULL;
	ws.style=CS_HREDRAW | CS_VREDRAW;

	RegisterClass(&ws);

	HWND hwnd;
	hwnd=CreateWindow("MyWindow","牛刀制作的窗口",
		WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,
		NULL,NULL,hInstance,NULL);


	ShowWindow(hwnd,SW_SHOWNORMAL);
	UpdateWindow(hwnd);

	MSG msg;
	while(GetMessage(&msg,NULL,0,0))
```

```
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return 0;
}

LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
```

注：上面的代码程序关不了，关闭后，大家可以在资源管理器中还能看到相应的程序在运行，到资源管理器中去关了就一切 OK 了，至于什么原因，我们下一讲再详细与大伙讲解。

**微软积木〇六八：消息循环**

如果你将上一讲的代码敲进电脑（注意，第一次请大家一句一句地敲，你会发现一些奇特的地方，例如，你写结构体的成员函数时，系统自动给你提示，一空格就上机了，特别地方便）。那么，你就可以在你的电脑上呼叫出一个窗体来，那是麻雀虽小，五脏俱全啊，当然，下面还有一个东东，大家可能不太明白：

```
MSG msg;
while(GetMessage(&msg,NULL,0,0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

为什么这里就能构成消息循环了呢？这何况这三个函数我们也没看见啊，那我们先别着急，我们先从一段 DOS 的程序谈起，建立一个控制台应用程序（即以 main 为入口的应用程序）。

我们做一个简单的系统，系统上有十个菜单，分别画一些图形，看程序：

```
#include <stdio.h>
void draw()
{
        printf("\tDOS 模拟画图函数\n");
        printf("\t1--画一个矩形  OK\n");
        printf("\t2--画一个三角形 OK\n");
        printf("\t3--画一个圆形  OK\n");
        printf("\t4--画一个星形  OK\n");
        printf("\t5--画一个菱形      ERR\n");
        printf("\t6--画一个实心三角形 ERR\n");
        printf("\t7--画一个实心圆形  ERR\n");
        printf("\t8--画一个实心星形  ERR\n");
        printf("\t9--画一个实心菱形  ERR\n");
        printf("\t0--退出        \n");
}
void main()
{
        int i;
        printf("\n\n");
        draw();
        while(i=getchar()-'0')
        {
                switch(i)
                {
                case 0:
                        printf("感谢你的使用，退出了\n");//注：这句话
```

可能显示不出来，呵呵。

```
                        break;
                case 1:
                        printf("□\n");
                        break;
                case 2:
                        printf("△\n");
                        break;
                case 3:
                        printf("○\n");
                        break;
                case 4:
                        printf("☆\n");
                        break;
                        //注，不继续向下做程序了，其他类似
                default:
                        printf("错误,其他代码还没有编写,自行添加\n");
                }
                if(i!=-38)draw();
        }
}
```

当然，这个程序大家可以自行去完善，当然，似乎这与我们的消息循环有些扯不上边，好，下面我们将一些程序提炼出来，做成函数：

```
i=getchar()-'0';
```

这一句，我们怎么提炼？再定义一个函数呗，看程序：

```
int GetMessage()
{
        int a;
        a=getchar()-'0';
        if(a==-38)a=getchar()-'0';
        return a;
}
```

敢这么用吗？呵呵，其实是意思到了就行了呗。（注，用了上面的程序后，后面的 if(i!=-38) 可以省略，因为我们让系统确定时，就有一个回车，而回车也算着是一个字符，所以多了一个）。

再将 switch(i) 提炼出来：

```
void DefDraw(int i)
{
                switch(i)
                {
                        ……//以上代码复制过来就行了,当然你可以将程序做全，由于字数限制，为了为大家多做些解释，就只有少做些代码了。
                }
}
```

这么一来，怎么样？有好处了吗？是不，有好处了，甚至牛刀还有一些想法，先看程序吧，我们的程序变成了什么了？

```
while(i=GetMessage())
{
        DefDraw(i);
        draw();
}
```

不错，一个很简单的循环，其实也就是消息循环的原型，但我们 Windows 程序的消息循环似乎不是直接调用的，不错，那里是系统自动调用的，根据我们提供的函数名：

ws.lpfnWndProc=MyProc;

当然，那是一个函数指针，我们放到以后的指针部分给大家详细讲，我们先继续我们上面的消息循环吧，我们假如要修改默认的消息循环，我们原来输入 1 是画一个矩形，而我们想输入 1 画一个菱形怎么办呢？很简单：

```
while(i=GetMessage())
{
        switch(i)
        {
        case 1:
                printf("◇\n");
                break;
        default:
                DefDraw(i);
                break;
        }
        draw();
}
```

是不是感觉有点像抢劫的？此山是我开，此路是我栽，要想过此树，留下买路财……

好像不对，应该是　此山是我栽……

还不对，是：

此山是我开，此树是我栽，要想过此路，留下买路财，牙崩半个“不”字，你来看——红刀子进，白刀子出，我是管杀不管埋

嘘！！！！　看样牛刀想当山贼还要回去练几年，但事实上上面的　1　的确被我们栏截了。

也就是说，是 1 就传不进默认函数了，传进默认函数的就不可能是 1 ，于是，系统为我们提供的默认的东东就可以自由地隐藏起来了，只提供一个我们看得见的默认函数：

DefWindowProc(hwnd,uMsg,wParam,lParam);

于是，我们想要实现我们自己的编程梦想那就只有去当山贼了，没办来，还是来个　switch　吧，那东西的确方便啊。

对了，上面的像 0 不就表示退出吗？于是，我们就可以用 #define 宏给定义一个特殊的变量，看：

```
#define DOS_EXIT 0
```

于是，后面的程序就可以变成了：

```
case DOS_EXIT:
    ......
    break;
```

事实上就是如此，我们先来看一段代码：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        switch(uMsg)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
```

大家注意，其中的类型不能变，但参数名称都是可以变的，看：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        switch(message)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

可能你觉得没有什么实在的意义，但事实会告诉你，我们是在做程序，不是做序在做我们，我们就必须知道我们哪里是可以动的，哪里是不可以动的，只有那样，我们做起程序来才会游刃有余。

为了让大家相信上面的 WM_DESTROY 是一个数字，牛刀找出定义给大伙看一看：

```
#define WM_NULL                 0x0000
#define WM_CREATE               0x0001
```

```
#define WM_DESTROY                    0x0002
#define WM_MOVE                       0x0003
#define WM_SIZE                       0x0005
```

看见了吧，都是十六进制的整数，那么我们直接使用整数效果其实是一样的：（就看我们怎么定义的了）

```
        case 0x0002:
                PostQuitMessage(0);
                break;
```

但问大家一个简单的问题，大家看到上面的代码，会知道这个 0x0002 是什么消息吗？所以说，还是“宏”好用，认了吧。

上面的 WM_DESTROY 是销毁窗口，请大家注意，销毁窗口并不代表着跳出循环。也就是说，上面的：

GetMessage(&msg,NULL,0,0)

返回值是仅仅当消息队列中有 WM_EXIT 消息发送过来才会退出循环，也就是说退出程序，而 PostQuitMessage(0); 正是可以发送此消息的函数，所以在程序中，我们遇到这种不仁不义不忠不孝的，那我们唯一的办法只有拦截了（我们是义匪啊，呵呵）。

其实系统是可以默认地为我们做这种事的，但是却没有做，为什么这为我们保存文件留下了一次机会。因为当窗体被销毁后，就发送这最后的一条消息，所以这一条消息是我们想做好各种后台工作的唯一机会了，因为当我们接到这一消息时，我们的窗口已经没有了。或者说，这个消息就是让我们有一次吃“后悔药”的机会，而还有一种消息， WM_CLOSE 这个消息则是窗体还没销毁，但你已经准备关闭窗体时，给你提供了一次的可以不退出程序的机会，当然，那时候，窗体还没有消失，所以如果不销毁窗口的话，我们还可以继续运行程序，我们看程序吧：

```
        case WM_CLOSE:
                if(IDOK==MessageBox(hwnd,"真的想退出程序吗？","退出程序就没机会了",MB_YESNO))
                {
                        DestroyWindow(hwnd);
                }
                break;
```

当然，DestroyWindow 函数就是发送刚才那个 WM_DESTROY 消息的函数。如果你点是，则发送那个消息，如果你不发送，则退出这一次消息循环，什么都不做。

其实我们的 Windows 程序就是这么回事，一回一回地在那转圈子，等消息，当消息进来了，就进行处理，否则就在那干等，从这一点上来说，如果同一程序，DOS 程序也能做，WINDOWS 程序也能做，那么必然是 DOS 程序的效率要高得多。

虽然是这样，我们还是很喜欢 Windows 程序，因为 Window 程序为我们在后台做的事情是太多太多了。以至于我们做一个程序那就只是响应几个消息而已。那么我们的程序能处理哪些消息呢？

·用户消息，当我们的鼠标在屏幕上划过，我们按一下键盘上的某个键等等，就会产生一个相应的消息。
·系统消息，当我们系统的计时器到时间了，当我们的窗口重新刷新了，当我们的电脑突然出现一些一特殊的情况了。
·自定义消息，用户自己定义的一些消息，做程序时有什么特殊的应用了，可以自己去定义一些消息，也可以让系统去响应，效果一样的。

如果牛刀的定义与书本上有出入，以牛刀为准，如果要考试，以书本为准。

说到消息，不得不说说两个参数： LPARAM WPARAM 这两个消息其实在 16 位的时代，一个是十六位，一个是三十二位的，以便处理两种不同长度的参数，但现在是三十二位机了，这两个都是三十二位了，所以也就没有以前那种长短大小的概念了，但名称却留下来了，我们知道就行，这两个参数是随着消息的不同而就义了不同的功能，当我们发送一个消息时，需要传进的附加参数就要通过这两个参数传递进来，例如我们点击鼠标时，鼠标在屏幕上的位置，如果我们按下键盘上的按键时，我们键盘上按下的字符等等，都是通过这两个参数传递进来的。看程序：（注：牛刀每给一次代码，都是可以加到你的程序中去运行与测试的，请大家别忘上机机演练。）

```
case WM_LBUTTONDOWN:
        int x,y;
        char ch[20];
        x=LOWORD(lParam);
        y=HIWORD(lParam);
        sprintf(ch,"x=%d,y=%d",x,y);
        MessageBox(hwnd,ch,"鼠标点击测试",MB_OK);
        break;
```

请大家注意，由于上面要用到一些字符处理的函数，所以请包含头文件 stdio.h 。（会吧？ #include <stdio.h>）

上面用到了两个宏 HIWORD LOWORD 其实就是取出 lParam 中的低字与高字，从上面的代码我们也会知道，其实我们 lParam 中字包含的是鼠标当前的 x 坐标，而高字包含的是当前的 y 坐标。

下面的 sprintf 是一个标准的 C 语言的函数，我们大家知道有一个函数是 printf 吧，相信大家一定很熟悉，当然 sprintf 与 printf 唯一不同的是 sprintf 将字符格式化在上面的第一字符串中，唯一而已，大家用几次，习惯了就知道怎么使用了。

当然，我们再来一个消息，来获得我们的键盘消息（注意，我们的消息的名称都是有含义的）

```
    case WM_CHAR:
            char str[20];
            sprintf(str,"你输入的按键为:%c",wParam);
            MessageBox(hwnd,str,"按键消息",MB_OK);
            break;
```

当然，你可以一直纳闷：

```
    while(GetMessage(&msg,NULL,0,0))
    {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
    }
```

循环中为什么有两个函数呢？其中上一个是解释消息，下面一个是消息处理，为什么要解释就是像上面的 WM_CHAR 其实这一个消息其实原来是不存在的，原来消息中有：

WM_KEYDOWN

WM_KEYUP

一旦遇到这两个消息，解释消息就会将解其解释为： WM_CHAR 并将此消息插入消息循环，还有我们鼠标的双击消息，甚至单击消息，也是通过解释消息来产生的。甚至有一些微软的软件中会用到鼠标的三击，那更是需要解释，没办法，人家厉害啊，呵呵。

当然我们也可以在一些消息中去模拟出这些消息，但同时处理多个消息总是不如只处理一个消息来得方便，何况 WM_KEYDOWN 给的还是扫描码，而不是 ASCII 码，来得总是没有我们的 WM_CHAR 来得方便啊。

当然，我们不得不向大家介绍一个我们建立一个窗口前，消息发送来的第一个消息 WM_CREATE 这个消息会报告窗体的长度宽度一些基本的信息（当然表示的是客户区，什么叫客户区，怎么在窗体上做一些我们想做的事情，我们下一讲再与大家详细介绍）。

对了，牛刀重写了教程，还是一个一个字符地重新敲进去的，但还没有完成，先给大家看个大概，呵呵，希望大家喜欢，以后有新的添加再重新发放给大家。

**微软积木〇六九：神笔在手**

还记得《神笔马良》的故事吧，那个笔是作者虚构的，我们这儿的——咳咳，也是虚构的。

先来回顾我们以前的一些内容吧，可能会有一些编程老手会嫌牛刀啰唆，不过没办法，既然讲就要将一些东东讲透，讲明白，不能讲得让人囫囵吞枣，是也不是？牛刀知道这是第二次讲怎样做一个窗体了，但目的与前一次不同，以前是让大家从 DOS 向 WINDOWS 转型，而这次是要让大家深入。

好了，不费话了，我们进入 WinMain 函数后，我们得到了第一个句柄，实例句柄，而中途中 CreateWindow 创建了一个窗口句柄，而如今我们还需要好多好多的句柄，这些句柄就必须我们自己去创建了，事实就是如此，当一个窗口（既然说到了窗口，我们就需要为我们的窗口装上玻璃是不？不过我们这次使用的不是透明的玻璃，而是一块“毛玻璃”，那为什么不用透明的呢？可以啊，你试试将 WinMain 函数中的 ws.hbrBackground=(HBRUSH)GetStockObject(GRAY_BRUSH); 改成 ws.hbrBackground=(HBRUSH)GetStockObject(NULL_BRUSH); 试试，透明的的确是透明的了，就怕运行起来你不太习惯，呵呵）创建时，我们可以通过窗口句柄 hwnd 对窗口进行操作，不过窗口中那空空的一大片东东，我们怎么对其进行操作呢？

为了对其进行操作，我们就也需要得到那一块区域的句柄，那一个句柄我们通常称之为“设备描述表”，牛刀称之为“画布”或者是“玻璃”，但绝对是“毛玻璃”。

当然我们不是为了复古，虽然透明的好是好，采光好，但我们的目的是在上面做一些东东啊，牛刀更喜欢将其看成是中国古代糊上窗户的“纸”，当然，把她当作纸感觉更贴切，你说呢？因为我们马上还要有“笔”，还要有“画刷”等等，既然都是画画的东西，为什么不能将其看成是“纸”呢？

好了，我们定义这个纸时，通常用的是句柄 HDC 如，我们定义了一张纸：

```
HDC hdc;
```

当然，我们还想有支笔，大家可不要小看这支笔啊，可“神”了，笔的句柄为 HPEN ，当然喽，画刷的句柄为 HBRUSH 。

对了，说到画刷，大家还记得我们在定义窗口的结构体时有一个赋值：ws.hbrBackground=(HBRUSH)GetStockObject(GRAY_BRUSH);

想起来了吧？那后面就是用 GetStockObject 取得一个系统默认的画刷来画我们这张纸。
GetStockObject 函数就是将系统为我们准备的几种东东送给我们用，系统给我

们装备了几种东东，

WHITE_BRUSH
LTGRAY_BRUSH
GRAY_BRUSH
DKGRAY_BRUSH
BLACK_BRUSH
NULL_BRUSH
HOLLOW_BRUSH
WHITE_PEN
BLACK_PEN
NULL_PEN

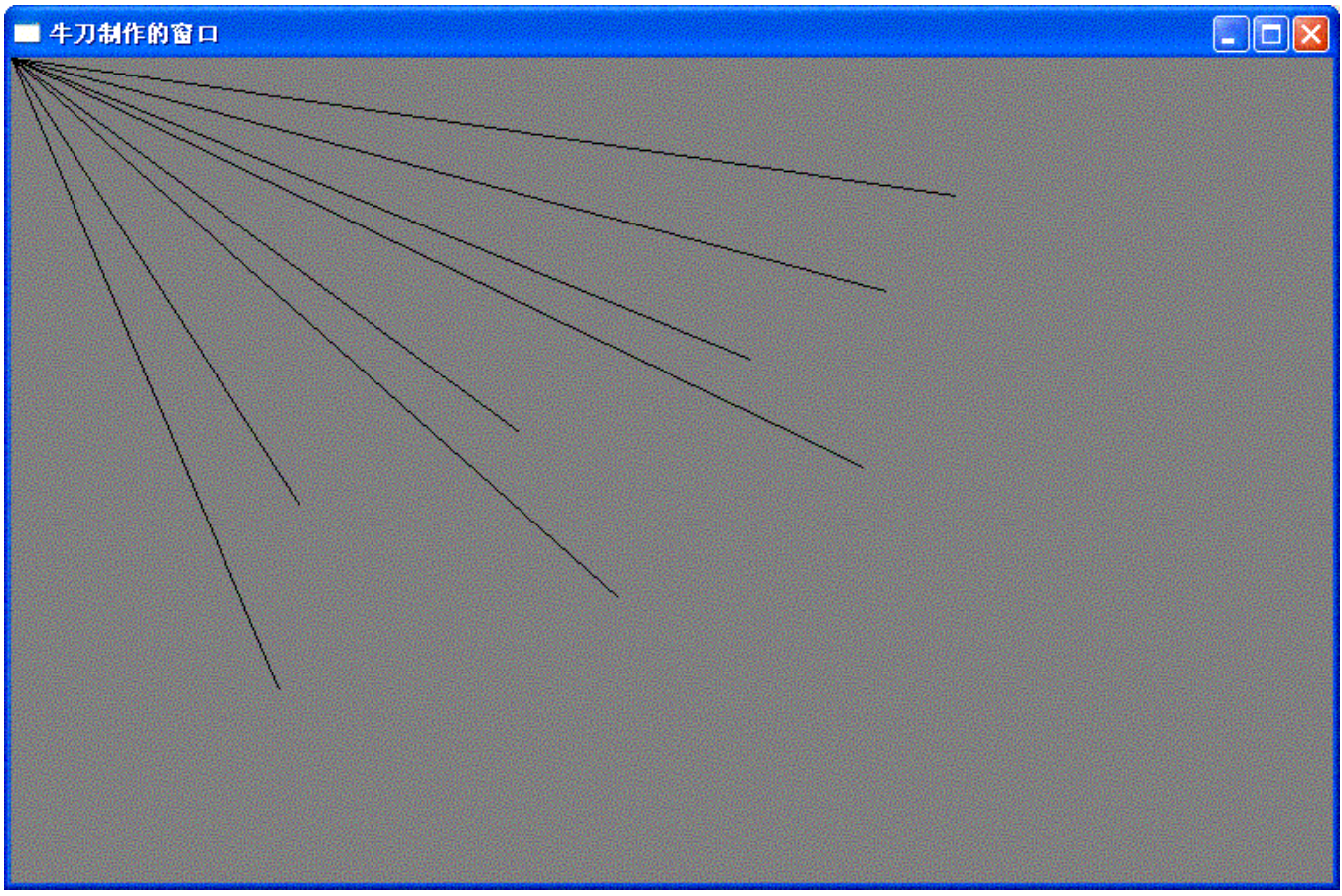
七把刷子三支笔，当然还有其他的一些东东，那么我们想办法去用用吧，怎么用呢？当然使用一些软件为我们提供的 API 函数喽，好了，我们上节给大家讲过一个鼠标按下的事件大家还记得吧？

```
case WM_LBUTTONDOWN:
        int x,y;
        char ch[20];
        x=LOWORD(lParam);
        y=HIWORD(lParam);
        sprintf(ch,"x=%d,y=%d",x,y);
        MessageBox(hwnd,ch,"鼠标点击测试",MB_OK);
        break;
```

我们知道了坐标，我们用 MessageBox 消息框给弹出当前的坐标来了，那我们这次从我们纸的左上角下你点的位置来画一条直线：

```
case WM_LBUTTONDOWN:
        int x,y;
        HDC hdc;
        x=LOWORD(lParam);
        y=HIWORD(lParam);
        hdc=GetDC(hwnd);
                LineTo(hdc,x,y);
        ReleaseDC(hwnd,hdc);
        break;
```

看图：




什么？就这么简单？不错，就是这么简单。下面我们看看在这里我们是怎么得到 HDC 的？很简单，一个函数 GetDC 就返回了我们当前窗口的那张纸。当然，这张纸用过后，一定要删去。原因嘛，很简单，有借有还，再借不难。

我们和大家说过，在 WINDOWS 中句柄是一个十六位的值，也就是说系统为我们最多分配 65536 也就是说有六万多个句柄，如果你拿了一个句柄不还，那么系统就一直不能使用这个句柄，下次用时就需要再次为你分配一个，这样，如果你的程序向系统要的句柄就会越来越多，系统运行速度会明显变慢，这就是大家玩大型游戏特别是网络游戏时，有时候运行时间长了，就会速度变得特别特别地慢？为什么？因为做程序的厉害的程序员们肯定是申请了什么句柄没有及时归还。（什么，那些老手还会出现这种情况？啊，越是老手越容易忘记这种细节末节。可以说，学编程之初，如果不养成良好的习惯的话，那么这种坏习惯会一直保留下去，所以请大家在写程序时， GetDC 与 ReleaseDC 一般同时写出，然后有什么代码在这两句代码中间去写。这是一个好习惯）。

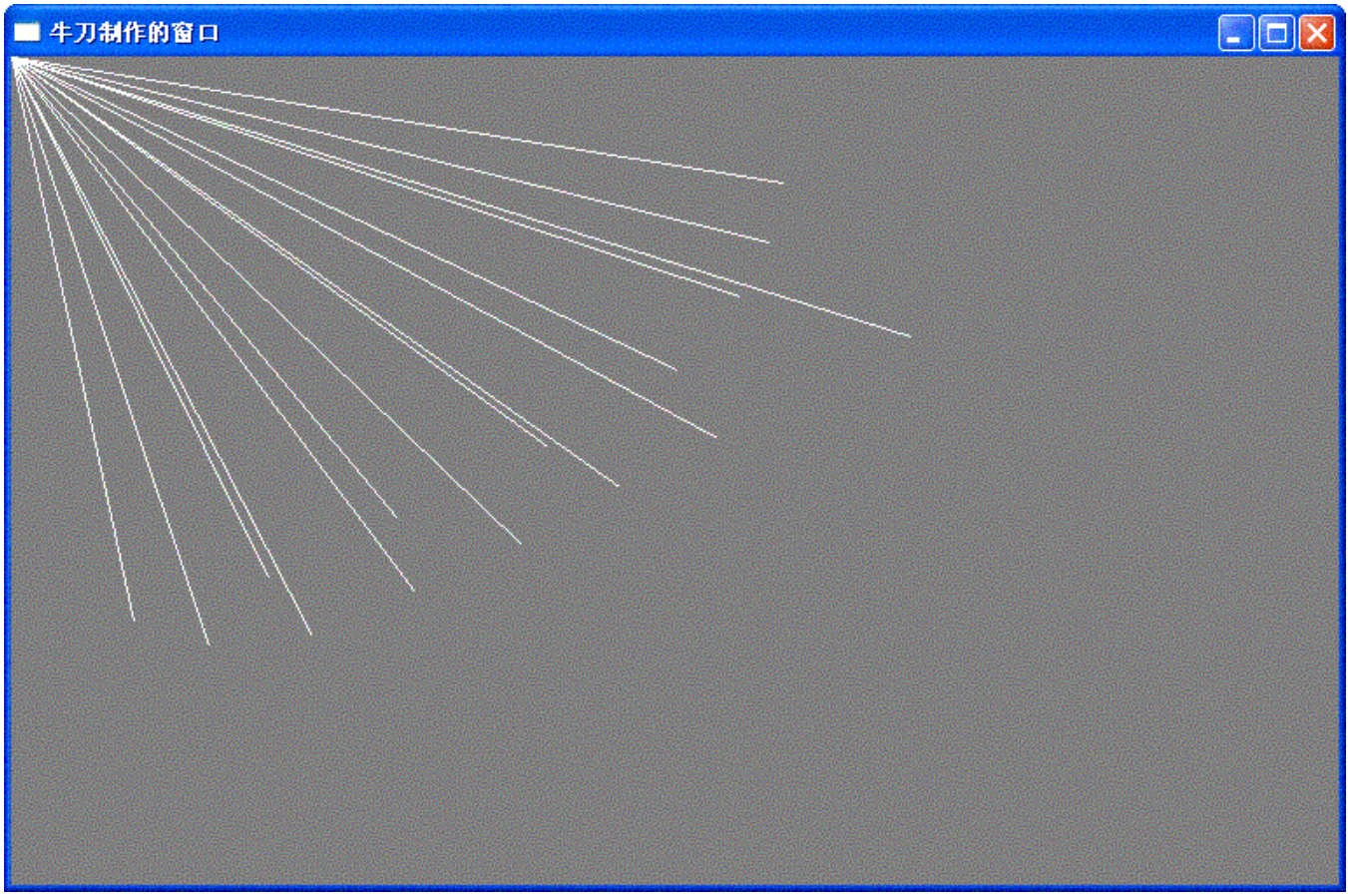
当然，如果你想归还，只要用函数 ReleaseDC 就一切 OK 了，那么如果你不归还会出现什么情况？大家删去看看，删掉后，程序没有任何不适，还是运行的一样的棒。也正因为如此，我们就一定要强调大家要有借有还，在学习编程的初期就养成一个良好的习惯。

```
hdc=GetDC(hwnd);
LineTo(hdc,x,y);
ReleaseDC(hwnd,hdc);
```

好了，我们在这上面再做文章，取到句柄，释放句柄，而在这其中画了一个线，用什么画的？用笔呗。好，既然是用笔，那么我们就可以去换支笔，怎么换？很简单：

```
hdc=GetDC(hwnd);
    HPEN pen;
    pen=(HPEN)GetStockObject(WHITE_PEN);
    SelectObject(hdc,pen);
    LineTo(hdc,x,y);
    DeleteObject(pen);
ReleaseDC(hwnd,hdc);
```

好了，换了支白色的画笔，看图：




当然，上面的两句可以变成:

```
SelectObject(hdc,(HPEN)GetStockObject(WHITE_PEN));
```

于是，我们的程序变成了：

```
hdc=GetDC(hwnd);
    SelectObject(hdc,(HPEN)GetStockObject(WHITE_PEN));
    LineTo(hdc,x,y);
ReleaseDC(hwnd,hdc);
```

简单多了是吧？

那么上面的 LineTo 是什么呢？画线呗，呵呵，但为什么一个点就可以从左上角去画一条线？我们不是说两点定一线吗？是的，我们屏幕上有一个默认的点，这个默认的点在屏幕的右上方，所以我们每次画个点就从右上角画下去，这

个默认的点，我们可以使用 MoveToEx 去改变之，当然， MoveToEx 有四个参数，不知扩展什么用的，我们一般将第四个参数设置 NULL，看程序：

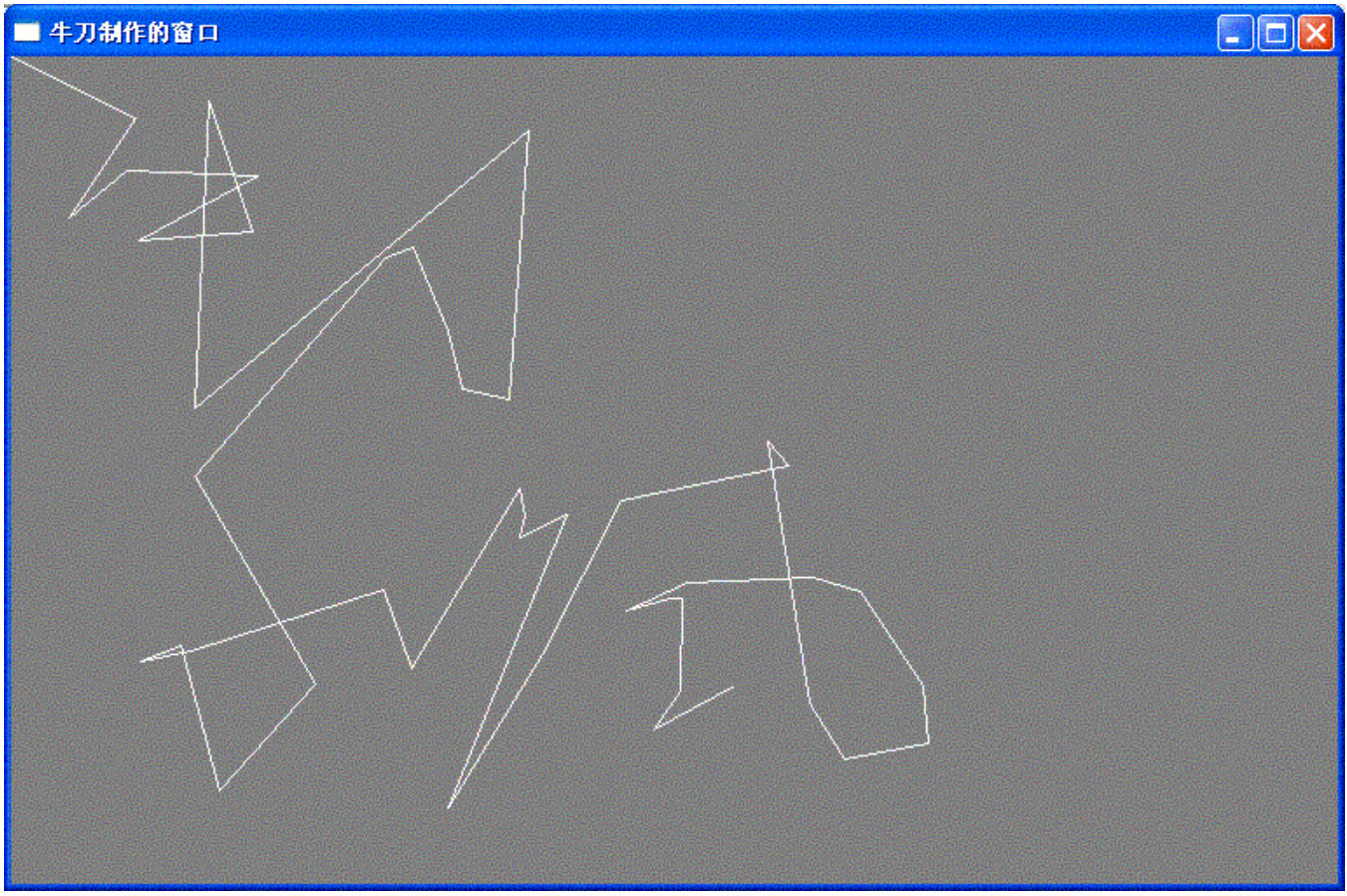
```
            hdc=GetDC(hwnd);
                SelectObject(hdc,(HPEN)GetStockObject(WHITE_PEN));
                LineTo(hdc,x,y);
                MoveToEx(hdc,x,y,NULL);
            ReleaseDC(hwnd,hdc);
```

再点几个点试试：

好像没什么变化啊，原来，每次进入了一次消息循环，这个默认点就复位了，不太死心，将 x,y 设为静态变量，俺就不信，看程序：

```
    case WM_LBUTTONDOWN:
            static int x,y;
            HDC hdc;
            hdc=GetDC(hwnd);
                SelectObject(hdc,(HPEN)GetStockObject(WHITE_PEN));
                MoveToEx(hdc,x,y,NULL);
                x=LOWORD(lParam);
                y=HIWORD(lParam);
                LineTo(hdc,x,y);
            ReleaseDC(hwnd,hdc);
            break;
```

好，看图：



是不是有种要种一个画图软件的欲望了？呵呵。
好吧，想做就做吧，那我们怎么做呢？我们当然需要有一个静态的 x,y 来放我们的当前坐标，我们还需要用一个静态的标志，当然是布尔型的就行，表示当前需要不需要画图。也就是我们的鼠标是不是按下。

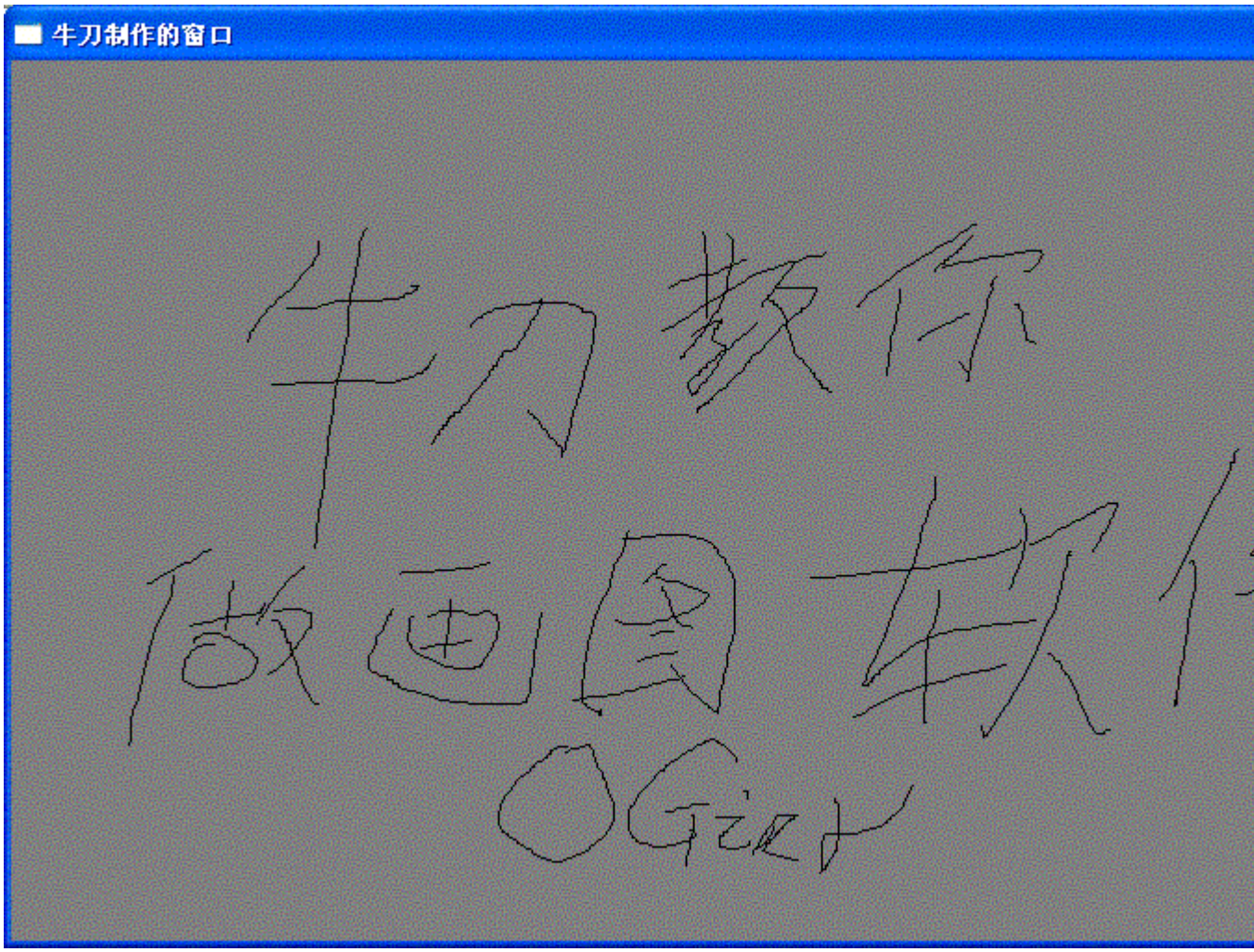
我们要知道鼠标的几种情况，也就是需要知道鼠标按键是不是被按下，是不是被抬起，如果被按下了，就设标志为真，否则为假，如果为真，则需要在移动时画线。

牛刀说得还够明白么？不管了，思路是这样，牛刀将用详细的注释来向大家解释这一切，请大家看程序：

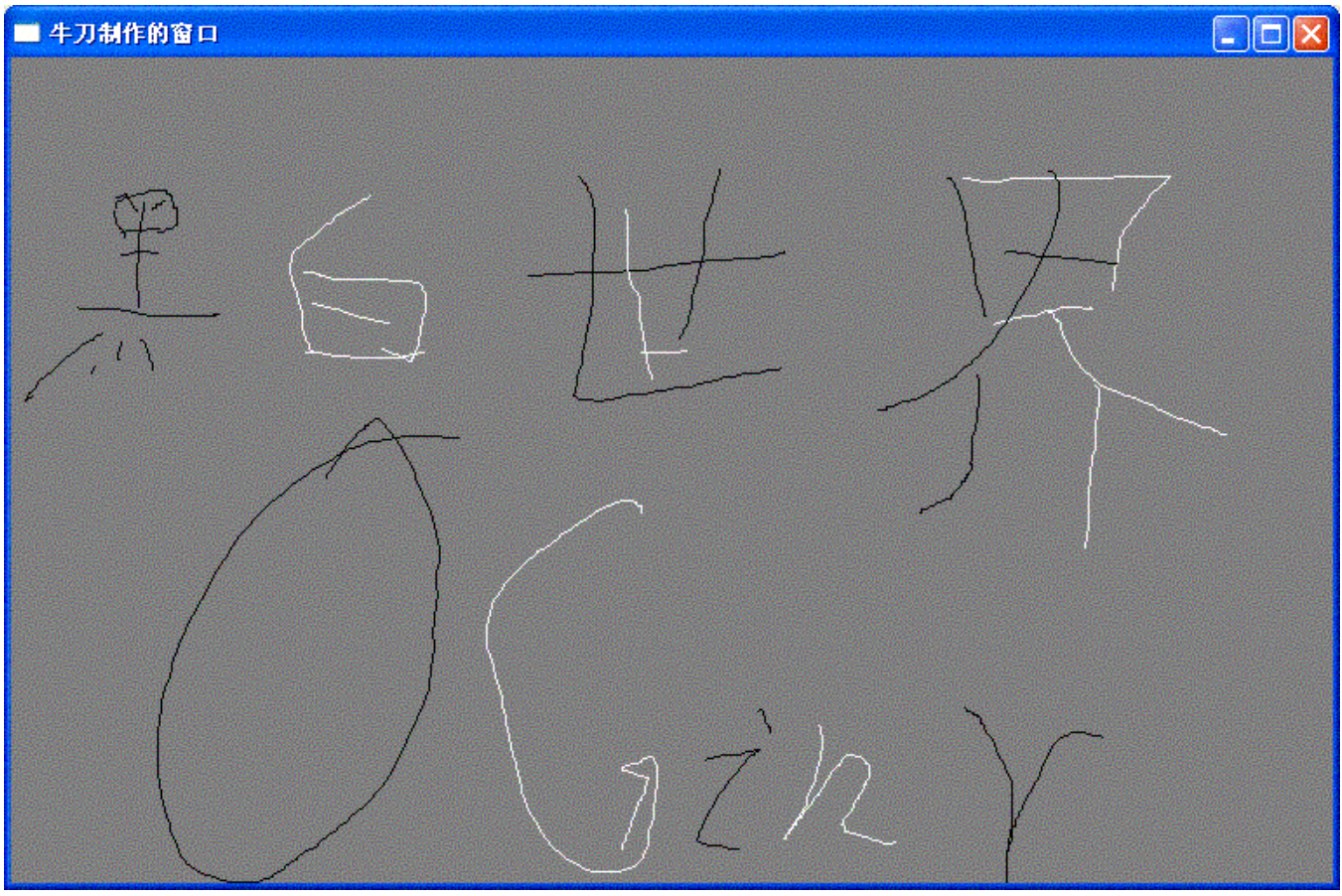
```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        static int x,y;
        static bool fla=false;
        switch(message)
        {
        case WM_LBUTTONDOWN:          //按下鼠标左键，想画图，将坐标设为当前值，标志设为真
                x=LOWORD(lParam);
                y=HIWORD(lParam);
                fla=true;
                break;
        case WM_LBUTTONUP:          //抬起鼠标左键，不画图了，标志设为假
                fla=false;
                break;
        case WM_MOUSEMOVE:          //移动鼠标，如果标志为真则画图
                if(fla)
                {
                        hdc=GetDC(hwnd);          //得到画纸
                                MoveToEx(hdc,x,y,NULL);
                                x=LOWORD(lParam);
                                y=HIWORD(lParam);
                                LineTo(hdc,x,y);
                        ReleaseDC(hwnd,hdc);          //删掉画纸，或者解释为“贴上玻璃”可能更贴切，呵呵
                }
                break;
        case WM_DESTROY:                  //关闭窗口
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
```

}

呵呵，有点感觉了：




用鼠标写字真难写，呵呵，写得不好，见笑了，事实上就是这样，画图软件中的一些代码也就不过如此。当然，你也可以再用一个静态变量去标志是鼠标左键按下还是右键按下，左键按下用黑笔画，右键按下用白笔画，拦截两个新的消息 WM_RBUTTONDOWN WM_RBUTTONUP 。牛刀就不给程序了，大家自己去构思去设置，看图：

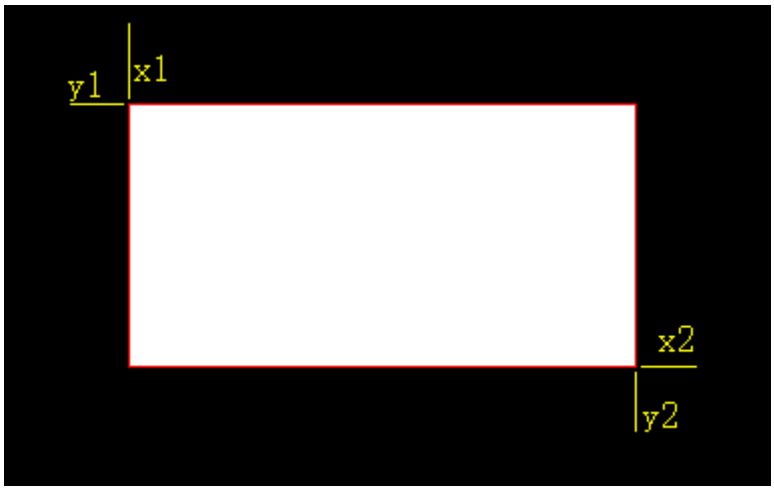



当然，想要做真正的画图软件，我们还需要画矩形，画圆，画一些其他类型的线，下面我们就来看一看一些图形的函数，

·画矩形：

需要两个点，左上角点与右下角点：

Rectangle(hdc,x1,y1,x2,y2);




以上的函数可以使用下面的程序测试：(下面的其他的形状也可以使用这个程序，牛刀不给源程序了，大家自行修改。

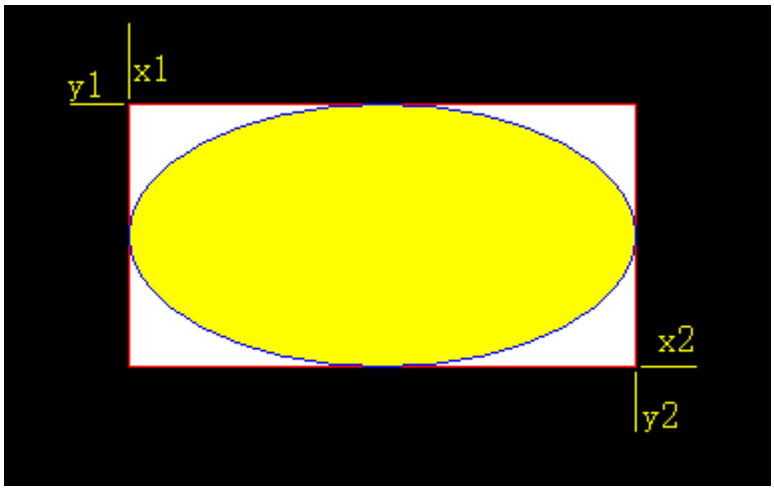
```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
```

```
{
        HDC hdc;
        static int x1,y1,x2,y2;
        switch(message)
        {
        case WM_LBUTTONDOWN:
                x1=LOWORD(lParam);
                y1=HIWORD(lParam);
                break;
        case WM_LBUTTONUP:
                x2=LOWORD(lParam);
                y2=HIWORD(lParam);
                hdc=GetDC(hwnd);
                        Rectangle(hdc,x1,y1,x2,y2);
                ReleaseDC(hwnd,hdc);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

·画椭圆

Ellipse(hdc,x1,y1,x2,y2);

椭圆相当于一个矩形的内切椭圆，看图，程序自己修改上面的测试程序即可。用鼠标在屏幕上拖动，然后放开就一切 OK




·圆角矩形

RoundRect(hdc,x1,y1,x2,y2,x3,y3);

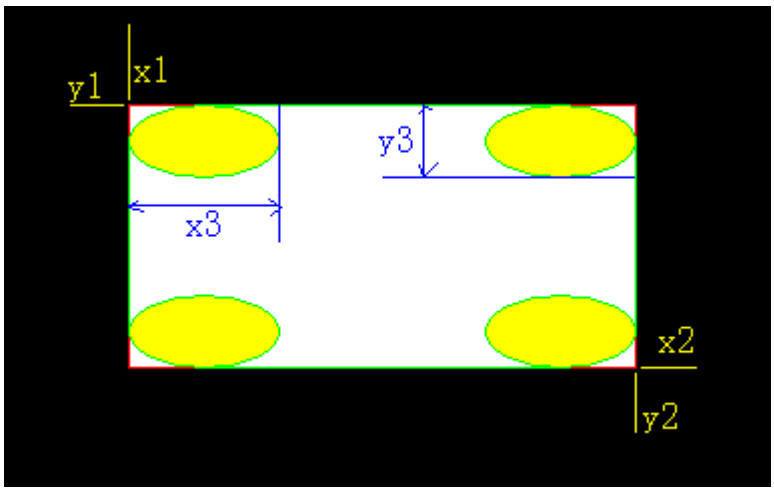
圆角矩形的角是圆的，所以需要多增加两个参数，大家可以用以下的程序去测试：

```
        case WM_LBUTTONUP:
```

```
x3=20;
y3=15;
x2=LOWORD(lParam);
y2=HIWORD(lParam);
hdc=GetDC(hwnd);
        RoundRect(hdc,x1,y1,x2,y2,x3,y3);
ReleaseDC(hwnd,hdc);
break;
```

（注：x3,y3 的定义与前面的变量定义放在一起。）

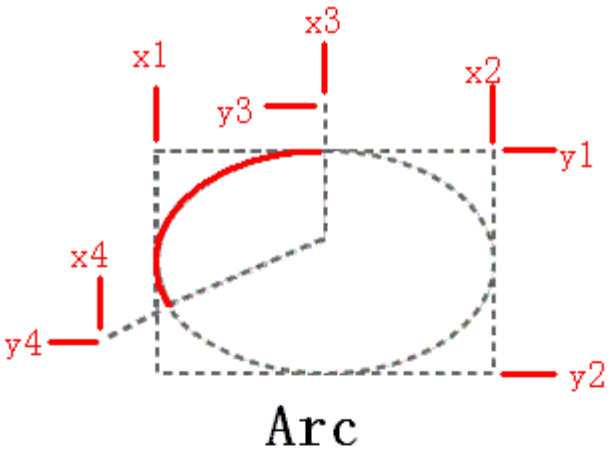
我们看看圆角矩形的后两个参数有什么含义：



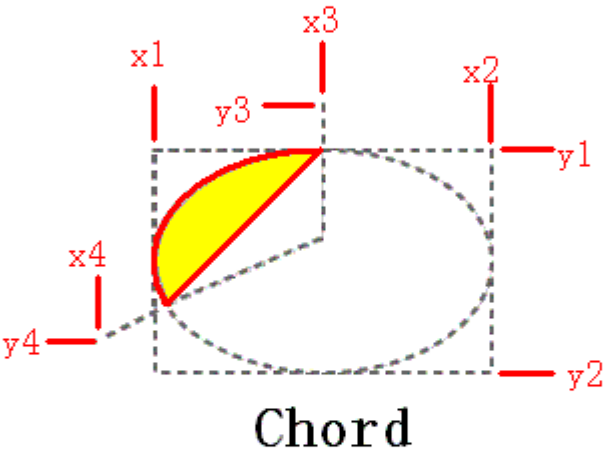

也就是说，圆角矩形的四个角相当于放了四个椭圆，而这四个椭圆的长径为 x3，短径为 y3，仅此而已，呵呵，好了，下面再来看一看下面三种图：弧、弓、饼

```
Arc(hdc,x1,y1,x2,y2,x3,y3,x4,y4);
Chord(hdc,x1,y1,x2,y2,x3,y3,x4,y4);
Pie(hdc,x1,y1,x2,y2,x3,y3,x4,y4);
```

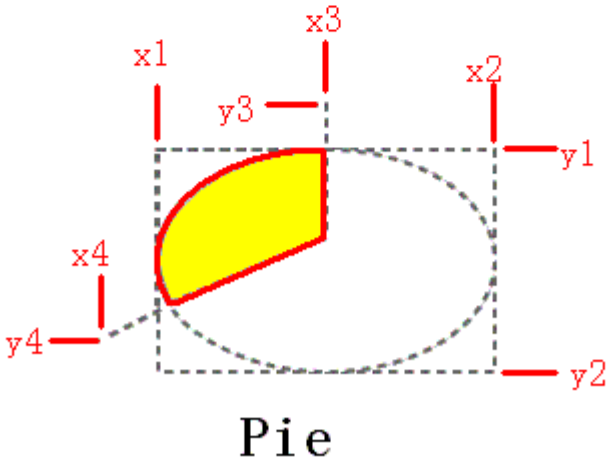
好家伙，九个参数，后面分别是四个点，看图：




Arc



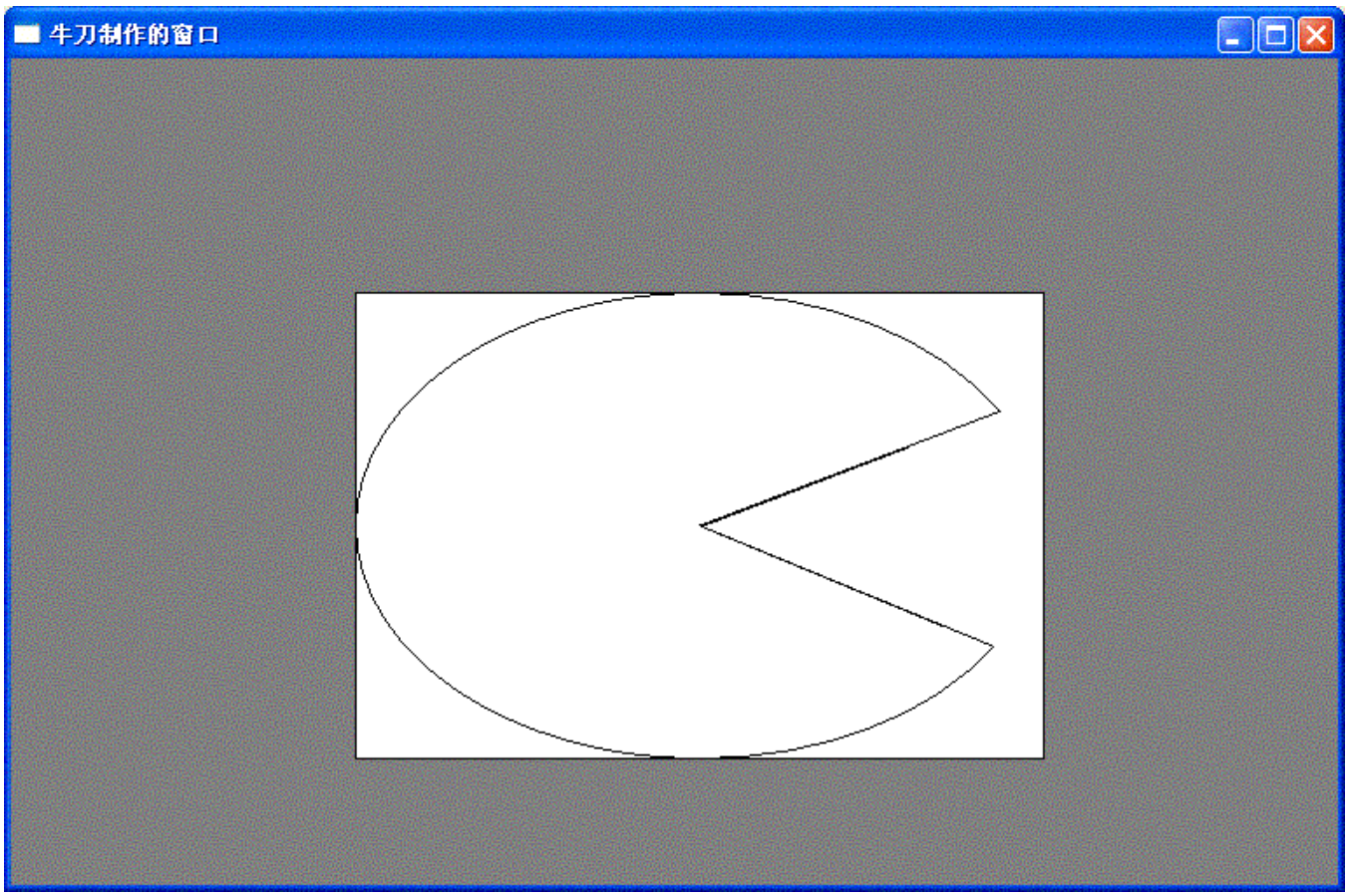

Chord




Pie

感觉怎么样，呵呵，前五个参数的后四个也是限定了一个矩形，而那段弧就是内切椭圆切下来的，然后四个参数一个是限定了弧的起点，一个是终点，两个点向矩形中心的延线切开一段弧，可能你会问，为什么不用弧上的点呢？你想是随意的两点好找还是弧上的两点好找呢？呵呵。好了，这一讲我们就到这儿吧，下面的程序可以测试上面的弧、弓、饼

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        HDC hdc;
        static int x1,y1,x2,y2,x3,y3,x4,y4;
        RECT rect;
        switch(message)
        {
        case WM_LBUTTONDOWN:
                x3=LOWORD(lParam);
                y3=HIWORD(lParam);
                break;
        case WM_LBUTTONUP:
                GetClientRect(hwnd,&rect);
                x1=rect.right/4;
                x2=3*rect.right/4;
                y1=rect.bottom/4;
                y2=3*rect.bottom/4;
                x4=LOWORD(lParam);
                y4=HIWORD(lParam);
                hdc=GetDC(hwnd);
                        Rectangle(hdc,x1,y1,x2,y2);
                        Arc(hdc,x1,y1,x2,y2,x3,y3,x4,y4);
                        MoveToEx(hdc,rect.right/2,rect.bottom/2,NULL);
                        LineTo(hdc,x3,y3);
                        MoveToEx(hdc,rect.right/2,rect.bottom/2,NULL);
                        LineTo(hdc,x4,y4);


                ReleaseDC(hwnd,hdc);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

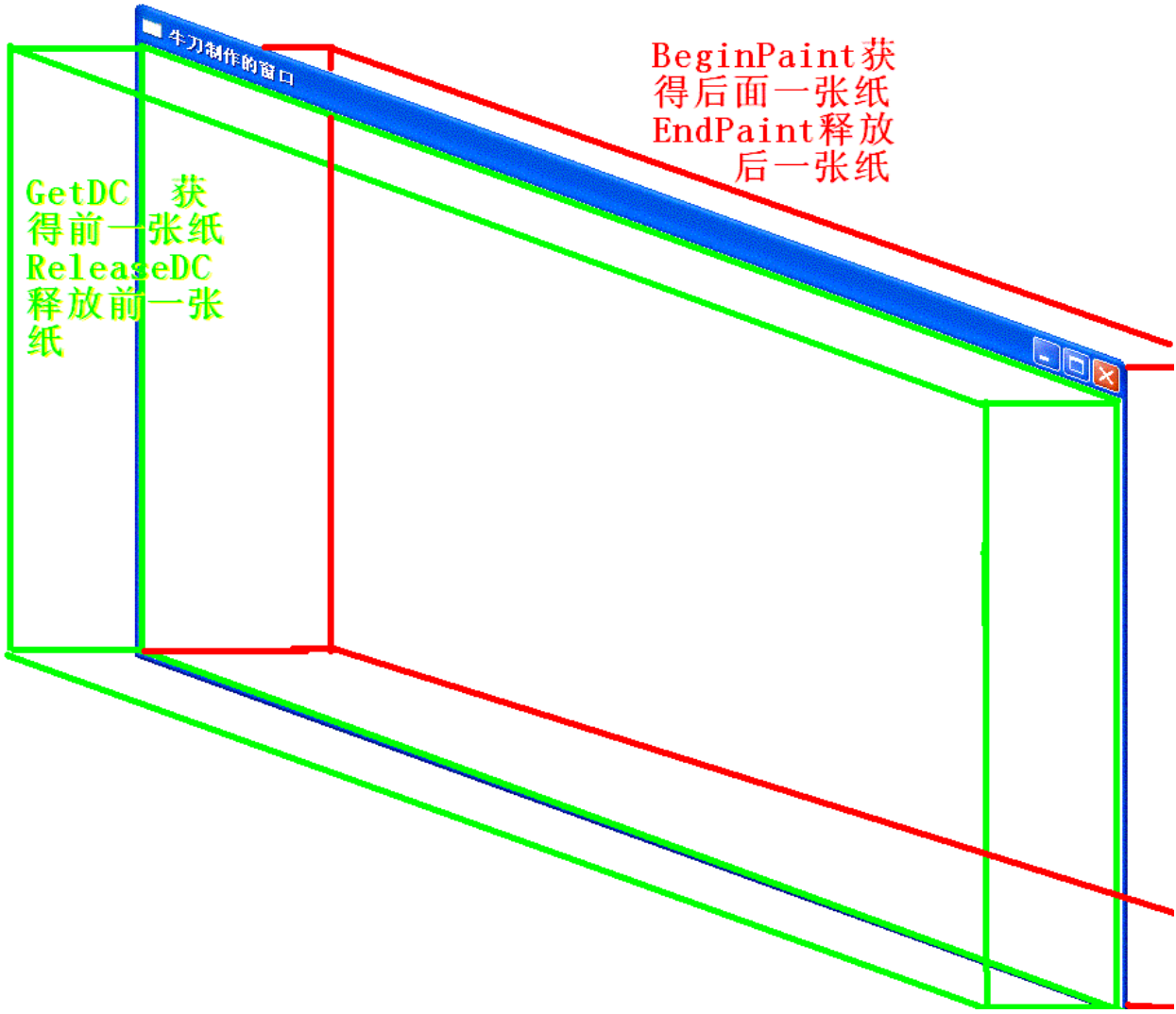



看像不像《吃豆豆》中的食客？呵呵。

**微软积木〇七〇：神奇玻璃**

上一讲我们讲到了窗户上的那张纸，我们也称之为“毛玻璃”，呵呵，为什么这样说呢？因为我们同一块玻璃，还有个消息会在当窗口出现后再次在上面画东东，那个消息就是　WM_PAINT

好了，一张纸有两面，我们用鼠标在上面画时，你可以看见，但如果在 WM_PAINT　中画，你可看不见，因为画在反面，但虽然看不见，等其画好了后，就会有一束神奇的平行光打到纸上，于是，我们在正面就看见了，而由于这张纸有一定的厚度，所以理论上有一定的距离，当然如果厚度不好理解的话，我们可以理解成有两张纸，平行地放着，相当于一张纸的两面，当在那张纸上画完后，就会有一束平行光将纸上的内容照过来，让我们看见。




实在不是牛刀想这么想，如果你是一个编程老手请允许牛刀像这样想，因为这种构思涉及到以后窗口与视口，甚至涉及到一些坐标变换的问题，其实我们取得的设备描述表是一样的，但在 WM_PAINT 中必须用　BeginPaint　去取得，而用 EndPaint 去释放，什么意思呢？可能有人还不些不懂，我们看程序吧：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        switch(message)
        {
        case WM_LBUTTONDOWN:        //在非 WM_PAINT 消息中
                hdc=GetDC(hwnd);

                ReleaseDC(hwnd,hdc);
                break;
        case WM_PAINT:        //在 WM_PAINT 消息中
                hdc=BeginPaint(hwnd,&ps);

                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

什么意思呢？意思是说，我们的 BeginPaint 与 EndPaint 函数中有一个结构 PAINTSTRUCT ，而这个结构中有一个矩形结构 RECT，表示某个区域，当这一段区域如果被其他窗口所覆盖后，再出现，就会由 WM_PAINT 消息对这个区域进行重画，画过后，用探照灯一打，我们就可以重新看见了，于是，这个窗口上只要在 WM_PAINT 中画的东西，就不会消失了，上一讲我们做的程序如果有其他窗口覆盖，那么覆盖的那一部分画的东西就不见了，因为那里没有重绘机制，而在我们 WM_PAINT 消息中就会先将消失的一部分用新的一块纸贴一下，然后重新画出消失的部分，再用灯一照，呵呵，好像挺难理解吧，看段程序吧：

```
case WM_PAINT:        //在 WM_PAINT 消息中
        hdc=BeginPaint(hwnd,&ps);
        TextOut(hdc,20,20,"在这里画的东东永远都消失不了",28);
        EndPaint(hwnd,&ps);
        break;
```

为什么消失不了呢？因为每次窗口看不见，再重新出现后，系统都会发送这个消息，所以这个消息很可惜的是不能调试，因为一调试就进入了死循环，永远看不到结果。

当然我们的程序也可以用 InvalidateRect(hwnd,NULL,TRUE); 来重新画一下，也就是发送一个消息，好了，做下面的程序：

```
case WM_LBUTTONDOWN:
```

```
            InvalidateRect(hwnd,NULL,TRUE);
            break;
```

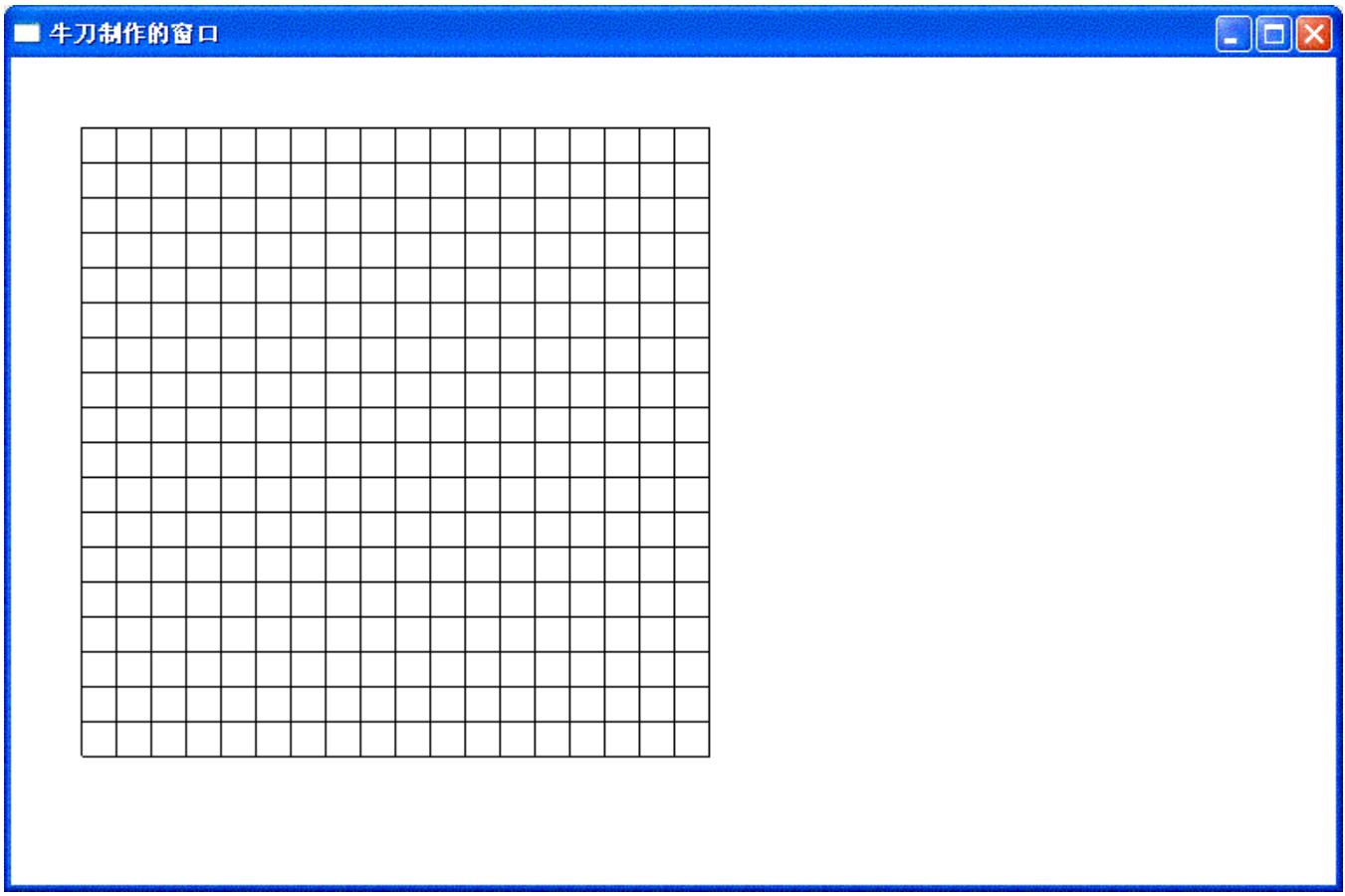
好了，你运行程序后，在屏幕上点一下，看见显示的内容一闪是吧？也就是说，前面一张纸被换了，并且重新画了一遍。（当然，上面第二个参数是可以传进重画的区域的，那我们以后再与大伙讨论）。

当然，也正因为如此，所以我们在这里做的程序只要程序不关，画的东东我们就能够看得见，好了，那我们就用这个消息来画一个围棋棋盘吧：

围棋棋盘是 19X19 路，画线就一切 OK 了。

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        int i,x,y;
        switch(message)
        {
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                for(i=0;i<19;i++)
                {
                        x=i*20+40;
                        y=40;
                        MoveToEx(hdc,x,y,NULL);
                        y=40+18*20;
                        LineTo(hdc,x,y);
                        MoveToEx(hdc,y,x,NULL);
                        y=40;
                        LineTo(hdc,y,x);
                }
                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

感觉好极了：




为了让大家能够真正地体会 WM_PAINT 的一些特性，我们再介绍两个消息：

WM_CREATE

此消息就在窗口产生之时第一个消息，并且只出现一次，所以如果想做的一次性任务，一般就放在这个消息中。

WM_TIMER

计时器消息，等待计时消息的到来，从而响应，当然喽，响应此消息是固定某一个时间发出的，是系统发出的，不是我们用户发出的，所以这个消息可能是我们介绍系统消息之一吧，但是想要系统为我们发送此消息，就要设置一个计时器，当然设置一次就行，所以一般都放在 WM_CREATE 消息中，我们看程序吧：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM
lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
```

```
        int i,x,y;
        switch(message)
        {
        case WM_CREATE:
                SetTimer(hwnd,0,100,NULL);
                break;
        case WM_TIMER:
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                for(i=0;i<19;i++)
                {
                        x=i*20+40;
                        y=40;
                        MoveToEx(hdc,x,y,NULL);
                        y=40+18*20;
                        LineTo(hdc,x,y);
                        MoveToEx(hdc,y,x,NULL);
                        y=40;
                        LineTo(hdc,y,x);
                }
                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

如果你上面点鼠标时没感到跳的话，这次感觉到屏幕在跳了吧？计时器消息是一个很好玩的消息，很容易实现一些动画的效果，大家还记得上节我们为大家做的《吃豆豆》中的食客吗？我们这就来做一个：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        RECT rect;
        static int x1,y1,x2,y2,x3,y3,x4,y4,y=0,ymax=0,dy=15;
        switch(message)
        {
        case WM_CREATE:
                SetTimer(hwnd,0,1,NULL);
```

```
                break;
        case WM_TIMER:
                y+=dy;
                if(y<1)dy=15;
                if(y>ymax-1)dy=-15;
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                GetClientRect(hwnd,&rect);
                x1=rect.right/4;
                x2=3*x1;
                y1=rect.bottom/4;
                y2=3*y1;
                ymax=rect.bottom/2;
                x3=x4=x2;
                y3=ymax-y;
                y4=ymax+y;
                Pie(hdc,x1,y1,x2,y2,x3,y3,x4,y4);
                EndPaint(hwnd,&ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

怎么样？闪得比较厉害吧，似乎看到了灯光在作怪，呵呵。

好了，下面我们向大家解释上面一些特殊的出现过的东东，以便大家理解。

·RECT

结构体，是一个矩形结构，其中有上下左右四个值，表示一个矩形。

·PAINTSTRUCT

结构体，是 WM_PAINT 消息专用的结构体，我们下面来看看其定义：

```
typedef struct tagPAINTSTRUCT { // ps
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
```

} PAINTSTRUCT;

看到了么，里面有句柄，有矩形区域，还有一些我们刷新屏幕所需要的一些变量与参数，从而可见 WM_PAINT 消息与其他消息所要处理的信息是有所不同的，所以一般常规情况下，我们有以下规则：

WM_PAINT 消息中，取得纸（设备描述表）用 BeginPaint 函数，释放用 EndPaint 函数，函数中使用到了结构体 PAINTSTRUCT
而在其他消息响应中，一定不能使用这两个函数，这两个函数是由 WM_PAINT 消息专用。

那么在其他消息中是不是就只能用 GetDC 呢？非也，还有一些函数如: GetWindowDC

可以取到窗口的 DC，我们看一段程序：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT message,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        static int x,y;
        switch(message)
        {
        case WM_LBUTTONDOWN:
                x=LOWORD(lParam);
                y=HIWORD(lParam);
                break;
        case WM_LBUTTONUP:
                hdc=GetWindowDC(hwnd);
                        MoveToEx(hdc,x,y,NULL);
                        x=LOWORD(lParam);
                        y=HIWORD(lParam);
                        LineTo(hdc,x,y);
                ReleaseDC(hwnd,hdc);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,message,wParam,lParam);
}
```

大家看到了什么了？你的线条可以画到标题栏上了是吗？不错，那就是我们取到的纸连窗户框都糊起来了，再来，将其中的参数变为 NULL：

hdc=GetWindowDC(NULL);

再画几条线看看，画哪儿去了？呵呵，hwnd 是我们窗口的句柄，而如果这是一个空值的话，那就取得了我们屏幕的 DC 了，从而使得我们可以在屏幕上画画，可惜的是，我们的屏幕上的鼠标点击的消息我们暂时还捕捉不到，如果能捕捉到，并且知道一些坐标的原因，那我们竟然可以在屏幕上画画，呵呵。

事实就是很好玩的，但我们一定要知道他们的原理，并且能够灵活地运用这些原理，否则我们就只能抄抄别人的程序，轮到叫你自己去做一个程序，就要挠头了，呵呵。

好了，这一讲到此为止，我们的 API 函数部分就宣布讲解结束了，当然还有好多的内容，我们放到 MFC 中去讲，我们在这儿只需要理解消息循环，与我们做一个程序的一般原则就行了，而我们真正做程序则基本上都是在 MFC 中，并且都是面向对象的，但是在 API 部分，我们需要让大家知道的就是，消息循环是怎么实现的，常用的消息循环，我们的纸与笔（当然，如何创建一个新笔，新画刷等内容我们通通地都放在 MFC 部分与大家慢慢地细聊，但我们的 MFC 其实还有好多的路要走，从下一讲开始，我们就需要从指针、类部分开始为我们的 MFC 去打基础），我们的 WM_PAINT 消息，如何获取设备描述表。

这一讲的实质性的内容不多，希望大家结合上一讲的内容，去理解一些图形的基本画法，试着自己做一些作图软件。

那么下面为了让大家真正地去体会消息循环的实质，我们这次再将上面的那小食客的程序改一改，不过这次，我们不做那种小路上的土匪了，我们打劫 WinMain 中的消息循环，看代码：

```
MSG msg;
while(GetMessage(&msg,NULL,0,0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

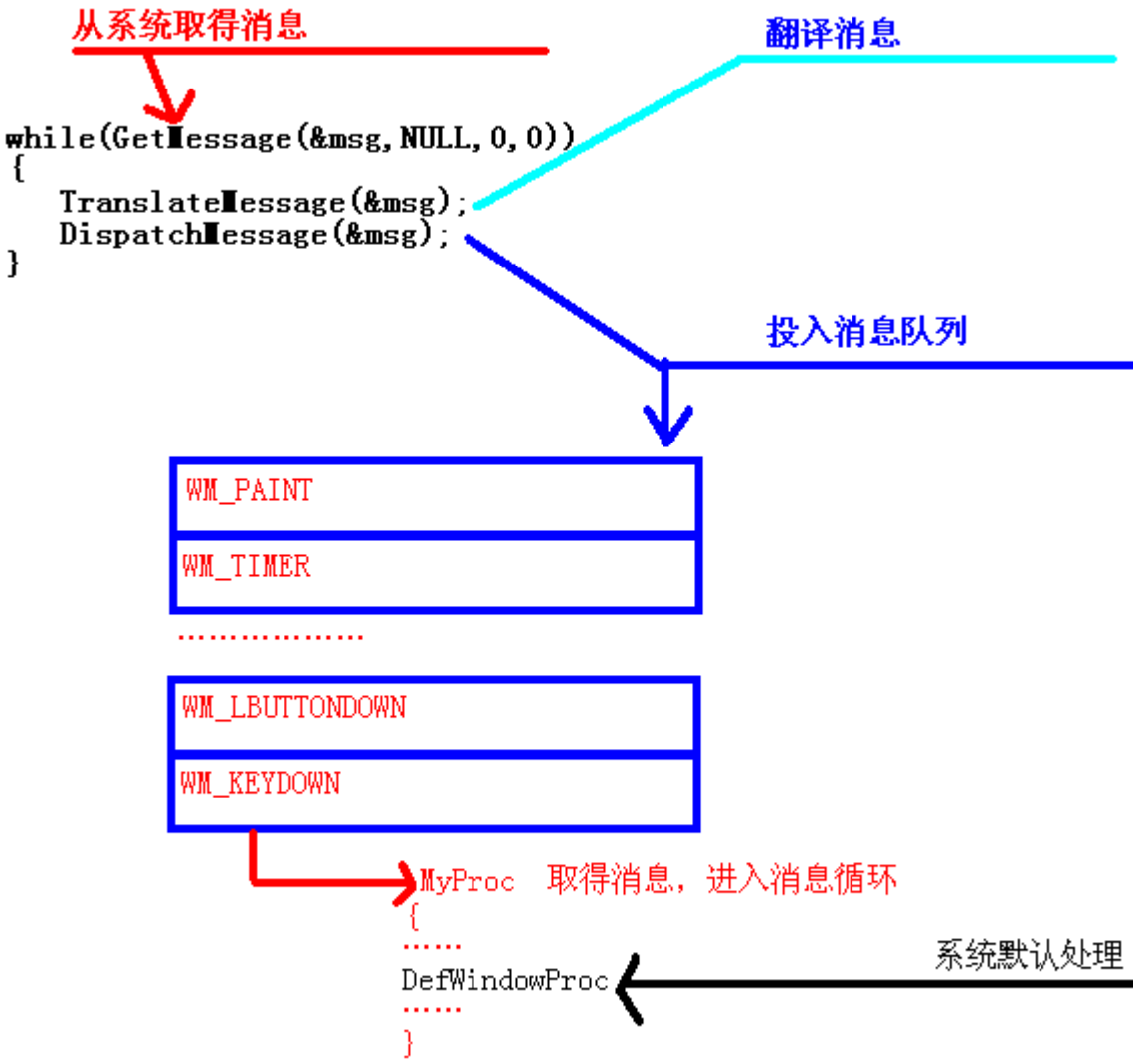
那么那个 MSG 结构体中有什么呢？

```
typedef struct tagMSG {    // msg
    HWND  hwnd;
    UINT  message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

看，竟然是一套全，那好，我们在其中打劫一个简单的吧， WM_TIMER 消息，就是说，不让那个 WM_TIMER 消息进入我们下面程序进入的消息循环了，那怎么做呢？一个消息，使用 if 语句就一切 OK 了呗，看程序：

```
while(GetMessage(&msg,NULL,0,0))
{
        if(WM_TIMER!=msg.message)
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        else
        {
                HDC hdc;
                int x1,y1,x2,y2;
                RECT rect;
                GetClientRect(hwnd,&rect);
                x1=rand()%rect.right;
                x2=rand()%rect.right;
                y1=rand()%rect.bottom;
                y2=rand()%rect.bottom;
                hdc=GetDC(hwnd);
                        Rectangle(hdc,x1,y1,x2,y2);
                ReleaseDC(hwnd,hdc);
        }
}
```

呵呵，拦截的怎么样，这种打劫的手段多火啊，呵呵，并且在这和打劫还有一个最大的好处就是：没有延迟。原来啊，我们 WINDOWS 消息机制中有一个队列机制，就是不管什么消息都要到队列中去排队，（注：是严格的排队哟，不管那产生的消息，是从队列的一头进，另一头出，当然了，这个队列就有 DispatchMessage 负责，而 TranslateMessage 我们与大家讲过，是翻译消息，那么，我们的消息响应机制就可以理解成如下一个图：




怎么样？这种图还能够接受吧？（显然上面打劫 WinMain 消息执行速度更快，因为不需要将消息放入消息队列，我们的快捷键技术就是应用的这种技术，所以说，如果你用 Ctrl+S 去保存东东，可以做到立即保存，一点都不耽搁）好了，下一讲我们与大家共同从指针开始探讨。

## 微软积木〇七一：奇妙指针

以上我们学的瓦工技术，怎么从 API 开始，去砌一个窗体，这一讲开始，我们就要讲缝纫技术，教大家怎么去做一件外套。或者用这种技术，怎么去拆一间房，呵呵。

当然，既然是缝纫，那就要一针一线地与大伙讨论，这一讲我们就开始讲讲我们缝纫技术中的“针”——指针。（不要学缝纫的人再用搜索引擎搜到这儿来哟，呵呵）

说实话挺好玩，由于计算机出现得比较晚，就什么东西都没有的用，然而，正因为没有什么东西用，所以学电脑的，做软件的就什么方面技术前沿的东西都敢拿来用，你以后看，我们还有裁剪技术，模板技术，父子关系，朋友关系，对外关系，对内关系，还有什么外部接口等等，都敢用，等着看吧，我们这一讲就先与大家将指针给“挑明”。

什么叫指针？别看名称，也不知道是怎么翻译过来的，其实用另一个名称更好——“地址”。

不错，指针就是地址，现在的地址一般 3 2 位，所以我们的指针就占四个字节，任何一个指针变量占四个字节，当然也有例外，那就是——数组。

为什么数组不是占四个字节呢？因为数组是分配好的指针，例如

int a[3];

那么 a 就是地个固定的地址，这个地址在编译过程中已经变成了一个固定的指针形成，也就是说 a[0] 如果是 0x0020 的话，那么 a[1] 就是 0x0024 ，而 a[2] 就是 0x0028 事实就是这样，因为 a 不能动，所以不需要一个内存去存储这个值，而仅仅是通过下标去获取数组中的相应的值。

所以 sizeof(a) 的值不是 4，而是 12，也就是说 a[0] a[1] a[2] 其实就像我们定义 int i,j,k; 两个变量一样，用时当作数据用，但可以用下标的方式去访问数据。

好绕人啊是不？所以有人问过 strcpy();的用法时，我们就需要知道，其实 strcpy 函数传进的是指针，也就是数组名，而不是数组中的数据，看一段程序，上机测试一下，仔细体会：

```
#include <stdio.h>
#include <string.h>
void main()
{
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char str[30];
        for(int i=0;i<26;i++)
```

```
        {
                strcpy(str,&ch[i]);          //请注意前面的取地址符，仔细体会，尽量将其看成指针
                printf("str=%s\n",str);
        }
}
```

上面取地址符就告诉我们，其实是可以用数组的名称当着固定的指针的。

正因为数组名是代表一个位置，在并不占内存单元，所以我们不能移动这个指针，也就是不能对这种固定的指针进行运算，如：

```
ch++;
```

那是错误的，但是一个指针就可以，还是上面那个程序，我们改用指针来再做一下：

```
        char ch[]="abcdefghijklmnopqrstuvwxyz";
        char *pch=ch;
        char str[30];
        for(int i=0;i<26;i++)
        {
                strcpy(str,pch++);
                printf("str=%s\n",str);
        }
```

看有什么感觉，是不是程序突然显得简洁了？不错，这也就是指针的魅力所在。

·知识点　数组可以看作是指针，是固定的指针，但数组名不占内存，所以不能对其进行移动操作，但的确是指针，我们定义一个指针，用 *　放在相应的类型后面就可以，指针都是四个字节，只要是内存中有地址的东东，都可以用指针指过去。

上面这个知识点是不是有点绕？没关系，我们任何一个数据，都可以用　*　定义一个指针，不管是一个整形、浮点型、字节型、结构、联合、类、甚至函数。

这些东东全是放在内存中的，所以只需要用一个　*　就可以定义相应的指针，用　&　就可以取得地址，看一个程序吧：

```
#include <stdio.h>
typedef struct tagA{
        int x,y;
} A;

void main()
{
        A a;
}
```

怎么样？定义了一个结构体 A ，那么我们如何取得这个结构体的指针呢？很简单，加 * 定义，加 &取地址呗：

```
void main()
{
        A a;
        A *pa;
        pa=&a;
}
```

这么容易就取到了结构体的指针了，那么对其数据进行操作也就简单多了，看程序：

```
        (*pa).x=4;
        (*pa).y=5;
```

不错，用 * 取得相应地址处的结构，然后用 . 号一样地使用结构中的数据，别忘了外面要加括号，当然，对于 (*pa). 这种结构，为了方便，我们的 C++ 专门定义了相应的指针类型数据的操作符 -> ，也就是说，用这个就可以直接操作指针中的成员变量了，例如上面的程序可以改为：

```
        pa->x=4;
        pa->y=5;
```

看，多简洁。原因就是这种指针的作用太大了，于是人们不得不为其增加一个运算符，方便啊，当然，我们以后涉及到 MFC 时就会真切地体会到其方便性了。

结构中还能放函数，看我们将 A 中增加一个函数：

```
typedef struct tagA{
        int x,y;
        void print()
        {
                printf("这是结构 A 中的 print 函数\n");
        }
} A;
```

怎么样？感觉好吧，呵呵，可以用成员变量一样地调用函数，继续在上面入口函数中加入：

```
pa->print();
```

运行程序，大家看怎么样？函数的运行与外部的函数一样的调用吧，方便，省事，于是，我们的这个结构就似乎有点智能了，我们再改一改程序：

```
#include <stdio.h>

typedef struct tagA{
```

```
        int x,y;
        void print()
        {
                printf("A 中的 x=%d,y=%d\n",x,y);
        }
} A;

void main()
{
        A a;
        a.x=4;
        a.y=5;
        a.print();
        A b;
        b.x=11;
        b.y=43;
        b.print();
}
```

怎么样？还可以吧？那么如果我们定义一个指针，那么这个指针就可以任意地指向 a 与 b ，从而执行他们其中的函数了：

```
        A *pa;
        pa=&a;
        printf("执行了 pa=&a 后\n");
        pa->print();
        pa=&b;
        printf("执行了 pa=&b 后\n");
        pa->print();
```

我们结构中还有一个很特殊的指针——this 指针，这个指针不用定义，就是指的自己，我们可以将上面的 print 函数改一改：

```
#include <stdio.h>

typedef struct tagA{
        int x,y;
        void print()
        {
                printf("本结构体的地址为%d\n",this);
        }
} A;

void main()
{
        A a;
```

```
    a.print();
    printf("结构体 a 的地址为%d\n",&a);
}
```

大家看结果：

```
本结构体的地址为 1245048
结构体 a 的地址为 1245048
```

地址一样，所以说如果在结构体中，合理地使用 this 指针，可以使得我们的程序制作变得事半功倍，看程序：

```
    void Setxy(int a,int b)
    {
        this->x=a;
        this->y=b;
    }
```

你可能说，不是复杂了吗？你以后自己编程时去比较比较，想知道梨子的滋味，最好去尝一个。

当然，指针的最大的好处就是形成链表，使得我们的计算机突然有了好多的管理功能，或者做成一个树，一个图，使得我们的计算机的计算机功能会突然变得那样低级，而特别是类中加入了函数的功能，使得链表上的每一个节点都突然地活了起来，从而可以实现一些智力性的工作，这一点让人们尤其地欣喜。

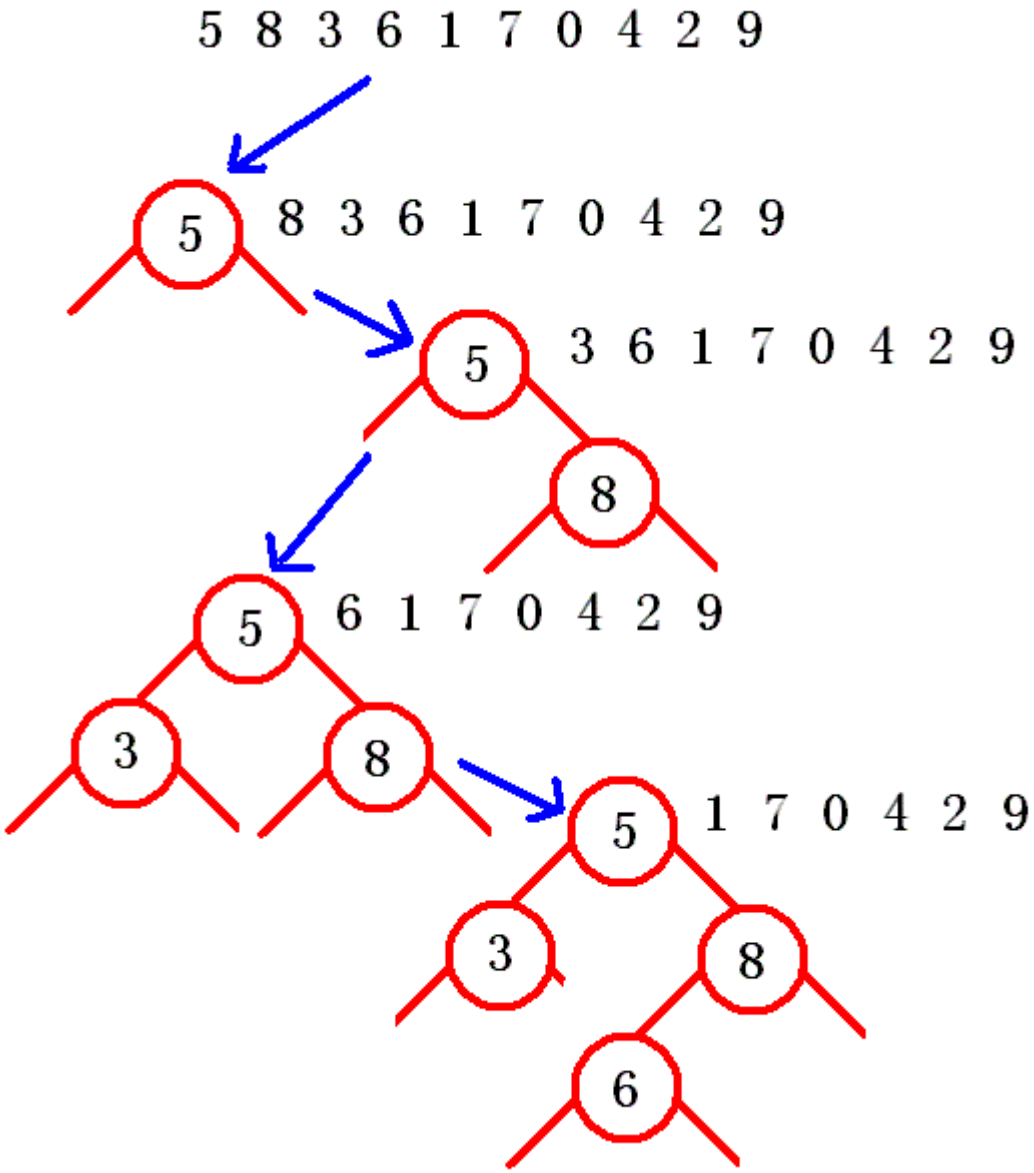
当然，在接下来那么短的文章中显然是不可能为大家做一个人工智能系统的，但我们可以为大家做链表与树以及图提供一些合理的建议。

·建议 最好是用类，而不是用结构，并且将所有的对链表的操作都放到类中，以使得我们的程序看起来显得很是智能。

牛刀以前曾经说过，我们的计算机是不可能实现真正的面向对象的，但即使不是真正的面向对象，对我们用面向对象的思维去思考问题一点妨碍都没有，好了，我们做个什么程序呢？做一款排序树吧，呵呵，什么叫排序数呢？当然是排序喽，排序的方法是有好多，但我们以前的算法都是面向过程的，而这次我们要面向对象。

那排序树的原理是什么呢？是一棵二叉树，所以我们要有两个指针，只要建立这棵树，也就是第一个数据放在根，然后将要插入的数据与根数据比较，如果比根大则放在右子树，如果小则放在左子树，只要放好了，树就做好了，而排序树的左根右的遍历就是从小到大排列，右根左的遍历就是从大到小排列，下面我们就以一组数据：

5 8 3 6 1 7 0 4 2 9 为例来做一棵排序树：




就是按图一那样，一棵一棵地排过去，当然喽，我们可以立即定义这个类：

```
#include <stdio.h>

class Tree
{
};

void main()
{
	int a[10]={5,8,3,6,1,7,0,4,2,9};
	Tree head;
}
```

当然喽，不排序我们也知道最终结果是 0123456789 ，呵呵，当然你可以换数字啊，随便换，只要不超过整数的范围就行。

那么我们这里由于要保存树的一些枝叉，所以我们要定义一些私有的数据，以前我们说过，私有的是类的默认，所以我们在类中就可以直接定义：

```
class Tree
{
        int x;
        bool fla;
        Tree *left,*right;
public:

};
```

当然，如果大家还记得类中有构造函数，我们就先使用一下，如果大家不能理解我们以后还要详细讲：（对了，类后应该有一个分号别忘喽）。

```
        Tree()
        {
                fla=false;
                left=NULL;
                right=NULL;
        }
```

我们这里的 fla 是准备干什么呢？我们的目的是为了表明当前有没有插入数据。默认为没有了，当然，下面我们就定义一个插入函数。

```
        void Insert(int a)
        {
        }
```

怎么插入呢，当然先看当前有没有数据了，即看 fla 如果没有则插入，否则就需要新建一个子树了，先确定没有数据

```
        void Insert(int a)
        {
                if(!fla)
                {
                        fla=true;
                        x=a;
                }
                else
                {
                        if(a<x)
                        {
                                if(!left)left=new Tree;
                                left->Insert(a);
                        }
                        else
                        {
```

```
                    if(!right)right=new Tree;
                    right->Insert(a);
                }
            }
        }
```

大家看我们是在干什么呢？节点根据自身的情况先判断有没有数据 fla ，如果没有，则插入，否则就看看数据大小，如果小，则看看有没有左结点，如果有就执行左结点的插入操作，如果没有就新建一个，再执行插入操作，而至于子结点的情况，就撒手不管了，原因很简单，因为子结点也有自己的函数， Insert 那都是子结点的事了，似乎与本节点无关了，厉害不，呵呵，于是，每个结点就似乎都活了，那么怎么排序呢？很简单，看程序：

```
    void Sort()
    {
        if(left)left->Sort();
        printf("%d\n",x);
        if(right)right->Sort();
    }
```

有什么感觉没有，该子结点的事都交给子结点去做了，似乎与自己没有关系。

大家是不是会想到递归调用啊？这不是递归么？

哎，如果你不用类，你还就得递归，如果你用了类，这就不是递归了，绝对不是递归，因为他们调用的函数其实是不同的啊，每一个指针都是指向不同类中的函数，也就是说，子类中的函数是后产生的，而不是原来的那一个函数，子类中的函数是我们通过指针去 new 出来的。而递归的定义是一个函数去调用自己，那才叫递归。

好了，我们看一看完整的程序吧：

```
#include <stdio.h>
class Tree;
class Tree
{
        int x;
        bool fla;
        Tree *left,*right;
public:
        Tree()
        {
                fla=false;
                left=NULL;
                right=NULL;
        }
        void Insert(int a)
        {
                if(!fla)
                {
```

```
                        fla=true;
                        x=a;
                }
                else
                {
                        if(a<x)
                        {
                                if(!left)left=new Tree;
                                left->Insert(a);
                        }
                        else
                        {
                                if(!right)right=new Tree;
                                right->Insert(a);
                        }
                }
        }
        void Sort()
        {
                if(left)left->Sort();
                printf("%d\n",x);
                if(right)right->Sort();
        }
};

void main()
{
        int a[10]={5,8,3,6,1,7,0,4,2,9};
        Tree head;
        for(int i=0;i<10;i++)
        {
                head.Insert(a[i]);
        }
        head.Sort();
}
```

类的子指使得我们的类有了生命，这是面向过程的语言所不能做到的。（对了，上面程序运行的结果想必大家都知道吧，说不说没太大的意思了，呵呵。

当然，我们做一个链表，那么这个链表就有了生命，自己能够根据人们的需要而自由插入、删除，其他的所有数据结构，一旦使用了类就似乎都有了生命，

其实说实话，大家看上面的 main 函数，多么简单，似乎仅仅是定义一个变量，然后与变量作两次对话变量就自动将结果告诉我们了，我们称这种变量为——对象。

**微软积木〇七二：面向对象**

对象，什么是对象？对象就是类吗？

对像不是类，什么类，例如我们都是人类，但你我就是对象，就这么简单，对象是一个单个体，而类是一种定义，仅此而已，所以对于我们来说，我们在制作类的过程其实与面向对象却房毫无关系，仅仅在我们作一个定义了，如上一讲我们的：

Tree head;

其中的 Tree 是类，而 head 就是一个对象，我们称这种定义为对象的实例化，也就是说，产生的 head 是一个对象，而 Tree 是一个类。

所以我们在设计一个类，有时候也称之为设计一个对象，但一般我们不这么称，从这一点上来说，我们这一讲就几乎与面向对象没什么关系，很可能我们一直都在讨论类，都没有讨论对象，但我们的确在设计所有对象所具备的种种类型。

就好像我们去做一件衣服，我们做的是什么？是布，是剪刀，是线，而不是衣服，但我们称用剪刀将布裁成小块然后拼成一件衣服的过程叫做“做衣服”，其实不是，倒不如说我们在“做布”，或“加工布”更为贴切，衣服就是一个成品，但我们加工的却并不是衣服，事实就是如此，绕人吧？那我们就不与大伙绕了，我们就从类谈起：

首先我们要知道我们为什么要使用类？当然有种种的目的，而基于这种种目的，我们的类才会有种种的功能，那么我们接下来就与大伙一个一个功能与目的的讨论。

大家还记得以前我们曾经给大家留过一个问题吗？程序一定要首先从入口函数 main 函数中去执行吗？还会不会有其他的情况，我们还没有进入 main 函数，我们就已经开始在执行程序了？

有，这就是我们类中的构造函数：看代码：

```
#include <stdio.h>
class A
{
};          //一定不要忘记这儿必须有一个冒号

void main()
{
        A a;
}
```

为了跟踪程序，我们在 main 中入口与出口处做一些注释：

```
void main()
```

```
{
        printf("进入了 main 主函数\n");
        A a;
        printf("正在退出 main 主函数\n");
}
```

程序运行结果大家可以想当然了是吧，呵呵，当然，我们知道，类中是可以有构造函数与析构函数的，我们加进这两个函数：

```
class A
{
public:
        A()
        {
                printf("进入了构造函数\n");
        }
        ~A()
        {
                printf("进入了析构函数\n");
        }
};
```

上面程序运行的结果会是什么呢？

```
进入了 main 主函数
进入了构造函数
正在退出 main 主函数
进入了析构函数
```

不错，大家看到，构造函数是在定义时调用，而析构函数是在退出主函数时调用，是这种情况吗？我们强行地加进一些花括号来看：

```
void main()
{
        printf("进入了 main 主函数\n");
        {
                A a;
        }
        printf("正在退出 main 主函数\n");
}
```

再运行程序，出现什么情况了呢？

```
进入了 main 主函数
进入了构造函数
进入了析构函数
正在退出 main 主函数
```

怎么样？很出你的意外吧？原来啊，我们提出一套牛刀作用域理论（当然了，这是事实理论，也是实践理论，但书不却不这么说，呵呵）

变量（对象）的作用域以定义为开始，以对应花括号结束为结束。

这句话什么意思呢？我们继续来看程序：

```
void main()
{
        printf("进入了 main 主函数\n");
        {
                A a;
                {
                        printf("进入花括号\n");

                        printf("退出花括号\n");
                }
        }
        printf("正在退出 main 主函数\n");
}
```

运行结果是什么呢？

```
进入了 main 主函数
进入了构造函数
进入花括号
退出花括号
进入了析构函数
正在退出 main 主函数
```

也就是说，我们这里对应花括号的意思是，从变量向后找右括号，遇到一个左括号与一个右括号为一组，如果出现成组的括号则忽略，直到出现不成组的右花括号则变量作用域结束，看例：

```
A a;{{}{}}}
```

上面出现了好多的花括号，我们分开来看

```
A a;{ {} {} }    }到最后一个结束，
```

会相信看牛刀教材的朋友们一看就明白怎么回事，不详细解释了，又不是去搞什么理论证明，但我们在 VC 排版中，一般成队的括号前面的缩进都一样，大家一看就知道变量的作用域了。好，我们再来看一段程序：

```
#include <stdio.h>
int i=5;
void main()
{
        int i=3;
        printf("i=%d\n",i);
}
```

想必大家一定知道，上面的输出结果，结果为 3 是不，那么如果我们再加上一句呢？

```
#include <stdio.h>
int i=5;
void main()
{
```

```
        printf("i=%d\n",i);
        int i=3;
        printf("i=%d\n",i);
}
```

结果是不是会很让你意外呢？

```
i=5
i=3
```

那么我们再来在其中加上一些中括号：

```
#include <stdio.h>
int i=5;
void main()
{
        printf("i=%d\n",i);
        {
                int i=3;
                printf("i=%d\n",i);
        }
        printf("i=%d\n",i);
}
```

结果是：

```
i=5
i=3
i=5
```

牛刀不希望大家迅速地看过去，牛刀需要大家仔细地去体会。如果说体会，牛刀再告诉大家一个作用域标识符（::）两个冒号，作用是使用全局变量，什么意思呢？

显然是我们大括号里面的 i 覆盖了外面定义的 i，而一旦出了大括号，外面定义的 i 又可见了，那么 :: 就是去看全局的 i，我们试着做一些程序，大家用心去体会，一定要体会：

```
#include <stdio.h>
int i=5;
void main()
{
        printf("i=%d\n",i);
        {
                int i=3;
                printf("i=%d\n",i);
                printf("::i=%d\n",::i);
                {
                        int i=1;
                        printf("i=%d\n",i);
                        printf("::i=%d\n",::i);
                }
```

```
        }
        printf("i=%d\n",i);
}
```

从上面的程序结构来看，我们知道，大括号的作用不一定是什么复合语句，而是限制变量作用域范围的标识符。（牛刀的理论够新鲜的吧，呵呵，上面的程序运行结果大家自己去电脑上运行一遍便知）

好了，我们还来那一个类的例子吧，那么如果我们的类也在 void main() 函数外面定义会出现什么情况呢？

进入了构造函数
进入了 main 主函数
正在退出 main 主函数
进入了析构函数

怎么样？程序还没有进入 main 入口函数，就有程序开始执行了，好了，我们做一个 MFC 的单文档的程序，假如名称叫 MFCTest ，那么你会看到程序生成的类中有一个类 CMFCTestApp ，是吧？这个类的老祖宗就是用的这种技术，在程序还没有执行 WinMain 函数时，就提前执行了，而下面那几个类都是他的类成员，所以那个类就一下子变成了一个全局的类，于是就管理着我们整个的窗口，并且提供了好多的函数，去得到一些我们所需要的东东。当然，我们可以去模拟一个，怎么模拟呢？ MFC 最后一个为 C，所以她提供的类都是以 C 开头的，我们以后要给大家介绍的类都是我们自己去测试（Test）用的，我们就都以 T 开头吧，我们建立一个窗口程序（注：不是 MFC 了啊，当然，也不是控制台了，想必怎么生成大家都会做吧？不多说了。）

在程序中写入一个入口函数： WinMain

```
#include <windows.h>
class TApp
{
public:
        TApp()
        {
                MessageBox(NULL,"我在还没到入口函数之前就执行了，呵呵","函数执行",MB_OK);
        }
        ~TApp()
        {
        }
};
TApp theApp;
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
        return 0;
}
```

怎么样？程序还没有执行，我们的 theApp 就提前开道了，呵呵。这有什么好处呢？我们的一些东西就可以放在里面了，例如，我们可以直接将类的各种

函数都塞进一个函数中去，看程序：

```
#include <windows.h>
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);
class TApp
{
public:
        void Init(HINSTANCE hInstance)
        {
        WNDCLASS ws;

        ws.cbClsExtra=0;
        ws.cbWndExtra=0;

        ws.hbrBackground=(HBRUSH)GetStockObject(GRAY_BRUSH);
        ws.hCursor=LoadCursor(NULL,IDC_ARROW);
        ws.hIcon=LoadIcon(NULL,IDI_APPLICATION);

        ws.hInstance=hInstance;
        ws.lpfnWndProc=MyProc;

        ws.lpszClassName="MyWindow";
        ws.lpszMenuName=NULL;
        ws.style=CS_HREDRAW | CS_VREDRAW;

        RegisterClass(&ws);

        HWND hwnd;
        hwnd=CreateWindow("MyWindow","牛刀制作的窗口",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAU
LT,
                NULL,NULL,hInstance,NULL);


        ShowWindow(hwnd,SW_SHOWNORMAL);
        UpdateWindow(hwnd);
        }
};
TApp theApp;
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nCmdShow)
{
        theApp.Init(hInstance);
        MSG msg;
```

```
        while(GetMessage(&msg,NULL,0,0))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return 0;
}

LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        switch(uMsg)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }
        return DefWindowProc(hwnd,uMsg,wParam,lParam);
}
```

效果一样哟，当然，我们封装的还不够，如果封装的再好一点的话，那么我们就定义一个 theApp 就为我们产生一个窗口，再定义一个就再产生一个，很简单哟。

当然 MFC 中的窗口不是由 theApp 产生的，而是由 CWnd 去产生的。当然，讨论到那儿我们再详细与大家说。

那么接下来我们准备干什么呢？我们再建一个类，将我们的消息循环包装一下下，呵呵，我们建立一个什么类呢？一个 TDC 一个是 TPaintDC 我们设置 TPaintDC 为 WM_PAINT 消息中专用，而 TDC 为非 WM_PAINT 消息用，好喽，我们试看看哟（但注意了，我们为了将相应的生命周期压缩在消息的一个循环中，我们在相应的消息循环中加上花括号（什么意思应该明白吧？压缩对象的生命周期）。我们假设定义了一个 TPaintDC 类：

```
class TPaintDC
{
        HWND m_hwnd;
        HDC m_hdc;
        PAINTSTRUCT m_ps;
public:
        TPaintDC(HWND hwnd)
        {
                m_hwnd=hwnd;
                m_hdc=BeginPaint(m_hwnd,&m_ps);
        }
        ~TPaintDC()
        {
                EndPaint(m_hwnd,&m_ps);
        }
```

```
};
```

大家看我们在干什么，一面存储物资，将 m_hwnd 得到了，一面生成物资（都是私有的），并且最绝的是我们在构造函数中将 BeginPaint 函数给使用了，而析构函数中调用了 EndPaint ，绝了，我们在消息中响应就显得格外简单，连取得 HDC 的过程都省了，什么原因？很简单啊，类给我们自动做了，看调用：

```
case WM_PAINT:
        {
                TPaintDC dc(hwnd);
        }
        break;
```

简单是简单，那这个 dc 我们怎么用呢？那就更简单了，我们定义什么函数有什么函数：看程序

```
void LineTo(int x,int y)
{
        ::LineTo(m_hdc,x,y);
}
```

请大家注意，如果在类中定义与基础的 API 函数相同的函数，如果要调用系统的 API 则需要加作用域标识符 :: 否则系统会认为默认调用自己，构成递归调用，千万在应用过程中注意。

好了，那么我们调用看一看：

```
case WM_PAINT:
        {
                TPaintDC dc(hwnd);
                dc.LineTo(200,300);
        }
        break;
```

效果真的是出色极了，当然，我们上一讲的所有函数都可以封装进上面的 TPaintDC 结构，这样一来，我们调用函数就显得格外方便，至少是少了一个参数，并且只要一打 dc. 马上后面就由系统可以根据我们的输入为我们自动填充了，真是方便。

事实就是如此，系统也正是利用这处面向对象的办法，给我们定义了好多好多的这种结构，使得我们的使用会更加方便，更加快捷。

提示 MFC 单文档程序中的 OnDraw 函数就是系统响应 WM_PAINT 消息而调用的一个函数，并且函数就给我们传进了一个指针 CDC *pDC 也就是说，大家用这个指针就可以自由地操纵背景的画图了，很是方便。大家上机做一个 MFC 就可以直接到相应的 View 类中，找到 OnDraw 函数，打 pDC-> 一试便知，可见我们在 MFC 中想对一些指针操作，那可是方便多了。

**微软积木〇七三：多重调用**

大家还记得上一讲我们说的那个 theApp 吧？虽然我们的功能可以实现，但事实上 MFC 却并没有帮我们将产生窗口的东东放在对应的 App 类中，而是放在了 Wnd 类中，而 App 类主要用来管理其他类来用的。

当然，这些细节我们以后再具体与大伙作详细讨论，毕竟我们还有好多的理论没有深入熟练的掌握，我们还需要知道好多好多的原理。

那么这一讲我们讲什么呢？我们用控制台应用程序吧，但我们包含一个 Window.h 头文件，当然我们的 stdio.h 文件也别忘了，毕竟我们在做 dos 程序。

那我们想做什么呢？我们同样做类，用类以我们的几个 Windows 常用的结构做一件外套。

什么结构的，最常用的就是：
POINT 类了，再一个就是 RECT 类，那么我们当然可以这样定义一个类：

```
class TPoint
{
        POINT pt;
public:
};
```

我们当然可以这样定义，但如果我们知道 POINT 结构中只有 x,y 两个变量的话，我们为什么不这样地定义：

```
class TPoint
{
        int x,y;
public:
};
```

这样不是更通俗易懂吗？当然，我们对于矩形区域来说，我们也可以按如下的定义：

```
class TRect
{
        int left,right,top,bottom;
public:
}
```

事实就是如此，我们可以很容易地来定义这些类，然后，然后嘛，当然加函数，使得这些类能够自动地去做一些事情。

有关自动做事的话，我们上节也已经用到过了，那就是构造函数与析构函数，可以做到，例如，如果我们类这样定义的话：（注：为了显示方便，我们在类中都定义了 print 函数：

```
class TPoint
{
        int x,y;
public:
        TPoint()
        {
                x=25;
                y=32;
        }
        void print();
        {
                printf("TPoint  x=%d,y=%d\n",x,y);
        }
};
```

上面的 TPoint 就是一个构造函数，由于肯定没有返回值，所以返回值类型都不需要有，并且是自动执行（方便啊，呵呵）。当然，我们一调用就显得格外地简单喽：

```
void main()
{
        TPoint pt;
        pt.print();
}
```

结果当然大家也知道，（不知道自己去运行一下下，呵呵）。

那么我们就寻思了（注：大家看书学习某个知识点不要认为这是知识点，你们可以想想微软那些人也是人，他们提出概念是不可能是个摆设的，肯定是出于某种目的，我们以后的实践也基于这一点原因与大家相继展开），那么如果我们要定义一个坐标为 (4,5) 的点怎么办呢？

于是，我们的第一个概念就诞生了，有人称之为多态性，其实牛刀这儿就称之为“多重函数”，或者称之为“多重调用”，什么意思呢？也就是说，一个函数名，可以定义出不同的函数，例如我们的构造函数就可以实现多态，我们怎么着再定义一个：

```
        TPoint(int a,int b)
        {
                x=a;
                y=b;
        }
```

怎么样？我们再看主函数部分，我们再加入一个：

```
void main()
{
    TPoint pt;
    pt.print();
    TPoint pt1(4,5);
    pt1.print();
}
```

怎么样？我们第一个函数没有传进参数，而第二个传进了两个参数，那么我们类真正构造时就是调用的不同的构造函数，那么好，再来，（俺就不信，哼!!!）：

```
    TPoint(TPoint & pt)
    {
        x=pt.x;
        y=pt.y;
    }
```

呵呵，够舒服的吧，我们不需要用：

```
pt2.x=pt1.x;
pt2.y=pt1.y;
```

这样的代码了，我们只需要在定义的时候做一个函数的传入就行了：

```
    TPoint pt2(pt1);
    pt2.print();
```

上面这种构造函数由于是将相同的类复制了一下，所以也叫复制函数，英文名称　Copy 所以这种函数在一些书上也称之为“拷贝构造函数”，当然喽，拷贝构造函数是默认的，也就是说这一个定义如果真的是想将一个与自身一模一样的类复制过来，那么这个函数定义不定义都可以，大家可以将上面函数删除了，再运行，一样的不报错，还可以正确运行与执行。

当然如果你要实现某种邪恶的目的的话，那么你的这个函数就一定要有了，举个例子吧，看程序：

```
    TPoint(TPoint & pt)
    {
        x=1+pt.x;
        y=2*pt.y;
    }
```

完蛋了，你的程序执行的结果会让你莫名其妙（当然了，对于你自己就是设置这个函数的人来说，你一定不会觉得有什么不妥的地方）。

```
TPoint   x=25, y=32
TPoint   x=4, y=5
TPoint   x=5, y=10
```

当然喽，不但拷贝构造函数系统为我们自定义了，就连等于号系统都为我们重载了，所以我们想将一个点的值给另一个的话，那尽管使用下面的程序：

```
    pt2=pt;
```

pt2.print();

所以这种多态性表现在函数方面就是——重载。也就是一个函数，可以有多个不同的定义，就像上面的等号也被系统默认地重载了，当然我们以后学过符号重载时（如果大家看过牛刀以前的教程，其实也讨论过了，呵呵），那么这种重载同样可以实现我们一些邪恶的用意，呵呵。

所以我们编程序的另一条原则就是合理原则。也就是说，我们做程序尽量不要去改变别人的习惯性的思维，如果用你的程序做出来 2+3=6 那么别人的肺会被你气炸的，呵呵。

说句题外话，就有人用过一种计算器的，10*10=110 当然喽，这种浮动是可控的，并且有遥控器，有 ABCD 四键，分别上浮 10%,20%,30%,40%，如果和你算帐的人用这种计算器算出一个 1527432.1526 那么你会怀疑这个结果有毛病吗？于是你就将钱给付了。所以我们做事真的要凭良心，不能做出这种伤天害理的事来。

听说过去的老式杆秤上是有三个字的，一两的位置上是“福”，二两的位置上是“禄”，三两的位置上是“寿”，所以古代人卖菜卖东西是不短秤的，过去人常说，短一两我们福就没有了，短二两我的禄就没有了，短三两我连寿都没有了，谁还乐意短你秤，所以古代的人卖东西如果事后发现短秤他们肯定是想方设法地找到你将短的那一部分给你补上，并且还补得很多，并且会跟你说，我不能因为短你这三两秤将我的寿给短了。

当然现在的人就不信那一套了，但我们做事最好还是要将良心放正，做事要先学会做人。

好像扯远了，呵呵。
当然喽，我们还可以定义如下的构造函数：

```
TPoint(LPARAM lParam)
{
        x=LOWORD(lParam);
        y=HIWORD(lParam);
}
```

这下可好了，我们真正去做程序的时候，我们的 TPoint 变成万能的了，不管传什么参数都可以通吃一切，当然喽，对于 TRect，你也可以很容易地用：

```
rect2=rect1;
```

轻易地将四个参数全部都传递过去，而不需要定义任何函数（还是那话，等号在类中是默认重载的）。

好了，我们定义几个 TRect 的构造函数，多重载几个，大家看感觉怎么样？

```
class TRect
{
        int left,right,top,bottom;
public:
        TRect()
        {
                left=right=top=bottom=0;        //可以这么用的，别忘记
了
        }
        TRect(int a,int b,int c,int d)
        {
                left=a;
                top=b;
                right=c;
                bottom=d;
        }
        TRect(TPoint pt1,TPoint pt2)        //请大家注意了，将 TPoint
中的 x,y
        {
    //改为  public  类型，不然会报错
                left=pt1.x;
                top=pt1.y;
                right=pt2.x;
                bottom=pt2.y;
        }
};
```

感觉怎么样？这就是矩形与点的关系，当然，我们可以为矩形做出好多的函数出来，以实现这种矩形的智能化，举个简单的例子吧（注：举的这个例子就不是重载了，呵呵，我们看看类都能给我们干些什么：（在矩形中定义）

```
TPoint * CenterPoint()
{
        TPoint *pt=new TPoint;
        pt->x=(left+right)/2;
        pt->y=(top+bottom)/2;
        return pt;
}
```

怎么样？我们以后要知道中心点坐标时，只要定义一个 TPoint *pt; 然后执行 pt=rect.CenterPoint(); 就一切 OK 了。这就是类的智能处理的能力。

虽然说是智能，但也只能说感觉上是智能了，感觉上我们定义了一个矩形，以往我们做程序的经验就是：你告诉我们的上下左右，我算算你中心坐标是什么，而现在你只需要问这个矩形类，你的坐标是什么呢？告诉我，呵呵，就是这种感觉。

可以说，类中的函数封装得越多，功能封装得越多，我们以后的开发就越简单，正是基于这一点，微软才在 MFC 中给我们做了大量的封装，使得我们做程序真的好像是在搭积木一样。

当然喽，重载与多态性不仅仅是在类中，正常的函数我们也可以实现重载，而系统在我们调用时，自动根据我们的参数情况与一些实际的情况来决定我们调用哪一个函数，正因为这种多个函数一个名称，或者说一个函数的多种状态，所以我们说这个函数是多态的。多态使得我们手上有什么东西就可以自由地使用什么东西。

大家还记得我们以前的画线类吗？需要传递三个参数是吧？而如果我们定义一个类，这个类中就已经将 HDC 保存的话，那么我们的画线函数就只有两个参数，可以这样调用：

```
CDC dc;
dc.LineTo(x,y);
```

当然也可以这样调用：

```
dc.LineTo(pt);
```

大家还知道矩形函数吗？可以很方便使用以下的三种函数：

```
dc.Rectangle(x1,y1,x2,y2);          //是这我们以前学过的，只是少了一个设备描述表句柄，也就是纸
dc.Rectangle(pt1,pt2);                  //用两个点来表示
dc.Rectangle(rect);                  //甚至可以这样来做
```

那这是怎么做到的呢？重载，而这种技术是什么技术呢？多态。

也就是说，定义了好多的函数，我们用哪一种？随便。

那大家以前还记得我们做过一个画饼的函数吗？ Pie 好像有九个参数是吧？而在新的我们封装了的 CDC 中可以只有三个：

```
dc.Pie(rect,pt1,pt2);
```

简单得多吧？事实就是如此美妙。

当然，最后要说一下的是，重载并不等于重名，计算机还是不允许重名的，例如你如果定义下面两个函数:

```
int max(int a,int b);
int max(int x,int y);
```

那是不允许的，也就是说，你调用时，不要说让计算机知道了吧，就你自己能不能知道 max(3,5); 调用的是哪一个函数呢

再一个就是合理，有些书上会给出一些重载的规则，其实有必要吗？我们看下面两个函数：

```
int f(int a,int b=5);
```

int f(int a);

如果有调用 f(21); 那么调用的是哪一个函数呢？你能判断吗？如果你不能判断的话，你怎么叫计算机去判断？一个很简单的道理，计算机是人造的，人做不了的事，计算机肯定是做不了的，再来：

int f(int a,int b);

floast f(int a,int b);

如果调用 f(34,56); 你知道调用的是哪一个函数吗？显然也不能知道吧，所以计算机也拒绝接受，这就是很简单的判断准则，而上面这种种的函数，我们就称之为“重名”了，而不是重载。

也就是说，你传进的参数要让计算机明白你的用意，你是想重载的哪一个参数。

从这一个角度来说，其实重载还是不同的函数。

我们的计算机是可以定义函数原型而不调用函数的，看一段程序：

```
#include <stdio.h>
void f(int a,int b);
void f(float a,float b);
void f(double a,double b);
void f(char a,char b);
void main()
{
        f(1,2);
        f(1.0,2.0);
        f(1.0f,20.f);
        f('a','b');
}
```

这段程序在编译的时候没有任何问题，但在链接时报错了：

```
main.obj : error LNK2001: unresolved external symbol "void __cdecl f(char,char)" (?f@@YAXDD@Z)
main.obj : error LNK2001: unresolved external symbol "void __cdecl f(float,float)" (?f@@YAXMM@Z)
……
```

大家看后面的名称：

f@@YAXDD@Z

?f@@YAXMM@Z

是不是很好玩，我们的函数不是原来的函数了，事实也是如此，其实在 C++中，我们每定义一个函数，在编译时，函数名就被我们的编译器通通地给“偷换”掉了，于是，我们看上去好象是调用的同一个函数，其实调用的仍然是不同的函数。所以从这个角度上来看，其实我们的计算机中其实是不存在真正的“重载”的，那种重载的概念，只不过是微软耍的一套花活罢了，当然即便是这样，也给我们编程带来了不少的方便，你说是不？

其实如果看过牛刀以前的教程的朋友一定还记得牛刀曾经说过一个事实，事实上在 VC 中根本就存在什么“类”，什么“结构”之类的东东，那些只不过是概念，而编译器其实是在根据概念提供的一套规则去将我们的程序编译成了其实什么结构都不存在二进制代码，仅此而已。

## 微软积木〇七四：祖孙奇缘

如果大家知道什么是第三者的话，那么微软就更恶心，通常的第三者也顶多破坏别人的夫妻关系也就罢了，而微软给我们提供的类竟然可以破坏别人的父子关系，在父子关系之间插一脚，硬活生生地将人家父子关系变成了爷孙关系，而他呢成了人家孩子的爸爸，成了人家爸爸的孩子，硬生生地将人家的“父子关系”变成了爷孙关系，这是怎么回事？看程序吧，在微软中的类中，有一种方法，可以定义子类：

```
#include <stdio.h>
class A
{
public:
void Say()
{
  printf("ASay:我是你爸爸。\n");
}
};
class B:public A
{
public:
void Ask()
{
  printf("BAsk 请问你是谁？\n");
}
};
void main()
{
B b;
b.Ask();
b.Say();
}
```

这里就讨论了类的继承了，原来这一节内容准备放后面讲的，可是如果那样的话，我们 MFC 的一些东东就需要往后拖，于是我们就提前给大家说了吧，类的继承就是在类名后加一个冒号，然后表示怎么个继承关系，再就是子类，也就是说，上面的 B 是 A 的子类，那么微软为什么要这样定义呢？为的就是能够拆掉微软的 MFC，可能你还不太明白，那我们还是先讲讲那父子俩的事吧。你看由于 B 是 A 的儿子，所以我们在定义 B 后，竟然可以在 B 定义的变量 b 中去调用 A 中的函数 Ask ，这就是继承，这是函数的继承，当然并不是 A 的什么函数 B 都可以继承的，那我们以后再说，大家看，这时突然跑来一个 C ，恶狠狠地对那个小孩子说：“我才是你爸爸”，会出现什么情况呢？

```
class C:public A
{
```

```
public:
void Say()
{
  printf("CSay 我才是你爸爸，哼!!! \n");
}
};
class B:public C //注意这儿也要改，你说 C 恶心不，给人家的爸爸给换了，呵呵
{
……
```

你说 C 恶心吧，愣是将人家的爸爸给换了，呵呵，这个程序调用的结果是什么呢？

```
BAsk 请问你是谁？
CSay 我才是你爸爸，哼!!!
```

你瞧瞧，你瞧瞧，收了个儿子，找了个爸爸，还不让爸爸来说话了，于是，这父子俩就理所当然地变成了爷孙俩，并且爷爷说话的权利都没有了，那么我们看，我们在 B 中也定义一个函数：

```
class B:public C
{
public:
void Ask()
{
  printf("BAsk 请问你是谁？ \n");
}
void Say()
{
  printf("BSay 我就是我，你们吵什么吵？再吵我解雇你们，哼哼哼!!!!!!\n");
}
};
```

大家再运行程序，看结果：

```
BAsk 请问你是谁？
BSay 我就是我，你们吵什么吵？再吵我解雇你们，哼哼哼!!!!!!
```

是不是很霸道？这种儿子一说话就不让父亲说话的霸道行为，我们称之为“函数的覆盖”，也就是子类函数覆盖父类函数。

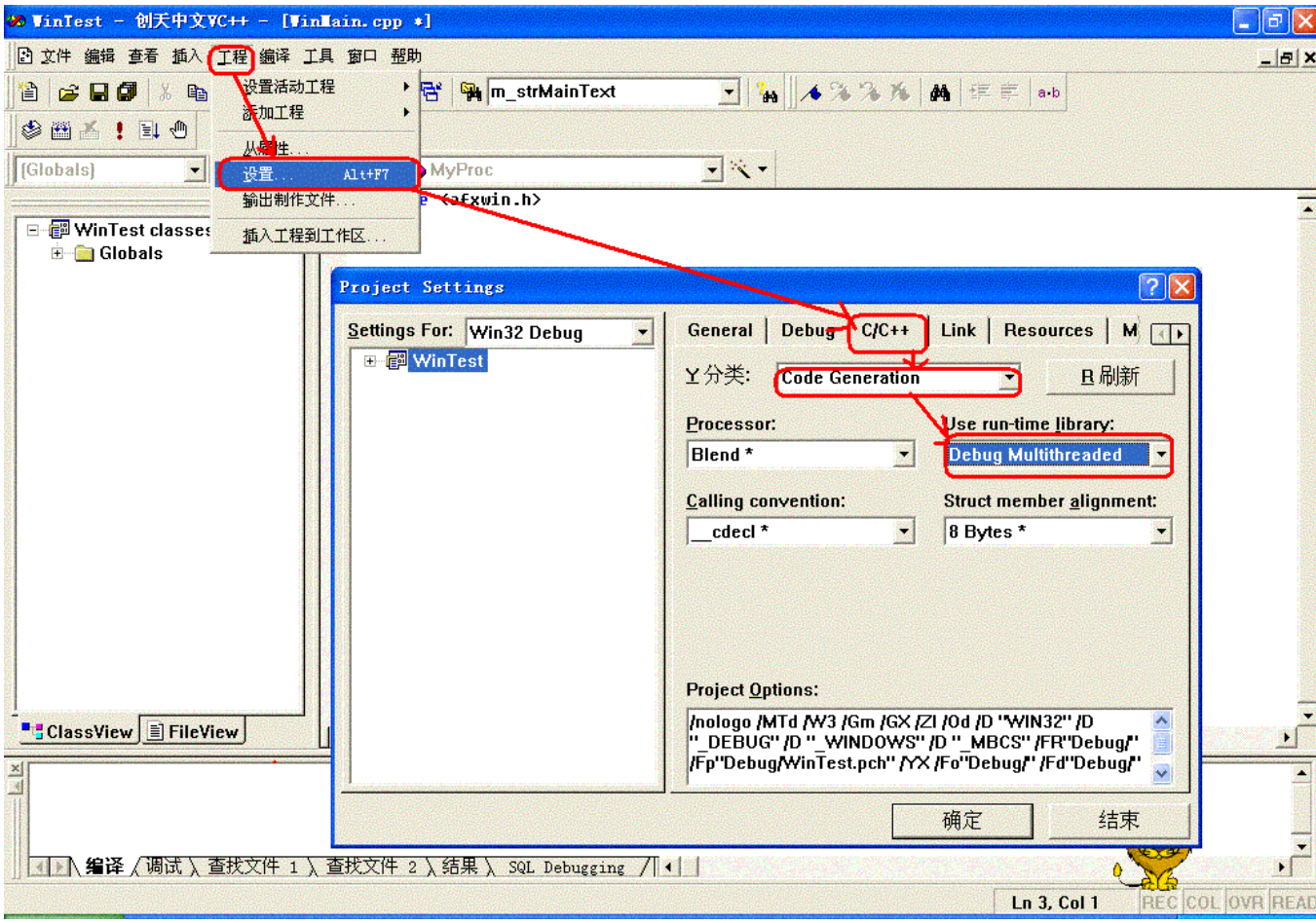
请大家注意，这里是同名函数，不是函数的重载，我们的同名函数不能共同出现在一个函数定义范围内，但不同的范围还是可以定义同名函数的，而在哪个区域定义的函数，就先调用定义区域的函数，这种现象我们称之为函数的覆盖原则，也就是说，一旦子类定义了某个函数，那么父类函数也就只能认命了，父亲这位置不好当啊，现在父母说话还管啥用，呵呵。

那么我们这里讨论的祖孙、父子，有什么用？当然有用了，父亲的相同函数是被覆盖了，可父亲与我们的不同函数不就继承下来了吗？于是，在类的继承中，就为我们提供了一套代码重用的机会。

那么，我们就来重用一下别人的代码，这次当然我们建一个 Windows 程序，并且在程序中包含进头文件 afxwin 什么？这是什么头文件？呵呵，大家

先包含进来就行了，我们下面做代码需要用到：

对了，和大家说一下，由于我们做一个 Windows 程序是单线程的，而 MFC 程序是多线程的，所以我们需要在 VC 中按下 ALT+F7 或者按菜单：工程/设置... 也弹出这个选项卡，我们点 [C/C++] 选项卡，设置按下图，在分类中选择 Code Generation ，而在上面第一个下拉菜单中选择 Debug Mulitithreaded ，选择多线程，先好了就按确定。




为什么要这样设置呢？这是 MFC 的需要，那是多线程的运行程序，好了，设置好了，就可以根着牛刀来做程序了：

先来定义两个类：

```
#include <afxwin.h>
class CMyFrame:public CFrameWnd
{
public:
CMyFrame()
{
  Create(NULL,"牛刀教我做 MFC");
}
};
class CMyApp:public CWinApp
{
```

```
public:
BOOL InitInstance()
{
  m_pMainWnd=new CMyFrame();
  m_pMainWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd->UpdateWindow();
  return TRUE;
}
};
CMyApp theApp;
```

怎么样，就输入上面的两个类，大家编译一下，运行，看到什么了？

怎么样？那个个窗体是怎么来的呢？看看，我们定义了一个 theApp 是不？大家还记得我们以前曾经和大家讨论过，这种全局函数是不需要进入 WinMain 函数就可以运行了是不？但由于我们这个类是有老爸的，所以他的老爸也就提前执行了，于是就进入了全局函数的构造，并且由老爸进入了 WinMain 函数，所以这些当儿子的就不需要进入 WinMain 就将窗体给整出来了，呵呵，怎么整的？大家看，不是在 InitInstance 函数中新建了一个 CMyFrame 吗？那个类的老子就是 CFrameWnd 啊，那东西就是建立一个窗体的，所以我们的窗体出来了。看样有个好老爸那就是省事啊，呵呵。好多的事都给我们做了，我们只需要在建立类后，新建一个窗口，然后 Show 出来就一切 OK 了。这就是很厉害的父子关系，呵呵。

好了，我们现在明白了一点，为什么我们要有父子关系，就是要将好多的代码放在父亲那里，甚至爷爷那里，我们要的时候，只需要定义一个类，就一切OK了，而什么消息循环啊，什么事都不需要我们干。当然喽，如果我们什么事都不干，那么我们岂不是也就定义出一个窗体出现吗？这款软件还有什么用？于是，父亲的一些函数，我们只要不定义，就用老爸的，如果我们定义了，哼哼，那就用我们的了，好阴险啊。

那我们还是还想想我们的消息循环吧，其实很简单，老爸将消息循环都做了，也就是默认的消息循环，只要我们不用的消息，就用老爸的，而只要我们定义了，就用我们的了，这就是函数覆盖的魅力。

事实就是如此，但有成百上千的消息要处理，你让你老爸去先将这些事情干了，你想累死你老爸啊，呵呵，何况如果还没执行程序，就先定义那么多的函数，那我们的软件做出来一定会很大，真的会很大，所以微软使用了另一套技术来解决了这套问题，我们称之为“消息映射”。

什么意思呢？大家还记得我们曾经做过消息程序吗？我们并没有定义所有的消息，而是在 case 后面加一个消息，然后将消息需要做的函数就定义在下面，也就是说，我们添加一个函数，就只需要这样添加就行了。

可老爸不乐意了，你想不通过继承就动你老爸啊？门都没有，但老爸还是比较开明的，他提供了一些宏，而不是函数。

请大家注意了，宏在编译过后是不占空间的，而函数编译过后是占空间的，所以说，我们定义再多的宏，也用不着去担心会增加我们的软件大小。

也就是说，老爸和儿子商量了一件事：“儿子，你听老爸说：……”：

老爸呀精力也有限，你给老爸帮帮忙，老爸给你一些宏，你调用就去捧捧场啊～～～饿饿饿哩咯哩饿～～～捧捧场。

也就是什么意思呢？父类并不提供所有的类，但提供一些宏，并且将定义类的所有代码都放到宏里面，再给宏做一个代号，如果子类一旦使用了某个宏，父类在我们 MFC 编译的时候，就由编译器根据子类的宏，在父类那儿自动生成相应的 case 代码去拦劫需要响应的消息。

明白了吗？消息机制是让我们当土匪，而消息映射机制就是我们想要劫谁，就告诉老爸，让老爸给你去当土匪去了，呵呵。而你就无情地对老爸说，老爸劫来的东东全给我处理，什么意思呢？函数覆盖啊。

也就是说，消息映射机制目的是为了让老爸减轻负担，还要告诉老爸，我要拦劫什么消息。于是，我们就干个我们在这消息中想干的事就一切OK了，而我们处理的这个消息，竟然被老爸给处理过了，给我们的竟然是处理过的成品，那给了我们什么呢？看段代码吧：

我们在 CMyFrame 函数中加入下面的代码：

```
afx_msg void OnLButtonDown(UINT nFlags,CPoint point);
DECLARE_MESSAGE_MAP();
```

大家看前面的 afx_msg 这是过去老版本的 MFC 库遗留下的东东，我们使用消息映射时用，大家知道就行了，以后我们一般情况下不主动地手工去添加这些代码。

而这里就照猫画虎就行了，这里的 afx_msg 其实是一个宏。

而下面的那一长串的大写，显然也是一个宏了，所以我们后面千万不要加分号，呵呵。

而大家看后面那函数，似乎像按下左键，是不？

当然，那是在类里面，而要通知父亲，我要调用哪个消息，就需要在类外面定义了，在类外面作如下定义：

```
BEGIN_MESSAGE_MAP(CMyFrame,CFrameWnd)
ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

怎么样？这里就是通知了父亲了，告诉父亲，我们这里需要调用 WM_LBUTTON 消息，当然，你可以说，不是没有这个消息，而是这个宏就为我们定义了这个消息，我们看定义吧：

```
#define ON_WM_LBUTTONDOWN() { WM_LBUTTONDOWN, 0, 0, 0, AfxSig_vwp, (AFX_PMSG)(AFX_PMSGW)(void (AFX_MSG_CALL CWnd::*)(UINT, CPoint))&OnLButtonDown },
```

怎么样？严然就是一段代码啊

当然，我们可以想像，前面的 BEGIN... 与后面的 END... 那一段，肯定也是一个宏，并且这个宏与里面所定义的东东放在一块儿，构成一大段的代码，由于这一段代码是全局的，所以这是一段全局的函数，而我们的父类就去调用我们这一段全局函数就实现了消息拦劫了，呵呵。

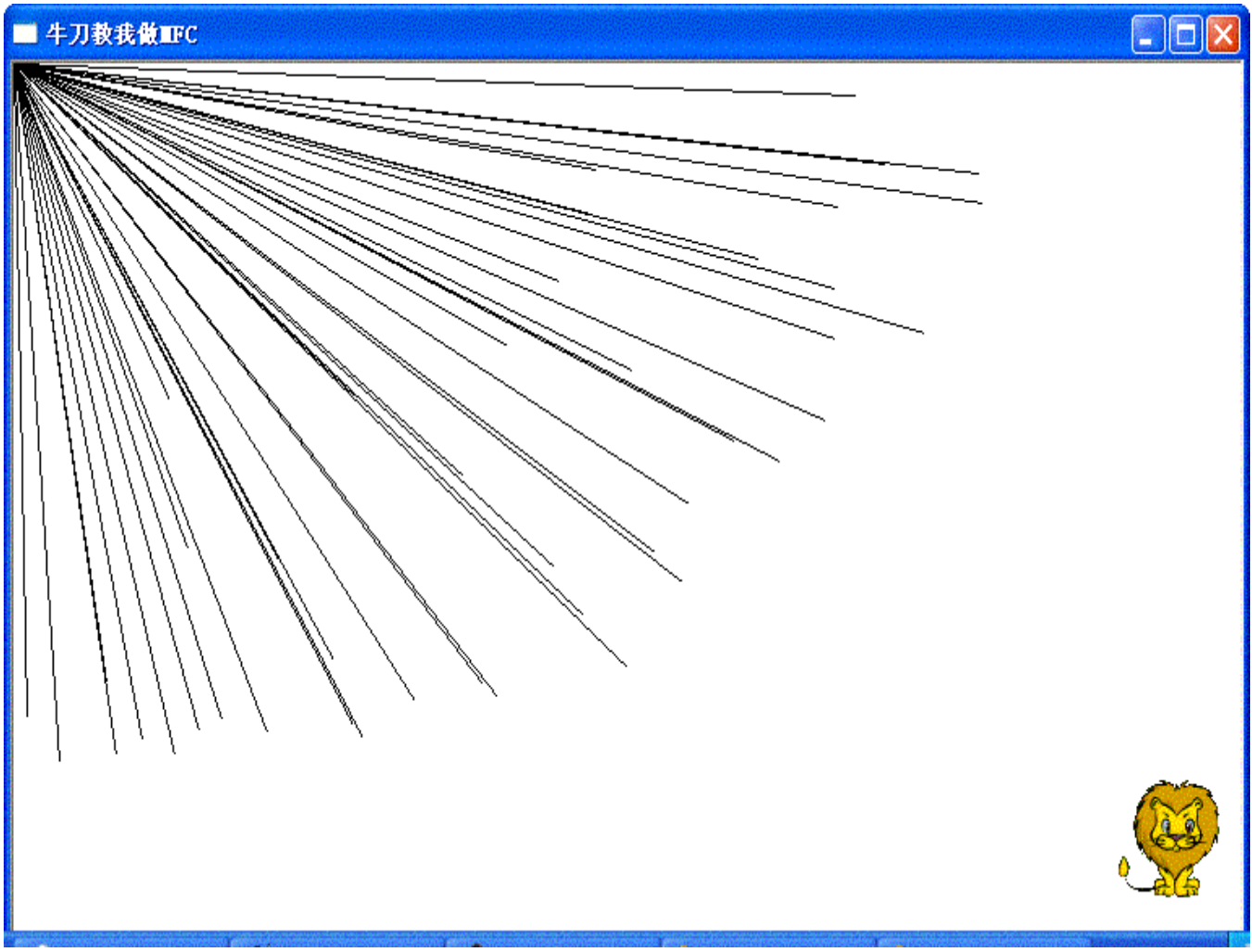
当然，如果你有兴趣，你可以将上面的几个宏都找出来，然后拼起来，看看到底是什么程序，这个程序通知了父亲去打劫什么消息，呵呵，好了，既然函数我们都定义了，那我们就去实现这个函数吧。

我们类中的函数可以放在类中实现，当然也可以拿出来，而在类中仅仅定义一个函数原型，那么我们将那个函数拿出来的结果就是：

```
void CMyFrame::OnLButtonDown(UINT nFlag,CPoint point)
{
}
```

怎么样，还是自家父亲好啊，呵呵，给我们的东东竟然不是 wParam 和 lParam ，而是处理好的 CPoint 类，用起来舒服啊，我们再来定义一个 dc ，去画几条线看看吧，呵呵：

```
void CMyFrame::OnLButtonDown(UINT nFlag,CPoint point)
{
CClientDC dc(this);
dc.LineTo(point);
}
```




是不是感觉比我们以前做消息循环的那一段省事多了(注:函数实现部分)?是,不错,函数实现部分的确省事了,但因为那是 CClientDC 为我们封装了大多数的东东,那我们却增加了更多的负担,例如上面宏的定义。

别忙,这只是我们为了帮大家理解 MFC ,其实以后我们自己根本不需要做那些东东,而是由系统统统给我们做了。好了,我们这一讲不与大家做过份地深入。

好了,父子关系,这是第一步,父子关系给我们提供了一种重用代码的机制,有了父子关系,微软给我们提供的一套 MFC 库,我们就可以用继承的办法全部给继承下来,而我们的类却不仅仅是需要继承,我们的类还需要去改变父亲的一些行为。以达到我们某种邪恶的目的,那么怎么去改变呢?

我们下一讲再详细地与大伙慢慢地来。

对了,大家有没有试图去用 CMyApp 去定义两个变量?如果你那样定义的话,那么你的系统一定会一运行就报错,原因很简单,你的程序调用了两次 WinMain 函数,而这是绝对不允许的,呵呵。

也就是说,虽然我们看不到 WinMain 入口函数,但的确是由 CWinApp 给我们去调用过了,所以我们以后做程序的任务就是:去定义我们所需要响应的消息,那就一切 OK 。

对了，顺便提一下，我们做程序中，可以去做第三者，当别人老子，不管怎么不伦理，那是程序，而我们在现实中最好不要去当第三者，不道德，别人能答应，别人的配偶能答应吗？呵呵，开个玩笑，别当真啊。

我们的好多现实中的不伦不类的东西都被微软的那帮人给实现了，你说微软恶心不，呵呵，我们以后还会有好多好多不同的理论，这里有父子，有朋友，有瓦匠，有衣匠……

严然就是一个小社会。

当然，微软的类还有更恶心的，看程序：

```
class father:public A
class mother:public A
class B:public father,public mother
```

你说微软什么意思？

**微软积木〇七五：父子争权**

经常都进不了论坛，建议大家开发我们自己论坛的专业登录软件，否则太不舒服了，呵呵。

这一讲我们不讨论第三者的问题了，但那的确是个大问题，我们以后给大家整 MFC 的时候还要向大家重点推荐，呵呵，当然也是用烂了的技术了。

这一讲我们还是要讨论父子关系，这种家庭关系不好调理啊，儿子夺父亲的权有时候还说得过去，如今大家都是一个孩子，哪能不一言堂呢？但类就不同了，那是多子多福啊，呵呵，开玩笑，但这里还是有问题的，那就是指针调用的时候，我们还看上一讲那个例子吧，这次我们不用第三者了，讲道理不能将道理复杂化，否则我们的道理就很难讲得清楚了。

好了，看代码：

```
#include <stdio.h>
class A
{
public:
        void Say()
        {
                printf("ASay 我是你爸爸。\n");
        }
};
class B:public A
{
public:
        void Ask()
        {
                printf("BAsk 请问你是谁？\n");
        }
        void Say()
        {
                printf("BSay 我就是我。\n");
        }
};

void main()
{
        B b;
        b.Ask();
        b.Say();
}
```

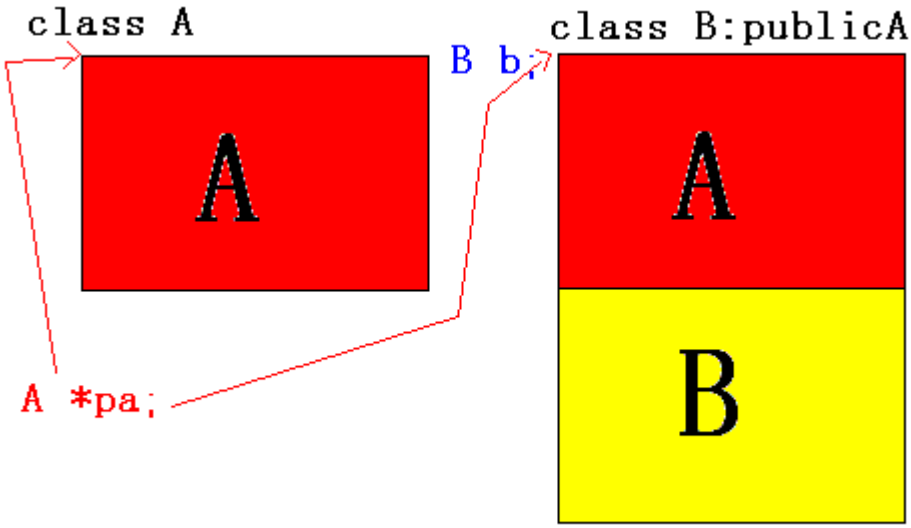
这个程序的运行结果大家都知道吧？

BAsk 请问你是谁？
BSay 我就是我。

子类的 Say 函数覆盖了父了，可大家注意了，我们可以定义一个父子针指向子类，但不能定义一个子类指针指向父类，大家看程序：

```
void main()
{
        B b;
        A a;
        B *pb;
        A *pa;
        pb=&b;
        pa=&a;
        pa=&b;
        pb=&a;
}
```

大家将上面代码敲进计算机，运行看看，只有一个报错，那就是 pb=&a; 而 pa=&b; 为什么没有报错呢？那我们要看类在内存中的“内存模型”




大家看图，B 是 A 的儿子，那么我们要在 B 中能够调用父类的内容，那么 B 在构造自己之前，就先去构造一个 A，构造好后（如上面的红色部分），构造好后，就再加上自己的内容（如自己后加的一些函数啊等等）就形成了右边图的结构。

那么我们定义了一个 A *pa; ，那么我们在这个 A 类型的指针向内存中指的时候，这个指针就会检查内存模型，看是不是有自己 A 这个类，一看果然有 A，于是就指了过去，并且不报错，明白？也就是说，其实我们是用子类，因为子类也构造了父类，所以我们就“骗”过了 pa 指针，使得 pa 认为自己指的正确了，而其实是错误的，也就是说，虽然我们指向了 B，但却不能够调

用 B 类中的 Ask 函数，大家看例：

```
void main()
{
        B b;
        A *pa;
        pa=&b;
        pa->Ask();
}
```

一编译就报错，不信大家去试试看，为什么，因为 A 类中就根本没有 Ask 这个函数。那么 A 中有 Say 这个函数啊，所以说我们可以顺利地调用下面的函数：

```
pa->Say();
```

当然运行的结果想必大伙也知道，应该是：

ASay 我是你爸爸。

什么原因我想大家也知道，因为 A 中有一个函数 Say 而定义的又是一个 A 的指针，所以就理所当然地调用 A 类中的函数喽，这样一样，用指针去调用儿子，想覆盖，那是连门与没有，于是，儿子与老爸闹起了矛盾，牛刀有幸拍了下来，下面是他们的对话：

B: 老爸你也太不像话了，我有 Say，为什么你说话啊。
A: 呵呵，儿子你不讲理啊，人家是找老爸的啊（A *pa;）
B: 不行，为应该覆盖你，凭什么人家一找你，你就越权啊（儿子哭了），呜～～～
A: 哎哎，别哭啊，傻孩子，人家找老爸时还没有你哪（注：人家本来就是指向老爸的指针嘛，怎么还知道你还有个儿子，是哪个儿子？），你不是让老爸犯难吗？
B: 是吗？说的也是，那这样好不好，你那先挂个虚职，到时间人家找你说话，你就先问问我，我不说了你再说，如果我说了你就不许说，怎么样？
A: 好吧，我也想这么做，可你让你老爸怎么知道什么话是你想要说的呢？什么话你不想说呢？
B: 这样吧，儿子出个主意，我想替你说的，前面就加上一个虚标志（virtual），怎么样，到时间你一看到这个标志一定要想起，你还有一个儿子哟？
A: 好吧，好吧，那一言为定，但没有虚标志的，你可不许干预老爸吧，好吗？
B: 那好吧，谢谢老爸。（孩子破涕为笑）

从此，父类中有了一类专门的函数，我们称之为“虚函数”，那是专门为儿子实现的，哎～～～天下哪个父母不疼孩子。

```
class A
{
public:
        virtual void Say()
```

```
        {
                printf("ASay 我是你爸爸。\n");
        }
};
```

当然，微软为了这个虚函数，也费了不小的脑筋，因为老爸也是巧夫难为无米之炊啊!!!（注意，老爸应该是男的，呵呵）。

怎么办，唯一的办法是，遇到这种虚函数，就将所有的本类中的函数与子类中的相同函数都给编译了，并且在运行时，才根据指针指向哪一个函数，就调用哪一个函数中的相应函数，这样一来，的确增加了不小的负担，但儿子的心愿终于满足，我们将这种技术称之为“迟绑定”技术。

其实虚函数的还有一个由来就是上一讲我们所讲的乱伦现象所引起的，看上一讲最后的代码：

```
class father:public A
class mother:public A
class B:public father,public mother
```

你想，如果爷爷那儿有一个 Say 函数，孙子要调用爷爷的函数，那么你说是从老爸那儿继承下来的呢？还是从老妈那儿继承下来的呢？

你说这是多么荒诞的事情啊，呵呵，可是人家微软做到了。

于是爷爷也只好打马虎眼，儿子们（你想想这老东西有多少儿子？），女儿们，们你孙孙要继承的函数，怎么来解决呢？算了吧，我也没办法了，这样吧，我这儿定义一个虚的，只一个啊，到时候，你们见到函数就调用就行了吧，我不管了，咳咳，我累了，休息去了啊，你们聊，你们聊。

你瞧，推得个干干净净，于是，爷爷也只有将 Say 加上了一个标志 virtual ，从而避免了一场纷争。

哎，老爷子难做啊，遇到微软的一帮鬼才们，什么办法，认了呗（注：这一点上来说，人家 Java 做得就很好，面而微软的人只会不断地发明新玩艺，使得一个 C++ 复杂透了，呵呵）。

当然，我们一般情况下，不是万不得已，尽量不要使得我们的类形成这种钻石结构（你瞧人家这名子起的，呵呵，钻石结构）。否则好多问题真的不容易处理。

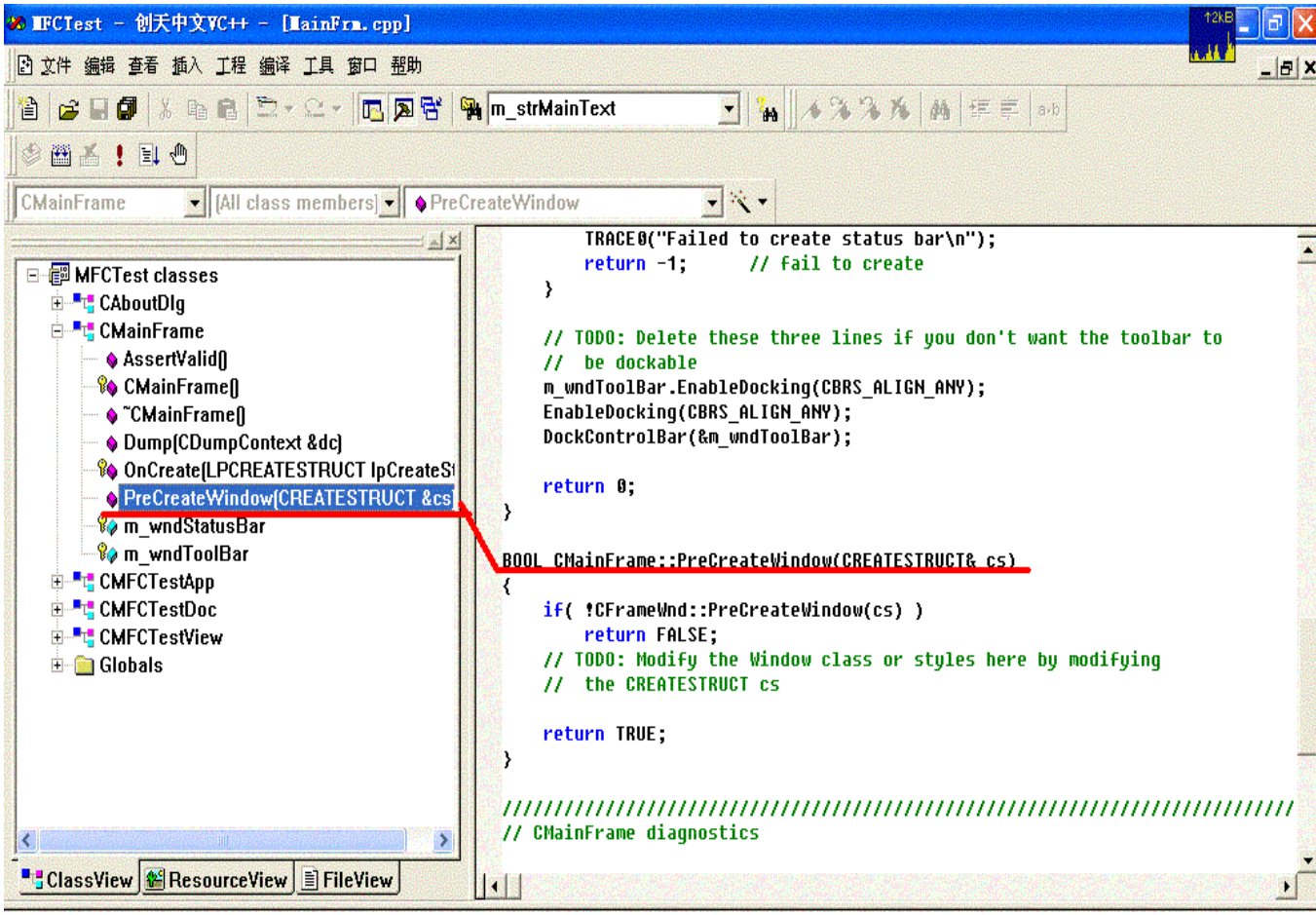
我们一般定义类时，都是树型结构，最多有些第三者跑进去当人家的儿子（其实还认了一个儿子）。哎，这是什么事道啊～～～

说人家的不好也说了，这下就说说人家的好处吧，有些权力下放给儿子，就使得我们老子做的事情上，儿子可以从中去改变：

大家建立一个 MFC 程序，名称设为 MFCTest ，单文档，以后我们绝大多数的讲解都是在这里：

大家看，还是五个类
CAboutDlg
CMainFrame
CMFCTestApp
CMFCTestDoc
CMFCTestView

这次我们打开 CMainFrame 类，看其中有一个函数：




大家看 PreCreateWindow 这就是老爸给我们子类提供的一个虚函数，当父类执行这个函数时，还没有生成窗体，老爸就跑来问儿子：

老爸要生成窗体了，你看看你还有什么要改的么？

儿子如果不改，就通过了，如果儿子想要改，那就简单了，看程序：

```
BOOL CMainFrame:reCreateWindow(CREATESTRUCT& cs)
{
```

```
        if( !CFrameWnd:reCreateWindow(cs) )
                return FALSE;
        cs.cx=20;          //我改你窗体的宽度
        cs.cy=60;          //我改你窗体的高度，呵呵，好了，老爸
        return TRUE;
}
```

那么下面就运行一下程序：

这是什么东西？呵呵，你自己去琢磨，你也太扣门了。

这老爸也真听话。

当然，这是怎么回事呢？大家还记得我们以前的程序吗？

```
class CMyFrame:public CFrameWnd
{
public:
        CMyFrame()
        {
                Create(NULL,"牛刀教我做 MFC");
        }
        afx_msg void OnLButtonDown(UINT nFlags,CPoint point);
        DECLARE_MESSAGE_MAP();
};
```

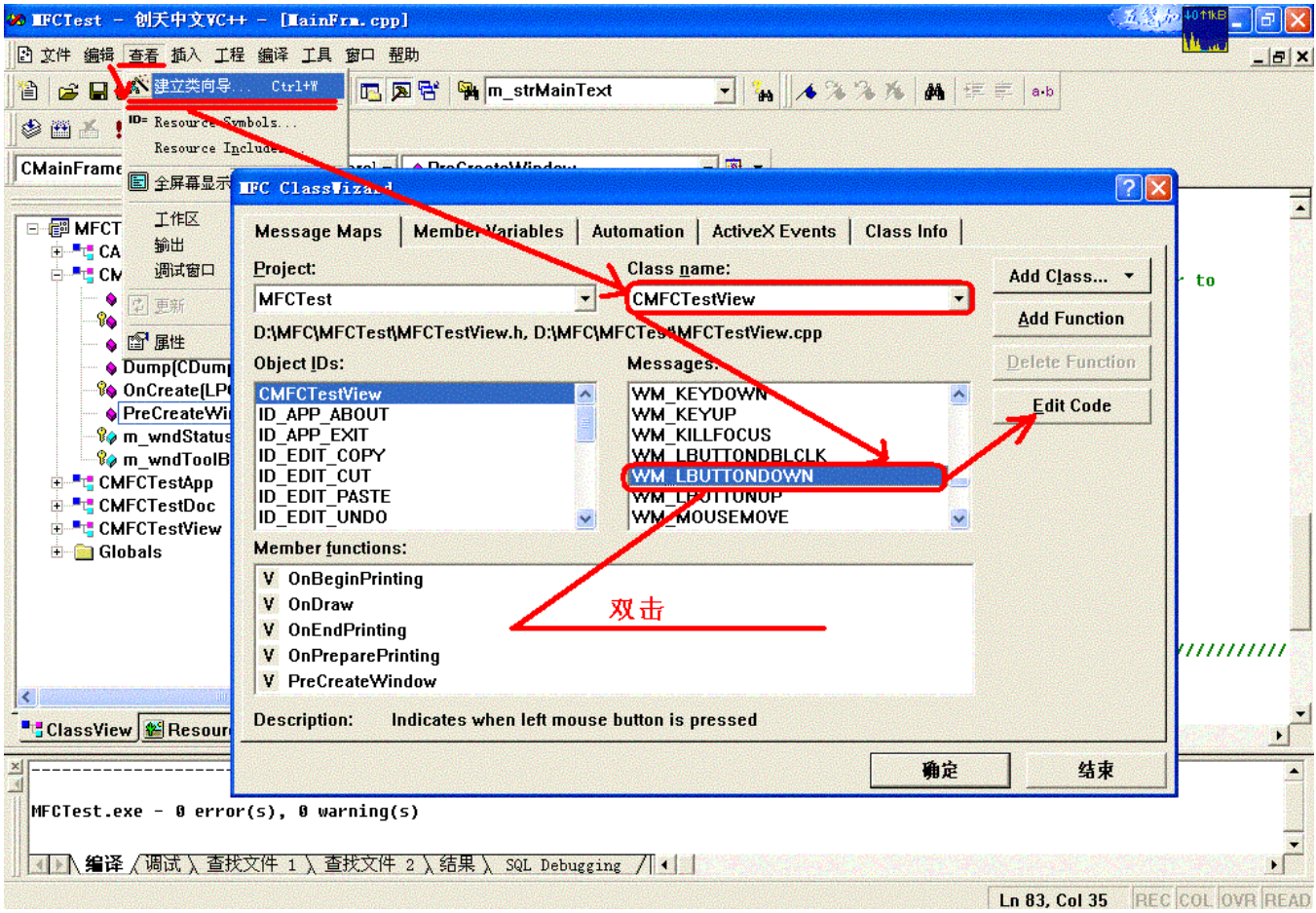
大家知道这个类吧，这个类的父类中就有一个虚函数，叫 PreCreateWindow ，是在窗体还没有建成之前就调用了，所以子类可以在这里去更改父类定义那个结构，也就是说，如果我们上面这个函数如果要有一次改更父亲的机会，就需要这样调用

```
        CMyFrame()
        {
                PreCreateWindow(CREATESTRUCT &cs);
                Create(NULL,"牛刀教我做 MFC");
        }
```

然后去创建我们的窗体就行了，当然，在 MFC 中，这种创建窗体的工作也由老爸给做了，很方便。

当然喽，好多的消息循环并没有用虚函数的办法下放，而是用消息映射的办法，所以我们想做一个消息循环就需要做好多代码，当然，这些工作，在我们做 MFC 程序时，系统会给我们做，系统有一个工具叫做 ：建立类向导，按快捷键 Ctrl+W 可以运行，当然也可以按菜单上的“ 查看/建立类向导 ” 来

运行：

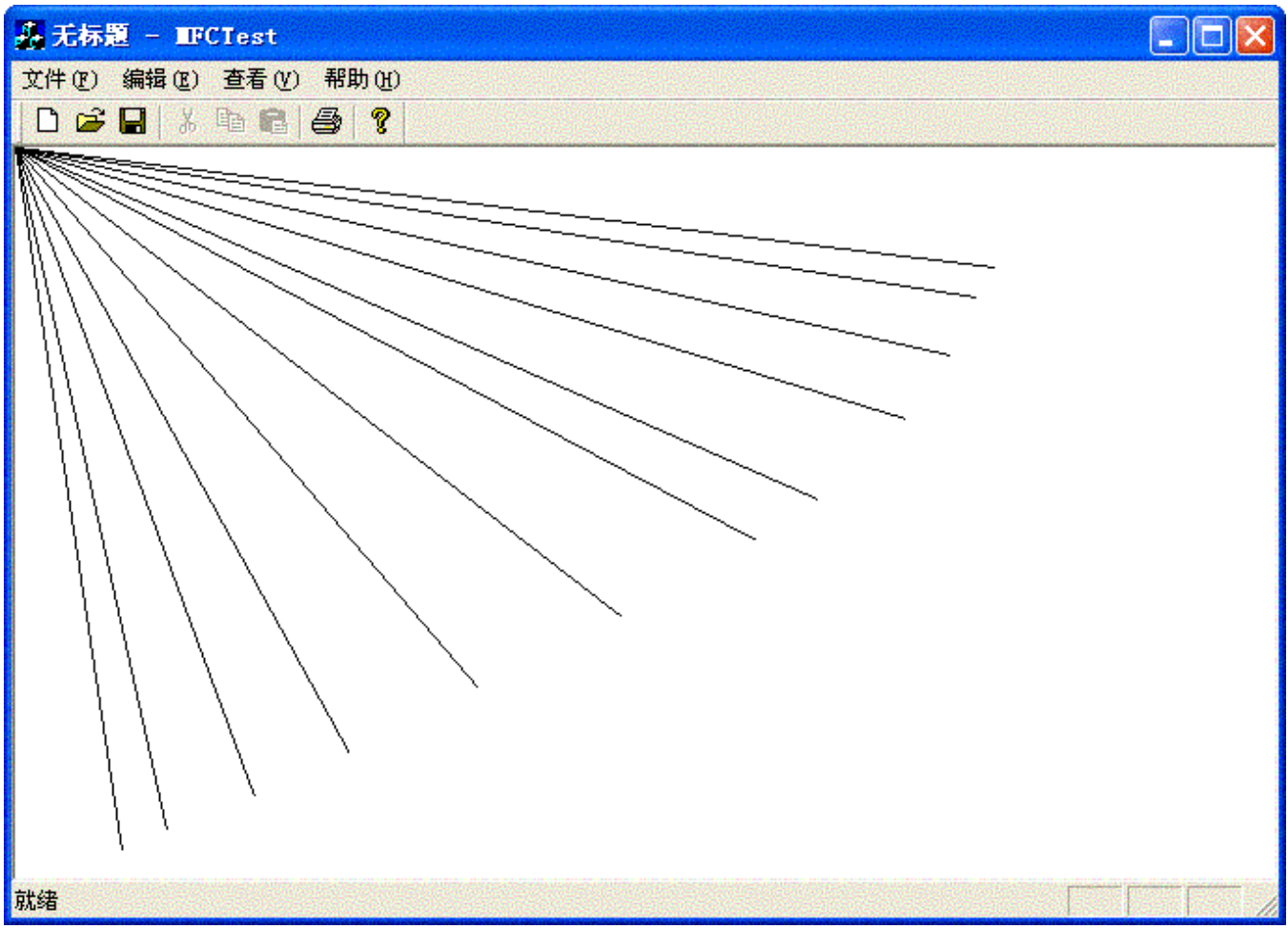



大家看，我们先在 Class name 下面选择 CMFCTestView ，在下面 Messages 下面的列表框中放着所有的虚函数与消息，我们想要响应哪个消息，只要在上面双击，然后按 Edit Code 就一切 OK 了，进入了下面的函数：

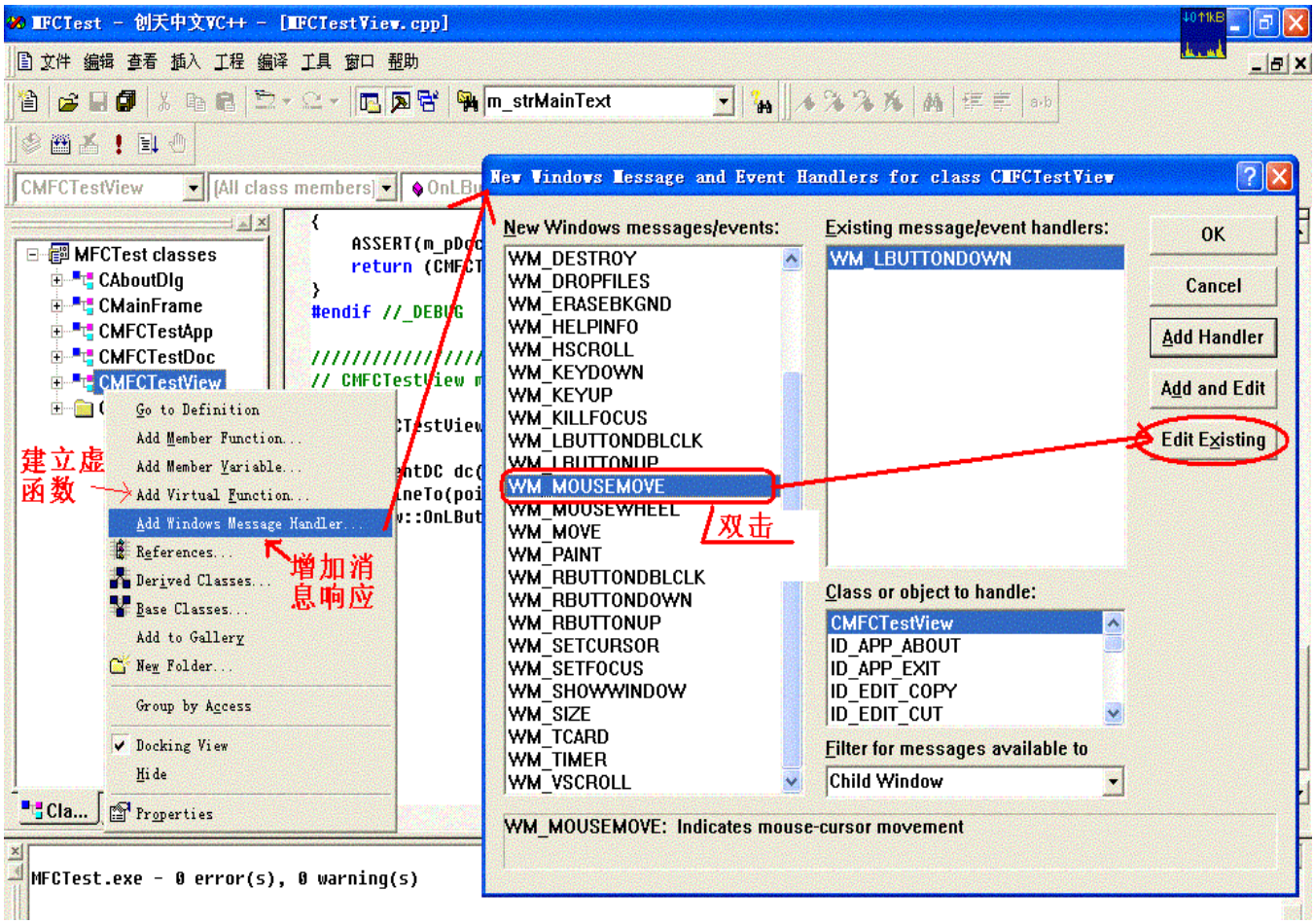
```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default

        CView::OnLButtonDown(nFlags, point);
}
```

舒服吧，连点的结构都送给我们了，我们想做程序怎么办？很简单啊：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        dc.LineTo(point);
        CView::OnLButtonDown(nFlags, point);
}
```




大家看，同样的程序，我们用好几种办法给大家做过了，大家是不是感觉，用 MFC 真是方便呢？呵呵。上面那个“建立类向导”可帮了我们的大忙了，一开始可能你会觉得不习惯，或者很繁，但随着我们的讲解，你逐步了解了内部的机理，你就会觉得太方便了，当然，我们还有一种办法去添加这些函数，那就是：直接在类上右击：（会弹出一幅菜单）




这次我们加一个 WM_MOUSEMOVE 消息，看代码：

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default

        CView::OnMouseMove(nFlags, point);
}
```

你瞧多好，又将点给了我们，这次我们还是用这个点画线，看看结果：

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        dc.LineTo(point);
        CView::OnMouseMove(nFlags, point);
}
```

同样的代码，在这里运行感觉不同。

对了，你说人家孩子多孝顺，每次调用过父类的消息后，还要再调用一下父类，也就是儿子说：老爸我处理好了，你有什么要处理的，你自己去处理吧。

正是这种机制，使得老爸也能默认地去处理一些东西，而不是一言堂。当然，

老爸怎么处理，那我们这些做儿子的类就管不着那么多了。

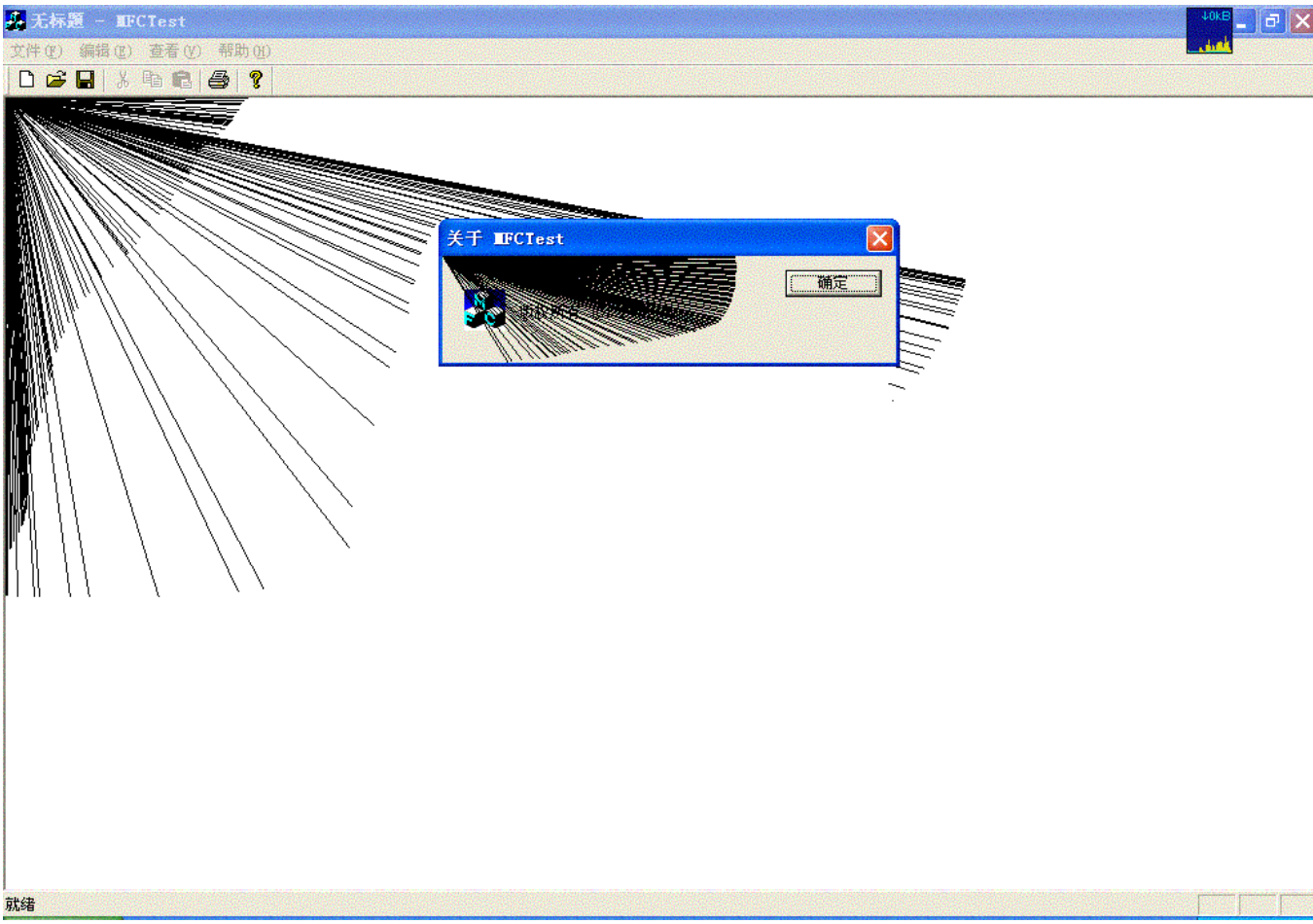
对了，大家还记得我们《微软积木》开讲时给大家画的那张图吗？大家看到每一个窗口都是一个类是吧？也就是说，我们的那一个对话框，也就是CAboutDlg 类喽，那好，我们在那对话框上做做手脚，大家为我们对话框也加一个 WM_MOUSEMOVE 消息响应函数（注：可以在建立类向导中，也可以直接在 CAboutDlg 类名上右键加入消息，于是，加入的消息为：

```
void CAboutDlg::OnMouseMove(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default

        CDialog::OnMouseMove(nFlags, point);
}
```

注意喽，对话框是窗体吧？只要是窗体，这种消息咱还就也加，呵呵，我们也加入与上面同样的代码：

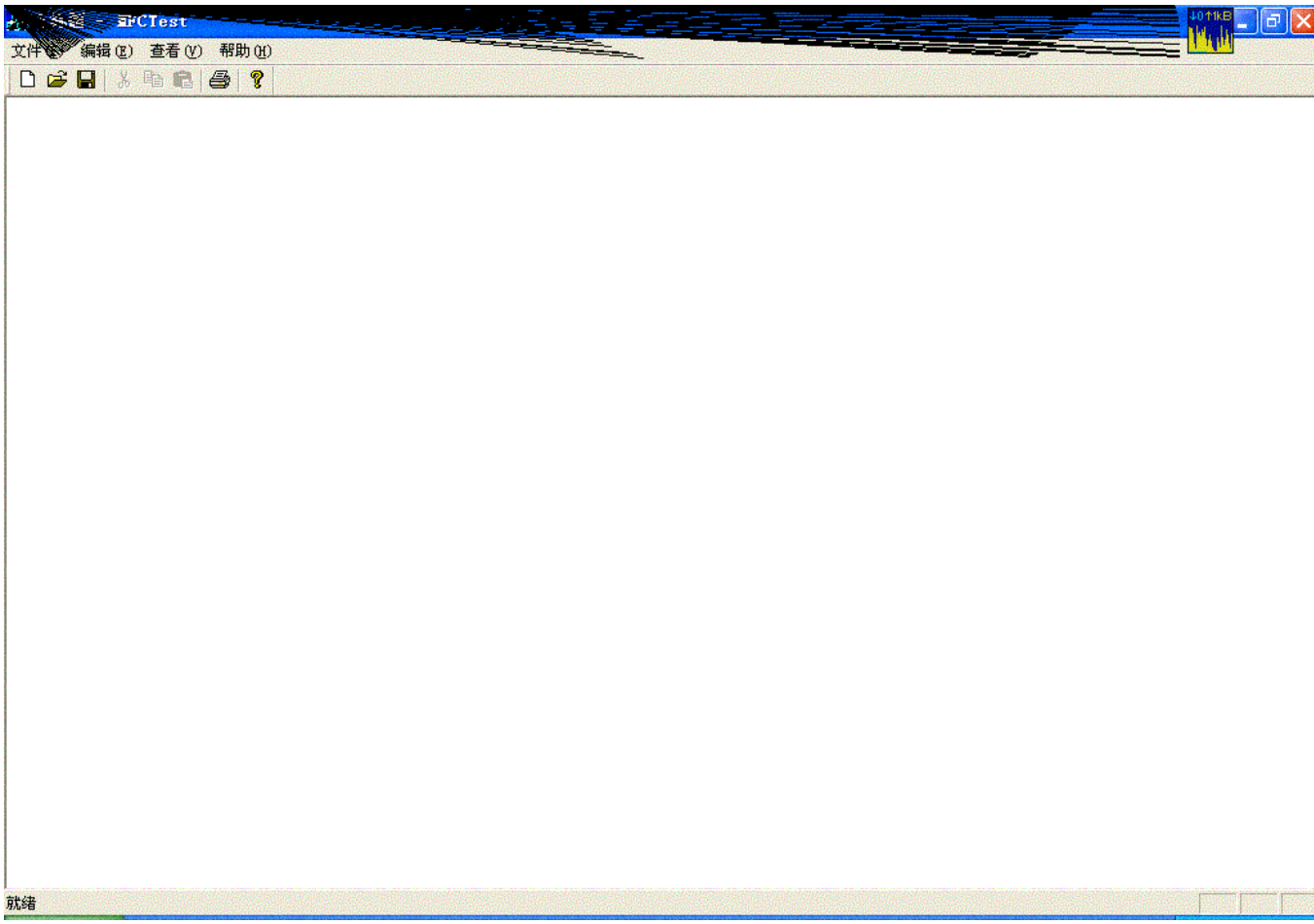
```
void CAboutDlg::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        dc.LineTo(point);
        CDialog::OnMouseMove(nFlags, point);
}
```

运行，按上面工具栏中的黄色问号或者直接按菜单中的“帮助/关于MFCTest”，看图:




永远不要怀疑自己的眼睛与能力，老子能干的事，只要是虚函数，儿子拿来也样的能干，这就是虚函数给我们带来的好处。也正因为如此，我们的虚函数，给了我们去整理老子的机会，你可以使用虚函数技术，使得老子产生的窗体面目全非。当然，要达到那种目标，我们当然还有好多的路要走，还是那句话，我们一口不能吃个胖。

大家看下面的图片，牛刀是在什么地方画线的，呵呵。


CTest
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪

**微软积木〇七六：光辉生命**

人总是要死的，或重于泰山，或轻于鸿毛。

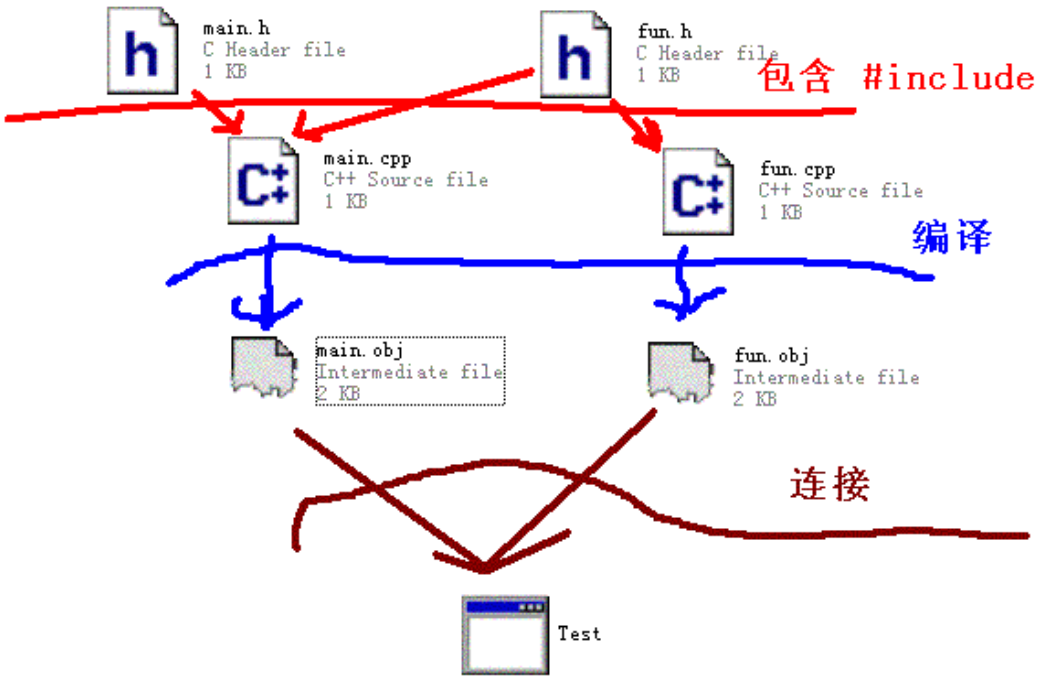
呵呵，任何一变量也都是有自己的生命周期的。我们又要来讨论这种问题了，我们先与大家做个两个文件的系统：（注：我们以注释的方法告诉大家文件名称，大家知道怎么建文件吧？

先建立一个控制台应用程序，然后加入两个头文件（ C/C++ Header File）：main.h fun.h 和两个 C++文件（C/C++ Source File） main.cpp fun.cpp，

好，我们在 main.h 中输入程序：

```
//main.h
#include <stdio.h>
#include "fun.h"
//main.cpp
#include "main.h"
void main()
{
}
//fun.h
#include <stdio.h>
void fun();
//fun.cpp

#include "fun.h"
void fun()
{
}
```

编译程序，到我们的 Debug 目录下面去看看：




大家看，其中包含了 stdio.h 我们没有显示给大家，但你会在你的 Debug 文件夹中看到相应的 vb60 之类的文件，其实也是包含在其中编译的，并且生成了一些中间文件。而产生的 main.obj 与 fun.obj 就是二进制文件了，到此，编译部分就结束了，而在没有链接之前，我们知道，fun 函数其实并不在 main.obj 中，而 main 中仅仅有函数原型。什么意思呢？ #include 其实就是替代的意思，我们将 stdio.h 删掉，我们重新定义一下我们的文件：

```
//main.h
#include "fun.h"
//main.cpp
#include "main.h"
void main()
{
}
//fun.h
void fun();
//fun.cpp
#include "fun.h"
void fun()
{
}
```

我们先看看 main.cpp 吧，这个将编译成 main.obj
其中的 #include "main.h" 中将会将 main.h 的全部东西全放进来，于是变成：

```
#include "fun.h" //注，这是放在 main.h 中的
void main()
{
}
```

当然，遇到了 #include "fun.h" ，又一次用 fun.h 中的内容替换，于是变成：

```
void fun();
void main()
{
}
```

看看，我们绕了一大圈，原来就是这两行代码啊，不错，被编译进了 main.obj 中了
同理，我们可以知道， fun.cpp 就相当于：

```
void fun();
void fun()
{
}
```

也就这么简单，也就是说，我们 main.obj 中只有函数原型，没有函数的定义，但编译并没有报错，大家还记得我们以前与大家讲过吗？我们只要定义函数原型，编译就不会报错，即使报错也只会在 Link （链接部分报错），也就是链接时，会将 main.obj 与 fun.obj
组装成一个代码 Test.exe 文件。

当然，我们只要在链接时，找到相应的函数，就可以正确编译，那为什么会这样呢？这就为我们以后的链接库打下了伏笔，当然喽，我们说这样做有一个好处就是，我们可以在每一个不同的文件中去定义不同的功能，并且这两个文件的定义可以不在一起，不在同一个系统，甚至不在同一台机上。

这样一来，我们的软件，就可以顺利地从某一种功能上切割开，分给不同的人去完成，使得我们做一个程序并不是一个人去单打独干。一般如果一个软件制作团体如果有１０个人的话，那么他们就可以同时做十种不同的软件功能，而在做完了软件后，将所有的 obj 文件拼接在一起，再链接一下就一切 OK 了。

这个大家知道了就好，我们知道，两个不同的文件中，全局变量是不能一样的，原因很简单，假如一个团队有两个人编软件，而我们在我们文件中定义了一个全局变量 i ，你也定义一个 i 如果不同的文件中 i 在链接的时候都一样，那么我做的相应的功能的变量就会影响你的软件中的变量，所以我们全局变量的生命周期一般就在一个文件范围内，大家看程序：

```
//fun.cpp
#include "fun.h"
int i=5;
void fun()
{
printf("fun: i=%d\n",i);
```

```
}
//main.cpp
#include "main.h"
int i=8;
void main()
{
printf("main: i=%d\n",i);
fun();
printf("main: i=%d\n",i);
}
```

这个程序从书本的理论上应该输出

```
i=8
i=5
i=8
```

吧，因为我们没有加 extern （注： extern 就是定义外部变量的），可惜输入的结果是：

```
main: i=8
fun: i=8
main: i=8
```

牛刀猜想可以是一旦链接成了一个 exe 文件，这两个变量，系统就也分不清了，所以从理论上来说，牛刀初步估计，从道理上来说，外部变量应该是指编译后不同文件之间的外部变量，从这一点上来说，为了测试，牛刀不得不教大家做 dll 文件，因为在那里，不同的文件中的变量是放在不同的文件中的，我们去试试看，当然，我们要重新建立一个工程：

文件/新建../[工程]/Win32 Dynamic-Link Library

建立一个名称为 fun 的工程：

当然，然后还建立一个 fun.h 文件，一个 fun.cpp 文件

```
//fun.cpp
#include "fun.h"
int i=5;
void fun()
{
printf("i=%d\n",i);
}
//fun.h
#include <stdio.h>
_declspec(dllexport) void fun();
[code]
```

在动态链接库中，就需要使用 _declspec(dllexport) 来表示那一个函数我们需要导出，因为链接库中的函数不一定都需要提供给别人，所以可以使用这种办法将需要提供给别人的函数导出来，也称之为动态链接库的外部接口。

好了，大家编译一下，会发现，相应的 Debug 中多了好多的文件，其中有两个是我们需要的：

一个是 fun.dll 一个是 fun.lib，其中的第一个函数就是我们的动态链接库，是纯二进制编译好的文件，而 fun.lib 就是我们导出函数的列表，记录着我们导出函数的信息。

好了，我们将这两个文件复制到我们刚刚前面建立的那个 main 工程下面，

再将原工程中的 fun.cpp 删了，然后我们在我们的 fun.h 头文件中，将文件内容改为：

```
#include <stdio.h>
extern void fun();
```

再按 Alt+F7 或者点击 工程/设置... 进入设置对话框，按 [Link] 选项卡，在下面的“对象/库模块”中加入： fun.lib 注意，和后面的一定要有空格（西文空格）

好了，到工作区中的 FileView 选项卡中，点 Source Files，将其中的 fun.cpp 删掉，不然会报错，好了，程序的设置就到此结束，如果大家对上面过程还不是很清楚的话，发贴问我，这种动态链接库的技术，我们以后会详细地与大伙讲。

好了，万事具备，编译一下，然后点那个惊叹吧，去运行看看吧，出现了什么结果？

```
[code]
main: i=8
i=5
main: i=8
```

一切如我们所愿，所以牛刀在这儿给大伙要纠正以前的一种习惯性思维，全局变量在编译后的多文件系统中是各自独立，如果想要互相只有一个变量怎么办？加一个 extern 标志，如下：

```
extern int i=5;
```

编译，然后到 main 文件中去，也将定义前加上 extern 如下：

```
extern int i=8;
```

也编译，好像结果没有变啊？对吧？呜～～～

别急，大家忘了一件事了，那就是，我们的 dll 制作后，每编译一次，都需要重新将生成的 fun.dll 文件 与 fun.lib 文件都重新的复制到新我们的工程中，（替换掉原来的文件）然后再编译：

竟然还没用，难道还要导出？

试看看吧：

```
//fun.cpp
#include "fun.h"
_declspec(dllexport) int i=5;
void fun()
{
i=4;
printf("fun: i=%d\n",i);
}
//main.cpp
#include "main.h"
_declspec(dllimport) int i;
void main()
{
printf("main: i=%d\n",i);
```

```
fun();
printf("main: i=%d\n",i);
}
```

终于，执行的结果为：

```
main: i=5
fun: i=4
main: i=4
```

大家仔细地去分析，但可惜的是不能在 main.cpp 中给变量 i 赋值，例如：

```
//main.cpp
void main()
{
int i=8;
printf("main: i=%d\n",i);
fun();
printf("main: i=%d\n",i);
}
```

那么结果立马变为：

```
main: i=8
fun: i=4
main: i=8
```

似乎 main.cpp 文件又为自己的区域定义了一个单独的，与外界不通气的全局变量。

从牛刀上面的实践来说， VC6.0 为我们提供的全局变量与外部变量之间有巨大的漏洞，也就是根本就不符合我们在书本上的那种变量的定义方案，也就是说，我们在编程过程中，尽量避免在不同的文件中定义相同的全局变量，如果你定义的话，那可能程序运行的后果你自己都没办法把握。

所以从这一点上来说，牛刀建议，在文件中不要定义全局变量，而宁可去定义一个局部的静态变量，那么什么叫局部的静态变量呢？很简单，在定义的变量之前加上一个 static ，看一段代码（先将上面的全局变量给忘了，呵呵，当然，如果你要做大型的软件，多人做，那么全局变量之间的通讯就需要在大伙商议一下了，当然，如果你对这种全局变量与外部变量有什么独到的见解，请告诉牛刀，也有可能是牛刀实践得不太到位）：

```
#include <stdio.h>
void fun()
{
int i=5;
i++;
printf("i=%d\n",i);
}
void main()
{
fun();
fun();
```

```
fun();
fun();
}
```

运行的结果呢？想必大家很清楚：

```
i=6
i=6
i=6
i=6
```

原因是，每当函数退出时，原来的 i 就不存在了，也就是说，程序在调用我们函数时，其中是在内存中生成我们的函数以及里面的定义，而一旦函数退出，这一段内存就失效了，不存在了，所以第二次调用时，又创建了一次函数与数据，于是，我们每次创建一次函数，就创建一个变量，于是，变量的值就最终也逃脱不了 5 与 6 的恶运，当然，我们有时候需要保持原来的结果，那么就需要在函数中相应变量前加上 static 这样：

```
void fun()
{
static int i=5;
……
```

那么，再次运行程序：

```
i=6
i=7
i=8
i=9
```

原来啊，只要使用了 static ，那么编译系统看到这个标志时就会仅仅为其分配一次内容，并且在整个程序的堆上分配，这样一来，只要这个软件运行没有停止，这个变量就一直保留着，就好像是静态的，所以我们也称之为“静态变量”。

静态变量与全局变量的使用差不多，但其只在局部区域有效，所以我们看下面的程序：

```
#include <stdio.h>
void fun1()
{
static int i=2;
i+=2;
printf("fun1 :i=%d\n",i);
}
void fun2()
{
static int i=1;
i+=2;
printf("fun2 :i=%d\n",i);
}
void main()
{
fun2();
```

```
fun1();
fun2();
fun2();
fun2();
fun1();
fun1();
fun2();
fun1();
}
```

请大家看下面的运行结果，仔细地去体会

```
fun2 :i=3
fun1 :i=4
fun2 :i=5
fun2 :i=7
fun2 :i=9
fun1 :i=6
fun1 :i=8
fun2 :i=11
fun1 :i=10
```

大家是不是发现两个函数中的 i 是单独变化的？（注意，奇数肯定是 fun2 的函数，偶数肯定是 fun1 的函数）

也就是说，没有万不得已的事情，建议大家不要使用全局变量只使用局部变量，而如果要这个函数中的定义数据不变，就定义为静态变量。

可能你会说，那函数之间如何通讯呢？通过参数啊，一样可以实现函数之间的通讯，虽然费事，但很保险。

好了，我们通过一个类，重新来认识一下变量的范围吧：

```
class A
{
int x,y;
public:
fun()
{
  int a,b;
}
};
```

大家看上面的例子，显然，成员变量 x,y 就与整个类的生命周期一致，而 a,b 则当函数 fun 执行结束了，那么 a,b 就会寿终正寝了，没办法，认命吧。

我们为什么会去强调这种变量之间的关系呢？原因很简单，这一讲的最后，我们就讲一讲原因吧，不给大家举具体的例子了。

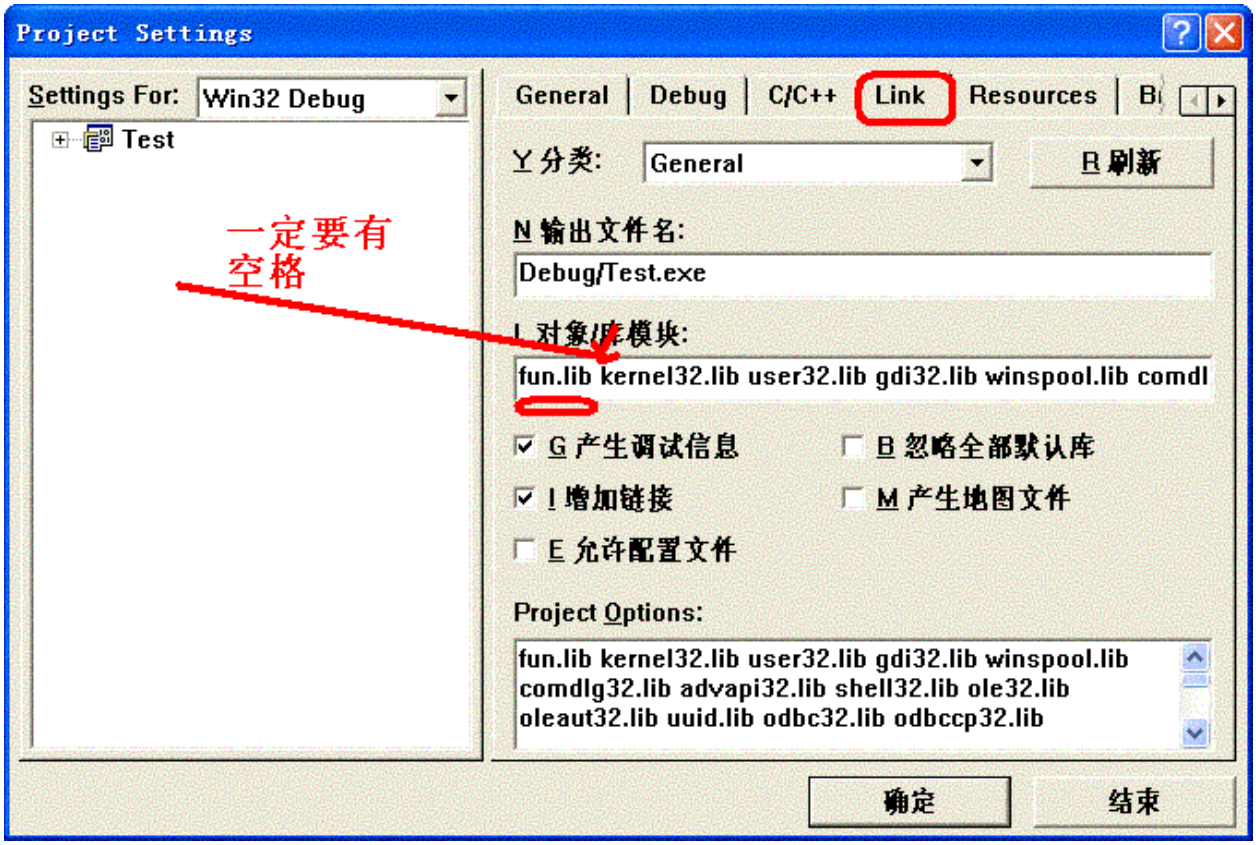
大家明白，我们的 CWnd 是一个窗口对象，但窗口对象却不是窗口本身，而只是窗口的一件外套，那么对于这个窗口来说，类中的数据成员就相当于这个窗口的全局变量，也就是说，窗口还没定义，某些数据成员就存在了，窗口消失了，数据成员还在，于是，我们就可以在某些窗口消失后，做一些后续的数据处

理工作，但由于这时候的窗口已经没有了，所以我们对窗口的所有操作都会失败。

我们以前曾经与大家讲过，类不是窗口，但却会有好多的人认为其是一个窗口，于是就按照窗口就是对象的方式去编程，于是，就会出现一些窗口都消失了我们还在对窗口进行操作，这种现象。

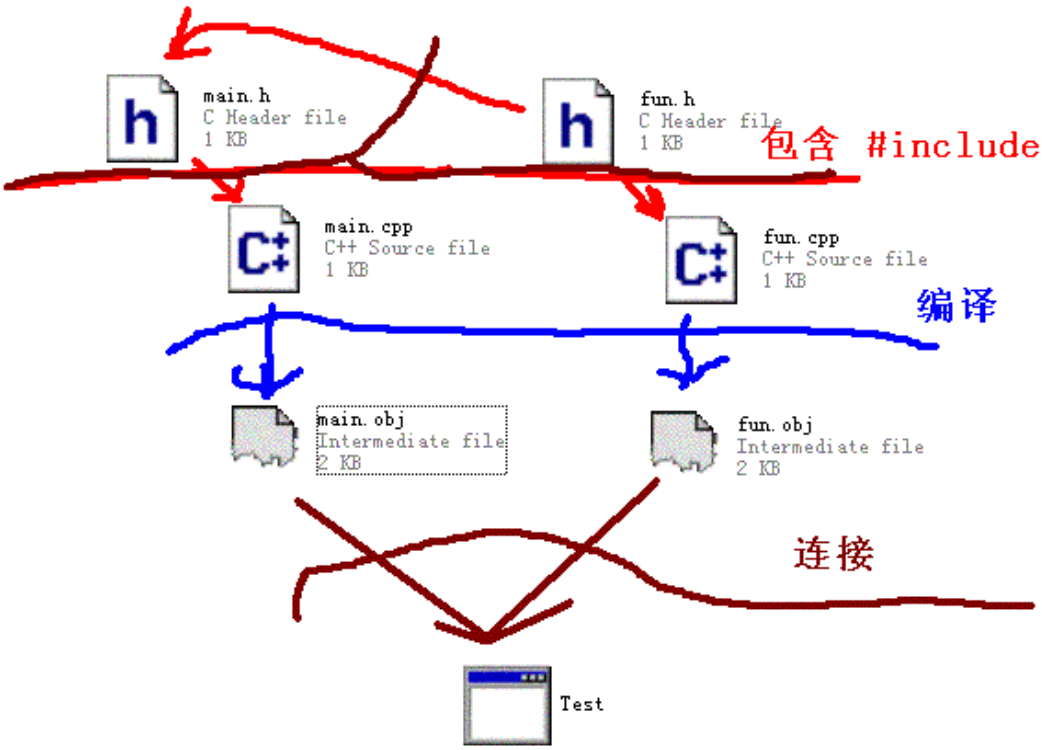
当然，窗口也并不是任何时候都比类的生命周期短，正因为窗口不属于类，所以类在消失之前，只要有别的类能够接受这个窗口，那么这个类完全可以交权，于是，在类中的局变量定义的窗口的生命周期可能比类的生命周期还要长。

当然，这些东东，我们会在以后继续与大伙聊，但理论必须掌握，对了，大家对全局变量与外部变量有什么自己的独到理解与实践经验，一定别忘了与牛刀分享。




呵呵，第一幅图好像有点问题，修改了一下，管理员看一下，为什么这里的 extern 好像失效了似的，不知道是什么原因，两个不同文件定义的变量不用外部变量标识竟然能够通用。

重画的图：


main.h
C Header file
1 KB
fun.h
C Header file
1 KB
包含 #include
main.cpp
C++ Source file
1 KB
fun.cpp
C++ Source file
1 KB
编译
main.obj
Intermediate file
2 KB
fun.obj
Intermediate file
2 KB
连接
Test

**微软积木〇七七：贴心父子**

C++ 中的父子经历了坎坎坷坷，一路走来，首先来了个第三者，后来父子又为了权利闹起了矛盾，但人家父子就是父子啊，还要静下心来贴心长谈。

```
#include <stdio.h>
class A
{
public:
        A()
        {
                printf("进入了 A（父类）的构造函数。\n");
        }
        ~A()
        {
                printf("进入了 A（父类）的析构函数。\n");
        }
};

class B:public A
{
public:
        B()
        {
                printf("进入了 B（子类）的构造函数。\n");
        }
        ~B()
        {
                printf("进入了 B（子类）的析构函数。\n");
        }
};

void main()
{
        B b;
}
```

运行结果是什么呢？是：

```
进入了 A（父类）的构造函数。
进入了 B（子类）的构造函数。
进入了 B（子类）的析构函数。
进入了 A（父类）的析构函数。
```

显然是先构造父类，进入了 A 的构造函数，然后构造 B ，而析构正好相反，那么就有问题了，我们现在定义的 A 的构造函数没有参数好办，如果有参数怎么办呢？因为我们总不可以用以下的办法：

```
B()
{
	A(2,3);
}
```

这种办法去向父亲吐露真心，因为父亲在构造时，儿子还没有出生，也就是说，父亲在已经执行构造函数时，子类还没有执行，而父类执行完毕了，父类的构造函数也就已经调用完毕了，所以一旦进入了子类，那么父类已经构造好，那调用父类的构造函数就没有任何意义了，所以我们儿子需要在还没有降生之前就需要将参数传递给父亲，没办法，函数中不能够传，那就在函数定义之前传递吧，于是，就有了以下的传参形式：

```
B():A(2,3)
{
}
```

也就是说，在定义 B 的构造函数后加一个冒号，然后放进父亲的构造的参数，这就是解决之道，我们看程序：

```
#include <stdio.h>

class A
{
public:
	A(int a,int b)
	{
		printf("儿子传进来的数据分别为：%d  与 %d\n",a,b);
	}
};

class B:public A
{
public:
	B():A(2,3)
	{

	}
};

void main()
{
	B b;
}
```

当然喽，既然能在还没有定义函数之前就可以给父亲赋值，那么也可以为一些数据赋值，如：

```
class B:public A
```

```
{
        int x,y;
public:
        B():A(2,3),x(6),y(7)
        {

        }
};
```

怎么那么像函数啊？不错，在这赋值不能用等号，当然，在其他地方赋值我们就只能用等号了，牛刀去试了试，好多次都失败了，呵呵。大家看其他哪些地方可以用可以告诉牛刀。

好了，这种生前调用的办法使得我们可以顺利地在子类还没有定义的时候就向父类去传参，这真是好，当然，一旦父类形成了之后，我们就不能通过这种办法去传参了，看程序：

```
#include <stdio.h>
class A
{
public:
        void Say(int a,int b)
        {
        }
};

class B:public A
{
public:
        void Say():A::Say(2,3)
        {
        }
};

void main()
{
}
```

上面的程序一运行就报错，显然是因为不是构造函数，那么我们怎么调用父类呢？只有这样调用：

```
class B:public A
{
public:
        void Say()
        {
                A::Say(2,3);
```

```
        }
};
```

那么上面的 :: 不是全局标志吗？不错，但在类中，如果前面加类的话，则表示调用相应类中的函数（请大家注意，是调用类中的函数，而不是调用对象中的函数，调用对象的函数，我们还是用点“.”，例如我们如果已经定义了： A a; 那么调用则应该是 a.Say(2,3); 而上面是子类调用父类的特殊形式，因为我们在制造类的时候，我们还没有去定义任何对象，所以我们根本不能用对象的调用办法。

当然，如果在类外，还可以使用上面的区域标识标符在类外为类定义函数，那时我们前面一定要加上返回值类型例如：

```
#include <stdio.h>

class A
{
public:
        void Say(int a,int b);
};
void A::Say(int a,int b)
{
        printf("这是类 A 的 Say 函数。\n");
}
class B:public A
{
public:
        void Say()
        {
                A::Say(2,3);
        }
};

void main()
{
        B b;
}
```

当然，我们看下面一段程序：

```
#include <stdio.h>

class A
{
public:
        void Say(int a,int b);
};
void A::Say(int a,int b)
```

```
{
        printf("这是类 A 的 Say 函数。\n");
}
class B:public A
{
public:
        void Say()
        {
                printf("这类是 B 的 Say 函数\n");
        }
};

void main()
{
        B b;
        b.Say(1,3);
}
```

大家看，上面是调用父类的 Say 还是子类的 Say？可以你会说，当然是父类的喽。

可事实很令人失望，牛刀编译的结果是：报错，不能够被编译，也就是说，子类如果有同名函数，也就不去管父类了，父类的重载对子类来说无效，那我们怎么解决这个问题呢？

当然一个办法是在子类中也重载一个相同的函数，去调用父类，看程序：

```
        void Say(int a,int b)
        {
                A::Say(a,b);
        }
```

当然，还有另一个办法，就是加一行关键字：using

```
class B:public A
{
public:
        using A::Say;
        void Say()
        {
                printf("这类是 B 的 Say 函数\n");
        }
};
```

原来编译器是一个懒家伙，而 using 就相当于一个监督员，呵呵。

其实我们编译器懒的例子还有好多呢，看程序：

```
#include <stdio.h>
void main()
{
```

```
        int i=2;
        if( 2|| i++)
        {
                printf("i=%d\n",i);
        }
}
```

大家看程序，当编译器看到 if 后面的或前面是 2（肯定不为假了）那么后面的 i++ 就不执行了，所以我们编程时，一定要注意这种问题，我们要知道编译器什么时候会偷懒，否则我们做程序时，出错会出得莫名其妙，你会怎么查程序都查不出错误来，而一运行结果就不正确，这就是服务器“偷懒”的结果，哎～～～，没办法啊!!!!

大家看下面的程序，我们将会经常用到：

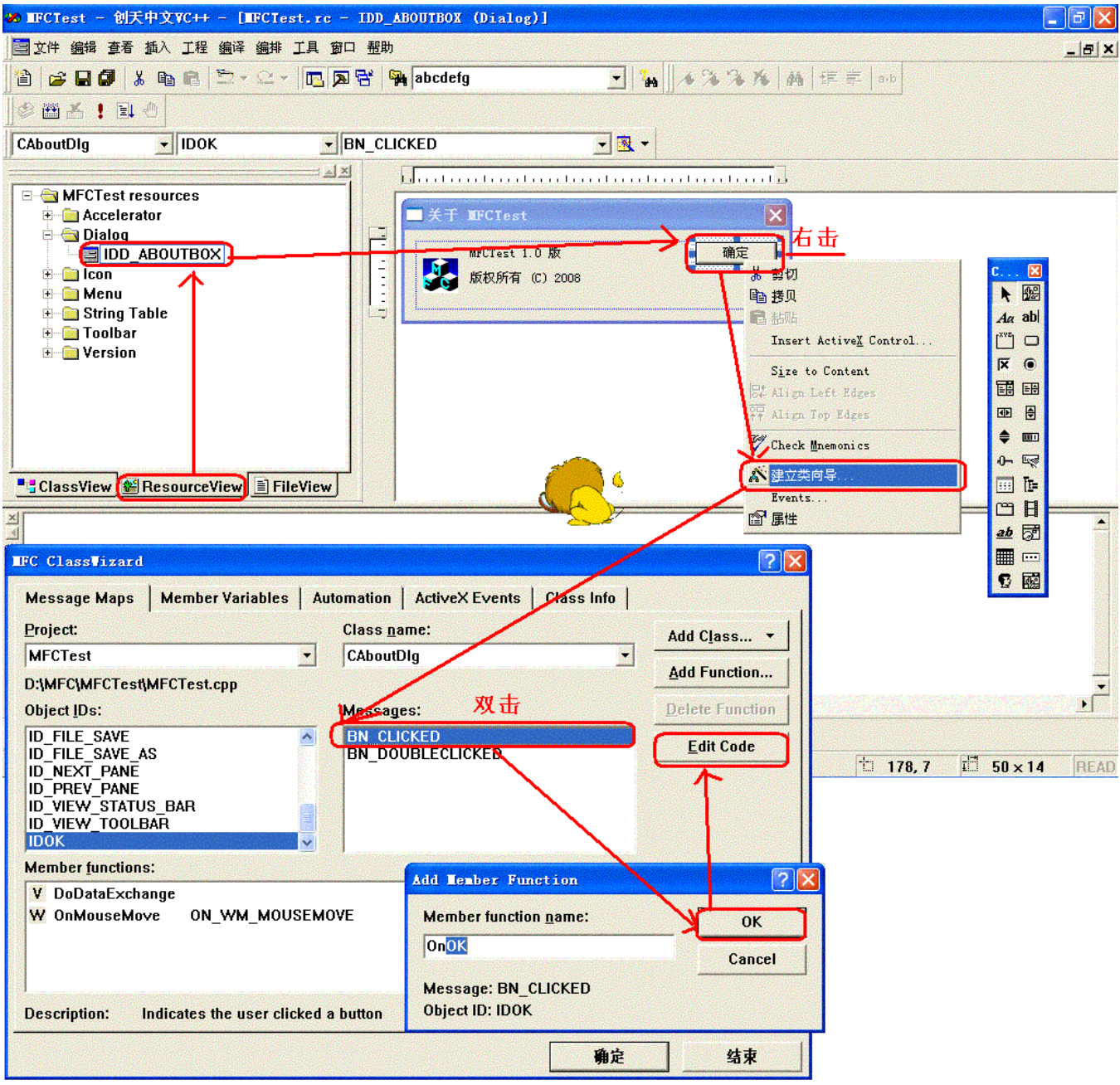
```
#include <stdio.h>
class A
{
public:
        void Say(int a,int b)
        {
        }
};
class B:public A
{
public:
        void Say(int a,int b)
        {
                A::Say(a,b);
        }
};
void main()
{
        B b;
        b.Say(4,7);
}
```

儿子每调用一次函数，就在覆盖了父亲之后就调用父类，你说这儿子多孝顺啊，呵呵，虽然父亲不能说话了，可人家的儿子还让父亲说，这种办法使得我们说话就自然地生成了一个顺序，由子类向父类一代一代地传，这有什么好处呢？好处就是：儿子处理过了，还可以将剩余没有完成的工作交给父亲处理。

好了，我们新建一个 MFC 程序，单文档。

到“工作区”的 ResourceView 选项卡上，双击 IDD_ABOUT ，然后在“确定”按钮上鼠标右击，选择“建立类向导”，在 Messages 下面的列表框中双击 BN_CLICK ，弹出一个建立相应的函数的对话框（注：函数为 OnOK），

按“OK”按钮，然后点“Edit Code”按钮，如图：




呵呵，费了半天劲，得到了一个函数而已：

```
void CAboutDlg::OnOK()
{
        // TODO: Add extra validation here

        CDialog::OnOK();
}
```

以上这种为按钮、控件、菜单等建立函数的过程，我们称之为“对……进行……消息响应”，例如我们上面过程我们称之为：对“确定”按钮进行

“BN_CLICK”消息响应。

响应好了，我们看那里只有一个调用父类的函数，那么调用父亲干什么的呢？是关闭这个对话框的，并且向返回一个 IDOK 的值，也就是说大家编译运行程序，再点“帮助/关于 MFCTest ”菜单，或者直接点上面的黄色的问号，会出现我们刚才响应消息的那个对话框，大家按“确定”，关闭了吧？那么好，我们进入上面的 OnOK 函数，将调用父类的函数 CDialog::OnOk 给删了，于是变为：

```
void CAboutDlg::OnOK()
{
}
```

大家再编译运行程序，然后点黄色的问号，再点“确定”，看我们还能够关掉对话框吗？也就是说，关闭对话框那种事的确是事后父亲帮我们做了，所以我们还是不要删的好，当然，我们可以加进一些还是可以的，看程序：

```
void CAboutDlg::OnOK()
{
        MessageBox("我点击了“确定”按钮");
        CDialog::OnOK();
}
```

大家再编译，看，我们一点“确定”，就跳出我们上面的消息框了，是不是，关闭后对话框不见了。

当然，你也可以将消息移到下面去：

```
void CAboutDlg::OnOK()
{
        CDialog::OnOK();
        MessageBox("我点击了“确定”按钮");
}
```

这次是对话框先不见了（注：被父亲先给关了），然后弹出了我们的消息框。

当然，我们也可以用以下的办法去做个判断：

```
void CAboutDlg::OnOK()
{
        if(IDYES==MessageBox("你确定要关闭对话框吗？","对话框测试程序",MB_YESNO))
        {
                CDialog::OnOK();
        }
}
```

这个程序就有点智能了，呵呵，可以根据我们的选择决定是否去调用父类的 OnOK 为我们去关闭对话框。

其实 MFC 就是这样，好多的事情都是父亲帮我们做了，我们要适时地想到父亲，因为只有那样，我们才能够真正地做到最大化的代码重用，而绝大多数的事情都变成了调用调用函数上，而将我们设置重点放在我们软件功能的设置

上。

牛刀以前给大家讲 API 时，其实好多的 API 都没与大家讲，为什么呢？因为我们以后还会与大家重点讲 MFC 中的相应函数，而如果我们在 API 都讲了，那我们以后 MFC 与大家讲什么呢？呵呵，再说， MFC 中学到的东西都简单啊。

那么，为什么我们不直接讲 MFC 呢？当然目的还是为了让大家能够明白，我们的 MFC 是怎么来的，我们怎么去改造 MFC 。

好了，我们好多的函数都是会去调用父类的，特别是我们通过消息映射的办法去建立的函数，都需要和父亲去通通气，看父亲那儿有什么好办法，如果有，就用了，呵呵，所以我们一般正常情况下，没有特殊情况，我们不去将这些函数删了，当然不删不代表不可以删，我们只要明白我们需要什么，我们不需要什么，我们就可以正确地驾驭我们的 MFC 了。

好了，我们下面就继续来与大家讲一讲我们对于类的正常操作，希望对大家有所帮助，我们以后经常使用的工具，一个是“工作区”中的 ClassView 选项卡，一种是 “建立类向导”。

当然，在使用这两个工具时，经常会出现问题，下面就两个问题讲一下解决之道：

·ClassView 选项卡中的类消失

这是我们做程序会经常会生的一件事，没有类会很麻烦，例如一些函数的提示信息没有了。

解决办法：关掉当前工程，到相应目录下，删除相应的 NCB 文件，然后再打开工程， VC 找不到 NCB 文件，会根据当前我们的工程情况重新对文件进行扫描，从而重新建立一个，我们的所有类就可以再次显示出来了。

·建立类向导中缺类

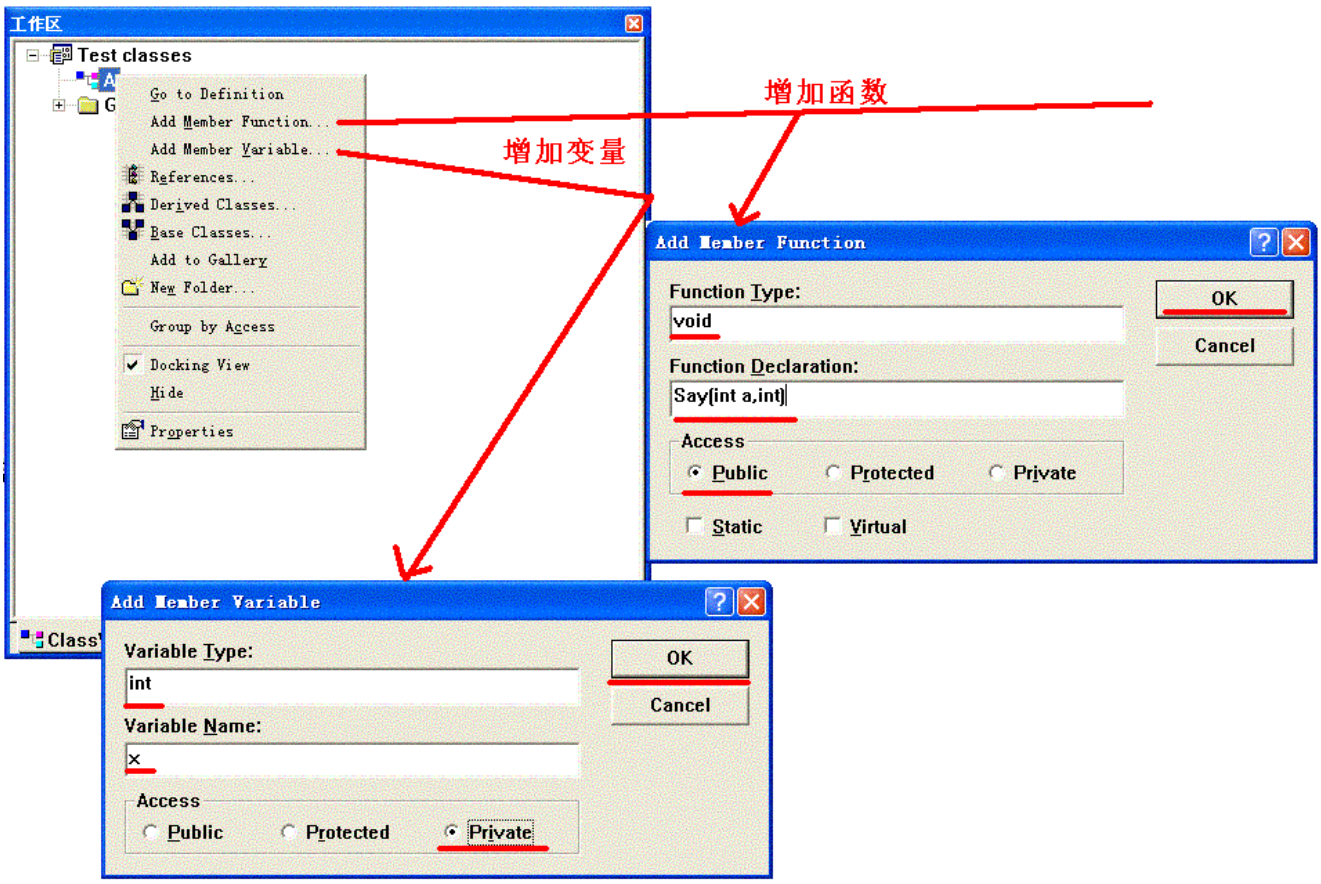
这个问题也经常会发生，和上面一样，可能是 VC 中的一个 BUG，出现这种情况也是删除文件，删掉 CLW 为扩展名的文件，然后再次打开 建立类向导时，就提示找不到文件，要你建入名称，名称与你删掉的那一个一样就行了，点击“确定”，就可以重新看到 建立类向导中我们消失的类了。

好了，我们既然上面的方法告诉大家了，下面我们就讲一讲我们怎么建立一个类，大家还是程序中输入下面代码：

```
#include <stdio.h>
class A
{
};
void main()
{
```

```
        A a;
}
```

做好后，我们就会在工作区的 “Classview”选项卡中看到我们的类：



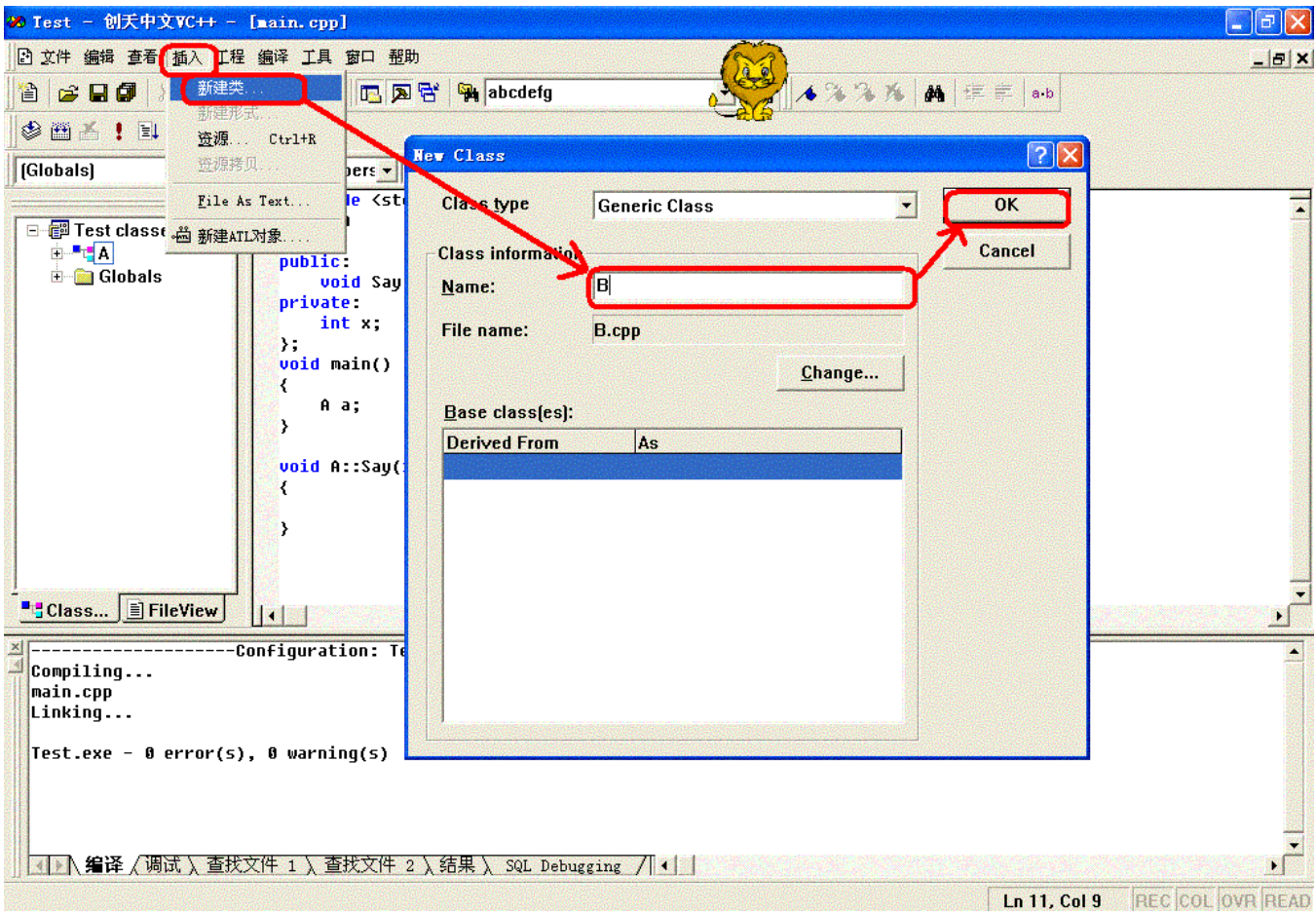

大家按上图做完后，我们的代码就变成了如此代码：

```
#include <stdio.h>
class A
{
public:
        void Say(int a,int);
private:
        int x;
};
void main()
{
        A a;
}

void A::Say(int a, int)
{
```

}

怎么样？感觉舒服吧？

那么我们怎么建立一个类呢？按菜单 “插入/新建类...”，会出现下面的对话框：




果然为我们建立了一个类，但同时也为我们建立了两个文件 B.h B.cpp ，方便啊，呵呵，当然，建立的文件也是相当丰富的喽：

```
// B.h
#if !defined(AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_)
#define AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif

class B
{
public:
        B();
        virtual ~B();
```

```
};

#endif



// B.cpp:
#include "B.h"

B::B()
{

}

B::~B()
{

}
```

你瞧，系统帮我们建立一个有构造函数，有析构函数的类。

**微软积木〇七八：熟脸父子**

随着大家编程的深入，会结识好多的父子，并且与这些父子混得越来越熟，从而成为熟脸，而对于这些熟脸，又有父子关系的类，我们称之为“熟脸父子”，我们显然是不可能用一讲的内容将所有的熟脸父子介绍给大家，但我们至少可以从理论上介绍一点。

字符串类：

字符串，相信大家一定不陌生，其实就是字符数组，当然，我们不可能过多地从理论上去与大家讲，当然，库中也会有一些函数，但我们这一讲不想用那些函数，我们与大家讨论字符串，那是一个类：

```
#include <stdio.h>
class TStr
{
public:
        char ch[200];         //我们最大处理 200 个字符
        int len;                 //字符串的长度
};

void main()
{
}
```

当然牛刀不想与大家实现所有类，也不现实，我们只是讲原理，我们既然是个字符串，那么我们就先来实现一个初始化一个字符串吧：

```
#include <stdio.h>
class TStr
{
public:
        char ch[200];         //我们最大处理 200 个字符
        int len;                 //字符串的长度
        TStr()         //没有参数的构造函数
        {
                len=0;
                ch[0]='\0';
        }
        TStr(char *str)
        {
                int i=0;
                while(ch[i++]=*str++);
                len=i-1;
        }

};
```

```
void main()
{
        TStr str("bacdefg");
}
```

我们当然可以用 printf 打印出一个字符串，看程序（由于我们的 str 中的 ch 是公有变量，所以我们可以使用如下打印函数：

```
void main()
{
        TStr str("bacdefg");
        printf("str=%s\n",str.ch);
}
```

那么我们连接字符串怎么办呢？我们使用重载加号“+”的办法：

```
……
        TStr operator+(TStr &string)
        {
                char str[200];
                int i=0;
                while(str[i++]=ch[i]);
                i=0;
                while(str[len+i++]=string.ch[i]);
                return TStr(str);
        }
};
void main()
{
        TStr str1("bacdefg");
        TStr str2("hijklmn");
        TStr str;
        str=str1+str2;
        printf("str=%s\n",str.ch);
}
```

请大家看，一旦我们对加号进行了重载，我们就可以使用加号去连接现两个字符串了，那是多么地方便啊!!!

好了，字符串的好多操作，我们都可以用类似办法去定义，以后我们会遇到 CString 类，就是一种这样处理字符串的类，我们到时候再说，我们下面再将输出函数进入一次符号重载，我们假设我们的屏幕是 tscr （注：screen 前面的 t 表示我们测试 test），那么我们再来定义一个类：

```
class TScr
{
};
TScr tscr;
```

这次我们将 tscr 设置成一个全局变量，并且就定义一个，好了，我们下面就对“>>”符号进行重载，这个符号在我们 C 中是位移，位我们现在将其表

示为向屏幕上显示东东，看程序，我们先做一个显示整数的：

```
……
        TScr &operator<<(int i)
        {
                printf("%d",i);
                return *this;
        }
        TScr &operator<<(float f)
        {
                printf("%ff",f);
                return *this;
        }
        TScr &operator<<(TStr str)
        {
                printf("%s",str.ch);
                return *this;
        }
};
TScr tscr;

void main()
{
        tscr<<5<<" "<<3.5f<<"\n";
}
```

当然，我们也可以用宏去定义：

```
#define endt "\n"
```

从而调用：

```
void main()
{
      tscr<<5<<" "<<3.5f<<endt;
}
```

来实现回车换行。

当然，我们也可以重载对键盘实现“>>” 的重载，从而将我们的输入输出变得一串一串的，非常方便，程序会根据我们需要去调用相应的重载函数。

正是基于这种思想，其实我们的 C++已经帮我们实现了这类函数的重载，我们可以包含进一个头文件 iostream.h，然后就可以顺利地使用重载了：

```
#include <iostream.h>
void main()
{
        cout<<"Good"<<3.14<<5<<21.8<<"牛刀教我做运算符重载"<<endl;
```

```
}
```

大家看，我们这儿的 cout 就相当于我们上面的 tscr ，而后面的几乎是什么数据都可以往里面放，那比 printf 可要方便省事多了。

当然，如果你想用 VC 干坏事的话，那还是 printf 方便，因为那里有后门，嘻嘻。

当然，我们也可以使用 cin 来取得数据，看程序：

```
#include <iostream.h>

void main()
{
        int i;
        cin>>i;
        cout<<"你输入的数据为:"<<i<<endl;
}
```

重载一下子使得我们的编程方便很多。

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
        string str1,str2,str;
        str1="字符串测试一";
        str2="字符串测试二";
        str=str1+str2;
        cout<<str<<endl;
}
```

打住，打住，这是怎么回事？怎么头文件后没有 .h ？ 怎么又多了 using namespace std; ？这是什么意思？

原来啊，这叫新式定义，呵呵，大家就这样写就行了，这是 STL 中的定义，原来啊， C++ 一直在发展，语法一直扩充，就好像有人与牛刀讨论过 void main 可以不可以用一样，我们一定要跟随形势。

当然喽，这些东西，牛刀不打算扩展下来与大家做详细地实践，因为我们还有另一套系统，也同样可以处理字符串，那叫 CString 是标准的 MFC 中使用的字符串类（请大家注意，这是类，呵呵），而 MFC 中绝大多数的函数都支持 CString ，那么我们就来用用这个 CString：

好了，那么我们就来建立一个 MFC 单文档应用程序吧，这次我们响应按钮的单击事件：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
```

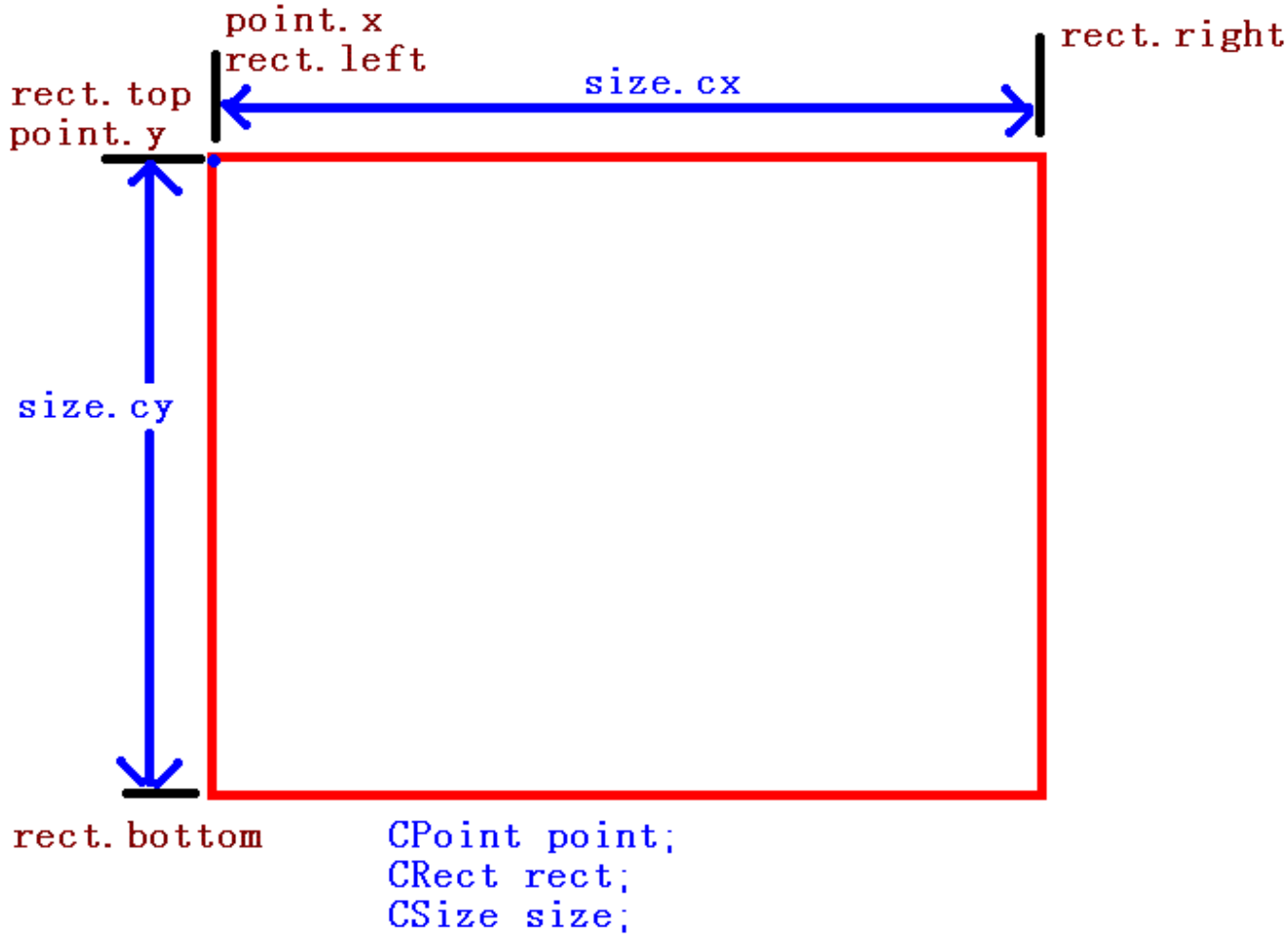
```
        CClientDC dc(this);
        CString str;          //我们定义一个 CString 类型变量
        str.Format("当前坐标：x=%d,y=%d",point.x,point.y);          //
是不是很像  printf  函数？呵呵
        dc.TextOut(point.x,point.y,str);
        CView::OnLButtonDown(nFlags, point);
}
```

怎么样？我们的操作就是这么简单，当然喽，大多数的东东，大家可以自己先去试着用看看，以后我们遇到什么问题，我们再与大家讨论这些函数的应用。

像上面的 CString 这样的使用就是完全面向对象的了，我们可以调用其好多成员函数直接去实现我们的一些目的。当然，像这样的熟脸，我们以后还会遇到好多：

CPoint　　//点
CRect　　//矩形
CSize　　//尺寸

上面三个是什么呢？一个是点，一个是矩形，一个是尺寸，就是长度与宽度，请大家看图：



CPoint point;
CRect rect;
CSize size;

请大家注意，我们在上面矩形的左上角设置了一个点，什么意思呢？我们这三个类可以自由地相加减，例如，左上角的点(CPoint)加上一个尺寸(CSize)就可以得到一个矩形，真的能吗？看起来能，那只是我们的一厢情愿，原来啊，点对这个函数没有重载，那么重载了什么函数呢？

```
CPoint
CPoint operator +( SIZE size ) const;
CPoint operator +( POINT point ) const;
CRect operator +( const RECT* lpRect ) const;
CSize operator -( POINT point ) const;
CPoint operator -( SIZE size ) const;
CRect operator -( const RECT* lpRect ) const;
CPoint operator -( ) const;

CRect
CRect operator +( POINT point ) const;
CRect operator +( LPCRECT lpRect ) const;
CRect operator +( SIZE size ) const;
CRect operator -( POINT point ) const;
CRect operator -( SIZE size ) const;
CRect operator -( LPCRECT lpRect ) const;

CSize
CSize operator +( SIZE size ) const;
CPoint operator +( POINT point ) const;
CRect operator +( const RECT* lpRect ) const;
CSize operator -( SIZE size ) const;
CPoint operator -( POINT point ) const;
CRect operator -( const RECT* lpRect ) const;
CSize operator -( ) const;
```

原来啊，我们一厢情愿的那一个函数被 CPoint operator+(POINT point)const; 函数已经给重载了，而我们重载可以，可不能重名啊，所以我们就不能如愿，所以从这一点上来看，我们做程序要尊重事实，不能想当然。

还是那句话，理论上来说，一个点加一个尺寸就可以等于一个矩形，但你不能这么做，因为系统不允许，那么我们就举几个允许的，大家自己去实践一下，牛刀就给大家做实践了，原因很简单，牛刀为你实践了，你自己也必须去实践，因为实践是一种能力，牛刀教给大家方法了，大家就要靠自己，只有那样，我们以后做程序才能够灵活。

实践内容：

• 向 CPoint 对象加上一个 CSize 对象
• 从 CPoint 对象减去一个 CSize 对象
• 从 CPoint 对象减去另一个 CPoint 对象，得到一个 CSize 对象
• 向 CRect 对象加一个 CPoint 对象或 CSize 对象
• 从 CRect 对象中减去一个 CPoint 对象或 CSize 对象

上面的代码大家一定要一个一个上手实践。

下面牛刀教大家一个函数，去实现部分对象操作，请大家对 MFC 的 鼠标移动消息进行响应，输入下面的程序：

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CPoint pt(5,5);
        CSize size(3,4);

        pt+=point;
        dc.SetPixel(pt,RGB(255,0,0));
        pt=point-size;
        dc.SetPixel(pt,RGB(0,255,0));
        dc.SetPixel(point,RGB(0,0,255));
        CView::OnMouseMove(nFlags, point);
}
```

请大家注意了，一定响应 View 类中的消息，别响应错了，嘻嘻。

上面的 SetPixel 是在屏幕上打点，前面是一个点，当然也可以用两个整形坐标，这个函数是重载了的，后面是一个 RGB 的色彩，按那是一个宏，按红绿蓝排列，值分别从 0 到 255，一共 256 种，所以三个值一乘则为:16777216 种颜色，因为我们肉眼看来已经很逼真了，所以也叫真色彩，当然，以前显卡上存储不了这么多颜色，或者为了节省空间我们用不了这么多颜色，人们就会使用调色板系统去选择一些颜色，当然，那么东东我们以后再说。

当三个值全为 0 则为黑色，全为 255 则为白色，而如果全一样，则呈现灰度效果，理论上来说，我们的系统只需要这一个函数就可以通吃图形系统的一切，但我们没办法那样做，因为那样做的话，那么我们的系统运行速度就会太慢，没办法，我们现在的电脑还不是很理想，所以就还会有其他一些元素，那我们以后详细地与大伙讨论。

当然，上面的程序运行后，大家轻移鼠标，仔细体会几种对象之间的关系，尽量做到对这几个对象的熟悉，因为我们以后用这几个对象的地方就是太多了。

原因很简单，我们用一个窗体，就是一个矩形，我们用鼠标，就会有一个点，我们们要知道长宽信息，就需要知道其长宽尺寸，好了，以后用到再与大家详细讨论吧。

下面我们重点讨论一下 CRect 对象，这个对象有一个很舒服和函数：PtInRect

这个函数的返回值是一个布尔值，参数是一个点，如果点在矩形之内则返回 TRUE 否则返回假值。

一个函数，就避免了我们使用类似下面的代码

pt.x>rect.left && pt.x<rect.right && pt.y>rect.top && pt.y<rect.bottom

似乎越来越感觉到对象的神奇，对吧？呵呵，还有更神奇的呢，那就是 IsRectEmpty 和 IsRectNull，这两个函数一个表示矩形为空，也就是什么都没有，一个表示矩形不存在。

不存在就是四个数据都为 0，而空则表示有左右值相等或上下值相等，也就是说，其实这已经不是一个矩形了。我们可以有选择地使用。

当然，还有一些函数会返回一些矩形上的点，大家一看函数就可以明白个八九不离十了：

```
rect.TopLeft();
rect.BottomRight();
rect.CenterPoint();
```

大家知道是分别返回什么位置的坐标了吧，呵呵，省事啊。

再告诉大家一个函数：

GetClientRect(&rect);

返回客户区的矩形，我们用这种办法在客户区的中心打一个红点，当然要在 OnDraw 函数中，而这个函数到底是怎么回事，我们以后会详细与大家解释，先用着吧，呵呵。

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CRect rect;
        GetClientRect(&rect);
        pDC->SetPixel(rect.CenterPoint(),RGB(255,0,0));
}
```

当然，我们也可以用下面简单的代码在客户区上画一个充满客户区的椭圆：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CRect rect;
        GetClientRect(&rect);
        pDC->Ellipse(&rect);
}
```

当然，具体的好多程序的设计方案，我们以后慢慢地详细与大家讲，以后还会遇到好多的熟脸，我们也不是一讲的内容就能介绍完毕的，希望这一讲对大家以后做程序能起到一定的作用，牛刀也就没有白忙了。

## 微软积木〇七九：神秘窗花

窗户，现在都用玻璃了，而过去都是白纸，没办法，那时候的人穷啊，而现在玻璃是玻璃了，却丢失了好多的传统。

过去有些地区因为窗户纸是一版白，显得特别地单调，所以为了增添生气，好多地区都形成了一种传统艺术——剪纸。

红色的纸剪成种种形状，贴在窗上，形成了中国独有的艺术门类。好像还申报了文化遗产。

当然，也有用其他东东糊的，例如夏布，夏布是什么牛刀也不清楚，估计肯定是织得比较疏的，眼很大的一种布了，人家糊起来作纱窗的嘛，传说纪晓岚修四库时，乾隆曾赏给他一个宫女，叫明玕，在用夏布糊窗时，纪晓岚来了，明玕可不，给纪晓岚来了一幅上联：“夏布糊窗，个个孔明诸格亮”，呵呵，一下子把这们这位才子给难住了。

当然，说到了窗纸，我们说一下，其实我们 Windows 编程中也有一种窗纸，我们称之为“画刷”，但只可惜这个画刷一般正常的情况下是白色的，所以我们感觉好像中间什么也没有一样，大家如果想看到，就做一段程序看看吧，当然在 OnDraw 函数中。

当然，由于我们画一个圆需要两个坐标，一个左上角坐标，一个右下角坐标，所以我们需要得到两个坐标，那我们怎么去得到呢？我们可以在按下鼠标时，得到一个点，松开鼠标左键时，我们再得到一个点，然后用这两个点来画一个圆。

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CView::OnLButtonDown(nFlags, point);
}
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
        CView::OnLButtonUp(nFlags, point);
}
```

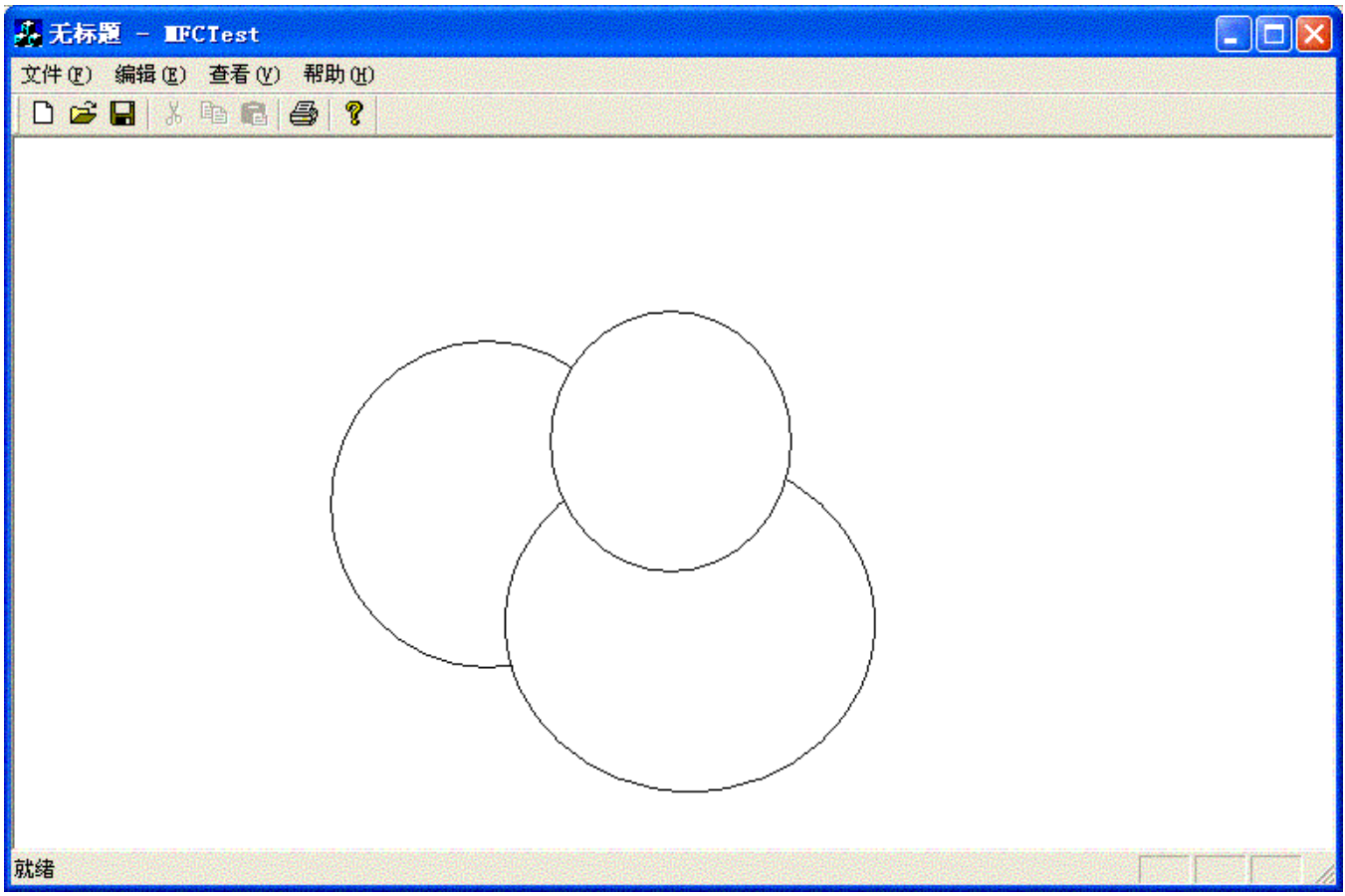
我们加入以上两个消息映射在 View 类中，请大家注意，一定要在 View 类中啊，呵呵。

现在我们看到了，在两个函数中，就有着两个点，可惜我们干看着点够不着啊，为什么呢？因为这两个点的生命周期不一下，一个是在按下鼠标时的，一个是抬起鼠标时的，于是，我们要将按下鼠标时的点给保存下来，怎么保存？用一个全局变量呗，看程序：

```
CPoint pt;          //我们定义一个全局变量
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        pt=point;          //我们保存一下全局变量
```

```
        CView::OnLButtonDown(nFlags, point);
}
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        dc.Ellipse(pt.x,pt.y,point.x,point.y);
        CView::OnLButtonUp(nFlags, point);
}
```

我们可以随便画几个圆：




大家看，后画的圆盖住了前面画的，所以这儿有一个白色的画刷，当然我们可以改了空画刷，我们马上给大家做一个，现在我们说一说上面那种全局变量的问题，当然是为了相互传递信息了，但由于我们以前说的那个原因，所以不是万不得以，我们都不定义全局的，那怎么办呢？因为我们窗体的生命周期肯定比 View 类的生命周期要小，所以我们完全可以在类中定义一个类的成员变量，也就是在类中定义一个成员变量，当然喽，类中定义的成员变量一般前面都加上一个 m_ 可能是习惯吧，我们一看就知道这是成员变量了。

我们可以在头文件中加入：

```
class CMFCTestView : public CView
{
```

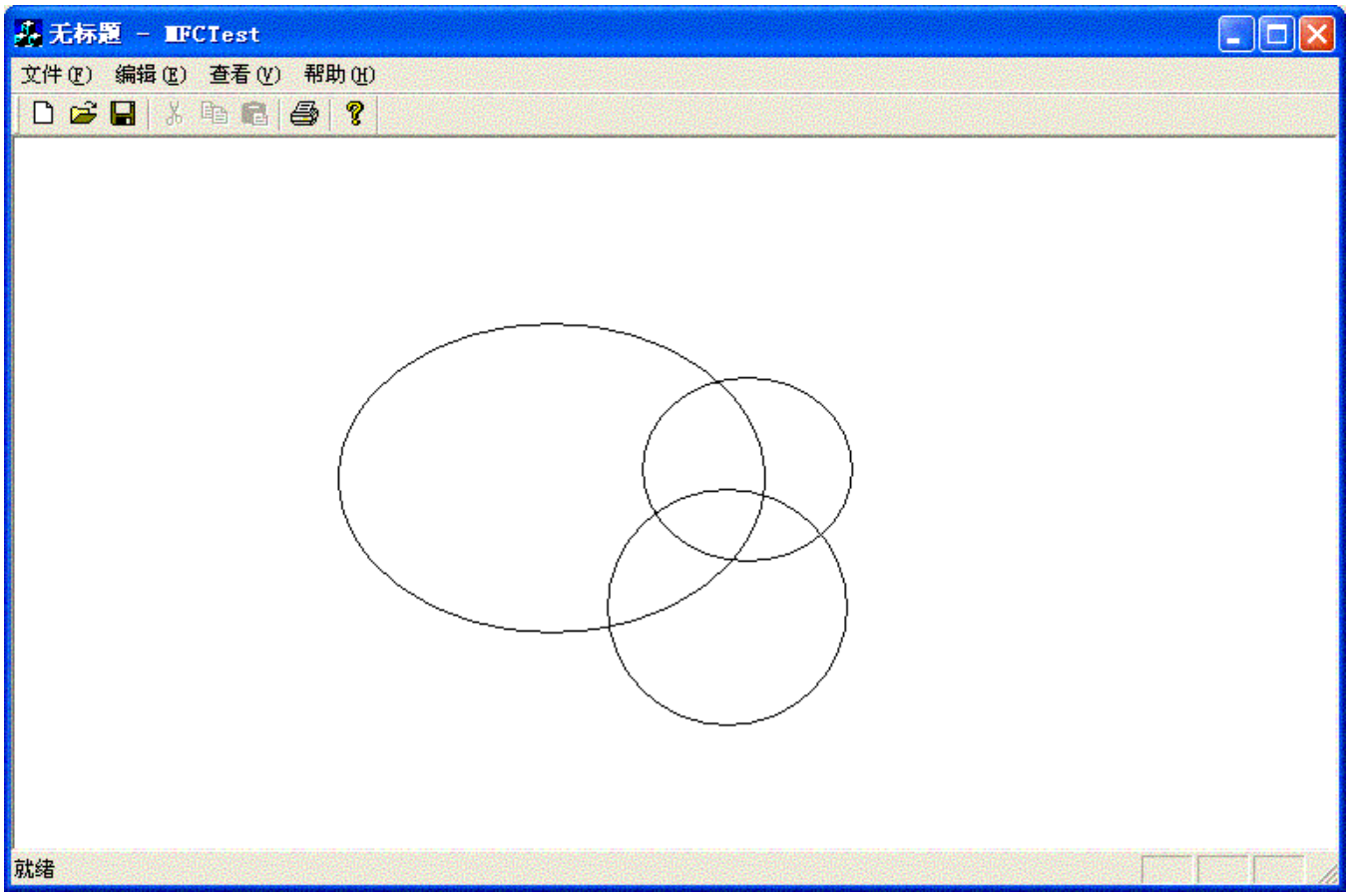
```
        CPoint m_pt;
……
```

大家知道，这默认是私有的。如果要定义一个公有的，就要在前面加 public: 了。

好了，我们再到两个消息映射函数中作如下的函数调用，但这次，我们使用一个空画刷，大家看：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_pt=point;         //这次我们用成员变量
        CView::OnLButtonDown(nFlags, point);
}
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CRect rect(m_pt,point);         //我们定义一个矩形结构
        dc.SelectObject((HBRUSH)GetStockObject(NULL_BRUSH));         /
/我们这次用一个空画刷，这儿的定义大家习惯吧？呵呵
        dc.Ellipse(rect);         //方便吧？
        CView::OnLButtonUp(nFlags, point);
}
```

怎么样？画几个圆试试：



大家看上面的图，几个圆之间没有覆盖了吧？但理论上，上面的画刷需要每次用过就删掉，当然，我们下面会教大家如何去删掉画刷。

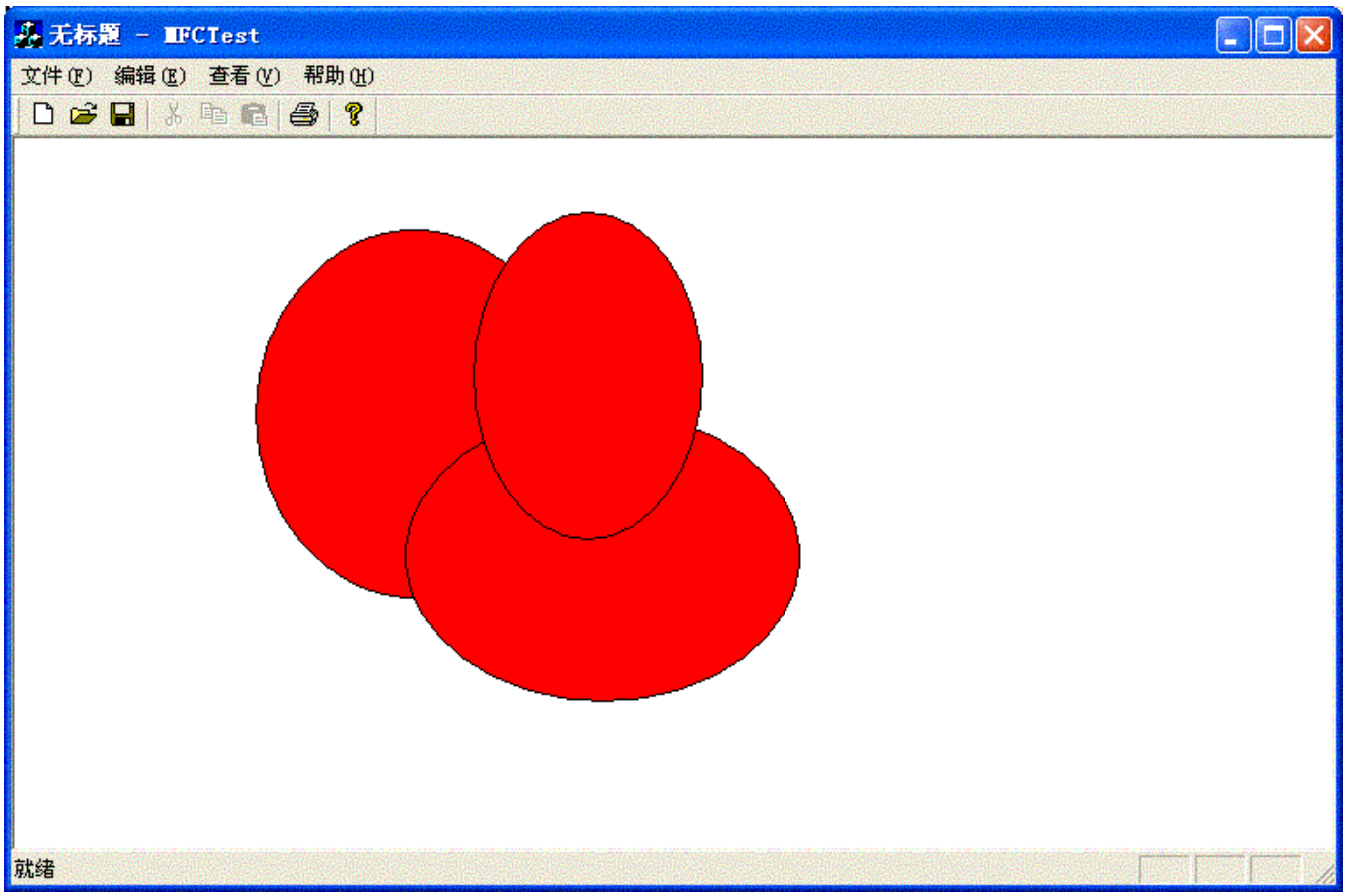
既然可以选择画刷了，可系统为我们做的画刷毕竟有限，所以我们也希望能做自己的画刷，如红色的，怎么做呢？

很容易，原来啊，系统也给我们定义了画刷与画笔的类：
CBrush
CPen

当然，我们去定义一个画刷，然后创建，当然喽，创建画刷有好几种方法，我们先选择一个简单的：

```
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
	CClientDC dc(this);
	CRect rect(m_pt,point);
	CBrush brush;
	//我们在画图之前必须选入画刷，否则还是用默认的画刷画图
	brush.CreateSolidBrush(RGB(255,0,0));//当然要先创建
	dc.SelectObject(&brush);		//再选进我们的设备描述表，下面就可以用画刷画图了。
	dc.Ellipse(rect);

	DeleteObject(&brush);		//用过当然要删去，这是原则，希望大家形成习惯
	CView::OnLButtonUp(nFlags, point);
}
```

好了，这下我们创建了一个红色的画刷，那么我们用这个画刷来画几个圆吧：




当然喽，大家一定要记得，在用过之后，记得删去画刷句柄（什么？这里来有句柄？啊，被 CBrush 包装起来了，呵呵）。

当然，上面的创建过程和选入过程可以同时进行，两个函数并成一个，看程序：

```
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CRect rect(m_pt,point);

        dc.SelectObject(CreateSolidBrush(RGB(255,0,0)));
        dc.Ellipse(rect);

        CView::OnLButtonUp(nFlags, point);
}
```

哎哟，出现一个大问题，没了画刷的定义，我们怎么删啊？

呵呵，我们可以选入一个系统画刷，于是为我们返回当前画刷，删就去行了：

```
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
```

```
        CClientDC dc(this);
        CRect rect(m_pt,point);

        dc.SelectObject(CreateSolidBrush(RGB(255,0,0)));
        dc.Ellipse(rect);

        DeleteObject(GetStockObject(WHITE_BRUSH));
        CView::OnLButtonUp(nFlags, point);
}
```

当然，你怎么知道我们的程序原来用的是白色画刷呢？所以我们一般正常做程序时，都使用如下的代码，先将系统画刷保存下来，然后选入我们以前的老画刷。

```
void CMFCTestView::OnLButtonUp(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CRect rect(m_pt,point);

        CBrush brush;
        brush.CreateSolidBrush(RGB(255,0,0));
        CBrush *pOldBrush=dc.SelectObject(&brush);          //我们选入一个画刷的同时，保存老画刷

        dc.Ellipse(rect);         //  画图

        dc.SelectObject(pOldBrush);          //选入老画刷
        DeleteObject(brush);          //删去画刷，不用了


        CView::OnLButtonUp(nFlags, point);
}
```

上面的做法是标准做法，我们以后做任何一款软件，都要记得使用过的资源要及时交还系统，养成一个良好的习惯，以便我们以后做出的程序显得更加强壮。

当然，像上面的选入空画刷以后，最后也要删除去创建的画刷，我们上面并没有做到，请大家自己去将最后工作做好。

我们讲了这么多，但我们这一讲却不准备与大家详细讨论这些，你说怪吧？那么牛刀要与大家讨论什么呢？

OnDraw 函数：

大家一定会奇怪，怎么没看见响应 WM_PAINT 的消息呢？原来啊，OnDraw 其实就是 OnPaint 调用的消息（什么？在哪里？我怎么没看见？）。大家别忙，这事情其实让老爸给“代理”了。我们不得不去看盾老爸，他那儿有什么新东东：

```
void CView::OnPaint()
{
        // standard paint routine
        CPaintDC dc(this);
        OnPrepareDC(&dc);
        OnDraw(&dc);
}
```

怎么样？第一次去拜方子类的老爸吧，呵呵，我们再去看看定义怎么样啊？

```
virtual void OnDraw(CDC* pDC) = 0;
```

看，果然是一个虚函数，可后面的 =0 是什么意思呢？

原来啊，这种后面有 =0 的类，并不是说这个类的值等于零，而是表示这个类的函数在本父亲那儿不去给实现了，那咋办？这种类只有送给儿子去实现了，对于有这种函数的类，我们称之为“抽象类”，也就是这个类就是摆摆老资格，一切这种不实现的函数全都交给儿子去办，所以说，这儿的类，其实父亲那儿是不参与整个函数的实现的，所以这个函数必须得有儿子来实现，于是，就有了我们的 OnDraw 函数。

当然了，大家看父亲的确是调用了 OnPaint ，也就是这个消息映射父亲的确是做了，并且将指针指给了 OnDraw ，所以儿子可以顺利地得到父亲那儿来的指针，去画图，那我们就画一个图吧，画什么呢？画一个红圆圈吧：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CMFCTestDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CRect rect;
        GetClientRect(rect);
        CBrush brush;
        brush.CreateSolidBrush(RGB(255,0,0));
        CBrush *pOldBrush=pDC->SelectObject(&brush);

        pDC->Ellipse(rect);

        pDC->SelectObject(pOldBrush);
        DeleteObject(brush);
}
```

这次，我们运行程序，按里面的菜单 文件/打印预览 看图：

这次看到什么了？打印预览中竟然也有一个红色圆圈，虽然小了点，但还是有的，这是怎么回事？原来啊，打印机的程序也调用这个 OnDraw 函数。

明白了吧，打印机那儿送来的 CDC 是打印机的 设备描述表，打印预览

那儿送来的是打印预览的设备描述表，而我们的窗体调用的是窗体传进的设备描述表。

什么意思有点明白了吧？也就是说，我们在 OnDraw 中的程序其实并不仅仅是让我们窗体去调用的，当然，如果你不信，我们就再来响应一个 WM_PAINT 消息吧，再次将老爸的权利拿过来，还是老一套，做消息映射：

```
void CMFCTestView::OnPaint()
{
        CPaintDC dc(this);
}
```

程序给我们定义了一个 dc 请注意，不是指针，呵呵，那么我们这次使用这个指针去画一个方块，当然喽，请大家不要删除以前的 OnDraw 程序，我们看程序：

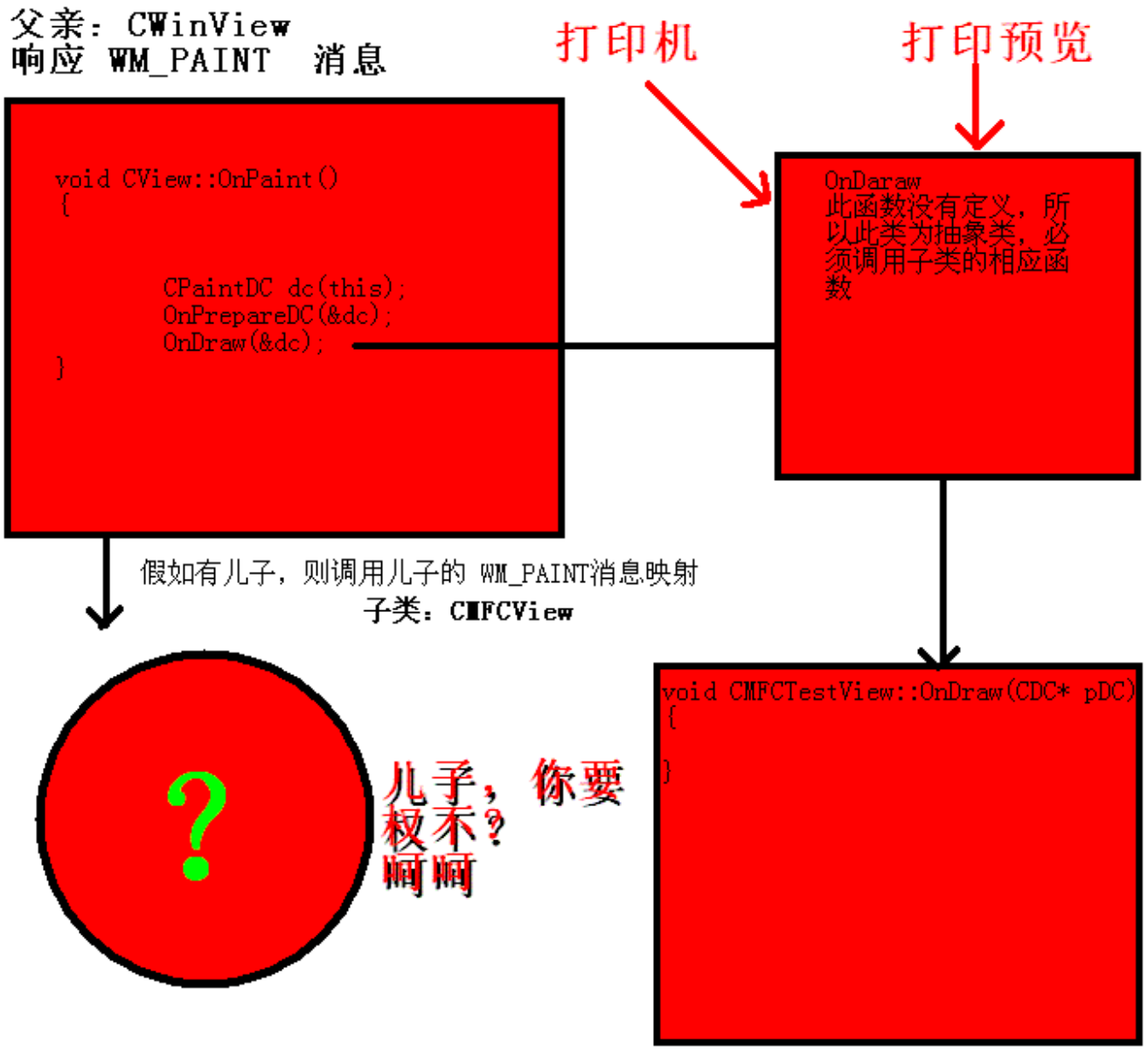
```
void CMFCTestView::OnPaint()
{
        CPaintDC dc(this);
        CRect rect;
        GetClientRect(rect);
        rect.left=rect.top=25;
        rect.right-=25;
        rect.bottom-=25;
        CBrush brush;
        brush.CreateSolidBrush(RGB(255,0,0));
        CBrush *pOldBrush=dc.SelectObject(&brush);

        dc.Rectangle(rect);

        dc.SelectObject(pOldBrush);
        DeleteObject(brush);

}
```

请大家运行程序，看到一个红色方块了吗？太大了，这下大家再去打开打印预览，看看有什么？

怎么回事？还是红色圆圈啊!!!

当然，充分证明打印预览调用的是 OnDraw 所以我们在屏幕上画东西还需要打印出来的话，大家一般都放在 OnDraw 去实现，而除非你不需要打印的话，你可以去重新映射一个 WM_PAINT 消息。

当然，我们的程序是怎么运行的，请看下面的图：



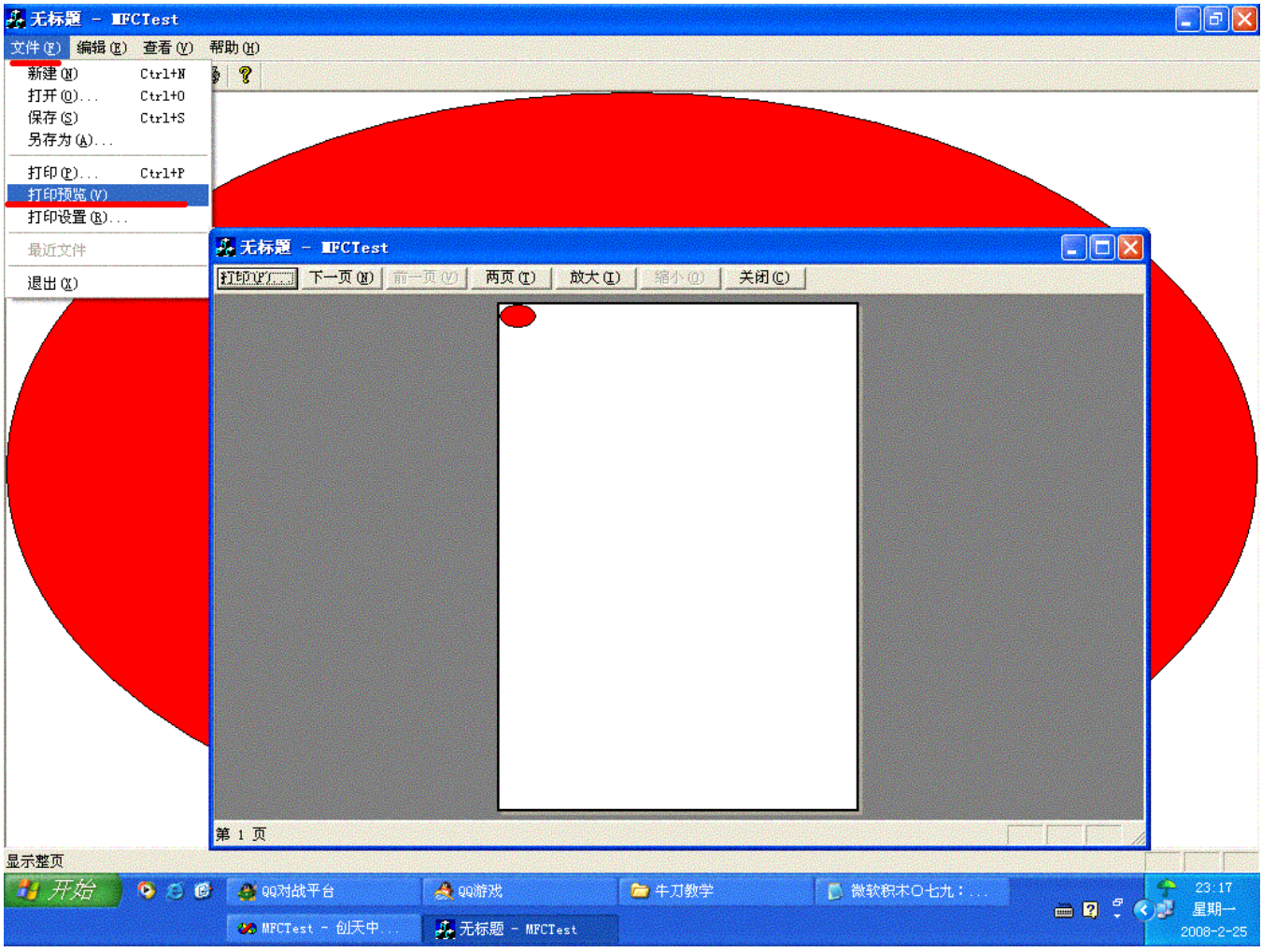

父亲已经将 OnPaint 消息响应了，并且随便儿子响应不响应，如果儿子响应，则父亲就交权，于是，在屏幕上就只显示儿子 OnPaint 消息中的函数了，于是 OnDraw 函数就在屏幕上就没有机会去调用了，当然，你在儿子的 OnPaint 消息响应中也可以去调用自己的 OnDraw 消息去画图，当然，或者也可以在以后再次调用一下父亲的 OnPaint 函数，让父亲去继续处理，那就随便你的想法了，呵呵。

当然，如果儿子一旦不去响应了，老爸你自己去调用吧，所以老爸就可以顺理成章地调用 OnDraw 函数了，而 OnDarw 却没有定义，所以必须去调用儿子，那种自己不能调用的函数，我们称之为“纯虚函数”，有纯虚函数的类我们

称之为“抽象类”，抽象类是不能定义对象的，也就是说，只有当老子的资格了，没办法，呵呵。

于是，就去调用儿子的 OnDraw 去画图了，于是我们只需要通过 OnDarw 就可以在屏幕上画图了，事实顺序就是这样。

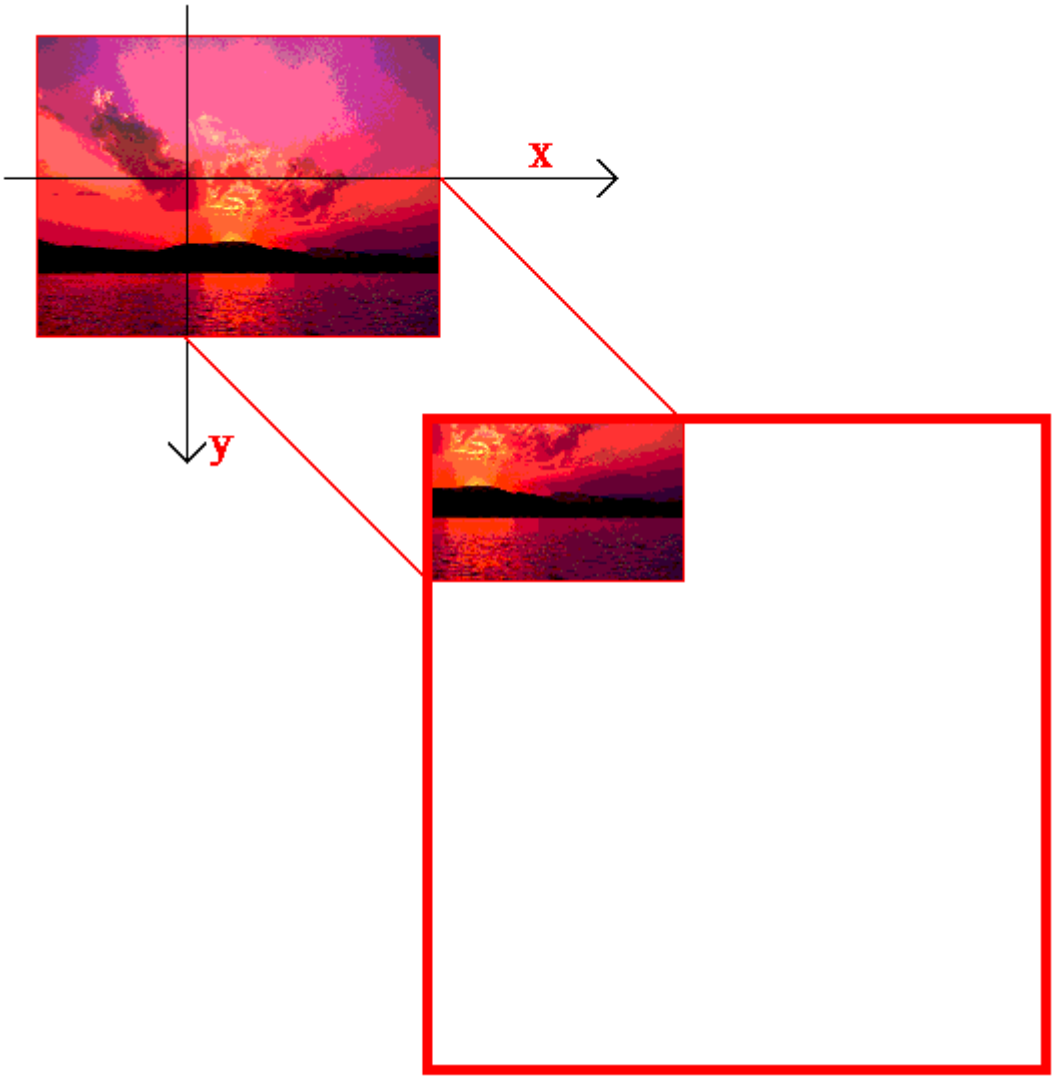
是不是有点绕？如果你觉得绕的话，那就只有慢慢地去理解与体会了，必须将其中的关系理清，因为我们从现在开始，我们就正式进入了微软 MFC 的内部，想必像牛刀这样详细讲解 MFC 的书不多，呵呵，但牛刀不管理解的正确与否，是与大家讲了，并且讲了十几讲，希望对大家有所帮助。



**微软积木〇八〇：窗纸探秘**

还记得牛刀给大家画过一个图吗？牛刀叫大家将窗纸想像成两层，前一层我们称之为“视窗”，就是我们看得见的部分，而后面的一层我们称之为“视图”。

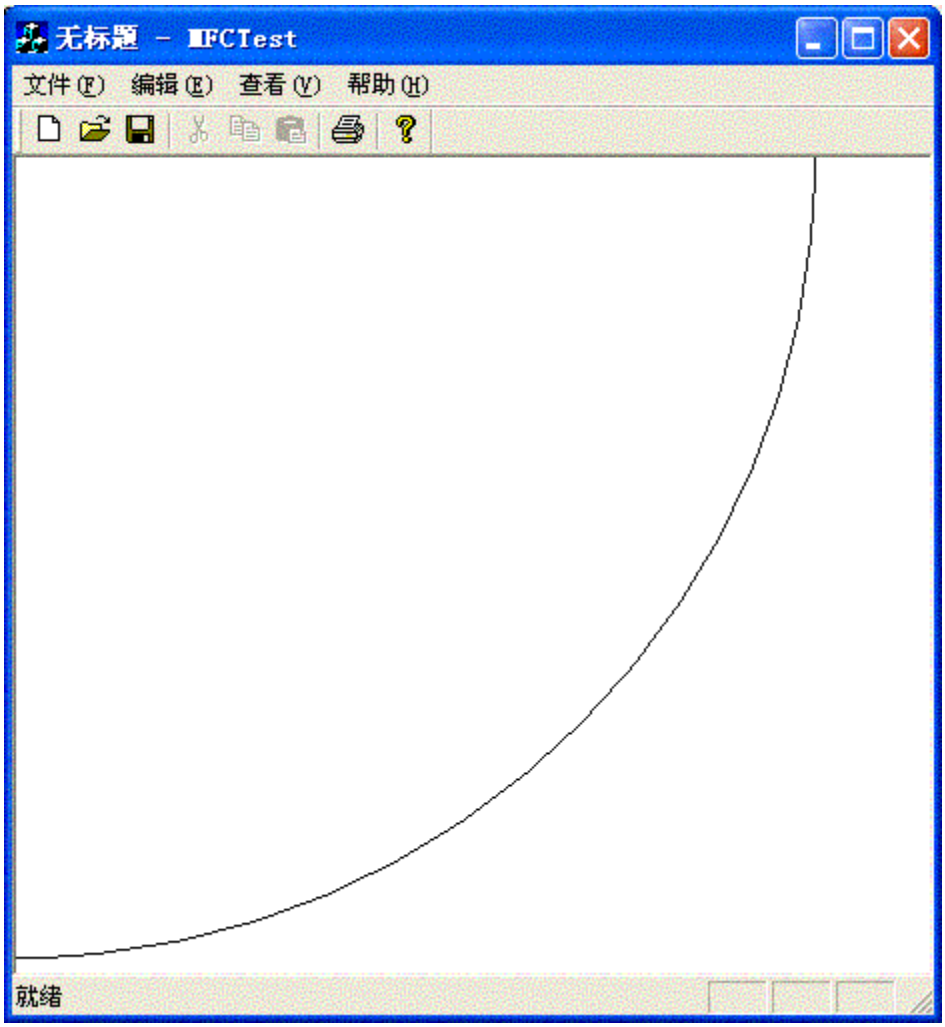
有些地方也会称前面的设备坐标，后面的为逻辑坐标，什么意思呢？举个例子吧，看图：




我们上面画坐标的那一幅图就位于一个逻辑坐标系中，请大家注意，我们这个坐标轴是向右 x 递增，向下 y 递增，与数学中的直角坐标系有所不同，为什么呢？因为我们显示设备甚至包括我们写字都是从左向右，从上向下的，而坐标系虽然有四个限象，但我们显示器的窗口上却只能显示一个限象中的一部分内容，再多一点的内容就显示不下了，于是，就不得不将要显示的内容在背面重新做一层，而前面显示的又是一层，也就是说，我们在窗口中画出来的东西，不一定都能够显示出来，请大家看程序，我们在 OnDraw 函数中去作图：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        pDC->Ellipse(-400,-400,400,400);
}
```

大家看，并没有出现任何不适的情况，反倒是成功地显示了四分之一个圆




那么，这个圆显然不是在我们屏幕上画的，而是在背后的另一张低上，也就是说在“视图”上画的，而我们的“视口”只显示了其中的一部分，我们称这种不同的画图方式称之为“映射”，也就是说，我们后面有一束灯光，将背面的低上的一部分内容投影到了我们前面的纸上，也正因为如此，我们就出现了“映射模式”的概念。

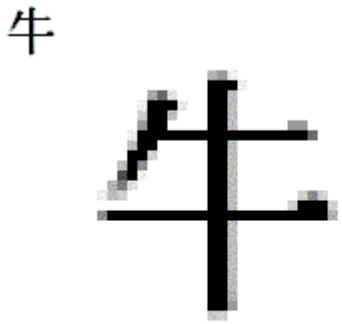
我们以前与大家说过，我们可以用不同的灯光来将后面的东西向前投影，那我们先讨论几种不同的灯光效果吧。

如果大家看前一章的有关打印预览的，可能你当时主很纳闷，为什么我们画一个很大的圆圈，结果打印预览出来的那么小呢？

这就是我们第一个映射模式：

MM_TEXT

这种映射模式是以像素为单位的，在数学中，我们知道，点是没有大小的，但那毕竟是理想的点，而事实上，我们显示器上，我们打印机上，点都是有大小的，并且大小还不一样，一般的激光打印机因为聚焦好，点打得就格外地小，而如果有针式打印机，那么一个点就会很大了，于是，我们为了更形象，请大家看下面的图，我们用不同大小的点显示同一个汉字：



大家看，后一个“牛”，是不是有点马赛克？那是牛刀为了表示后一个点比较大的意思，后一个点的大小是前一个的五倍，那么这个字的大小显然就是前面一个字大小的 25 倍，没办法，人家占的是面积啊，呵呵。

而显示器上的点其实还就蛮大的，你仔细看，肉眼可以分辨出来，当然如果你用放大镜，或者干脆滴上一滴水，再看看，那点就有点吓人了，并且还有三种不同的颜色。

当然相比较而言来说，激光打印机上的点就要小得多，所以同一幅图片，你用激光打印机打出来不一定有显示器上的效果理想，于是，人们就会想出其他的种种办法，来统一这些尺寸，于是，就形成了其他种种的映射模式：（注：上面的 MM_TEXT 我们称之为“像素模式”，后面一并与大家讨论）

固定比例映射模式

| 映射模式 | 逻辑单位 |
| --- | --- |
| MM_LOENGLISH | 0.01 英寸 |
| MM_HIENGLISH | 0.001 英寸 |
| MM_LOMETRIC | 0.1mm |
| MM_HIMETRIC | 0.01mm |
| MM_TWIPS | 1/1440 英寸 |

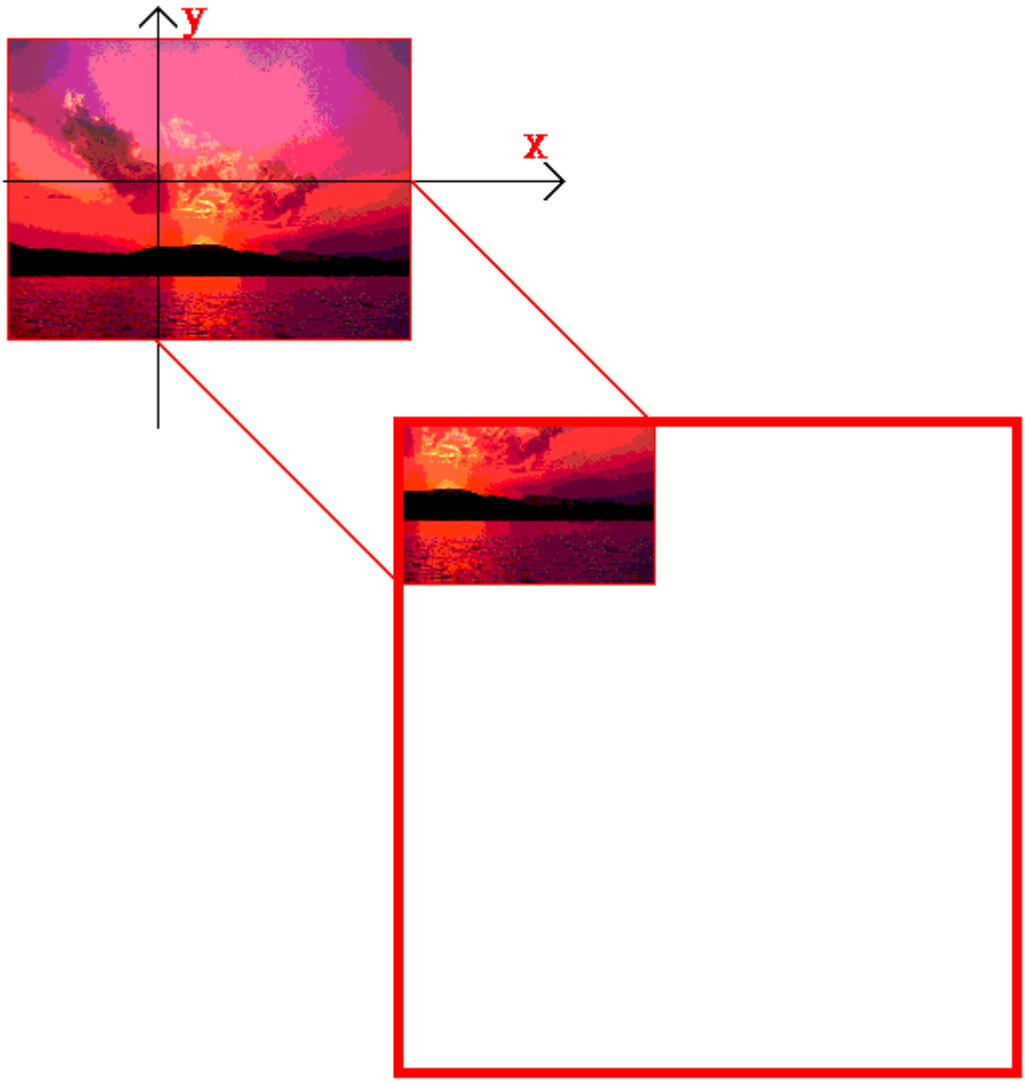
那么这些是怎么做到的呢？原来啊，虽然显示器的点显示的与打印机不同，但显示器是知道显示器点大小的，而尺寸/点的大小=每个尺寸需要打印点的个数，打印机也一样，只要用这用模式，我们显示器，打印机都会将其直接转换成若干个点去打印，看程序：（打印模式的设置，我们用 SetMapMode）

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        pDC->SetMapMode(MM_LOMETRIC);
        pDC->Ellipse(0,0,500,500);
}
```

怎么回事？显示的东西到哪儿去了？你一运行程序，是不是发现屏幕上一无所有呢？

原来啊，这种固定比较的模式，是向右递增，向下递减的。

也就是说，我们显示区域其实都是显示的全负那一边的图形，看图：



也就是说，你想显示的是右上角的图形了，而不是右下角，而右下角的坐标是负的，所以程序应该这样改：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CMFCTestDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDC->SetMapMode(MM_LOMETRIC);
        pDC->Ellipse(0,0,500,-500);
}
[code]
```

当然，是不是这样？可不可以这样理解呢？我们做个程序，在屏幕上画一些坐标点试试，这次我们就需要使用 CString 了，看程序：

```
[code]
void Drawxy(CDC *pDC)
{
        int i;
        CString str;
        for(i=-2000;i<2000;i+=200)
        {
                str.Format("%d",i);
                pDC->TextOut(i,0,str);
        }
        for(i=-2000;i<2000;i+=200)
        {
                str.Format("%d",i);
                pDC->TextOut(0,i,str);
        }
        pDC->MoveTo(0,-2000);
        pDC->LineTo(0,2000);
        pDC->MoveTo(-2000,0);
        pDC->LineTo(2000,0);
}
void CMFCTestView::OnDraw(CDC* pDC)
{
        CMFCTestDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDC->SetMapMode(MM_LOMETRIC);
        Drawxy(pDC);
}
```

我们上面的 Drawxy 专门画坐标，我们做成了程序了，大家以后一调用就行了从我们画的坐标上来看，我们一个以 0.1mm 为单位的，竟然整个屏幕画起来都不到 2000，怎么回事呢？呵呵，你的屏幕有 200mm 也就是宽度超过 20cm 吗？显然不至是吧，呵呵，当然喽，从图中，我们也看到（注：牛刀不贴

图了，贴太多没意思了，你自己的 VC6.0 也一样地出软件，是不是，还比牛刀出的图清楚)：的确是向右增加，向下递减的。圆点在左上角，对了，这个圆点是可以移动的哟，看，由于我们的坐标是向右增，向下减，那么我们就将坐标原点移动到 (200,-200) 处吧，移过来后，大家再看显示结果：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CMFCTestDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDC->SetMapMode(MM_LOMETRIC);
        pDC->SetViewportOrg(200,200);
        Drawxy(pDC);
}
```

很可惜的是牛刀移动了好几次，最后发现，圆点的移动，只能按设备坐标点移动，也就是说，一次只能移动 200 个像素，并且向右增，向下增的，那我们没有办法，因为我们这次是对“视窗”作移动啊，而我们的模式设置对视窗是没有用的，我们的视窗只起到一个模式映射的作用，什么意思呢？像我们上面的确是将圆点移动了是吧？不错，移动的是在 OnDraw 函数中移动的，而出了这个函数就对这些数据统统地不起作用了，举例来说，我们去响应鼠标的移动消息：

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CString str;
        str.Format("point.x=%d,y=%d     ",point.x,point.y);
        dc.TextOut(0,0,str);
        CView::OnMouseMove(nFlags, point);
}
```

请大家看,我们的结果,还是以设备的坐标为坐标去报我们鼠标当前的坐标，于是，就形成了极不舒服的感觉，那么怎么办？为了解决这种问题，我们的 DC 给我们提供了一套函数：

LPtoDP
DPtoLP

用来将设备坐标与逻辑坐标进行转换。

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CString str;
        dc.DPtoLP(&point);
        str.Format("point.x=%d,y=%d     ",point.x,point.y);
        dc.TextOut(0,0,str);
        CView::OnMouseMove(nFlags, point);
}
```

好像还没有用啊，这是怎么回事？

我们说过，我们对这些东东的设置都是有生命周期的，在我们对 OnDraw 函数设置后，但在对鼠标移动中并没有作坐标设置啊，所以我们在鼠标移动的程序中，还要重新去设置一下坐标与映射模式。

```
void CMFCTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CString str;
        dc.SetMapMode(MM_LOMETRIC);
        dc.SetViewportOrg(200,200);
        dc.DPtoLP(&point);
        str.Format("point.x=%d,y=%d    ",point.x,point.y);
        dc.TextOut(0,0,str);
        CView::OnMouseMove(nFlags, point);
}
```

于是，我们的系统就可以正确地显示坐标了，当然，如果我们什么函数中都要这样设置，那岂不是太太太太太太麻烦了？于是乎……

大家还记得我们上一讲给大家介绍的那位老父亲吗？

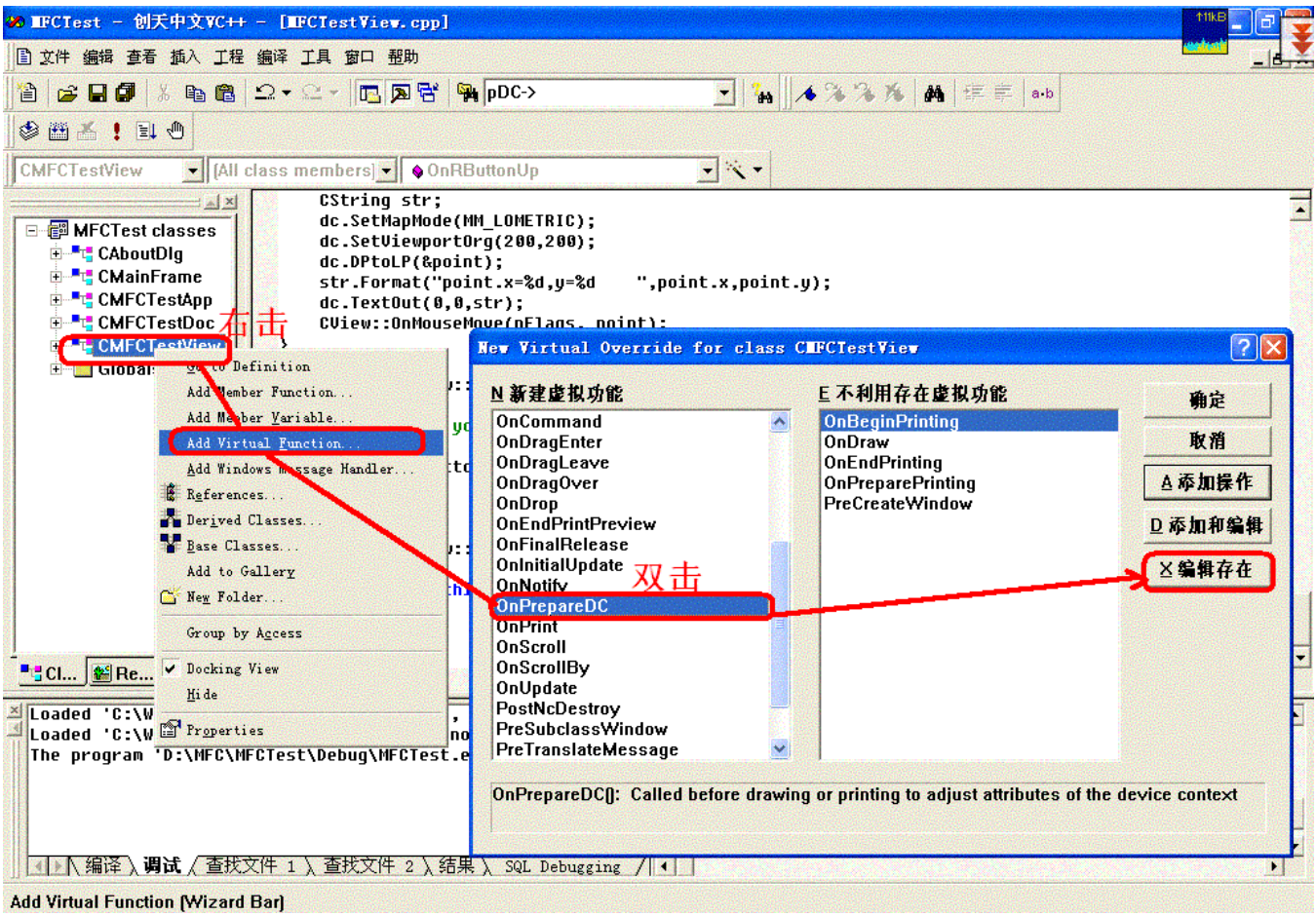
```
void CView::OnPaint()
{
        CPaintDC dc(this);
        OnPrepareDC(&dc);
        OnDraw(&dc);
}
```

老父亲那定义了 dc ，我们说过了，调用 OnDraw ，我们说过了，那么还有 OnPrepareDC 我们还没有说，那个函数就是专门用来设置我们窗口的模式的。

那个函数不是消息映射，而是一个虚函数，所以我们要以虚函数的方式来继承，或者在我们使用滚动窗体时，并不要去改变她。

什么意思？也就是说，大家看我们每一个窗口边上的滚动条，如果我们向下拖动，那么里面的内容就向上跑是不是？那是老父亲给的，而做此事情的函数就是老父亲的 OnPrepareDC 函数。

当然，我们去让儿子去抢抢老父亲的功劳吧，我们在 CMFCTestView 上左击，这次选择 Add Virtual Function...




双击相应虚函数，然后单击“编辑存在”，就可以将上面的设置模式的代码挪到这里来了，当然，在需要设置模式时，调用一下这个函数就一切 OK 了。

```
void CMFCTestView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
        CView::OnPrepareDC(pDC, pInfo);
}
```

当然，如果你没有映射老爸的 WM_PAINT 消息，OnDraw 中就没有必要去调用了，如果你映射了相应的函数，那一切后台工作就需要自己去做喽。

注意一点就是，这个 OnPrepareDC 函数调用老父亲的函数最好不要删，除非你自己现有的功底能够去替代老爸的工作，你才可以删，不然你的程序就可能会出现异常，往往你自己都可能不知道怎么回事，因为这里的好多东西是很难调试的甚至有些映射模式与我们的思路是相反的。

当然，还有两种映射模式是“可变比例”的映射模式，有两种：
MM_ISOTROPIC
MM_ANISOTROPIC

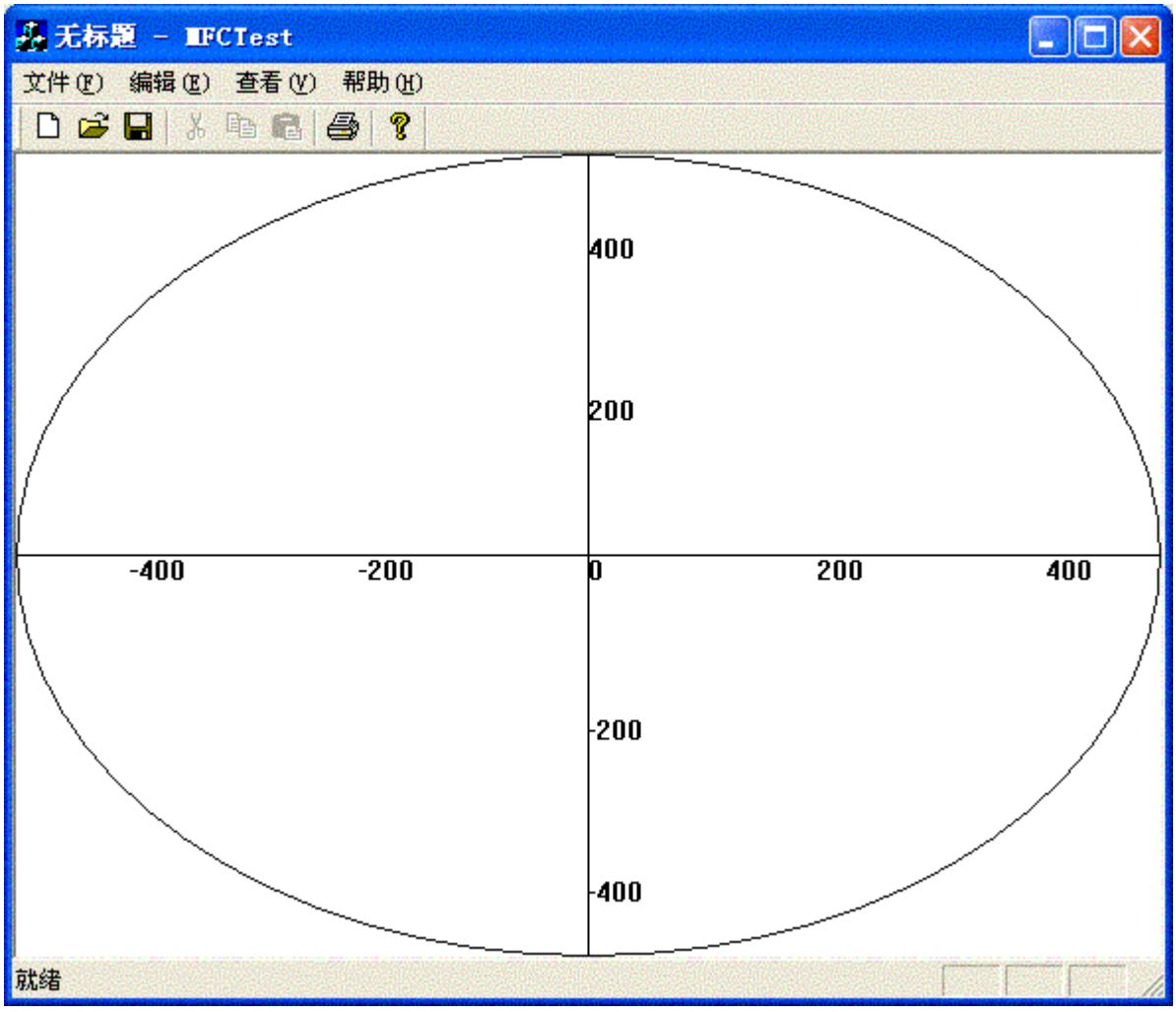
上一种的纵横比例是 1:1 ，也就是说是等比变换，也就是说，一个圆怎么

变换总是一个圆，而下一个变换就有可能将一个圆拉成一个椭圆。

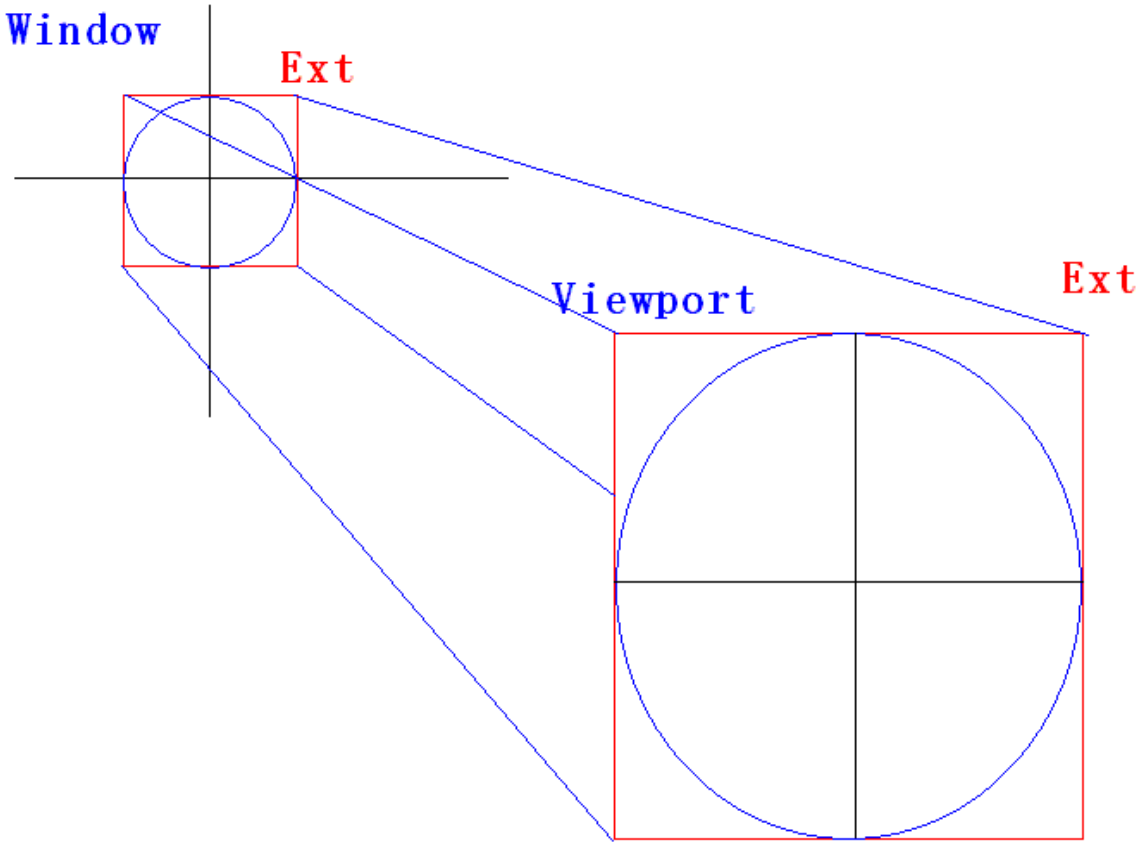
什么意思？也就是说，以上的 MM_TEXT 与固定比例因子的变换是用平行光去照射我们的屏幕的，而这里就是用一束类似于点光源去照射我们的屏幕，于是，就会出现千奇百怪的现象。

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CRect rect;
        GetClientRect(rect);
        pDC->SetMapMode(MM_ANISOTROPIC);
        pDC->SetWindowExt(1000,1000);
        pDC->SetViewportExt(rect.right,-rect.bottom);
        pDC->SetViewportOrg(rect.right/2,rect.bottom/2);
        pDC->Ellipse(-500,-500,500,500);
        Drawxy(pDC);
}
```

大家看下图：




这儿的程序可能有理解上有些难度，我们再画一幅图与大家去理解一下：




也就是说，上面的模式是可变比较映射模式，而 Window 就是指的逻辑坐标，而 Viewport 就是指的视口坐标，

也就是通过 SetWindowExt 与 SetViewportExt 去先行设计一下视口与逻辑坐标的范围，当然可以不同，然后，由系统根据这两个范围的大小不同，用一定的点光源进行放大或缩小地照射，就像放电影一样将逻辑坐标（胶卷）中的影像投射到设备坐标（屏幕）上，从而使得逻辑坐标中画一个圆，不管设备坐标怎样变化，逻辑坐标中的图形都可以很容易地随着设备坐标的变化而变化。

就是这样，不知道能不能明白？

当然，用这种不规则的映射，可以很容易地实现数学坐标的屏幕实现，就可以变得很方便了，请大家自由地更改上面的一些参数，然后运行程序，仔细地去思考，好多教材会给你一些公式，牛刀在这儿不给，因为那些公式太抽象了，并且很是绕人，真的如果大家有兴趣，可以自己去算出这些公式，然后正常使用，我们一般做程序时，只要映射模式能够设置好就一切 OK 了，我们并不需要去在整个编程过程中去考虑我们的坐标映射，而是一旦设置好了坐标映射，我们就会在我们的映射坐标中去制作我们的软件。

也就是说，我们做程序，千万别将做程序的过程弄倒了，否则就会很麻烦。

洗完澡穿衣服很舒服，穿好衣服再用水冲可能就很麻烦，喝茶之前吹吹，可能就比喝进去之后再吹要有效得多，事实就是如此，我们做软件就是需要知道我们该先做什么，后做什么，只有那样，我们做起软件来才能够游刃有余。

**微软积木○八一：信手涂鸦**

我们上一讲对画纸的性质有所了解，当然，我们以后如果需要，我们还要讲，为什么呢？因为我们好多东西真的不是一讲两讲能够讲了很明了的。

好了，这一讲与大家来画画儿玩吧，当然，还是用以前的点、线、圆、方、弧、弓、饼等等这些东东，

想要画图，那我们就需要有画笔，画刷，画纸。

画纸，我们上一讲与大家讲过了，下面我们先讲画笔吧：我们向系统申请一个画笔以后：

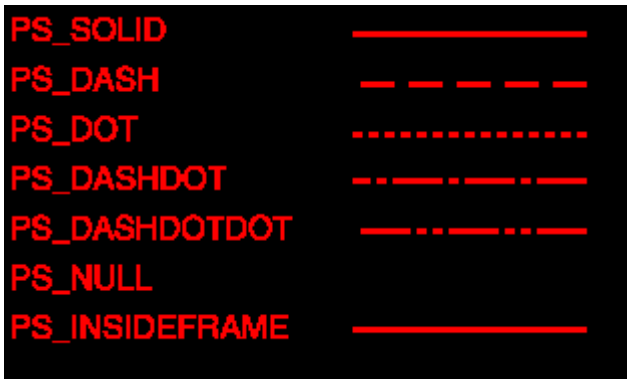
```
CPen pen;
```

我们会有好几种办法来创建一个画笔：

```
CreatePen( int nPenStyle, int nWidth, COLORREF crColor );
```

创建画笔会有三个参数，第一个参数是画笔的类型，一共有七种类型（当然，扩展的画笔会更多）。

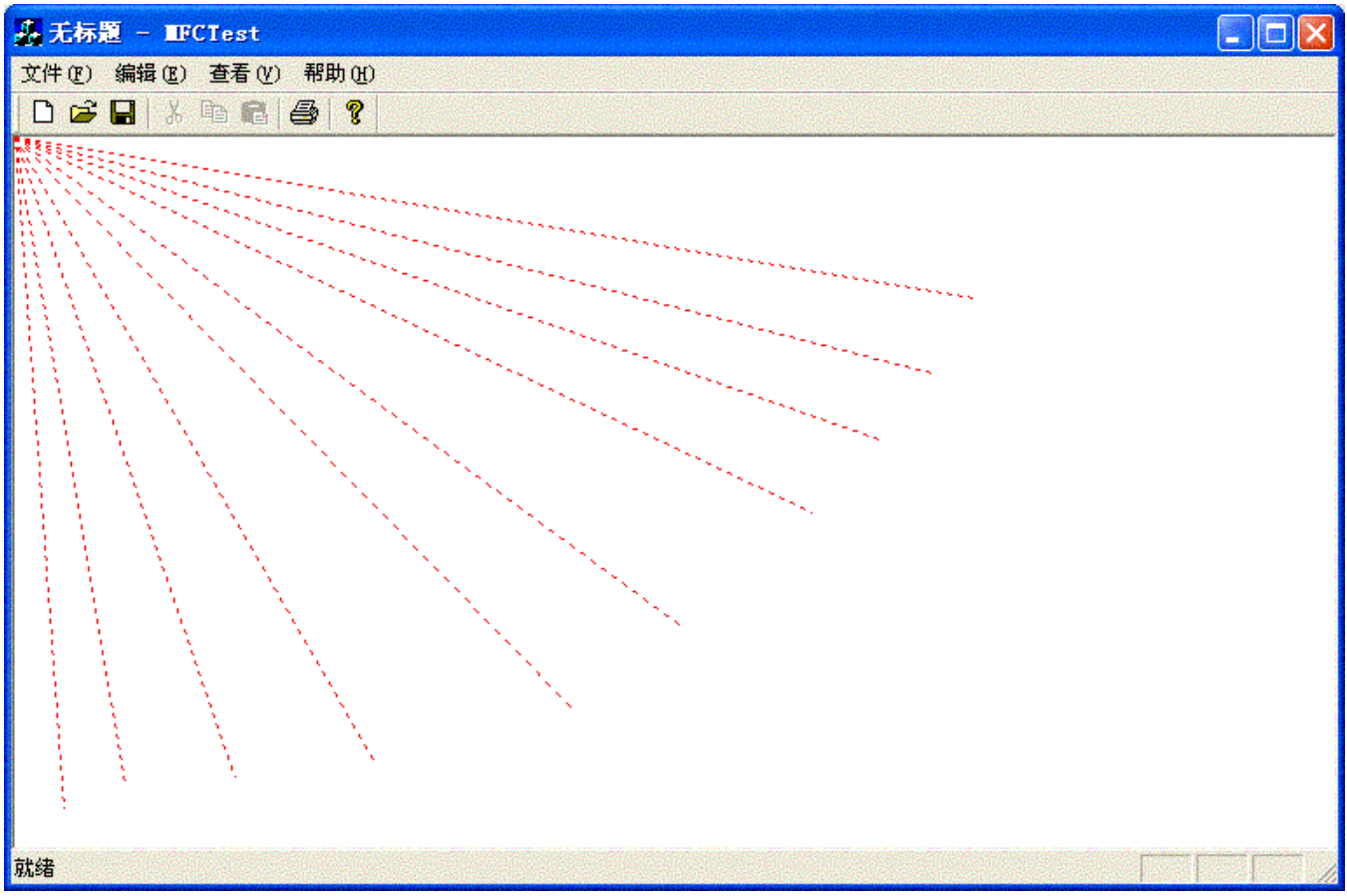
PS_SOLID　　创建一支实线画笔
PS_DASH　　创建一支虚线画笔
PS_DOT　　创建一支点线画笔
PS_DASHDOT　　创建一支虚点线画笔
PS_DASHDOTDOT　　创建一支虚点点线画笔
PS_NULL　　创建一支空画笔
PS_INSIDEFRAME　　创建一支画笔，此画笔的风格不受指定矩形的限制




当然，以上的有点线虚线的几种画笔当且仅当第二个参数为 0 或为 1　时有效，也就是说，一旦画笔的宽度大于 1，就只能画实线喽，呵呵，好了，我们做些程序看看吧，这次我们映射鼠标按下的消息：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
	CClientDC dc(this);
	CPen pen(PS_DOT,1,RGB(255,0,0));
	CPen *pOldPen=dc.SelectObject(&pen);

	dc.LineTo(point);

	DeleteObject(pen);
	dc.SelectObject(pOldPen);

	CView::OnLButtonDown(nFlags, point);
}
```

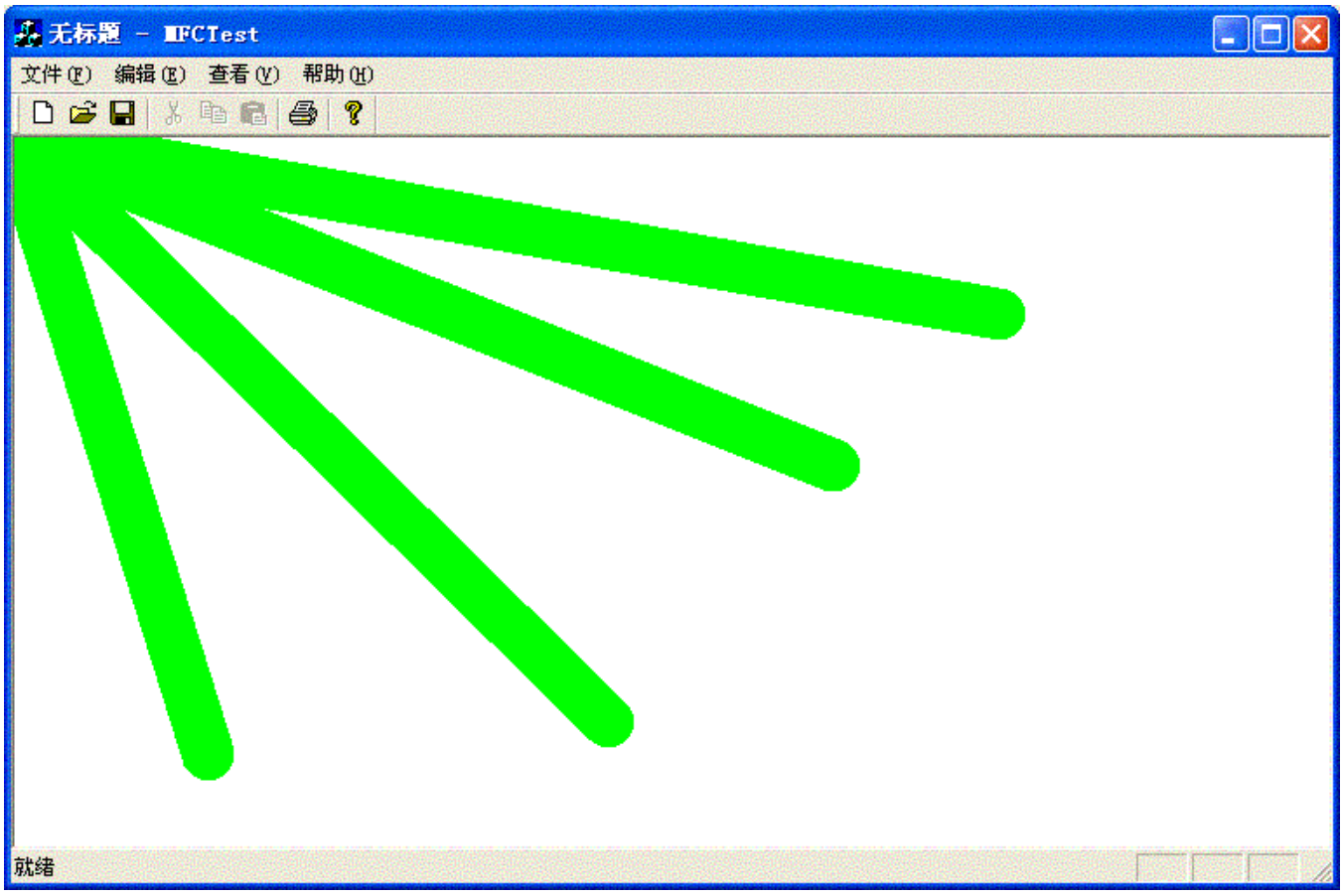
好嘛，同一个程序又一次地试验喽，呵呵。




怎么回事，怎么没用 pen.CreatePen 函数啊，呵呵，这就是类的优势，在构造函数中就将这些东东给做了，当然，我们可以将画笔改为:

CPen pen(PS_DOT,30,RGB(0,255,0));

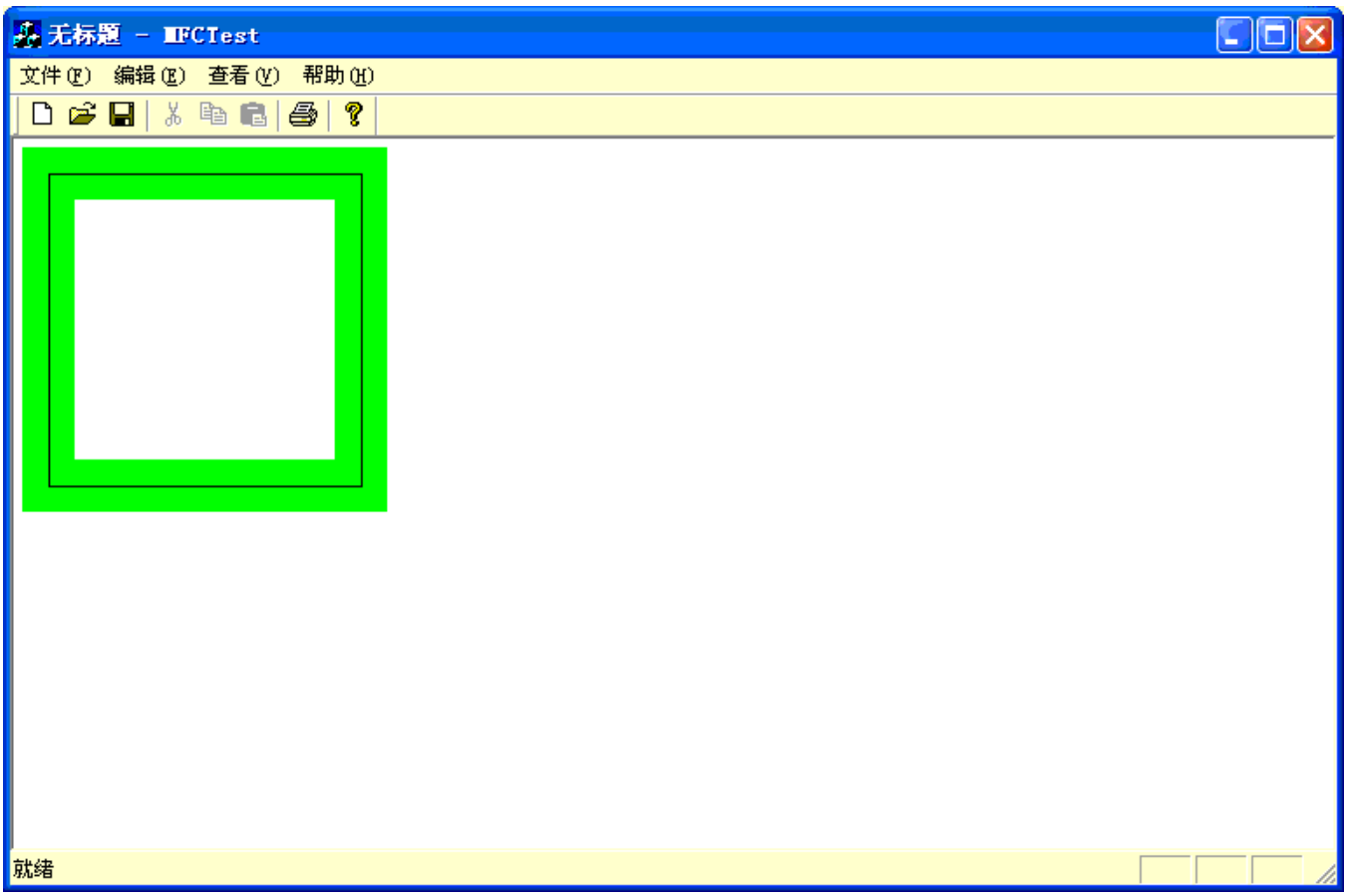
请大家注意，牛刀并没有将类型变掉，但马上我们可能就看不到类型了：




大家看比较恐怖吧，呵呵，事实就是如此，正因为如果，当我们去画一个矩形时，就会出现一些问题：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
	CClientDC dc(this);
	CRect rect(20,20,200,200);

	CPen pen(PS_DOT,30,RGB(0,255,0));
	CPen *pOldPen=dc.SelectObject(&pen);

	dc.Rectangle(rect);

	DeleteObject(pen);
	dc.SelectObject(pOldPen);
	dc.SelectObject(GetStockObject(NULL_BRUSH));
	dc.Rectangle(rect);
	CView::OnLButtonDown(nFlags, point);
}
```

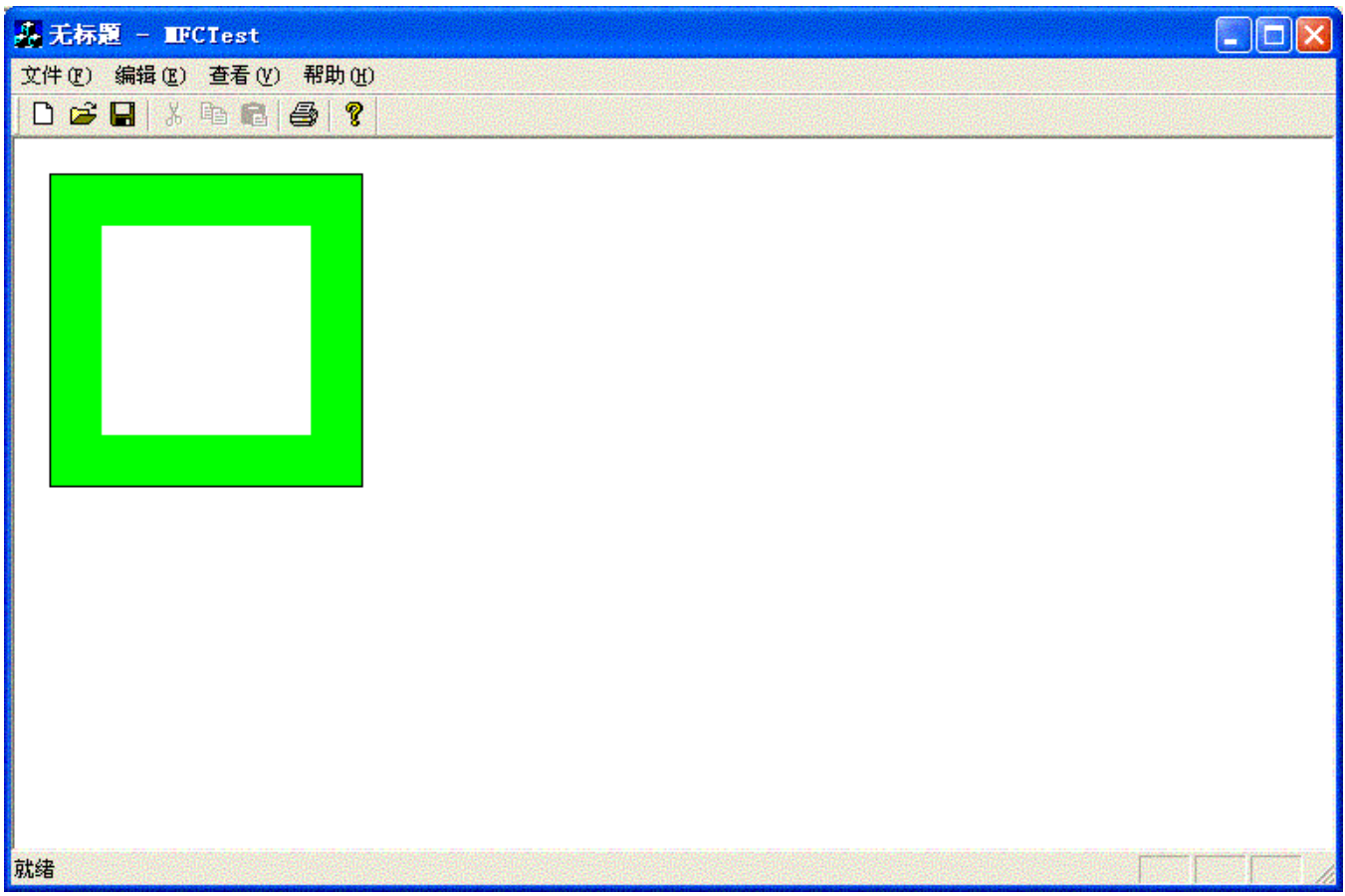
点击一下鼠标后，会出现图中效果：（理论上上面需要删除空画刷，但我们还没有讲到那儿，这儿先不删了，但以后大家做程序可不能这样，呵呵。




好了，我们将画笔类型改为第七种类型：

```
CPen pen(PS_INSIDEFRAME,30,RGB(0,255,0));
```

大家再看图：




怎么样，画笔被限定了在矩形里面了。

好了，画笔讨论完了，下面我们再来讨论画刷：

```
CBrush brush;
BOOL CreateSolidBrush( COLORREF crColor );
BOOL CreateHatchBrush( int nIndex, COLORREF crColor );
BOOL CreatePatternBrush( CBitmap* pBitmap );
```

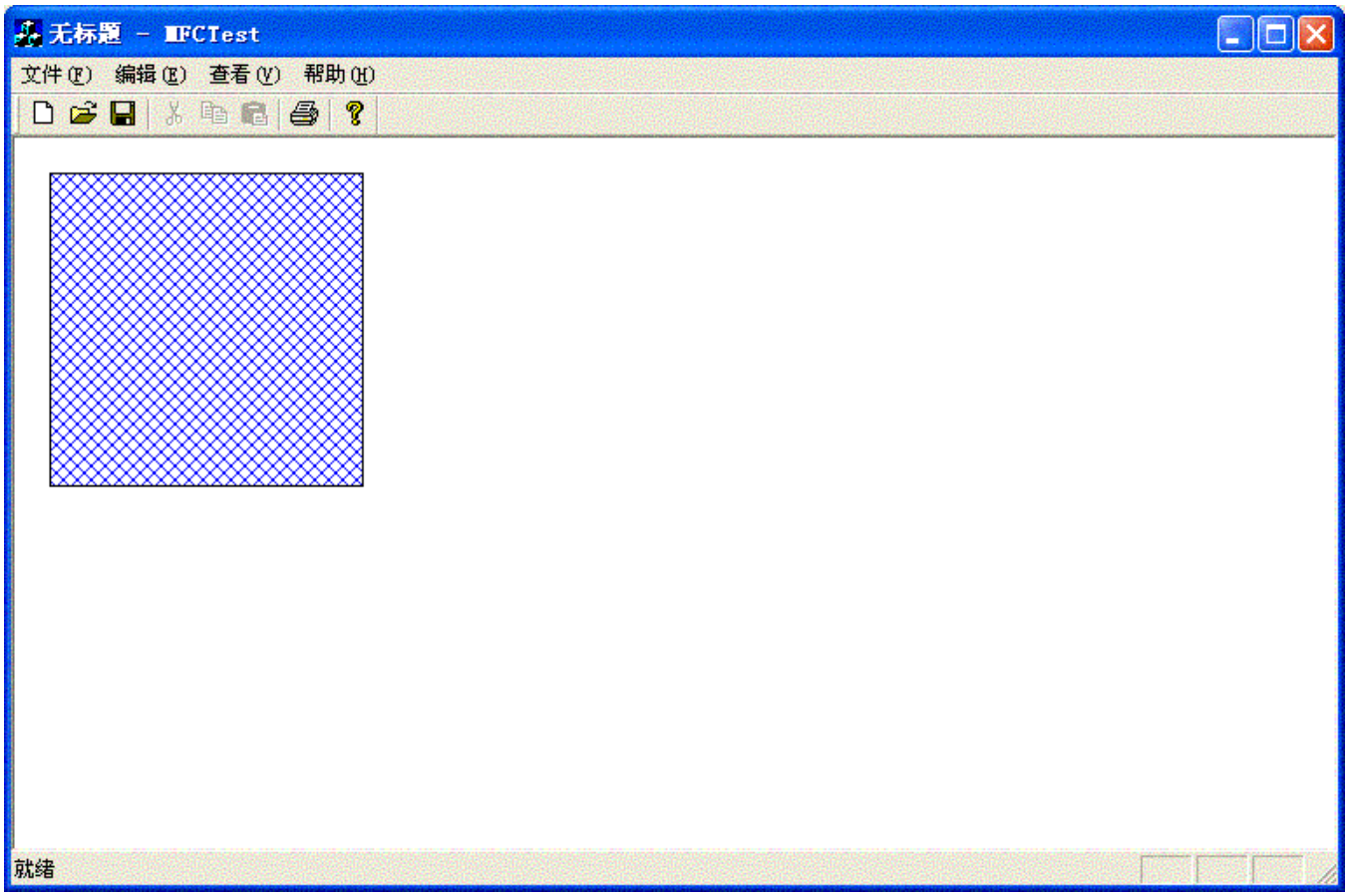
我们可以创建一个颜色画刷，可以创建一个有类型的画刷，可以创建一个位图画刷。

当然喽，颜色画刷我们以前创建过，下面我们就创建第二个画刷吧：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        CRect rect(20,20,200,200);
        CBrush brush;
        brush.CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,255));
        CBrush *pOldBrush=dc.SelectObject(&brush);
```

```
        dc.Rectangle(rect);

        DeleteObject(brush);
        dc.SelectObject(pOldBrush);

        CView::OnLButtonDown(nFlags, point);
}
```

上面画了一个带斜解交叉线的画刷，看图：




HS_BDIAGONAL ////// <br>
HS_CROSS ++++++ <br>
HS_DIAGCROSS XXXXXX <br>
HS_FDIAGONAL \\\\\\\ <br>
HS_HORIZONTAL ======= <br>
HS_VERTICAL |||||||

当然，CreatePatternBrush 需要一幅位图，其实位图操作很简单，很容易，大家先按菜单上的 插入/资源...

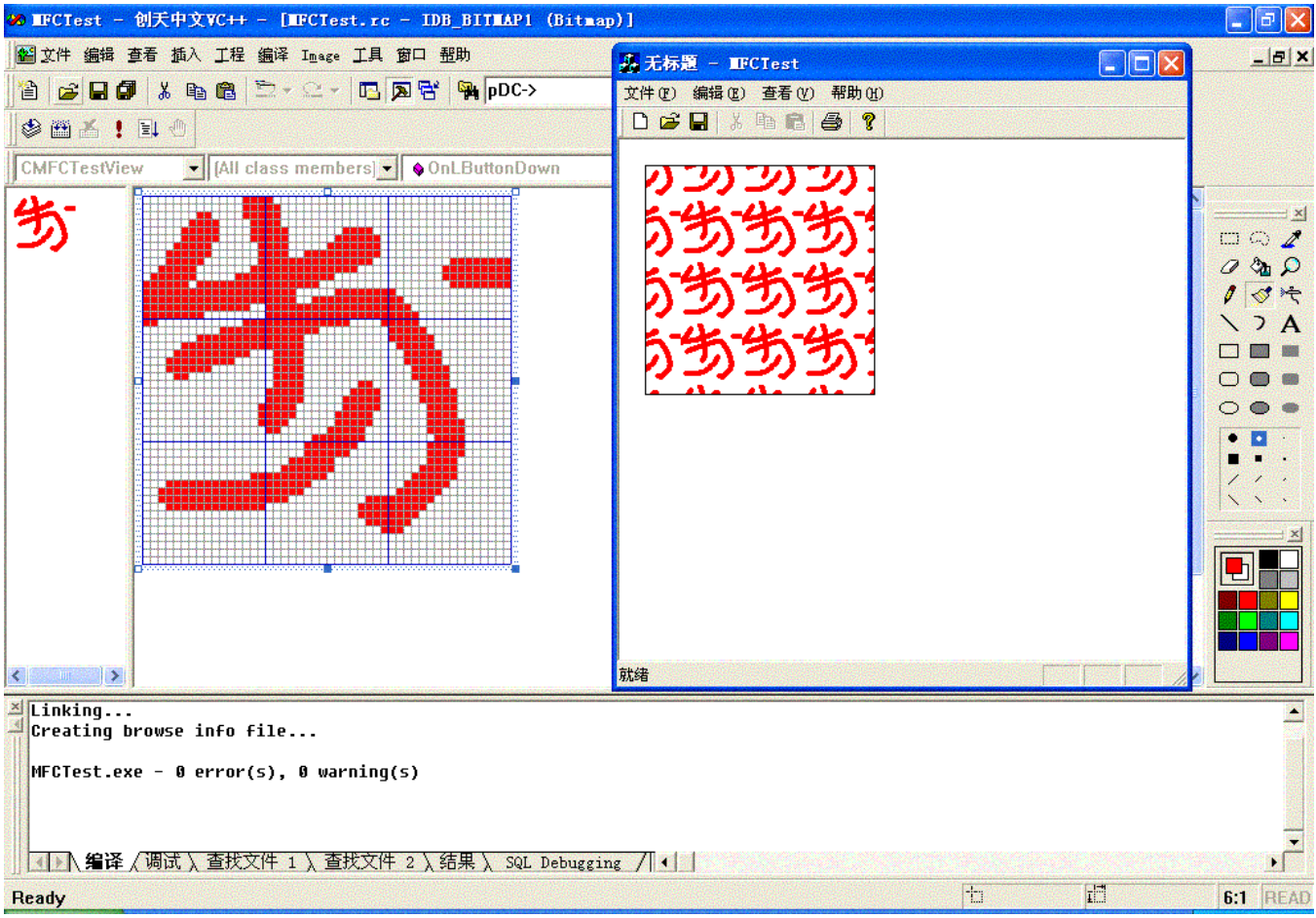
对话框中选择 Bitmap ，然后按“新建”按钮，于是，就出现一个新建按钮对话框，然后在里面画一些东东，再在程序中加入下面的代码：

……

```
CBitmap bmp;
bmp.LoadBitmap(IDB_BITMAP1);
brush.CreatePatternBrush(&bmp);
```

……

就这么简单，大家看程序运行结果：




虽然图比较寒碜，但意思总算到了，呵呵，有关位图的操作，我们以后再与大家详细地讲解。

好了，现在与大家说一个现实的问题就是，上面的一些图形，只要（注：打这个词时，竟然打出“曼”来，呵呵）是封闭的图形，就会在其中用画刷填充内部，而不闭合的图形那就用不着画刷喽。

好了，以前我们讲的那些东东已经是成了形的一些图形了，可以说，大家只要能够理解，用起来一般都没有什么太大的问题，那么我们下面给大家介绍的是另外的一套画图工具：

Poly-　系列画图工具：

也就是多个的东东，什么意思呢？例如多个的线，多个的曲线等等，什么意思呢？

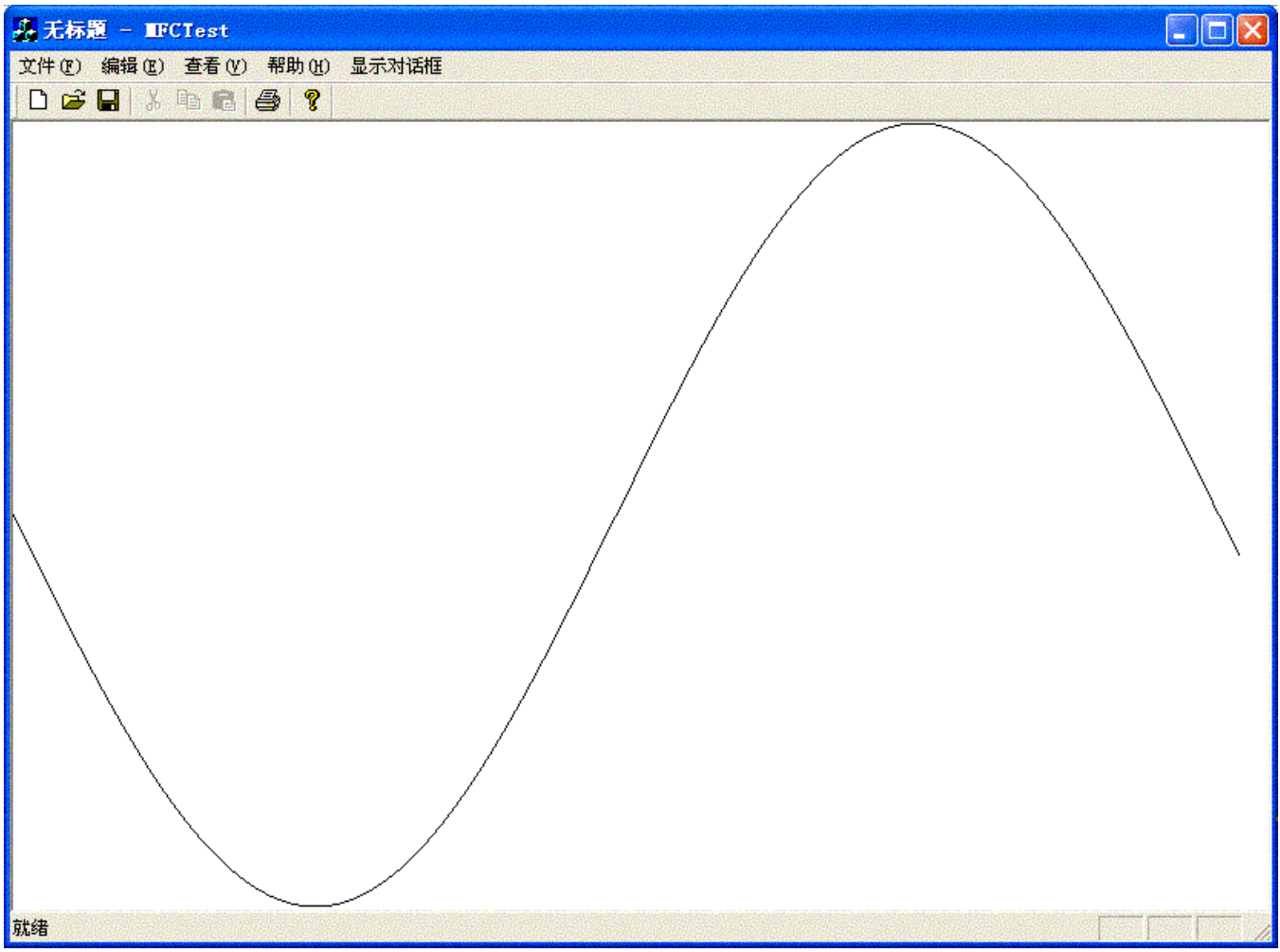
大家知道什么是正弦函数吧，画那种曲线的函数没有，那么我们有一种办法，就是用多个的直线去逼进，效果一样地很神奇。

那么既然是要画多个的线，那么就需要有多个的点，我们先来定义 800 个点吧：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CMFCTestDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        CPoint pt[800];
        int i;
        for(i=0;i<800;i++)
        {
                pt[i].x=i;
                 pt[i].y=255+int(sin(i/125.0)*255);//由于我们要用要正弦函数  sin  所以请大家包含头文件  math.h
        }
        pDC->Polyline(pt,800);
}
```

请大家用 #include 将 math.h 包含进来

图形效果挺好




感觉挺平滑吧？其实理论上来说，什么曲线都可以用以上的办法来实现，然而 Windows 系统还给我们提供了贝赛尔曲线，这个名词可能随着翻译的不同有不同的名称，但函数为：

PolyBezier

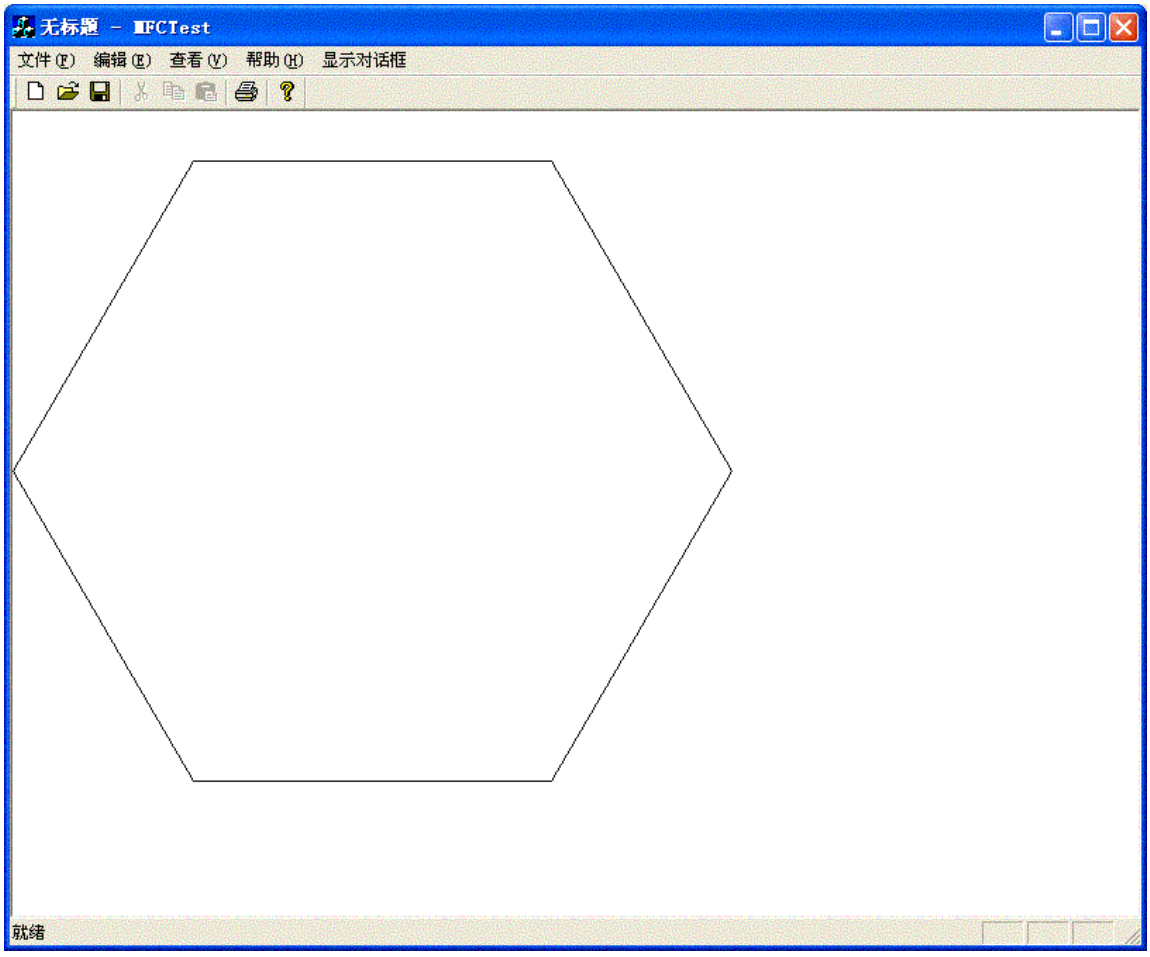
这个函数需要 3*n+4 个点，也就是说，通过四个点限定一个范围，而 Bezier 曲线总是在其中三点形成的一个小于 1 8 0°的角范围之内，所以说，这种曲线平滑性很好，之所以被计算机图形学上很普遍地使用。

举个例子吧，我们想用七个点画一个六边形（注意，一定要用七个点，因为 7=3+4），那么我们先使用画直线的来画一下看一看：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
	CMFCTestDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPoint pt[7];
```

```
        int i;
        for(i=0;i<7;i++)
        {
                pt[i].x=255+int(255*cos(i*3.141592654/3));
                pt[i].y=255+int(255*sin(i*3.141592654/3));
        }
        pDC->Polyline(pt,7);
}
```

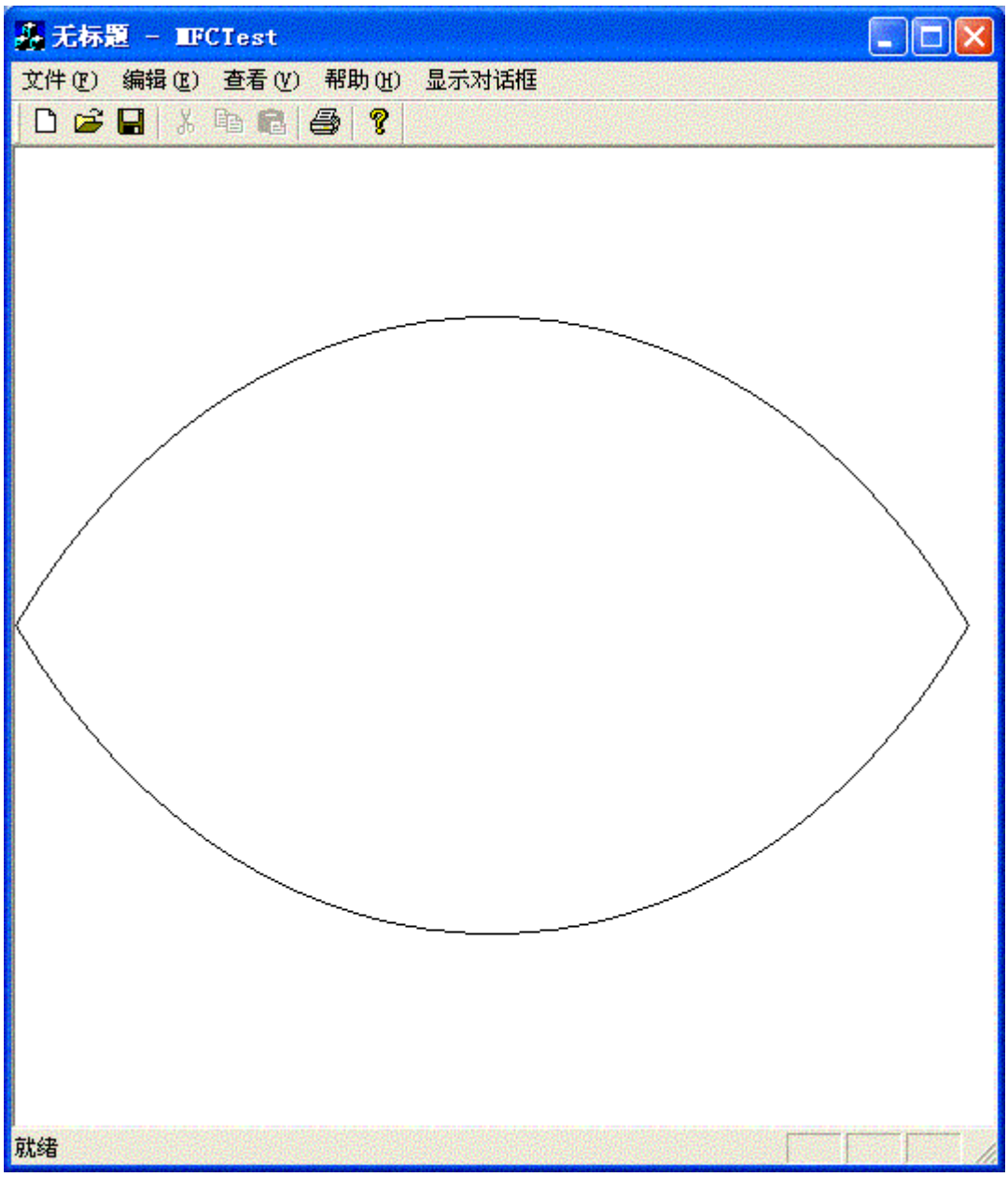
大家看图，是一个六边形是吧？




那我们同样的程序，我们仅仅将画线的部分用 Bezier 曲线来替代：

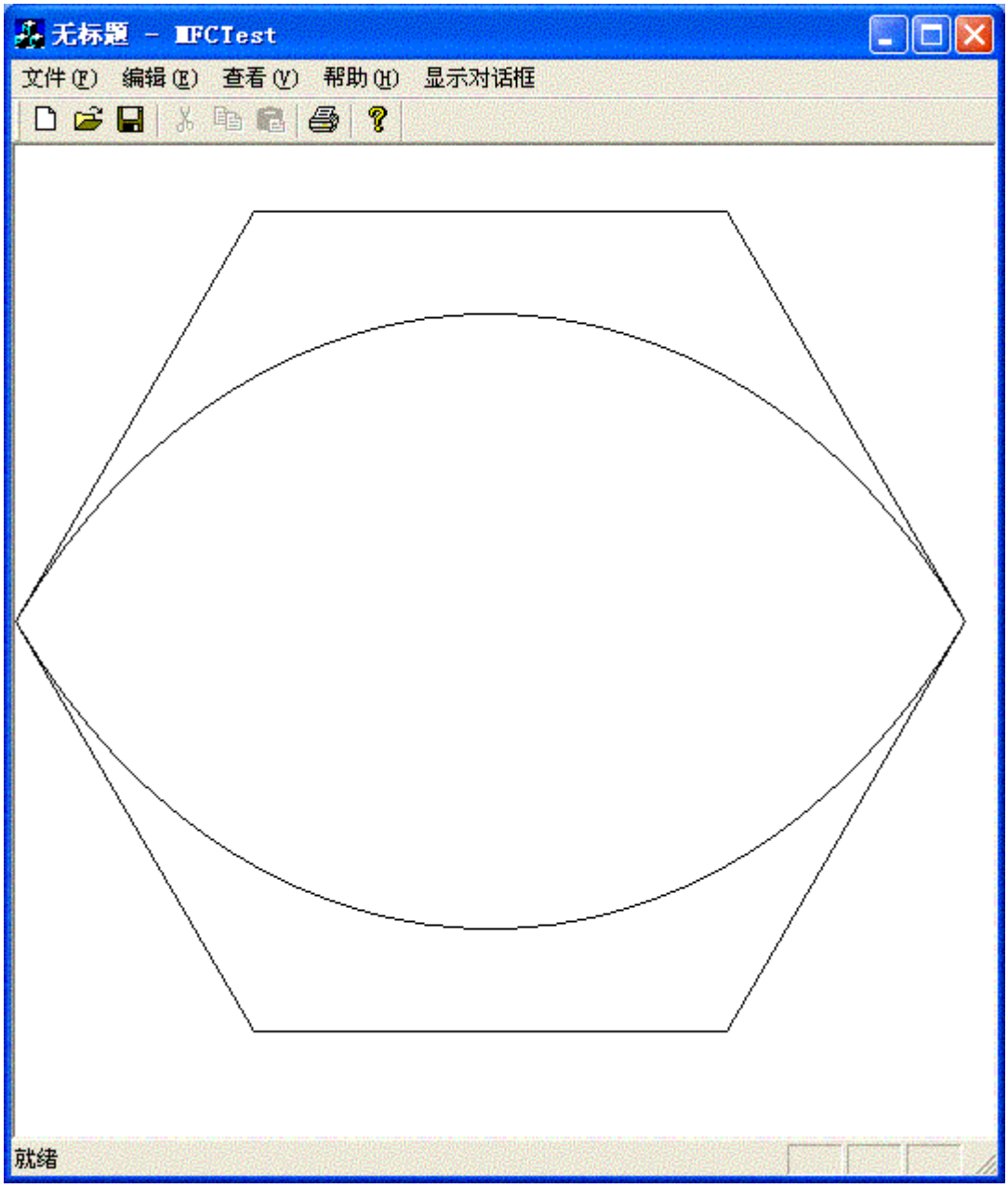
pDC->PolyBezier(pt,7);

我们想像得挺好，可是平滑倒是平滑了，可惜不是一个圆，而是：




这是怎么回事呢？原来啊，大家将上两个放一起看看：

```
pDC->PolyBezier(pt,7);
pDC->Polyline(pt,7);
```




也就是说，第一点，第四点，第七点.....，这些点是端点，而其他点控制着曲线的形状，从这个角度来看，我们就必须 3 4 5 点，6 7 8 点等这些点尽量在同一条直线上，才会显得平滑。

```
void CMFCTestView::OnDraw(CDC* pDC)
{
	CMFCTestDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CPoint pt[100];
	int i;
	for(i=0;i<100;i++)
```

```
        {
                pt[i].x=255+int(255*cos(i*3.141592654/48));
                pt[i].y=255+int(255*sin(i*3.141592654/48));
        }
        pDC->PolyBezier(pt,100);
        pDC->Polyline(pt,100);
}
```

牛刀用了 100 个点去逼进，效果已经相当地好了。

当然，大家可以申请一个有四个点对象的数组，（在 View 类中）

CPoint m_pt[4];

然后，我们在构造函数中定义四个点：

```
CMFCTestView::CMFCTestView()
{
        for(int i=0;i<4;i++)
        {
                m_pt[i].x=m_pt[i].y=0;
        }
}
```

当然，我们只定义四个为 0 的点显然不行，于是，我们需要映射一个 WM_SIZE 消息，这个消息是当我们窗口刚产生或者改变窗口大小时，会发送此消息，此时，就可以取得窗口的大小了：

```
void CMFCTestView::OnSize(UINT nType, int cx, int cy)
{
        CView::OnSize(nType, cx, cy);

        m_pt[1].x=cx/4;
        m_pt[1].y=cy/2;
        m_pt[2].x=3*cx/4;
        m_pt[2].y=cy/2;
}
```

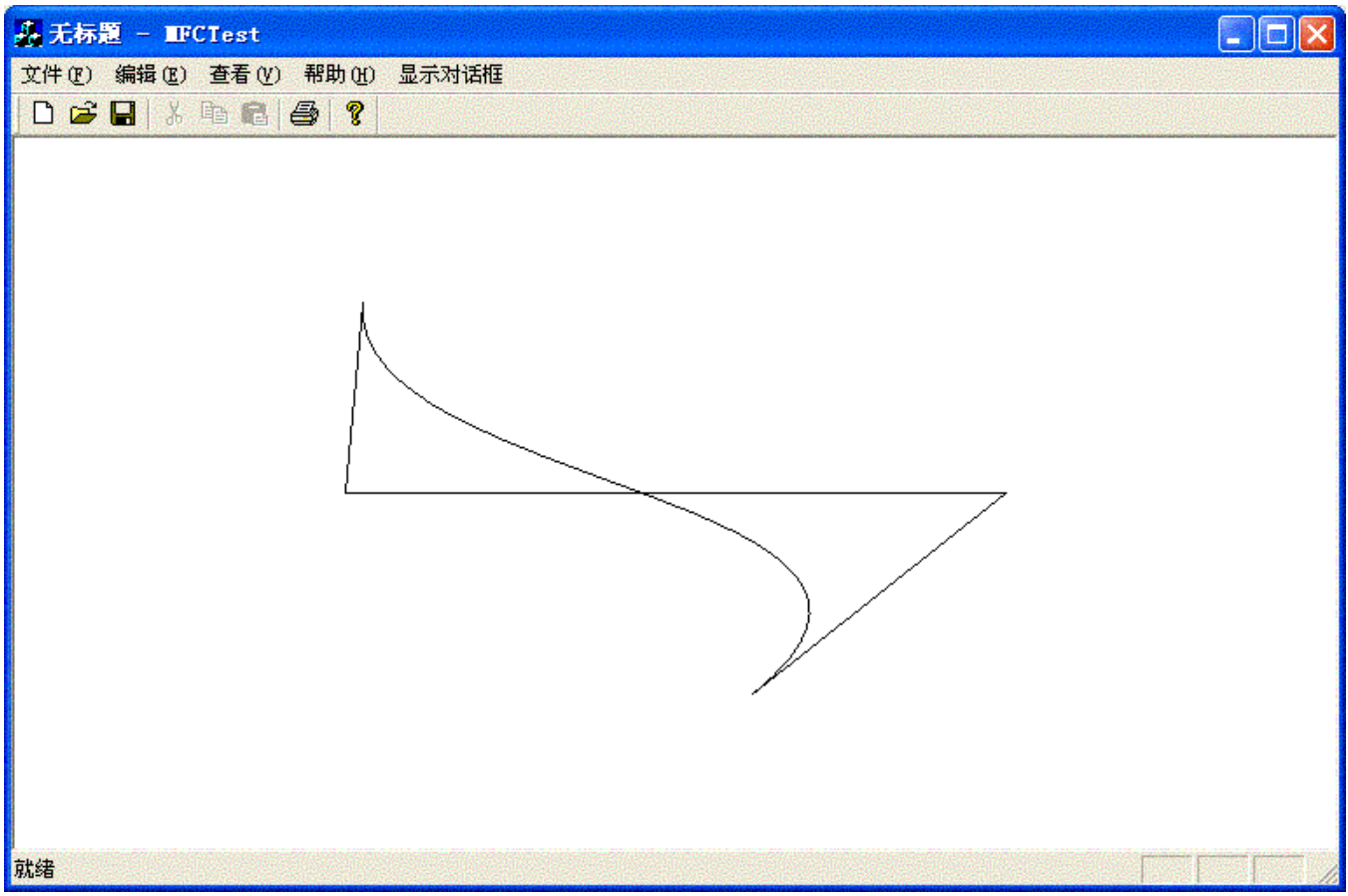
当然，我们在 OnDraw 函数中，以这四个点去画一些线：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        pDC->PolyBezier(m_pt,4);
        pDC->Polyline(m_pt,4);
}
```

大家已经可以运行程序了，当然，我们用鼠标左键与左键去改变 m_pt[0] 与 m_pt[3] ，当然喽，改变过，我们应该要重新调用 OnDraw 函数，可惜的是，如果调用，又要创建一个设备描述表，好烦，于是，有一个特殊的函数，大家看：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_pt[0]=point;
        Invalidate();
        CView::OnLButtonDown(nFlags, point);
}
void CMFCTestView::OnRButtonDown(UINT nFlags, CPoint point)
{
        m_pt[3]=point;
        Invalidate();
        CView::OnRButtonDown(nFlags, point);
}
```

其中的 Invalidate() 就是那个特殊的函数，这个函数使我们屏幕强行地换掉一张白纸，然后重新绘图，于是就逼着系统去调用 OnPaint 函数，当然喽，于是，老父就调用子类的 OnDraw 函数去重绘喽，绘制的效果挺好：




当然，大家也可以固定住第一个点与第四个点，而用鼠标去改动第三个第四个点，从而再次去体会我们的 Bezier 曲线的魅力。

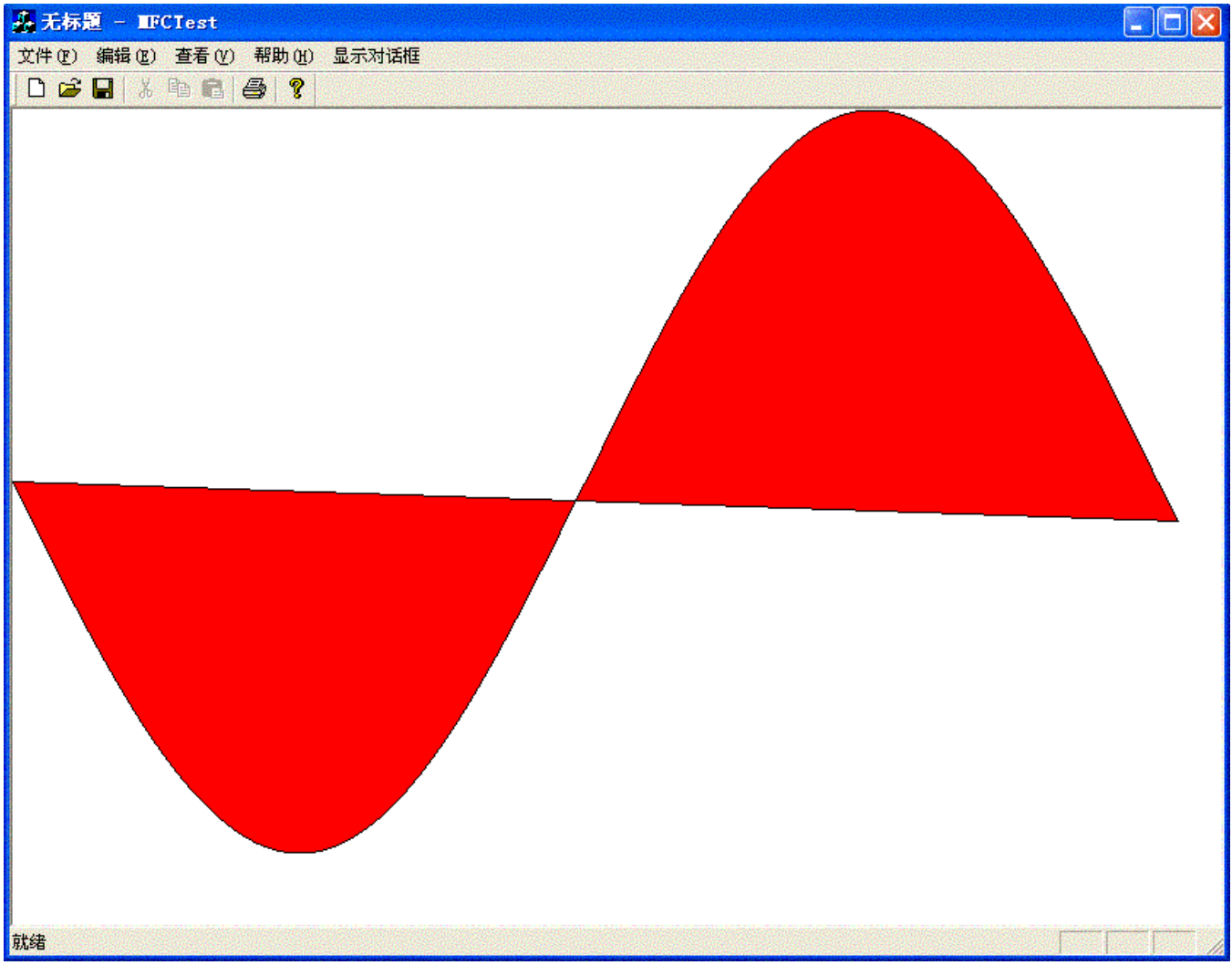
还有一种，就还是上面两种函数，但后面多了一个 To ，这种函数就会从

我们的默认点开始向后画线，并且改变默认点，也就是说，上面的 PolyBezier 和 Polyline 是不改变系统默认点的，而 PolyBezierTo 和 PolylineTo 是会改变坐标默认点的，大家可以自己去构造一些测试函数去测试一下，如果牛刀的构想上有问题，还请大家指正。

当然还有一种为 Polygon 是画封闭的线了，并且使用画刷填充，请大家看程序：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
	CMFCTestDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	CBrush brush;
	brush.CreateSolidBrush(RGB(255,0,0));
	CBrush *pOldBrush=pDC->SelectObject(&brush);

	CPoint pt[800];
	int i;
	for(i=0;i<800;i++)
	{
		pt[i].x=i;
		pt[i].y=255+int(sin(i/125.0)*255);
	}
	pDC->Polygon(pt,800);

	DeleteObject(brush);
	pDC->SelectObject(pOldBrush);
}
```

当然，画出来也是一片红，呵呵：




PolyPolyline 表示多组，每组有几个点，可以画出虚线效果，感觉挺好，大家自己去试验，当然，这些试验需要合理的计算。

最后，再说一说有关文字的显示吧，其实文字的显示很简单，就是 TextOut ，但真正要做到灵活运用就不是 TextOut 的事喽，这里给大家几组函数，每一组函数大家都上机去试试（总不会要牛刀一个一个都给大家试高吧，呵呵，那也不现实）：

SetTextAlign
SetTextColor
SetBkColor
SetBkMode

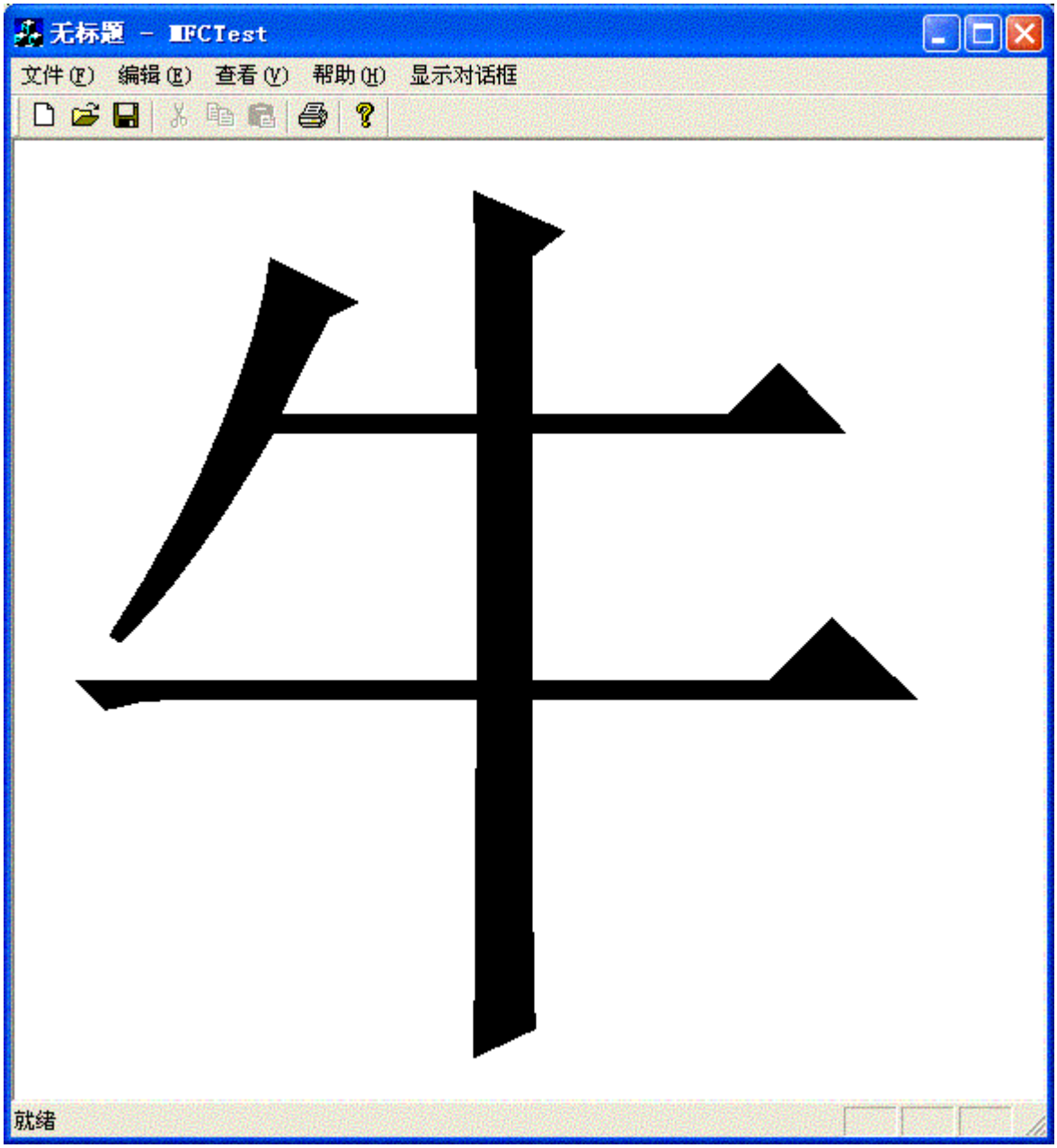
上面这些以后与大家详细讨论，这一讲我们要与大家讲一讲我们显示文字中的字体问题，因为以后需要用到。

字体也是一种资源，忆往昔，字体都是好多点。那种字库，我们称之为点阵

字库，现在基本不用了，而现在用的基本上一种字体的轮廓字库，或者有人就称之为向量字库或者是“所见即所得”。

当然，这一讲不与大家展开讲了，来一段代码，结束我们这一讲的内容吧：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
	CFont font;
	font.CreatePointFont(4000,"宋体");
	CFont *pOldFont=pDC->SelectObject(&font);
	pDC->TextOut(0,0,"牛");
	pDC->SelectObject(pOldFont);
	DeleteObject(font);
}
```



**微软积木〇八二：钟表传奇**

与大伙说了那么多，也不知道大家烦不烦，腻不腻？

好了，这一讲下来的几讲，与大伙用我们所学的知识，做几个软件吧，做软件的过程中，如果需要用到什么知识，我们再详细地与大伙讨论。

做什么软件呢？做个时钟吧（注：原来要做个分形软件的，这一讲其实做了一大半了，可惜后来发现做分形理解起来太复杂，做好后作为软件发上来给大伙开开眼界吧。）

时钟，有两种，一种是指针时钟，走针的那种，另一种是数字时钟，显示数字的那一种。

当然，显示是另外一回事，我们首先需要得到我们计算机的时间，这才是大事，呵呵。

MFC 中为我们封装了一个类：CTime ，这个类是专门用来表示时间的，好，我们就先新建一个 MFC 工程，名称定为 Time 吧，单文档哟。

当然，我们想要软件运行时，屏幕就是正方形的，就是长度与宽度相等，所以在 CMainFrame 类中的 PreCreateWindow 中，定义高度与宽度，例如，我们假设我们的长度与宽度都定为 600，

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        if( !CFrameWnd::PreCreateWindow(cs) )
                return FALSE;
        cs.cx=cs.cy=600;
        return TRUE;
}
```

上面只要加一行代码就行了，挺容易的，请大家注意，我们在此只讲一些原理性的东东，做出来的界面不见得能好到哪去，并且大家记得，我们如果在这里要画一个大圆的话，半径应该是 300。好吧，我们因为画时间的函数在多处需要用到，所以我们在 View 类中加上一个自己定义的函数 DrawTime(CDC *pDC)，请大家注意，对于这个 CDC ，我们其实还有好多东西没讲，以后说到哪儿讲到哪儿吧。

```
void CTimeView::DrawTime(CDC *pDC)
{

}
```

当然，我们这儿有设备描述表了，但我们需要一个类 CTime。

```
CTime tim;
```

请大家注意一个现实的问题，我们定义了这一个类，但却得不到现在的计算机时间，这是我们类的特性所决定的。我们的类仅仅是一种定义，当然，并不是说，不可以在构造函数中去构造当前的时间，而是因为我们这个类不仅仅有取得

当前时间的功能，我们可以任意定义一个时间，那根据我们的需要来应用。

那么，我们想取得当前时间怎么办呢？用类中的一个静态函数来实现。

什么？什么叫静态函数？真的不懂吗？

好了，我们下面就详细与大伙来说说静态函数吧，静态函数与函数中的静态变量其实理论上差不多，我们还是先建立一个控制台应用程序吧，然后再定义一个类：

```
#include <stdio.h>
class A
{
public:
        A()
        {
                printf("进入了 A 的构造函数\n");
        }
};
void fun()
{
        int i=0;
        i++;
        printf("i=%d\n",i);
}
void main()
{
        A a;
        fun();
        fun();
        fun();
        fun();
}
```

请大家注意一定不要忘记了在类的最后加上分号。

运行结果为：

```
进入了 A 的构造函数
i=1
i=1
i=1
i=1
```

当然，我们说过，如果我们在 int i; 前加上一个关键字 static ，即：static int i;

那么，这个变量在整个软件的运行过程中，只被分配一次，当编译器第二次

遇到这一定义时，就跳过忽略不定义了，并且这种定义被一直保存着，所以其中的数据也一直被保存着，所以运行结果会变为：

进入了 A 的构造函数

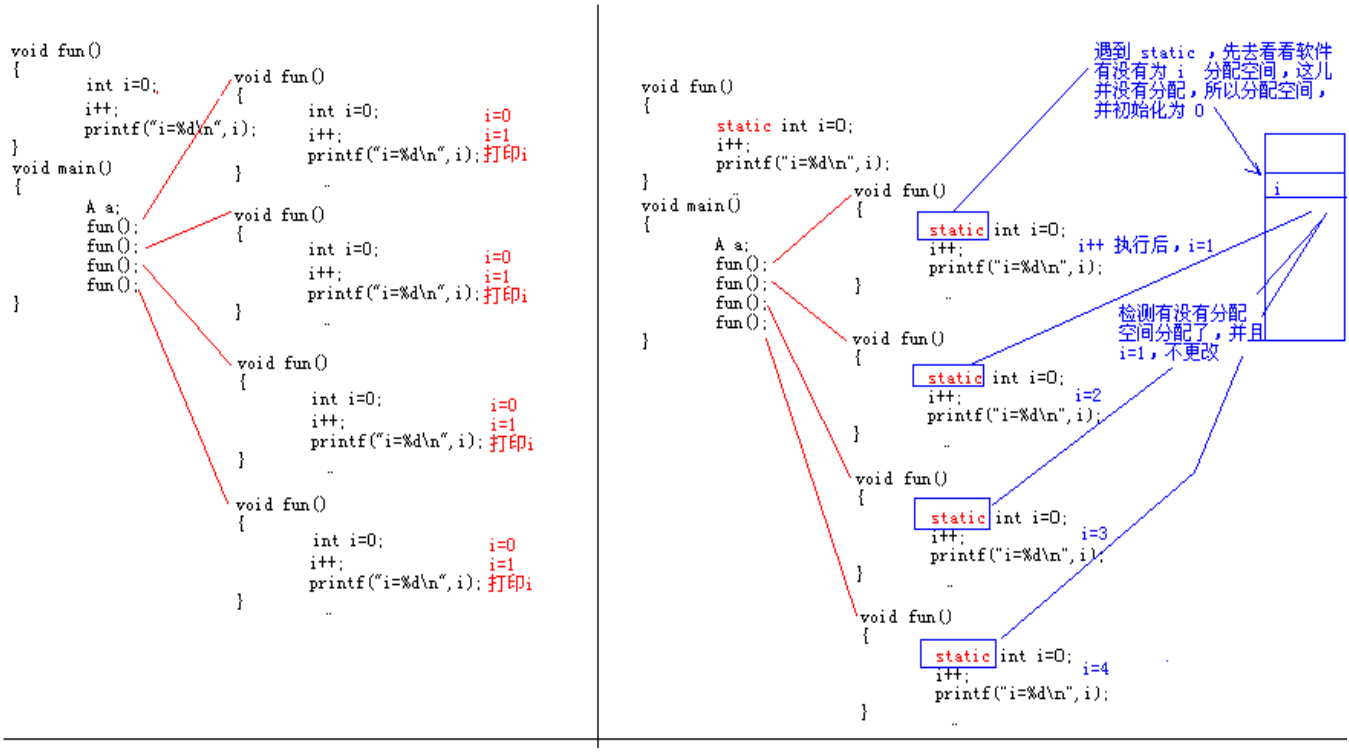
i=1

i=2

i=3

i=4

大家看下图：




这种遇到 static 就检查系统有没有为其分配空间的机制，牛刀称之为“静态化”，也就是说，理论上，这个 i 因为定义了 static 了，所以在软件运行过程中，只要第一次为其分配了空间了，那空间就一直存在在那儿，并且直到我们的程序运行结束，才销毁，但大家注意，在函数中定义的这一个数据，虽然空间一直存在在那儿，但我们毕竟不能够在函数外面去调用到这个变量，这个变量也只能在函数里面使用，从而保证了软件空间的安全性，所以说，这种变量虽然生命周期与全局变量是一样的，但却在其他地方不能被访问的。

好了，说过了函数，我们再来说说类：（我们下面就删去上面 fun 函数了，对了，大家看到没有，静态变量是一定要在定义之时赋值的，不能在定义之后赋值，如果在定义之后赋值，那就是两码事了：

```
void fun()
{
        static int i;
        i=0;
        i++;
```

```
        printf("i=%d\n",i);
}
```

如果是那样的话，那么，我们对其赋值就会使得静态的特性形同虚设了。

好了，我们也在类中申请一个静态变量：

```
#include <stdio.h>
class A
{
        static int i;
public:
        A()
        {
                printf("i=%d\n",i);
        }
};
int A::i=0;
void main()
{
        A a;
}
```

请大家注意，我们在定义这种静态变量时，大家也注意到，我们是在类的外面为其赋值的，为什么要这样做，而不能像上面那样，直接在类中使用：

static int i=0;

呢？原来啊，大家应该知道，类是有权限的，也就是说，在类中的操作必须是对类的成员进行，而这种静态的变量是类的成员吗？

不错，是类的成员，那么，我们假如使用下面的类：

static int i=0;

那么我们如果定义了两个变量：

A a,b;

请问大家一个很简单的问题，这一个 i 是属于 a ，还是属于 b 呢？

请大家注意了，我们对函数编译时，遇到静态变量没关系，编译过去就行了，因为那其中的 i 不管是不是静态的，只是一个 i ，如果是静态的，那么在运行时，就为其分配一个全局的空间，如果不是静态的，那么就不存在这个问题。

但是 A a,b; 这样的代码就不同了，函数是在运行时定义那个变量，而这两个类中的变量就必须要在编译时就确定下来。

你说编译器编译是一个翻译的过程，她怎么知道你将会在 a 中对 i 怎么处理，在 b 中对 i 怎么处理呢？

还是那句话，编译器在编译时，不能够为我们分配空间，所以这种空间的分配就不能够在类中进行，原因很简单，函数仅仅是一个函数，而类中的成员变量是每个对象中都有，类中的成员函数是每个类中都有啊，是不一样的。

也正因为如此，人们只有强行地规定：类中的静态变量不属于任何对象，只属于类，也正因为如果，我们在定义类的时候，不能够对静态变量进行空间分配，而这个事性只有拿到类外面来说，于是，就出现了以上的那种情况。

当然，只要在类外对其进行了定义了，空间也就分配了，空间一分配了，在类中就可以对这个变量当着成员变量一样地操作。

正因如此，我们类中的静态变量只有在类外进行定义，并且定义出的类的静态变量属于所有的对象所共有。

也正是如此，我们在类还没有定义成对象的时候，我们类中的静态变量就已经产生，而且不受任何一个变量的独享。

好了，类的静态变量讨论完了，下面还要讨论另一个重要的问题，类的静态函数：

静态函数也好理解啊，只是将函数前面加上一个 static 就一切 OK 了呗，好，我们来看：

```
#include <stdio.h>
class A
{
public:
        static int i;
        int x,y;
public:
        A()
        {
                printf("进入了类\n");
        }
        static void print();
};
int A::i=0;
void A::print()
{
        printf("进入了类的静态函数\n");
}
void main()
{
        A a;
        a.print();
```

```
}
```

大家看这个静态函数好理解吧，在对象中也可以调用是吧？更怪怪的就是，这种静态函数与静态对象也一样，不属于任何一个类，所以说，我们在没有定义成对象时，就可以调用这个函数，原因很简单，这个函数也不属于任何一个对象，而是一个静态的函数，在没有定义对象的时候，就存在，也就是说，我们可以在没有定义类时，就这样调用：

```
void main()
{
        A::print();
}
```

请大家注意，因为这个函数不属于任何一个对象，所以这个对象是全局的，如果是 public 类型的，在任何类的外面都可以通过类名去访问。当然，在类中的函数中访问，就不需要通过类名也可以去访问喽。

对了，我们再对 print 函数来改一改：

```
void A::print()
{
        printf("x=%d,y=%d\n",x,y);
}
```

我们想像得很好，想通过这个函数来访问类中的 x 与 y 值，可惜我们做不到，为什么呢？大家一定注意，这个函数是全局的函数，不属于任何一个对象，而属于这个类，也就是说，这个函数定义之时，还没有任何一个对象产生呢？而 x,y 都是属于对象的，也就是不是全局的，你还没有对象就想要人家对象中的成员变量，你那不是跟姑子要孩子么？所以说，对于类中的静态函数，有明文规定：

·不允许向姑子要孩子

噢，不，打错了，应该是：

·静态的函数不允许使用类中的成员变量，除非……除非……

除非这个变量也就静态变量，也就是这个变量也是对象没有产生之时就有了，才能够使用，于是，下面的程序就可以编译通过：

```
void A::print()
{
        printf("i=%d\n",i);
}
```

对于这种可以多个类共享的变量，或者是多个类共享的函数，我们通常都可以将其做成静的。

CTime 中就一个函数，GetCurrentTime() 这个函数就是取得当前的时间，就是静态的，我们当然可以顺利地使用这个变量，去取得当前的时间喽，当然，返回的值就是一个 CTime 类型，所以我们可以使用下面的代码：

```
tim=CTime::GetCurrentTime();
```

呵呵，想将一个理论讲明白真得下一番功夫，是不？当然，如果牛刀能够将

大家讲明白了，牛刀的目的也就达到了，好了，对于 CTime 来说，还有一个很好玩的函数：Format，返回的可是一个字符串哟，非常好用：

```
CString str=tim.Format("现在的时间为%H 时%M 分%S 秒");
pDC->TextOut(0,0,str);
```

大家看是不是还可以理解？那 %H%M%S 是不是有点像我们 pirntf 中的转换字符啊，呵呵，我们看几种格式：

%D 天
%H 时
%M 分
%S 秒
%% 百分号

就是这样，呵呵，知道就行。

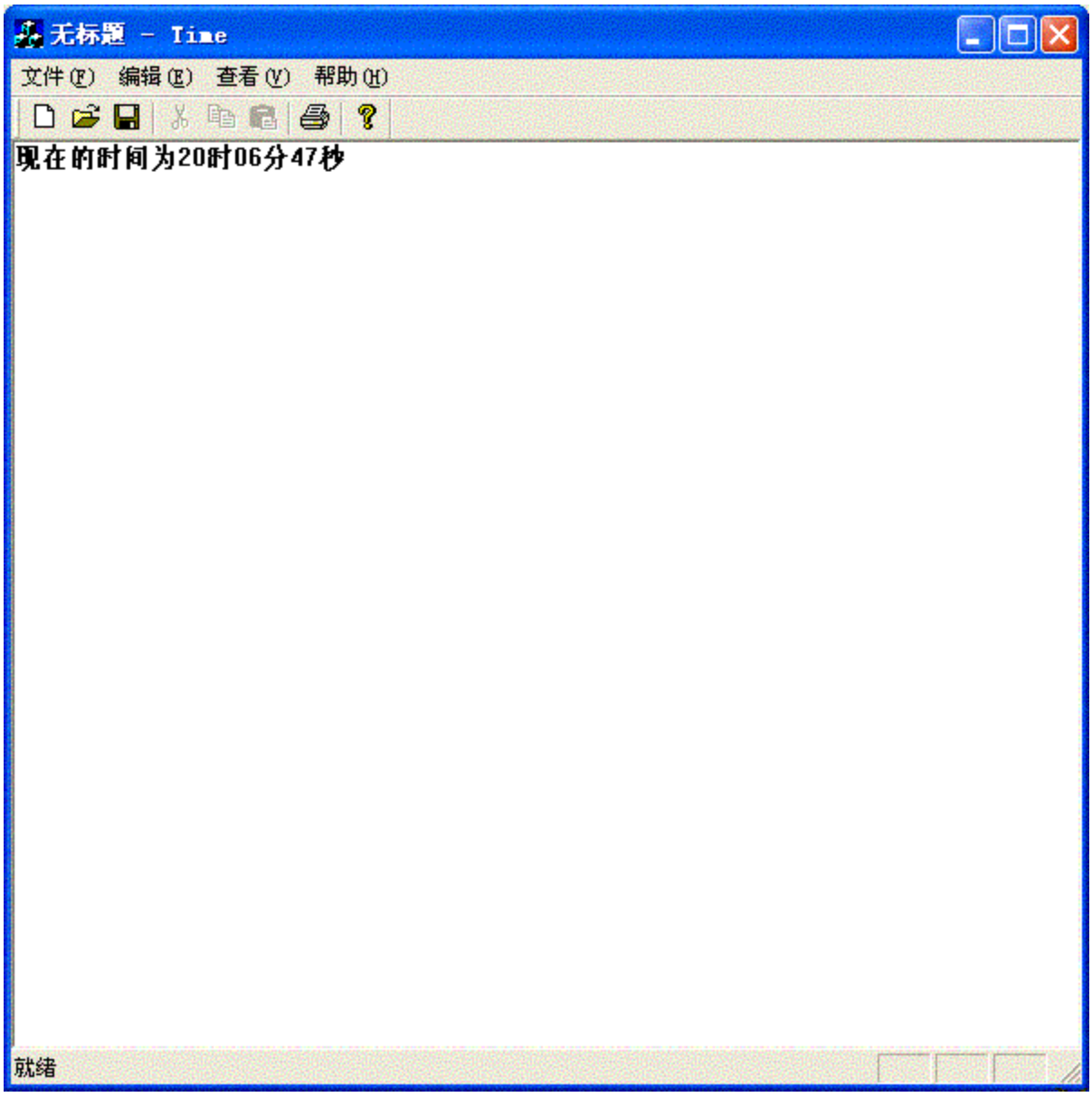
那么，接下来，我们就需要将其用 TextOut 显示出来，当然喽，做出来的程序就是很简单喽：

```
void CTimeView::DrawTime(CDC *pDC)
{
        CTime tim;
        tim=CTime::GetCurrentTime();
        CString str=tim.Format("现在的时间为%H 时%M 分%S 秒");
        pDC->TextOut(0,0,str);
}
```

当然，目前还不能运行出你想要的结果，为什么？因为我们还需要在 OnDraw 函数中去调用一下，才会运行 OK 喽：

```
void CTimeView::OnDraw(CDC* pDC)
{
        DrawTime(pDC);
}
```

好了，显示出来已经非常棒喽：




是不是感觉还可以，当然，我们的时钟还不能够算是完美，什么原因呢？一个缺憾就是：我们的时钟不能够真正地显示时间，因为画面是静止的，那怎么办呢？大家一定会想到我们的时钟消息 WM_TIME 吧？

那好，我们就响应两个消息，一个 WM_CREATE 用来创建时钟，一个是 WM_TIMER 消息，用来显示时钟：

```
int CTimeView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;
        SetTimer(0,200,NULL);
        return 0;
}
```

```
void CTimeView::OnTimer(UINT nIDEvent)
{
        CClientDC dc(this);
        OnDraw(&dc);
        CView::OnTimer(nIDEvent);
}
```

这次运行感觉就非常地好喽。

好是好，还是不太满意是吧？为什么呢？因为我们这种时钟也是太粗陋了，不过这种编程的技术可是实实在在的啊，呵呵，我们这一讲讲的一些理论可是真的哟。

当然，这个时钟，我们以后还不断地会去美化她，我们这一讲呢，再来与大伙回顾一下本讲的知识点：

·类中的静态变量，必须在类外面进行定义，这个变量属于这一个类，不属于任何一个对象。
·类中的静态函数，不属于任何一个类，所以在此函数中，不允许使用类中的对象，可以使用类中定义的静态变量，对于静态函数，可以用对象去调用，也可以在类外直接用类名去调用。
·类中的静态函数，可以是类的共性的，共有的函数不，类的静态变量，一般保存类中共同的数据，地合适的时机选用。

举个简单的例子吧，我们可以将企业的员工信息或者学生信息定义为一个类，而员工或学生的个数是随着类的产生多少或者有无员工与学生的转职或转学所决定的，所以一个企业或者学校的员工数目或学生数目就可以申请为一个静态变量，如果一个对象构造，只需要对这个静态变量加一，如果一个对象析构，只需要对这个静态对象减一就一切 OK 了。

静态变量与函数可以为我们的计算机节省好多的资源。

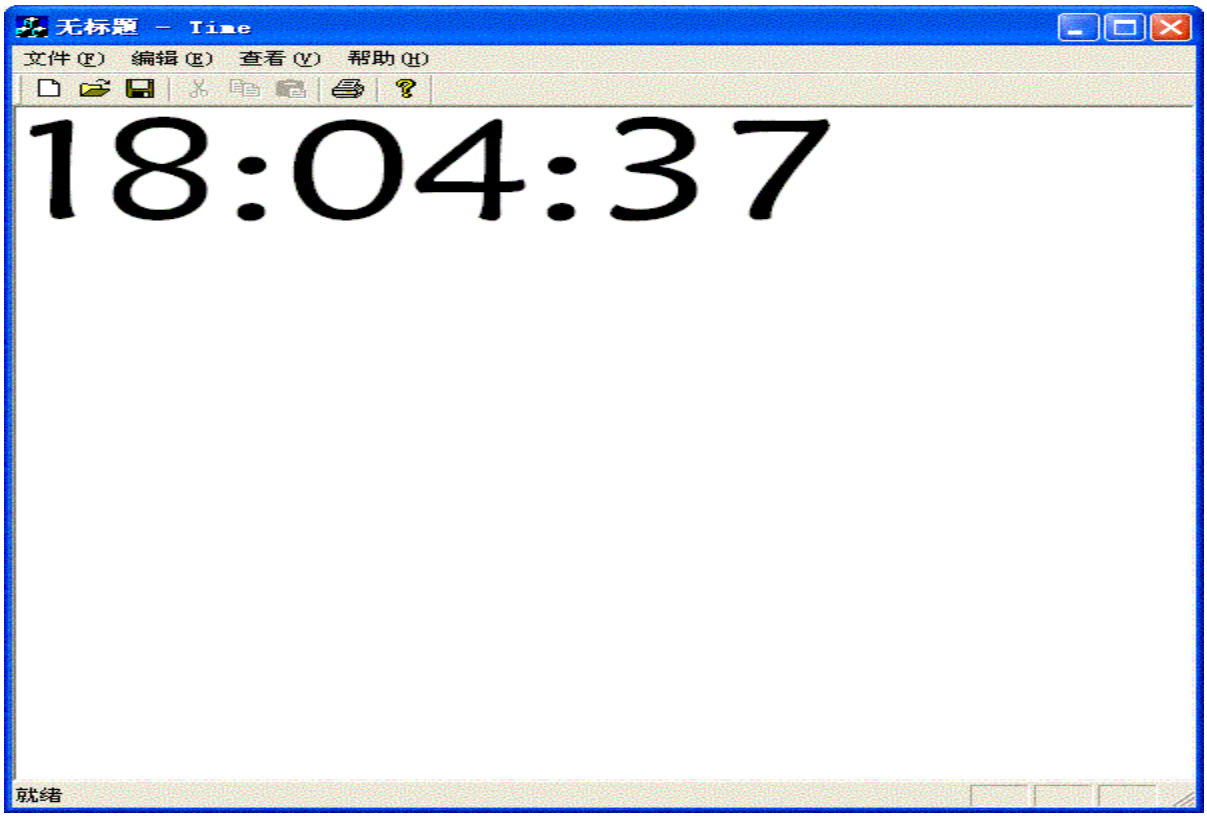
**微软积木〇八三：时间显示**

还是上一讲，我们调用的那个函数：

```
void CTimeView::DrawTime(CDC *pDC)
{
CTime tim;
tim=CTime::GetCurrentTime();
CString str=tim.Format("现在的时间为%H 时%M 分%S 秒");
pDC->TextOut(0,0,str);
}
```

当然，我们为了显示得舒服，我们可以加入一些例如字体，当然，我们将 Format 也给变了：

```
void CTimeView::DrawTime(CDC *pDC)
{
CTime tim;
tim=CTime::GetCurrentTime();
CFont font;
font.CreatePointFont(800,"华文新魏");
CFont *pOldFont=pDC->SelectObject(&font);
CString str=tim.Format("%H:%M:%S");
pDC->TextOut(0,0,str);
pDC->SelectObject(&pOldFont);
DeleteObject(&font);
}
```



当然，字体与画刷、画笔一样，是一种选入后需要释放的资源，或者说，字体也是一种句柄，只不过被 CFont 给包装起来喽，其实我们对字体的操作很复杂，但这个字体的类却给了我们一个很简单的函数： CreatePointFont ，可以创建一个字体，第一个是大小，理论上可以任意，但第二个参数设置时，请大家注意，你的字库中有没有这种字体，如果有就可以使用，使用后，计算机系统会自动地为我们的程序去挂接新字体，并且使得新的字体符合我们的需要。

用过字体后，也需要释放与删除，一切都与画笔画刷一样，我们可以选入，可以使用。

当然喽，我们以后再用到再详细地与大伙讲，我们可以选择一款合适的字体去做我们的时钟。

好，我们既然可以用字符去显示时间，那已经是没有问题的喽，那么我们以后的讨论中，就重点与大伙讨论指针时钟的设置，当然，可能会用到一些别人的图片，但不是出于商业目的，相信别人都会容忍的吧，呵呵，希望别人不会来找牛刀的麻烦，呵呵。

行了，那么我们怎么办呢？我们准备与大家分成两部分去实现，一个部分是画时钟的背景，我们再定义一个函数：DrawFace 再一个定义时钟的指针，我们再定义一个函数：DrawHand，好喽，看程序吧，呵呵。

```
void CTimeView::DrawTime(CDC *pDC)
{
DrawFace(pDC);
DrawHand(pDC);
}
```

我们在 DrawFace 中，我们画时钟的背景，时钟点数啊，什么的，呵呵，而在 DrawHand 中，我们去画时钟的指针。

当然是画时钟背景比较简单的喽，不就是画二十个点嘛，当然，为了画这十二个点，与以下的应用，我们需要包含一个 math.h 文件，并且定义一个宏 PI

```
#include <math.h>
#define PI 3.141592654
```

请大家注意，这两句后面都不要分号，因为不是语句。

好了，相信大家都感觉得到应该怎么画，人不过是从圆心向外，每隔 5°画一个点嘛，说是这么说，但这里请大家注意喽，我们需要知道圆心的位置，我们还需要知道画点离圆心的长度，所以我们要在类中定义这些数据：

```
CPoint m_ptOrigin;
int m_long;
int m_hourlong;
int m_secondlong;
int m_minutelong;
```

怎么样？在构造函数中对这些数据进行初始化吧，一定要在构造函数中哟，呵呵：

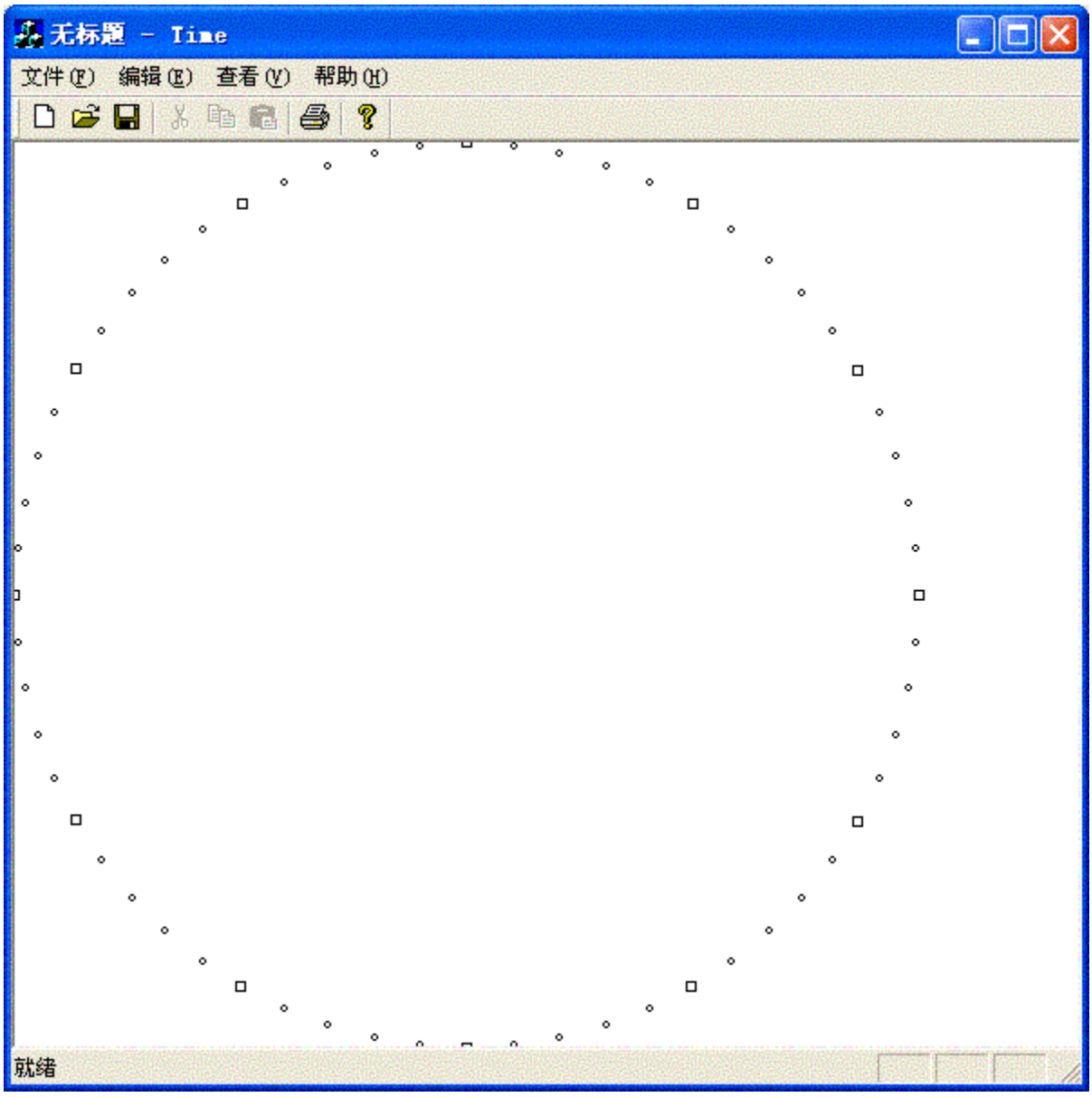
```
CTimeView::CTimeView()
{
m_ptOrigin=CPoint(250,250);
m_long=250;
```

```
m_hourlong=150;
m_secondlong=240;
m_minutelong=200;
}
```

当然，你可以改变这些值，当然，做好后你自己再去改改看什么效果，我们先与大家做函数吧：

```
void CTimeView::DrawFace(CDC *pDC)
{
double ct=0.0;
double dct=2.0*PI/60.0;
CRect rect;
CPoint pt;
int i;
for(i=0;i<60;i++)
{
  pt=m_ptOrigin+CPoint(int(m_long*cos(ct)),int(m_long*sin(ct)));
  rect=CRect(pt.x-2,pt.y-2,pt.x+2,pt.y+2);
  pDC->Ellipse(rect);
  ct+=dct;
}
ct=0.0;
dct=2.0*PI/12;
for(i=0;i<12;i++)
{
  pt=m_ptOrigin+CPoint(int(m_long*cos(ct)),int(m_long*sin(ct)));
  rect=CRect(pt.x-3,pt.y-3,pt.x+3,pt.y+3);
  pDC->Rectangle(rect);
  ct+=dct;
}
}
```

当然，这些数据需要大家自己去调节调节，因为我们做出来的显示区域并不是刚好 600X600，因为我们还有工具条与状态条要占去好多的空间，所以我们要注意这些并不是如我们想像的地方，呵呵，做软件就是这样，不可能一步到位，都按着我们规划好的来的，如果是那样，那做程序就真的会很简单了。




背景界面是画好了，下面就来画指针吧：

```
CTime tim;
tim=CTime::GetCurrentTime();
```

我们得到时间很容易，我们上来讲也告诉大家如何通过时间的 Format 去格式化一个字符了，那么这一讲我们要得到什么呢？当然，先得到当前的“秒”数，那当然好办，因为 CTime 类就有相应的函数，给我们返回这个整形值：

```
int sec=tim.GetSecond();
int min=tim.GetMinute();
int hou=tim.GetHour();
```

简单吧，的确很简单，如果不是很简单，那么我们的类的概念也就白提了，是也不是？

下面我们就要根据这三个数据来画线，当然是先画秒针：（下面的程序，我们不加注释，大家看什么意思，呵呵：）

```
void CTimeView::DrawHand(CDC *pDC)
{
CTime tim;
tim=CTime::GetCurrentTime();
int sec=tim.GetSecond();
int min=tim.GetMinute();
int hou=tim.GetHour();
double ct=-PI/2.0+sec*2.0*PI/60.0;
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_secondlong*cos(ct)),int(m_secondl
ong*sin(ct))));
ct=-PI/2.0+min*2.0*PI/60.0+ct/60.0;
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_minutelong*cos(ct)),int(m_minutel
ong*sin(ct))));
ct=-PI/2.0+hou*2.0*PI/12.0+ct/60.0;
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_hourlong*cos(ct)),int(m_hourlong*
sin(ct))));
}
```

如果大家能够理解上面的程序的话，那我们的钟面程序就已经能够很精确地显示时间了可惜的是，我们每显示一次，就会多出一个指针，还有一个就是，如果我们的指针颜色能够变一变，那该多好？于是，出于以上的目的，我们先在 DrawFace 中，重新画一次我们的整个图片：

```
GetClientRect(rect);
pDC->Rectangle(rect);
```

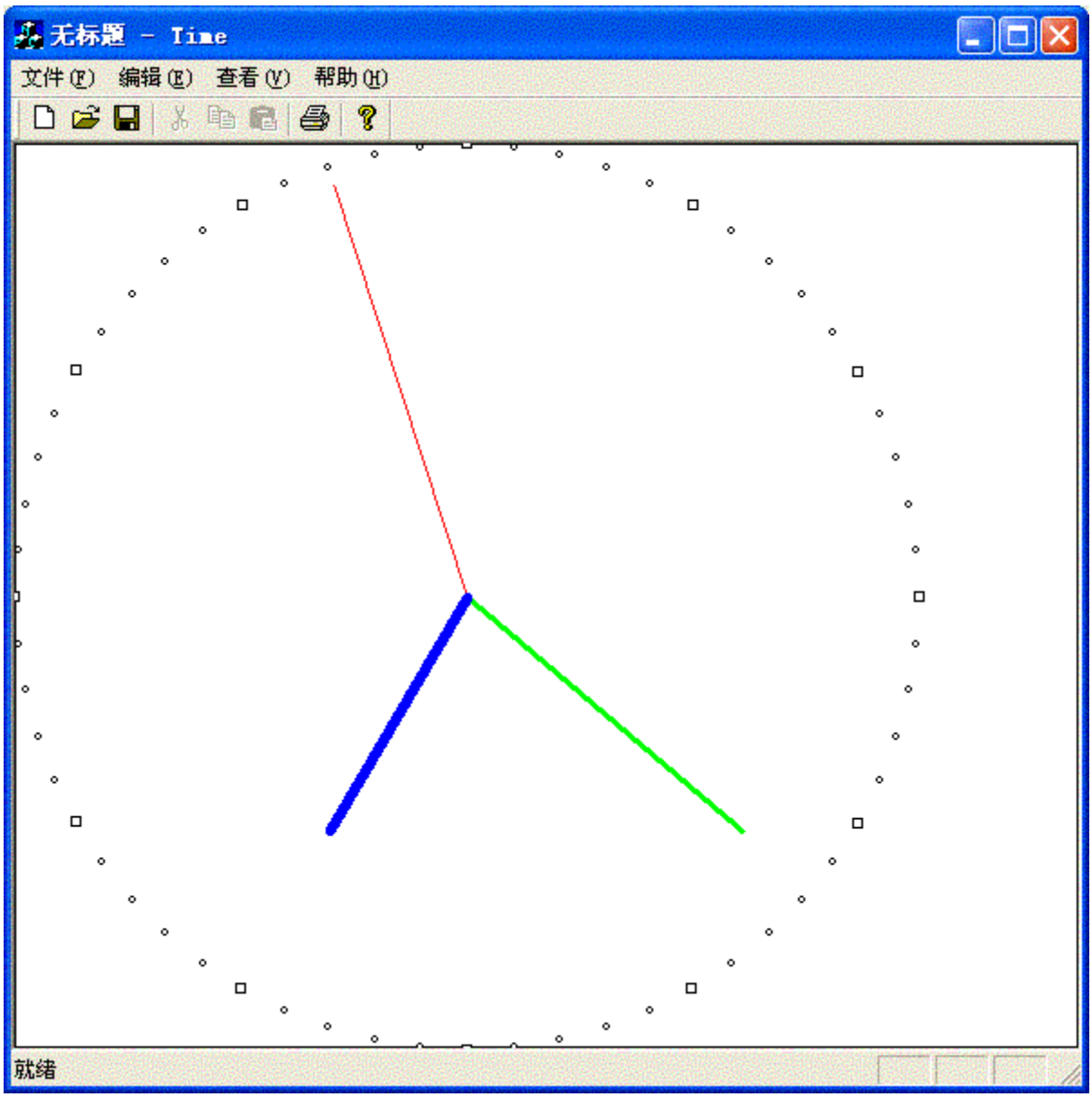
有了以上两句，我们的时钟显示就会十分地好，当然喽，从显示角度来说而已。

好了，我们再将指针的颜色用画笔去重绘一下：

```
.....
CPen pen(0,1,RGB(255,0,0));
CPen *pOldPen=pDC->SelectObject(&pen);
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_secondlong*cos(ct)),int(m_secondl
ong*sin(ct))));
pen.DeleteObject();//注意，我们要再次创建就要先删除
pen.CreatePen(0,3,RGB(0,255,0));
pDC->SelectObject(&pen);
ct=-PI/2.0+min*2.0*PI/60.0+ct/60.0;
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_minutelong*cos(ct)),int(m_minutel
ong*sin(ct))));
pen.DeleteObject();
pen.CreatePen(0,6,RGB(0,0,255));
```

```
pDC->SelectObject(&pen);
ct=-PI/2.0+hou*2.0*PI/12.0+ct/60.0;
pDC->MoveTo(m_ptOrigin);
pDC->LineTo(m_ptOrigin+CPoint(int(m_hourlong*cos(ct)),int(m_hourlong*
sin(ct))));
DeleteObject(pen);
pDC->SelectObject(pOldPen);
}
```

这样一来，我们的表面就有那么点感觉了，虽然还是不怎么漂亮，但总比那单调的表面要好多了。




很可惜的是，这个画面有点跳，这种现象是由于我们在 DrawFace 函数中用白色画刷对整个客户区画方块引起的，也就是说，我们在窗口中画图的速度还不够理想，速度还跟不上，我们的肉眼还感觉得出来画图的速度，没办法，我

们的计算机处理的速度不理想。

也正因为如此，人们为了使得画图速度变快，好多的画图的部分就放到显卡中去完成了，从而对外提供一批操纵显卡的 API 函数，使得我们画图时，屏幕与内存的通讯量降低，从而增加了画图的速度，国内国外好多的游戏就是用的这种技术。

什么意思呢？原来啊，我们的显卡也是一台电脑，本身就是，里面甚至有好多的贴图库，矢量库，还有好多的算法，严然处理的能够比我们的计算机处理的数据要多多啦。

那么那个为什么就快呢？原因就是，我们显示器上显示的内容其实就是显卡上的内容，在 DOS 时代，这种显示的内容也是放在内存中的，原因很简单，那时候我们的电脑只显示 26 个字母与一些特殊的数字，仅此而已，所以完全可以放在内存中处理得了，而现代的计算机不要说是 1024*768 的吧，就是 800*600=480000，如果再表示不同的颜色，你如果用内存，那么需要多大的内存？再说有这么大的内存吗？

于是，这部分工作就移到了显卡上，而内存中的程序就与显卡在拍电报：

显卡显卡，我是内存，我是内存，我请你在 200,200 点位置打一个点

显卡明白，显卡明白

显卡显卡，我是内存，我是内存，我请你在 201,201 点位置打一个点

显卡明白，显卡明白

.....

显卡显卡，我是内存，我是内存，我请你在 500,500 点位置打一个点

显卡明白，显卡明白

于是，我们的电脑上就出现了一个 200,200 到 500,500 位置画的一条线，下面我们就用这种办法来画一条线：（当然喽，这次我们不在那个时钟上做了，大家新建一个 MFC 工程，或者在其他 MFC 工程中去做）

```
void CMFCTestView::OnDraw(CDC* pDC)
{
for(int i=200;i<500;i++)
{
  pDC->SetPixel(i,i,RGB(0,0,0));
}
}
```

果然是需要与显示拍几百次，甚至更多次的电报的，所以这个速度很慢，当然喽，像上面这种二三百次对于我们的电脑来看简直就是小菜一碟，我们再来一次，这次多拍一点：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
for(int i=20;i<500;i++)
{
  for(int j=20;j<600;j++)
  {
   pDC->SetPixel(i,j,RGB(0,0,0));
  }
}
```

}

大家运行一下程序，是不是看见我们窗口上的黑方块是怎么画出来的？是吧？也就是说，上面画了几万次，我们的电脑就已经初步显示出其“慢”来了。

所以为了减少拍电报的次数，显卡与内存之间就相互之间定义了一种协定：

显卡显卡，我是内存，我是内存，显卡听到请回答

显卡听到，显卡听到，请讲话

请显卡将当前坐标移到 200,200

显卡明白，显卡明白

请显卡从当前坐标到 500,500　画一条直线

显卡明白，显卡明白

于是，显卡从 200,200 到 500,500 画了一条直线。

pDC->MoveTo(200,200);

pDC->LineTo(500,500);

显卡其实就是这样实在，显卡的功能直接影响到内存需要画东西的速度。

人们为什么要做种种的图形图像的　API　函数，就是要减少显卡与内存的通信数据量，从而提高屏幕绘制的速度，于是，就形成了许多的专业的图形函数，需要相应的显卡的支持，如　OpenGL 或者是 DirectX　等等，牛刀的电脑显卡就不支持后一种，从而好多游戏都玩不成，很可惜，哎～～～。

当然，像图片这样的东西，其实我们与显卡之间也有两种通信方式：

因为我们需要将图片上所有的信息都通知显卡，所以我们需要将图片中的所有点都告诉显卡，就有两种办法，一种就是：

一个点一个点地向显卡那儿报：

显卡显卡，第 0,0 点位置的 RGB 值为 0,0,0

显卡明白，显卡明白

显卡显卡，第 0,1 点位置的 RGB 值为 231,23,432

显卡明白，显卡明白

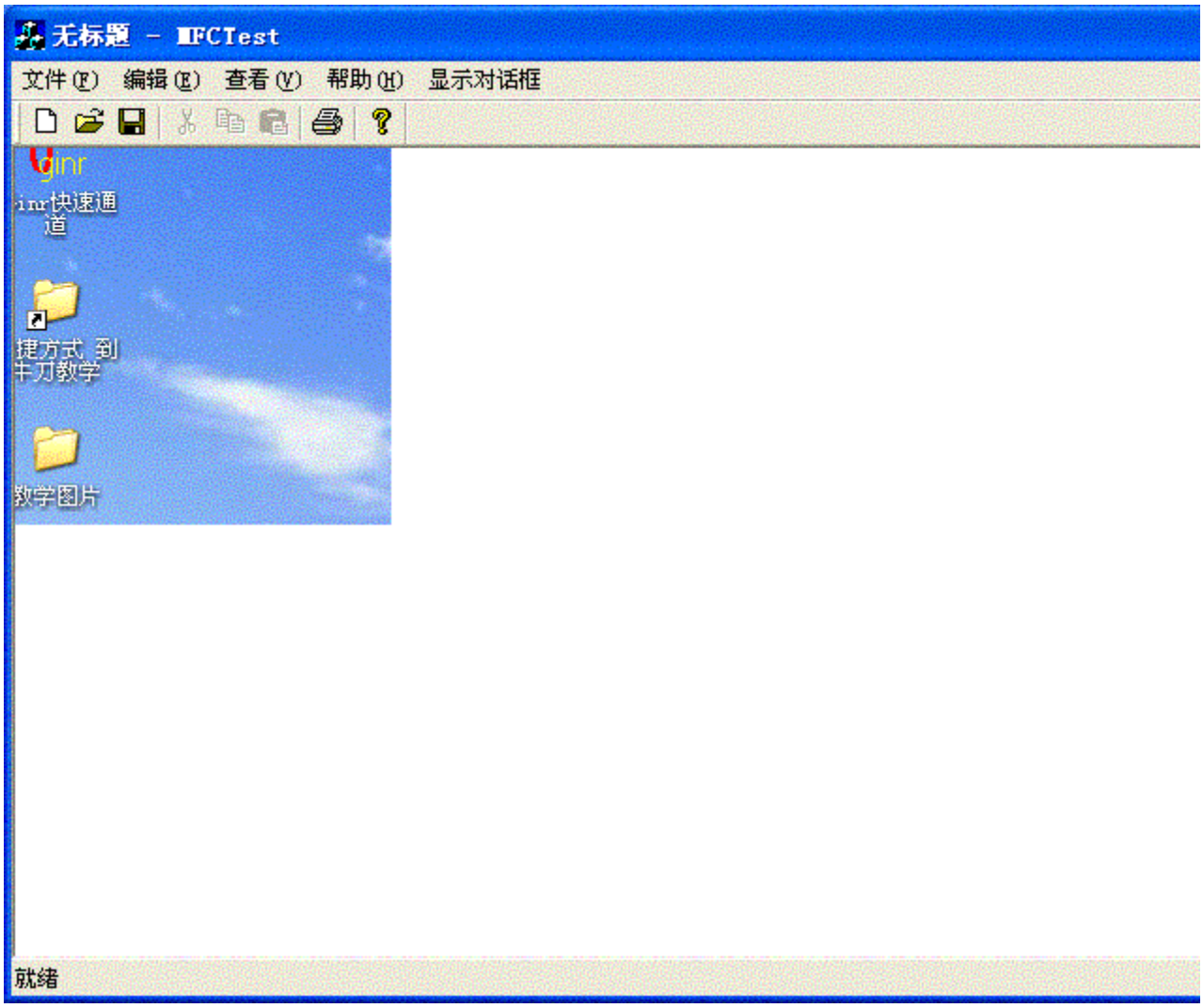
.....

事实果真如此，我们使用这种技术完全可以在屏幕上画出一幅图片的，大家这一讲先将下面一段程序敲上电脑，试试我们以上的那种技术，另一种技术，我们将会在以后的章节中与大家详细讨论：

```
int CMFCTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
  return -1;
SetTimer(0,100,NULL);
return 0;
}
void CMFCTestView::OnTimer(UINT nIDEvent)
{
HDC hdc=::GetDC(NULL);
CPoint pt;
GetCursorPos(&pt);
COLORREF clr;
```

```
CClientDC dc(this);
for(int i=-100;i<100;i++)
{
  for(int j=-100;j<100;j++)
  {
   clr=::GetPixel(hdc,pt.x+i,pt.y+j);
   dc.SetPixel(i+100,j+100,clr);
  }
}
CView::OnTimer(nIDEvent);
}
```

当然，以上的程序，你可以充分地体会到什么叫做速度，你的画面总是一跳一跳的，这就是我们内存与显卡一直拍电报的代价。




当然，下一讲，我们就要解决这个问题。

**微软积木〇八四：莫惜金衣**

这几讲与大家讲时钟，突然想起了唐代杜秋娘的一首诗——《金缕衣》：

劝君莫惜金缕衣，劝君惜取少年时。
有花堪折直须折，莫待无花空折枝。

好多的事都被淹没在时间中了，你洗手时，时间从你指间流过。

行了，不发感慨了，还来看看昨天的最后一段代码吧：（牛刀改了改）

```
int CMFCTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;
        SetTimer(0,100,NULL);
        return 0;
}

void CMFCTestView::OnTimer(UINT nIDEvent)
{
        HDC hdc=::GetDC(NULL);
        CPoint pt;
        GetCursorPos(&pt);
        COLORREF clr=::GetPixel(hdc,pt.x,pt.y);
        CClientDC dc(this);
        dc.SetPixel(20,20,clr);

        CView::OnTimer(nIDEvent);
}
```

在 OnCreate 消息映射函数中，不需要与大家详细说吧？其实仅仅是创建一个时钟而已，第一个参数其实会通过下面 WM_TIMER 响应函数的 UINT nIDEvevt 中传递过来，表示我们当前需要处理哪一个时钟，也就是说，我们可以在 OnTime 函数中处理若干不同的时钟，当然，我们一般可以用 switch 来选择某一个时钟来处理。

当然，如果我们仅申请一个时钟那么这个参数的意义就不是太大。

SetTimer 的第二个参数是设定的时间间隔，以毫秒为单位，也就是说，如果设定为 1000 ，那么每隔一秒钟就会发一个 WM_TIMER 消息响应函数，但有一个缺点，就是这个时钟不是很准，原因是我们的每一个消息都是发送到消息队列中的，也就是说，产生的消息一般情况下是不容易得到及时的处理的，所以这里的误差很大，所以一般我们如果要精确定时的话，不使用这一个消息，而在

一般的动画制作中，这个消息那是很常用的喽，因为从动画的角度来讲，我们的这种时钟也就足够精确的了，呵呵，以后有空再与大家详细讲动画吧。

第三个参数其实是一个函数调用，如果设置为 NULL ，则这个设置会每隔一段时间就发送一个 WM_TIMER 消息，如果这儿是一个函数指针，那么就不发送消息了，而是隔一段时间就调用相应的时钟响应函数，那当然就比发消息要准一点喽，呵呵。

我们一般使用喜欢用其发消息，然后做消息响应，因为在消息中，很容易得到一些窗体信息，从而整合我们的程序，当然喽，如果我们仅仅为了计时，可以定义函数，毕竟那么要比发消息精确一点，是不？

其实在 Windows 时代，想做到真正的精确的确不是很容易的，因为毕竟 Windows 给我们的都是封装过了的 API 函数啊，呵呵。像如果你想做个实时系统啊，做个设备控制啊，那还真得去学学汇编才行，但这儿既然讲到 MFC 了，就暂时不与大家去复古了，呵呵。

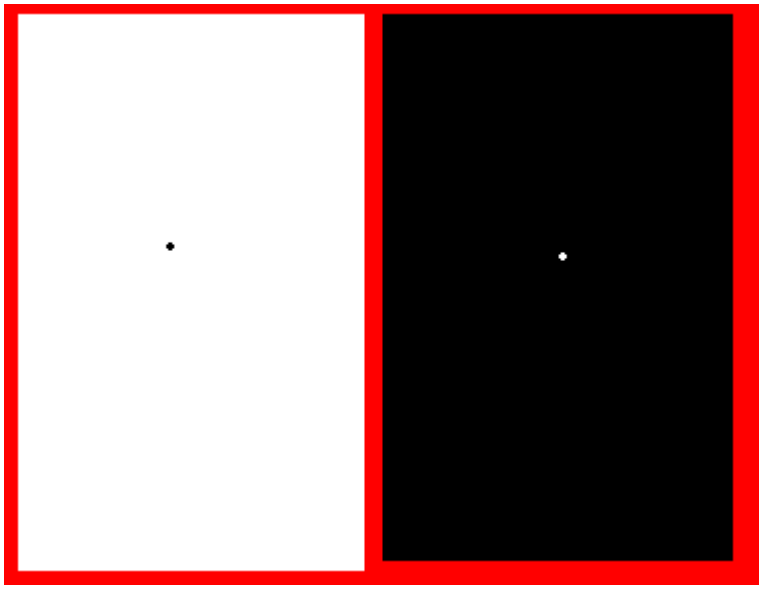
好，再看看 WM_TIMER 的消息响应函数中的情况， HDC ，这个句柄想看过前面教材的应该知道，必须有的哟，当然，我们下面还有一个句柄被 CClientDC 给封装了。我们为什么要用句柄，而不使用土封装的类呢？因为我们这次需要得到的是屏幕上的句柄，所以我们拿不到屏幕的指针（至少我们目前的水平还拿不到，是不是？）。但我们可以用全局函数 ::GetDC 传进一个 NULL 参数，从而得到整个屏幕的句柄，所以大家知道，为什么我们要用句柄了，因为我们没有窗口句柄可以使用，当然如果你能得到窗口的句柄，一样可以封装的哟，假如你上面不加上 “::”，你就会看到 GetDC 中不需要参数，所以你根本不能通过传参数办法来得到屏幕句柄。

好绕啊，如果你还不明白的话，你就知道这儿我们想得到屏幕就必须使用 API 函数，而不能使用 MFC 的技术就行了。

下面是定义一个点，当然，我们一旦鼠标移出屏幕后，鼠标的任何消息，我们的窗口都接收不到了，所以想得到鼠标当前位置，是不能通过消息的办法获得，只能用 GetCursorPos 函数去获得鼠标在前所在的坐标信息。

COLORREF 是一个双字类型的数据，也就是四个字节，所以可以表示卅二位的数据可惜因为红绿蓝只有廿四位数据，所以一定有八位为零，而新型的３２位的颜色值中，可以用八位作为亮度信息，那么我们的电脑看起来就更为丰富了，对了，告诉你一个秘密，我们的 RGB(0,0,0) 是什么颜色？可能你认为是黑色，但请大家注意，我们的显示器是做不到真正的纯黑色的，除非你是等离子的电视，并且其中有亮度信息去控制每一个点的亮度（牛刀是这样想的，呵呵）。其实这没办法，因为我们的显示器上的点根本就不是黑的，而是有三种颜色的，你想真的不发光，那也不是黑的啊，呵呵。你的显示器就是关了，上面也不是纯黑色的啊。

::GetPixel 是得到屏幕上某点的颜色，而 SetPixel 是设置某点的颜色。请大家注意，不要说屏幕上某个像素有没有点，而应该说屏幕上某个像素的颜色是什么。这一个像素，可能是白色，也可能是黑色，那么黑色就是点吗？白色就不是点吗？请大家看两个图：



是不是在左边，大家会将黑的当着点，而在右边，大家又会将白的当着点？其实就是这样，大家尽量不要认为我们的 SetPixel 是打点，而这个函数仅仅是改变像素的值，真的，别被我们的眼睛给蒙了。

对了 SetPixel 在改变像素值之前又可以将当前点的未改变之前的值返回给我们，效果与 GetPixel 效果一样，只不过是一个改变了当前像素的值，一个没有改变，仅此而已。

可奇怪的是，昨天截取屏幕的那最后一段程序，瑞星的小狮子就是截不下来，呵呵，不知什么原因。

行了，对于以上的程序，我们就解释到这儿吧，这儿是简化了，上一讲那个程序来处理图片，大家一定会觉得——那是那么慢，是吧？

的确，通过内存与显卡的拍电报来处理图片，果真是效果不好，太慢了，于是，我们就需要将大部分事情交给显卡去做，那样就快多喽：

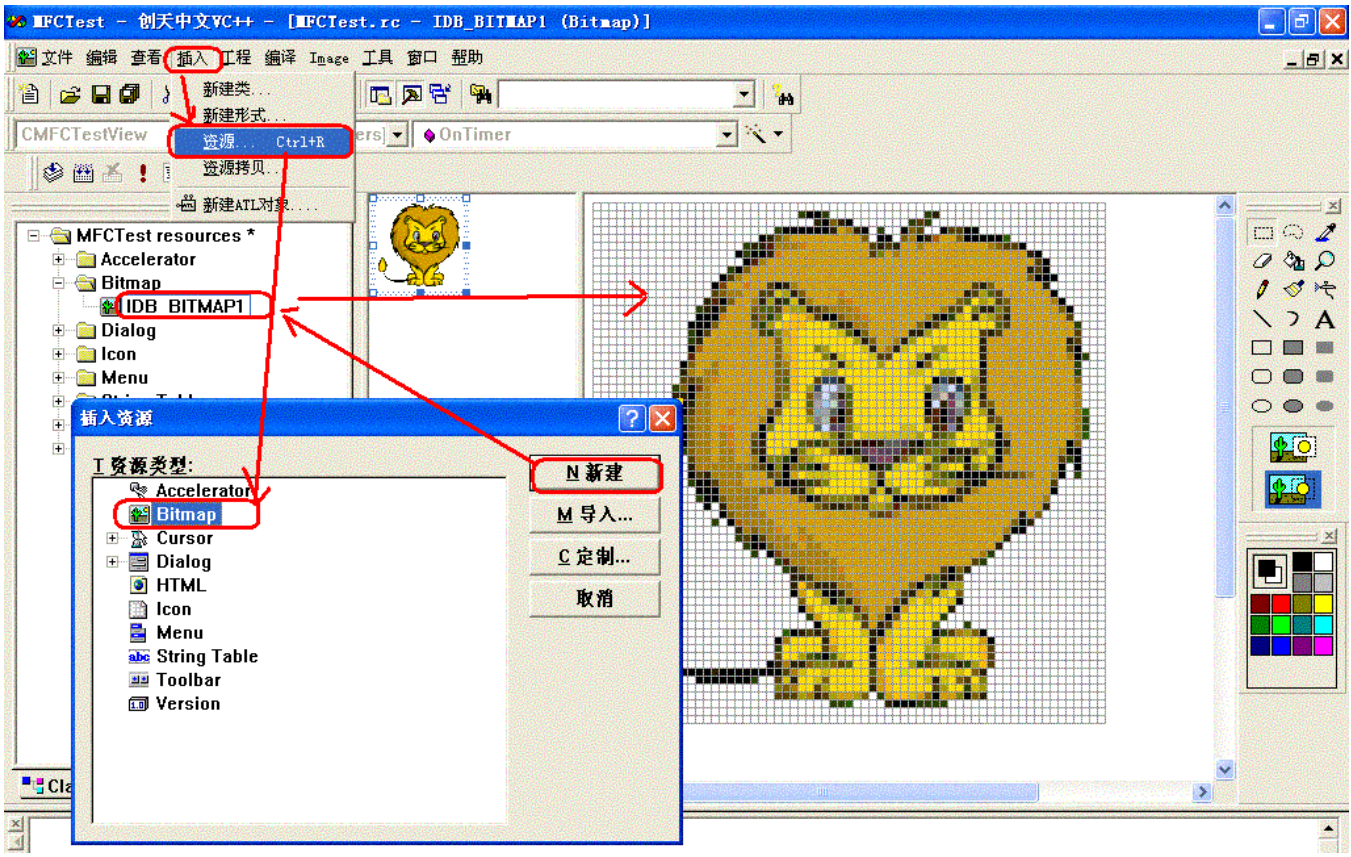
显卡，显卡，我是内存，我是内存，显卡听到请回答。
显卡听到，显卡听到，内存有何事件。
内存这有一幅图片，显卡请帮我处理。
显卡知道，那么请你传真过来吧
好，请接传真。
*兹兹兹……*
显卡接到全部传真，这幅图片好好漂亮。

请你帮我贴在屏幕上好吗？
显卡明白，显卡明白。

于是，我们的屏幕上就有了一幅图片。

可能你还觉得过程不得劲是不？那么我们就用我们的 VC 去创建一幅图片吧：

这次咱们就使用瑞星的小狮子去做图片了，好多朋友对我屏幕上那个小狮子感兴趣，这回牛刀就选中她喽，看图：




这幅图，我们改个名吧，大家在 IDB_BITMAP1 上按鼠标右键，按最下面一个 Properties，弹出一个属性框，在其 ID 后面的文本框中输入 IDB_LION，回车确认，好喽，我们进入 OnDraw 函数，去作图去喽：

```
CBitmap bmp;
bmp.LoadBitmap(IDB_LION);
```

当然你可以说，为什么不用 Create 去创建一个啊？当然可以创建的啦，但我们这儿没有必要，也就是说，这种事情，都由编译器给我们去办喽，呵呵，你还别说，在没有这种资源编辑器的时代，你以为你要创建这些东东不需要用

Create　去创建啊？那简直是做梦，呵呵。

不错，一个　LoadBitmap　就将我们的创建工作做完喽，就是那内存与显卡说了，我这儿有一幅画，呵呵，你想想啊，显卡那没这幅画啊，于是就只有对内存说，你啊等等啊，我这准备一张纸（大家还知道牛刀在电脑中说的那张纸吗？其实就是一个　HDC　，就是一个设备描述表的句柄，当然，我们假假这张纸是放在我们的内存中的，其实也是在我们内存中的，只不我们从理解上，理解成在显卡中的更好理解，如果有内行人士看到这放牛刀一马啊，呵呵，牛刀这也被逼没办法，在这瞎说呢，呵呵）。你再传啊。

于是：

```
CDC mydc;
```

当然，这张纸还不能用，为什么呢？我们才定义这张纸啊，现在我们需要将这张纸初始化一下，什么意思呢？因为我们在内存中，定义一个句柄后，这个句柄有多大？（大小）有多少种颜色？（色彩），以及其他的一些属性，都要定义的，那多费事啊，于是，显卡有办法啊，就根据我当前的设备描述表假造一份吧，于是，这种造假的纸，我们称之为兼容纸，也称兼容设备描述表或兼容 DC。

```
mydc.CreateCompatibleDC(pDC);
```

请大家注意，虽然这里我们说是显卡要准备的，其实这张纸还在我们的内存中，原因很简单，因为我们需要在内存中对这张纸进行操作，所以说，理解上是在显存那儿，但事实上我们的操作，还是在内存中进行的，也正因为如此，我们就可以将这幅图片先放到这张纸上，我们就假设传真过去了，其实还没传，那怎么放到那张纸上呢？还是用　SelectObject　函数啊，和画刷，画笔一样一样的啊：

```
mydc.SelectObject(&bmp);
```

好了，传真过去了，真的传真过去了吗？还没有。

但为什么我们说那张纸是显卡那儿的呢？因为如果你说不内存中的，那通过 pDC 去创建兼容 DC 你说不太好理解，其实是显卡将　DC 给传过来了：

内存内存，显卡这遇到技术障碍，没纸了，先传个句柄给你，你给我在你的内存中创建一个兼容 DC，然后你的那张画画到兼容的　DC　上吧。那样我就可以知道那张画了。

那怎么传给你呢？你那又没纸。

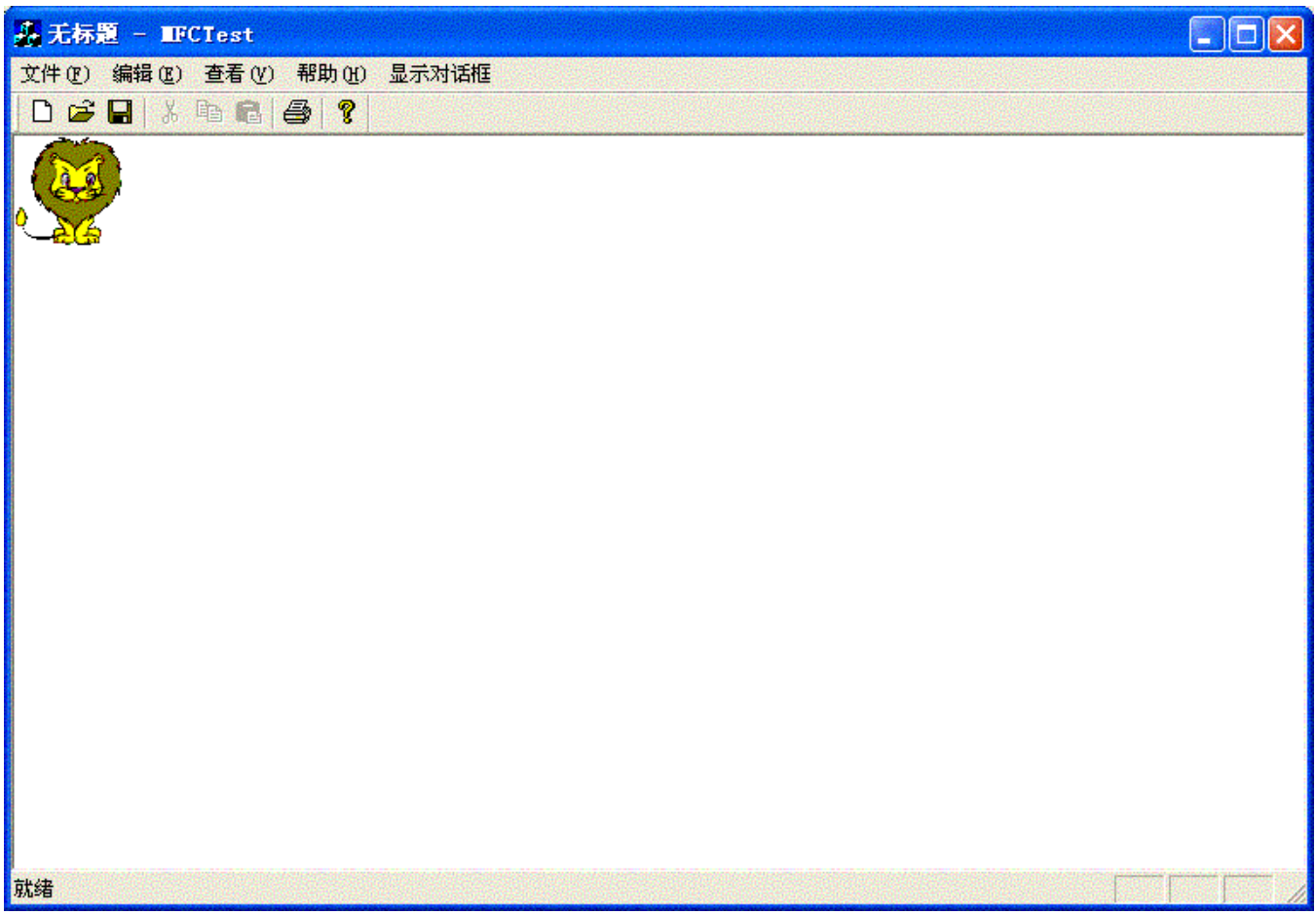
嘻嘻，你傻得可爱，如果全靠电脑了，谁还用纸啊，我当黑客去将兼容　DC 给弄过来就行了。

啊!!! 别，我怕怕。

呵呵，怕也没用，我这一个 BitBlt 就可以过来了，你看：

pDC->BitBlt(0,0,200,200,&mydc,0,0,SRCCOPY);

效果不太好，但总算没白费：




那小狮子有点变色了，什么原因，因为我们的图片颜色值只有 16 色，所以效果不好，当然，如果要效果好的，那就需要改改其调色板了，什么叫调色板，怎么改，我们以后再说，这一讲，大家将究着看吧，呵呵。

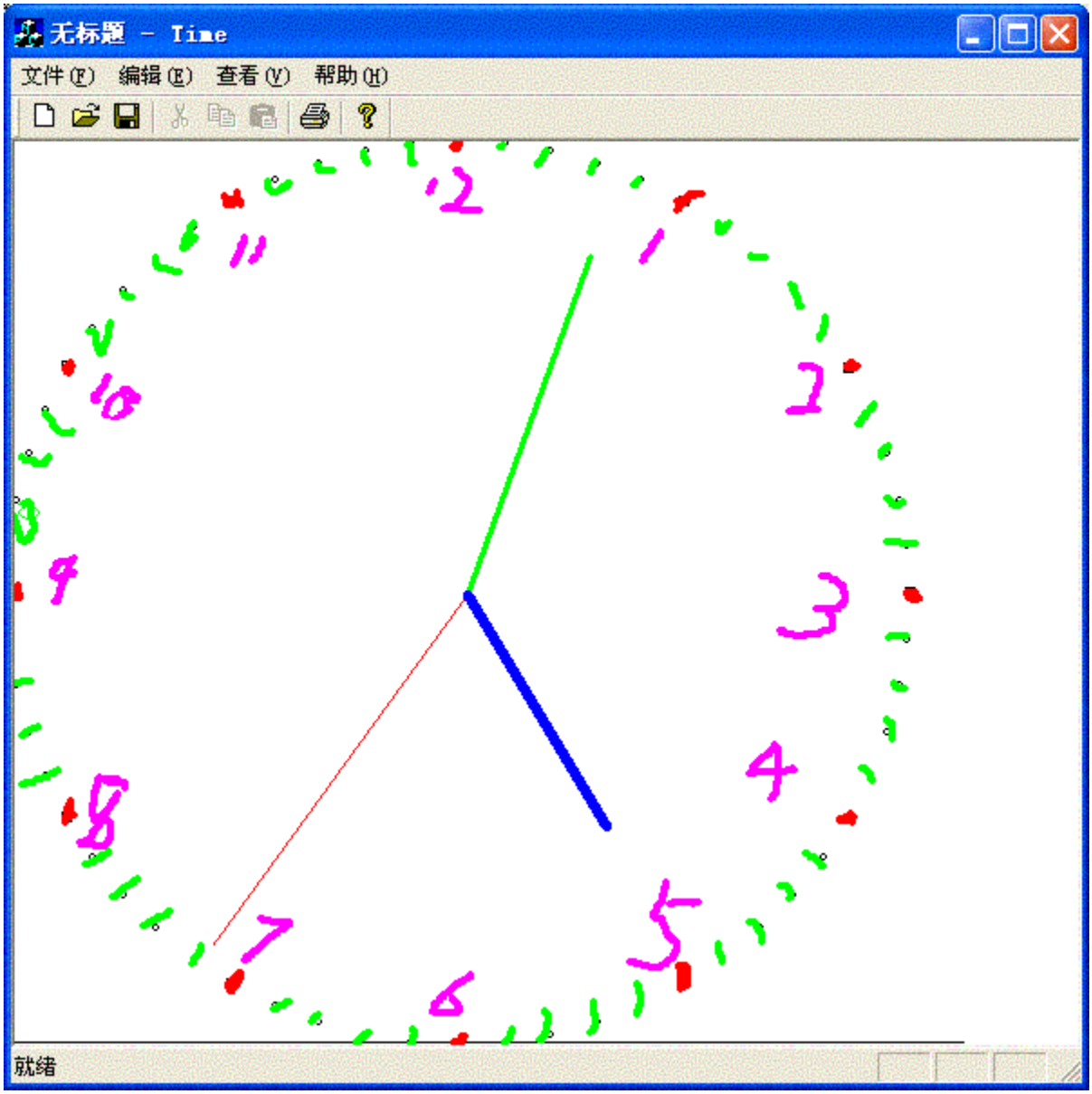
从上面来说，我们需要做一幅贴图，我们就需要以下的过程：
·定义一幅图片 一般使用 LoadBitmap 来载入
·定义一个内存的 DC，并且用当前需用的设备 DC 来初始化一个兼容的内存 DC
·将图片选入兼容的内存 DC
·使用 BitBlt 将内存中的 DC 传真给显卡去显示图片(真正的传真应该是这一步，但也不完全是，因为任何两个相容 DC 都可以通过这个函数进行图形传输。

慢着慢着，不是讲时钟吗？是不是牛刀这一讲好像跑题了？

牛刀好冤枉啊，牛刀讲的是技术啊，没有这个技术，我们怎么与大家深入呢？好啊，为了证明牛刀没有跑题，牛刀就将这种技术用到我们的 Time 软件中来：

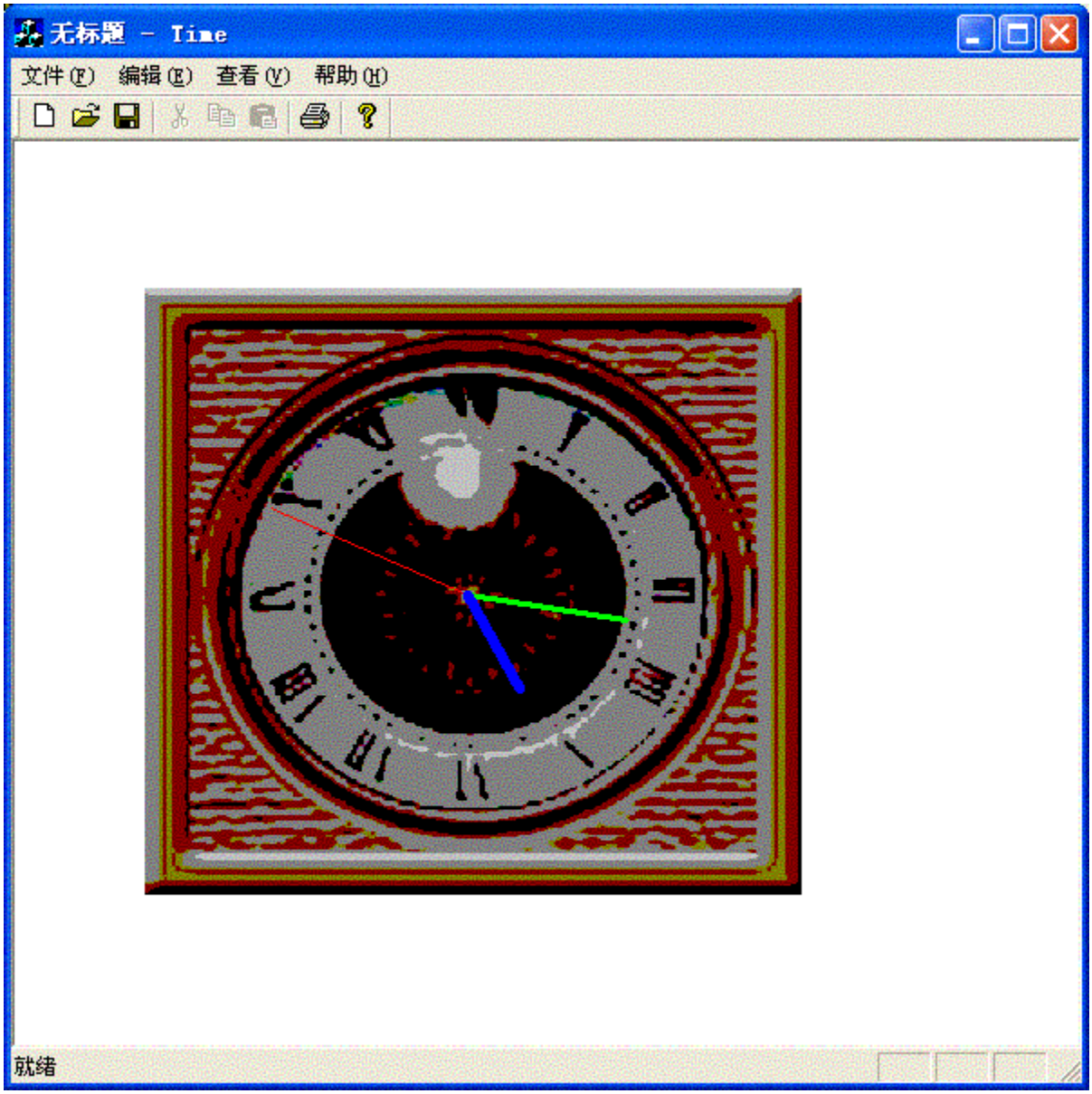
```
void CTimeView::DrawFace(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TIME);
        CDC timDC;
        timDC.CreateCompatibleDC(pDC);
        timDC.SelectObject(&bmp);
        pDC->BitBlt(0,0,550,500,&timDC,0,0,SRCCOPY);
}
```

看，一下子，我们的 DrawFace 函数变得多么得简炼？而效果却是，咳咳：



感觉怎么样，关键的一点还有，我们的画布跳得不是那么厉害了，这是为什么呢？因为我们需要的数据都是传真传过去的啊，而不是一个一个点地拍电报过去的啊，呵呵，就是这么厉害。

当然喽，你只要选择一幅好的图片，你的软件就会显得格外地妙不可言：




效果当然仍旧不是太多，但如果比起原来那种单调的感觉来说，那可是一天一地喽，至少我们知道，我们的软件怎样做做得漂亮了，是也不是？

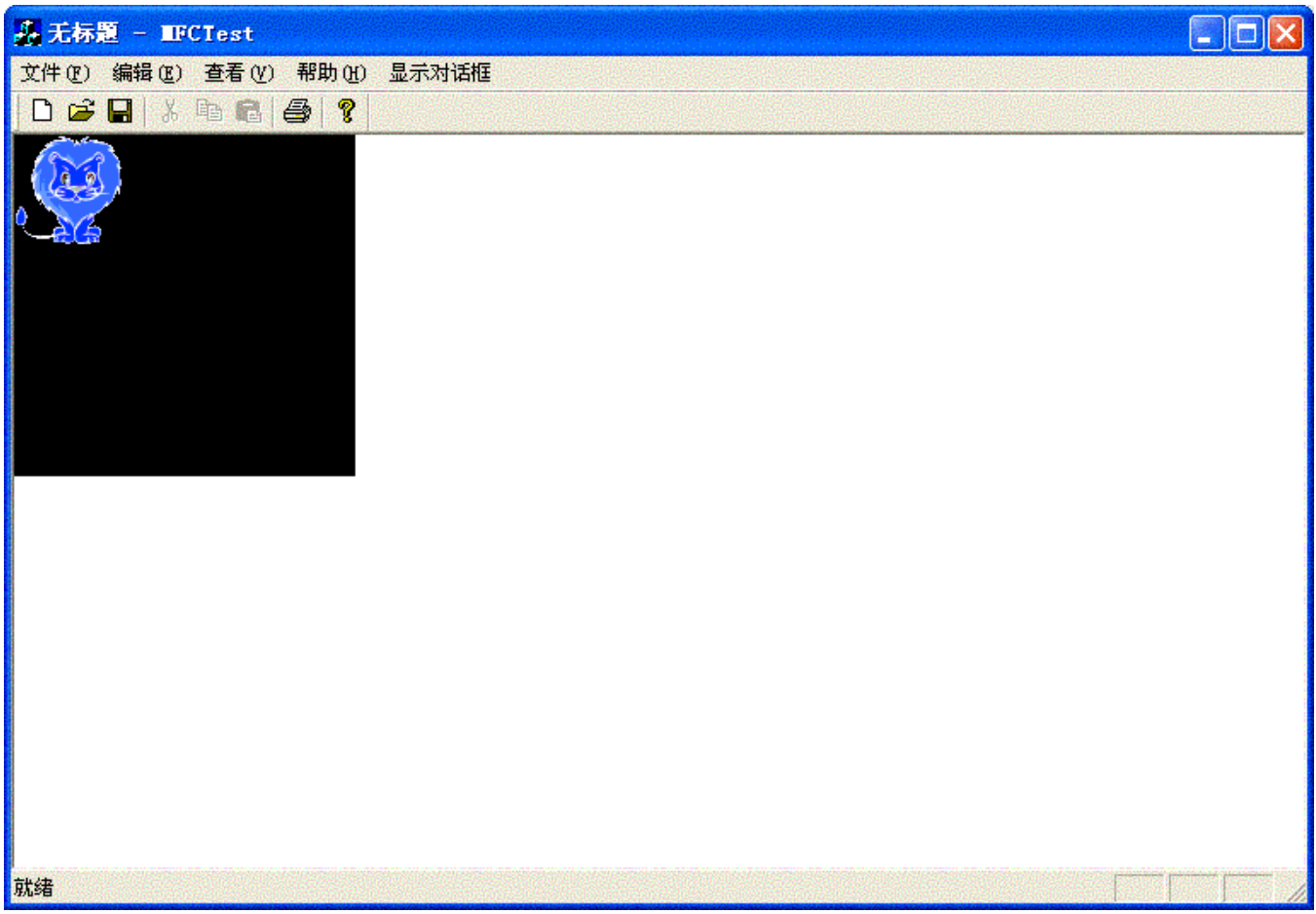
好了，接下来，我们将与大家讨论这个 BitBlt 这个函数，当然，这个函数，也不是这一讲就能够讨论完的，因为这个函数的功能好强大好强大说给你你很可能不太相信，这个函数竟然不是处理一幅图像的函数，而是同时处理三幅图像的一个强大的函数，我们称之为“位元传送函数”。

当然，这个函数功能强大，但比较复杂，所以我们不得不从一个简单的函数 PatBlt 谈起

我们还回到上一章我们画的那个小狮子那儿，大家用下面的代码：

```
……
        pDC->BitBlt(0,0,200,200,&mydc,0,0,SRCCOPY);
        pDC->PatBlt(0,0,200,200,PATINVERT);
}
```

这次我们看程序，怎么回事？




这是怎么回事？具体是怎么回事？我们下一讲再具体与大伙讨论。
知识点回顾：

·定义一幅图片 一般使用 LoadBitmap 来载入
·定义一个内存的 DC，并且用当前需用的设备 DC 来初始化一个兼容的内存 DC
·将图片选入兼容的内存 DC
·使用 BitBlt 将内存中的 DC 传真给显卡去显示图片

只要我们需要在我们的屏幕上贴图，就必须有上面四步，一步都不能少，当然，以后我们还将详细地展开。

**微软积木〇八五：桌面贴图**

这一讲暂时与我们的时钟的设置离开一会儿，专门来讲一讲贴图的事。原因很简单，不会去做贴图，那么我们的软件就会少去一块风景，何况我们好多的软件，都是需要良好的图形界面的。

可以这样来说，懂得了贴图的技术，其实不需要用 Windows 也可以做出 Windows 的效果来，当然，这种技术我们不打算详细地展开，因为其中还需要自己去设计各种的消息机制，一个是有点烦，二一个是，我们目前的技术还达不到那种要求，呵呵，还是那句话，一口不能吃个胖。

这儿与大伙说一句，上一讲，我们讲过贴图需要有四个步骤，其实并不一定都要经过那四个步骤的，我们来回忆一下我们以前曾经讲过的一个贴图的方案——位图刷子，我们在 OnDraw 函数中，做一个画背景的图吧：DrawBk

```
void CMFCTestView::OnDraw(CDC* pDC)
{
DrawBk(pDC);
}
void CMFCTestView::DrawBk(CDC *pDC)
{
}
```

这里我们准备用刷子将整个底面背景给刷一遍，如果大家还记得以前的代码，一定会可以写出以下的代码：

```
void CMFCTestView::DrawBk(CDC *pDC)
{
CBitmap bmp;
bmp.LoadBitmap(IDB_BACK);
CBrush brush;
brush.CreatePatternBrush(&bmp);
CRect rect;
GetClientRect(&rect);
pDC->FillRect(&rect,&brush);
}
```

看，我们并没有使用上一讲我们用的技术，就完成了贴图：

```
CBitmap bmp;
bmp.LoadBitmap(IDB_BACK);
```

创建并选入一幅位图

```
CBrush brush;
brush.CreatePatternBrush(&bmp);
```

用位图定义一个画刷

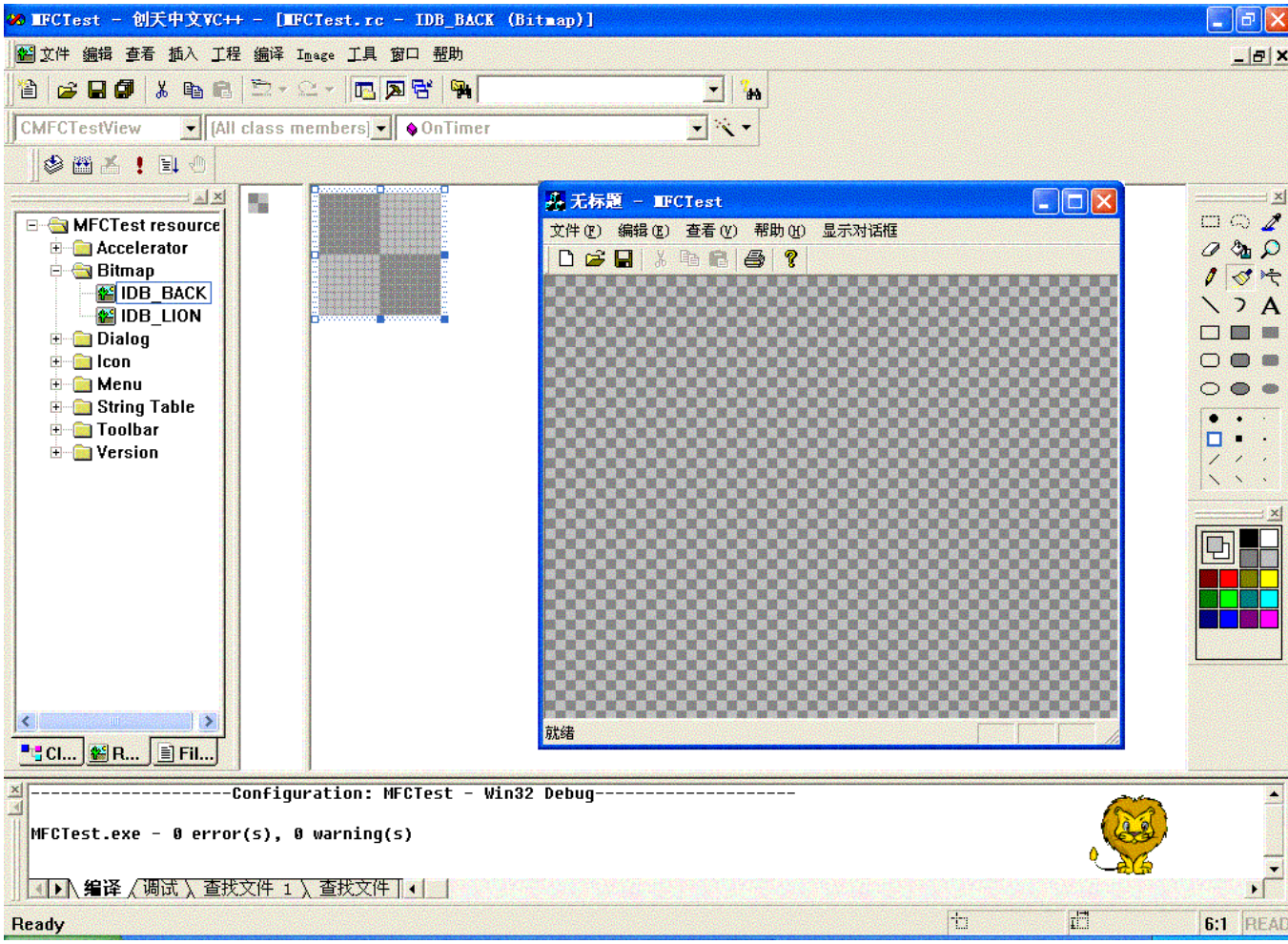
```
CRect rect;
GetClientRect(&rect);
```

得到客户区的矩形区域

```
pDC->FillRect(&rect,&brush);
```

用画刷去填充这块矩形

事实就是这样，做出来的效果如下：




如果大家用过 PhotoShop 是不是感觉这有点像透明的效果啊？呵呵牛刀的目的也就在于此，我们假假地将这当着一个透明的背景，这样，我们在上面的操作，我们就知道是怎么回事喽。呵呵（注：想做个 PhotoShop 软件吗？相当简单，这几讲讲过了，你就可以去试试，当然一下子做出那么多的功能来显然是不现实的，但一个一个的功能去实现那是完全可以的。）

好了，当然，要做出那样的软件来还需要一步一步地慢慢来。

我们先来测试一下 PatBlt ，其实也是图形传送，其实这次传送的图形，就是我们的刷子与我们背景图片的结合，当然，为了大家能够真正地理解这把刷子，我们用一个位图刷子，并且在黑方块中画了个白圆作为刷子的图像，去做一些测试，所以我们定义一个函数叫 PatTest：

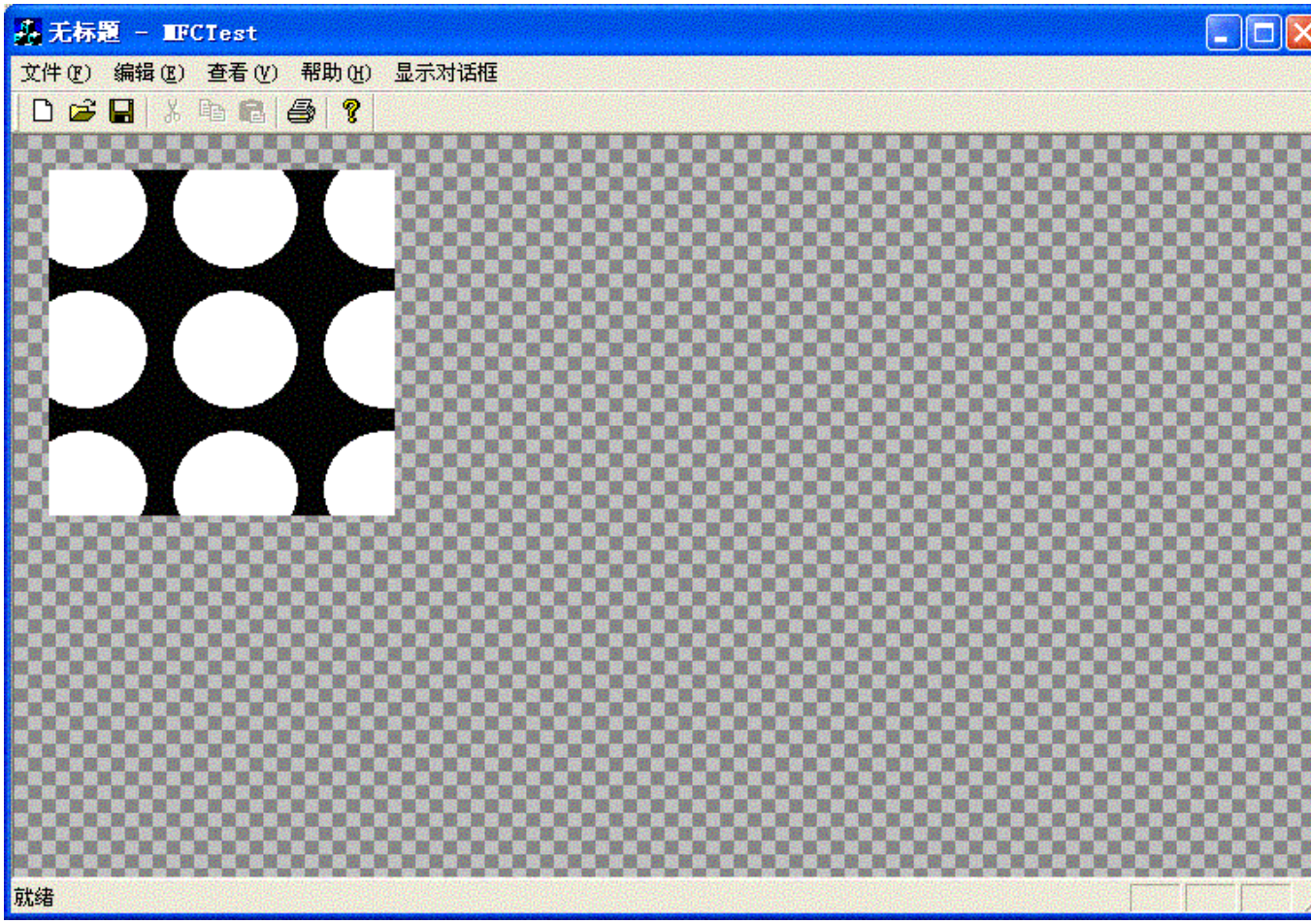
```
void CMFCTestView::OnDraw(CDC* pDC)
{
DrawBk(pDC);
PatTest(pDC);
}
void CMFCTestView::PatTest(CDC *pDC)
{
```

```
}
```

这一次，我们一样地创建一个图片刷子，只不过这次我们使用一个 IDB_BRUSH 的图片：

```
void CMFCTestView::PatTest(CDC *pDC)
{
CBitmap bmp;
bmp.LoadBitmap(IDB_BRUSH);
CBrush brush;
brush.CreatePatternBrush(&bmp);
pDC->SelectObject(&brush);
pDC->PatBlt(20,20,200,200,PATCOPY);
}
```

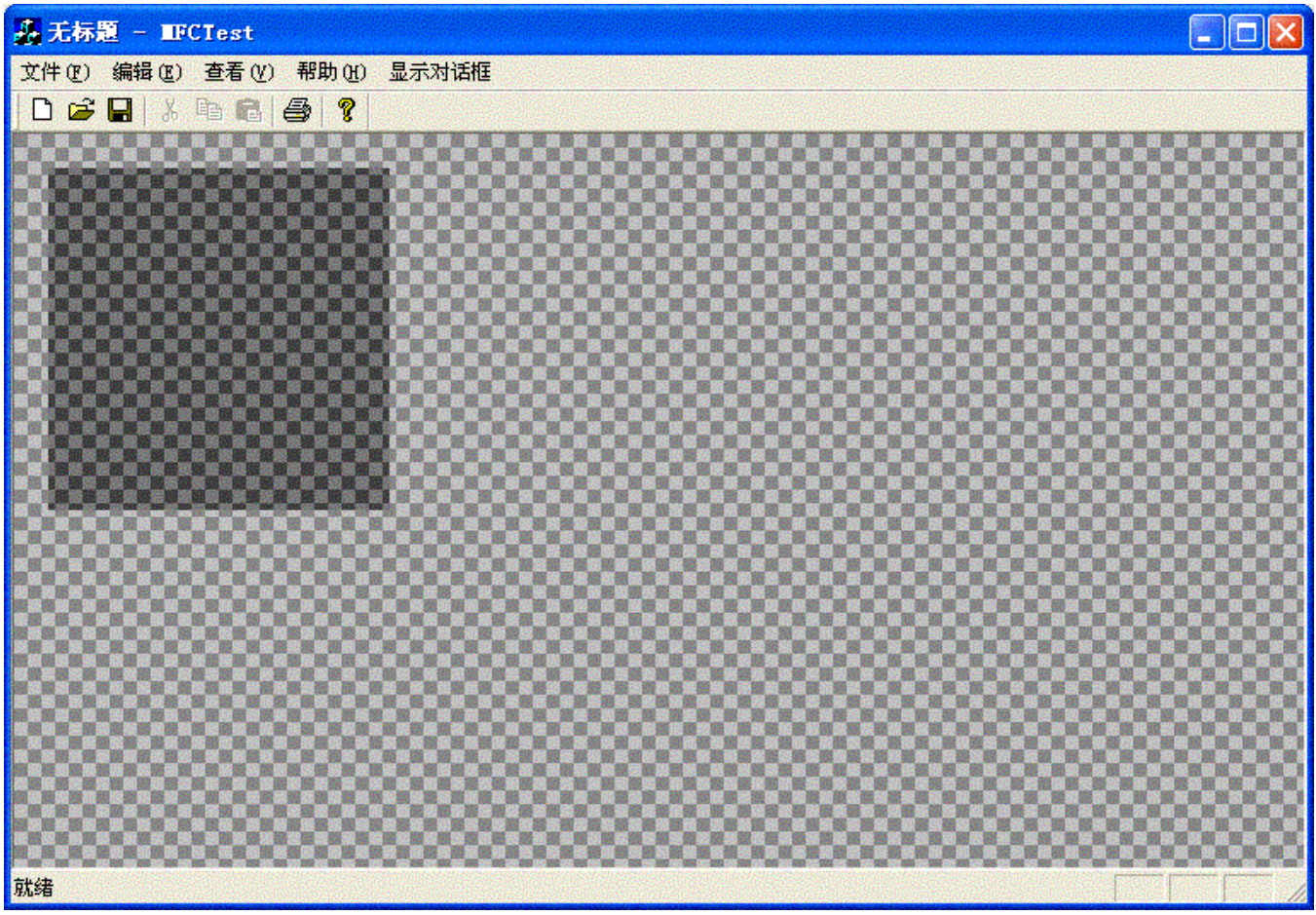
这当然是对选入的刷子进行操作喽，呵呵， PatBlt 中的第一个与第二个参数分别为左上角的横纵坐标，第三个为宽度，第四个为高度，而第五个参数即为模式，例如我们上面的图，是 PATCOPY 模式，也就是显示的就是将刷子复制过来了，那么效果如下图：




当然，我们可以很容易地改变一个模式，如：

```
pDC->PatBlt(20,20,200,200,DSTINVERT);
```

效果如下：




大家看，没有用画刷，而是背景的颜色被反转了，当然，系统还为我们定义了一些其他的效果，如：

PATINVERT

BLACKNESS

WHITENESS

当然，还有好多没有命名的，那是怎么回事呢？

我们将刷子上的图案称之为“图案（P）”，而我们贴到的地方称之为“目标（D）”，那么，我们图案与目标之间就会有十六种组合：

P D 结果

0 1 2 3 4 5 6 7 8 9 A B C D E F

0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1

0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1

1 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1

0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

是不是有点像在看《天书》？没关系，我们下面就来解读这本《天书》

P D 结果

0 0

0 1

1 0

1 1

这儿我们称之为“位元”操作，也就是我们是按一位一位进行操作的，那么我们假

设 0 就为黑色，1 为白色，那么假设我们现在要在刷子（图案[P]）上为黑色 0 时显示背景色，而白色部分不变，那么我们将上面找到黑色的部分，也就是 P 下面是 0 的那个位置，我们在结果上将 D 的结果写上，如下：

P D 结果
0 0 0
0 1 1
1 0
1 1

看，就这样，我们就可以保证我们刷子的黑色部分被背景替代，而白色部分呢？我们假设需要将白色的部分显示成黑色吧，也就是说，反个颜色，那么我们只需要在 P 为 1 的那里填上 0 ，也就表示反色了：

P D 结果 注释
0 0 0 图案为黑的部分
0 1 1 背景不变
1 0 0 图案为白的部分
1 1 0 显示黑色（结果为 0）

好了，我们将结果的从下向上看形成一串二进制索引：0010

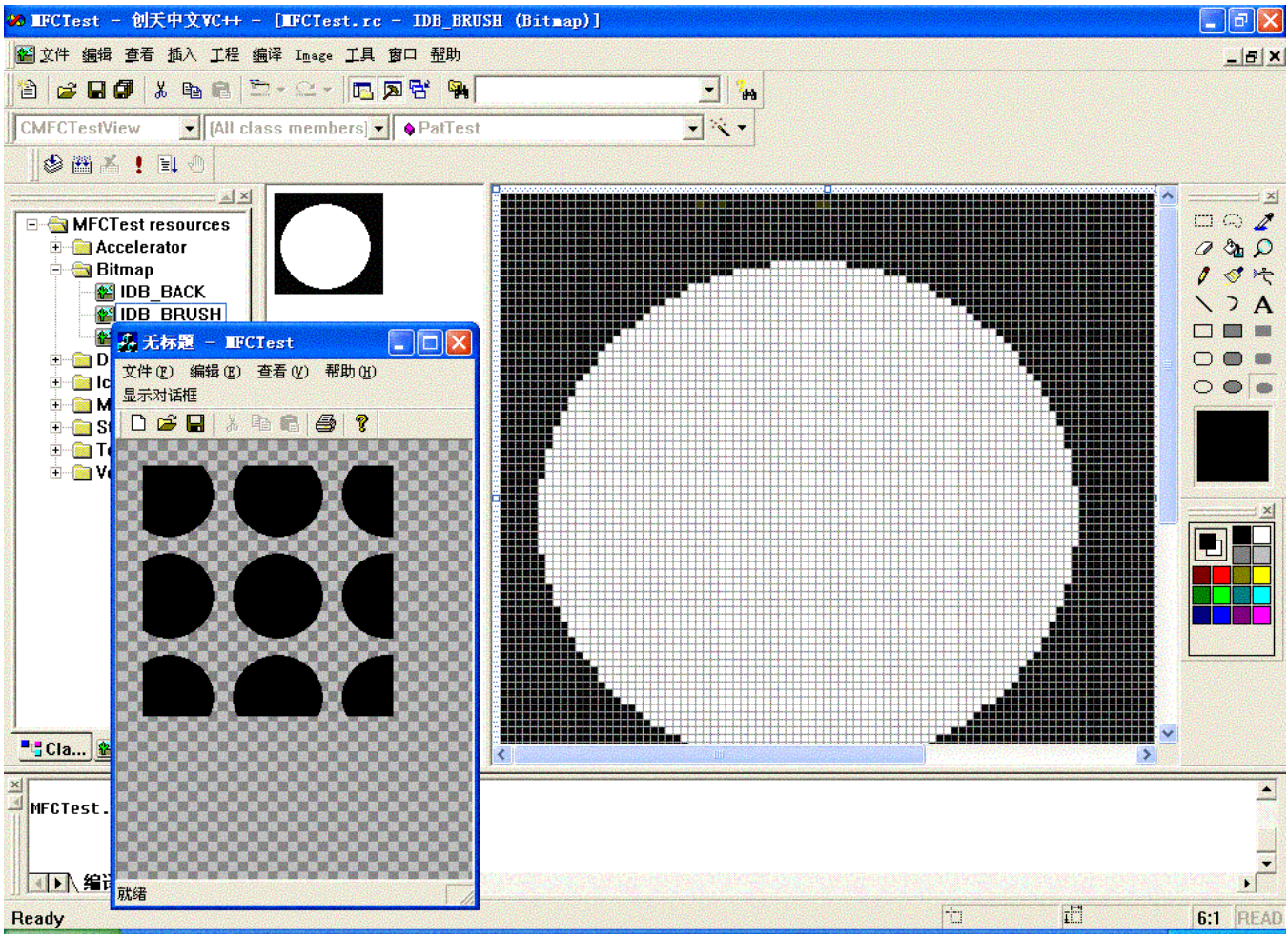
我们通过这个索引去查找一个模式代码，可查表得：

0x0a0329

好了，那么我们就用这个代码去 PatBlt 一个区域，看看能不能达到我们的目标：

```
pDC->PatBlt(20,20,200,200,0x0a0329);
```

看看我们的效果：




大家可以看到，我们黑色的外矩形在我们程序的显示中，变成了透明，我们可以看见底面的背景了，而我们设定为白色的圆，顺利地变成了黑色，这就是我们“模式”的设置的原理。

于是，正因为有这种效果，使得这种黑白的图案可以自由地将我们需要擦除的部分图形像蒙上一块画布一样，所以在 PhotoShop 中，这种黑白图片，也称之为“蒙板”。

当然喽，蒙板不一定必须是黑白图片，下面我们将里面的白色变成红色，然后大家再运行程序，一看，怎么感觉中间的圆有反色的效果了？是吗？给大家去试验，牛刀就不贴图了，相信大家都可以得到同样的结果的，并且反色是反色，是不是也变得透明了，或者说是半透明了，如果大家打游戏，这下就知道半透明的图片是怎么来的吧？

当然，牛刀不喜欢反色，那不反色怎么办呢？还是来看上面的那个《天书》中的模式设置：

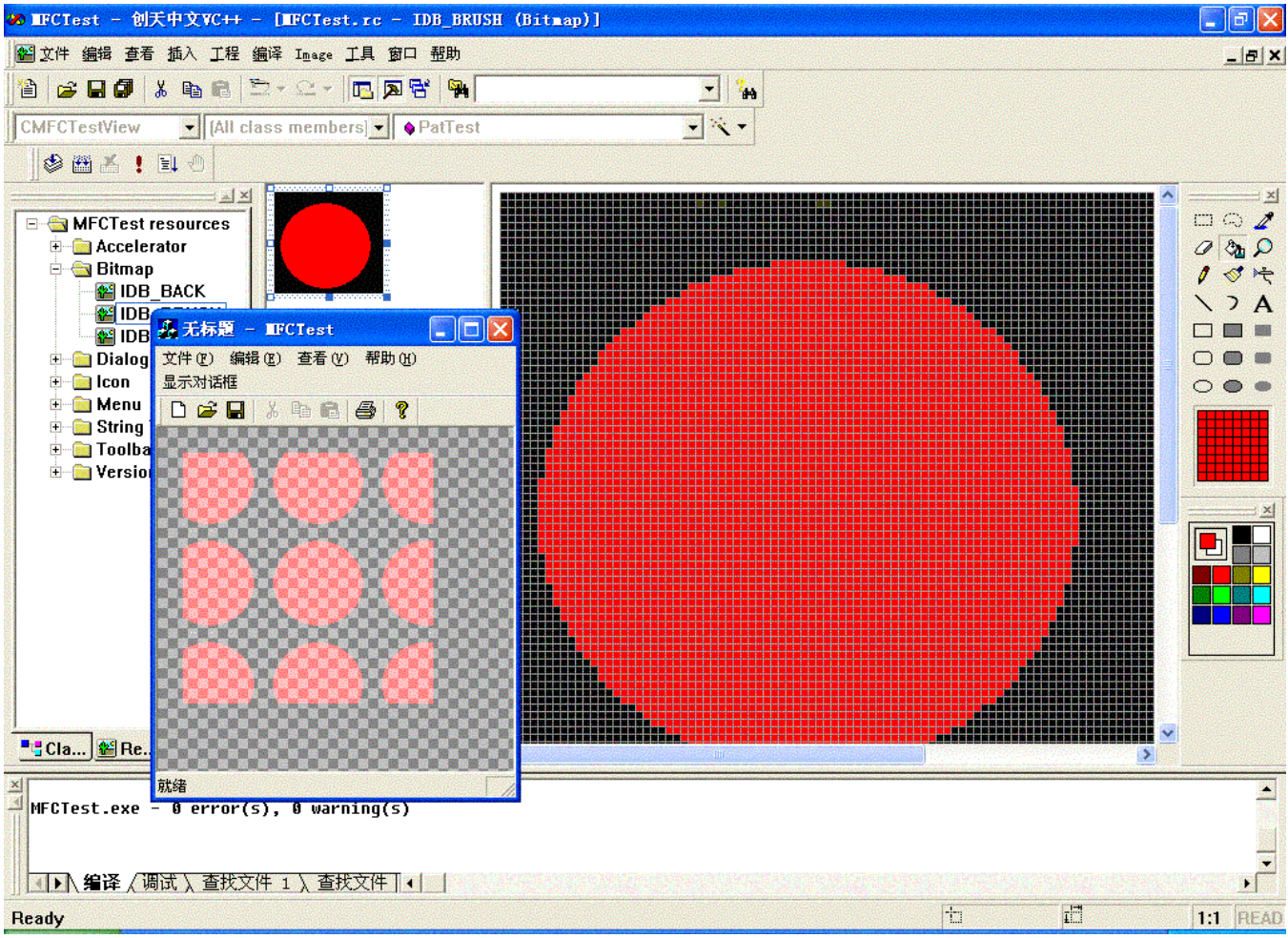
P D 结果 注释
0 0 0 图案为黑的部分
0 1 1 背景不变
1 0 1 图案为白的部分
1 1 1 显示不变

于是，从下往上是　1110　我们再查一个表就可以得模式代码：　0xfa0089

好，再来：

```
pDC->PatBlt(20,20,200,200,0x0fa0089);
```

感觉真舒服：




你可能对牛刀有意见了，那么那个表在哪儿呢？好了，下面就给大家这款表：

| 1 1 0 0 P | | |
|---|---|---|
| 1 0 1 0 D | 布尔操作 | 模式代码　常用定义 |
| 0 0 0 0 | 0 | 0x000042 BLACKNESS |
| 0 0 0 1 | ~(P \| D) | 0x0500A9 |
| 0 0 1 0 | ~P & D | 0x0A0329 |
| 0 0 1 1 | ~P | 0x0F0001 |
| 0 1 0 0 | P & ~D | 0x500325 |
| 0 1 0 1 | ~D | 0x550009 DSTINVERT |
| 0 1 1 0 | P ^ D | 0x5A0049 PATINVERT |
| 0 1 1 1 | ~(P & D) | 0x5F00E9 |
| 1 0 0 0 | P & D | 0xA000C9 |
| 1 0 0 1 | ~(P ^ D) | 0xA50065 |
| 1 0 1 0 | D | 0xAA0029 |
| 1 0 1 1 | ~P \| D | 0xAF0229 |

1 1 0 0　P　　0xF00021 PATCOPY
1 1 0 1　P | ~D　　0xF50225
1 1 1 0　P | D　　0xFA0089
1 1 1 1　1　　0xFF0062 WHITENESS

请大家看，上面常用的几个都定义了，当然是用 #define 来定义的喽，当然，你认为你会常用的，你也可以加以定义，但从理解的角度，相信你如果学过汇编那一部分，或者对计算机的二进制原理比较熟悉，理解起来要省力得多。

有人也曾经问过牛刀，是先学习 MFC 还是先学习《数据结构》，我们学习的目的是什么？你的知识越雄厚，你学习的效率越高，你理解问题的能力越强，任何知识都不会白学的，所以，对于这些问题，牛刀给大家指条明路——那就是：

你学习的目的是什么？学习是你自己的事，别人的建议仅仅是一种建议，书还要你自己去一本一本地读，知识点还需要你一个一个地去掌握，这是别人谁也替代不了的，是不？

如果你的基础不扎实，那么你学到这估计会有点吃力，会碰到拦路虎，这是肯定的，但牛刀却不能因为就停下来为你狠补基础，看到这儿，相信大家应该明白，学习应该怎么学了吧？

有能力的人跟着牛刀继续：（说实话，牛刀不想说这句话，但这一段牛刀还就必须这么说，因为这一段的内容真的需要你自己去理解了，再说句大实话，下面的 BitBlt 如果你叫牛刀跟你详谈，那么牛刀可以又要和你从计算机的结构说起，那就说来话长了，何况，这些内容，在牛刀以前的教材中都有，大家如果有不明白的地方，那就需要去“温故而知新”喽，呵呵）

下面牛刀就与大家说说这个 BitBlt 这个其实是一个三元操作，这个我们也称之为“位元传输”或“位元块传输”，大家还记得我们上一讲与大家说，需要制作兼容 DC 吗？原因很简单，画刷是一个句柄，设备描述表是一个句柄，而我们的 HBITMAP 也是一个句柄，画刷可以选入当前设备，画笔可以选入当前设备，可就是位图不能直接选入当前设备，因为如果位图也能直接选入当前设备的话，那这儿就没有 BitBlt 什么事了。

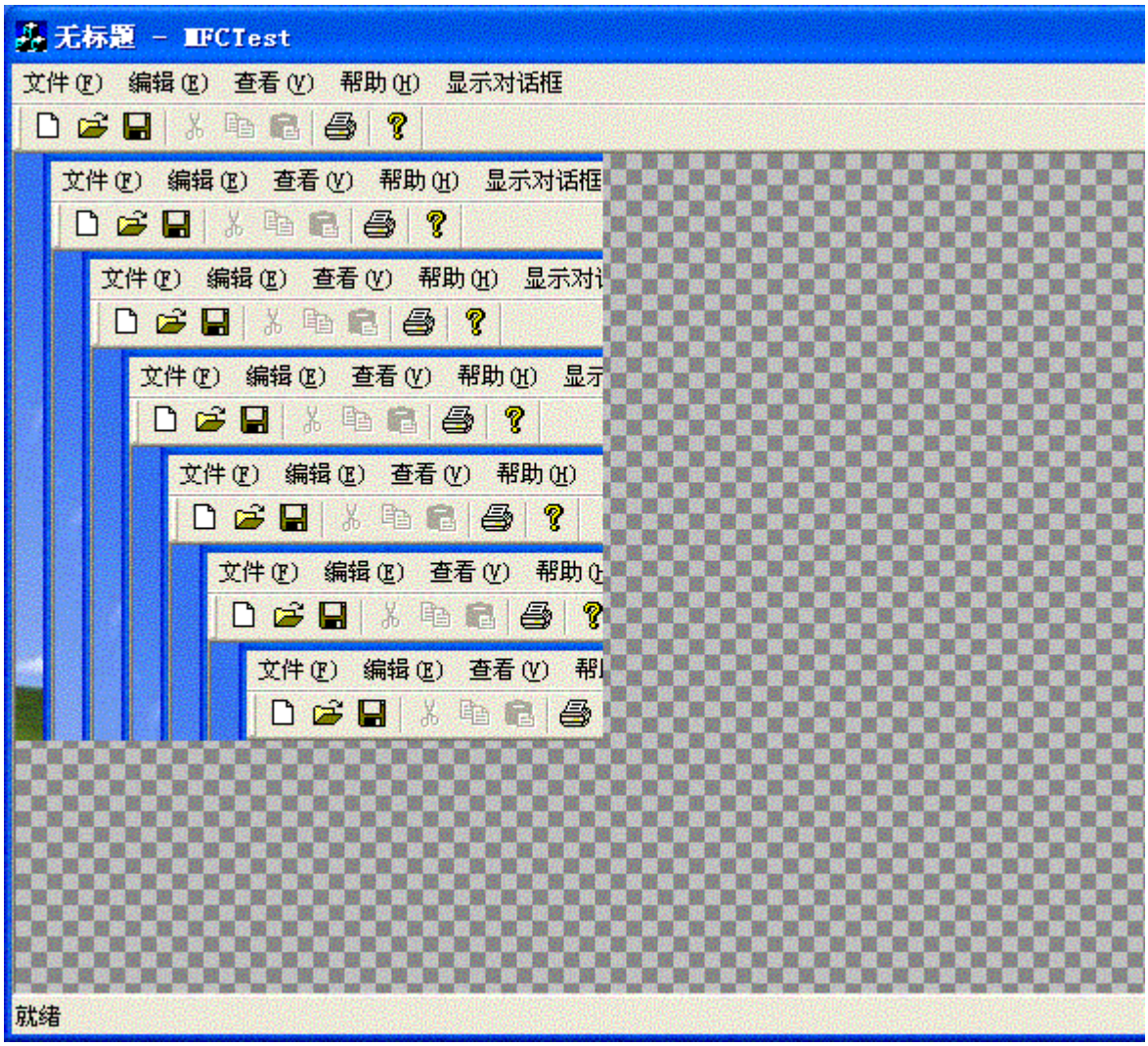
为什么这么说呢？大家可以想，我们画刷选入当前设备时，我们并没有用画刷去填充，我们画笔选入当前设备时，我们也没有用画笔去画图，当然，我们将位图选入当前设备，得到的结果也是同样的一片空白。

我们画刷可以用填充的办法表现出来，画笔可以用作图函数表现出来，然而却就是没有能够表现位图的函数，而位图的表现，就必须要 BitBlt ，而这个函数却必须要一个设备描述表，也就是一张纸，这张纸所以也只有从兼容 DC 中来，当然，还是那句话，如果你能够找到这张纸，那么即使不使用兼容 DC ，那么也可以用 BitBlt 去顺利地将图像传送过来，大家看下面的程序：（注：还记得前面我们为大家用打点法传过来的图像吗？这次我们不用那种办法了：

```
int CMFCTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
  return -1;
SetTimer(0,100,NULL);
return 0;
}
void CMFCTestView::OnTimer(UINT nIDEvent)
```

```
{
HDC hdc=::GetDC(NULL);
CPoint pt;
GetCursorPos(&pt);
HDC dc=::GetDC(m_hWnd);
::BitBlt(dc,0,0,300,300,hdc,pt.x,pt.y,SRCCOPY);
CView::OnTimer(nIDEvent);
}
```

这种感觉就很好了，大家自己将程序做进电脑，然后去比较比较：




上面的这种代码段很容易做成一个屏幕捕捉程序，很可惜的是，牛刀桌面上的那个小狮子还是捕捉不下来，呵呵，太怪怪的了。

同样，对于 BitBlt 来说，其中的前两个数据为某个点，后面两个为宽度高度，再后面就是源图像，后面是源图像的坐标，再后面就是图像传输模式。

当然，画刷，我们还称之为“图案（P）”，画到的地方还称之为“目的（D）”，而传进来的图像，我们称之为“来源（S）”，那么我们就可以用这三个数据来定义一个索引：

P S D 结果

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

再来与大伙讨论讨论这个数据吧，呵呵，假设我们不需要画刷，也就是 P 那一行，对后面的结果没有影响，我们还是黑色部分想要透明，白色部分不变，怎么样？牛刀详细过程就不解释了，大家看结果：

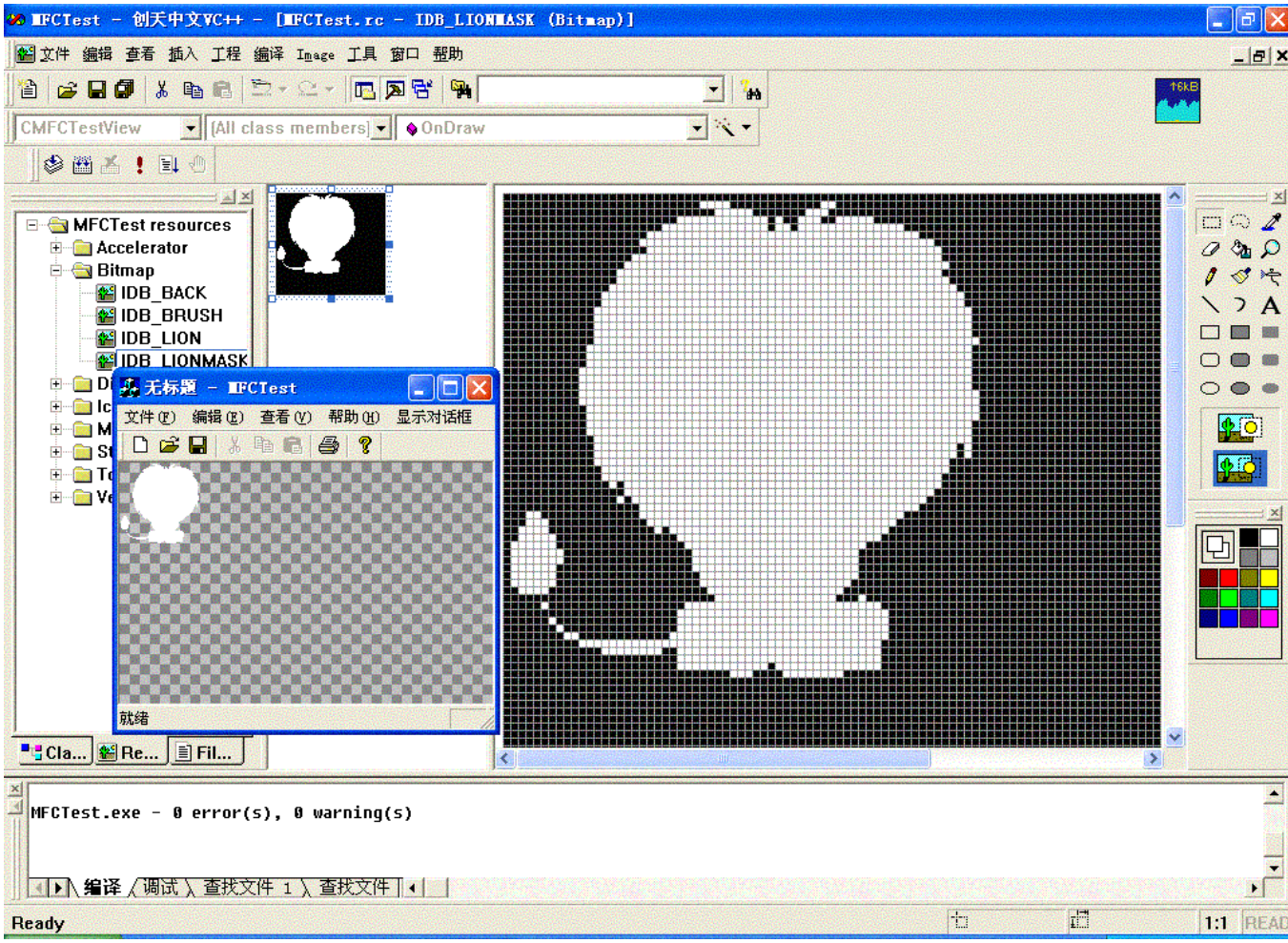
P S D 结果
0 0 0 0
0 0 1 1
0 1 0 1
0 1 1 1
1 0 0 0
1 0 1 1
1 1 0 1
1 1 1 1

对于黑色部分，我们想透明，也就是 S 那一行如果为 0 ，那么后面的结果值就与 D 部分一样，而白色部分，也就是 1 的部分不变， 1 还是 1 ，于是，从下到上，形成的结果为：11101110=0XEE，那么我们去找 0XEE 四索引，得到的值为： 0xee0086

于是，我们就可以用这个值去将黑色部分变成透明，而白色部分不变，请大家看：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
DrawBk(pDC);
MaskTest(pDC);
}
void CMFCTestView::MaskTest(CDC *pDC)
{
CBitmap bmp;
bmp.LoadBitmap(IDB_LIONMASK);
CDC dc;
dc.CreateCompatibleDC(pDC);
dc.SelectObject(&bmp);
pDC->BitBlt(0,0,300,300,&dc,0,0,0xee0086);
}
```

看效果：




怎么样？是不是有点想做一幅透明效果的图片的愿望了？下次吧，呵呵。

**微软积木〇八六：光栅技术**

什么是光栅，可能说来话长，本来是物理上的一个技术，后来我们的显示器上用上了光栅，从此就有了像素的概念，并且有一定的规定，来显示电视画面，所以逐渐地，人们也对于这个图片处理的技术，也称之为“光栅操作”，以上是牛刀的理解，具体是不是这样，牛刀也没有专门去查资料，什么是光栅技术，在这里也就是图片处理吧，或者就是 BitBlt 用的一种技术，既然说到 BitBlt ，我们就继续上一讲的话题，透明图片：

我们上一讲通过上面的操作我们在软件桌面上画了一个白色的小狮子，当然，我们也希望再用我们原来的图片，去套那个白色的小狮子，如果套的好的话，那么我们就实现了透明图片喽：

P S D 结果
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

我们需要背景是白色的部分，就画我们图片：
P S D 结果
0 0 0
0 0 1 0
0 1 0
0 1 1 1
1 0 0
1 0 1 0
1 1 0
1 1 1 1

那么我们黑色部分呢？那就黑色的呗：
P S D 结果
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 1
1 0 0 0
1 0 1 0
1 1 0 0
1 1 1 1

好，10001000=0x88，我们找到代码　0x8800c6

我们做程序：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        DrawBk(pDC);
        MaskTest(pDC);
        DrawLion(pDC);
}

void CMFCTestView::DrawLion(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_LION);
        CDC dc;
        dc.CreateCompatibleDC(pDC);
        dc.SelectObject(&bmp);

        pDC->BitBlt(0,0,300,300,&dc,0,0,0x8800c6);
}
```

说出来你可能不信，效果好得让人吃惊：

当然，牛刀贴出来的图已经经过处理的了，大家一定要自己去做一个，对了，牛刀告诉过大家，我们新建的图片一般都是十六色的，所以请大家如果真正地使用２５６色或者真色彩请最好通过导入的办法导入进来，相信大家做这种导入应该没有问题，如果有问题的话，发贴子问牛刀，呵呵，原来准备给大家贴张图的，后来一想，真的没必要，能看着牛刀的教程到这儿的，这点问题不是小菜一碟？

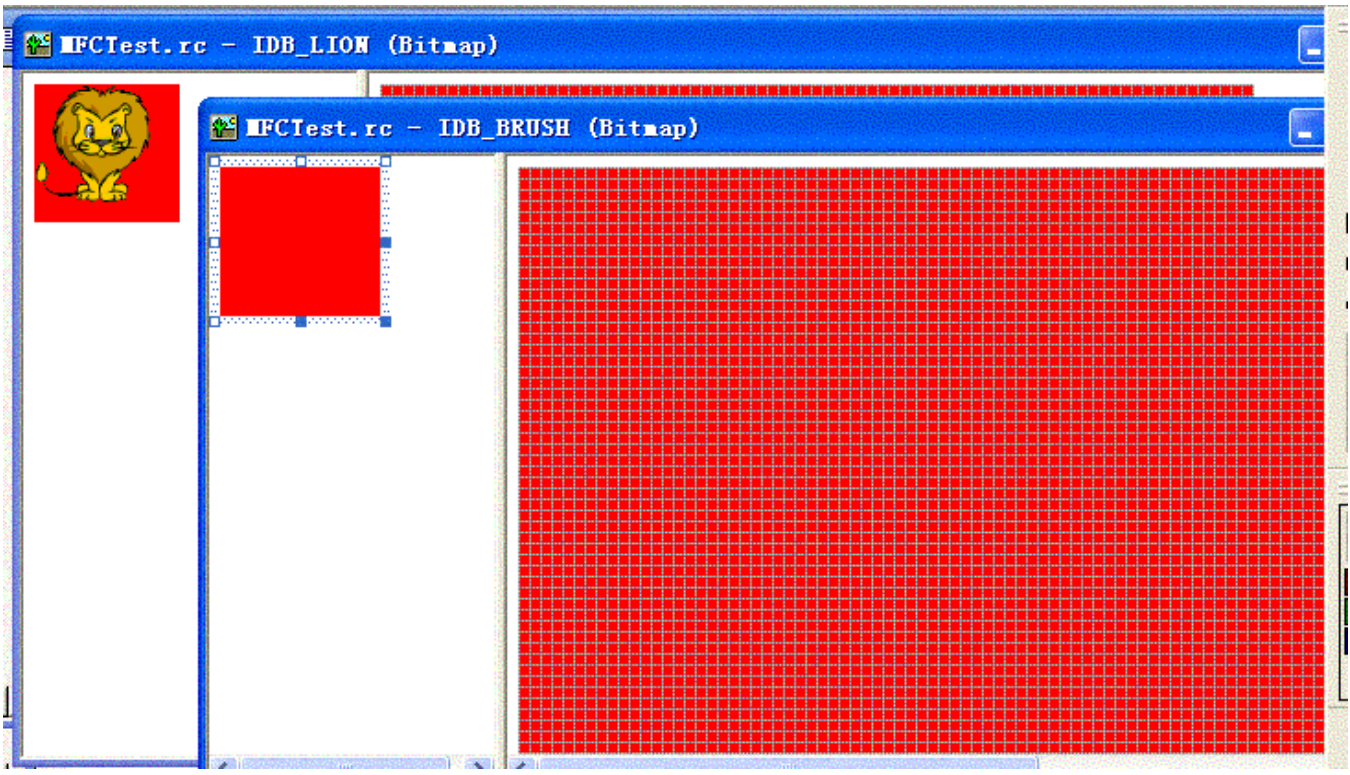
当然，大家可以在图片上的非图片区双击，可以出现图片的属性框，其中有个叫“C 颜”的，好像翻译的不行，其实应该是颜色数，或者翻译成“调色板”才好，当然，我们可以在后面选　256　或者其他，当然可以是单色，但不可以是真色

彩，所以大家导入真色彩位图后，不能使用 VC6.0 去编辑，这是一大遗憾，不过还好，我们有好多作图的软件，何况时间要不了多久，我们就可以开发出我们自己的图像处理软件来了，呵呵，一定的。

好了，我们上讲与大家使用的蒙板是我们的自己做的，那么我们很想利用程序去自动生成蒙板，那样我们不就可以省一块蒙板图片了吗？

好，我们就以上的思路，我们推倒我们前面的一切软件构思，我们重新去构思我们蒙板，也就是说，我们打算生成一块蒙板：

怎么做呢？我们打算将小狮子的背景需要做成相应背景蒙板的那一部分涂成红色，看图：




我们还是用那个小狮子，但这次我们将背景与小狮子的背景都涂成红色，我们可以去看一看小狮子的图片大小：宽度高度全为 76，呵呵，我们先在屏幕的背景上贴上一个方块：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        DrawBk(pDC);
        DrawLion(pDC);
}
void CMFCTestView::DrawLion(CDC *pDC)
{
```

```
        DrawRect(pDC);
}

void CMFCTestView::DrawRect(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_BRUSH);
        CBrush brush;
        brush.CreatePatternBrush(&bmp);
        CRect rect(0,0,76*2,76);
        pDC->FillRect(&rect,&brush);
}
```

请大家注意，我们做了一行两个方块，我们作一些对比用。下面，我们先在第一块图的地方，贴上一块图：

```
void CMFCTestView::DrawLion(CDC *pDC)
{
        DrawRect(pDC);
        DrawTest(pDC);
}
```

我们就是与大家玩一玩的，所以名子就叫 DrawTest 吧，呵呵：

```
void CMFCTestView::DrawTest(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_LION);
        CDC dc;
        dc.CreateCompatibleDC(pDC);
        dc.SelectObject(&bmp);

        pDC->BitBlt(0,0,76,76,&dc,0,0,SRCCOPY);
}
```

当然，我们这一次贴上去的图是什么样就什么样，一点都不变，那么下面我们需要用将我们红色的部分做成蒙板，也就是说，红色的部分现在要变成白色，有这种特性的位运算就是“异或”，如果大家还记得牛刀以前的教程的话，大家一定知道，异或就是相同颜色则变为 1 ，否则变为 0

哎，如果你不知道，我们就可以不管那些，我们直接在我们下面的模式设置中去设置就行了：

P S D 结果
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1

1 1 0
1 1 1

当然，我们不一定需要用异或，我们可以这样看，上面的刷子我们用不着，所以第一列，我们管不着，而后面的两个，如果我们一样的想变成黑色，不一样的变成白色，怎么弄呢？先试着玩玩吧：（大家要勤于思考哟）

P S D 结果
0 0 0 0
0 0 1
0 1 0
0 1 1 0
1 0 0 0
1 0 1
1 1 0
1 1 1 0

好了，通过上面的来一调色，我们的相同部分就可以变成黑色了，而不同部分呢？变成白色吧：

P S D 结果
0 0 0 0
0 0 1 1
0 1 0 1
0 1 1 0
1 0 0 0
1 0 1 1
1 1 0 1
1 1 1 0

好了，我们连着在第二块方块上用上面的 01100110=0x66 所表示的模式：0x660046

……

```
pDC->BitBlt(76,0,76,76,&dc,0,0,0x660046);
```

背景被我们成功地变成了黑色，就是 0 啊，呵呵，可惜的是，我们不能将我们不需要变成黑色的部分变成白色，于是，我们就只能用一个判断来做这件事喽：

```
for(int i=0;i<76;i++)
{
        for(int j=0;j<76;j++)
        {
                if(pDC->GetPixel(i+76,j))
                {
                        pDC->SetPixel(i+76*2,j,RGB(255,255,25
5));
                }
                else
                {
                        pDC->SetPixel(i+76*2,j,RGB(0,0,0));
```

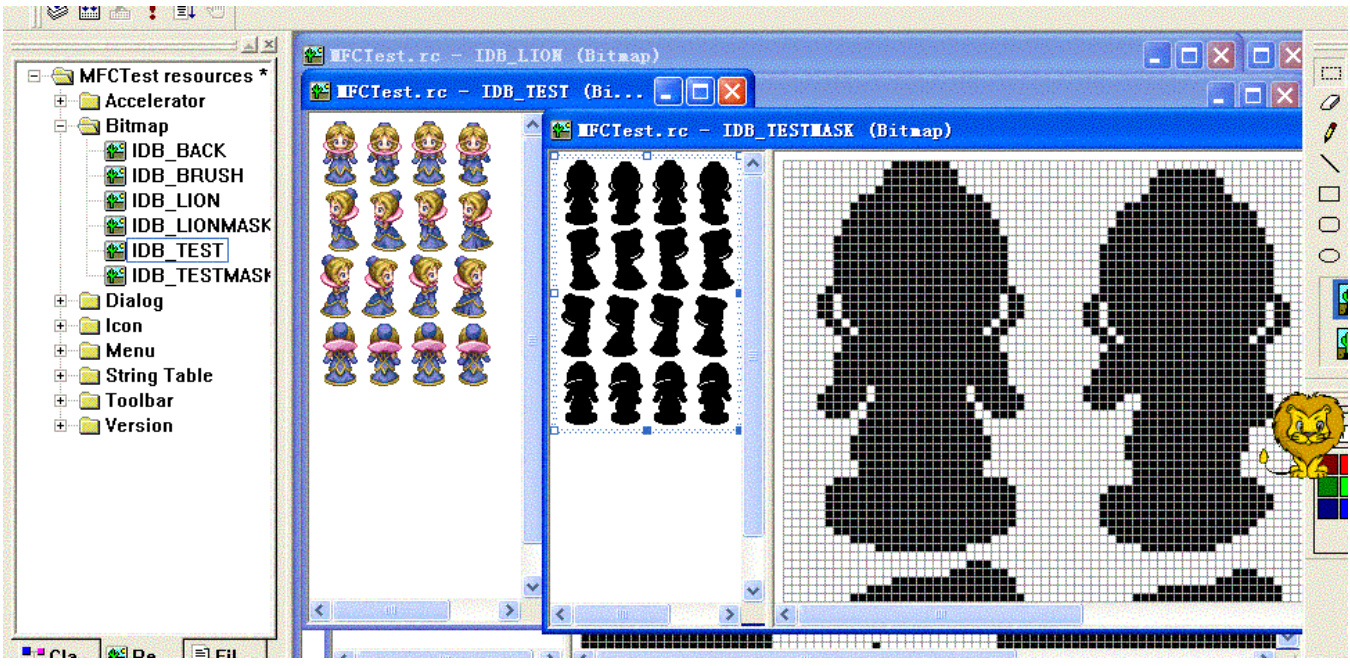
```
                }
            }
        }
```

当然喽，你可能会说，这不是很慢吗？，当然，你与显存拍电报肯定慢，但我们这讲的是道理，你以后真正做程序，一般这种过程都是在兼容 CD 中完成的，那就相当地快，当然，你也可以直接将上面的程序改为：

```
        for(int i=0;i<76;i++)
        {
                for(int j=0;j<76;j++)
                {
                        if(pDC->GetPixel(i+76,j))
                        {
                                pDC->SetPixel(i+76*2,j,pDC->GetPixel(
i,j));
                        }
                }
        }
```

就可以直接实现透明，效果很好，但请大家注意的，还是这种东西不能直接在屏幕上画，当然，你处理一幅图片这样做很简单，但如果你做动画，这样做速度就会相当地慢了，所以我们一般做程序时，我们可能会申请很多很多的这种画布空间，并且做成我们想要的东东，这样就可以在适当的时候直接用 BitBlt 调进显存去显示出我们自己的图片了，做成动画也很简单。

当然做动画还有一种技术，就是将所有的图片都放到一幅图上，要用时，用同一程序直接调用，即可以用 BitBlt 去每次调用不同的位置就可以形成动画了，看下面的图片：



我们做一个测试图片与测试蒙板，用 IDB_TEST 与 IDB_TESTMASK 表示两幅图，那我们做下面的程序：

```
void CMFCTestView::OnTimer(UINT nIDEvent)
{
        static int i=0;
        CClientDC dc(this);
        DrawBk(&dc);
        DrawTest(&dc,i);
        i=(i+1)%16;
        CView::OnTimer(nIDEvent);
}
void CMFCTestView::DrawTest(CDC *pDC,int i)
{
        DrawTestMask(pDC,i);
        DrawTestPic(pDC,i);
}

void CMFCTestView::DrawTestMask(CDC *pDC,int i)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TESTMASK);
        CDC testdc;
        testdc.CreateCompatibleDC(pDC);
        testdc.SelectObject(&bmp);

        pDC->BitBlt(0,0,32,48,&testdc,(i/4)*32,(i%4)*48,0xBB0226);
}

void CMFCTestView::DrawTestPic(CDC *pDC,int i)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TEST);
        CDC testdc;
        testdc.CreateCompatibleDC(pDC);
        testdc.SelectObject(&bmp);

        pDC->BitBlt(0,0,32,48,&testdc,(i/4)*32,(i%4)*48,0x8800c6);
}
```

上面这个小动画就不能以图片的形式给大家了，做个软件给大家吧，大家下载玩玩，当然，你只要有图片，也可以很容易做的哟。

当然，可能上面这种过程你觉得应该可以做成一个函数，有图片，有蒙板，就一个函数就可以实现透明效果了，有没有这样的函数呢？呵呵，牛刀还真的不知道，如果有谁知道告诉牛刀。（牛刀在这方面的研究的确不深啊，呵呵。）

像上面那种传输毕竟是一比一的传输，可能真的很不过瘾，于是，又有一种传输形式：

```
void CMFCTestView::DrawTestMask(CDC *pDC,int i)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TESTMASK);
        CDC testdc;
        testdc.CreateCompatibleDC(pDC);
        testdc.SelectObject(&bmp);

        CRect rect;
        GetClientRect(&rect);
        pDC->StretchBlt(0,0,rect.Width(),rect.Height(),&testdc,(i/4)*
32,(i%4)*48,32,48,0xbb0226);
}

void CMFCTestView::DrawTestPic(CDC *pDC,int i)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TEST);
        CDC testdc;
        testdc.CreateCompatibleDC(pDC);
        testdc.SelectObject(&bmp);

        CRect rect;
        GetClientRect(&rect);
        pDC->StretchBlt(0,0,rect.Width(),rect.Height(),&testdc,(i/4)*
32,(i%4)*48,32,48,0x8800c6);

}
```

就是上面的 StretchBlt 比 BitBlt 多了两个参数，就是源代码的地方多一个宽度与一个高度，我们可以将源图片的宽度与高度伸缩成目的图片的高度与宽度，从而实现画面的放缩，其他的解释与 BitBlt 一样，所以上面的代码做出来的软件大家可以去将窗口调节调节，看看效果，效果不是太好，呵呵。

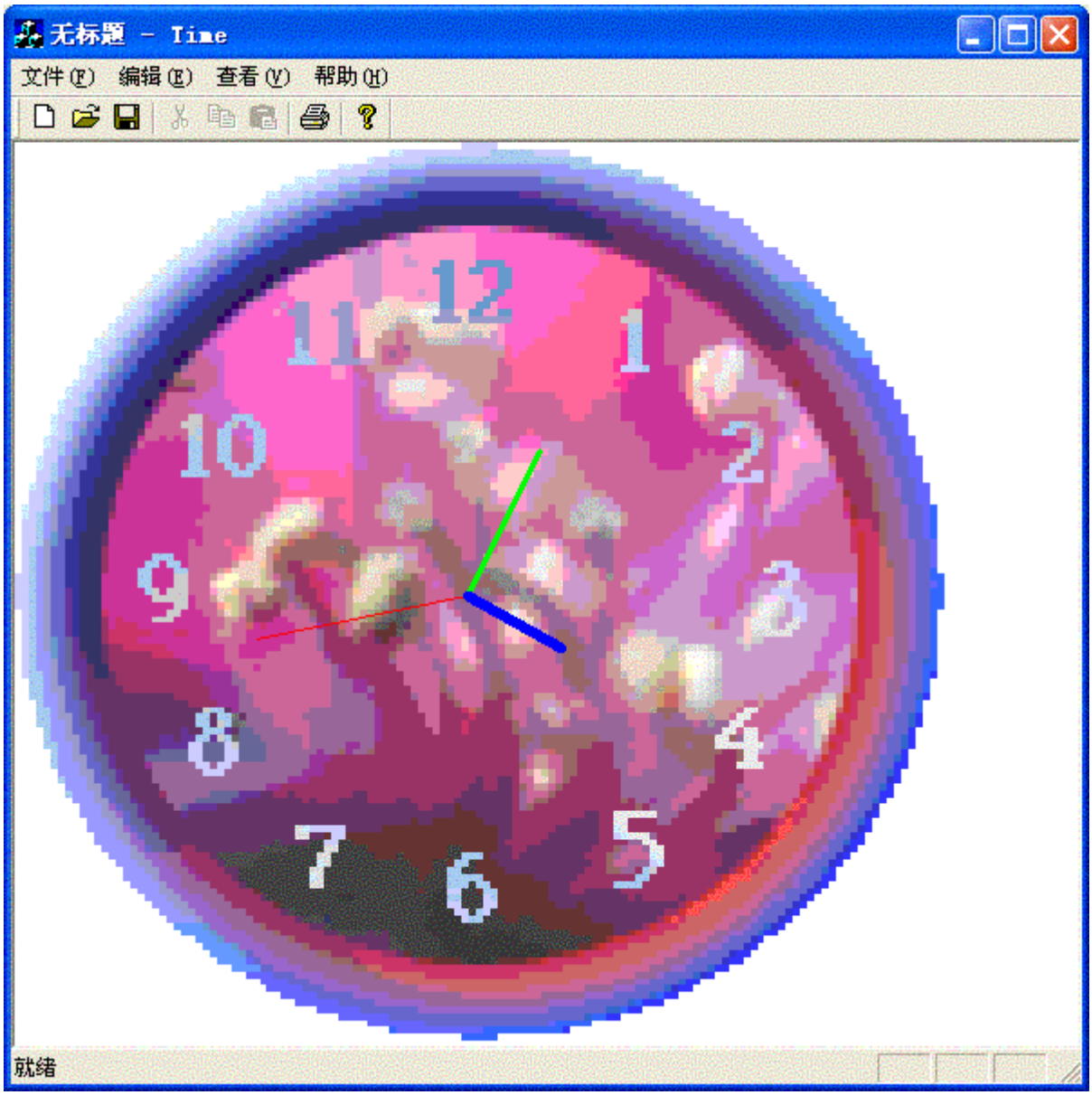
值得提起的是， StretchBlt 可以通过将一些长度宽度变成负的，从而实现图片的翻转，这一点很是实用。

大家一定会想，那么那个表呢？我们会在文件中一并发给大家，希望大家多去玩玩，能够弄明白，一些常用的技术，可以自己做成函数，或者干脆做成一个类，想要用的时候，就直接将函数或者类插到我们当前的程序中来就一切 OK 了，很是方便，当然这种制作函数或者类的办法，我们在以后的教程中慢慢与大家去品谈。

那么我们还回到我们的老问题上来吧，我们想做一款时钟，那么使用贴图的办法去制作，那更是小菜一碟哟，看程序：

```
void CTimeView::DrawFace(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TIME);
        CDC timDC;
        timDC.CreateCompatibleDC(pDC);
        timDC.SelectObject(&bmp);

        pDC->StretchBlt(0,0,550,500,&timDC,0,0,138,130,SRCCOPY);
}
```

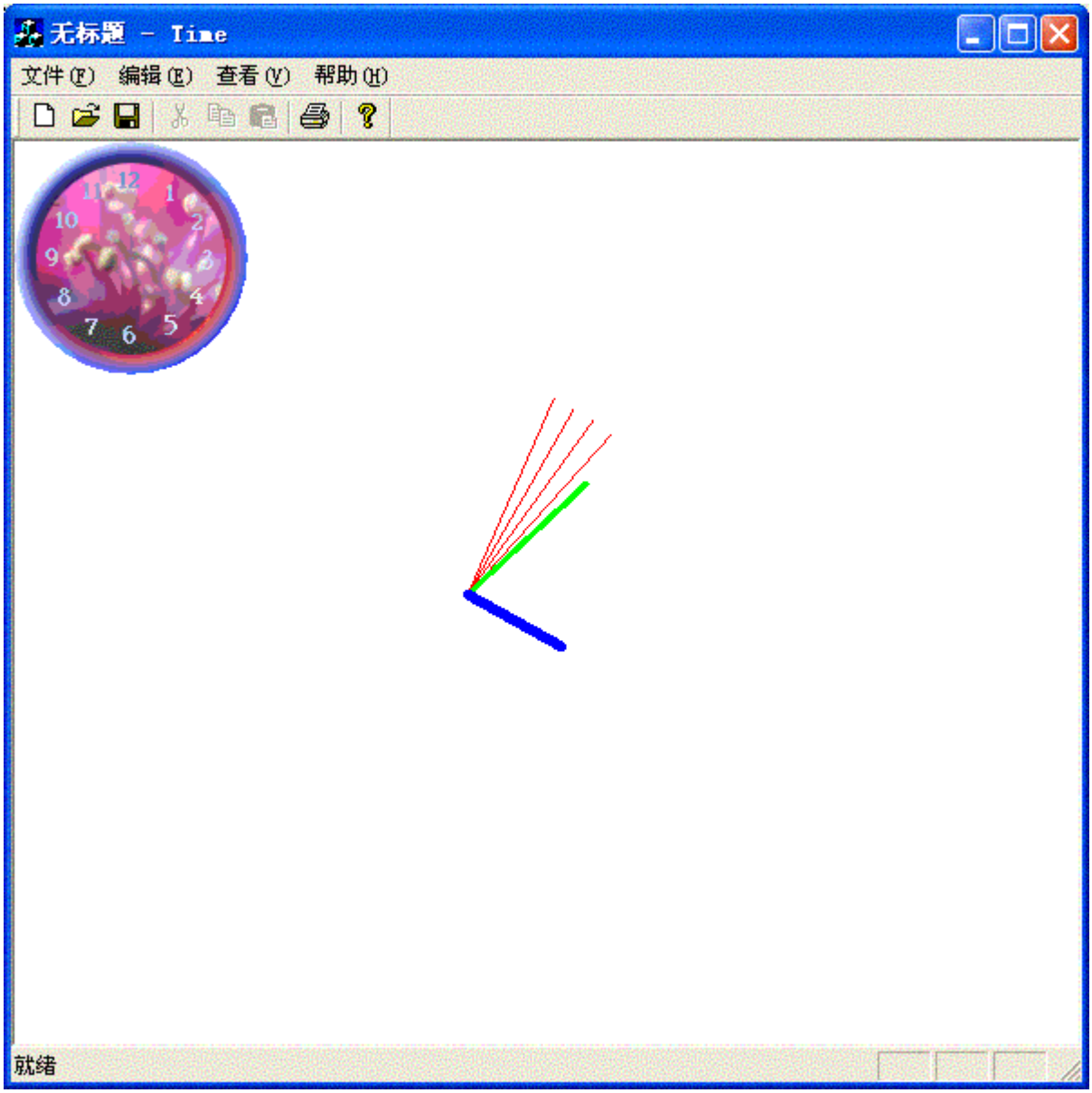
效果不用说了，用 StertchBlt 拉伸过的图简直是惨不忍睹：



当然，如果我们不使用这种拉伸技术的话，大家看：

```
void CTimeView::DrawFace(CDC *pDC)
{
        CBitmap bmp;
        bmp.LoadBitmap(IDB_TIME);
        CDC timDC;
        timDC.CreateCompatibleDC(pDC);
        timDC.SelectObject(&bmp);

        pDC->BitBlt(0,0,138,138,&timDC,0,0,SRCCOPY);
}
```

这种感觉好是好喽，可是又有问题喽：



大家看，我们的指针是在外面显示，假如我们不用程序，而可以在软件中自由地调节时钟圆心，时针、分针与秒针的长度那该多好啊!!!

当然，想要达到这个目的，请大家继续牛刀的下一讲——对话框技术

知识点回顾：

透明图片效果
先制作蒙板，在背景上蒙上一层白色或者黑色
再将图片限制在蒙板里面进行位块传输或者拉伸传输

技术要点，必须学会使用模式设计的办法去设计我们的模式，然后查表得到我们所需要的模式代码，然后在位块传输中使用，以达到我们的目的。

作业：（呵呵，MFC 部分，牛刀还没布置过作业，可能这一章与上一章的好多东西，如果有朋友基础不扎实的话，可能看不来，这一讲我们布置一个作业啊，呵呵）

板画效果是照像业中的一种技术，需要得到两张底片，一张正片，一张负片，然后将两张底片叠在一起，并且根据需要错开一点，得到合适的错开位置后，底片上好多地方原来颜色相近的地方就变成黑色，分界区域当然不是黑色，于是可以形成负片的板画效果，然后用照像纸对其进行长时间爆光，洗出照片来就是相当美妙的板画效果了。

根据以上原理，做一款软件，不允许使用 SetPixel 等打点函数，做一款软件，将图片转换为板画效果。

题目有难度，牛刀将不公布此题的答案，大家自己去做软件试着玩玩，就会知道 PhotoShop 中的滤镜是怎么回事了。

**三元光栅操作代码**

| 布尔功能 | 光栅操作 | 常用名称 |
|---|---|---|
| 00 | 00000042 | BLACKNESS |
| 01 | 00010289 | - |
| 02 | 00020C89 | - |
| 03 | 000300AA | - |
| 04 | 00040C88 | - |
| 05 | 000500A9 | - |
| 06 | 00060865 | - |
| 07 | 000702C5 | - |
| 08 | 00080F08 | - |

| | | |
|---|---|---|
| 09 | 00090245 | - |
| 0A | 000A0329 | - |
| 0B | 000B0B2A | - |
| 0C | 000C0324 | - |
| 0D | 000D0B25 | - |
| 0E | 000E08A5 | - |
| 0F | 000F0001 | - |
| 10 | 00100C85 | - |
| 11 | 001100A6 | NOTSRCERASE |
| 12 | 00120868 | - |
| 13 | 001302C8 | - |
| 14 | 00140869 | - |
| 15 | 001502C9 | - |
| 16 | 00165CCA | - |
| 17 | 00171D54 | - |
| 18 | 00180D59 | - |
| 19 | 00191CC8 | - |
| 1A | 001A06C5 | - |
| 1B | 001B0768 | - |
| 1C | 001C06CA | - |
| 1D | 001D0766 | - |
| 1E | 001E01A5 | - |
| 1F | 001F0385 | - |
| 20 | 00200F09 | - |
| 21 | 00210248 | - |
| 22 | 00220326 | - |
| 23 | 00230B24 | - |
| 24 | 00240D55 | - |
| 25 | 00251CC5 | - |
| 26 | 002606C8 | - |
| 27 | 00271868 | - |
| 28 | 00280369 | - |
| 29 | 002916CA | - |
| 2A | 002A0CC9 | - |
| 2B | 002B1D58 | - |
| 2C | 002C0784 | - |
| 2D | 002D060A | - |
| 2E | 002E064A | - |
| 2F | 002F0E2A | - |
| 30 | 0030032A | - |
| 31 | 00310B28 | - |
| 32 | 00320688 | - |
| 33 | 00330008 | NOTSRCCOPY |
| 34 | 003406C4 | - |

| | | |
|---|---|---|
| 35 | 00351864 | - |
| 36 | 003601A8 | - |
| 37 | 00370388 | - |
| 38 | 0038078A | - |
| 39 | 00390604 | - |
| 3A | 003A0644 | - |
| 3B | 003B0E24 | - |
| 3C | 003C004A | - |
| 3D | 003D18A4 | - |
| 3E | 003E1B24 | - |
| 3F | 003F00EA | - |
| 40 | 00400F0A | - |
| 41 | 00410249 | - |
| 42 | 00420D5D | - |
| 43 | 00431CC4 | - |
| 44 | 00440328 | SRCERASE |
| 45 | 00450B29 | - |
| 46 | 004606C6 | - |
| 47 | 0047076A | - |
| 48 | 00480368 | - |
| 49 | 004916C5 | - |
| 4A | 004A0789 | - |
| 4B | 004B0605 | - |
| 4C | 004C0CC8 | - |
| 4D | 004D1954 | - |
| 4E | 004E0645 | - |
| 4F | 004F0E25 | - |
| 50 | 00500325 | - |
| 51 | 00510B26 | - |
| 52 | 005206C9 | - |
| 53 | 00530764 | - |
| 54 | 005408A9 | - |
| 55 | 00550009 | DSTINVERT |
| 56 | 005601A9 | - |
| 57 | 00570389 | - |
| 58 | 00580785 | - |
| 59 | 00590609 | - |
| 5A | 005A0049 | PATINVERT |
| 5B | 005B18A9 | - |
| 5C | 005C0649 | - |
| 5D | 005D0E29 | - |
| 5E | 005E1B29 | - |
| 5F | 005F00E9 | - |
| 60 | 00600365 | - |

| | | |
|---|---|---|
| 61 | 006116C6 | - |
| 62 | 00620786 | - |
| 63 | 00630608 | - |
| 64 | 00640788 | - |
| 65 | 00650606 | - |
| 66 | 00660046 | SRCINVERT |
| 67 | 006718A8 | - |
| 68 | 006858A6 | - |
| 69 | 00690145 | - |
| 6A | 006A01E9 | - |
| 6B | 006B178A | - |
| 6C | 006C01E8 | - |
| 6D | 006D1785 | - |
| 6E | 006E1E28 | - |
| 6F | 006F0C65 | - |
| 70 | 00700CC5 | - |
| 71 | 00711D5C | - |
| 72 | 00720648 | - |
| 73 | 00730E28 | - |
| 74 | 00740646 | - |
| 75 | 00750E26 | - |
| 76 | 00761B28 | - |
| 77 | 007700E6 | - |
| 78 | 007801E5 | - |
| 79 | 00791786 | - |
| 7A | 007A1E29 | - |
| 7B | 007B0C68 | - |
| 7C | 007C1E24 | - |
| 7D | 007D0C69 | - |
| 7E | 007E0955 | - |
| 7F | 007F03C9 | - |
| 80 | 008003E9 | - |
| 81 | 00810975 | - |
| 82 | 00820C49 | - |
| 83 | 00831E04 | - |
| 84 | 00840C48 | - |
| 85 | 00851E05 | - |
| 86 | 008617A6 | - |
| 87 | 008701C5 | - |
| 88 | 008800C6 | SRCAND |
| 89 | 00891B08 | - |
| 8A | 008A0E06 | - |
| 8B | 008B0666 | - |
| 8C | 008C0E08 | - |

| | | |
|---|---|---|
| 8D | 008D0668 | - |
| 8E | 008E1D7C | - |
| 8F | 008F0CE5 | - |
| 90 | 00900C45 | - |
| 91 | 00911E08 | - |
| 92 | 009217A9 | - |
| 93 | 009301C4 | - |
| 94 | 009417AA | - |
| 95 | 009501C9 | - |
| 96 | 00960169 | - |
| 97 | 0097588A | - |
| 98 | 00981888 | - |
| 99 | 00990066 | - |
| 9A | 009A0709 | - |
| 9B | 009B07A8 | - |
| 9C | 009C0704 | - |
| 9D | 009D07A6 | - |
| 9E | 009E16E6 | - |
| 9F | 009F0345 | - |
| A0 | 00A000C9 | - |
| A1 | 00A11B05 | - |
| A2 | 00A20E09 | - |
| A3 | 00A30669 | - |
| A4 | 00A41885 | - |
| A5 | 00A50065 | - |
| A6 | 00A60706 | - |
| A7 | 00A707A5 | - |
| A8 | 00A803A9 | - |
| A9 | 00A90189 | - |
| AA | 00AA0029 | - |
| AB | 00AB0889 | - |
| AC | 00AC0744 | - |
| AD | 00AD06E9 | - |
| AE | 00AE0B06 | - |
| AF | 00AF0229 | - |
| B0 | 00B00E05 | - |
| B1 | 00B10665 | - |
| B2 | 00B21974 | - |
| B3 | 00B30CE8 | - |
| B4 | 00B4070A | - |
| B5 | 00B507A9 | - |
| B6 | 00B616E9 | - |
| B7 | 00B70348 | - |
| B8 | 00B8074A | - |

| | | |
|---|---|---|
| B9 | 00B906E6 | - |
| BA | 00BA0B09 | - |
| BB | 00BB0226 | MERGEPAINT |
| BC | 00BC1CE4 | - |
| BD | 00BD0D7D | - |
| BE | 00BE0269 | - |
| BF | 00BF08C9 | - |
| C0 | 00C000CA | MERGECOPY |
| C1 | 00C11B04 | - |
| C2 | 00C21884 | - |
| C3 | 00C3006A | - |
| C4 | 00C40E04 | - |
| C5 | 00C50664 | - |
| C6 | 00C60708 | - |
| C7 | 00C707AA | - |
| C8 | 00C803A8 | - |
| C9 | 00C90184 | - |
| CA | 00CA0749 | - |
| CB | 00CB06E4 | - |
| CC | 00CC0020 | SRCCOPY |
| CD | 00CD0888 | - |
| CE | 00CE0B08 | - |
| CF | 00CF0224 | - |
| D0 | 00D00E0A | - |
| D1 | 00D1066A | - |
| D2 | 00D20705 | - |
| D3 | 00D307A4 | - |
| D4 | 00D41D78 | - |
| D5 | 00D50CE9 | - |
| D6 | 00D616EA | - |
| D7 | 00D70349 | - |
| D8 | 00D80745 | - |
| D9 | 00D906E8 | - |
| DA | 00DA1CE9 | - |
| DB | 00DB0D75 | - |
| DC | 00DC0B04 | - |
| DD | 00DD0228 | - |
| DE | 00DE0268 | - |
| DF | 00DF08C8 | - |
| E0 | 00E003A5 | - |
| E1 | 00E10185 | - |
| E2 | 00E20746 | - |
| E3 | 00E306EA | - |
| E4 | 00E40748 | - |

| | | |
|---|---|---|
| E5 | 00E506E5 | - |
| E6 | 00E61CE8 | - |
| E7 | 00E70D79 | - |
| E8 | 00E81D74 | - |
| E9 | 00E95CE6 | - |
| EA | 00EA02E9 | - |
| EB | 00EB0849 | - |
| EC | 00EC02E8 | - |
| ED | 00ED0848 | - |
| EE | 00EE0086 | SRCPAINT |
| EF | 00EF0A08 | - |
| F0 | 00F00021 | PATCOPY |
| F1 | 00F10885 | - |
| F2 | 00F20B05 | - |
| F3 | 00F3022A | - |
| F4 | 00F40B0A | - |
| F5 | 00F50225 | - |
| F6 | 00F60265 | - |
| F7 | 00F708C5 | - |
| F8 | 00F802E5 | - |
| F9 | 00F90845 | - |
| FA | 00FA0089 | - |
| FB | 00FB0A09 | PATPAINT |
| FC | 00FC008A | - |
| FD | 00FD0A0A | - |
| FE | 00FE02A9 | - |
| FF | 00FF0062 | WHITENESS |

**微软积木〇八七：人机对话**

可能牛刀以上几章与大伙讨论 BitBlt 等技术时，要求有些近乎苛刻了，当然，牛刀可以与其他教材上一样，给大家一个 SRCCOPY ，举几个示例就完事，那根本不需要你掌握模式设置方面的内容，当然好多教材上也是这么写的，不过如果是那样给大伙实践的话，那牛刀教程的闪光点也就没了，大家说是不？上几讲的内容牛刀试图讲得讲大伙都能够明白，后来写得过程发觉越写越不太现实，因为位操作技术实在太抽象，没有一定的基本功，就是写得再详细，可能还是会有人明白不了当然那些技术可是图片处理的基本技能哟，要做得一款好游戏，一款好软件，那么那些技术如果不使用，那么我们的软件就会失去好多生动的元素。

当然我们这一讲开始，接下来就要讲人机对话方面的内容了，如果大家前面的还没懂也没关系，一是一个一个程序都要上机，模式代码建立过程要一个一个地上手，第二就是需要明白一些基本的原理，如果大家还不能懂的话，发贴子给我，我在以后的课程中开专讲给大家从 BMP 图片的结构、组成、显示等一基本的原理与大家讲起，其实 MFC 最重要的就是封装，但很可惜的是，好多的知识点也被无情地“封装”起来了。为了不至于使本教程出现跳跃感，我们只有再次将这些理论知识往后放一放了，可要知道那些知识本身就可以写出一本书。

好了，上一讲最后留下了一个问题，如果我们要在软件生成之后，能够去改变一些东东怎么办，我们的最佳的手段就是——对话框。

相信大家对对话框一定不会陌生，假如你很陌生的话，你看牛刀的教材也不可能一直尾随着 8 0 多期，是不？呵呵。

对话框有两种，大家可以选择一款软件的“文件/打开...”，就可以弹出一个对话框，请大家注意，这种对话框在我们没关闭之前，我们的程序通常是用不了，不响应的，这种对话框，我们称之为“模式对话框”，打开对话框就是一种模式对话框。

还有一种对话框，如查找对话框，大家可以按“编辑/查找...”打开的对话框基本不影响我们对程序的编写，这种对话框，我们称之为“非模式对话框”。

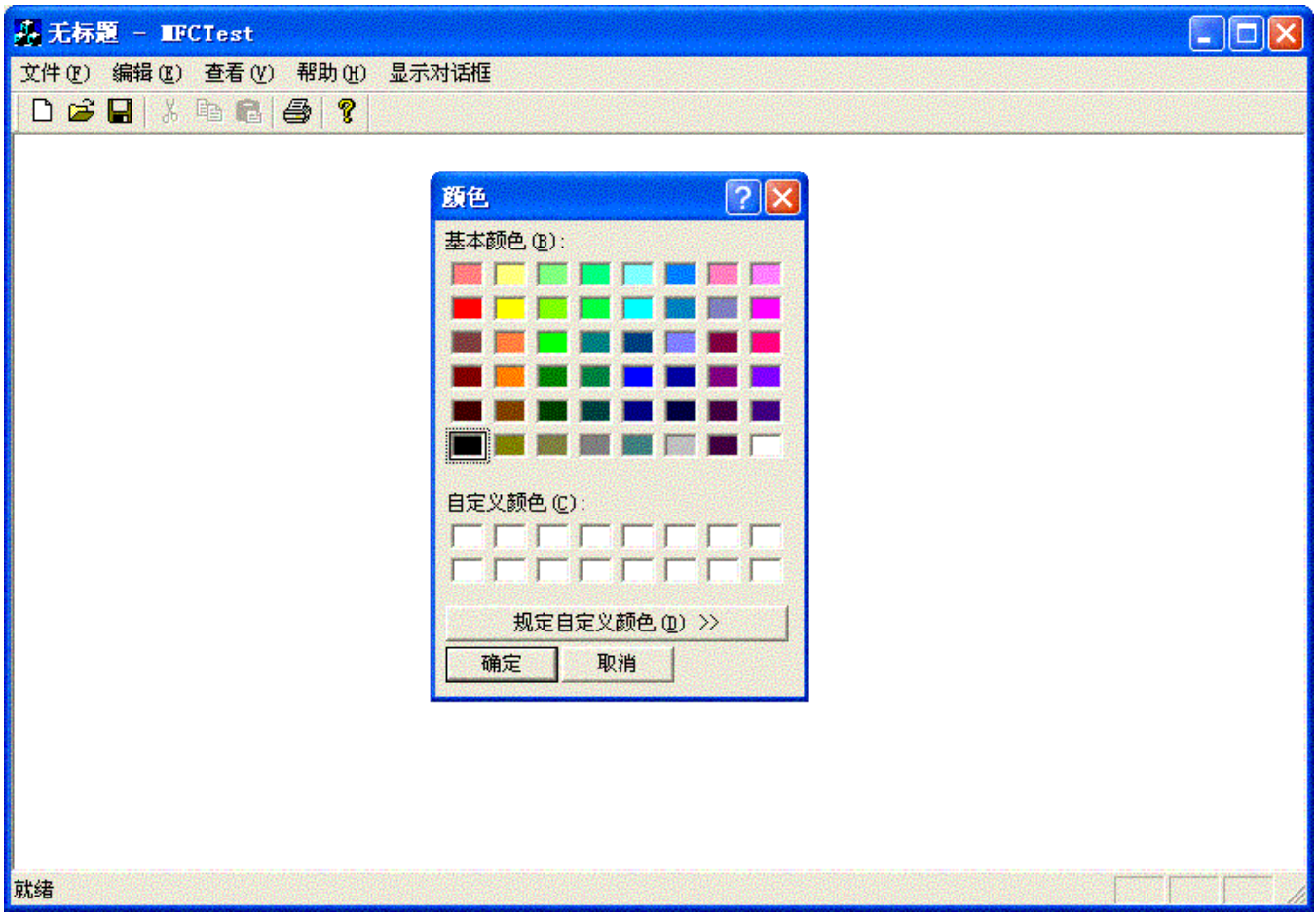
当然，模式对话框也分“通用对话框”与“非通用对话框”，通用对话框其实就是系统中固定的一些对话框，像颜色对话框啊，字体对话框啊，打开对话框啊，等等。

举一个例子吧，颜色对话框的类为： CColorDialog 是一个类，几乎是定义一下就可以用，例如，我们可以到类中去响应鼠标左击操作：

```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CColorDialog dlg;
        dlg.DoModal();
```

```
        CView::OnLButtonDown(nFlags, point);
}
```

不错，定义一下，然后一个 DoMoDal 就可以定义出这个对话框了：



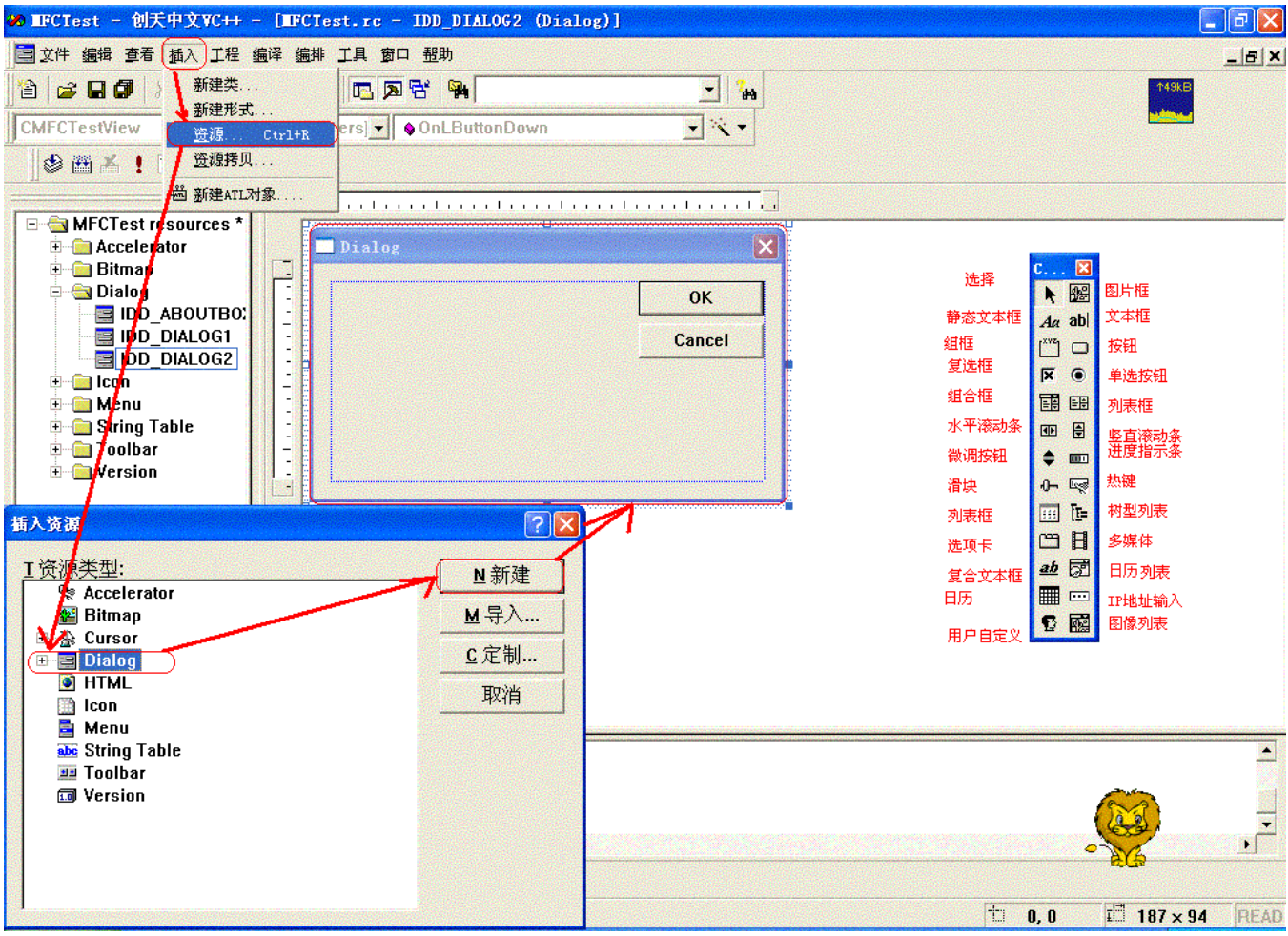

虽然没什么大用，但至少告诉我们，想整出一个对话框，那一点都不费劲。只需要一个 DoModal ，一直到这个对话框退出之前，我们都不需要对其编任何的程序，这一切都是类中去给我们做了。

也正因为如此，大家知道，我们在出现对话框时，只要这个对话框不退出，那么 DoModal 这个函数就不返回，也就是说，我们的所有的消息都不能够投到我们的程序中了，而只能投入到对话框上，所以说，只要对话框在，那么我们的 DoModal 这个函数就停在那儿不往下走，似乎我们的程序一下子就变成了单线程的了，也正因为如此，对于这种模式对话框，其生命周期也就是对话框存在时的周期，所以这个对话框在哪儿定义都是一个样，也正因为如此，这种模式对话框，我们通常定义成函数中的一个非静态变量。

是不是有些太抽象？没关系，我们先做一个对话框，再与大家详细讨论有关操作：

对话框一是一种资源，可我们以用“插入/资源...”的办法插入，大家可以看到对话框前面有一个加号，表示下面有很多种，大家不需要点开加号，直接在对话

框（Dialog）上面双击就行了，插入一个通用的：



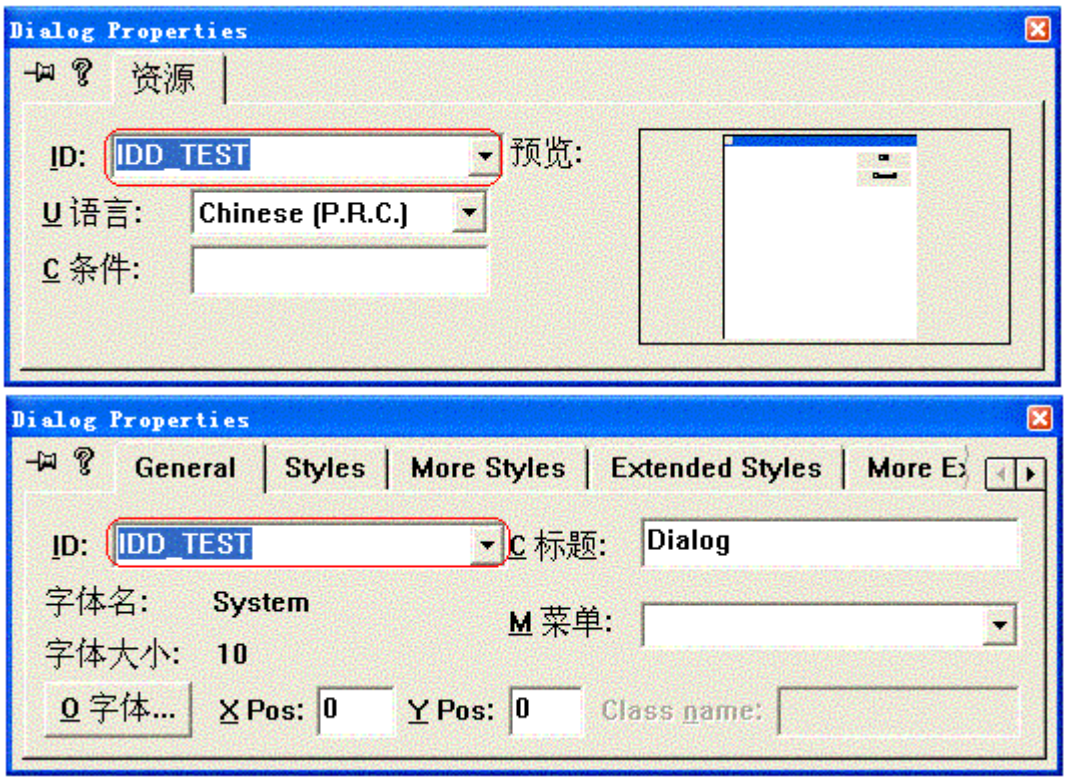

说实话，感觉没有 VB 舒服，使用上也没有 VB 来得方便，但如果你真的会使用了，功能一定比 VB 要强大。

如果你已经会去使用 VB 了，并且做一些不大的，要求不高的软件，那牛刀建议大家还是去使用 VB ，这一点上来说，VC 与 VB 是不可同日而语的。毕竟 VB 中的后台封装太多了，我相信 Delphi 在这方面做得也肯定比 VC 要好，这是一定的。

牛刀并不是一味去说 VC 的好话的，好就是好，不好就是不好，但 VC 毕竟是 VC ，与 VB 、Delphi 根本就不是一个层次上的东西。

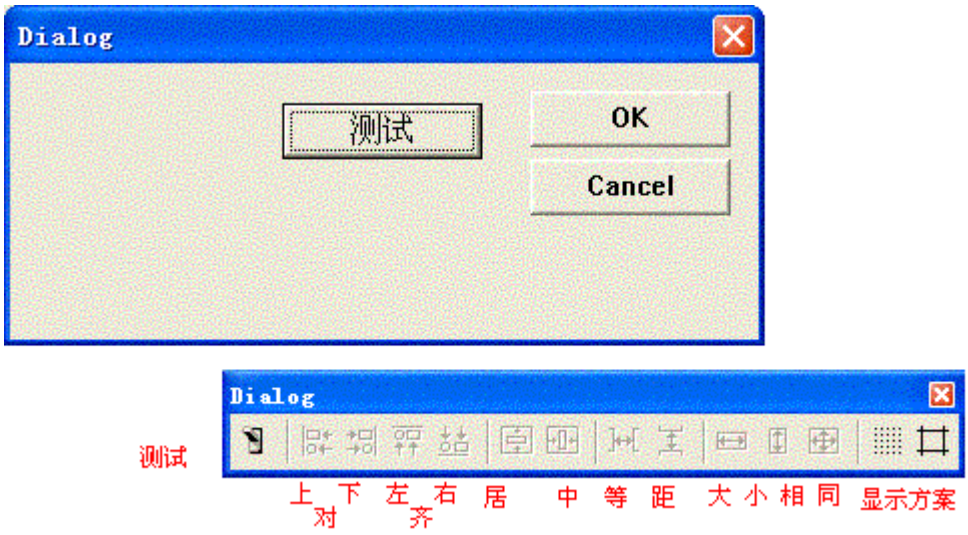
怎么说呢？VB 与 Delphi 的方便性使其失去了好多的灵活性，从这一点上来说， VC 比那两款编程软件要强。

我们同样地在名称上右键，然后选择“Properties...”，或者在对话框上右键选择“属性...”，就可以弹出一个属性对话框，将名称改为：“IDD_TEST”




当然，我们在对话框上右击出的属性中，有好多选项卡，可以表示对话框的样式，显示模式等一系列的东东，例如，我们标题可以改为“我的对话框”

我们默认生成的对话框上有两个按钮，一个 OK，一个 CanCle，她们由我们的父亲给处理了，当然，作为儿子，也可以通过响应的办法来处理这两个按钮，当然，为了与之比较，我们再加上一个按钮，名称设定为“测试”，名称为“ID_TEST”

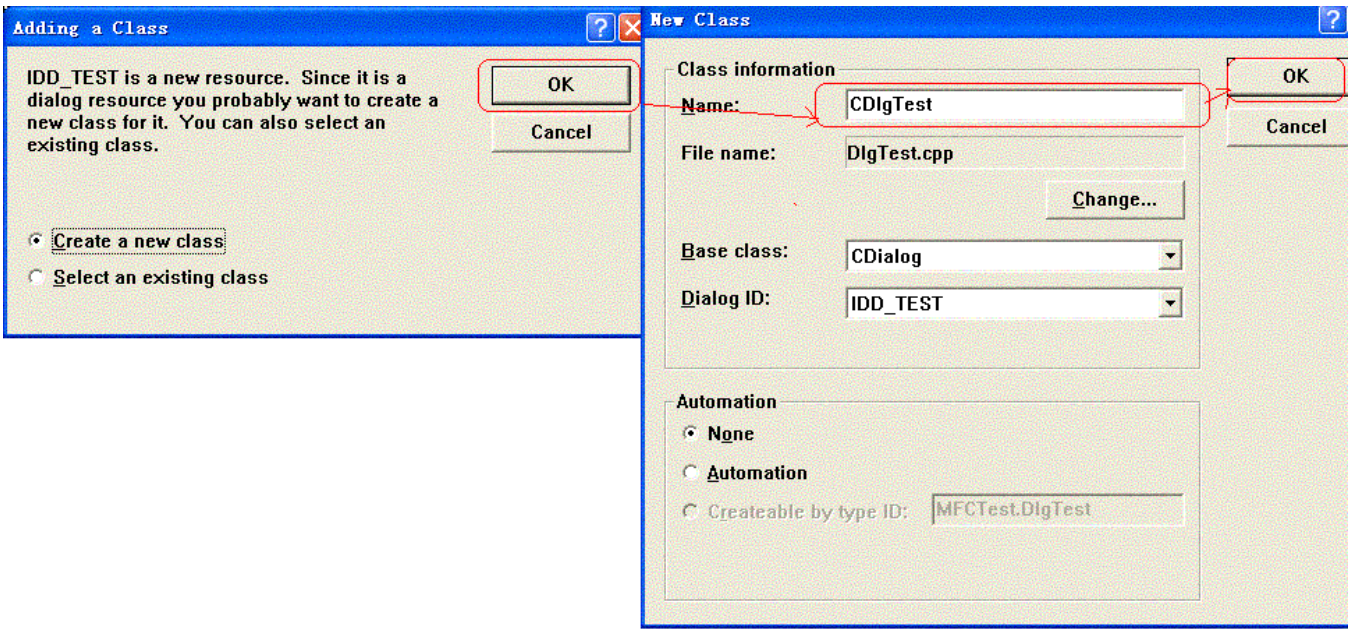


当然，我们上面还给大家一个工具条，其中第一个工具是测试我们的对话框，如图中所示：

对齐工具是当我们选择多个控件时（控件就是放在对话框上的东东），他们是靠左对齐还是靠右对齐，而居中是控件在窗口中的位置，间距也是指多个控件之间是否等距，而大小分同宽，同高，高宽相同，用以上的这些工具，很容易使得我们设置的界面很美观，也希望大家能够熟练使用。

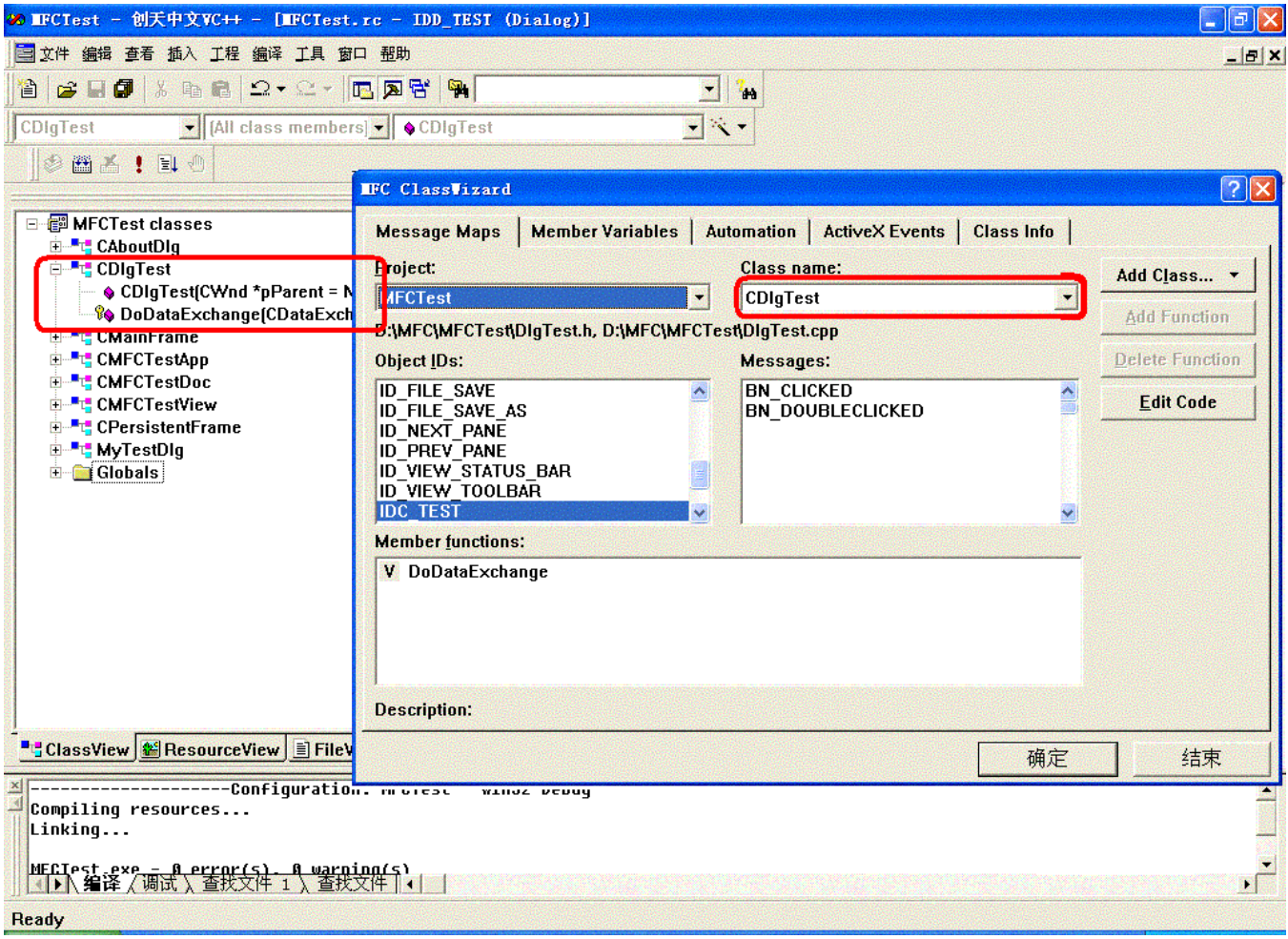
后面显示方案一种是显示限点，或者显示限线，VB 中是以点为限的，控件都自动对齐到点，而 VC 中就有两种显示方案，当然如果你用熟了，其实是后一种好用，具体用法，大家自己去摸索吧，有空发点源代码给大家，大家自己去看牛刀是怎么做的。

行了，大家双击对话框的空白部分或者右键选择“建立类向导...”，就会出现如下的对话框：




大家按 OK，然后在 Name 后建立自己的类，一般我们习惯于在前面加一个 C ，表示是 MFC 中的类，而“Base class”不变，即选择“CDialog”，按“OK”。

好了，我们就顺利地为我们的程序建立了一个新类：“CDlgTest”。




大家可以到类中去看一看构造函数，奇怪的竟然没有析构函数，看样那东西放给老爸了，呵呵。

噢，对了，大家注意好多 MFC 程序的类中的虚函数都是前面带有 virtual ，好像很奇怪吧？析构函数怎么能有同名函数呢？

原来啊，在类中，析构函数有专门的规定，也就是说，析构函数的的继承是可以不同名的，编译器专门用另一套方案给处理了，呵呵。

不说这些了，我们看上去是看看代码吧：

```
CDlgTest::CDlgTest(CWnd* pParent /*=NULL*/)
        : CDialog(CDlgTest::IDD, pParent)
{
}
```

大家看，构造函数用那种特殊的方法向父亲传递了两个参数 pParent，其中的 pParent 是父类窗，默认值是 NULL，上面表示出来了，而 CDlgTest::IDD 是什么呢？

enum { IDD = IDD_TEST };

竟然是将我们的对话框的名称传了过去了，并且是一种枚举类型。

大家是不是好像突然明白了什么，一个 enum 就使得我们的子类可以与父亲进行交流，将好多的事情去交由父亲进行处理了。

例如我们按 OK 按钮，那么，父亲就通过 DoModal 返回 IDOK 如果是 Cancle 则返回 IDCANCEL。

既然如此，我们去 App 中定义个类，当然，还是那句话，那个类是我们自己创造的，所以我们需要在我们的在我们需要使用的那个地方，那个文件的前面需要包含进我们类的头文件：

```
#include "DlgTest.h"
```

包含进这个类后，我们就可以在我们的一些消息响应函数中去使用我们这个类了，看程序，我们还是响应鼠标左键消息：

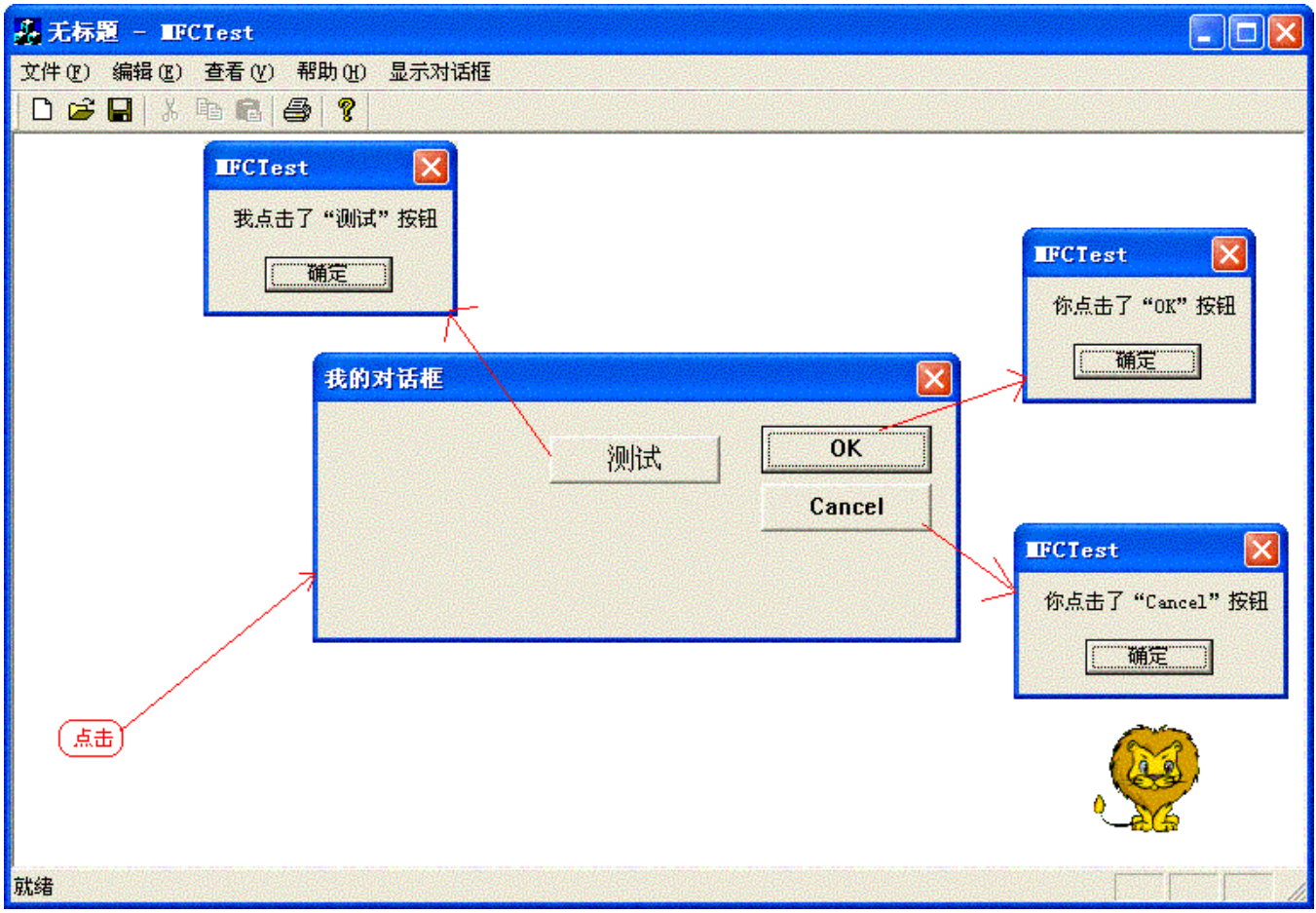
```
void CMFCTestView::OnLButtonDown(UINT nFlags, CPoint point)
{
	CDlgTest dlg;
	if(IDOK==dlg.DoModal())
	{
		MessageBox("你点击了“OK”按钮");
	}
	else
	{
		MessageBox("你点击了“Cancel”按钮");
	}
	CView::OnLButtonDown(nFlags, point);
}
```

当然，对于这两个按钮，我们的老爸给我们处理了，那么对于我们的测试按钮呢？在按钮上双击，会出现一个消息响应函数： OnTest

这就对了，在这儿就可以响应我们的单击此按钮的消息了：

```
void CDlgTest::OnTest()
{
	MessageBox("我点击了“测试”按钮");
}
```

于是，我们的一款小软件就基本成形了：

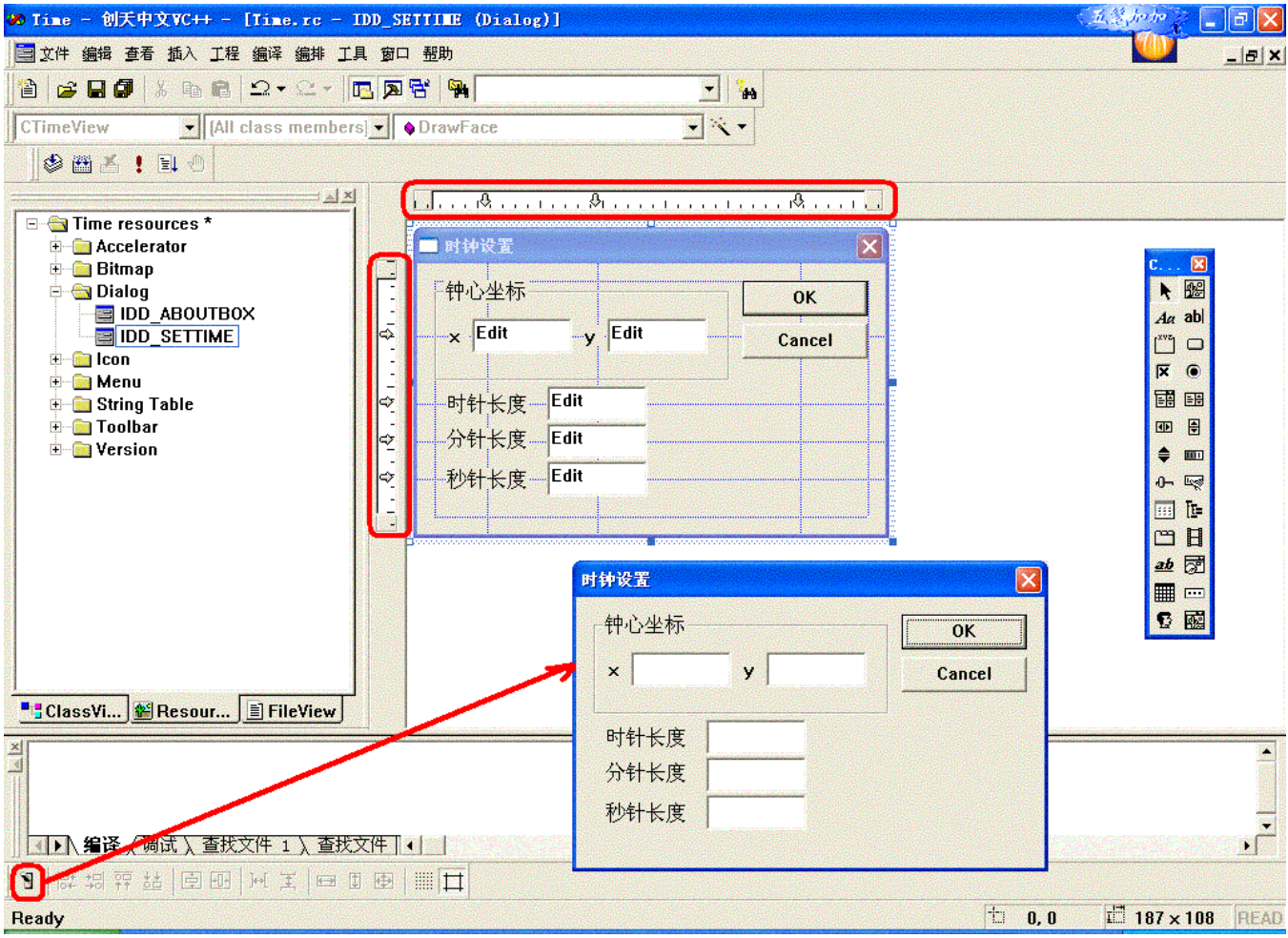



呵呵，不要以为这不是软件啊，任何一款复杂的软件，都是用这一套套简单的技术去实现的哟。

静态文本档，在 VB 中，被称之为“标签”，在 VC 中，静态文本是可以响应消息的，但需要设置，当然我们一般都当标签用。

而文本档的功能就相当丰富了，当然，我们现在包括以后我们给大家做教程，都是需要用到什么功能，我们就使用什么功能，说实话，一下子给你介绍那么多功能，你能记住吗？是不？当然，牛刀既然告诉你的功能，尽量保证你在实践过后，能够迅速地使用在自己的软件之中，牛刀的目标也就达到了。

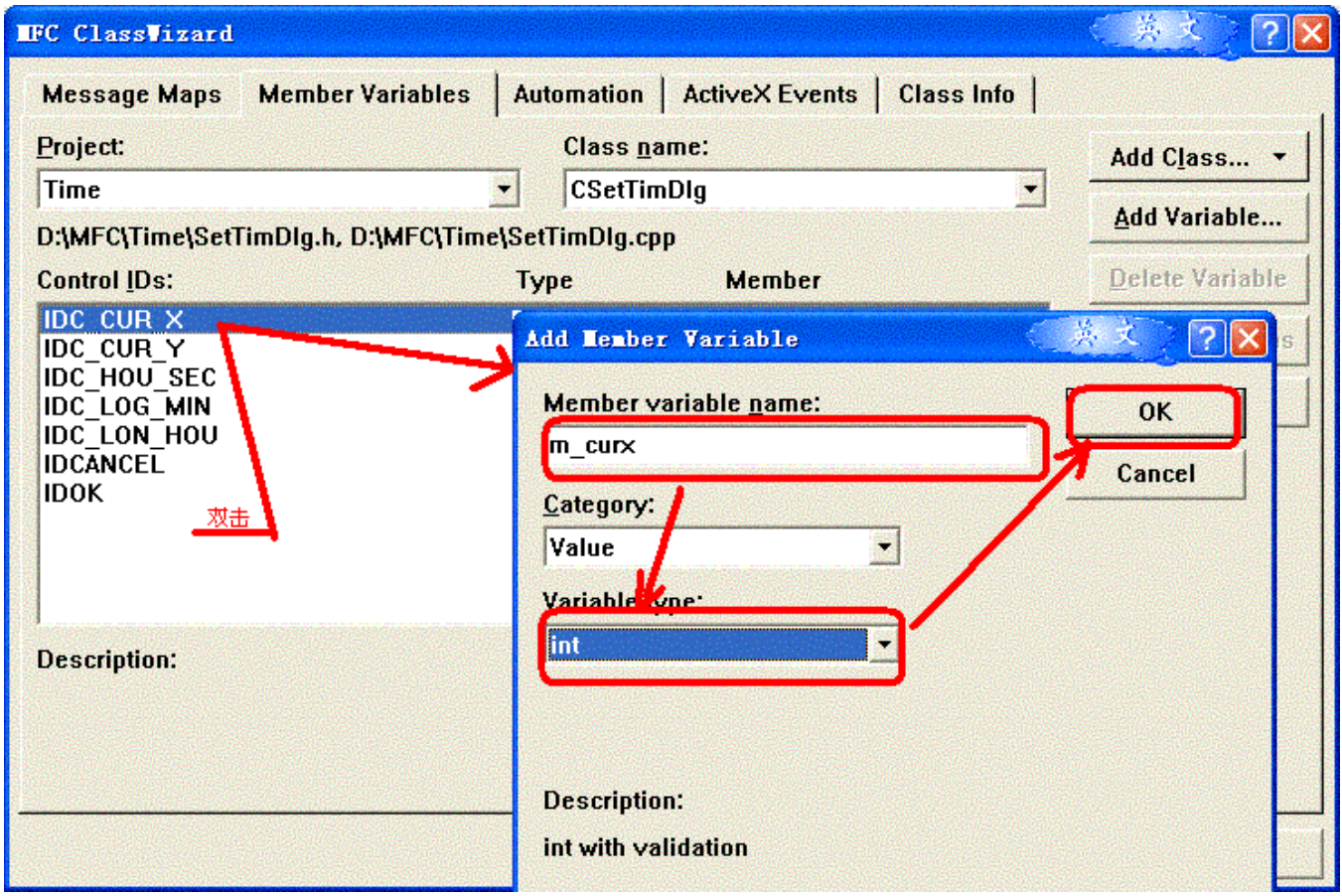
好了，这次我们到　Time　那一个软件中，新加一个对话框，按下面的图形去建立一个对话框：




| 控件 | 名称 | 标题 |
|---|---|---|
| 对话框 | IDD_SETTIME | 时钟设置 |
| 分组框 | IDC_STATIC | 钟心坐标 |
| 静态框 | IDC_STATIC | x |
| 静态框 | IDC_STATIC | y |
| 静态框 | IDC_STATIC | 时针长度 |
| 静态框 | IDC_STATIC | 分针长度 |
| 静态框 | IDC_STATIC | 秒针长度 |
| 文本框 | IDC_CUR_X | |
| 文本框 | IDC_CUR_Y | |
| 文本框 | IDC_LON_HOU | |
| 文本框 | IDC_LON_MIN | |
| 文本框 | IDC_LON_SEC | |

按上面的去设计，请大家注意，分组框与静态文本的名称都一样IDC_STATIC 奇怪吧，呵呵，这些都是摆摆样子，真正有功能的还是文本框，我们使用了五个，一个代表钟心的坐标，另三个分别代表钟的时针、分针、秒针

的长度，仅此而已，当然，这些定义都是整数，所以我们需要将这些整数拿出来，那好，我们先双击对话框，为其做一个类： CSetTimDlg ，然后我们在“建立类向导”中按开 Member Variables 选项卡，大家看到我们定义的几个文本档的名称了吧？双击每一个，会出现如下图的对话框，去给我们映射一些成员变量，大家如图上所选：




```
        int             m_curx;
        int             m_cury;
        int             m_longhour;
        int             m_longminute;
        int             m_longsecond;
```

几个文本档的参数设置如上，设置完后，我们就可以在 CSetTimeDlg 类中看到上面几个变量了，这几个并且都帮助我们在构造函数中初始化了：

```
CSetTimDlg::CSetTimDlg(CWnd* pParent /*=NULL*/)
        : CDialog(CSetTimDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CSetTimDlg)
        m_curx = 0;
        m_cury = 0;
        m_longhour = 0;
```

```
        m_longminute = 0;
        m_longsecond = 0;
        //}}AFX_DATA_INIT
}
```

大家看，这多好？也就是说，这些东西，一旦建立对话框后，里面的数据都初始化成了 0 了，有信，我们现在就到 View 类中去响应鼠标右键按下的函数，来制作一个模态对话框：

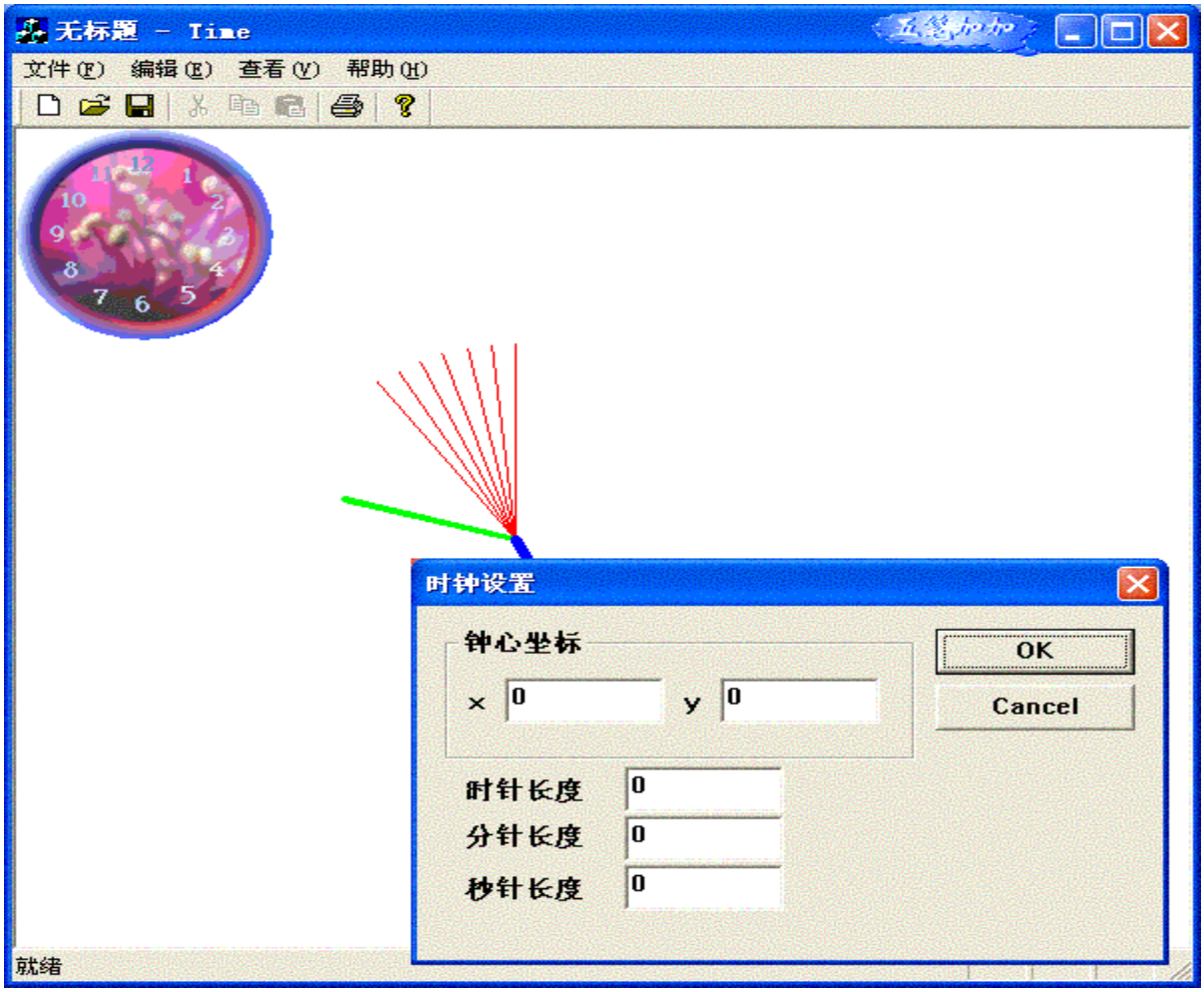
大家千万别忘了头包含：

```
#include "SetTimDlg.h"
```

然后在相应函数中输入下面的代码：

```
void CTimeView::OnRButtonUp(UINT nFlags, CPoint point)
{
        CSetTimDlg dlg;
        dlg.DoModal();
        CView::OnRButtonUp(nFlags, point);
}
```

于是，我们只要在我们的软件中按左键，就可以出现对话框喽：



是不是很有点那种感觉了？可惜的是，这时候，我们的对话框仅仅是出来了，相通过这个对话框来改变我们软件画面的上显示，那就必须将里面的数据传出来，那么数据怎么传呢？这些数据为什么可以传出来呢？我们的数据与控件到底是怎么整合在一起的呢？我们可以不可以将数据传进去呢？

等等等等这一系列的内容，我们下一讲与大家详细道来。

·内容回顾

如何创建一个对话框？
如何为对话框绑定一个类？
如何产生模态对话框？

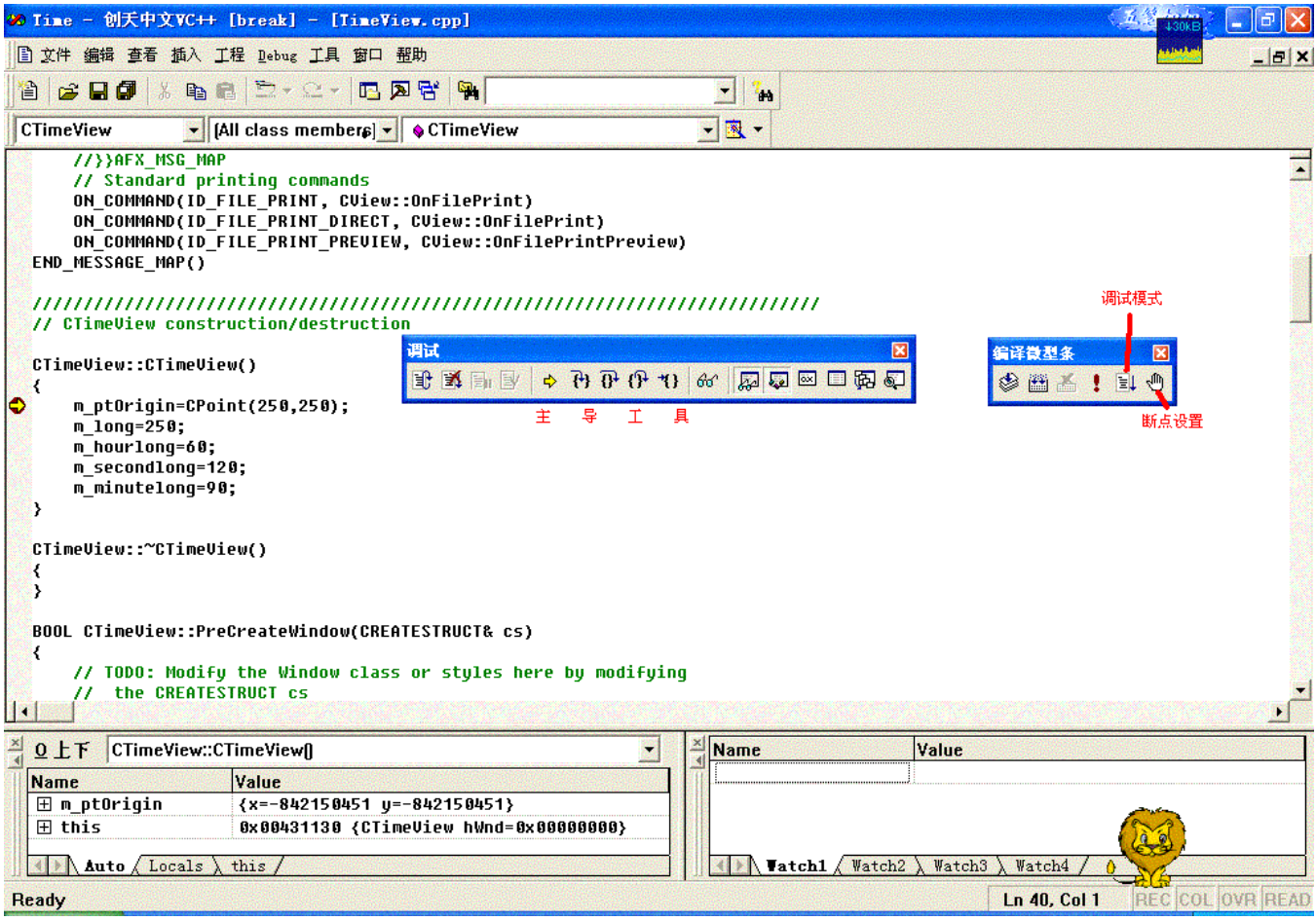
·作业，在时钟设置对话框中，响应 OnOK 函数，在此函数中显示确认信息，如果用户选择不退出，则不退出对话框，否则退出。仔细体会父类中 OnOK 函数的功能与作用。

在对话框中按回车键或 ESC 键，体会回车键与 ESC 键在对话框中的不同作用。

## 微软积木〇八八：谍影重重

想要真正地理解对话框的生成与销毁的过程，我们还得当回间谍，去探探老爸在儿子背后所干的事，既然想当间谍，那么我们就需要使用一定的手段与技术——调试技术：

请大家看下图，是我们常用的几种调试的按钮：




上图牛刀没有详细给大家注释，断点设置中的小手是设置断点的，而边上就是进入调试模式，而我们正常所使用的就是主导工具，当然，后面的是一些工具，至少我们这一讲用不到，大家可以先不关心，第一个黄色箭头是到当前调试点，而后面大括号上分别有箭头，看得明白吧？进入大括号（或者是函数，类，子类，第二个，则是运行语句，复合语句也当一个，函数也当一个地调试，第三个是退出当前括号、类、函数第四个是到当前光标所在行，就这样，大家自己去玩玩，说实话，这些东西，你自己去玩玩，那可以达到不言自明，如果你不上机，那么理论再好，你也没办法上手是不？

有些能力的确是“磨”出来的，好了，我们将从下面开始调试：

```
void CTimeView::OnRButtonUp(UINT nFlags, CPoint point)
{
CSetTimDlg dlg;
dlg.DoModal();
```

```
CView::OnRButtonUp(nFlags, point);
}
```

过程就不给大家描述了，描述个过程可能就需要不至一讲的内容了，毕竟我们这不是屏幕录像，呵呵，好了，牛刀先去试着调试了，一会儿告诉大伙儿结果：

```
{
CSetTimDlg dlg; //创建  CSetTimDlg 类的对象
CSetTimDlg::CSetTimDlg(CWnd*        pParent        /*=NULL*/)        :
CDialog(CSetTimDlg::IDD, pParent)
dlg.DoModal();
int CDialog::DoModal()
  CSetTimDlg:OnInitDialog 创建对话框
...其他初始化...
CDialog::OnInitDialog
  CWnd::UpDateData(FALSE)
   CSetTimDlg::DoDataExchange
用户输入数据
用户单击 OK 按钮
CSetTimDlg::OnOK //如果有的话
...其他确认处理...
  CDialog::OnOK
    CWnd::UpDateDate(TRUE)
     CSetTimDlg::DoDataExchange
   CDialog::EndDialog(IDOK) //销毁对话框
CView::OnRButtonUp(nFlags, point);
} //析构  CSetTimDlg  类的对象
```

看上面的东东可能有点难度，那么我们就分开来与大家讨论吧：

```
void CTimeView::OnRButtonUp(UINT nFlags, CPoint point)
{
CSetTimDlg dlg; //这儿构造了  CSetTimDlg 类
dlg.DoModal(); //这儿开始调用父类分如下过程
  产生对话框
  销毁对话框，返回对话框返回值
CView::OnRButtonUp(nFlags, point);
} //析构  CSetTimDlg  类
```

上面的告诉了我们，当我们定义了一个对话框类，其实根本没有产生这个对话框，然而，我们就可以对数据进行初始化了。

当窗口退出后，  DoModal 结束，但我们的对话框的类并没有结束，我们此时可以将对话框中的定义的数据进行导出

也就是说，当我们窗口没有产生，我们就可以对其中的一些值初始化，当窗口消失后，类中的成员变量还可以拿出来用。

那我们就纳闷了，那么这个成员变量是如何传进我们的窗口，而当我们窗口销毁之前，数据又是如何传到我们类中来的呢？

原来啊，那些过程都是由  CSetTimDlg:;DoDateExchange 去处理的，怎么处理我们以后再说，但是，我们一般情况下不直接调用这个函数，因为我们没办

法告诉窗口我们需要将数据传进还是传出，请大家注意啊，窗口的处理过程在老爸那儿啊，需要老爸下放的函数，我们才可以调用，你说是不？所以老爸下放了一个函数（噢，这一个函数可是爷爷下放的了啊，哈哈）

CWnd::UpDateData　　其中的参数为假则将数据传送给窗体，如果参数为真则将窗体中的数据传递给成员变量。

当然，这两个函数也是在窗体初始化之前或之后，自动调用，也就是说，那就没我们什么事了，我们只需要在 DoModal 之前，为类中的数据初始化就可以对窗口数据初始化，而窗口退出后，我们就可以顺利地使用类成员函数，其实说实话，这时候也就与我们的窗体没什么关系了，为了测试，我们试试看，请大家输入以下程序：

```
void CTimeView::OnRButtonUp(UINT nFlags, CPoint point)
{
CSetTimDlg dlg;
dlg.m_curx=m_ptOrigin.x;
dlg.m_cury=m_ptOrigin.y;
dlg.m_longhour=m_hourlong;
dlg.m_longminute=m_minutelong;
dlg.m_longsecond=m_secondlong;
if(IDOK==dlg.DoModal())
{
  m_ptOrigin.x=dlg.m_curx;
  m_ptOrigin.y=dlg.m_cury;
  m_hourlong=dlg.m_longhour;
  m_minutelong=dlg.m_longminute;
  m_secondlong=dlg.m_longsecond;
}
CView::OnRButtonUp(nFlags, point);
}
```

很可惜，牛刀 View 类中的定义与 CSetTimDlg 中的定义不一致，呵呵，大家定义的时候注意了，这就是马虎的代价，呵呵，像我们这种程序，改一改或者干脆就不改都无所谓，但如果你的代码让别人读就会使得别人感到很纳闷喽，好，看我们运行两次不同效果：

感觉怎么样？是不是有点那种感觉，还可以？很可惜的是，这种调用每次都需要重新设置，非常繁琐，那么我们如果需要设置一次就一直使用怎么办呢？那么这种设置结果放在存储器上就行了，例如放在硬盘上，存成一个文件，或者放在注册表中，或者干脆就用鼠标在屏幕上一点就可以直接改变这些值。

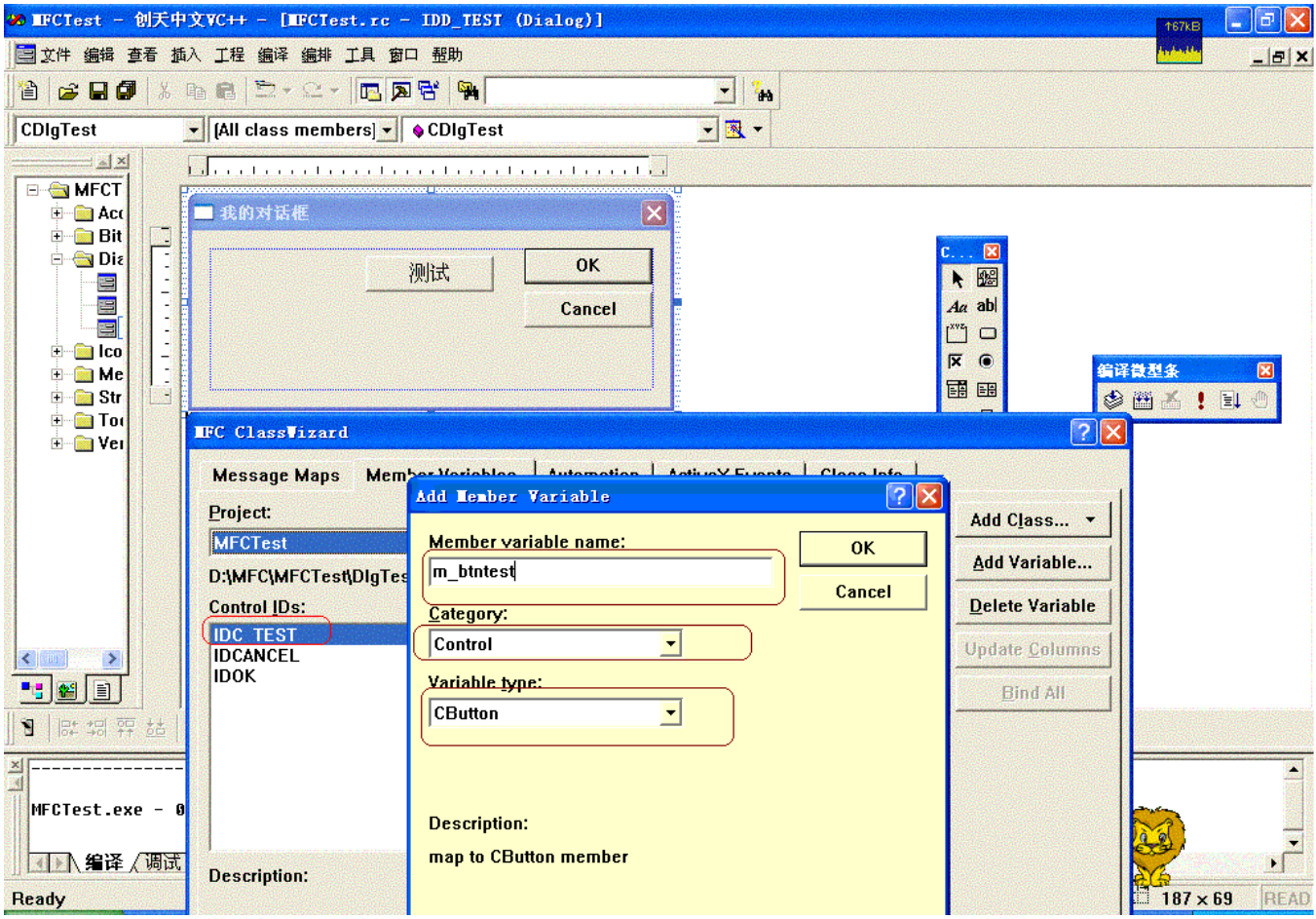
当然，从软件设置方面来说，我们设置到这儿，我们这款软件已经相当优秀了，你觉得呢？

好了，我们这款时钟的软件就讲到这儿了，以后我们教大家存储技术或其他一些相关技术时，大家就可以去完善这个软件，使得我们的软件越来越完善，越来越完美。

那么我们接下来要干什么呢？接下来就要来点恶心的了，呵呵，大家还记得前面我们与大家讲的“第三者”的事吗？

想要明白这个道理，就们就要知道，我们对话框上，只要是方的东西，就都是一个小窗口，这个窗口与我们以前教给大家的那个 View 一样，是一个窗体，也有句柄，甚至也可以给其发送各种消息，可惜的是，我们对这些东东，好多的消息都隐藏了，也就是说，按钮，就只当按钮用了，文本框就当文本框用了，当然有其一套的使用原则，所以用起来非常方便，但也正因为如此，就失去了其好多的灵活性，为了恢复其灵活性，我们就不得不在这个父子之间“插入”一个“第三者”，通过第三者，去实现我们的目的了。

还回到我们以前曾经举过的一个例子，如果你真的按牛刀教材一个一个地去做的话，你的电脑里应该有一个对话框，这个对话框中有一个标题是“测试”的那个按钮，是不是？ ID 值为 IDC_TEST，好，我们在那个对话框上，去右键一下，选择“建立类向导”，还到第二个选项卡“Member Variables”，我们在 IDC_TEST 上双击，在上面填上 m_btntest，一定要“确定”：




确定后，大家可以看到我们的对话框类中有一个定义：

CButton m_btntest;

我们将其改成：

CMyBtn m_btntest;

当然，我们还可以看到下面的函数中有：

```
void CDlgTest::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CDlgTest)
DDX_Control(pDX, IDC_TEST, m_btntest);
//}}AFX_DATA_MAP
}
```

那是干什么的？ DDX？ 那就是宏，完成我们成员变量与窗体之间的数据交换的，也就是我们运行 UpDateData(FALSE); 后要调用的函数，其中的宏，如果没有这些，我们就不能够与窗体交换数据。

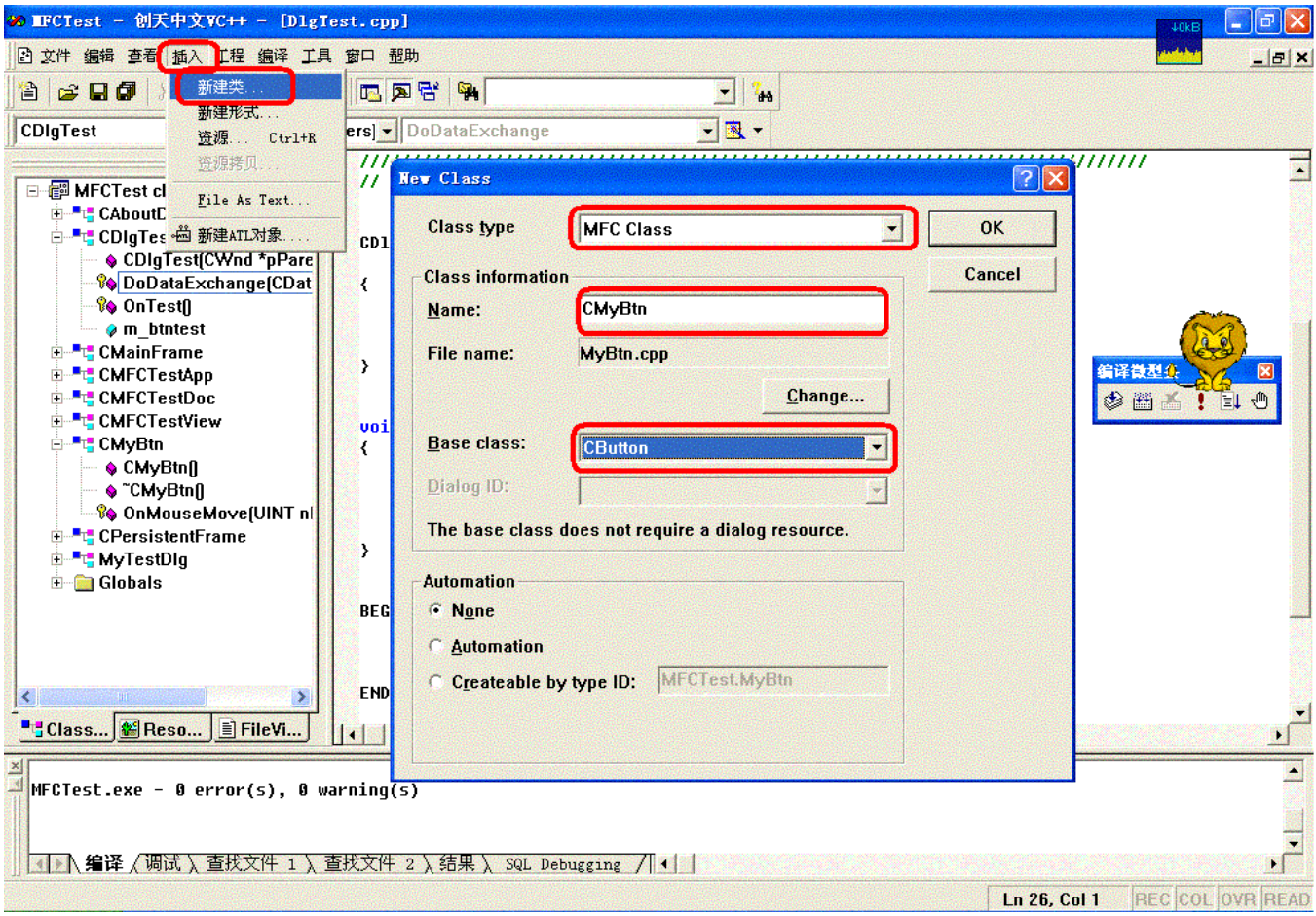
当然，这些细节我们以后再与大家详细地描述，但大家看下面的：

CMyBtn m_btntest;

不错，我们是将名子改了，可惜我们并没有这个类啊，那怎么办呢？自己去建一个吧：

按“插入/新建类...”，

Class type　中选择 MFC Class
Name　中输入　CMyBtn //当别人老爸
Base class　中一定要选择　CButton //当别人儿子
怎么样？就这样就插入了一个类了，当人家儿子的爸爸，当我家老爸的儿子，于是，形成了独有的结构：




CMyBtn m_btntest; //这儿收了一个儿子
class CMyBtn : public CButton //这儿认了一个老爸
于是乎，我们可以凌驾与儿子的头上，为所欲为了，干什么呢？响应一个鼠标移动的消息吧：

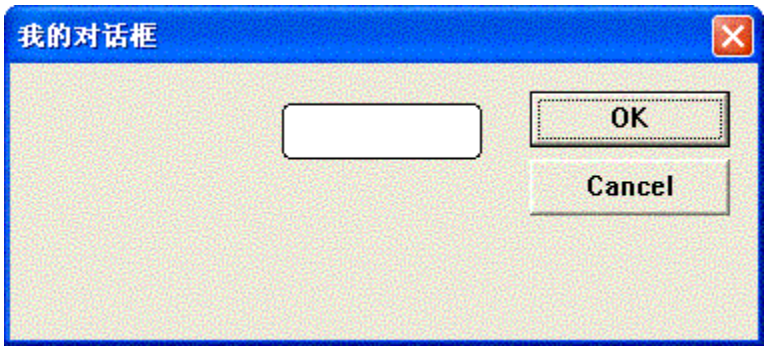
```
void CMyBtn::OnMouseMove(UINT nFlags, CPoint point)
{
CClientDC dc(this);
dc.LineTo(point);
CButton::OnMouseMove(nFlags, point);
}
```

干什么？呵呵，你想不到，在按钮中移动鼠标还可以干什么吧？看图：

当然喽，建立大家在不要按钮中响应单击事件，当然也不要将按钮的功能改变成为编辑框的功能，或者将文本框的功能变为列表框的功能，也就是说，我们在为儿子去制造父亲的时候，尽量不要去改变儿子的基本功能，如果改变了还是那句话，可能别人会认为很奇怪。

咱们响应一个 WM_PAINT 消息吧：

```
void CMyBtn::OnPaint()
{
CPaintDC dc(this);
CRect rect;
GetClientRect(&rect);
CPoint pt(8,8);
dc.RoundRect(&rect,pt);
}
```

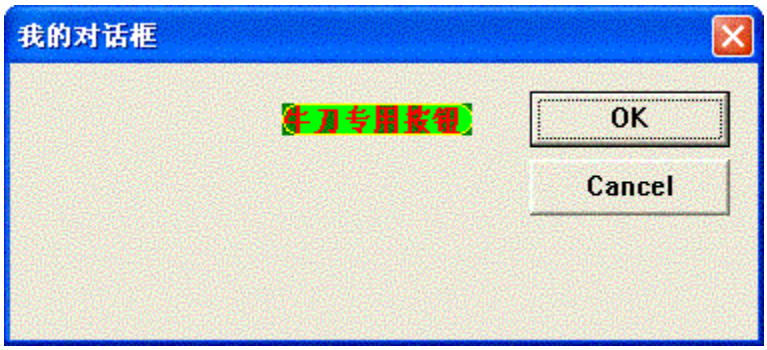


怎么回事？我们的那一个按钮哪儿去了？怎么只剩下一个白白的方块？

大家去点点那个方块，就知道是怎么回事了。

突然有一个想法，弄一幅图片在按钮中去“贴图”怎么样啊？是不是牛刀的想法很厉害？说干就干：

```
void CMyBtn::OnPaint()
{
CPaintDC dc(this);
CBitmap bmp;
bmp.LoadBitmap(IDB_BTNTEST);
CDC btndc;
btndc.CreateCompatibleDC(&dc);
btndc.SelectObject(&bmp);
dc.BitBlt(0,0,95,16,&btndc,0,0,SRCCOPY);
}
```




有点什么感觉？是不是觉得自己越来越伟大了？呵呵

这种父子插足的办法可以说是几乎通吃一切，我们可以对你看到的程序中的所有东西进行所有你想要的修改，并且修改得不露痕迹。

大家还知道牛刀在 MFC 教程的一开始与大家说过的吗？我们的 MFC 就像一个积木，我们的类的技术就像是一套全面的瓦工技艺，微软是用这套技术去盖起一种软件构架，而我们却可以用相同的技术去拆掉微软给我们的盖起的一切，当然喽，我们是想拆哪就拆哪，想不拆，咳，咱们就还用微软的，既然微软给了我们这种东西，咱们凭什么不去用啊？

当然，就是这套技术，同样的原理，我们可以自由地去创造我们想像中的任何一款按钮，圆形按钮，三角形按钮，甚至我们可以是用按钮来显示一幅图片，使得我们的按钮显得格外地美观，漂亮：

大家看上面的按钮怎么样？是不是觉得很神奇？是不是很想拥有自己的例

似的按钮？其实原理都是一个样，都是需要采用这种父子“插足”的办法去更改那按钮的显示，只要我们的编程水平到位了，那么我们想做一个很美观的界面，那岂不是“小菜一碟”？

上面的图片可能会引起一点麻烦，可咱们学习就是要博采众长是不？（那按钮，牛刀可没取得使用权啊，呵呵，但愿没人来找牛刀的麻烦。牛刀声明啊，有谁找牛刀麻烦牛刀就删了，请想要欣赏的人尽快欣赏，多点点这个链接啊，不然说不准哪天，这个按钮图片就没有了，呵呵）

其实说实话，我们做程序的思路应该很开阔，再者，就是我们就是要注重基础知识的扎实，我们的基础知识只要打扎实了，好多的东东其实是很容易理解的，而如果我们对类的继承还一窍不通的话，我们的这些程序我们看起来肯定是怪怪的，一点都不奇怪。

既然说到知识产权了，我们当然应该尊重别人的知识产权，例如我们使用别人的一些东西时，我们如果出于演示与教学的目的，那我们的软件产品就不要商品化。

当然，这种父子插足的办法，我们还可以直接去使用别人的类来达到，怎么弄呢？例如我们这次就对 OK 按钮进行加工加工吧：

大家使用“建立类向导”为 IDOK 新建一个按钮控件的成员变量

CButton m_btnok;

改类为：

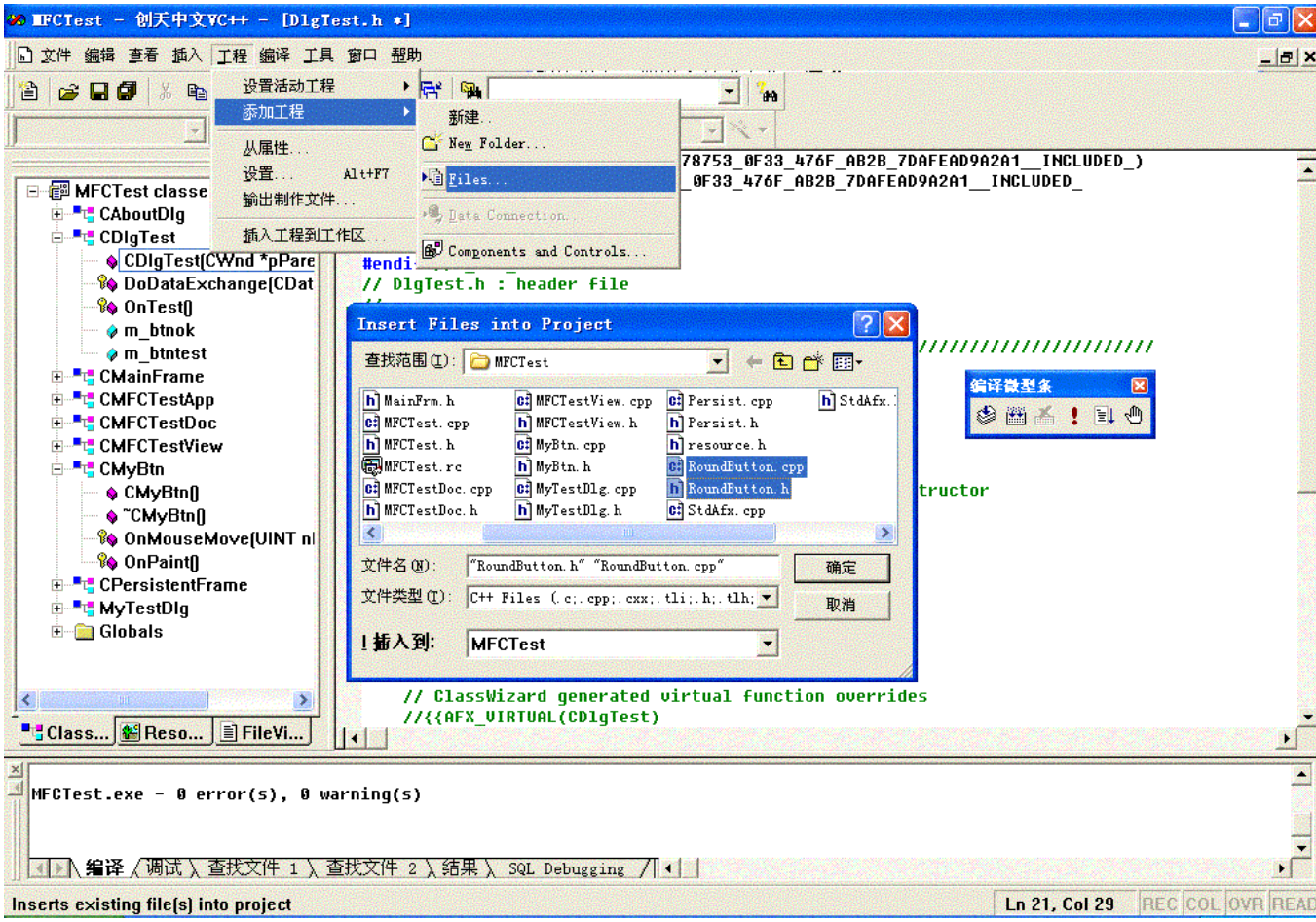
CRoundButton m_btnok;

将附件中的两个文件复制到我们的工程目录下，

点菜单“工程/添加工程/Files...”，

将新添加的文件加入进来。

在自己的文件中相应位置加入相应的头包含（注：大家自己思考应该在哪个文件，什么位置加，加什么样的头包含，牛刀这儿不给大家说明了，大家应该学会自己思考问题。）




编译文件，运行文件，看结果：

是不是有那么点感觉了？如果去网络上去找一找，其实有好多功能强劲的类的文件，我们都可以用这种办法很容易地加进来，或者再到人家那里再改一改代码，很容易地能够达到我们所需要达到的功能，可是，还是请大家尊重别人的劳动成果，这种软件，大家拿来自己研究研究，提高自己的编程水平可以，千万不要动不动就将别人的成果占为已有，那样做至少不道德，你认为呢？

## 微软积木〇八九：投机取巧

说实话，我们的按钮，包括其中那些控件，其实都是一个一个小窗口，请大家注意了，我们运行的，保要我们能够看的的，基本上都是窗口。

我原来想说，我们看见的方的，都是窗口，其实我的想法是错误的，我们其他不规则形状的窗口，其实也是从方形窗口上切开来的，当然怎么切，我们以后再讲，因为如果现在想讲的话，那我们的题目就不应定为“投机取巧”喽，呵呵。

那么我们投的是什么机？取得是什么巧呢？

原来啊，我们对于这种像按钮啊这些东东，其实也必须像窗口一样，需要建立消息循环，需要为其构造各种消息响应，然而，这些东西，系统统统地为我们自动地做喽，而需要我们响应的消息，往往就是很少很少，并且在 MFC 中，系统自动给我们创建一些函数，去直接使用这些消息，这不是“投机取巧”么？

什么意思，例如我们以前大家点击“测试”按钮， MFC 并没有给我们定义一个按钮的类，并且在类中要响应 WM_LBUTTONUP 消息，而是直接给了我们一个 OnTest 函数，让我们直接在其中处理相应事务，这不是“投机取巧”么？

什么意思，就像你有一个老爸，你想吃桔子，老爸帮你将皮扮开了，你想吃鸡收，老妈帮你剥了壳了，就是这个意思。

所以，假如我们定义一个 m_btntest 的话，那么我们应该去处理 WM_LBUTTONUP 消息，而老爸说，不用了，你告诉老爸你想干什么吧，老爸帮你干，就是这个意思。

```
void CDlgTest::OnTest()
{
MessageBox("我点击了“测试”按钮");
}
```

大家看到这个测试按钮的响应了吗？其实不应该这样响应，这是老爸帮我们响应的，其实应该在我们上一讲讲的那个自己的类中这样定义：

```
void CMyBtn::OnLButtonUp(UINT nFlags, CPoint point)
{
MessageBox("我松开了“测试”按钮上的鼠标");
CButton::OnLButtonUp(nFlags, point);
}
```

就是这样，那么这个消息是怎么传递过来的呢？是儿子向老爸发了一个消息，将按钮的点击与按钮的 ID 号从 WPARAM 和 LPARAM 传递过来了。

而儿子发送的这个消息就是 WM_COMMAND 消息，当然喽，父亲这里用的不是继承机制，而是消息映射机制，是儿子发送消息给老子，所以，我们这些儿子控件就都需要有一个老子，而其类型中则都要选上 WS_CHILD

好了，为了证明我们的构想，我们去建立一个按钮，这次我们不用对话框编辑器了，这次我们在 View 窗体中直接建立，当然喽，因为我们需要这个按钮的生命周期与窗体同步，所以我们需要的是一个全局变量，当然，我们一般不用全局变量，所以我们可以在 View 类中申请一个成员变量：

```
class CMFCTestView : public CView
{
CButton m_mybtn;
……
```

然后，请大家注意，我们创建按钮必须在窗体创建之后，才能进行，所以这种工作不能够放在构造函数中进行，因为构造函数运行结束了，窗体也没有创建好，那么窗体什么时候创建呢？我们响应一下 WM_CREATE 消息吧，输入如下代码：

```
int CMFCTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CView::OnCreate(lpCreateStruct) == -1)
  return -1;
m_mybtn.Create(″ 牛 刀 制 作 （ 按 钮 ） ″,WS_CHILD
|WS_VISIBLE ,CRect(0,0,200,200),this,123);
return 0;
}
```

请大家注意，我们这是创建一个按钮，所以需要将我们的按钮的一些信息通过参数传进去，如：

·第一个参数，按钮上显示的名子，你想用什么就用什么

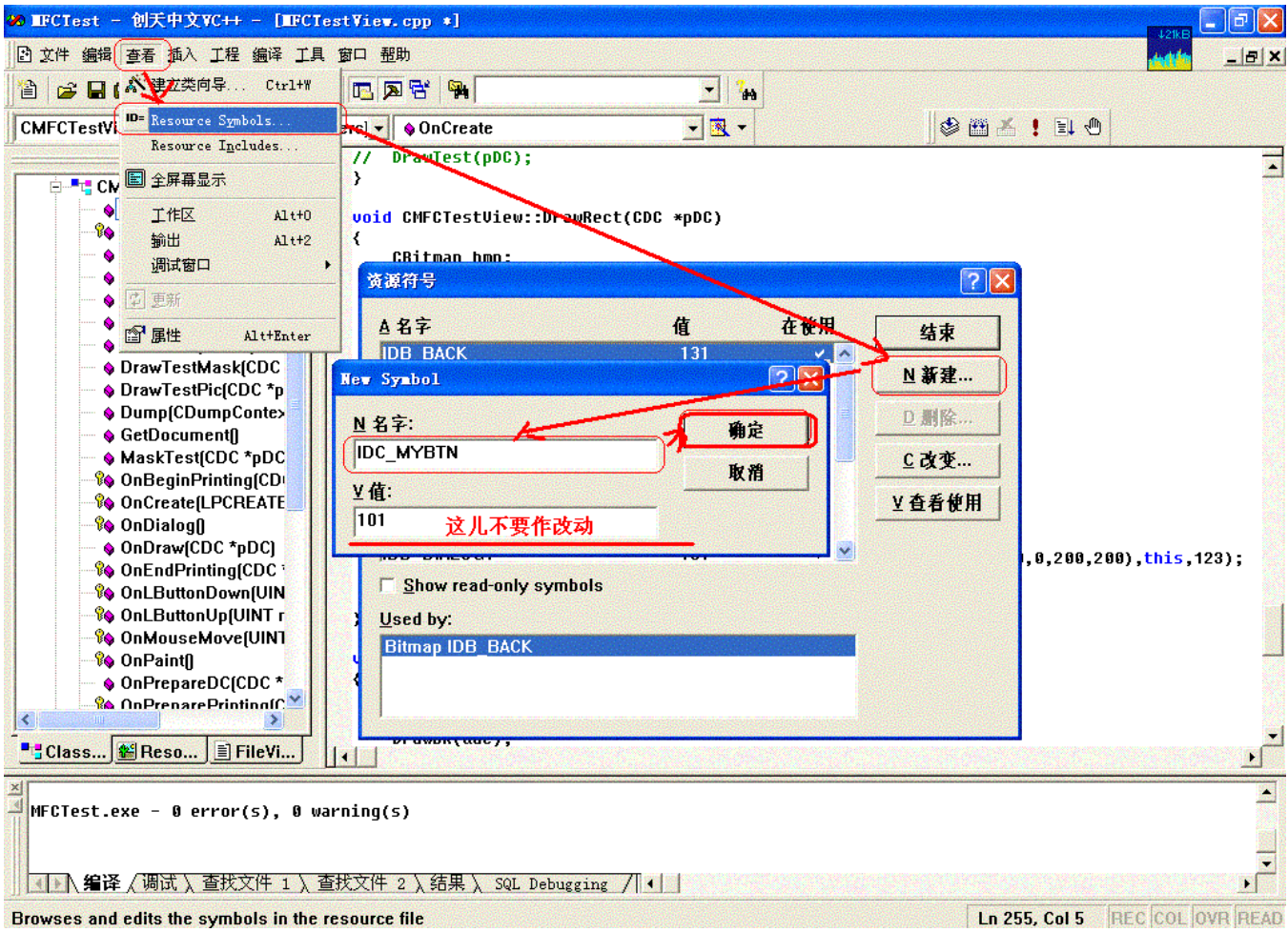
·第二个参数，按钮的类型，因为按钮必须是子窗体，所以一定需要 WS_CHILD 消息，因为我们需要按钮显示出来，所以要用 WS_VISIBLE 消息，当然，如果不用这个消息，那么这个按钮即使产生了，我们也是看不见的，必须再次调用 m_mybtn.ShowWindow 函数来显示按钮，这下大家就知道 ShowWindow 的作用了吧，呵呵。

·第三个参数，按钮显示的矩形区域，当然，我们这里用了一个 CRect 为什么可以这样用呢？因为区域类有一个静态的构造函数，可以直接调用，当然点类 CPoint ，尺寸类 CSize 等都可以这样使用，很是方便，就不需要再定义了。

·第四个参数，是一个指针，是指明父亲，牛刀上面说的 WM_COMMAND 消息，其实是子类向父亲发的一个通知，我们称之为“通告消息”，以后我们说菜单时，我们也还要讨论这个消息，而这个如果我们传进的是什么指针，按钮就创建在什么父对象上，通告消息就会发给谁。

·第五个参数，就是这个按钮的 ID ，我们可以任何定义一个值，当然，我们教大家可以这样做，但如果大家自己做时，一定要为你的按钮定义一个 ID 用 #define 或者最安全的办法是通过 查看/Resource Symbols.../新建...

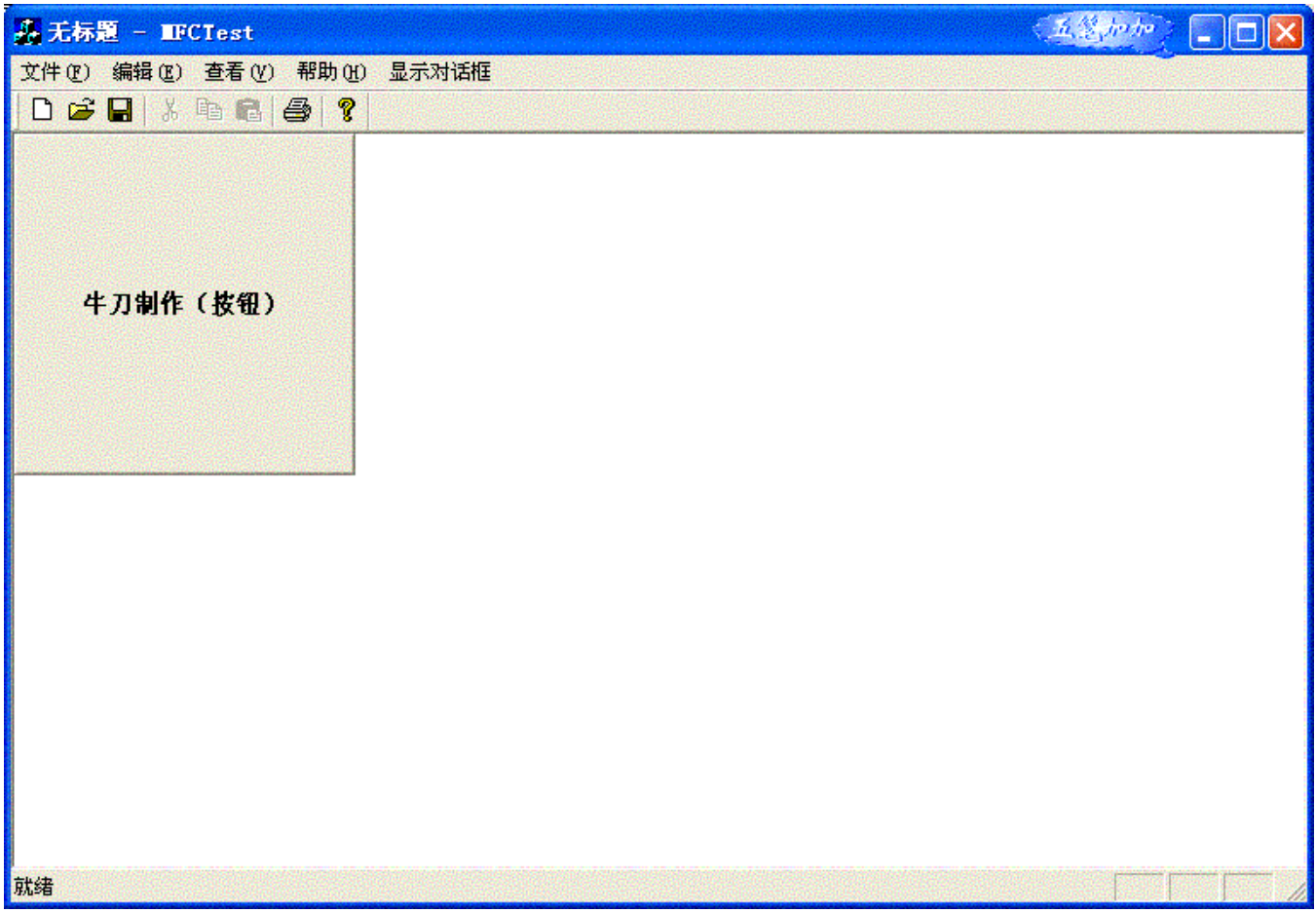
来定义一个资源名称，然后通过这个资源名称来使用资源：




于是，我们上面的代码就可以变为：

m_mybtn.Create(" 牛 刀 制 作 （ 按 钮 ） ",WS_CHILD |WS_VISIBLE ,CRect(0,0,200,200),this,IDC_MYBTN);

我们生成的按钮如下图所示：




当然，很可惜的是，这样添加的动态的按钮，“建立类向导”显然是不可能自动为我们添加消息响应的，所以这一切过程都需要我们自己去完成。

当然喽，怎么添加，大家找一段我们以前做的代码，去看看别人是怎么加的，然后照猫画虎，就行了。

说实话，照猫画虎应该是我们以后做程序所必然要掌握的一项基本的技能，因为你不可能将 MFC 或者 API 学得面面俱到，当然牛刀也显然不可能将所有细节都讲的面面俱到，是也不是，那么，我们的最佳方案，就是从别的的代码中能够学到自己想学的知识，这是一种能力，当然喽，你只要能够琢磨出一个，依葫芦画出一个瓢，那么，以后再遇到类似的问题就必然会迎刃而解了。

当然，还有一种情况，就是我们可以用好多的数据，通过一个循环，来创建好多的按钮，这种时候，如果要通过一个函数对所有的按钮进行响应，那么我们有两个办法，一个办法就是需要自己做出一个自定义的消息，按照自己的思路，做出一个消息，发送到消息队列中去，从而响应我们自己定义的消息，那就“自定义消息”的技术，还有一种办法，就是上一讲我们讲到的，干脆就用一个第三者，当然，在使用第三者的时候，就注意能够将我们的按钮的按钮信息（例如我们单击了第几个按钮了）通过某种办法让父亲知道，虽然这个父亲是新让的，但毕竟是父亲啊，这年代，哎～～～，新来一个父亲竟然，不说了，伤感～～～

当然，牛刀在此仅作一些提示性的建议，如果大家想要达到这些目的，那还得自己去钻研钻研哟，呵呵。当然我们这一讲不打算讲这些，因为如果我们讲这些的话，那叫“严重跑题”。

其实说实话，按钮部分如果不严格来说的话，我们已经给大家讲掉了，其实我们对于一般的按钮处理基本上是向父亲发一个“通告消息”，然后由父亲的相应的通告函数去处理，就是这样，就是通过这样一个消息，实现了我们的“投机取巧”，从而为我们的按键处理争得了大量宝贵的设置过程，从而使得我们能够从繁重的消息映射之中解脱出来，用很简单的办法去实现我们程序的基本功能。

很可惜，的是，通告消息，在我们类的定义中竟然找不到了，呵呵，没办法，谁让人家直接用相应的消息函数去就将好多事都做了呢？哎～～～

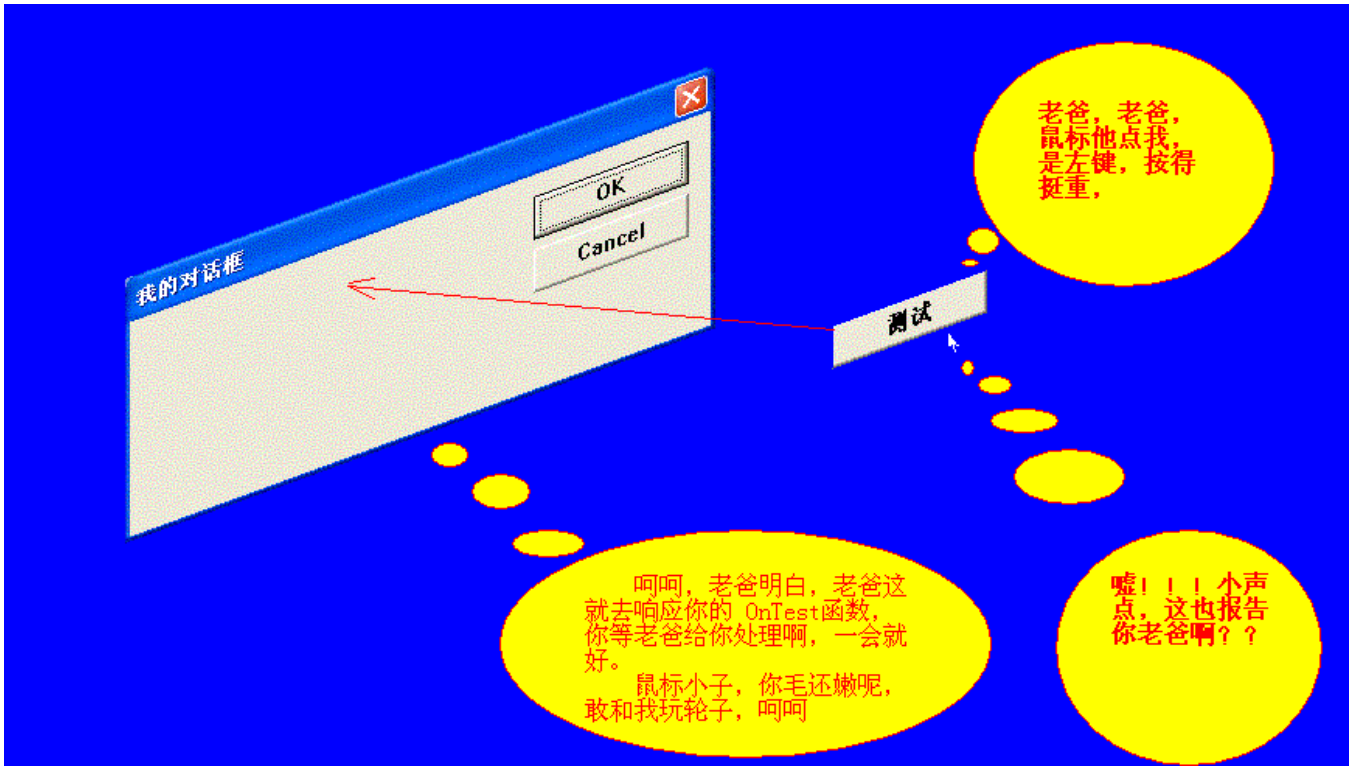
也就是说，这些被我们称之为“控件”的窗体，都没有自己的消息循环，或者说这种消息循环已经被系统帮着我们自动地去做了，我们只要在对话框上画出一个按钮，那么，这个按钮就仅仅是在我们按时，向父窗体发一个通知，老爸，有人非礼我了，于是老父就问：他想干什么？儿子说道：

这个嘛，说来话长～～～

～～～老爸呀，这个人的名子他叫鼠标，点的是左键在我身上，儿子告诉你这个消息啊～～～～～

饿饿饿抬饿随便你老爸怎么响应嗯～～～～～啊～～～饿饿抬抬饿抬抬饿～～～老爸：好好说话，干什么你？不就是一个鼠标消息吗？值当的吗？啊？

于是，其实不在儿子那，在老爸那，就有了我们的 OnTest 消息，请大家注意哟，其实这个消息并不是响应 WM_TEST 消息，而是响应的 WM_COMMAND 消息，我们称之为“通告消息”，也就是相当于儿子与老爸在发短信，如下图：




其实菜单用的也是这种技术，说出来你还别服。但大家想一个问题，菜单的老爸是谁呢？

View ?

Doc ?

MainFrame ?

App ？

说实话，如果你说是谁吧，那就是 MainFrame 是正宗的老爸，但这个老爸呀，也不太保守，竟然将这种特权四处下发，说实话， View 不是 MainFrame 的子类，但却是子窗体（儿子），这是怎么回事呢?原来啊，这个 View 也是一个窗体，标标准准的是 MainFram 的儿子，可是呢，这个儿子和老子关系处理得比较好，老子那儿由菜单发过来的消息，竟然可以一个不拉地发给这些儿子们，当然不是通过虚函数喽，也是通过消息吧，也就是说，一个菜单上的消息，在上面四个类中，都可以去响应。那当然既然响应就有顺序之分啦，但虽然有顺序，却是截断性的，这个消息只要没响应过，就按以下顺序向下分，直到有响应消息为止：

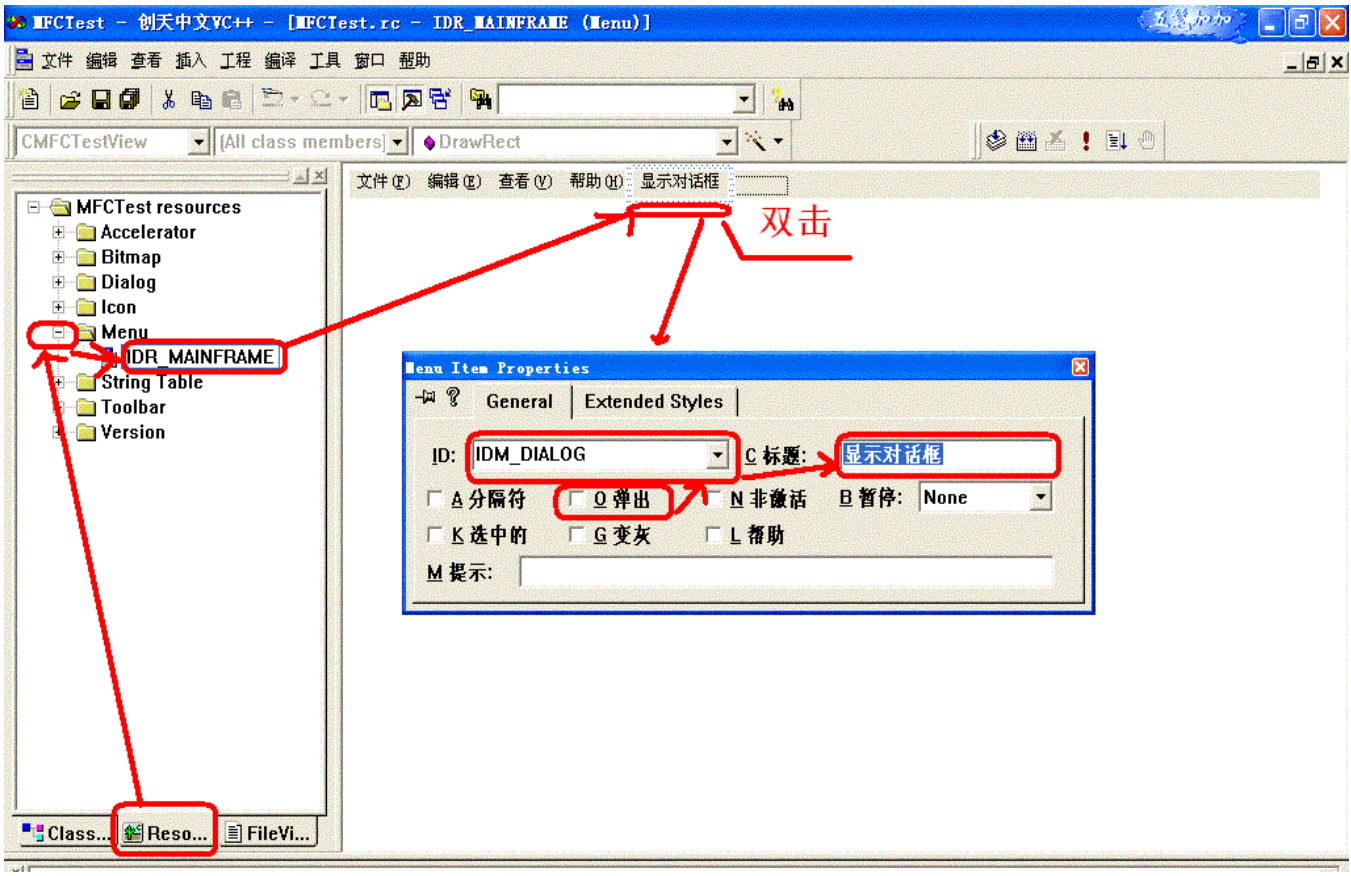
View

Doc

MainFrame

App

当然，你可以从上面的去一个一个地测试，当然，一般情况下，我们的菜单都响应 View 消息，当然，如果是读写文件啊等等，我们就可以去响应 Doc 如果我们要对外面的窗口进行操作，就可以去响应 MainFrame ，如果大家想干一些坏事，那就去响应 App 的喽，当然，牛刀不可能跟你一项一项去试，牛刀只教大家如果响应 View 中的通告消息吧：




如果是顶层菜单，那出一般“弹出”前面被复选的，也就是说，那个菜单中只表示一个名子，我们不对此菜单作消息响应，所以这个菜单一般就不需要 ID 号，但如果我们将复选标志去掉，就可以输入 ID 值了，然后我们就可以对此菜单进行响应。而“标题”中一般是名称，当然，如果是字母，那么字母前面加一个“&”

符号，那么就会在相应的字母下面加一个下划线，则此字母就被设置成了一个菜单的激活码，也就是大家看我们文件菜单上不是 F 下面有一下划线吗？那么我们按 Alt+F 就可以直接打开此菜单，下面的都可以使用。

对了，大家可以将菜单中的某一个菜单的“弹出”复选，看能达到什么效果，其实牛刀建议大家学习电脑知识不要去翻书本，绝大多数的知识可以在“上手”的过程中直接学到，牛刀总结出以下几点电脑学习建议，大家试试，可以发现好多东西其实根本不需要从书本上去学：

·鼠标左键要常用
·单击双击加三击
·没事右键弹弹看
·经常出现一菜单
·不同位置出不同
·注意留心翻一翻
·键盘三个功能键
·经常手指放上边
·点点击击击击点
·不怕电脑坏半天
·功能键
·F1
·如果你
·有问题
·常常上去点一点
·通常是帮助
·一般不会变
·常用快捷键
·经常理一理
·Ctrl+C 复制
·Ctrl+V 粘贴
·Ctrl+X 是剪切
·编程请注意
·习惯是关键
·不要去改变
·常用快捷键
·鼠标常与三键用
·Shift、Ctrl 与 Alt
·经常使用快捷键
·电脑越用越熟练
·假如你真能精通
·鼠标请你靠一边

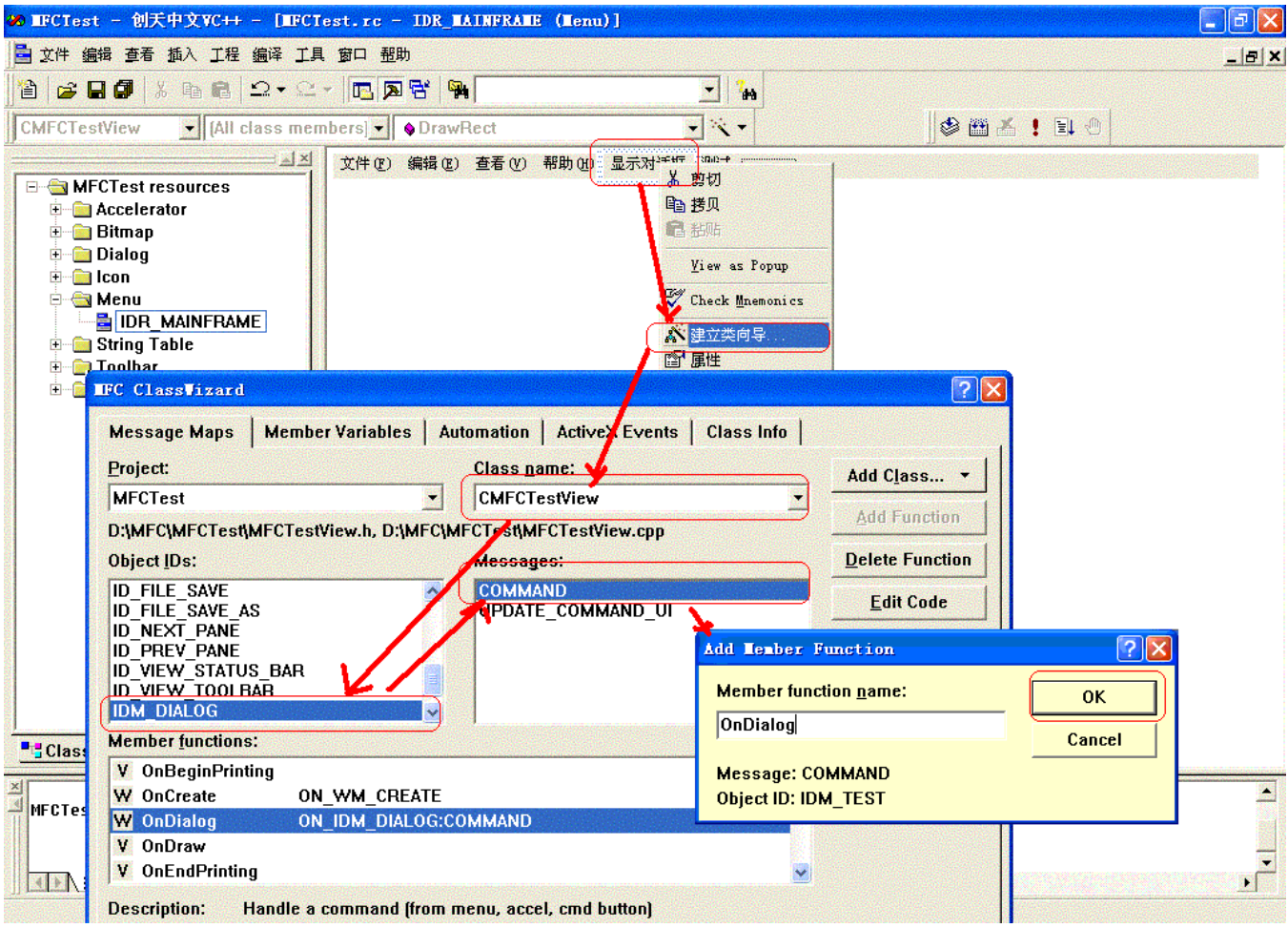
说实话，在 DOS 时代过来的人，是不经常去用鼠标的，他们的用什么？快捷键。牛刀曾经见过一个打字员，说给大伙信吗？她自己说的，一分钟能打 300 多个汉字，牛刀也看过他打字的过程，全盘盲打，连功能键，小键盘都不需要眼睛去看，她的眼睛整个过程都不离开稿件，呵呵。当然喽，对于我们搞程序的来说，键盘速度可能并不是主要，但我们做程序，特别是做菜单时，就一定要考虑

到别人的习惯，如果你　Ctrl+V　是表示删除，别人一定很纳闷。

当然喽，我们现在的 MFC，其实好多的快捷键系统都给我们做了。

行了，大家看图上还有一个“提示”，可以用　\n 分开，前面的一般显示在下面的状态栏中，而后面以后我们讲“工具条”时再与大伙讨论。

当然喽，如果你做完了以上的工作，就可以在相应的菜单上右键/建立类向导




从而，一步一步，就可以建立一个函数了，与按钮的响应几乎一样：

```
void CMFCTestView::OnDialog()
{
CDlgTest dlg;
if(IDOK==dlg.DoModal())
{
  MessageBox("你点击了“OK”按钮");
}
else
{
  MessageBox("你点击了“Cancel”按钮");
}
}
```

大家这就可以将程序敲进我们的计算机去看看运行效果了。

## 微软积木〇九〇：盗亦有道

不知道大家对牛刀的这一系列的教程有什么看法没有？是不是感觉牛刀似乎有点一“驴一脚，马一脚”的感觉？

当然肯定和传统的教材有所不同，一般的教材肯定会这一讲是什么呢？“菜单”，那一课是什么呢？“对话框”……

而牛刀的教材与那些不同的明显的一点，就是按照我们通常的思路去走的，上一讲吧，与大家讲了讲菜单，可我们暂时还不想深入下去，不然就似乎有点“跑题”的感觉，呵呵。

我们这一讲与大家讲什么呢？就讲讲我们“盗”的问题。

大家搞程序时间长了必然会有一个能力，我看到一款好软件，需要注册，当然舍不得兜中的钱啦，于是，呵呵，不是要注册吗？咱们也不注册，也不破解，咱做一个。

你可想没有想过，那其实也是“盗版”，听说有一种说法，代码相同的只要超过７０％就会涉嫌这一个——“盗”。

所以我们说，我们不可能不“盗”，但——盗亦有道：

牛刀在 MFC 第一讲曾经与大家说过，品一段代码，那是一段思维，事实果真如此，例如我们贴一幅图吧，代码几乎就已经固定是那种形式了，你怎么改逃脱不了与别人的代码一样，你能说那是“盗”吗？

当然喽，有人去搞破解，那一点的确不道德，但有一点是肯定的，搞破解省时省力。

从破解的一套理论来说，那的确是不费很大的事的，原因简单，我们当输入一个密码时，如果我们输入“12121212121212121212”，那么这个数据不管怎么加密，在内存中肯定有一个时刻是这一串数字，那是逃不掉的，找到这一串数字，再去前后找跳转指令，嘿，你还甭说，那是百分之百找得到，就是哪一个指令是跳到注册完成的位置，哪一个句是为密码加密，哪一句是指令加密等等，这一套技术，那需要相当扎实的基本功，必须相当扎实。

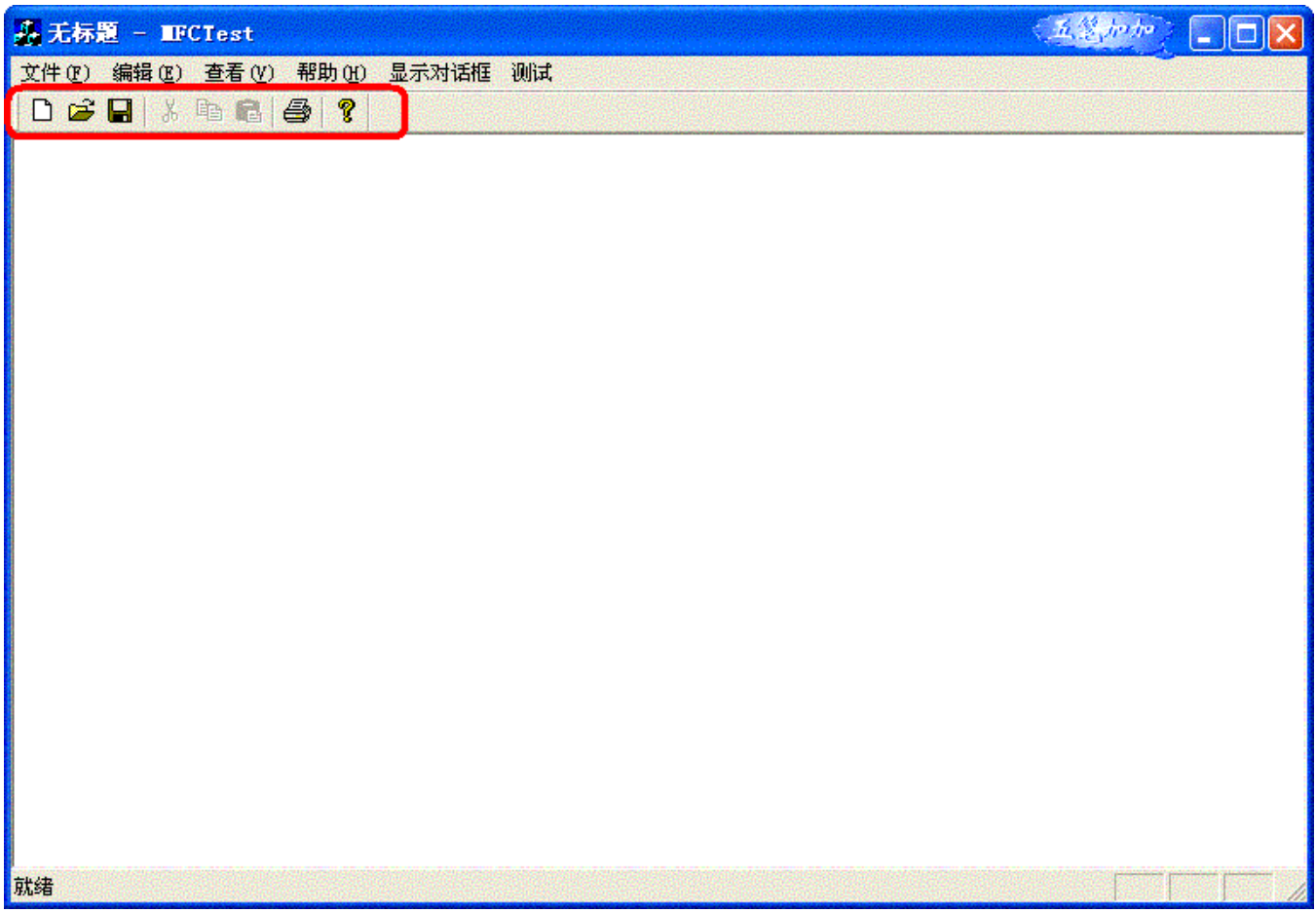
大家不要试图用别人做好的破解软件去做破解，如果你不明白其中的道理，再好的破解软件，破解的能力必定有限，但如果你自己明白其中的道道就不同了，你可以自己去做出一款破解软件来。

当然喽，咱不跑题啊，呵呵。

我们要与大家讲的不是去破解别人的代码，而是我们拿到别人的代码应该怎么去读，怎么去看，怎么去理解。

牛刀以前曾经提出一个说法——知识块，你掌握的知识块越大，你的阅读代码的能力越强，还不瞒大家，牛刀对安全方面的知识就是不足，所以论坛上好多的安全方面的代码，其实牛刀是读不来的，正常，从这个角度来理解，可能大家就明白牛刀的良苦用心了，牛刀的教程并不是按大家所想像的，什么菜单，什么工具栏、状态栏，对话框，一个一个专题地讲，而牛刀与大家的理解就不是这样，牛刀认为不管是菜单、工具栏、状态栏、对话框、包括所有的控件，那都是一个一个窗口，我们可以随意地去用“老子插足”的办法，去改变我们想要改变的一切，这一讲咱来个比较恶心的——“插足工具栏”

工具栏就是图上那一块：




红框框起来的那一块，说实话，对于工具栏的编程，咱们这一讲不想与大家讲，那个编程比较简单，也是与菜单栏的响应一个样，用“投机取巧”的办法，在父窗体上响应，并且一般其 ID 与相应的菜单用同一个 ID ，这样一来，我们几乎不需要为其编程，直接画出来，并且定义一个 ID，就一切 OK 了，所以说格外地方便很可惜的是，大家在 MFC 为我们自动生成的代码中，几乎看不到相应的相应函数，怎么回事？当然全在父亲那里。

好了，我们进入 MainFrame 中去看定义，一定有一个定义：

CToolBar m_wndToolBar;

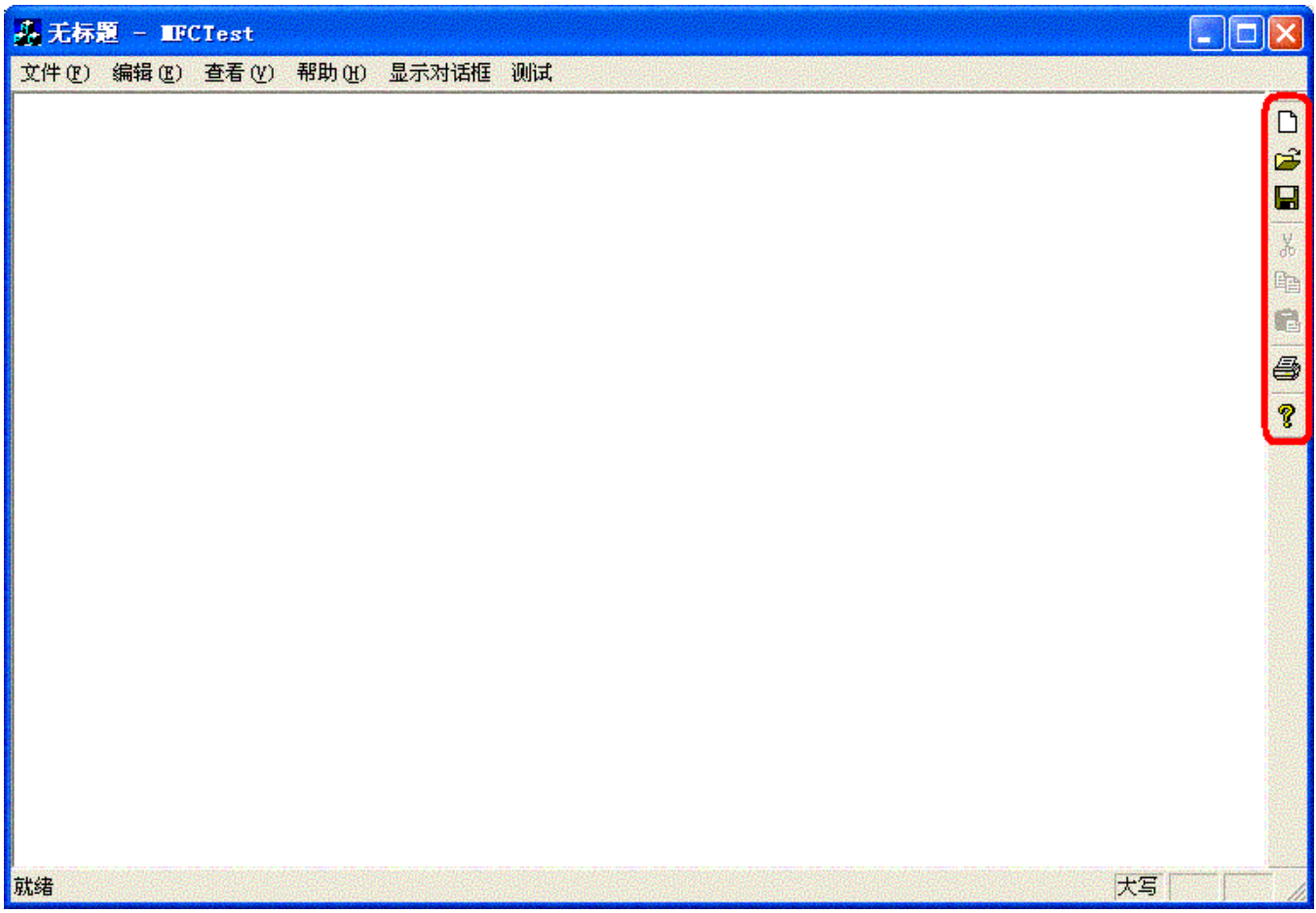
当然，既然定义了，就有相应的创建函数，在 OnCreate 函数中：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
  return -1;

if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP
  | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
  !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
  TRACE0("Failed to create toolbar\n");
```

```
  return -1;
}
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
return 0;
}
```

我们为什么这一讲的题目叫“盗亦有道”呢？请大家自己去创建一个新的工具条，然后思考，自己去为自己的软件加上一个新的工具条，靠左，怎么样？

上面的主要代码是 m_wndToolBar.CreateEx 那一部分，其中有一些东东，大家都可以改一改，变一变，删一删，加一加，等我们对这段代码有一个很好的了解了，那么我们再去创建一个我们自己的工具条，然后再试图去实现一些功能，当然了，我们可以先将上面的 CBRS_TOP 可以改成 CBRS_RIGHT ，然后大家看会出现什么情况？




不错，事实就是如此，我们只要明白了其中的道理，那么我们“偷”什么有什么，呵呵。

学习编程的目的就是“为我所用”。别人好的代码，我就是要学，学来了，就是我们自己的，我们脑袋中的有用代码越多，我们做起程序就是越方便，思路就会越清晰。

一段好的代码必然是值得一品的。

好了，闲话少说，我们还是来看看那个定义：

CToolBar　m_wndToolBar;

改为：

CMyBar　m_wndToolBar;

当然喽，需要加入一个类，类的名子叫 CMyBar 类的父类叫 CToolBar

请大家注意，你选择父类时，没有 CToolBar ，你先选择 CToolBarCtrl，然后建好类后，进去将所有的 CToolBarCtrl 改为 CToolBar 就行了，这种就算是 VC 没有考虑到位，这不奇怪，大家一定不要畏首畏尾，缩手缩脚，该改动的地方是大刀阔斧，绝不留情，呵呵。

的确有些人会对 VC 为我们生成的代码有一种神圣感，或者生怕改出问题来，如果你现在也有这种感觉，那牛刀告诉你，你应该去补补基础了，牛刀已经将好多基础的东东给你都讲了，你还有什么不敢的呢？

当然，改的地方有两处：

```
class CMyBar : public CToolBar
BEGIN_MESSAGE_MAP(CMyBar, CToolBar)
```

在哪里，自己去找找，当然，你改一处，另一处不同系统会报错，这也是一个不错的办法。

当然喽，在 CMyBar m_wndToolBar; 定义的那个头文件前需要加上相应的头包含：

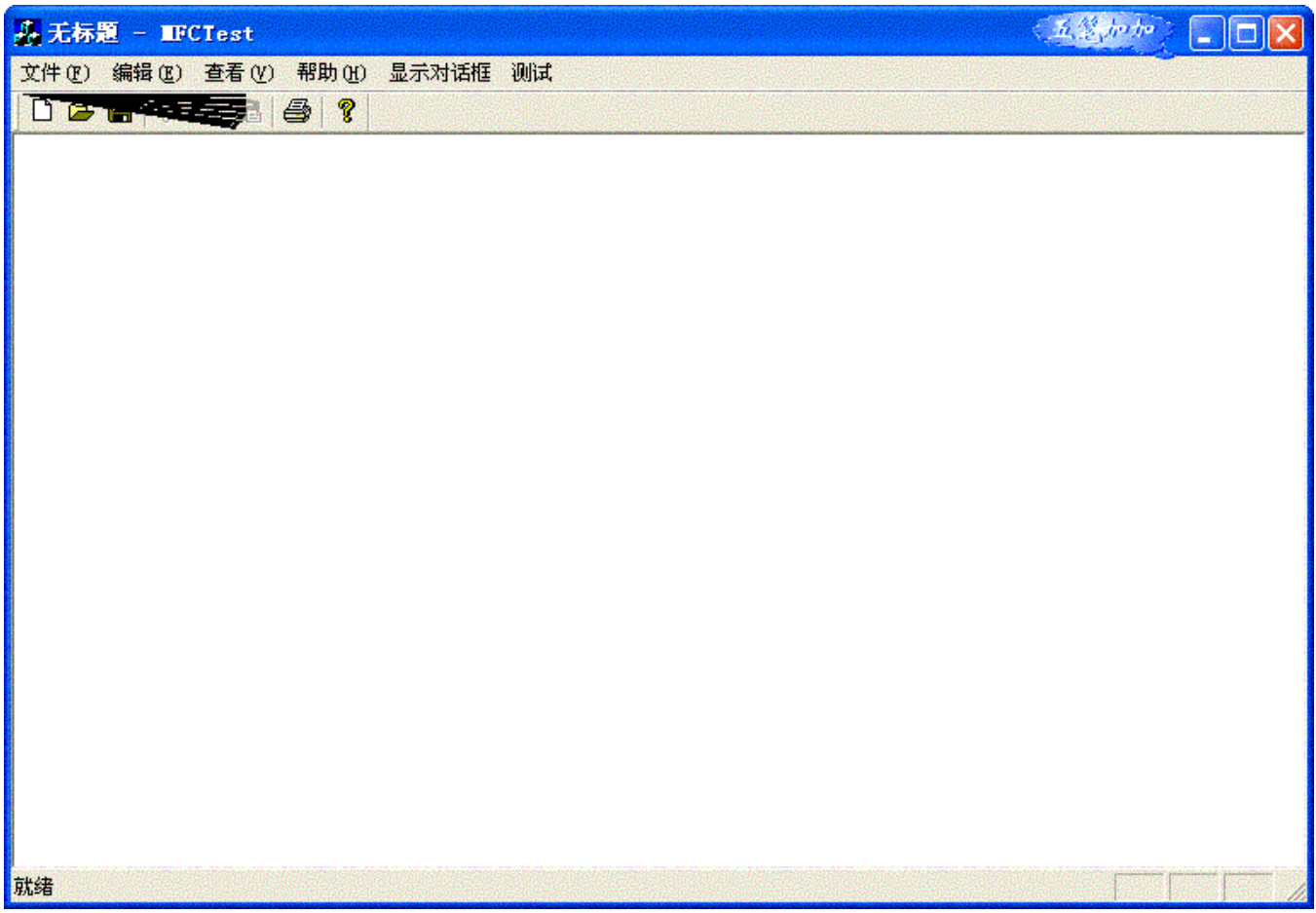
```
#include "MyBar.h"
```

然后我们就可以编译了。

当然，编译后的生成软件与我们以前的软件并没有什么区别，这就是我们仅仅是“插了一下足”，而没有去行使自己的权力，那么，下面我们就要行使当父亲的权力了，大家仍然在 CMyBar 类中响应鼠标移动的消息：（还是以前的老一套，呵呵）

```
void CMyBar::OnMouseMove(UINT nFlags, CPoint point)
{
CClientDC dc(this);
dc.LineTo(point);
CToolBar::OnMouseMove(nFlags, point);
}
```

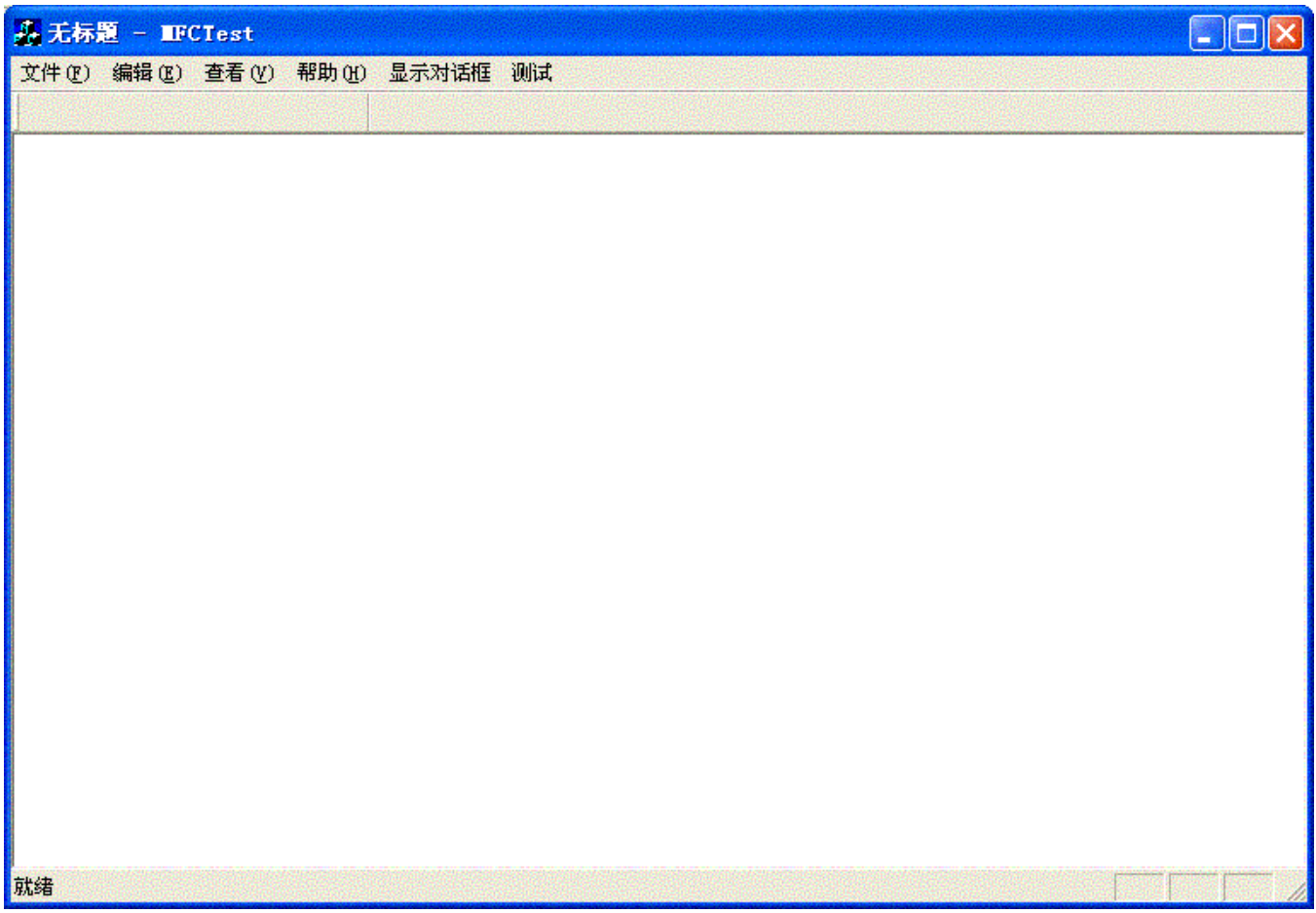
当然，效果嘛，呵呵：




显然我们也可以响应 WM_PAINT 消息：

```
void CMyBar::OnPaint()
{
CPaintDC dc(this);
}
```

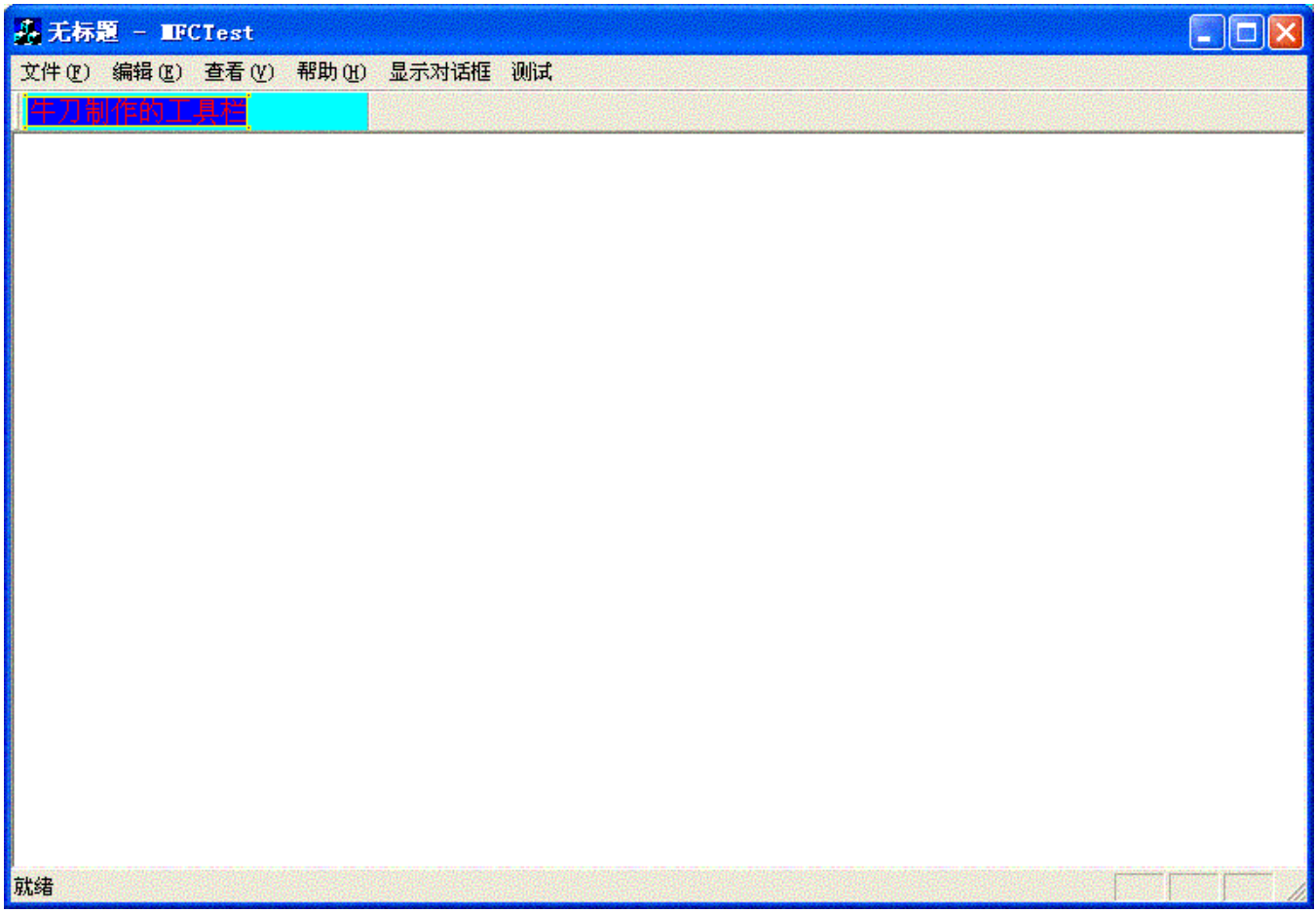
当然，你只要儿子响应了，根据虚函数的定义，我们知道，老子的消息就没办法响应了，于是，显示出来的效果就是一片空白。




当然喽，一张白纸可以画最美的图画，我们同样可以在上面去贴图，或者干一些其他的事：

```
void CMyBar::OnPaint()
{
CPaintDC dc(this);
CBitmap bmp;
bmp.LoadBitmap(IDB_TOOLBAR);
CBrush brush;
brush.CreatePatternBrush(&bmp);
CRect rect;
GetClientRect(&rect);
dc.FillRect(&rect,&brush);
}
```

当然，我们上面的也是一种贴图办法，很实用，呵呵，大家看效果：




当然，牛刀这图仅仅作一个演示地作用，如果大家做的时候，那就要考虑各个按钮的位置，功能，大小等等好多好多的情况了，我们系统默认的按钮是十六色的，我们通过这种办法，就可以很容易地做出一个真色彩的效果的按钮，并不是很困难的，当然，也必然会有好多的人将这些功能做成一个一个的类，其实如果我们想去“盗”，找个类，加到我们工程中就一切 OK ，但还是那句话，我们明白了其中的道理，我们做出程序来才会更显得灵活。

下面的那一个方块可就是状态栏了，大家用相同的办法，试着去响应响应状态栏的一些消息，看能不能也在上面贴上一幅图片，加上一行文字，甚至加进几个控件，从而使得我们的状态栏也显得格外地生动、活泼。

当然，你也可以在状态栏的插足过程中去响应 WM_TIMER 消息，从而使得状态栏中的字动起来、跑起来，甚至滚起来，其实我们只要明白其中的道理，我们本来就可以随心所欲的，或者说，我们可以变得无所不能。

牛刀其实告诉你一个事实，牛刀教你去理解了某种机理，其实真的比你将书上的一段代码敲进电脑能够运行得起来要好得多，好多的 VC 书籍入手点往往是“例程”，有时候你真的去买一本书，这本书整个从头到尾会在教你每一个例子，当然这些例子很是实用，但如果你不明白其中运行的机理，很可能你就仅仅变成了一个能将书本代码搬上电脑的机器，而不是一个能够真正用自己头脑去做程序的编程者了。

由此，我们这一讲的接下来的部分，我们不打算再讲新内容了，而是停下来与大家回顾我们 MFC 教学以来的一些知识点。

·知识点回顾

·类　请别忘了，每一个类后面都要有一个分号

·类中有默认的构造函数、析构函数、拷贝构造函数

·类中默认地为我们重载了等号运算符“=”

·类中可以定义静态变量，但静态变量定义时不能赋值，并且只能而且一定要在类外对其进行初始化

·类中可以定义静态函数，但静态函数不可以使用类中的非静态变量

·类中的静态函数与静态变量不属于任何对象，类只要定义了，这些静态的东东就一直存在，直到程序结束

·类中的函数可以进行重载，但不能重名

·类中要调用全局函数，要使用全局标识符“::”

·类中有一个隐形指针，这个指针不需要定义，指针指向类本身，这个指针就是——this

·类　可以有子类

·类与子类之间有三种继承关系，公有，私有，保护

·当子类公有或保护继承父类时，父类中的公有或保护成员变量与函数，子类可以访问

·当子类私有继承父类或父类的函数是私有属性时，子类不能访问父类的对应成员变量或函数

·当子类公有继承父类时，父类的公有变量或成员函数，定义的对象可以使用，否则对象中不得使用

·当子类与父类有同名函数时，遵循同名覆盖的原则，父类的相应函数将得不到执行

·当子类有父类的同名函数，但父类中有相应函数的重载时，父类的重载函数得不到执行，会报错，需要想使用父类，请用 using 关键字

·当定义一个父类指针时，可以指向子类，但定义的子类指针不可以指向父类，但如果想指向父类，可使用强制转换的办法，但结果就是，转换后的类不可以再调用子类函数中定义的数据或函数。

·当定义一个父类指针时，如果需要动态地调用子类的函数，则需要在父类的函数名前加上　virtual　使其变成虚函数，使用迟绑定技术去调用子类的函数，　MFC　中的大多数技术就是使用这种技术，去实现对父类创建的窗口，子窗口，控件等东西进行改变的。

·当定义一个虚函数后面加上 =0　后，这个虚函数就不能在本类中定义，而必须在子类中定义，这种函数我们称之为“纯虚函数”，有纯虚函数的类，我们称之为“抽象类”，抽象类不能定义一个对象，只能做老爸，哎，老爸难当啊。

·正因为使用了虚函数的概念，所以可以在子类与父类之间插入一个中间的类，去继承儿子的父类，去当父亲的子类的类，从而实现“插足”，从而架起来座联系父子的桥，从而去拆掉 MFC　，实现我们自己的邪恶目的。

·贴图　贴图有两种方案，一种通过刷子去刷，一种通过常用的贴图技术

·刷子方案

·定义并选入一幅图片

·定义一个刷子，并且用图片去定义这个刷子

·使用刷子

·删除刷子与图片资源

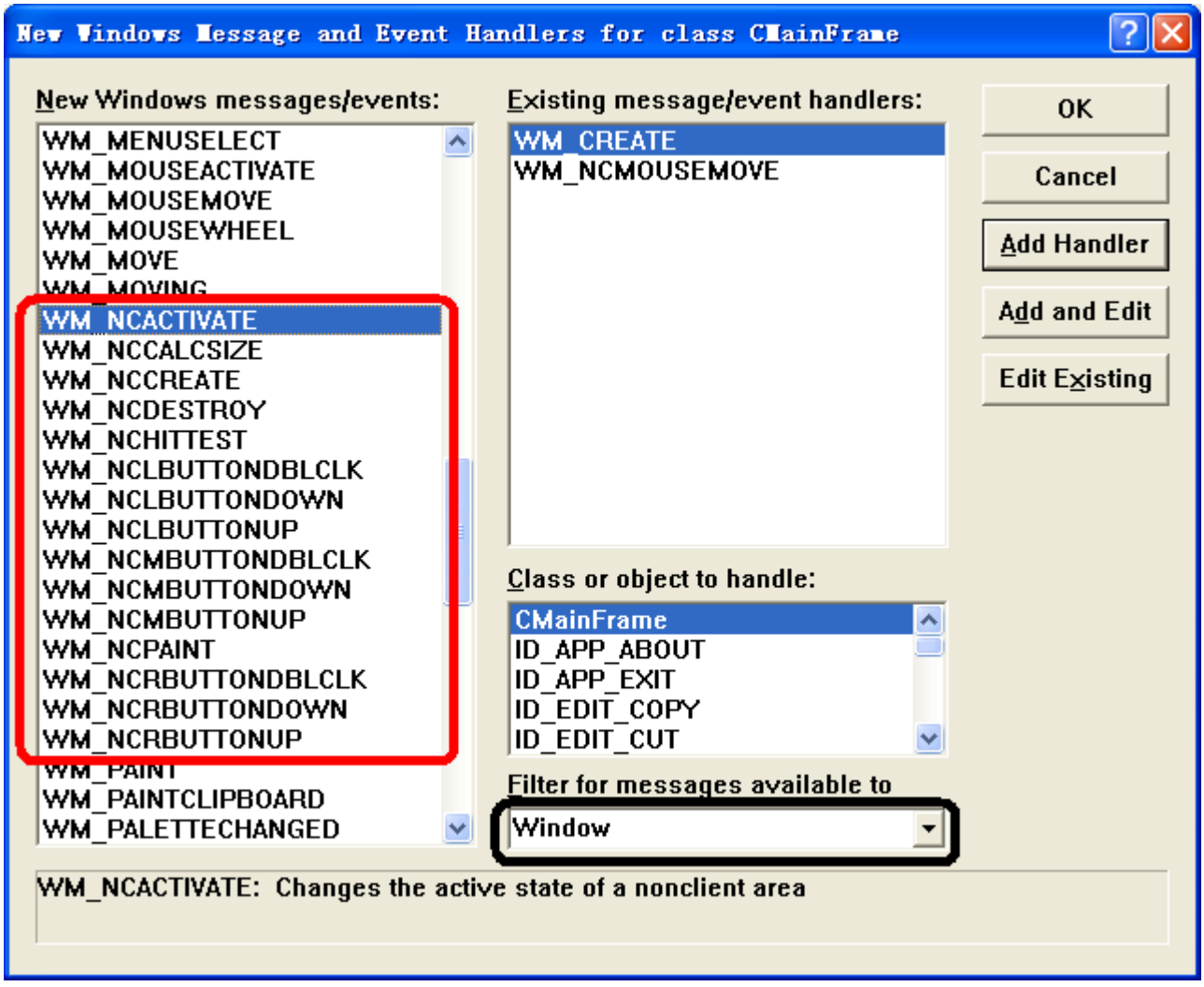
·贴图方案
  ·定义并迁往一幅图片
  ·定义一个兼容 CD，并且选入图片
  ·使用兼容 CD，用 BitBlt 等函数进行贴图
  ·删除兼容 CD 与图片资源

**微软积木〇九一：面目全非**

如果你能看到这儿，那么你应该很相信你自己的能力，你完全可以使得你的窗口面目全非，怎么办？

其实我们只要响应一些特殊的消息，我们很容易地，甚至“插足”的办法都不要用，就可以直接在标题栏、边框上面贴图，什么消息呢？Windows 消息，呵呵，肯定是 Windows 啦，可怪就怪在 MFC 给我们的默认消息并不是 Windows 的，呵呵。

好了，由于我们需要对标题栏等进行一些操作，但根据我们前面的内容我们可应该知道，标题栏应该属于 MainFrame 类的，也就是说，大家去响应 MFC 类，当然喽，大家在类上右键菜单 就行：

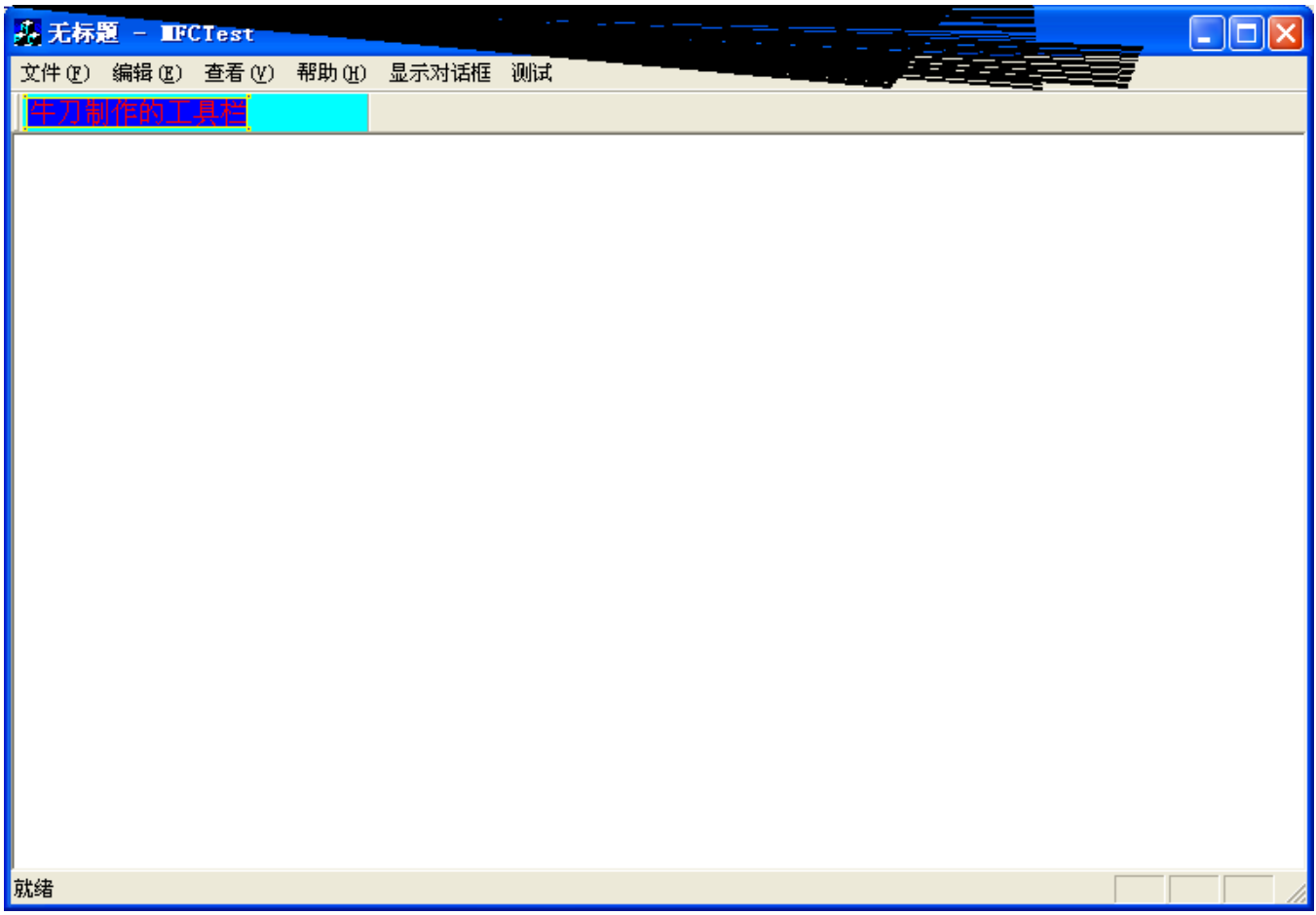



黑框中框起来的选择 Windows 就可以看到红框选起来的一些消息喽，前面都带有 NC 表示非客户区，也就是边框啊，标题啊什么的，呵呵，大家看后面的一些东东，很容易猜出一些意思来，不信，大家还是使用 WM_NCMOUSEMOVE 来响应一个函数看看：

```
void CMainFrame::OnNcMouseMove(UINT nHitTest, CPoint point)
{
```

```
CFrameWnd::OnNcMouseMove(nHitTest, point);
CDC *pDC=GetWindowDC();
CRect rect;
GetWindowRect(&rect);
point.x-=rect.left;
point.y-=rect.top;
pDC->LineTo(point);
ReleaseDC(pDC);
}
```

大家看效果：




以前牛刀也曾经做过这个程序，但当时没有给代码，其实就是这个效果，呵呵（对了，这一讲我们用 png 格式的图片文件，感觉怎么样？ gif 格式的效果太差了，呵呵）

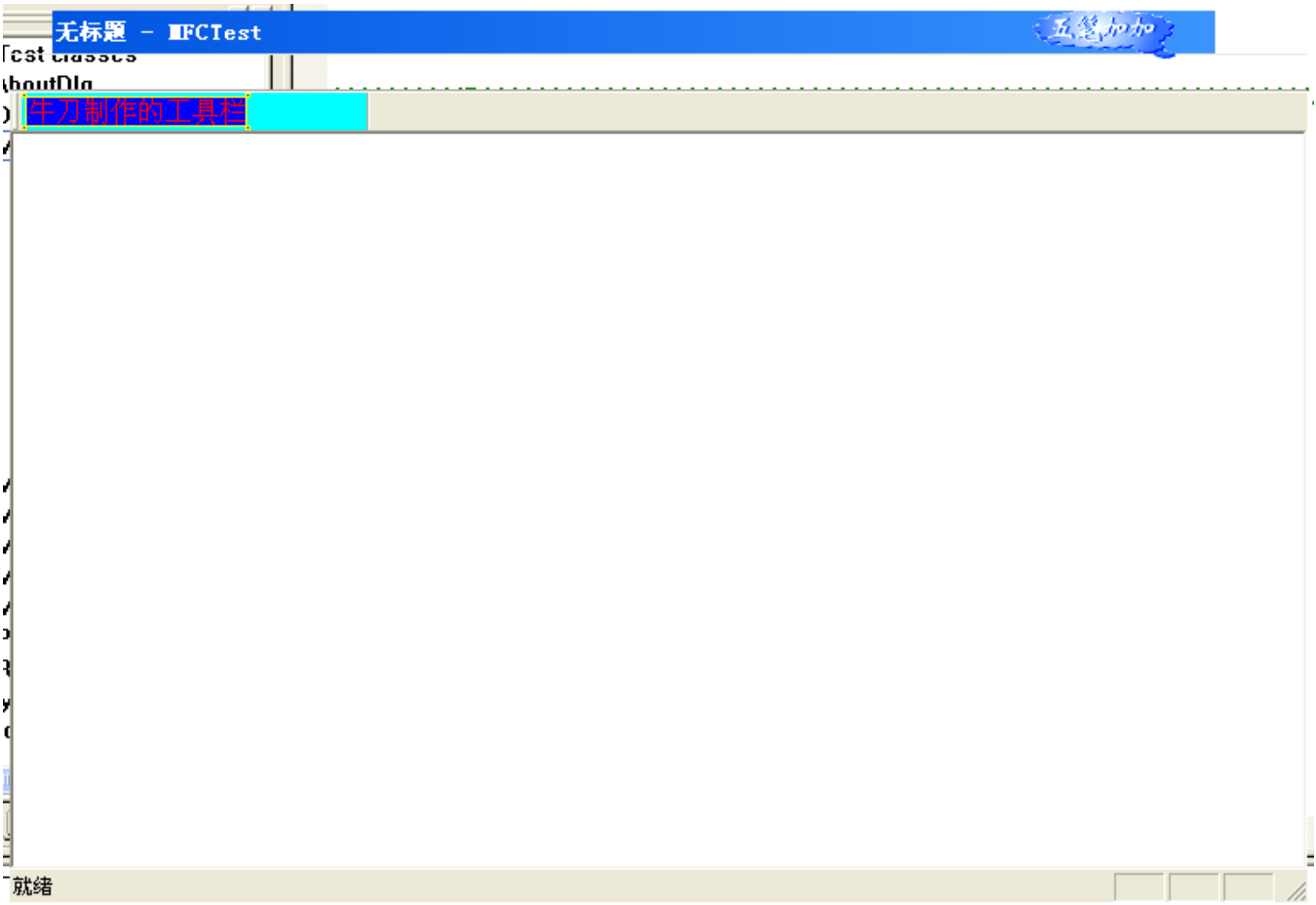
既然可以做出类似的效果，那么如果去贴图当然也是一个不错的主意了，于是，我们就需要对那一个一个的消息至少有一点了解，大伙说是也不是？

```
void CMainFrame::OnNcPaint()
{
CPaintDC dc(this);
CRect rect;
GetWindowRect(rect);
```

```
dc.MoveTo(rect.left,rect.top);
dc.LineTo(rect.right,rect.bottom);
}
```

如果大家真的这样制作的话，那么打开我们程序时，会见到线一闪，又没了，也就是说，并没有像我们想像的那样，在标题栏中画一道线。

那么肯定有其他的消息在响应 WM_NCPAINT 消息之后，又重新响应了一下，并且进行重绘，当然，就是绘制了标题栏部分：




大家可以看到，标题栏、菜单、边框等部位其实都是透明的。

但是大家可以使这个窗口隐藏一下，再显示出来，就正常了，大家可以点一下关闭按钮，会发现出现了一个老式的小叉，这一切的一切在说明，系统肯定有个 NC 消息给我们将这些东西给做了，那么是什么消息呢？

复杂透了，好多消息都会引起一些标题栏的重画，例如：

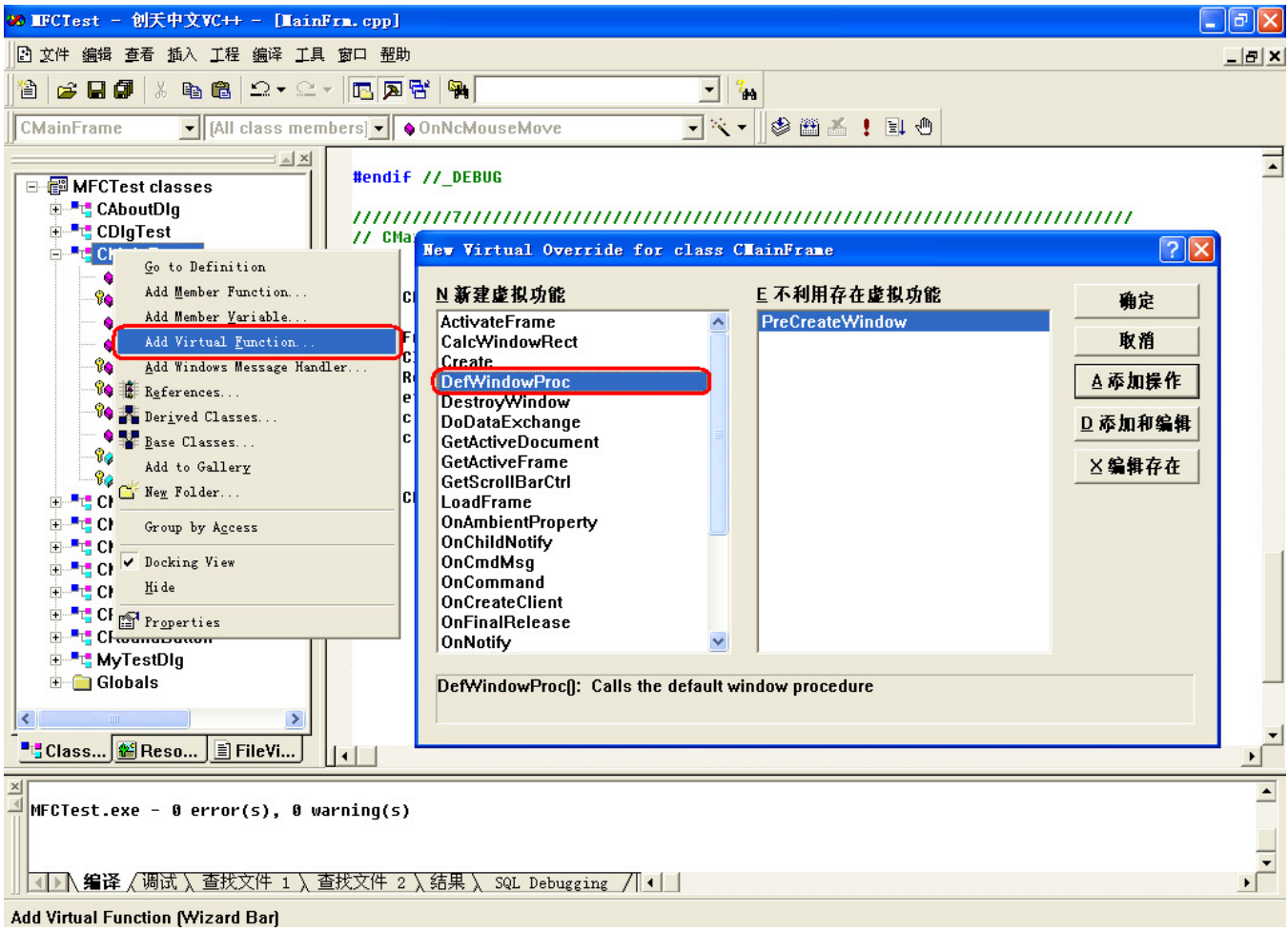
WM_MOVE
WM_PAINT
WM_NCPAINT
WM_NOTIFY
WM_NCACTIVATE

等等，当然，一个办法就是响应每一个消息，另一个办法就是——打劫（呵呵，这事咱们干多了）。

当然，你找得到消息循环的入口吗？在那儿打劫当然可以，这次我们打劫系

统默认消息处理函数：

那当然得加入虚函数了，因为这个函数不是消息哟，好在老爸为我们准备了这个虚函数，如下的加入办法：




```
LRESULT CMainFrame::DefWindowProc(UINT message, WPARAM wParam, LPARAM
lParam)
{
// TODO: Add your specialized code here and/or call the base class

return CFrameWnd::DefWindowProc(message, wParam, lParam);
}
```

你瞧，这儿调用的是父类的 DefWindowProc ，这可为我们创造了极好的机会，去“打劫”啊，呵呵，看看牛刀怎么打的，这次咱们不用 switch 语句了，咱们用 if 语句：

```
LRESULT CMainFrame::DefWindowProc(UINT message, WPARAM wParam, LPARAM
lParam)
{
if(message== WM_PAINT || message==WM_NCPAINT || message==WM_MOVE ||
message==WM_NCACTIVATE || message==WM_NOTIFY)
{
  //干坏事的代码
```

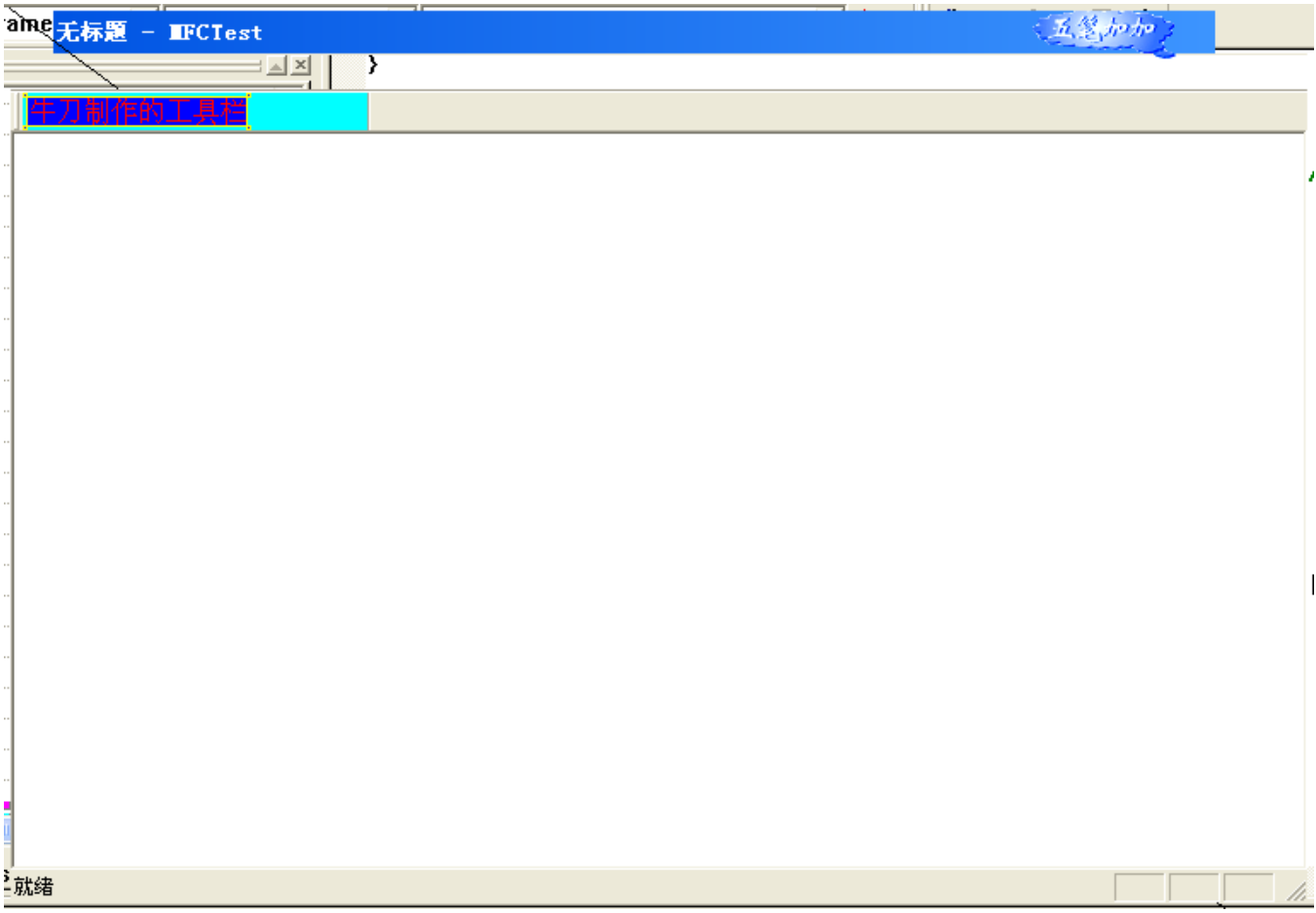
```
}
else
{
  return CFrameWnd::DefWindowProc(message, wParam, lParam);
}
}
```

请大家注意，这里代码不补全的话，可能会出现一些错误，就是如果是这些消息的话，就会没有返回值，所以我们对这些消息处理完毕后，建议大家再返回一个值，让父亲去处理，当然也可以不理那老头子，呵呵（但不理出问题牛刀可不负责任哟，呵呵。）建议大家还是用以下结构：

```
LRESULT CMainFrame::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
if(message== WM_PAINT || message==WM_NCPAINT || message==WM_MOVE || message==WM_NCACTIVATE || message==WM_NOTIFY)
{
  //干坏事的代码
}
return CFrameWnd::DefWindowProc(message, wParam, lParam);
}
```

当然，这里干什么，那就不由得系统要怎么处理了，呵呵，我们将前面的代码移过来：

```
LRESULT CMainFrame::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
if(message== WM_PAINT || message==WM_NCPAINT || message==WM_MOVE || message==WM_NCACTIVATE || message==WM_NOTIFY)
{
  CDC *pDC=GetWindowDC();
  CRect rect;
  GetWindowRect(&rect);
  pDC->LineTo(rect.right,rect.bottom);
}
return CFrameWnd::DefWindowProc(message, wParam, lParam);
}
```



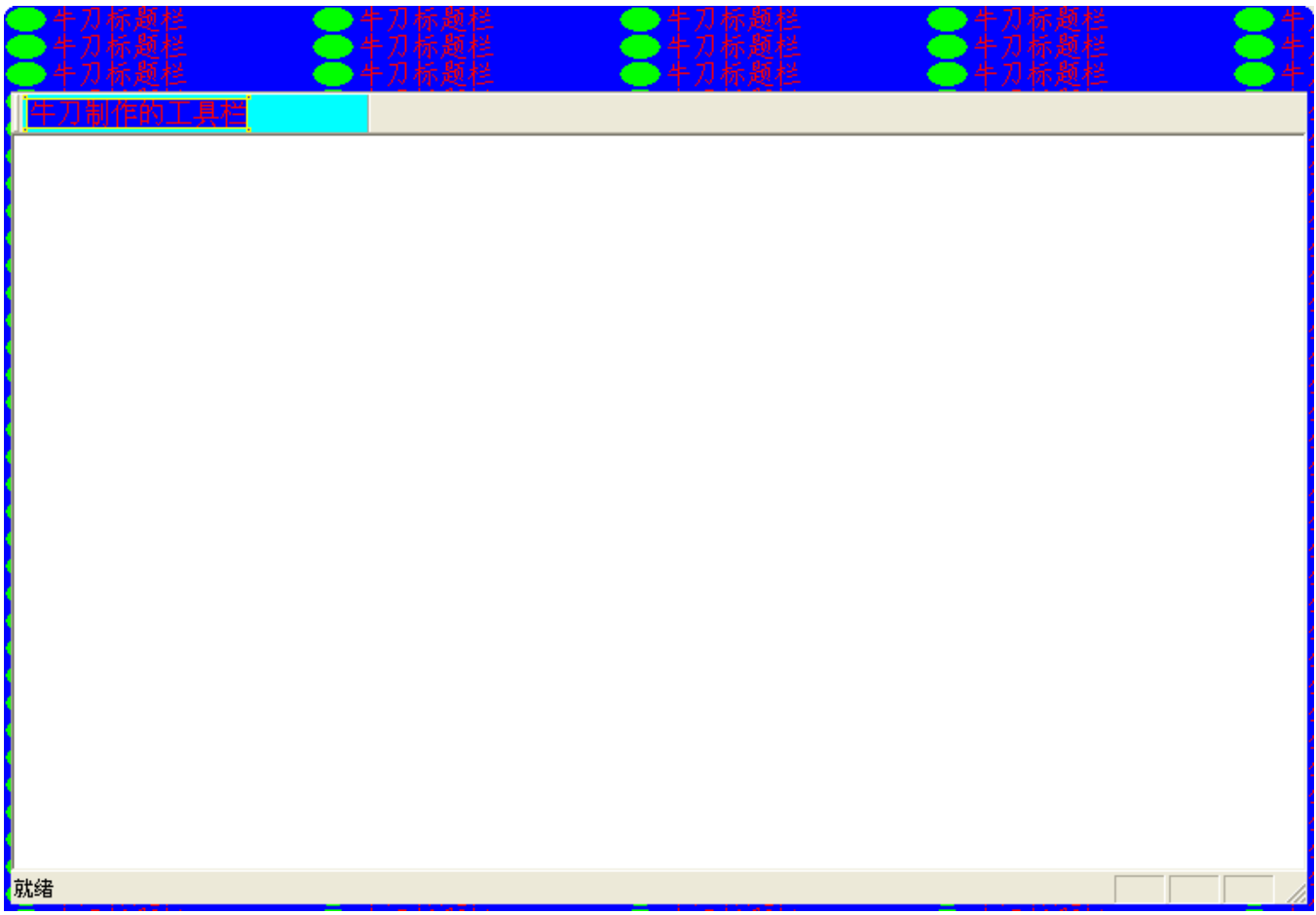

这次，大家看见那一根线条了吧？但可惜的是，我们只看到了一部分的线条，而另一部分被 View 类的其他消息所覆盖了。

既然该透明的地方也透明了，该有线条的地方也有线条了，那么我们接下来就可以顺利地向我们的标题栏与边框上贴图了：

```
LRESULT CMainFrame::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
if(message== WM_PAINT || message==WM_NCPAINT || message==WM_MOVE || message==WM_NCACTIVATE || message==WM_NOTIFY)
{
  CDC *pDC=GetWindowDC();
  CRect rect;
  GetWindowRect(&rect);
  rect.right-=rect.left;
  rect.bottom-=rect.top;
  rect.left=rect.top=0;
  CBitmap bmp;
  bmp.LoadBitmap(IDB_TITL);
  CBrush brush;
  brush.CreatePatternBrush(&bmp);
```

```
  pDC->FillRect(&rect,&brush);
}
return CFrameWnd::DefWindowProc(message, wParam, lParam);
}
```

大家看效果：




当然，大家图片出来后，不能够再次出现上面的一些打劫的消息，如果再出现的话，那么我们的 View 类中的消息与工具栏、状态栏中就全部被覆盖，原因很简单，我们的这种消息打劫机制不能引起他们子窗体的重画。

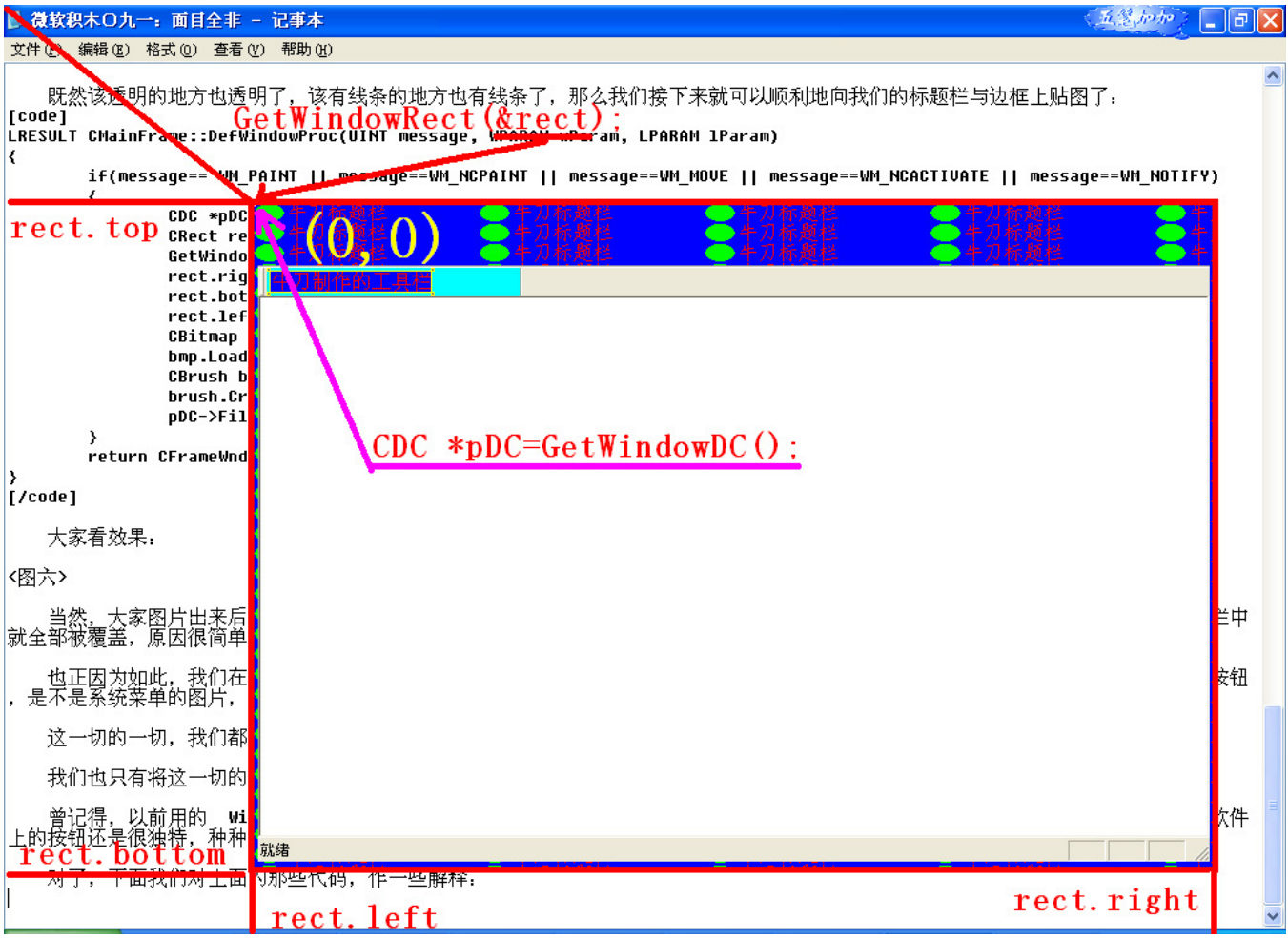
也正因为如此，我们在处理这些标题栏、边框的时候要特别地注意，千万一块一块测试，看看当前位置是不是标题栏，是不是最大化按钮，是不是系统菜单的图片，是不是最小化或关闭按钮，是不是边框，当前最大化按钮处于什么状态.....

这一切的一切，我们都需要去考虑。

我们也只有将这一切的一切都考虑全了，才能够做出真正的属于我们自己的，面目全非的窗体来效果非常好。

曾记得，以前用的 Win98 的时候，有些软件上的按钮就已经做得很像 WindowsXp 按钮了，当时看上去非常地漂亮，现在看有些软件上的按钮还是很独特，种种原因，牛刀这里就给大家贴图了，大家可以自由地发挥自己的思路，去做出我们自己所需要的漂亮界面。

对了，下面我们对上面的那些代码，作一些解释：




.

GetWindowRect(&rect);

由于此函数返回的值是以屏幕坐标为参照系的，所以其 rect.left 并不等于 0，rect.top 当然也不会为零。

而我们使用：

GetWindowDC 取得的是窗口的句柄，这个句柄包括标题栏，所以其 (0,0) 点在标题栏的左上角，所以我们为了在这个窗体中绘图，那么就必须使用

rect.right-=rect.left;

rect.bottom-=rect.top;

rect.left=rect.top=0;

去调节这几个数据，从而使得我们对窗口的标题栏贴图时，能够正确贴图。

而我们过去在客户区贴图时，就没有这个问题，因为：

GetClientRect 返回的就是客户区的矩形，就是以客户区为坐标的，所以不存在这个问题。

上面我们是讲了讲我们非客户区贴图的一些原理，当然，你一旦弄明白了，你就可以贴出自己的很好看的界面出来，当然，如果你真的对足够多的消息响应了，并且原理知道了，你可以在你的非客户区画出自己的最大化最小化按钮，与其他的一些东东，其实都很容易实现，那些东东具体怎么实现，我们以后再与大详聊。

既然我们这一讲内容叫——《面目全非》那么非客户区已经被我们拿下来了，那么客户区怎么办呢？咱们使用响应 WM_CTRCOLOR ：

```
HBRUSH CDlgTest::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

// TODO: Change any attributes of the DC here

// TODO: Return a different brush if the default is not desired
return hbr;
}
```

这个消息其实很容易实现，传递一个画刷给老爸就行了，这儿的儿子当然只是调用了一下老爸，然后传递一个老爸给的画刷，你瞧多会“借花献佛？

我们自己来创建一个：

```
HBRUSH CDlgTest::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
HBITMAP bmp;
bmp=LoadBitmap(AfxGetApp()->m_hInstance,MAKEINTRESOURCE(IDB_MYBACK));
HBRUSH hbr;
hbr=CreatePatternBrush(bmp);
return hbr;
}
```

这一个程序的显示效果也挺好，呵呵：

图片效果很好，可是大家看，静态文本框是牛刀后画上去的，上面除了字的部分，其他地方也被用画刷给贴上了，这是显然的。也就是说，我们控件中的好

多地方也被用画刷给贴图了，可为什么我们的按钮就没有被贴呢？

原因啊，很简单，因为按钮是一种自绘类型的控件，什么意思呢，意思就是说，我们的按钮并不是一成不变地放在那儿，而是随着我们的点击，按钮需要变化状态，这一点不要说我们自己可以做软件，就是用惯了电脑也知道了，按钮点下与弹起感觉是不一样的，也就是说，其实用的是两张甚至更多张的图片，从而实现鼠标点击的按下的效果，这种可以随着不同的时间或者响应不同消息时，有不同的表现的按钮，牛刀称之为“自绘”控件，对于这类控件，我们想要改变其背景，那唯一的办法就是——父子插足。

当然，那种技术，我们早就与大家讨论过了，呵呵。

到此为止，我们就可以完全玩转我们手中的窗体了，让窗体去实现我们的想法。当然，接下来的时间，我们与大伙将上面一段程序代码讨论一下：

```
HBITMAP bmp;
bmp=LoadBitmap(AfxGetApp()->m_hInstance,MAKEINTRESOURCE(IDB_MYBACK));
```

定义一个位图句柄，并选入位图

```
HBRUSH hbr;
hbr=CreatePatternBrush(bmp);
```

用位图创建一个画刷句柄

我们一直在 API 那里，没详细与大伙说这种 API 的位图与画刷，因为我们以后有一套 MFC 的技术，但有时候，我们还就真的不得不使用 API ，此时，我们就需要用全局的 API 函数了，但我们这儿为什么没有用“::”全局标志呢？因为我们的 CDlgTest 中并没有相应的函数，而相应的函数全在 CDC 类中，而我们这没有 CDC 的对象，那就必然调用全局的 API 函数了。

全局的 LoadBitmap 需要两个参数，一个参数是程序句柄，一个是位图资源，这儿我们知道，我们整个 MFC 程序，并没有显式地调用 WinMain 函数，所以我们的程序句柄直接拿是拿不到的，好了 App 那里为我们保存了一份，于是，我们就可以用 AfxGetApp 函数去得到 theApp 的指针，通过指针取得句柄就一切 OK 了。

这里我们还有一个就是 MAKEINTRESOURCE 宏，我们知道，资源其实在程序中是一个整数，而我们需要让程序知道的就是资源的位置，所以就需要一个宏，将我们的整数转换成那个位置，这就是这个宏的作用，说实话， DOS 时代，我们不需要这个宏，因为我们那时可以直接操作每一个像素，MFC 的时代，我们不需要这个宏，因为这个宏被封装起来了，呵呵，所以说，经常用用 API 函数，可以使得我们编程水平更上一个新台阶。

至于后面那个画刷句柄，那就是一个函数的问题了，那一点问题都没有。

最后与大家说一说这个 Afx 的问题，这个打头的所有东东都是 VC 初期的遗留，就好像 LPARAM WPARAM 一样，都是历史的遗留，但现在要告诉大家的一个事实就是，只要以 Afx 打头的函数，就是一个全局函数，我们在任何一个地方都可以去调用，事实就是这样，用起来很是方便，特别是需要用到一些句柄之类的东东，那简直是方便极了。

## 微软积木〇九二：编辑控件

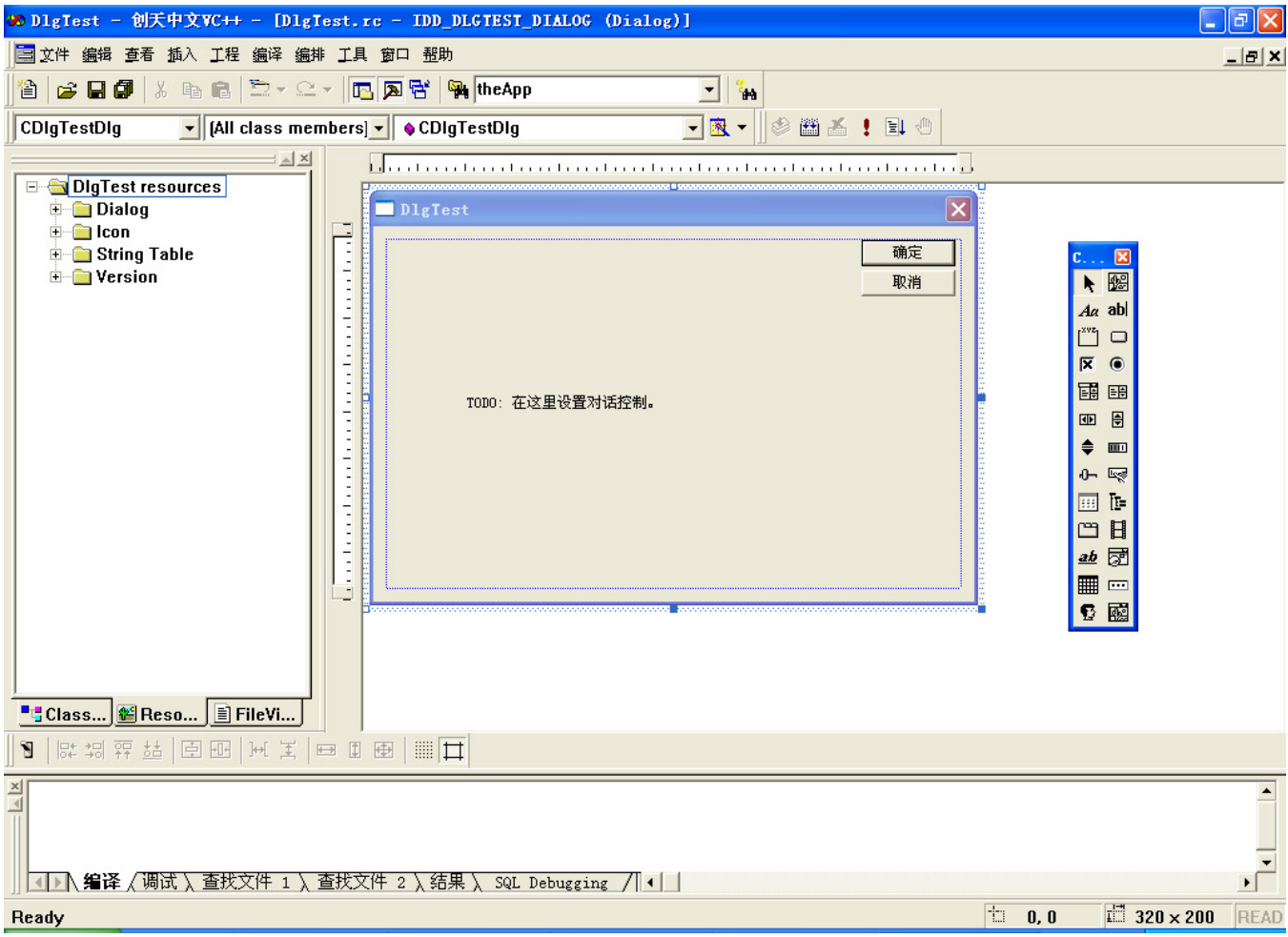
与大家从类的角度，从父子插足的角度详细地介绍了有关 MFC 的构成，从 API 一直到对话框、菜单、工具栏、状态栏与非客户区域。

这一讲开始以及接下来几讲，我们从一个个控件的角度，去横向地观察一下，我们的控件是如何在我们的程序之中发挥作用的。

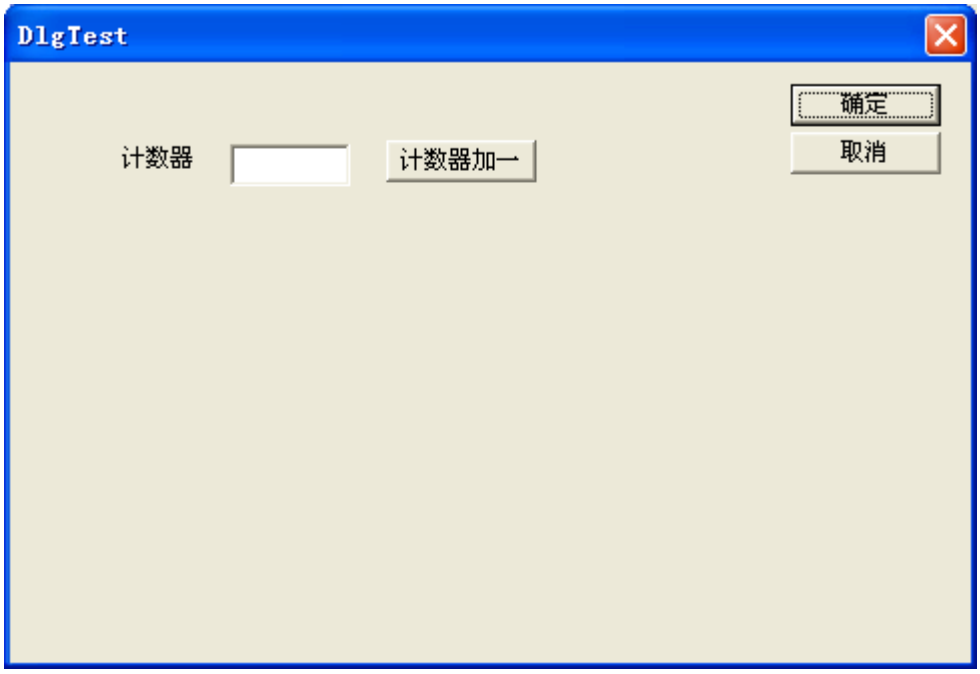
我们当然可以从一个单文档的程序中去 DoModal 一个对话框，可我们没有必要，我们这一讲与大家直接建立一个对话框工程。

当然是 文件/新建/[工程]/MFC AppWizard(exe)

名称就叫 Dlgtest 吧，在第一步选择第三个，对话框，选择“完成”，我们就做好了一个对话框了。

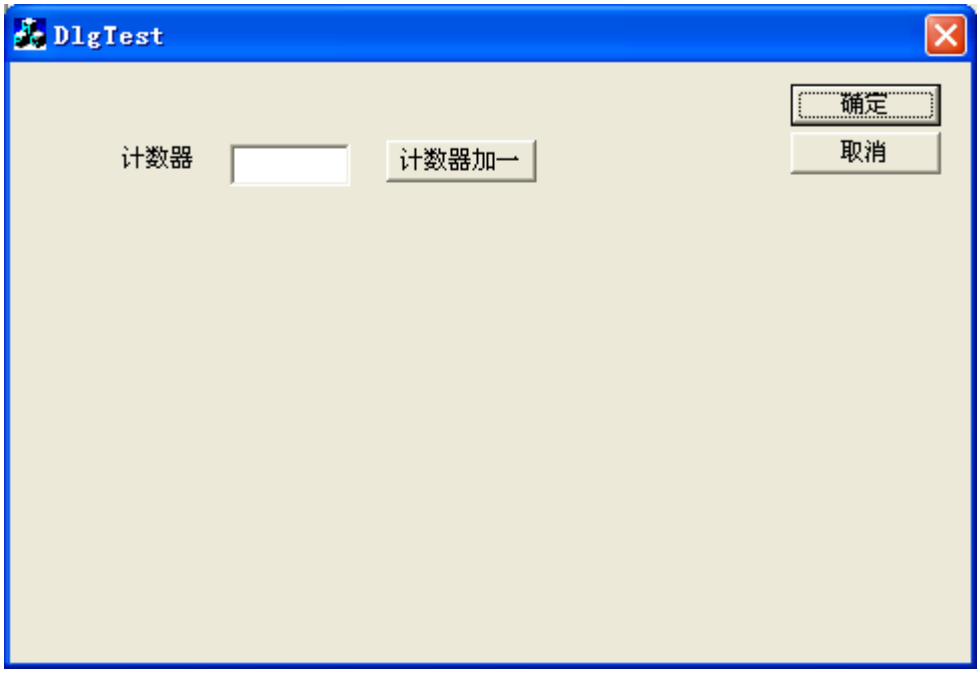



大家看上面，告诉大家可以在什么位置设置对话框，我们画几个控件：




大家像上面的图那样画几个控件，一个静态文本，标题改为“计数器”，其他什么东东都不要改。一个编辑框， ID 号为 IDC_EDIT_INC ，一个按钮，ID 号为 IDC_BTN_INC ，标题为：“计数器加一”

牛刀设置这几个东东，可以说是什么目的，大伙都知道了吧，呵呵，我们就是想将按钮一按，前面文本框中的内容加一，就这样简单，大家这就可以编译一下程序了：

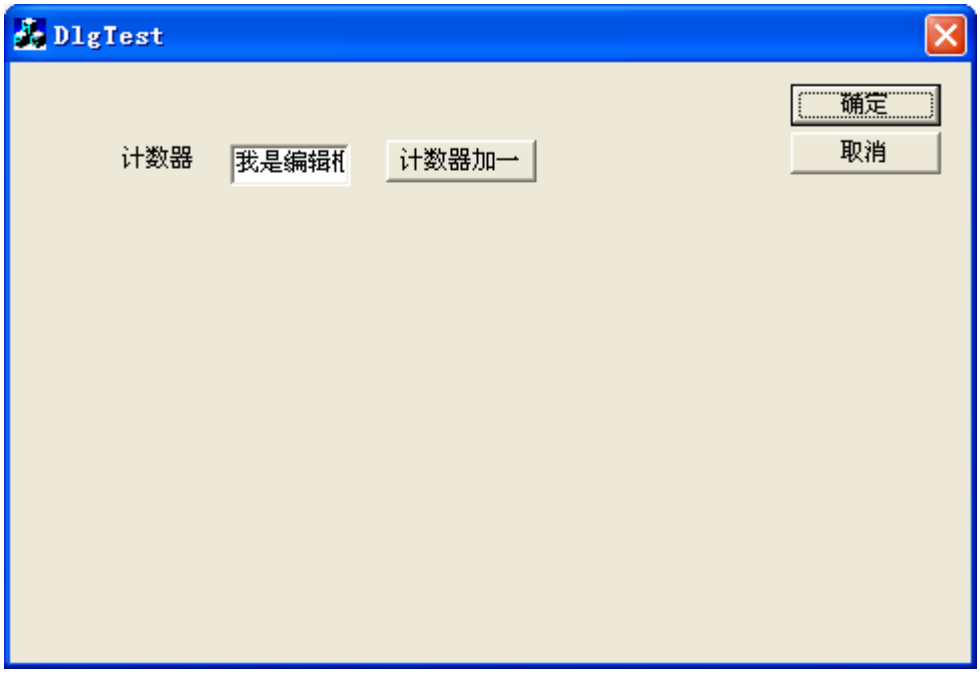



以上是程序运行的结果，显然，我们是不满意的，我们希望一运行，编辑框中的内容就为 1 ，那样才好，呵呵。

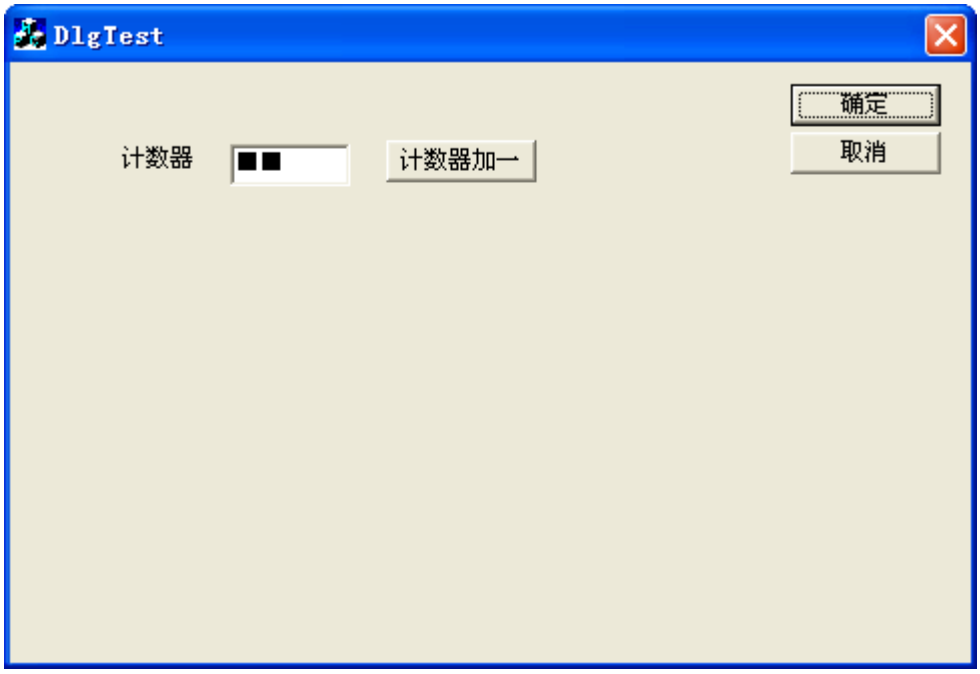
那很简单啊，我们先来看看我们程序的几个类吧：

CAboutDlg

CDlgTestApp
CDlgTestDlg
很是简洁，就这三个类，当然是 App 管着 Dlg，Dlg 管着 About 了，呵呵
CDlgTestDlg 中有一个初始化的函数叫 OnInitDialog 就是初始化我们的对话框的，我们可以在这里设置这个对话框中编辑框的标题：
GetDlgItem(IDC_EDIT_INC)->SetWindowText("我是编辑框");
运行效果如下：



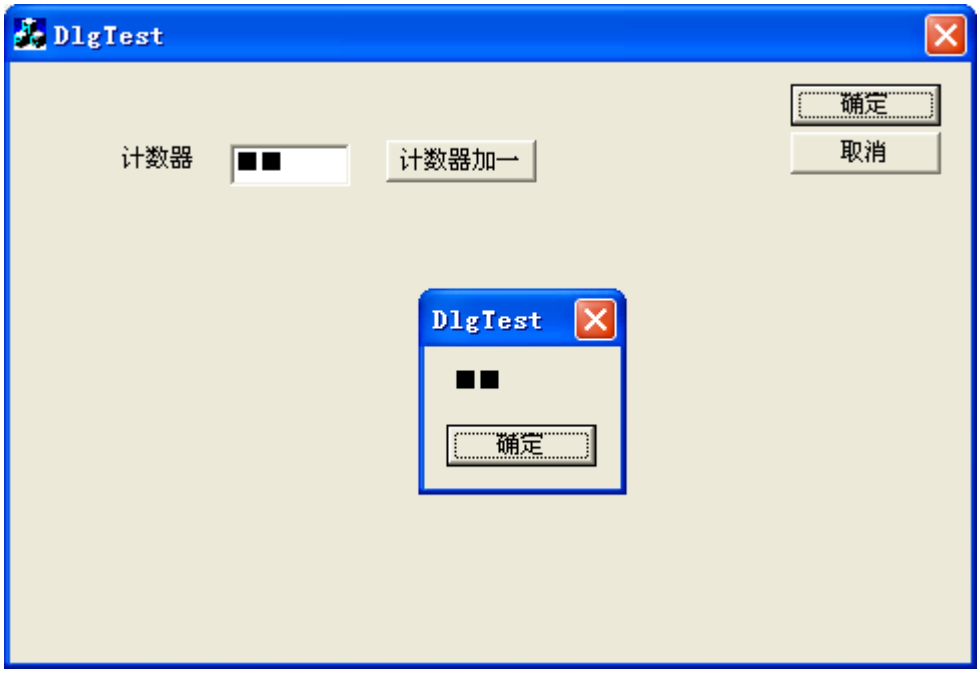

怎么样？效果很是不错的。
当然，我们可以很简单地将上面的内容改成任何的文字，例如：
GetDlgItem(IDC_EDIT_INC)->SetWindowText("■■");
怪怪的是吧，呵呵，那么我们的显示当然也是怪怪的：




好了，怪就怪吧，咱们就怪怪地用，呵呵，下面，我们双击我们加进的按钮，就可以用另一个函数去获得这个文一的字符串，然后再将其长度加 1，然后再截取某段字符串中的相应长度，嗯～～，这样说是不是感觉还不知牛刀在干什么？先看下面的程序：

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
str1="■■■■■";
GetDlgItem(IDC_EDIT_INC)->GetWindowText(str2);
MessageBox(str2);
}
```

大家就先别管　str1，我们看程序运行结果：



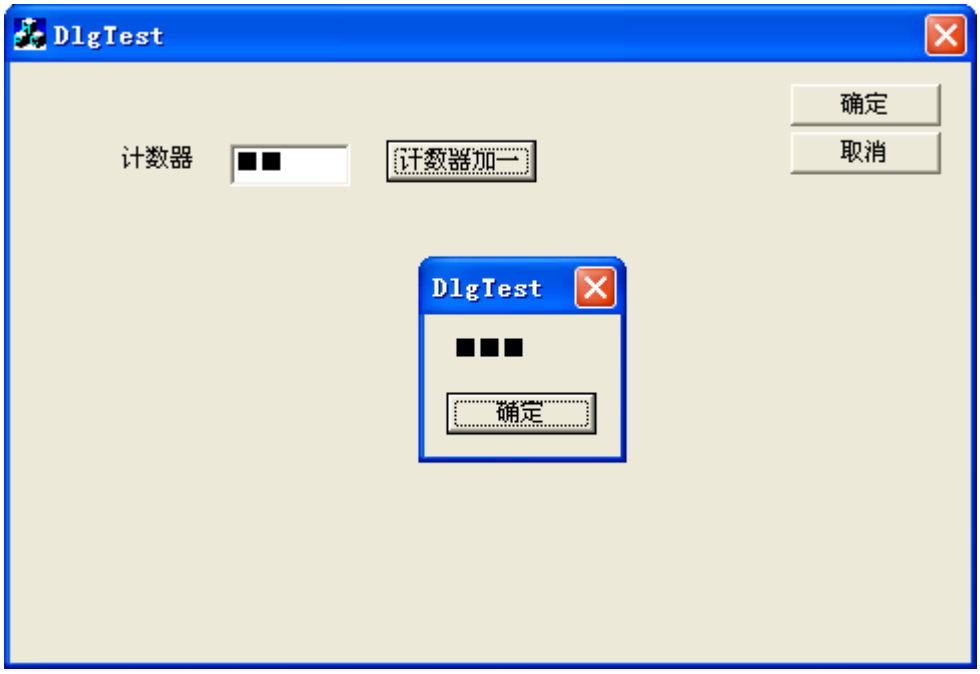

上面的图证明我们的确取到了文本框中的数据了。

既然取到了，那么就可以得到其长度，然后将长度加 1，再从 str1 的字符串中截取一段这个长度的字符串（对了，加 1 再余 5，这样就不至于一直变长了，呵呵，一直变长没意思）。

先看程序吧：

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
str1="■■■■■";
GetDlgItem(IDC_EDIT_INC)->GetWindowText(str2);
int i=str2.GetLength(); //类好用吧，呵呵
i=(i+2)%10; //我们中文汉字是两个字节
str2=str1.Left(i);
MessageBox(str2);
}
```

看下图：



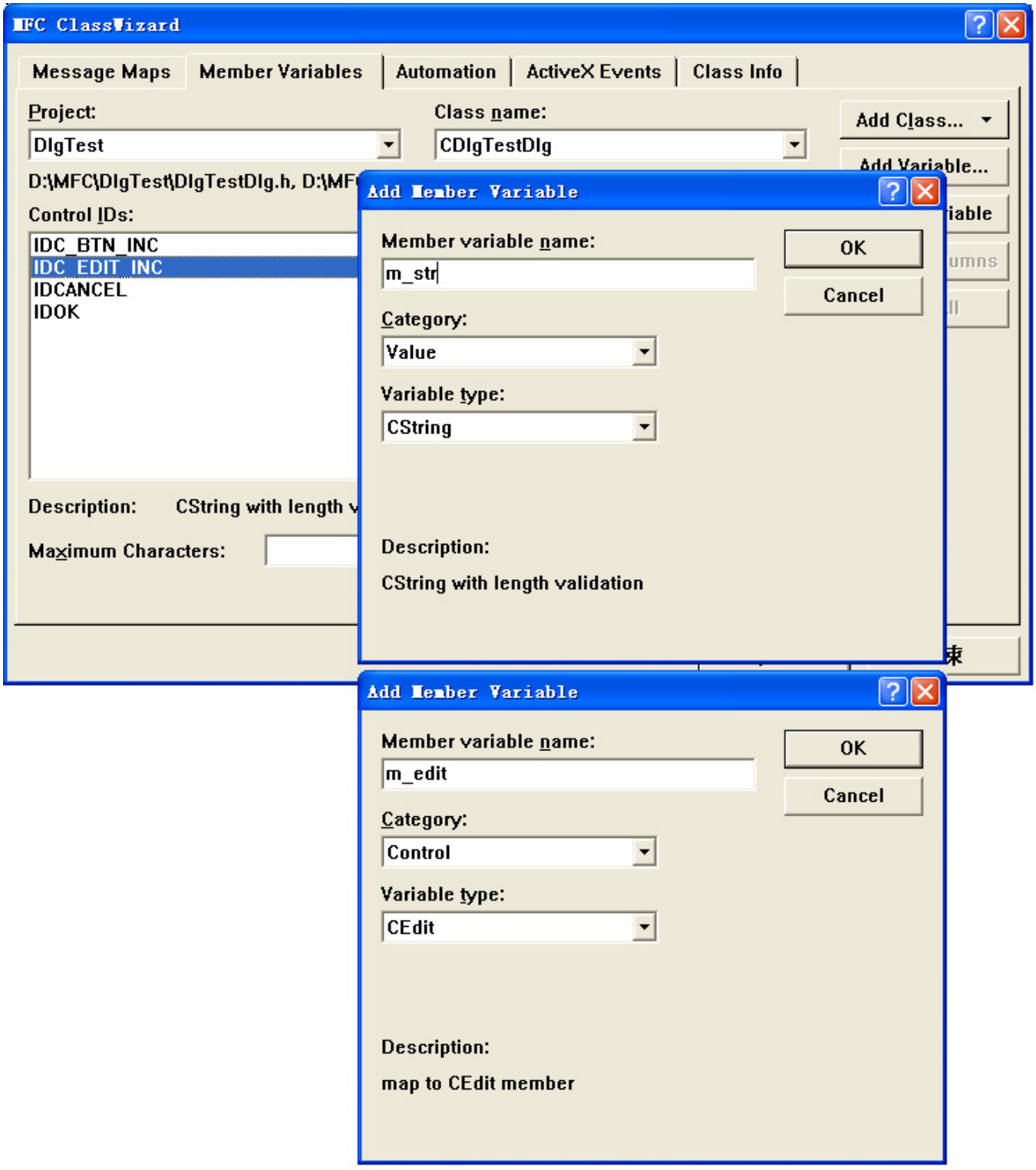

我们的试验又成功了，那么接下来，我们就要将字符串再写回编辑框：

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
str1="■■■■■";
GetDlgItem(IDC_EDIT_INC)->GetWindowText(str2);
int i=str2.GetLength(); //类好用吧，呵呵
i=(i+2)%10; //我们中文汉字是两个字节
str2=str1.Left(i);
GetDlgItem(IDC_EDIT_INC)->SetWindowText(str2);
}
```

好了，大家连续点那个按钮，看看什么效果？

是不是感觉有点像进度条了，呵呵，拉长一点，再加一点长度，还真像呢，我不是说过嘛，其实所有的控件之间没有什么本质的区别，其实都是一个窗体，以后我们大家自己也可以改变甚至定义自己的控件。

当然，我们还会有一些办法来取得编辑框中的内容，一种办法当然是在“建立类向导”中的“Member Variables”选项卡上，为 IDC_EDIT_INC 绑定一个变量，这里我们绑定两个一个是控件，另一个是 CString 类型：一个是控件类型，类型为 CEdit ：




CEdit m_edit;

CString m_str;

好了，这个成员变量就直接代表了我们控件或控件中的内容，那么我们编程就相应地简便地多了：

```
void CDlgTestDlg::OnBtnInc()
```

```
{
CString str;
str="■■■■■";
MessageBox(m_str);
}
```

非常可惜，我们没有拿到任何的字符，大家可以将程序做上电脑看效果，牛刀不贴图了，呵呵，图再贴多了没意思，牛刀这既然是实践的教程，那么结果都让你看到了，那岂不是很不好玩？

什么原因呢？大家还记得我们以前讲过，有一个函数 DoDataExchang 这个函数吗？这个函数是负责数据交换的，必须调用这个函数后，才可以将数据从控件中交换出来，或交换进控件：

```
void CDlgTestDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CDlgTestDlg)
DDX_Control(pDX, IDC_EDIT_INC, m_edit);
DDX_Text(pDX, IDC_EDIT_INC, m_str);
//}}AFX_DATA_MAP
}
```

对了，大家看上面的注释，好玩啊，前面分别带了两组大括号，什么意思，真是怪怪的。

原来啊，这对大括号就是让我们的“建立类向导”看的，于是，带出了这怪怪的大括号，其实大家移出来也没事，删掉也没事，就是移出来或删掉，我们以后用“建立类向导”，就没办法管理了，仅此而已。

可惜的是，上面定义的仅仅是宏，所以我们一般不调用这个函数的，而是调用父类的 UpdateData 函数，传进真则控件向变量中传数据，传进假则数据向控件中传送数据，于是，我们这样一改，就一切 OK 了：

```
void CDlgTestDlg::OnBtnInc()
{
CString str;
str="■■■■■";
UpdateData(TRUE);
MessageBox(m_str);
}
```

当然，我们改完后，也要调用这个函数，但那时就应该传假了，很简单，看程序：

```
void CDlgTestDlg::OnBtnInc()
{
CString str;
str="■■■■■";
UpdateData(TRUE);
int i=m_str.GetLength();
i=(i+2)%10;
m_str=str.Left(i);
```

```
UpdateData(FALSE); //大家先删掉看效果
}
```

是不是感觉比上面的要简单得多呢？

当然，我们下面就再用控件看一看效果：

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
char ch[10]="123456";
str1="■■■■■";
m_edit.GetWindowText(str2);
int i=str2.GetLength();
i=(i+2)%10;
str2=str1.Left(i);
m_edit.SetWindowText(str2);
}
```

好玩的是这儿就没有必要用 UpdateData 函数了，大家思考为什么，牛刀调了好多次，一直效果不好，后来才恍然大悟，呵呵，有时候有些事去做与不做效果真的是不同的，大大的不同啊。

当然，还有一个函数 GetDlgItemText，可以直接替代上面的 GetDlgItem(IDC_EDIT_INC)->GetWindowText ，大家看程序：

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
str1="■■■■■";
GetDlgItemText(IDC_EDIT_INC,str2);
int i=str2.GetLength();
i=(i+2)%10;
str2=str1.Left(i);
SetDlgItemText(IDC_EDIT_INC,str2); //这也是上面相应的替代啊，呵呵
}
```

当然，想得到控件的数据或者设置数据，也可以做另一个方案，就是向控件发送消息。

我们知道，子控件其实也有消息循环，只不过已经被 Windows 给封装起来了，那么我们怎么去发送消息呢？

SendMessage

看程序，大家思考：

[code]

```
void CDlgTestDlg::OnBtnInc()
{
CString str1,str2;
char ch[10];
str1="■■■■■";
::SendMessage(GetDlgItem(IDC_EDIT_INC)->m_hWnd,WM_GETTEXT,10,(WPARAM)
ch);
```

```
// ::SendMessage(m_edit.m_hWnd,WM_GETTEXT,10,(WPARAM)ch);
// GetDlgItem(IDC_EDIT_INC)->SendMessage(WM_GETTEXT,10,(WPARAM)ch);
// m_edit.SendMessage(WM_GETTEXT,10,(WPARAM)ch);
str2.Format("%s",ch);
int i=str2.GetLength(); //类好用吧，呵呵
i=(i+2)%10; //我们中文汉字是两个字节
str2=str1.Left(i);
SetDlgItemText(IDC_EDIT_INC,str2);
}
```

我们注释起来的与没有注释的共有四种发送消息的办法，关于发送消息的方法与原理，我们以后有空专门再讲，这里大家知道可以这样处理就一切 OK 了。

当然，下面我们也可以用以上四种办法，去发送 WM_SETTEXT 消息，效果一样的。

当然，还有一个函数就是 SendDlgItemMessage 大家自己也去看看，这些函数其实就是多个函数的组合，大家对某一个函数明白了，其他的函数也就自然明白了。

好了，上面我们明白了如何使用控件的最基本的办法，那下面我们干什么呢？那既然要做那种“父子插足”之事喽，呵呵。

大家将定义的：

CEdit m_edit;

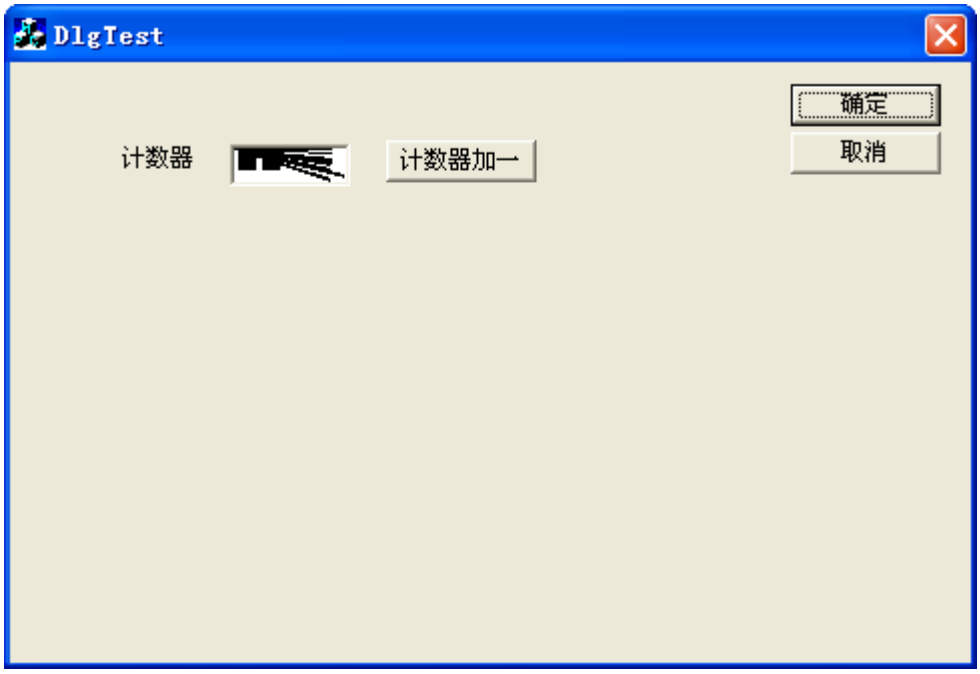
改为：

CMyEdit m_edit;

然后再加上一个类，以 CEdit 为父类：

当然可以老一套，去行使你的权力，为其加上一个鼠标移动的消息，然后去画线，呵呵，你也可以做出同样的效果来：

```
void CMyEdit::OnMouseMove(UINT nFlags, CPoint point)
{
CClientDC dc(this);
dc.LineTo(point);
CEdit::OnMouseMove(nFlags, point);
}
```




当然，你也可以在上面贴图，可惜的是，一旦你贴图，那么我们编辑框中的字就看不到了，那怎么办呢？

我们可以在字下面贴图，呵呵。

那么我们怎么在背影贴图呢？我们可以响应 WM_ERASEBKGND 消息：

```
BOOL CMyEdit::OnEraseBkgnd(CDC* pDC)
{
CBitmap bmp;
bmp.LoadBitmap(IDB_EDITTEST);
CBrush brush;
brush.CreatePatternBrush(&bmp);
CRect rect;
GetClientRect(&rect);
pDC->FillRect(&rect,&brush);
return TRUE;
}
```

贴上来了，一运行，一点效果都没有，为什么，因为啊，我们要在这里输出文本，而文本也是有背景的，于是，我们到父类中去响应 WM_CTLCOLOR

```
HBRUSH CDlgTestDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
if(pWnd->GetDlgCtrlID()!=IDC_EDIT_INC)
{
  return hbr;
}
else
{
  HBRUSH brush;
```

```
  brush=(HBRUSH)GetStockObject(NULL_BRUSH);
  pDC->SetBkMode(TRANSPARENT);
  pDC->SetTextColor(RGB(255,0,0));
  return brush;
}
}
```

当然喽，我们还要响应一下 EN_CHANGE 和鼠标的抬起消息，因为我们需要在编辑框改变时重画：

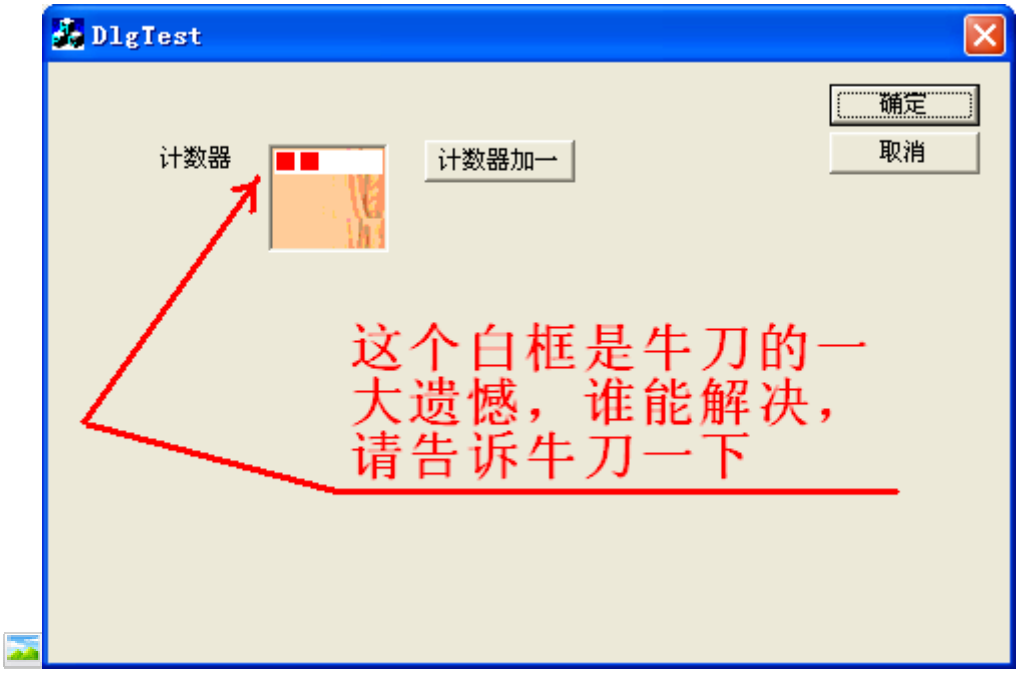
```
void CMyEdit::OnChange()
{
Invalidate();
}
void CMyEdit::OnLButtonUp(UINT nFlags, CPoint point)
{
Invalidate();
CEdit::OnLButtonUp(nFlags, point);
}
```

很可惜的是，牛刀怎么调都会有一个白色的方块围在周围，而没有办法去除，为什么 SetBkMode 在这里失效了？这至少对牛刀来说，目前还是个谜，还有一个，就是响应 WM_CTLCOLOR 直接在 CMyEdit 中去响应也会失效，当然，那可以想像应该是被父窗体的相应函数给重画了，所以原来牛刀有下面的代码，但没有用，不知为什么。

```
HBRUSH CMyEdit::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
HBRUSH hbr = CEdit::OnCtlColor(pDC, pWnd, nCtlColor);
HBRUSH brush;
brush=(HBRUSH)GetStockObject(HOLLOW_BRUSH);
pDC->SetBkMode(TRANSPARENT);
pDC->SetTextColor(RGB(255,0,0));
return brush;
}
```

到底是 BUG 还是牛刀的思路上有了毛病，牛刀自己也说不清，有空多看看别人的代码，懂了再与大伙详细地讲解。

当然喽，如果你明白其中的道理，或者有其他的想法，还请告诉牛刀，牛刀做到这里时，真的是有点不知该怎么办好喽，下面是效果贴图：

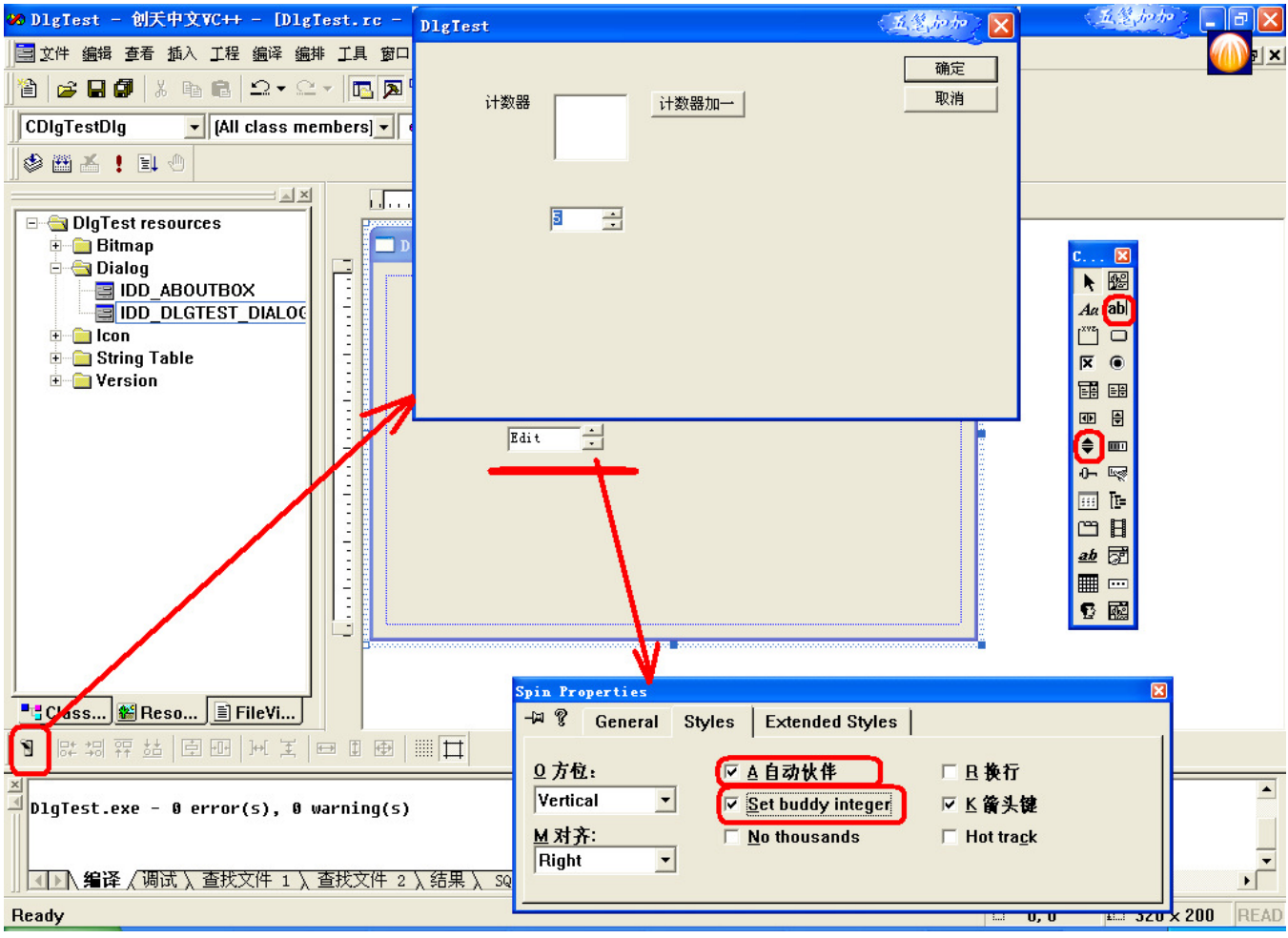

DlgTest
确定
取消
计数器
计数器加一
这个白框是牛刀的一
大遗憾，谁能解决，
请告诉牛刀一下

## 微软积木〇九三：周边控件

上一讲我们与大家还是比较详细地介绍了编辑控件，其实我们也可以将编辑控件绑定一个整型变量，或者做一些其他的设置，而有一些控件就自然成了一种编辑控件的周边控件。什么意思呢？

例如旋钮控件，滚动条控件，等等，主要负责一些输入，而还有一些控件，如进度条控件等，主要负责一些输出。

甚至旋钮控件本身根本不需要去编程，只要在编辑框画出来之后，再画一个旋钮控件，然后设置一些属性就可以自动使用了，很是方便：




大家看图，设置两个参数，我们连编程都没有必要，就可以直接使用了，这让牛刀想起了 VB ，呵呵，我们 VC 中一样可以做得非常出色，当然，为了实现一些特殊的目的，我们也可以对其进行编程，当然，我们先定义一个指针指向这个控件，然后去设置就行了：

例如，我们可以在 OnInitDialog 中去设定一些参数，例如我们要设置 1

到 1 2 十二个月，那么我们就可以这样设置：

```
                                                CSpinButtonCtrl
*pSpin=(CSpinButtonCtrl*)GetDlgItem(IDC_SPIN1);
        pSpin->SetRange(1,12);
```

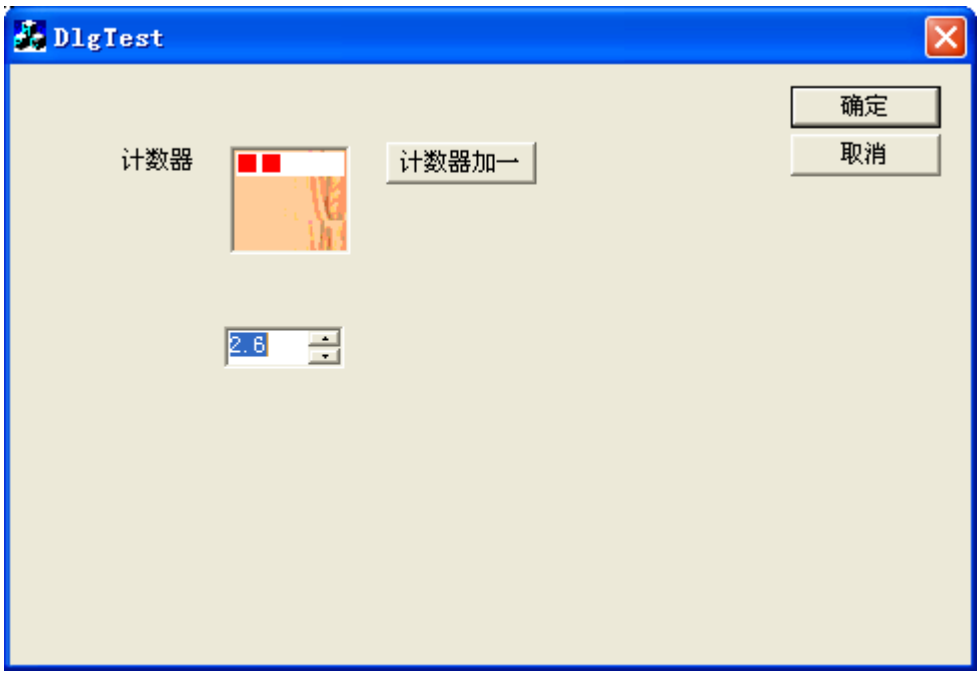
当然，有时候我们也会需要使用一些小数，例如我们需要其值从 0.0 到 20.0 每次变动 0.2 个数据，那么我们也可以这样设置：

```
 CSpinButtonCtrl *pSpin=(CSpinButtonCtrl*)GetDlgItem(IDC_SPIN1);
        pSpin->SetRange(0,100);
```

然后，我们响应 WM_VSCROLL 消息，输入以下的代码：

```
void CDlgTestDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
        if(nSBCode==SB_ENDSCROLL)
        {
                return;
        }
        if(pScrollBar->GetDlgCtrlID()==IDC_SPIN1)
        {
                CString strValue;
                strValue.Format("%3.1f",(double)nPos/5.0);
                ((CSpinButtonCtrl*)pScrollBar)->GetBuddy()->SetWindow
Text(strValue);
        }
        CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

我们就可以运行试试效果了：



当然，我们看看上面的程序，显然我们按这些旋钮时，会出现这种调用，前面一个是看我们有没有调到头，调到头了就不调了，而后面是测试看是不是旋钮，如果是旋钮（对了，有些书上也会称之为“微调按钮”，一样的，呵呵，反之，牛刀没看到其在哪儿旋转了，呵呵）。

里面的程序就显然很简单的了，定义一个字符串，给这个微调按钮的伙伴贴上，呵呵。

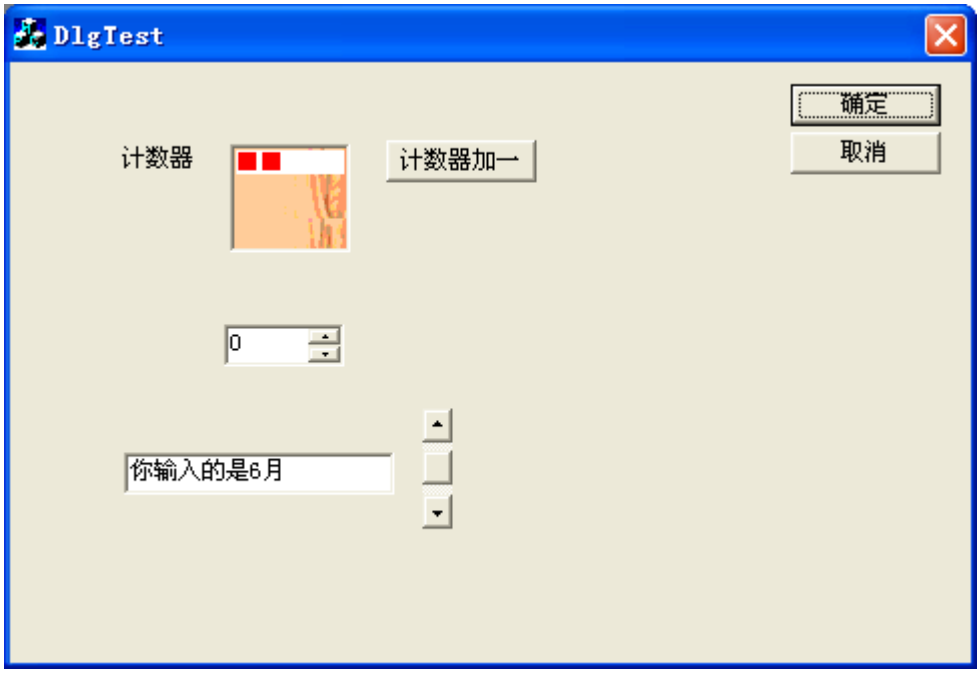
滚动条也一样：

```
CScrollBar *pSBar=(CScrollBar*)GetDlgItem(IDC_SCROLLBAR1);
pSBar->SetScrollRange(1,12);
```

上面的代码应该放在哪里大家还有数吧？好下面我们就响应滚动消息：

```
void CDlgTestDlg::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
        int i=pScrollBar->GetScrollPos();
        CEdit *pEdit=(CEdit*)GetDlgItem(IDC_EDIT2);
        switch(nSBCode)
        {
        case SB_THUMBPOSITION:
                pScrollBar->SetScrollPos(i);
                break;
        case SB_LINEDOWN:
                pScrollBar->SetScrollPos(++i);
                break;
        case SB_LINEUP:
                pScrollBar->SetScrollPos(--i);
        }

        CString str;
        str.Format("你输入的是%d 月",i);
        pEdit->SetWindowText(str);

        CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

这个消息应该同样好理解吧，我们建立的是竖直滚动条，如果我们建立横向滚动条则需要响应 WM_HSCROLL 消息，不做编起程序来是一样的：




当然，上面我们没有响应所有的消息，消息有如下几种：

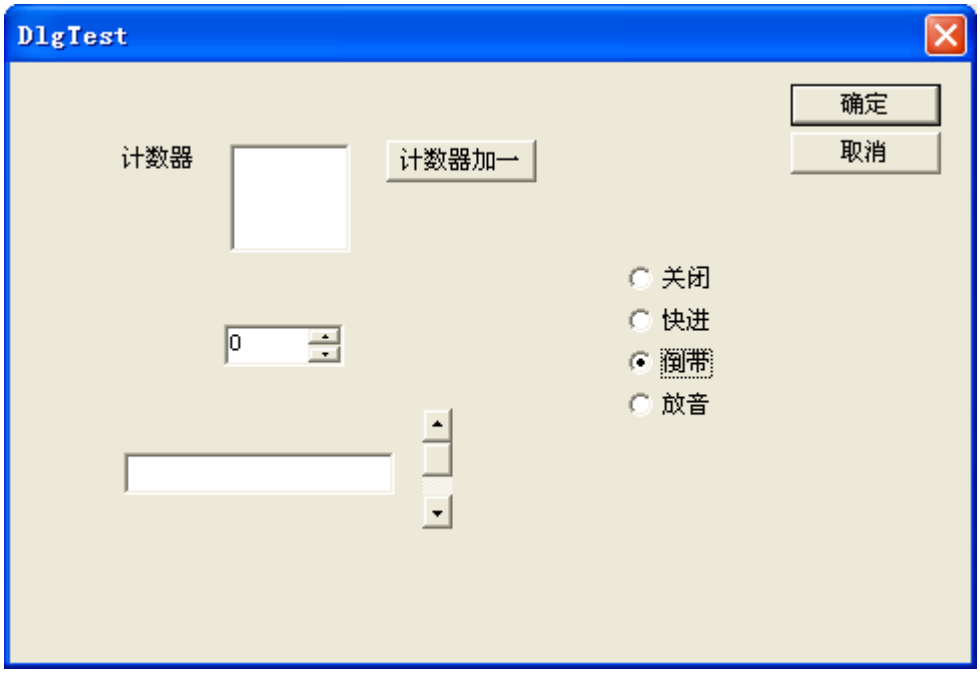
SB_BOTTOM
SB_ENDSCROLL
SB_LINEDOWN
SB_LINEUP
SB_PAGEDOWN
SB_PAGEUP
SB_THUMBPOSITION
SB_THUMBTRACKSB_TOP

大家看了应该能明白是什么意思吧？呵呵。

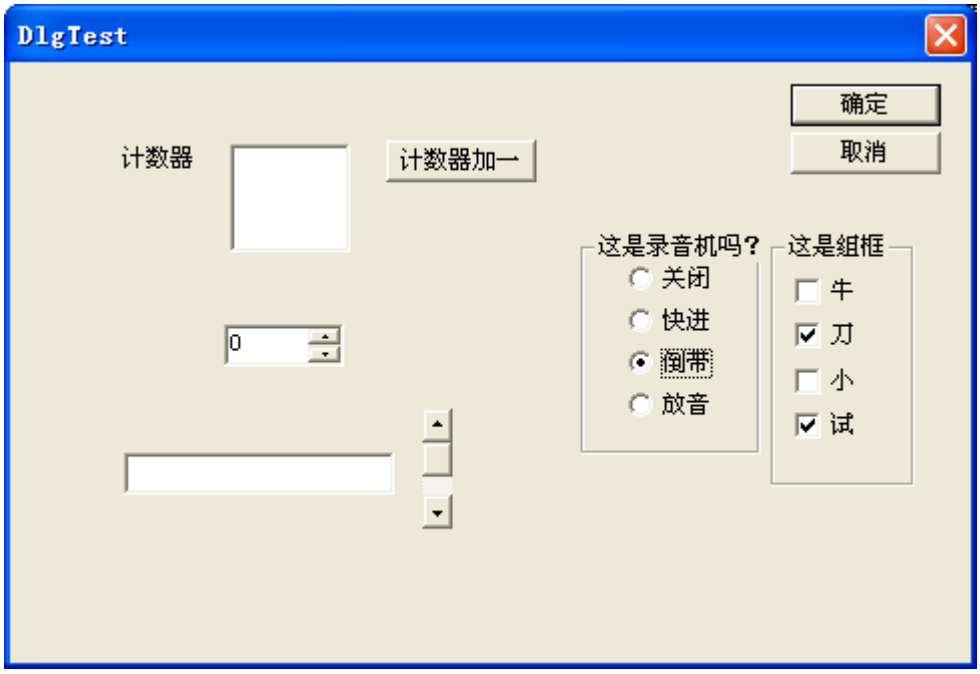
下面我们对上面的程序解释一下：

上面的程序，我们对控件，都使用了　定义一个对象指针的办法，直接去指向控件，然后操作，这种办法十分有效、安全，并且使用起来特别地舒服，效果和我们在“建立类向导”中定义差不多，但由于我们是直接指向控件的指针，所以我们不需要用　UpdateDate　去与类交换数据了。

还有一类控件，称之为单选按钮与复选按钮，单选按钮有点像我们老式录音机的按钮，按下一个，其他的就弹起了，我们可以弄几个试试：




大家看，点点看，当然，我们可以将这些按钮用一个组框给框起来，复选按钮也一样，看效果：




属性的第二个选项卡上有　Push-like　选起来，就好像按钮一样，效果很好：

我们甚至可以为其插入一个图标或者位图，大家玩过一些软件吗？上面一个小锁，一点就开了，一点又锁上了，特别地好玩，你通过自己的努力，当然也可以去实现一个，当然喽，牛刀在这里不和大家玩这种花活了，呵呵，那种就不是技术问题了。咱们这儿与大家讲的是编程的技术，呵呵。

当然，属性面板的第一个选项卡上有一个组标志 Group 咱们使用单选按钮一定要使用这个标志的，当然是第一个单选按钮使用这一个标志，然后就可以在“建立类向导”的第二个选项卡中看到这一个 ID 号了，（不选组看不到这个 ID 号嘀）。

当然喽，我们去定义一个 ID 号，

```
int             m_rec;
```

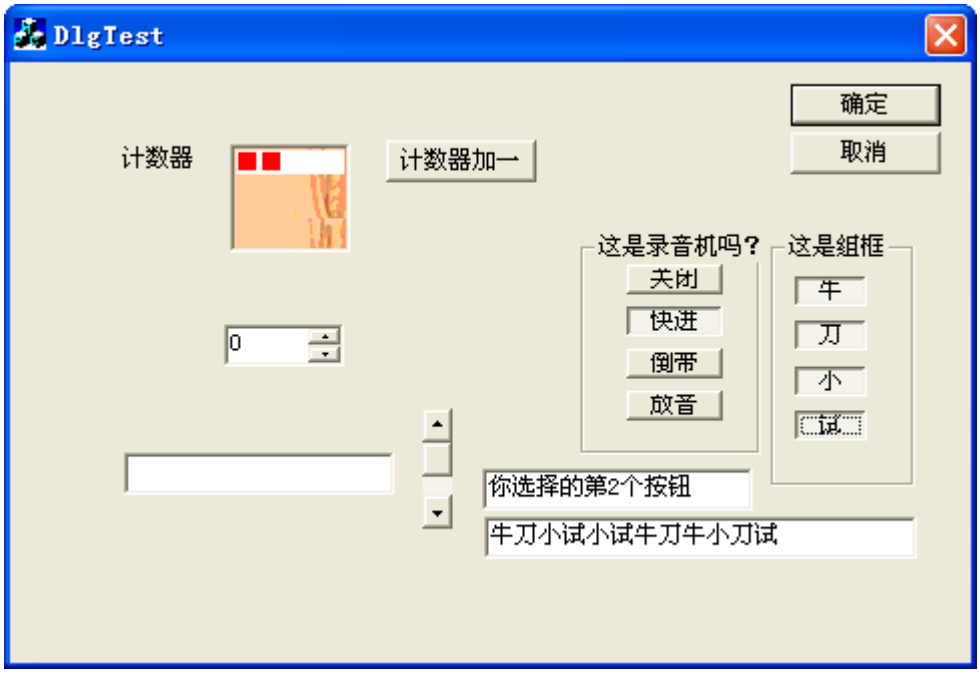
请注意，使用“建立类向导”去定义啊，除非你对整个 MFC 的来龙去脉都熟悉了，你可以直接在你的头文件或 CPP 文件中直接加代码，没有熟悉到那种程序，还建议大家按规矩来。

好了，我们响应四个单选钮与组选钮的 BN_CLICKED 消息：

```
void CDlgTestDlg::OnRadio1()
{
        CString str;
        UpdateData();
        str.Format("你选择的第%d 个按钮",m_rec+1);
        GetDlgItem(IDC_EDIT3)->SetWindowText(str);
}

void CDlgTestDlg::OnRadio2()
{
        CString str;
        UpdateData();
        str.Format("你选择的第%d 个按钮",m_rec+1);
        GetDlgItem(IDC_EDIT3)->SetWindowText(str);
}

void CDlgTestDlg::OnRadio3()
{
        CString str;
        UpdateData();
        str.Format("你选择的第%d 个按钮",m_rec+1);
        GetDlgItem(IDC_EDIT3)->SetWindowText(str);
}

void CDlgTestDlg::OnRadio4()
{
        CString str;
        UpdateData();
        str.Format("你选择的第%d 个按钮",m_rec+1);
        GetDlgItem(IDC_EDIT3)->SetWindowText(str);
}

void CDlgTestDlg::OnCheck1()
{
        CString str;
```

```
        GetDlgItem(IDC_EDIT4)->GetWindowText(str);
        str+="牛";
        GetDlgItem(IDC_EDIT4)->SetWindowText(str);
}

void CDlgTestDlg::OnCheck2()
{
        CString str;
        GetDlgItem(IDC_EDIT4)->GetWindowText(str);
        str+="刀";
        GetDlgItem(IDC_EDIT4)->SetWindowText(str);
}

void CDlgTestDlg::OnCheck3()
{
        CString str;
        GetDlgItem(IDC_EDIT4)->GetWindowText(str);
        str+="小";
        GetDlgItem(IDC_EDIT4)->SetWindowText(str);
}

void CDlgTestDlg::OnCheck4()
{
        CString str;
        GetDlgItem(IDC_EDIT4)->GetWindowText(str);
        str+="试";
        GetDlgItem(IDC_EDIT4)->SetWindowText(str);
}
```

好了，我们看看效果：




上面的复选框其实牛刀没有对其数据导出，真正使用时，也应该在“建立类向导”中，将数据导出，然后作出判断，从而使得我们的程序真正地做到名符其实，呵呵。

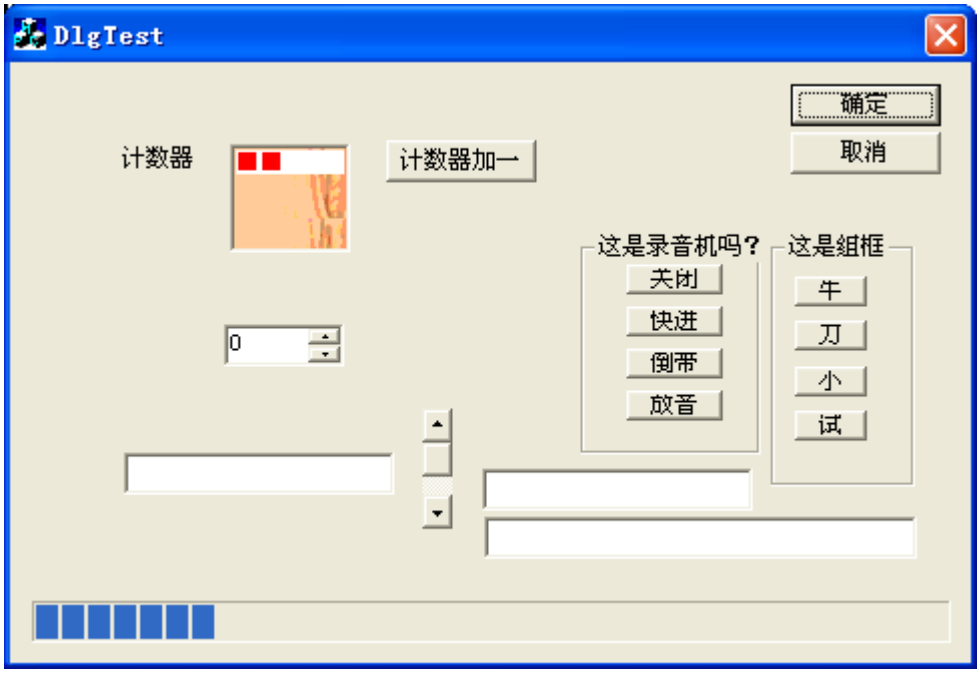
进度条，其实这个控件有点暧昧，应该属于输出型的控件，但我们一般不认为其是输出型的控件，而一般的用途是视觉安慰。

当然我们运行一个软件时，软件由于设置的不完善，运行过程比较慢，怎么办？用一个进度条放在上面，告诉你到哪儿了，你可能会看着高兴，哇，要结束了，要结束了，你还不要说，离开这东东还就真的不行。特别是一些大型的软件，如果运行速度真的很慢，如果没能进度条，你很可能就认为电脑已经“死机了”，恨不得把电脑砸了，但现实就是有了进度条，你的眼前就一片光明。

所以牛刀没有将这个控件与电脑的输出扯上一些关系，好了，不多说了，在窗体上建立一个吧，下面我们就来做程序：

```
                                        CProgressCtrl                    *
pProg=(CProgressCtrl*)GetDlgItem(IDC_PROGRESS1);
        pProg->SetRange(0,100);
        pProg->SetPos(20);
```

大家看效果：




当然，大家可以到属性的第二个选项卡将“平滑”选上，我们的选项卡上就会显得“平滑”了，呵呵显然的嘛。

一个进度条，如果不动，那么什么感觉也没有咱们下面想将其整动起来，咋整呢？

计时器呗，显然的哟，好，咱们来启动计时器：（当然首先得创建是也不是？）

```
int CDlgTestDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CDialog::OnCreate(lpCreateStruct) == -1)
                return -1;
                SetTimer(0,100,NULL);
        return 0;
}

void CDlgTestDlg::OnTimer(UINT nIDEvent)
{
                                                CProgressCtrl                   *
pProg=(CProgressCtrl*)GetDlgItem(IDC_PROGRESS1);
        int i=pProg->GetPos();
        i=(i+1)%100;
        pProg->SetPos(i);

        CDialog::OnTimer(nIDEvent);
}
```

当然，这个贴图贴不出效果来，大家在机上试试就知道怎么个好玩了，呵呵。

有一点遗憾的就是，我们的进度条如果能够在中央显示百分比那该多好啊!!!

怎么办？大家想出办法来了吗？

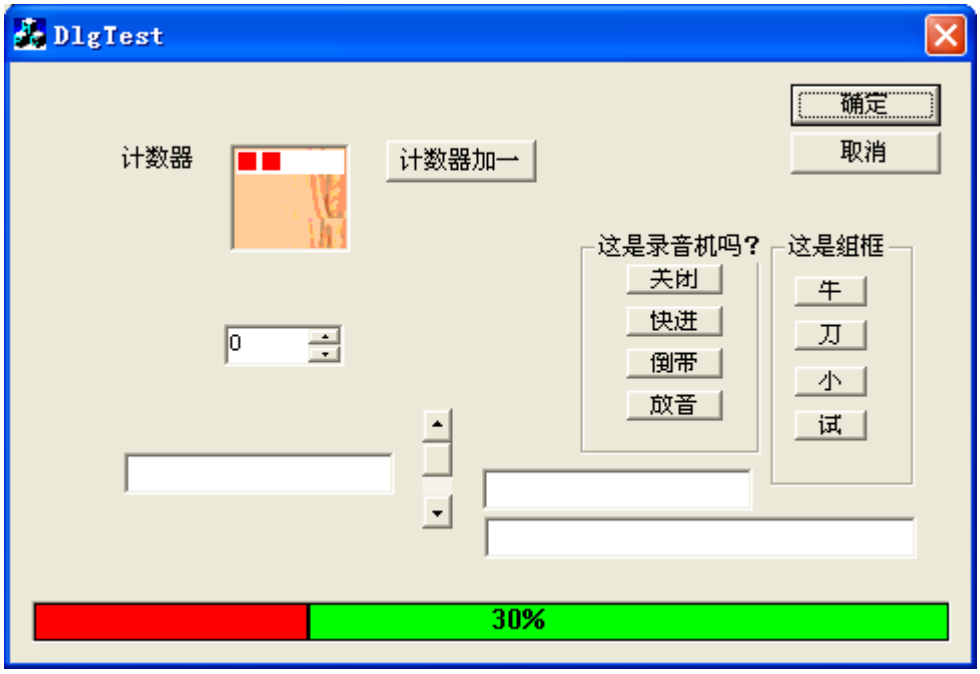
对，父子插足，咱们来做一个：

CMyProg m_prog;

制作过程咱们不说喽，省空间，呵呵：

```
void CMyProg::OnPaint()
{
        CPaintDC dc(this);
        int nPos=GetPos();
        CRect rect,recttest;
        GetClientRect(&rect);
        CString str;
        str.Format("%d%%",nPos);
        CPoint pt=rect.CenterPoint();

        CBrush brush;
        brush.CreateSolidBrush(RGB(255,0,0));
        CBrush *pOldBrush=dc.SelectObject(&brush);
        recttest=rect;
        recttest.right=rect.right*nPos/100;
        dc.Rectangle(recttest);
        recttest=rect;
        recttest.left=rect.right*nPos/100;

        brush.DeleteObject();
        brush.CreateSolidBrush(RGB(0,255,0));
        dc.SelectObject(&brush);
        dc.Rectangle(&recttest);


        dc.SetBkMode(TRANSPARENT);
        dc.TextOut(pt.x,0,str);

        DeleteObject(brush);
        dc.SelectObject(pOldBrush);
}
```

感觉怎么样？咱们给这进度条的 WM_PAINT 给改了，下面是效果，当然是不能给你显示动态的喽，当然，你自己可以做出软件来看看效果：




从效果，从感觉上来看，都还不错，你说呢？

其实牛刀在做上面的程序之前，牛刀也试过其他的办法，当然牛刀这个办法是最省事的，就是用两个不同的刷子画两个矩形，然后在中央（其实不是正中央啊，呵呵）贴上汉字，就一切　OK　了嘛。

当然，是不是突然有一种冲动，将这个进度条的显示给改了，弄一幅位图放上面该是什么情况啊，呵呵。真是个美妙的想法，这个想法牛刀就不去做了，大家自己去实现一下，看看你的能力有多高，你会被你自己的编程水平吓一跳的。

显然滚动条我们也可以用贴图的办法去贴上一幅图，将其效果也改了，那个微调按钮嘛……改，那个单选框、复选框嘛——改。

反正这种“父子插足”的技术，可以这样说，是“通吃一切”的技术。

当我们回顾这种技术时，我们不经要惊叹微软那些家伙的聪明，还有好多的计算机前辈们，给我们的编程制造了多么美妙的空间，可以任我们的思路自由地发挥，我们愿意怎么干，我们就怎么干，我们的一切，都掌控在我们自己的手中。

当然喽，周边控件还有“滑杆控件”，我们不讲了，我们现在的水平其实已经可以自己去任造一款控件了，使用这种控件还不是小菜一喋？

当然还有什么日期控件啊，热点控件啊，IP 地址控件啊，这些控件，大家随便拖进对话框，然后用和以前一样的技术，我们只要将这些控件的指针指过去，然后一打“->”，立马后面会有一个列表，我们要的函数全在这里，然后看看名称，一般什么意思可以猜出来，如果还不明白的话……

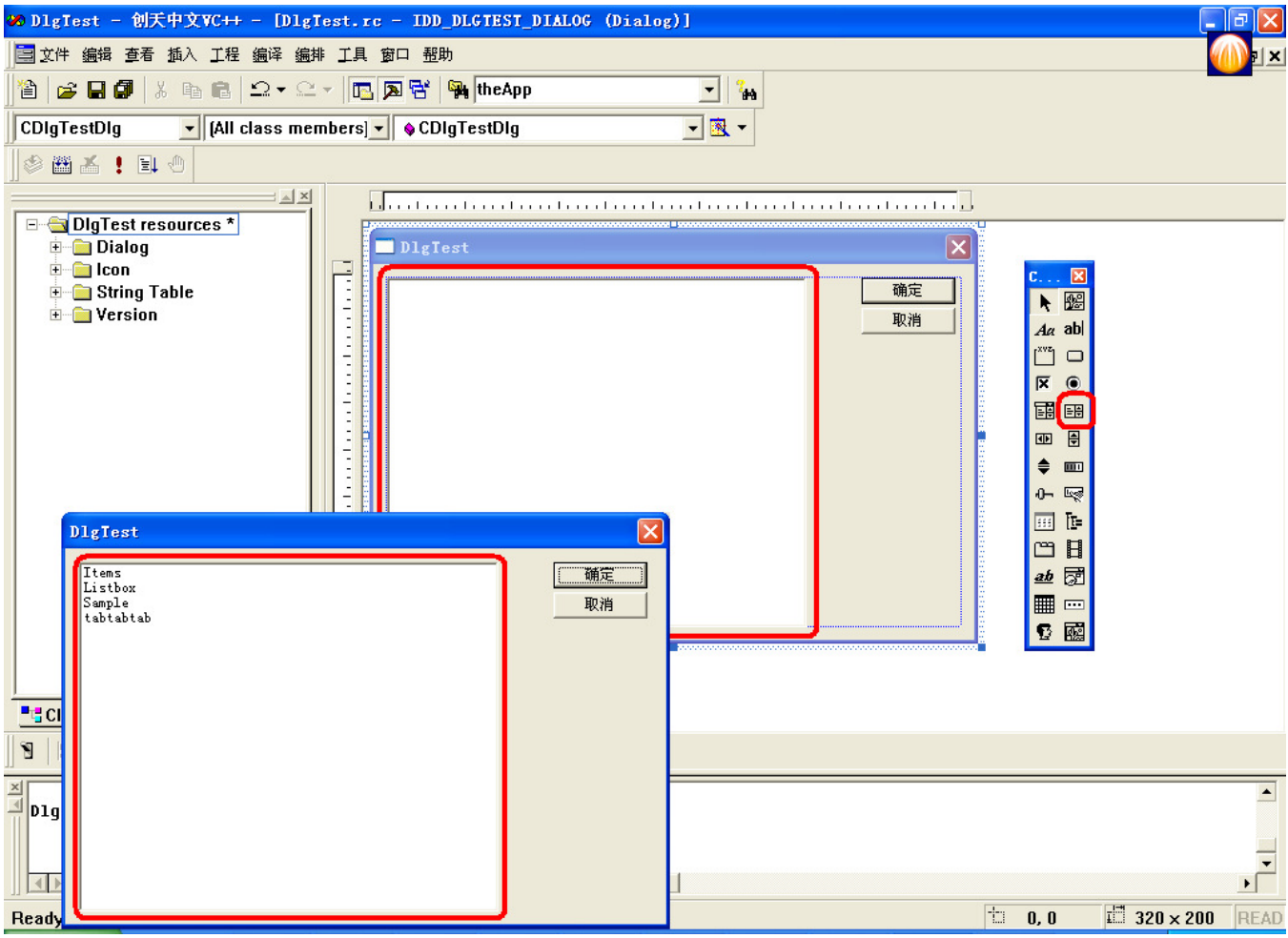
大家千万要记住，按“F1”，你不要告诉牛刀你的电脑上没有 MSDN ，呵呵，那可惨了。那么就上网找找吧，如果不怕病毒的话，用搜索引擎，或者建议大家干脆就在网页的首页多链几个编程的网站，我们有空就点过去好了，呵呵。

还是那句话，牛刀是不会与你详细地讲控件的每一个细节，但牛刀会告诉你怎么做，我们“实践”的目的就在于此。

当然，这也应该是牛刀这套教材的一大特色吧，呵呵，希望对大家有所帮助，以后看到的东西，都是我们熟悉的东西，那么，牛刀这套教材的目的也就达到了。

## 微软积木〇九四：列表控件

列表控件，什么控件呢？是一个能加入一个一个的数据，然后显示出来的控件。我们画一个出来给大家瞧瞧：



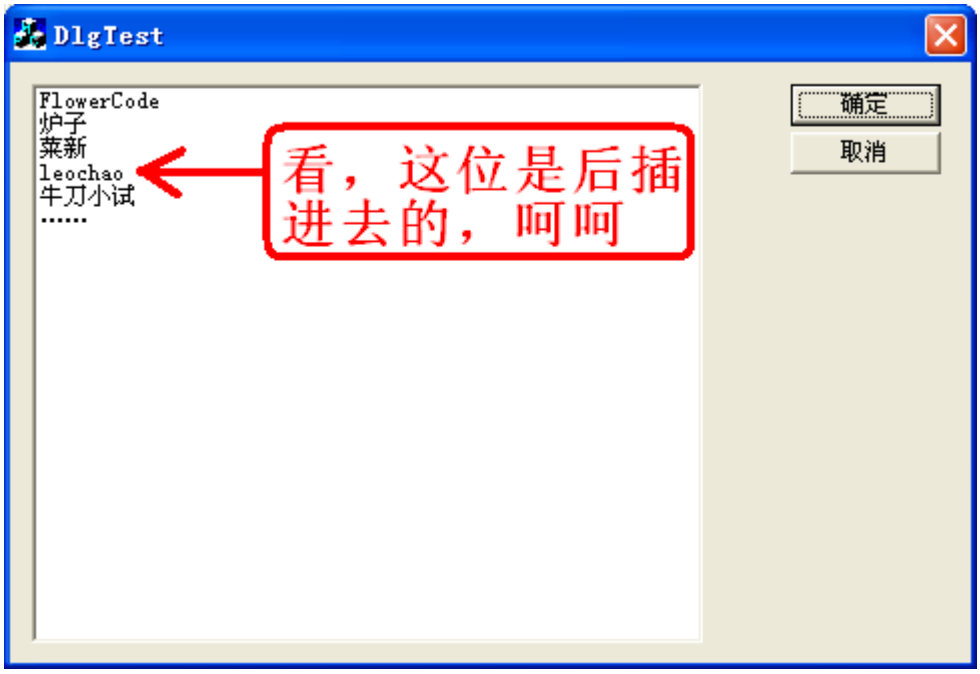

就是那样，其中可以好像列表一样地列一排排的东东，呵呵。

那么我们怎么使用呢？（说实话，将控件拿到图上，那没什么说的，根本就没有什么说的，呵呵）

那么下面我们就来编程吧：

```
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
pLB->InsertString(-1,"FlowerCode");
pLB->InsertString(-1,"炉子");
pLB->InsertString(-1,"菜新");
pLB->InsertString(-1,"牛刀小试");
pLB->InsertString(-1,"……");
pLB->InsertString(3,"leochao");
```

插入前面第一个参数是 -1 的话，那么后面就是插入到最后一个，否则就插入到前面那个数字所表示的位置，大家看下面的图：




请大家注意啊，插入的位置索引是从 0 号索引开始计数的。

```
void CDlgTestDlg::OnSelchangeList1()
{
MessageBox("我点了列表框按钮");
}
```

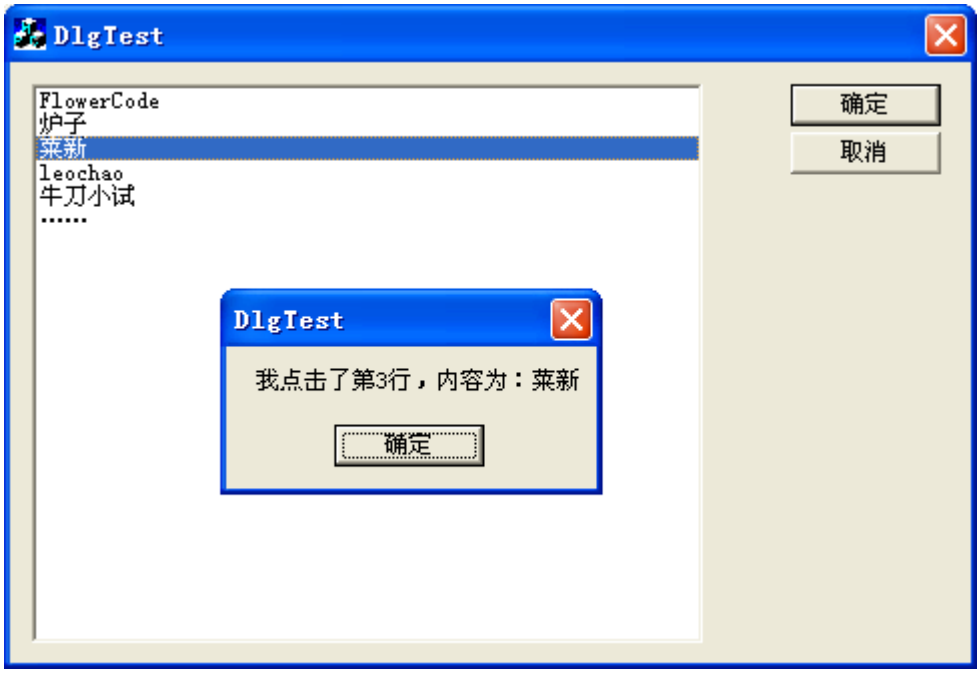
当然，我们都是在父类中做这些事的，所以如果想要使用相应的类中的东东，那就必须还要用个指针指过去，呵呵：

```
void CDlgTestDlg::OnSelchangeList1()
{
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
CString str;
int i=pLB->GetCurSel();
str.Format("我点击了第%d 行",i);
MessageBox(str);
}
```

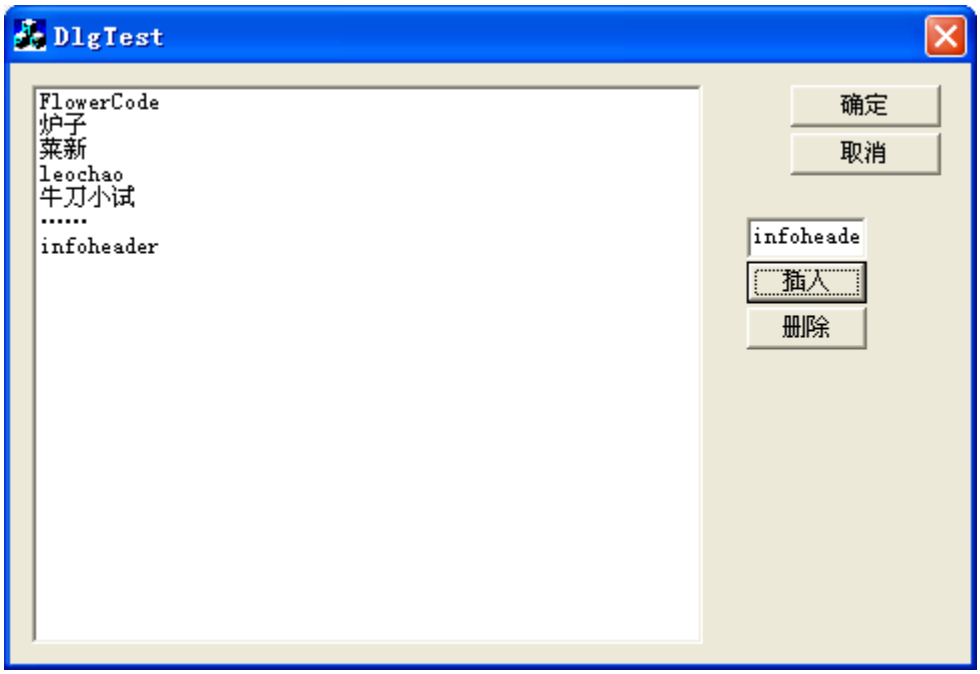
不贴图了，呵呵，大家自己看效果，我们用 GetCurSel 取得我们选取的行后，我们就可以大干我们自己想要干的事喽，请看：

```
void CDlgTestDlg::OnSelchangeList1()
{
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
CString str,str1;
int i=pLB->GetCurSel();
pLB->GetText(i,str1);
str.Format("我点击了第%d 行，内容为：%s",i+1,str1);
MessageBox(str);
}
```

注意，上面返回的 i 是从零号开始的，所以我们加了一个 1，效果如下：




当然，我们可以做一些相关的管理软件，看啊：

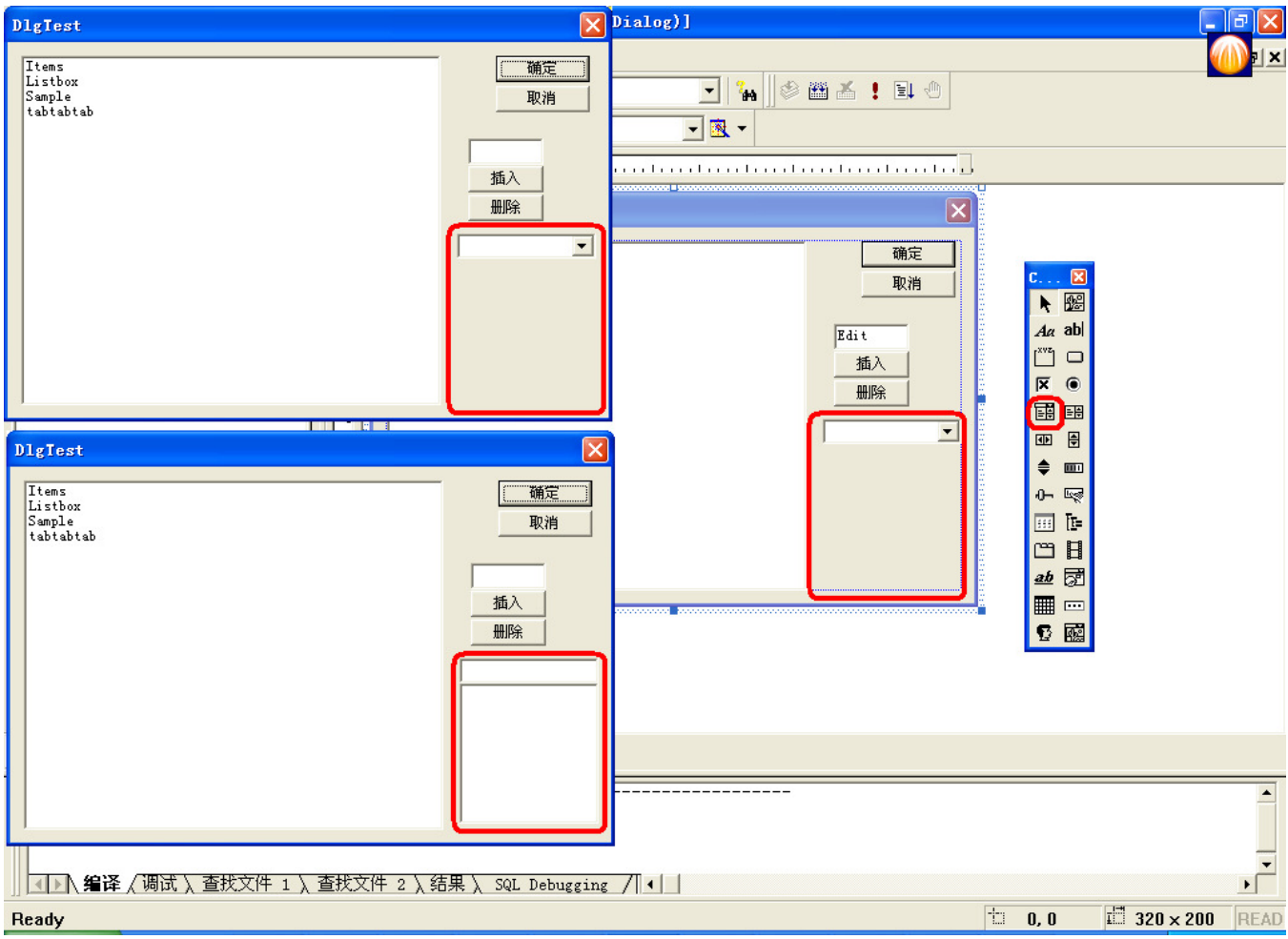



效果怎样？呵呵

```
void CDlgTestDlg::OnSelchangeList1()
{
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
CString str;
int i=pLB->GetCurSel();
pLB->GetText(i,str);
CEdit * pedit=(CEdit*)GetDlgItem(IDC_EDIT1);
pedit->SetWindowText(str);
```

```
}
void CDlgTestDlg::OnButton1()
{
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
CString str;
CEdit * pedit=(CEdit*)GetDlgItem(IDC_EDIT1);
pedit->GetWindowText(str);
pLB->InsertString(-1,str);
}
void CDlgTestDlg::OnButton2()
{
CListBox *pLB=(CListBox*)GetDlgItem(IDC_LIST1);
int i=pLB->GetCurSel();
if(-1!=i)
{
  pLB->DeleteString(i);
}
}
```

当然，我们上面用了一个文本框，一个列表框，是也不是，我们的控件中有一款专用的控件：Combo



上面的控件有三种形式，一种相当于一个文本档与一种列表框的组合，还有一种是带下拉列表的，有两种，一种是可以在文本框中输入内容的，一种是只能选择的，大家试试看就知道了，和列表框控件的使用几乎一样，使用也一样，特别地舒服，呵呵。

好了，以上的一切，我们都有机会去慢慢地研究，琢磨，那么下面咱们干什么呢？

嘻嘻，老一套呗，贴一幅图来玩玩：

技术要点：父子插足

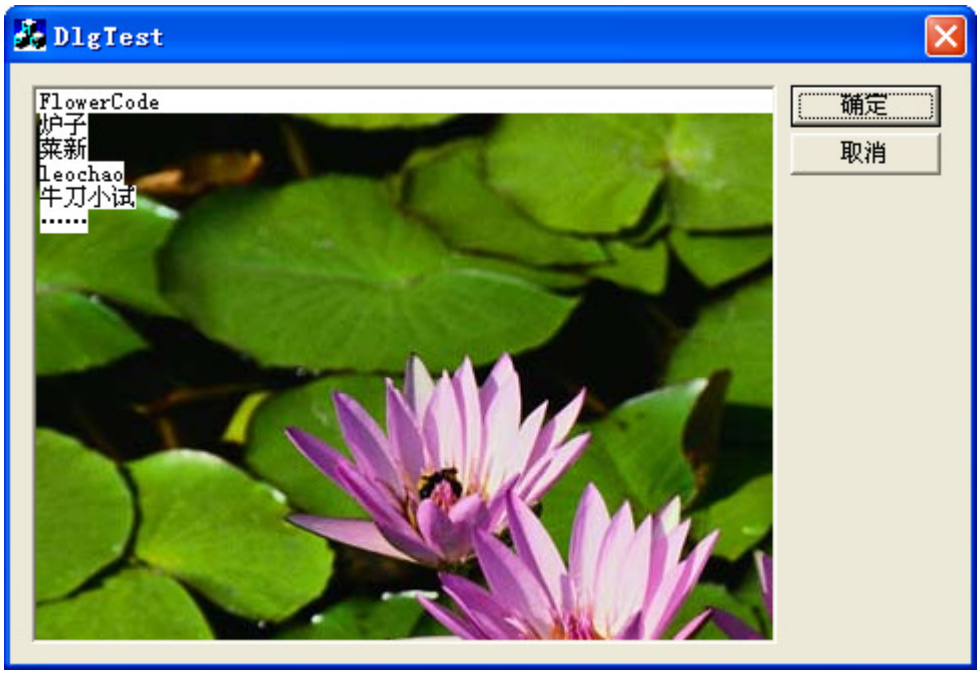
CListBox m_list;

改为：

CMyList m_list;

```
BOOL CMyList::OnEraseBkgnd(CDC* pDC)
{
CBitmap bmp;
bmp.LoadBitmap(IDB_MYLIST);
CBrush brush;
brush.CreatePatternBrush(&bmp);
CClientDC dc(this);
CRect rect;
GetClientRect(&rect);
dc.FillRect(&rect,&brush);
return TRUE;
}
```

看效果：



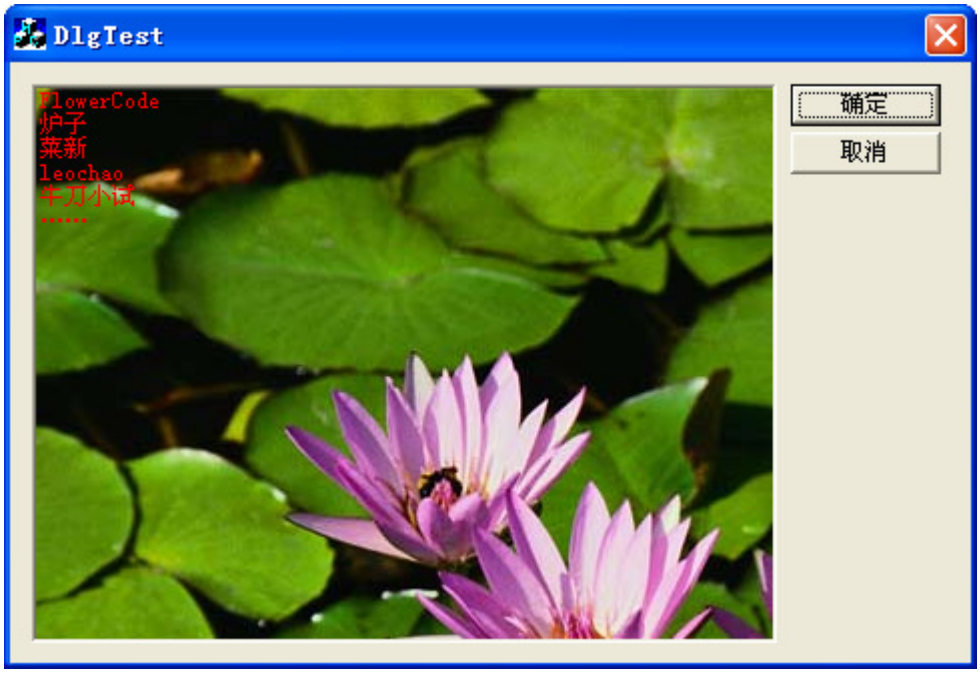

这次我们看到了文字下面的白的了吧？再做响应一个 =WM_CTLCOLOR 消息吧：（特别注意，一定要选择前面带等号的那个消息，我们以前编辑框不能

响应这个消息可能也是这个原因，大家回去改一改调试一下那一个程序看看：

```
HBRUSH CMyList::CtlColor(CDC* pDC, UINT nCtlColor)
{
pDC->SetBkMode(TRANSPARENT);
pDC->SetTextColor(RGB(255,0,0));
return (HBRUSH)GetStockObject(NULL_BRUSH);
}
```

对了，说一下吧， SetBkMode 其实是设置文字的模式，什么意思呢？我们大家用文字在一块图片上用 TextOut 贴图看看，会有一个白色的方块，而模式设为 TRANSPARENT 意思就像那一块白色的方块就变得透明了，呵呵。

效果如下：




当然，你点击就会有问题，所以我们还要响应一些如点击啊什么的，去重新刷新一下，再者，就是如果出现了滚动条，我们要去重画滚动条，就要去设置 WM_VSCROLL 和 WM_HSCROLL 消息。

总之，我们编程序时，如果我们哪个地方有问题，不满我们的意，我们就需要去更改，当然，这是父子插足的最大缺点，所以说，设置一个完全符合我们需要的类也并不是像我们想像得那样太容易，所以我们要尊重别人的劳动成果。

好了，牛刀下面就不编程了，与大家聊聊天吧，呵呵列表框其实是一款很优秀的软件，我们如果用的好的话，那么我们的好多工作几乎可以自动。

首先，我们的线性数据结构，我们就可以通吃，当然喽，如果要当数据结构用的话，大家可以将列表框隐藏，只要我们知道这个数据结构就可以了：

·当链表用

我们向前找数据只需要将当前的位置减一，向后找只要加一就一切 OK ，很是方便，用不着指针了

插入变得异常地方便，我们只要找到位置，执行插入操作就一切 OK 了。

删除当然也只是一行代码就可以搞定

排序，那几乎是设定一个属性的事如果你的软件中放一个隐形的列表控件，

人家没人能看得出来的，几乎可以不需要任何的排序算法，我们只管将数据往表中插，然后拿出来就是我们想要的数据了。

当然，有一点可惜的是，我们的链表排序是字典排序的。

什么意思，大家看下面的向个数字

2 5 17 55 123

这几个数据的排序顺序是什么？

可能你会想，上面的不就是按顺序的吗？可是，如果你放在对话框中，就没有这么简单了，那必然成为以下顺序：

123

17

2

5

55

如果你没有学过的话，你很难想像，但的确是这样的，我们的列表框排序是按高位排序，而不是低位排序的。于是就出现了上面的:

123<17<2<5<55

这样的现象。所以说，数学运算还就是算了吧。

队列，队列是先进先出的结构，就是我们排一个队，就从一头进，从另一头出，排在你后面的人总是不可能从你前面出来，这就是队列，当然排列也就很简单喽

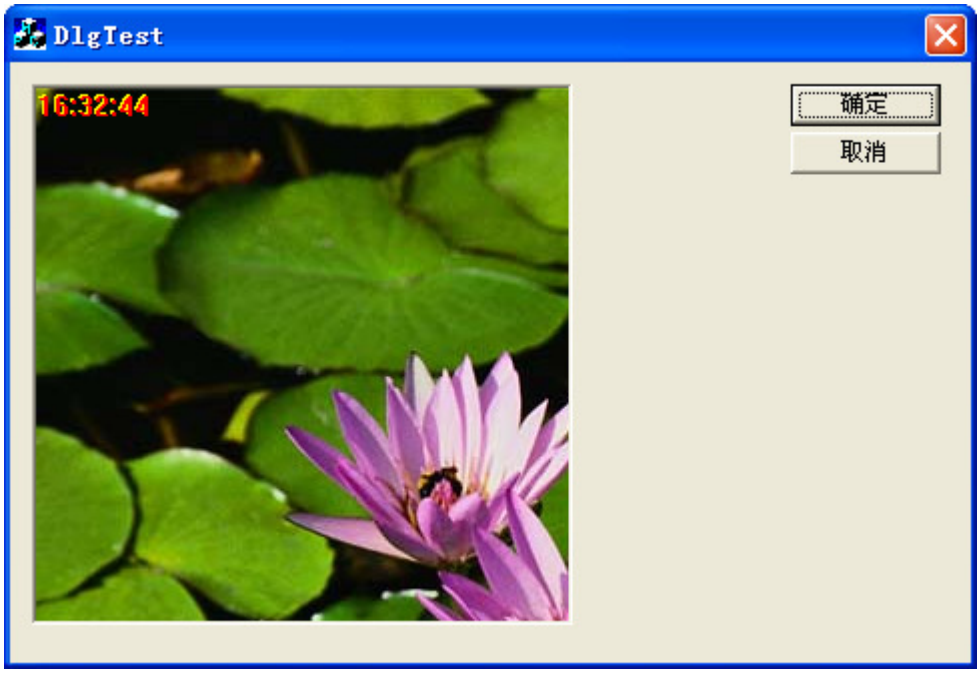
栈，是先进后出的结构，就像大家洗盘子，放时总先放最上面，拿也是先拿最上面一个（大家不要告诉我，你要么就全部拿走哟，呵呵。）

相信以上的程序，大家用列表都可以实现吧？牛刀不与大家详细举例子了。

牛刀突然有了一个奇怪的想法，我们将列表框改为其他的控件怎么样啊？嗯，想个控件吧，咱们将其改为一个控件，控件上显示一面时钟怎么样啊？这样的控件大家看过么？咱说干还就干：

```
void CMyList::OnPaint()
{
CPaintDC dc(this);

CTime tim;
tim=CTime::GetCurrentTime();
CString str;
str=tim.Format("%H:%M:%S");
dc.SetBkMode(TRANSPARENT);
dc.SetTextColor(RGB(255,255,0));
dc.TextOut(0,0,str);
dc.SetTextColor(RGB(255,0,0));
dc.TextOut(1,1,str);
}
```

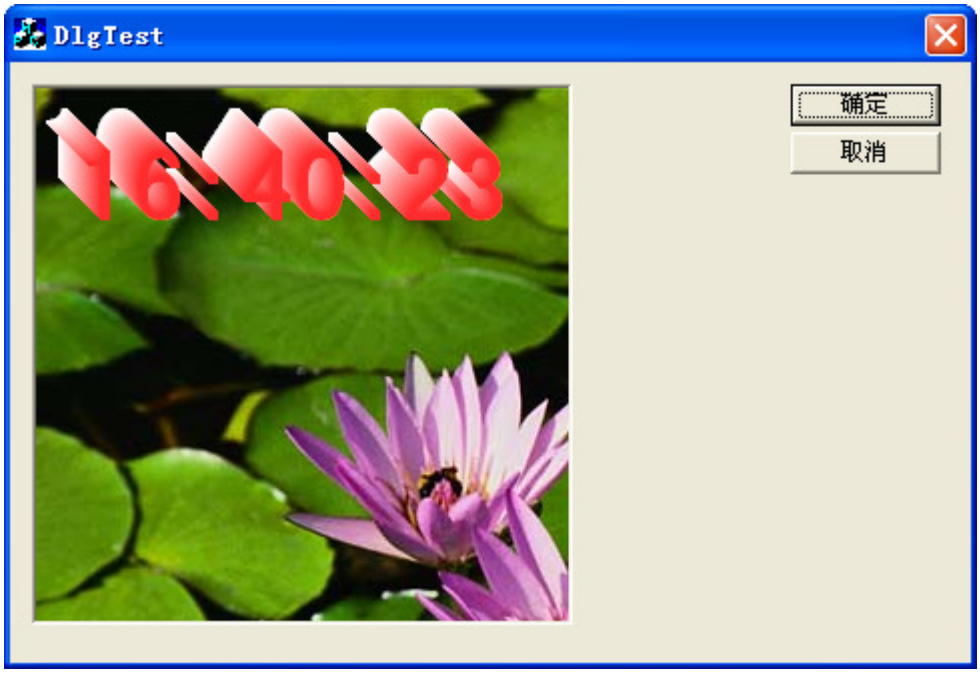
看下面的图片效果：




还有点立体感哟，有个想法，如果我们将字体选大一点，字再用过渡色过渡一下，会出现什么现象呢？

```
void CMyList::OnPaint()
{
CPaintDC dc(this);

CTime tim;
tim=CTime::GetCurrentTime();
CFont font;
font.CreatePointFont(400,"黑体");
CFont *pOldFont=dc.SelectObject(&font);
CString str;
str=tim.Format("%H:%M:%S");
dc.SetBkMode(TRANSPARENT);
for(int i=0;i<22;i++)
{
  dc.SetTextColor(RGB(255,255-i*10,255-i*10));
  dc.TextOut(i,i,str);
}
dc.SelectObject(pOldFont);
DeleteObject(font);
}
```

呵呵，这种感觉啊～～～～




好了，下面我们就要将这幅图片给弄动起来喽，当然简单，隔一段时间去刷新一下就行了呗：

```
int CDlgTestDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
if (CDialog::OnCreate(lpCreateStruct) == -1)
  return -1;
SetTimer(0,1000,NULL);
return 0;
}
void CDlgTestDlg::OnTimer(UINT nIDEvent)
{
Invalidate();
CDialog::OnTimer(nIDEvent);
}
```

效果不错，可就是闪得厉害，呵呵，看大家有什么好的办法没有，牛刀在这里又遇到了一大难题，就是在控件中去创建计时器，竟然是没有用的，不知道什么原因。

于是，就只好在父窗体上去刷新喽，不过总算是“动”了起来喽。

牛刀去调试了一下，我们用工具创建的对话框中的控件，其中竟然没有调用 OnCreate 函数，也就是根本就没有创建这一个过程，也就是说，控件不是画出来的，而是按照某种约定，事先就存在那儿了，那么，其他消息呢？牛刀不服，去调用一下其构造函数看一看：

呵呵，牛刀暂时投降了，整不出来了，大家都来出主意啊。

通过以上实现，牛刀要提醒大家一句，不要试图去响应控件的所有消息，但可以响应控件的独有消息。

说实话，牛刀目前对控件的所有消息还研究得不深，不能够将所有消息的使

用都告诉大家，有些东东是需要我们一个一个地上手去试的，但像这种全面介绍每一个控件全部消息，并且提供例子的书毕竟很少了，再说了，即使有这种书，往往一本书上也搁不了几个的例子，大家在论坛上可能果真是看不出来啊，呵呵，牛刀这套教材如果真去出书的话，那估计不是“厚厚的一本”能概括得了的，牛刀估计应该需要分卷，呵呵。

从控件的角度来说，我觉得管理员倒可以开一个新专版，将每一个控件都分门别类，然后大家自主地写东西，投稿、研究。（有些东西是真的需要研究的，而我发现论坛上有好多的文章是发一发代码了事，注释也很少，其实安全方面的东西吧，说实话，与其大家都在高手过招，还不如都将原理讲出来，大家相互切磋，当然不泛我们说，上这些论坛的都是高手，可能就是牛刀不开眼在这写一些基础性的文章，但在网络上看到，写基础性文章的论坛也真的太少了，当然也感谢管理员，每次都会将牛刀的一些贴子加精，我想管理员的目的也应该希望大家多出类似的贴子，可这可能就是一种呼吁，真正地做这个工作的人，估计应该是少之又少。可怜可悲啊!!!）

我们可以研究每一个控件的每一个特性，每一个控件的所有消息。

从这一点上来说，其实ＶＣ做的就是没有ＶＢ好，ＶＢ中的消息在那里称之为“事件”，既然列出来的事件，就是可以使用的，而我们ＶＣ中则是“大而全”，有时候令人选择的时候很难把握，呵呵。

好了，以后的路，牛刀可能也要摸索着走了，也就是说，这个 MFC 专题的好多内容，牛刀也要摸索着走过，也就是说，牛刀发贴子不可能象前面这些讲一样，每天出一期了，何况，牛刀感觉，讨论得好像不是很热烈，不知什么原因，牛刀突然发现牛刀在论坛上其实是很孤单，好像真正的高手懒得理牛刀，而真正不是高手的，可能也早就看不下去了，什么原因，很简单，我们的车开得太快了，能跟得上的人，估计不多。

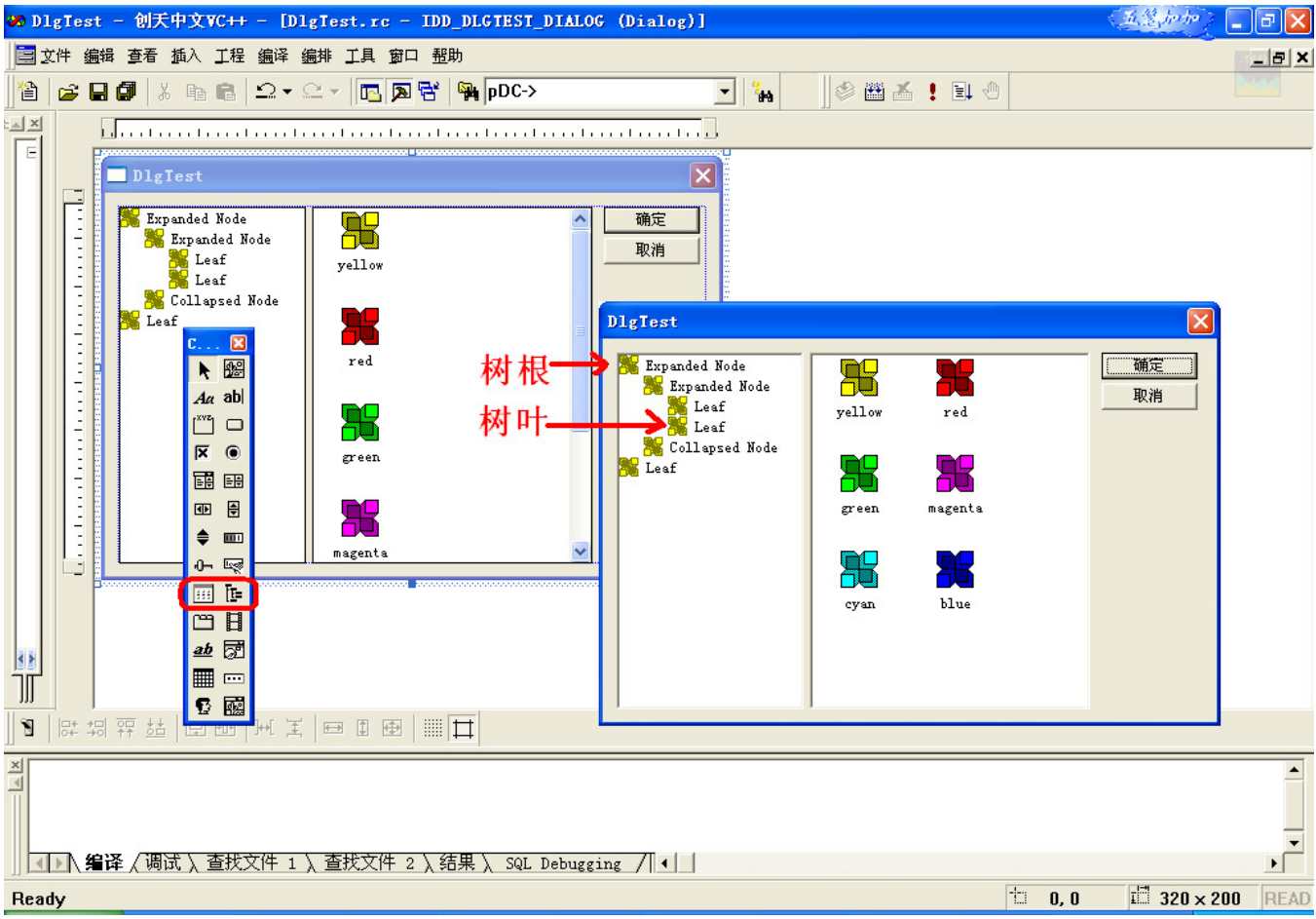
不过更可惜的是，反面的贴子也同样不多，还是很怀念那段与“梦影”争辩的日子的，真的，一个人每天写东西，而没有回馈，那真是太单调了，呵呵。

总之，希望论坛越办越好，希望牛刀的贴子能够帮助你扫清学习路上的拦路虎，如果你相信牛刀的话，那么请你拿起任何一本编程书籍的话，牛刀的建议就是——实践、实践、再实践。

在实践中去理解每一个知识点。牛刀的教程就一直是这么做的，相信你也会这样做，而不是每次都将牛刀的教程当着一本休闲消遣的东东，呵呵。

## 微软积木〇九五：列表与树

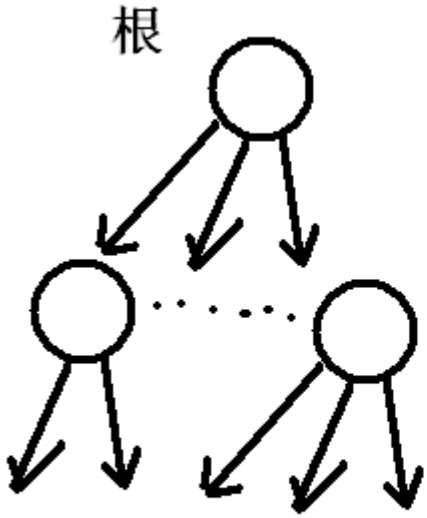
我们上一讲讲的是列表控件，这一讲我们要说说一些关于“视窗”的控件。大家先在对话框中画上两个控件吧：

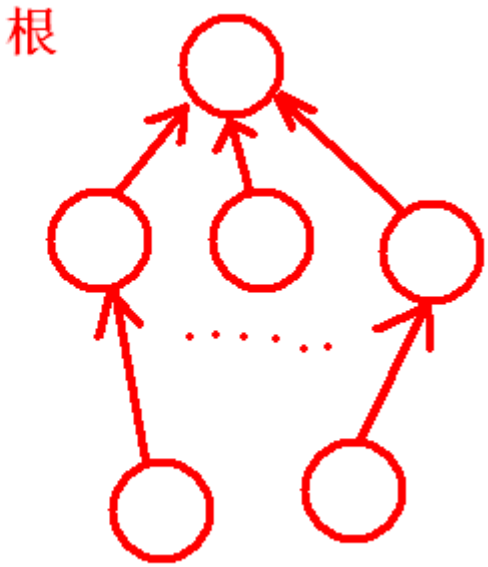



大家一看就知道哪个是树，哪个是列表吧，呵呵，左边是树，右边是列表。

下面我们就讲讲一些理论吧，呵呵，当然如果你都明白了，那么你就跳过去吧。

树，其实有两种树，大家看：




大家看左边，当然我们称之为“树”从父亲向儿子指，显然有多少的儿子，就需要多个指针，然而右边的红色的树就方便了，只要一根指针就一切 OK 了。

呵呵，我们的树型控件就是使用左边的那一种技术，当然喽，在控件内部怎么使用，使用的是什么技术，我们就管不着了，也就是说，在这个列表视图与树型视图中，需要传送给控件的东东就是不 InsertString 那么简单了。

为什么？

·一、树型结构，使得我们不可能用简单的插入操作就可以一步插入到位

·二、这两个视图中都需要用到图标或者图片

·三、这两个视图中的关系的确比较复杂，需要好多结构信息或者位置信息

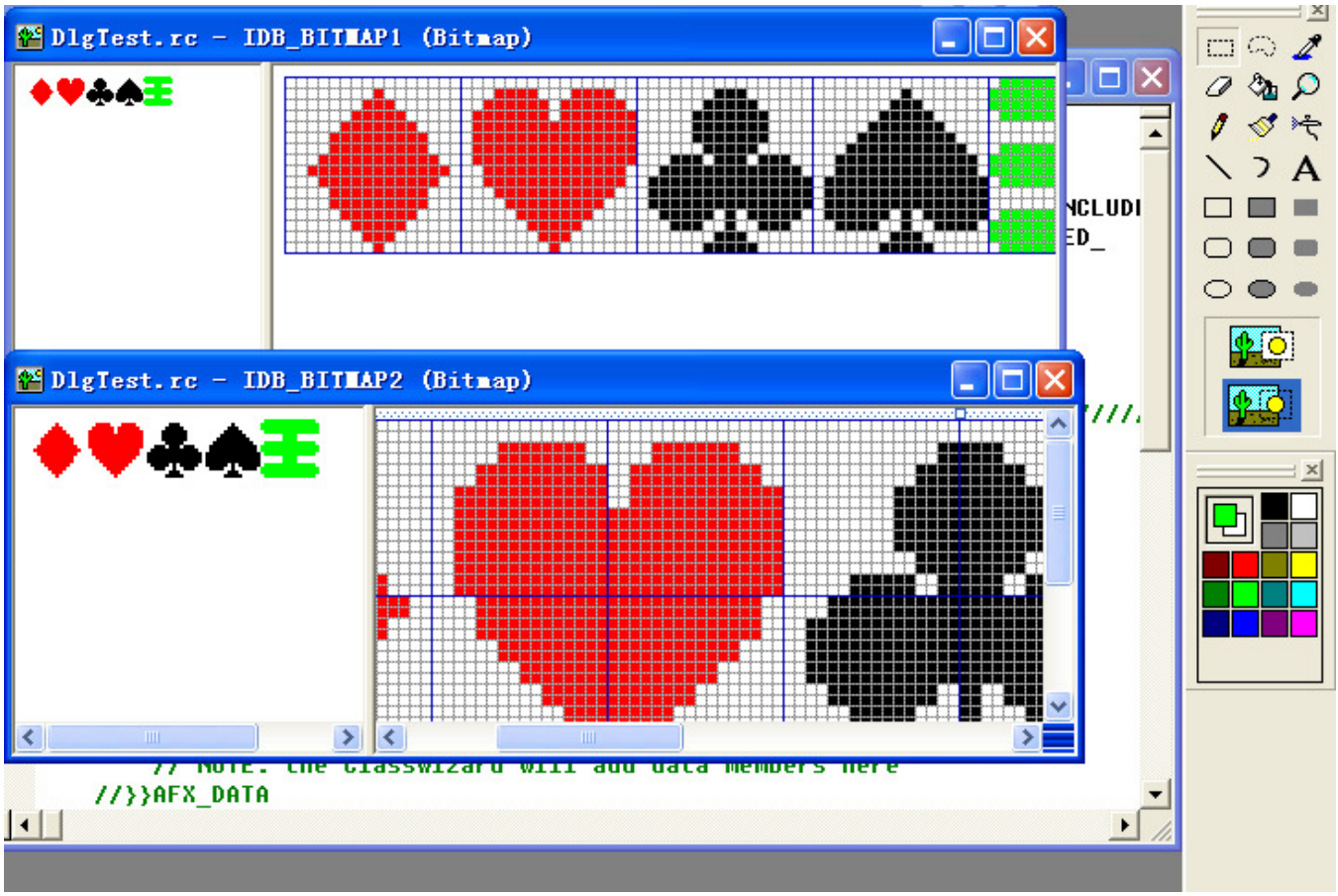
也正因为如此，我们的 WM_COMMAND 消息向父亲发送消息往往就会显得数据量不够，为了说明这一点，我们下面先来谈谈消息的构成吧：

在我们 MFC 开始几讲，我们曾经讲过消息，但那里我们一直没有提到消息是如何构成的，一般的消息中有四个参数，第一个参数是句柄，当然喽，在 MFC 中，句柄都默认了，所以有三个参数，第一个参数就为消息名称，其实是一个无符号整数，而第二个，第三个参数分别为 WPARAM 和 LPARAM ，不要看这两个参数不起眼，作用可不小，当然，我们这一讲不给大家举例子了，因为如果举例子，那么这一讲就不应该叫《列表与树》了，而要改成《消息的结构》了，呵呵，开句玩笑。

WM_COMMAND 消息，我们一直没有给大家讲解其结构，因为我们一直

在 MFC 中做程序其实其一个表示控件的 ID 号，另一个表示消息的数据，当然喽，我们不详细地与大家展开，有空我们专门做一讲去讨论。

但这两个视图类的就不同了，要传送的数据量太大了，于是，系统专门定义了一个 WM_NOTIFY 消息，去传送这些结构，当然，这一讲我们也不想讲这个消息，这个消息你在 MFC 的消息队列中找不到，其实已经被父类定义为虚函数了，到父类的虚函数中去找 OnNotify 函数，当然喽，我们也不准备在这一讲用，我们这一讲只是告诉大家的，的确有这么个消息，去可以响应并且为我们的这两个视图控件定义了一个结构，为了理解，我们先将这个结构的问题放一放，我们先为这两个控件准备两张图片，我们插入两张图片，一张是 64*16 的，一张是 128*32 的，好，下面我们就看这两幅图片：




牛刀想干什么，大家心知肚明吧，呵呵，好了，我们现在有个想法，这些图片怎么用到我们的视图中呢？

当然，可能你想到用 BitBlt 但有一个问题，大家知道在什么位置贴图吗？大问题啊～～～～。

当然想解决这种问题的确是很难，然而，人家 MFC 给我们一种新型的类 CImageList 说明的一点，在 VB 中，并没有这两个控件，但如果你加进来后，其中有一个 CImageList 控件，但我们这不需要，只需要定义一下，然后新建一个就好了，下面我们看看程序，我们定义两个成员变量吧：

```
CImageList m_SmallImageList,m_LargeImageList;
```

好吧，我们在 OnInitDialog 中去创建吧，很简单：

```
m_SmallImageList.Create(IDB_BITMAP1,16,0,RGB(255,255,255));
m_LargeImageList.Create(IDB_BITMAP2,32,0,RGB(255,255,255));
```

大家看什么意思也许能猜出来吧，第一个参数是位图，而第二个参数是每幅位图的宽度，后面两个大家就用 0 就行了，最后一个是蒙板，什么颜色就会将相应的颜色变成透明，比我们以前的 BitBlt 好用多了吧？呵呵。

当然，我们也可以使用 Add 去加，但我们这儿用不着了，行了，我们下面就还使用以前的技术去用一个列表指针指过去，然后想办法插进一些东东：（先插树型控件吧）

```
TV_INSERTSTRUCT tv;
```

怎么样，这就是树型控件的结构：

```
#define TV_INSERTSTRUCT         TVINSERTSTRUCT

#define  TVINSERTSTRUCT         TVINSERTSTRUCTA

typedef struct tagTVINSERTSTRUCTA {
    HTREEITEM hParent;
    HTREEITEM hInsertAfter;
#if (_WIN32_IE >= 0x0400)
    union
    {
        TVITEMEXA itemex;
        TV_ITEMA  item;
    } DUMMYUNIONNAME;
#else
    TV_ITEMA item;
#endif
} TVINSERTSTRUCTA, FAR *LPTVINSERTSTRUCTA;

typedef struct tagTVITEMEXA {
    UINT      mask;
    HTREEITEM hItem;
    UINT      state;
    UINT      stateMask;
    LPSTR     pszText;
    int       cchTextMax;
    int       iImage;
    int       iSelectedImage;
```

```
    int       cChildren;
    LPARAM    lParam;
    int       iIntegral;
} TVITEMEXA, FAR *LPTVITEMEXA;
#define TV_ITEMA                TVITEMA
typedef struct tagTVITEMA {
    UINT      mask;
    HTREEITEM hItem;
    UINT      state;
    UINT      stateMask;
    LPSTR     pszText;
    int       cchTextMax;
    int       iImage;
    int       iSelectedImage;
    int       cChildren;
    LPARAM    lParam;
} TVITEMA, FAR *LPTVITEMA;
```

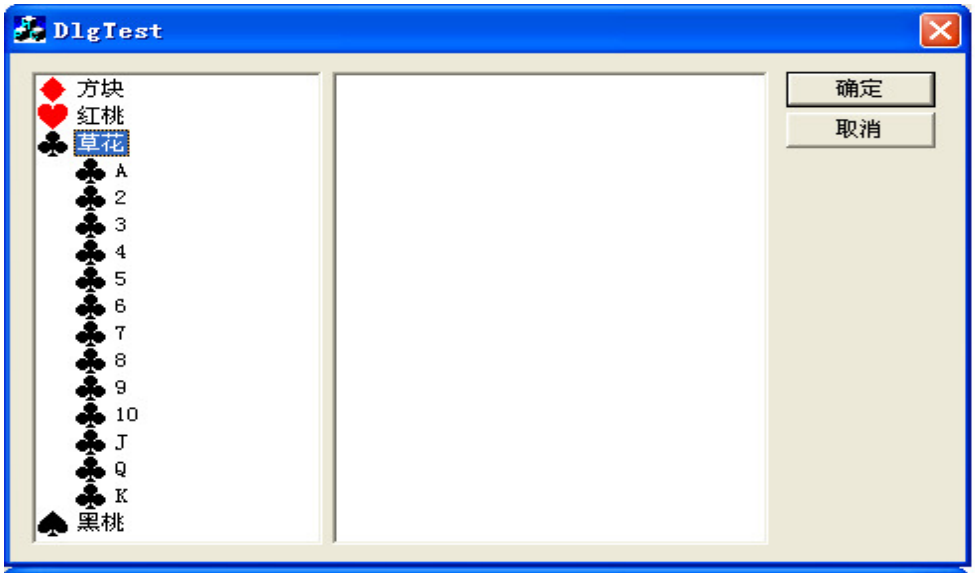
太复杂了吧？呵呵，咱们不搞复杂的，咱们拣主要的：（咱们在代码中给出注释）

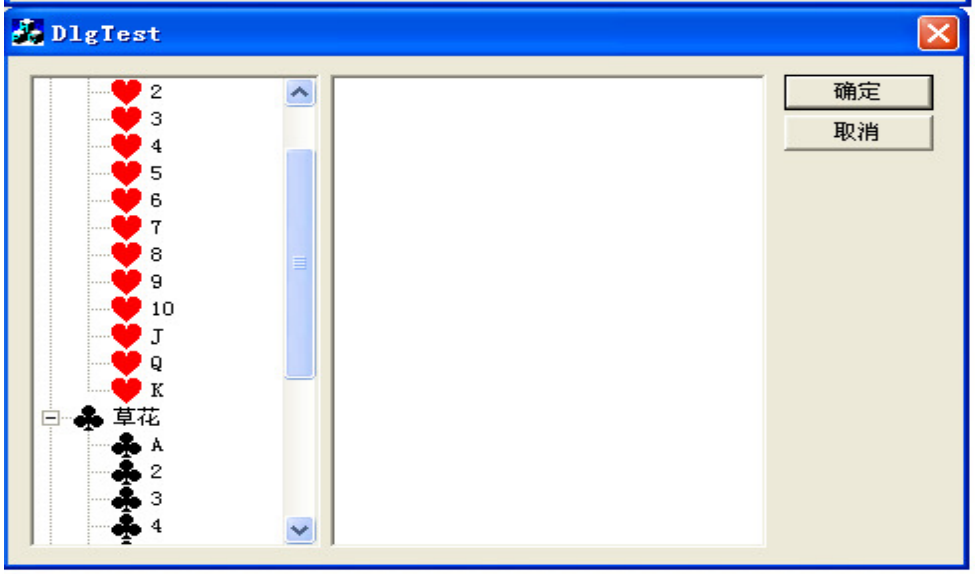
```
        TV_INSERTSTRUCT tv;
        char *szcard[4]={"方块","红桃","草花","黑桃"};
Char*sznum[13]={"A","2","3","4","5","6","7","8","9","10","J","Q","K"};
        CTreeCtrl* tree=(CTreeCtrl*)GetDlgItem(IDC_TREE1);
        HTREEITEM hItem;
        int i,j;
        tree->SetImageList(&m_SmallImageList,TVSIL_NORMAL);
        tv.item.mask=TVIF_TEXT|TVIF_IMAGE|TVIF_SELECTEDIMAGE;
        for(i=0;i<4;i++)
        {
                tv.hParent=NULL;        //外层，没父亲，孤独的孩子
                tv.hInsertAfter=TVI_LAST;//插入到最后
                tv.item.pszText=szcard[i];//文本是什么
                tv.item.iImage=i;//指定表项的图像
                tv.item.iSelectedImage=i;//指定表项选择时的图像，这里我们与上一个图像用一样的
                hItem=tree->InsertItem(&tv);//怎么样？这就插进去了，插的方式都在上面的结构体中定义了，返回了一个句柄，我们下面要用，不用可以不将其取下来
                for(j=0;j<13;j++)
                {
                        tv.hParent=hItem;//这儿就用到上面返回的句柄了，相当于上面树中的那个单指针的树
                        tv.hInsertAfter=TVI_LAST;
                        tv.item.pszText=sznum[j];
```

```
                    tv.item.iImage=i;
                    tv.item.iSelectedImage=i;

                    tree->InsertItem(&tv);
                }
            }
```

[code]

大家可以看看效果，当然，还可以改改属性再看看效果

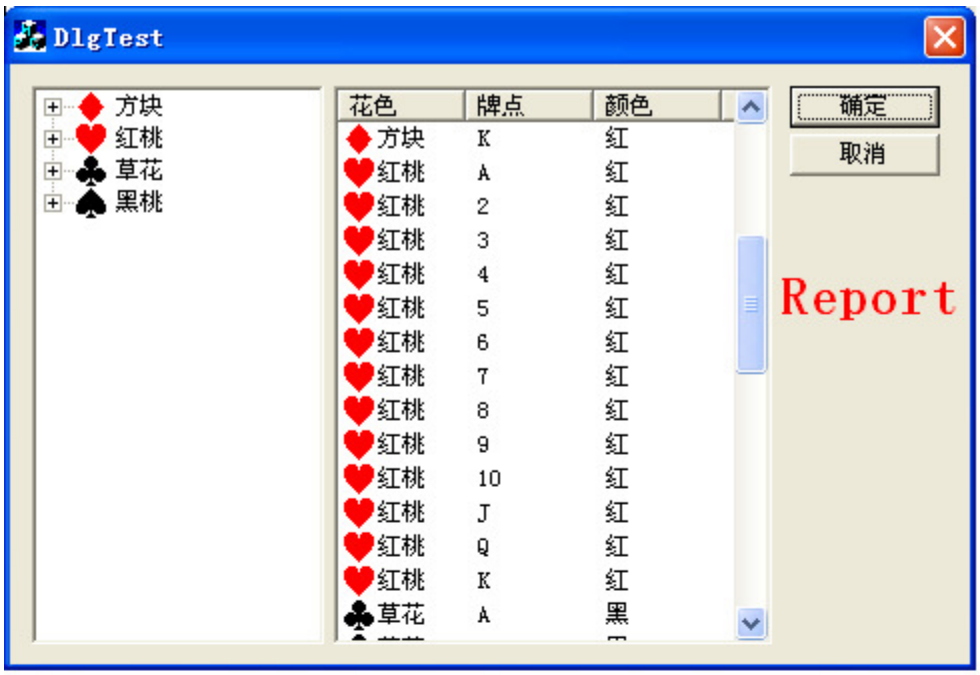


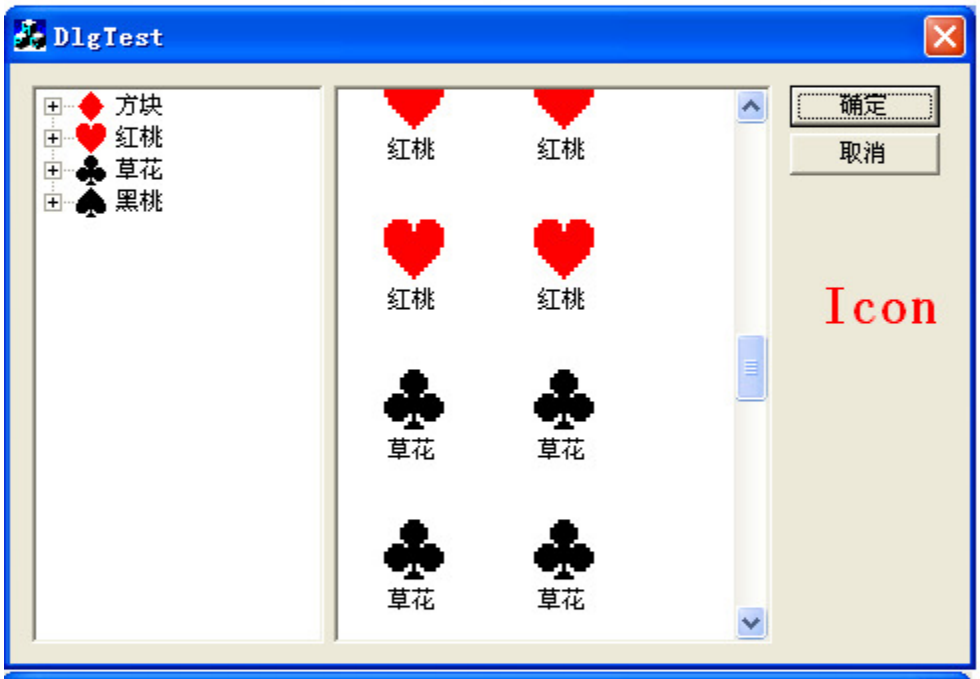



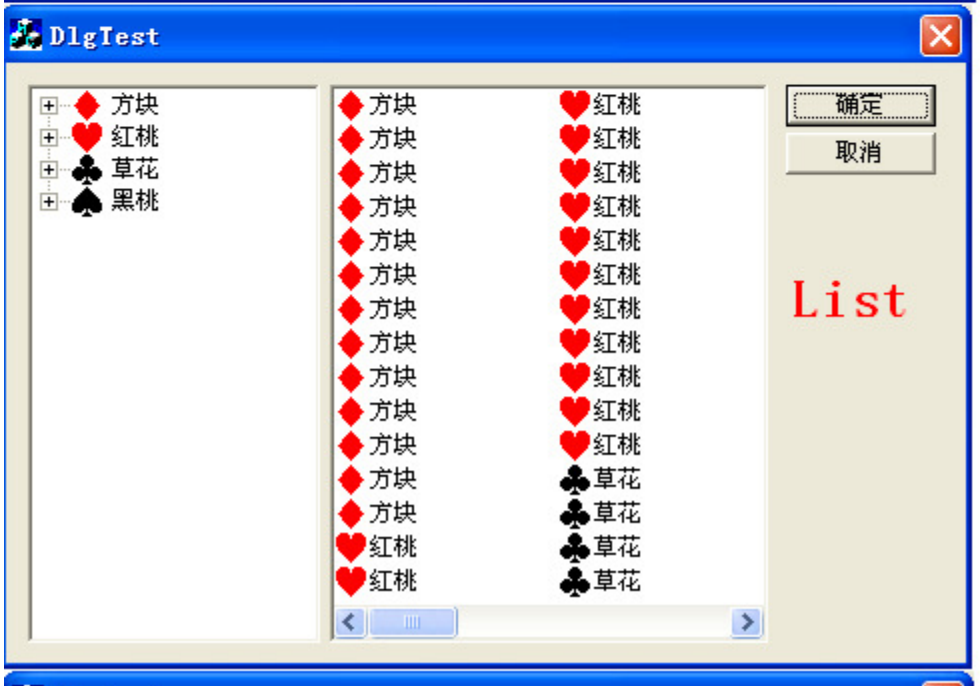
大家将 List  的属性第三个选项卡的“查看”后的属性改为“Report”，这样，我们就可以看到列表视图的变成有标题方案了，我们下面就对列表视图编程（当然，我们也在程序中给大家注释了，相应的结构我们就不在这儿给大家列出来了）。

```
CListCtrl* pList=(CListCtrl*)GetDlgItem(IDC_LIST1);
LV_COLUMN lvc;
LV_ITEM lvi;
char *szTitle[]={"花色","牌点","颜色"};
char *szColor[]={"红","红","黑","黑"};
pList->SetImageList(&m_SmallImageList,LVSIL_SMALL);
pList->SetImageList(&m_LargeImageList,LVSIL_NORMAL);

lvc.mask=LVCF_FMT | LVCF_WIDTH |LVCF_TEXT |LVCF_SUBITEM;//列属性
lvc.fmt=LVCFMT_LEFT;         //列对齐
for(i=0;i<3;i++)         //插入列
{
        lvc.pszText=szTitle[i];         //列标题
         lvc.cx=pList->GetStringWidth(szTitle[i])+40;//设置列宽
        lvc.iSubItem=i;                          //第几列?
        pList->InsertColumn(i,&lvc);         //插入列
}
lvi.mask=LVIF_TEXT | LVIF_IMAGE;         //表项属性
lvi.iSubItem=0;
for(i=0;i<4;i++)         //插入表项
{
        lvi.pszText=szcard[i];  //表项文本
        lvi.iImage=i;                  //表项图片
        for(j=0;j<13;j++)
        {
        lvi.iItem=i*13+j;         //表项位置
        pList->InsertItem(&lvi);         //插入表项
         pList->SetItemText(i*13+j,1,sznum[j]);         //以下两句分别插入表项其他属性
        pList->SetItemText(i*13+j,2,szColor[i]);
        }
}
```

当然，我们也可以调节不同的属性，看不同的效果：


DlgTest
方块
红桃
草花
黑桃
花色
牌点
颜色
方块 K 红
红桃 A 红
红桃 2 红
红桃 3 红
红桃 4 红
红桃 5 红
红桃 6 红
红桃 7 红
红桃 8 红
红桃 9 红
红桃 10 红
红桃 J 红
红桃 Q 红
红桃 K 红
草花 A 黑
确定
取消
Report


DlgTest
方块
红桃
草花
黑桃
红桃
红桃
红桃
红桃
草花
草花
草花
草花
确定
取消
Icon


DlgTest
方块
红桃
草花
黑桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 红桃
方块 草花
方块 草花
红桃 草花
红桃 草花
确定
取消
List

DlgTest

对了，感觉好像不对劲，第一个列应该是“牌点”，第二个是“花色”才好，大家自己去改改，再重新看看效果，牛刀就不改了，呵呵。

从我们上面的创建过程来看，我们对于树型视图与列表视图，我们主要的构造方法就是：

先造结构，再插入，也就是说，我们只要插入，就将结构中的所有信息都传递给我们的列表与树型控件了。

好了，有一点遗憾的是，这几个结构，在我们打“.”时，后面没有列表，什么都需要我们自己输上电脑，大家不要输错哟，呵呵，这是唯一的缺憾。

嗯，下面再与大家聊聊天吧，呵呵，树型结构，其实是数据结构中的一个相当重要的结构，可以说，我们现代的 3 D 技术，人工智能等等绝大多数技术，都需要用到树型结构。

甚至我们以前与大家讲过排序树，其实就是一个树型结构的最简单的应用。

对了，问大家一个问题，排序很简单吗？排序的算法很简单吗？可能你会说：“那还不简单？只要算法好，我们就可以用一个函数调用就一切 OK 了。

果真是那样吗？不尽然吧？如果你想将十个数据，甚至百个数据排序，当然不会有问题。

那么，如果一亿个数据，或者百亿个数据，给你排序，你怎么办？可能那么多的数据，你的内存根本就装不下，你怎么排？

再说数据你内存中就装不下，那么程序放哪儿？这一系列的问题你考虑过吗？

当然，现代的编程技术不需要我们考虑这个问题了，因为我们的 Windows 系统用了一种“虚拟内存”的技术。

然而，这个问题给你去解决，你怎么办？呵呵，牛刀给大家出难题了。

例如，牛刀给你一千个数据，只准你的程序中对十个数据排序，那么你怎么用能排序十个数据的程序对一千个数据进行排序？这可是一个大问题啊。

于是，在数据结构中，就引入了“外排序”这一概念。

大家不要以为牛刀说的这些问题仅仅是理论上讲讲，可如果你真正的要去做一个巨型的数据库，这种问题就会显得十分地尖锐与突出。

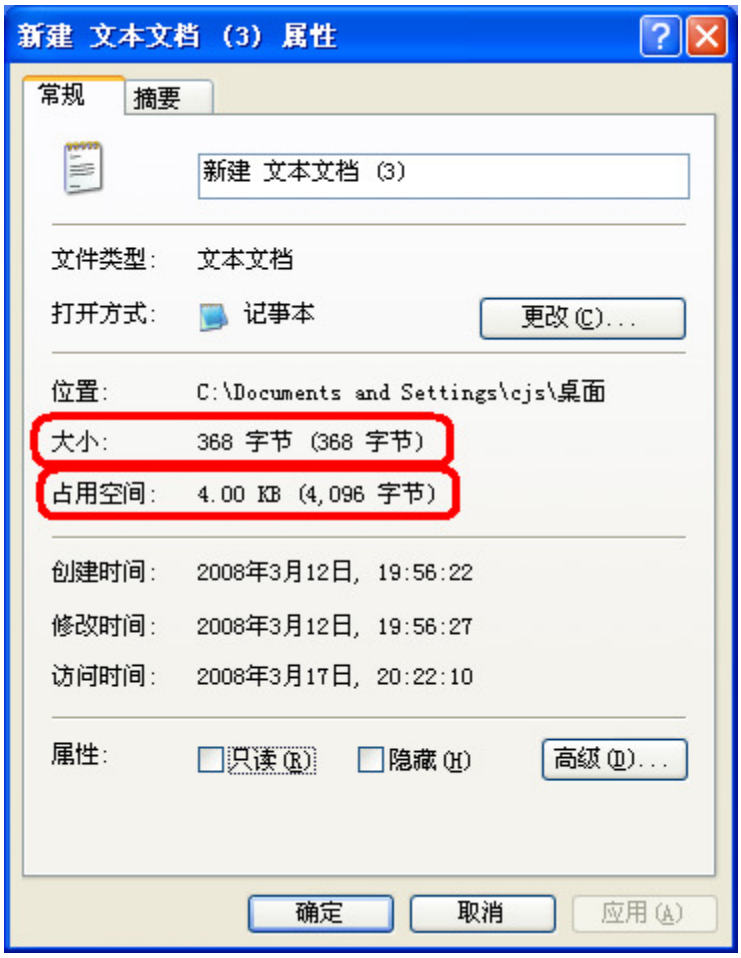
所以，我们一旦出现了那些的种种问题，就会出现了一个新型的课题——数

据调度。

大家用过碎片整理程序吗？

其实所谓的碎片，其实就是我们电脑的特殊的存储方式造成的，可能现代的人对这个问题不关心了，因为我们电脑的存储空间现在太大了，并且我们的电脑运行速度也很快了，可是，这在电脑出现的初期，就是一个很大的问题。

因为我们电脑随着格式化的方式的不同，电脑读硬盘上的数据其实一次并不是读一个字节，也不是读 4 个字节，而可能一次只能读取 1024 个字节，也就是说，这 1024 个字节是不可再分的，你的软件，或者数据文件，不管是不是 1024 个字节，都按 1024 个字节的整数倍存储，大家看图：




大家看，牛刀电脑的格式化方式，每个块是 4096 个字节，那么像我们上面那个文档，就会有 4096-368=3728 字节 的空间没有使用，而现在可能这个问题存在得不多了，而以前，我们的电脑硬盘很小的时候，一个很大的文件，就可能以链表的方式存放在不同的地方，于是，就形成了一个零零碎碎的空间分布，而删除一个软件时，就会出现好多零零碎碎的空间，我们称这种空间就叫做碎片。

碎片的整理就是典型的“外排序”。而且一定是“外排序”，除非你的内存比你

的硬盘空间还要大。

转了一圈，大家终于明白了，我们每提出一个课题，都有其实用的价值与实用性了，是吧？

以往牛刀电脑内存很小时，碎片整理程序本身就可能运行半天甚至一天的时间，但现在硬盘大了，内存大了，牛刀几乎不去运行碎片整理程序了，呵呵，并且现代的碎片整理程序速度也很快。

对了，告诉大家一个秘密，碎片整理会使得你的电脑上能存放下更多的东东的哟，呵呵，不是因为你的电脑空间变大了，而是在整理碎片时，将那些没有用的空间给让出来了。（这是真的，空间小到一定的程度，这一部分的空间就没用了，什么原因？

很简单，大家玩过“推箱子”吗？如果箱子推到了边上，那么那一个空间小人就进不去了，于是，那一个空间就相当于报费了，而我们想使用那一个空间，就需要将箱子推开，而推开箱子的过程，就是碎片整理。

**微软积木〇九六：抽丝剥茧**

首先感谢管理员给牛刀专门开了子版块，说实话，一开始还真不习惯，呵呵。

有人已经感到有些东东有难度说，难掌握了，说实话，感觉到难并不奇怪，也不怕，可为什么会感觉到难呢？

因为我们的类将好多好多的细节都给“封装”了，从而使得我们的控件一旦其中出现的东东有些多的时候，就会显得不知所措，应该属于正常。

那么我们这一讲就先抛开 MFC ，从 API 的角度再来谈谈我们的 API 编程。

说实话， API 做出来的软件会显得十分得小巧，其做出的程序的效果也并不比 MFC 差，可惜的是，其主导的思想却不是面向对象的，而是面向过程的。

所以大家不要以为一提到 Windows 就面向对象，其实根本不是那么回事，面向对象只是一种编程的技术，是一种思路，而不是说我们真的做程序就是面向对象的了。

好了，闲话少说，我们接下来就做一个 API 的程序吧，我们也称之为“Windows”程序，怎么建立大家还知道吧？（注：不知道看牛刀以前的教程，牛刀这里不教大家建立过程了。）

好，我们当然不会给大家立刻就讲解程序，我们先与大家讲讲我们一些硬件，先与大家讲讲“键盘”吧，你可以将你电脑的鼠标给拿了，可你怎么也不可能将你电脑的键盘给拨了，是也不是？说起来，键盘的历史可要比咱们的电脑要早得多，可以说是电脑的“爷爷的爷爷”了。

键盘起源于 1874 年的第一台打字机，其实说给你不信，还是由中国的传统印刷术传过去了，不过我们中国人发明的印刷术，我们中国人可能用起来特别得不方便，因为我们中国的汉字太多，而一旦传到国外，那可得势了，外国西文国家只有 26 个字母，与一些标点符号，并且再加上一些特殊的键，可以实现换行，就是上面的滚轴向上旋转一定的距离，当然，回车的功能其实可能需要手工的喽，可能后来有了电机就不需要了。

开始的键盘是机械的，其实使用的不是“复印”的技术，而是“印刷”的技术，这与中国的印刷术不同的是，印刷术是需要排一次版，多次印刷，而外国的打字机只是替代了手工抄写工作。

当然，不知什么时候，有了专用的印刷纸张，人家打印机打印出来的东东也可以直接用油印机去印了，当然中国人是当仁不让，将这种打字机改造了过来，而下面的铅字就不是２６个字母了，而是一个大拼盘，盘子里面全是汉字，打字

的人牛刀估计是不可能实现盲打的，那种打字机牛刀看过，上面的汉字全是反的，想找到相应的字，那就得费一番功夫，呵呵，不过那就是很先进的了，老式的排版，那可真的是铅汉字，一个一个地拿着向印刷盘中拼啊，一手拿着排版盘，一手去找汉字，拼盘那只手越来越重，那些人的肩膀都是一高一低，应该说是都有“职业病”，而每排一版都要跑上上百公里（什么？你不信？呵呵，牛刀瞎说的）。

好了，还看看我们的键盘吧，西文键盘可不需要去找，每一个键都固定了，所以发明了第一个键盘，那当然是按字母顺序排列的了，可为什么我们现在看的这个键盘怪怪的？

那是因为，那时候的机械式的键盘，打印臂是一个键位一个打印臂，打得快时，可能一个打印臂还没有来得及放下，另一个打印臂就上去了，于是在中央就容易卡住，说实话，可能设置者也被卡繁了，于是，为了解决这个问题，就将每一个字母按照使用频率，将使用频率高的字母用使用频率低的字母给分开，从而降低了键盘之间卡臂的风险了。

于是，键盘就变成了我们现在的样子。当然，当出现了电子计算机后，这种卡臂的风险就自然不存在了，但为了纪念这位键盘设置者，他使用的键盘的键位图就被沿用了下来。

还是那句话，现代的键盘是可以按字母顺序的，但人们已经都习惯了这种传统的键盘，与这种键盘已经有感情了。

说实话，大家看键盘，其实还有规律的，我们还可以隐隐约约地看到当然设置者改动的痕迹，大家看，“JKL”不就是在一起？“CDE”也是放在一起“MN”也是放在一起，不是完全打乱的，当然我们也会看到，使用频率较高的 “A”与使用频率较低的“Z”放在一起，从而使得键盘的布局更加合理。

好了，不说了，于是乎，当键盘使用到电脑上后，键盘的扫描码就与 ASCII 码发生了冲突，这是怎么回事呢？

原来啊，计算机的设置之初，是没有键盘的，初期，是插插头，拨插头，不要以为不可能啊，那时候编一个程序可是体力活，呵呵，并且插头之间还不能有接触不良，否则计算机的计算结果就会有误。

后来打孔机与读卡机发明后，人们都用读卡机去输入程序了，程序的输入工作一下子就轻了下来。

也正因如此，所以我们的 ASCII 码的编码顺序是按照字母表的顺序的。

与是 扫描码与 ASCII 码之间就出现了碰车，但是，好的是，我们计算机是自动的，于是这种对应的关系就可以送给计算机自己去完成，于是，我们的计算机处理的代码就会有 扫描码 与 ASCII 码两种。

而其实我们是可以不通过驱动，每个程序都有自己的代码转换程序的，但其实是没有必要，我们只需要使用操作系统的服务就一切都可以了。

但我们可以直接使用扫描码，自己的程序中去实现一些特殊的转换，从而使得我们可以用键盘甚至外接手柄去控制我们的游戏，控制我们的其他软件设置。

当然，现在甚至还有一些键盘可以实现功能键盘，键盘上的铵键可以自主地调换，每调换一次，换一次驱动程序就一切 OK 了，当然，这一切的技术，我们通过程序也都可以实现的，那么我们下面就通过程序的办法，去看看我们的键盘操作：

在 API 编程中，我们所谓的编程，那基本上是消息的响应，当我们按下或抬起一个键或多个键时，就会产生下列的消息：

| | 按下按键 | 抬起按键 |
|---|---|---|
| 非系统键 | WM_KEYDOWN | WM_KEYUP |
| 系统键 | WM_SYSKEYDOWN | WM_SYSKEYUP |

可能你会觉得奇怪，什么叫系统键，什么叫非系统键呢？

原来啊，大家看，我们的电脑上会有好多的功能键如：

Ctrl
Alt
Shift

当我们按这些系统的组合键时，就会产生系统键消息，我们也称之为“快捷键”，例如 Alt+Tab 可以切换当前的程序，那么我们可以想像，如果我们做以下的程序：

```
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
        return 0;
```

如果我们将这三个消息给打劫了，那么我们一旦进入我们的程序，那么我们想使用 Alt+Tab 转移我们的当前运行程序，那将会变成了不可能了，呵呵，厉害吧。

既然我们提到了当前程序，我们就给大家提提下面几个概念：

·活动窗口　　在运行的窗口
·当前窗口　　当前我们正在使用的窗口
·顶层窗口　　显示在顶层的窗口，这个窗口往往其他程序变成当前窗口了，顶层窗口也会在当前窗口之上
·获得焦点　　这个名词不知是从什么地方来的，总之就是将一个非当前窗口变成当前窗口的过程，我们称之为“获得焦点”
·失去焦点　　将一个当前窗口变成非当前窗口的过程，我们称之为“失去焦点。

在获得焦点与失去焦点之时，系统就会向其窗口发送：
WM_SETFOCUS
或
WM_KILLFOCUS
消息，我们可以做如下的程序，去试一个程序：

```
        switch(uMsg)
        {
        case WM_SYSKEYDOWN:
        case WM_SYSKEYUP:
        case WM_SYSCHAR:
                MessageBox(hwnd,"我按下了系统键","系统消息",MB_OK);        //很可惜的是 Alt+Tab 还能够正常响应，原因不知为什么？呵呵，并且只有 Alt 可以响应
                return 0;
        case WM_SETFOCUS:
                MessageBox(hwnd,"本程序获得了焦点","系统消息",MB_OK);
                break;
        case WM_KILLFOCUS:
                MessageBox(hwnd,"本程序失去了焦点","系统消息",MB_OK);
```

上面的程序有二个问题，
·一、这个程序是一个“伪死循环”，理论上说，这个程序是退不出来的，原因大家自己分析，所以我们要将上面获得焦点的部分程序都改了。
·二、这个程序只能响应 Alt 按键，并且不能将 Alt+Tab 按键消息给过滤掉。

也就是说，我们上面程序的目标并没有达到。

牛刀的程序也就是这样，实践出来的，而不是单单给一个结论，但有些书上却说可以阻止 Alt 的一切操作，至少在我们的程序中，达不到这个目标。

下面我们就来看看我们的铵键焦点的消息，注意喽，不能用 MessageBox 给出消息，因为一旦出现了 MessageBox 消息对话框，窗体就必然失去焦点，就引发了一个 WM_KILLFOCUS 消息，于是，又出现一个对话框，关掉又出现一个，于是就出现了“死循环”。

于是，我们改变程序如下：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        switch(uMsg)
        {
        case WM_SETFOCUS:
                hdc=GetDC(hwnd);
                TextOut(hdc,0,0,"本窗体得到了焦点",16);
                ReleaseDC(hwnd,hdc);
```

```
                break;
        case WM_KILLFOCUS:
                hdc=GetDC(hwnd);
                TextOut(hdc,0,0,"本窗体失去了焦点",16);
                ReleaseDC(hwnd,hdc);
                break;
```

……

既然讲到这儿了，我们就多说几句：

· API 的所有争对窗体的函数都需要有窗体句柄 hwnd

· 当我们在非 WM_PAINT 中获得设备描述表都要使用 GetDC 或者其他相关函数，不能使用 BeginPaint 去获得设备描述表

· 当我们在非 WM_PAINT 中释放设备描述表时，都要使用 ReleaseDC 去释放设备描述表，同时要提供窗口句柄，表明要释放哪一个窗口中的设备描述表

· 请注释向系统申请完东西需要记得归还一般我们同时将两句代码都写出，然后以添加代码的方式编程

如果我们运行程序，可以看到如下的效果：

大家看图中，当我们使用了灰色背景画刷后，我们就会看到文字后有一个白

色的背景框，所以说，我们的文字，如果重叠后，就会出现后面覆盖前面的情况，看下面的程序：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        HDC hdc;
        PAINTSTRUCT ps;
        switch(uMsg)
        {
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                TextOut(hdc,0,0,"汉显测试",8);
                TextOut(hdc,5,5,"汉显测试",8);
                EndPaint(hwnd,&ps);
                break;
```

……

大家看下面的图：

大家看后贴上去的背景档住了先前的汉字，是吧？所以我们想要将这个背景拿掉，就需要使用 API 函数去设置背景模式：

……

```
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                SetBkMode(hdc,TRANSPARENT);
                TextOut(hdc,0,0,"汉显测试",8);
                SetTextColor(hdc,RGB(255,0,0));
                TextOut(hdc,3,3,"汉显测试",8);
                EndPaint(hwnd,&ps);
                break;
```

……

这样，显示效果就很好了。

呵呵，感觉好像牛刀有点跑题了，是也不是？不是谈键盘吗？

不错，我们是谈键盘，可是遗憾的是，我们现在的 Windows 可以有好多的窗口，而我们只有一个键盘，那么我们这么多的窗口怎么去使用这个键盘呢？是也不是？

所以说，我们的原则就是：

·获得焦点的窗口可以获得键盘的输入消息

·也就是说，当前窗口才可以获得键盘消息，而其他的窗口是得不到键盘消息的。

可能你纳闷了，或者你一句话就将牛刀的嘴给封住了——那么，输入法在没有获得焦点时，是怎么得到键盘消息的呢？

那就需要使用到“动态链接库”的技术，当然，即使下一讲，我们也不打算给大家详细讲解动态链接库的技术的，大家在这儿先明白是怎么回事就行了，我们以下内容与接下来的几讲，我们还是讲“键盘”，当然，也可能带点鼠标方面的知识。

再与大家闲聊几句，可能你会在想，牛刀不是与大家在讲 MFC 吗？怎么讲讲讲讲讲起了 API 来了？

·从理解方面来说，MFC 的确是需要很深厚理论基础，当然也需要深厚的 API 基础，如果没有 API 基础，那么你也能说，好多东西能够明白，而好多东西，你根本掌握不了，这就是有些人为什么看到牛刀讲一些特殊的控件时，会有些感到难用的原因了。

·从包装角度来说，MFC 的包装的确是很厉害，甚至将好多的实现细节都给包装进去了，所以我们想要了解我们程序的细节，我们在学习一段 MFC 之后，我们也需要再度回到 API 去理解那些我们以前可能不曾理解的内部细节。

·当我们完全明白了，了解了编程的细节之后，我们新的 MFC 的巨轮就可以启航了。

·当我们完全明白了，了解了 API 的细节，并且有了深厚的 C++功底之后，我们自己就可以研究出我们自己的类库了，到那时候，我们可以很简单地摒弃 MFC 去设计，去构思我们自己的类库。虽然没有必要，但我们必须有这种能力，你说是也不是？呵呵。

**微软积木〇九七：虚拟键码**

说实话，可能大家觉得有点“跑题”，不是讲 MFC 吗？怎么忽然讲到了 API 上来了？

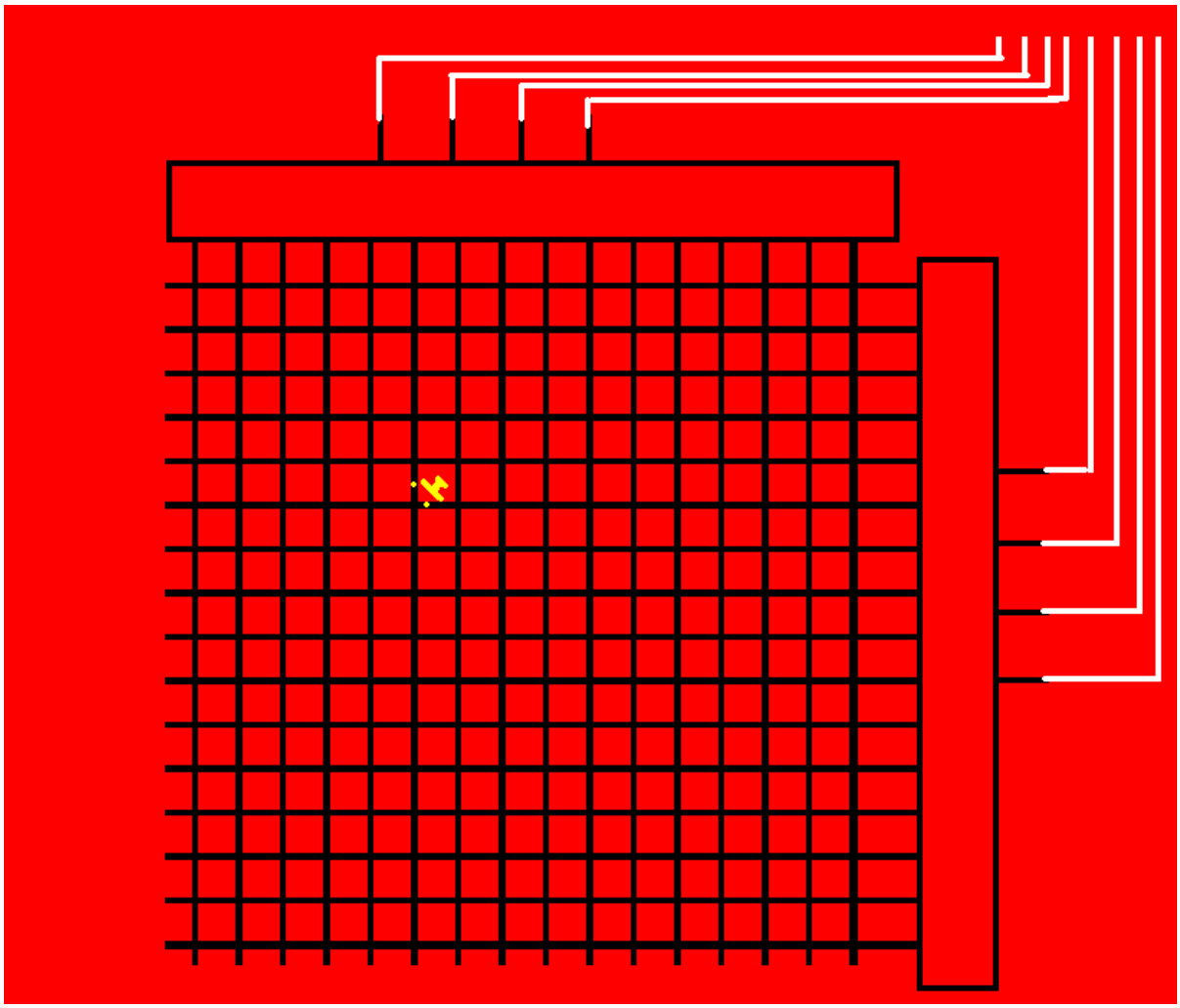
没办法，我们的教程不是向对读者的，我们的教程是面向“实践”的，如果你去买一本关于 MFC 的相关的书籍，那些书的定位肯定就认为读者已经充分掌握了 API 的基础内容了，而我们没办法作这样的定位，我们讲到有关 MFC 解决不了的事，那就必须得回到 API ，将好多的问题与知识点一一击破，否则，我们的教程根本没办法支持下去。

那么我们的还能叫做《微软积木》么？能，MFC 的积木是为我们做好的积木，是成品，而我们这一讲以及接下来几讲，是要告诉大家，我们如果自己去做出一款我们自己的“积木”来，我们是要从木头本身开始，一步一步地教大家如何做“积木”，呵呵。

行了，闲言少叙，我们书归正传：

我们上一讲与大家讲过，我们的键盘其实送进来的键码为“扫描码”，那么什么是扫描码呢？就是我们键盘其实是好多好多的小开关，当我们按下键盘时，有些开关是开的，有些开关是关的，如果是开的，那么键盘就输入相应的键喽，如果是关，那么就不会输入这个字符了吧，说起来简单，那怎么实现呢？

一个办法就是直接使用开关，然后产生一个八位的二进制，去让电脑停止运行，从而去响应我们的键盘传入的信息，大家看图：



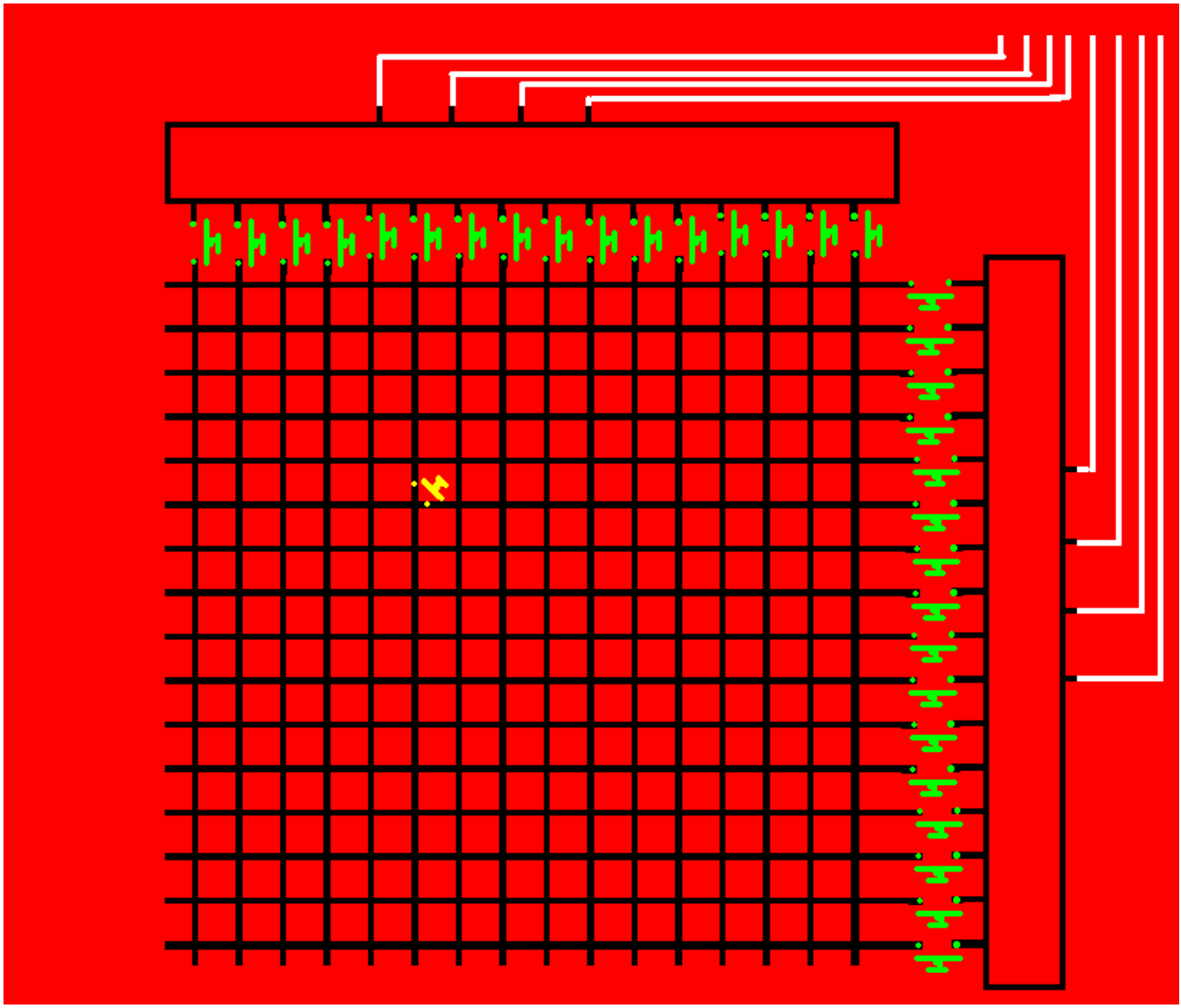
我们图中只给大家做了一个按键（黄色的啊，呵呵，像吗？），其实这里可以有 2 5 6 个按键，当然，真正是用不到这么多的按键的，那怎么办？用几三条线去表示功能键 Alt Ctrl Shift ，所以理论上我们只可以用这八个字节的键盘去设置 64+3=67 个按键。

其实上面的仅仅是理论，我们的键盘当然不能这么设置了，我们键盘可以用十六位线啊，当然也可以用 3 2 位线或 6 4 位，随便啊，

可惜的是问题来了，如果我们增加一倍的线，就会使得我们的键盘的接口增加一倍，现在都是 USB 接口了，老式的接口其实就是有很多眼，好多针的，那种键盘，我们就称之为“传统键盘”吧。

当然，技术是在飞快地发展，我们的键盘也不例外，我们的键盘现在使用了一种技术，称之为“扫描技术”。

什么叫扫描技术呢？也就是说，某一个时刻，键盘中发送一个键码去测试某一个键，如果这个键按下了，那么这个键码就被置成 1（这是牛刀的理论啊，事实上可能不一定是这样），如果这个码没有被按下，就被置成 0 ，大家看下面的图：



请大家看，如果我们测试图中的黄色的那个按键，那么我们就需要将上面的第六个开关关上，右边的第六个开关关上，那么也就是说，我们假设上面四根是控制上面十六个按键的话，那么上面就放上 0110 右边的也放上 0110 然后测试的结果送到进我们准备好的寄存器就一切 OK 了。

当然，你可能说，那也只能测试一个按键的状态啊，是不？

不错，可惜我们计算机测试的速度快啊，反肯定在你的手按下某个键来没来得及抬起，人家就可能已经若干次的全盘扫描就结束了，于是，就可以得到键盘上的按键信息，也正因为现代的键盘都是采用的这种扫描技术，所以其产生的码我们也称之为“扫描码”。

与以前的键盘接口比起来，现代的扫描键盘的接口就要少得多，可能只有四五根脚就够了，甚至像现代的 USB 设备，就是四根脚，并且还有两根是送电的，呵呵。

这说明什么呢？说明现代的 USB 技术，或者其他的一些技术，用的是数据流传送的方案，也就是说，一位一位地传送数据，而不是一次传送八位或者更多。

这样做的好处是什么呢？我们可以很简单地改变一下驱动，就可以将一个十六位的程序变成 3 2 位的。当然理论上讲得通，但现实不会像我们想像得那么理想的。

好了，那么从传统的机械键盘到扫描键盘，我们的扫描键盘其实完全有能力实现直接扫描出 ASCII 码，但还是为了兼容，我们的扫描码其实与机械键盘一样，其实应该属于一种键位码，举个例子吧：

大家看我们最上面一排的键盘，键位分别为“QERTY...”是吧？那么他们的扫描码就分别为：

Q 16
W 17
E 18
R 19
T 20
Y 21

事实就是如此，但是我们在 Windows 编程中，我们得不到这些代码，而我们能够得到的，那也只能是——虚拟代码。

我们的 WM_KEYDOWN WM_KEYUP WM_SYSKEYDOWN WM_SYSKEYUP 中的 WPARAM 参数中，就包含这些虚拟代码，我们试着做一个程序让大家看看：

……

```
int i;
char ch[20];
switch(uMsg)
{
case WM_KEYDOWN:
  i=wParam;
  sprintf(ch,"wParam=%d=>%c",i,i);
  MessageBox(hwnd,ch,"扫描码测试",MB_OK);
  break;
```

……

请大家注意，虚拟代码没有扫描码那样晦涩难懂，但虚拟码就是虚拟码，是一键一码，例如，我们传统的键盘中，主键盘上的 1 与副键盘上的 1 虚拟码是不同的，而大写的 A 与小写的 a 虚拟码则是相同的，所以从这一点来说，我们我们虚拟码代码了键位。

当然，好多的虚拟码都不是用 ASCII 表示的，所以其虚拟码都在 Windows 的头文件中定义了相应的宏，供我们使用，例如：

```
#define VK_LBUTTON        0x01
#define VK_RBUTTON        0x02
#define VK_CANCEL         0x03
#define VK_MBUTTON        0x04
#define VK_BACK           0x08
#define VK_TAB            0x09
```

……

是不是怪怪的，VK_LBUTTON 这些不是鼠标的按键吗？不错，可请大家放心的是，你按键盘上的键是绝对不会产生鼠标按键的消息码的，这个大家尽管放心好了。

当然，第三个 VK_CANCEL 则表示同时按下了 Ctrl+Break 键，这个键在 DOS 年代相当重要，可能被称之为“冷启动”，而使用的 Ctrl+Alt+Del 则被称之为“热启动”。

也就是说，不管谁编的 DOS 程序，只要使用 Ctrl+Break ，就可以重新启动这个程序，而不需要关闭电脑。

当然，也会有其他一些虚拟代码喽：

```
#define VK_BACK           0x08
#define VK_TAB            0x09
#define VK_CLEAR          0x0C
#define VK_RETURN         0x0D
#define VK_SHIFT          0x10
#define VK_CONTROL        0x11
#define VK_MENU           0x12
#define VK_PAUSE          0x13
#define VK_CAPITAL        0x14
……
#define VK_ESCAPE         0x1B
……
#define VK_SPACE          0x20
……
```

当然，像 Page Up 啊，左箭头右箭头啊，像 Insert 啊，Delete 啊等等等等，都有相应的虚拟键码。

有一些虚拟键码是给系统用的，例如，我们按下 Prt Sc 键时，系统就会自动地将屏幕上的内容以图片的形式复制到剪贴板中，我们系列教材中的屏幕截图就是使用的这种技术截下来的，呵呵。

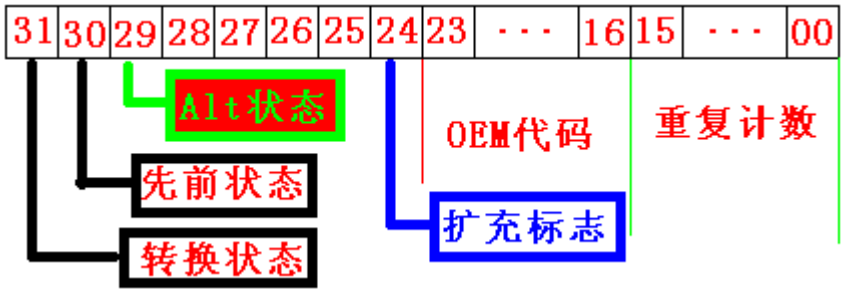
当然，我们做程序也同样可以对这些虚拟键码进行编程，效果一样的。

好玩的是，我们的功能键正常从 F1 到 F12 ，一般我们使用 F1 到

F10 但我们的定义中却有 2 4 个功能键虚拟码的定义，当然显然我们可以很容易地去扩充我们的键盘喽，可能以后或者以前的键盘上就会有二十四个功能键，呵呵。

我们一般用不到那么多，大家知道就行了。

上面我们说了，wParam 中定义了虚拟键码，那么 LPARAM 呢？




重复计数

这个计数估计现在用不到，呵呵，当我们的消息产生的按键消息比我们处理消息的速度快的话，那么系统会将这些消息组合成一个消息，并且将这个重复计数加一，就这样，用不着了。

OEM 扫描码

什么是 OEM 扫描码？就是扫描码呗，呵呵，大家可以做以下程序看一看：

……

```
HDC hdc;
int i;
char ch[20];
switch(uMsg)
{
case WM_KEYDOWN:
  hdc=GetDC(hwnd);
  i=HIWORD(lParam) & 0xff;
  sprintf(ch,"扫描码为：%d",i);
  TextOut(hdc,0,0,ch,15);
  ReleaseDC(hwnd,hdc);
  break;
```

……

大家将上面的代码敲进电脑，然后顺序按键盘上按键，就可以看到我们的扫描码了，大家看“Q”是不是 16 ？W 则为 17 顺序来着的，呵呵。

扩充标志

这个标志表示我们的键盘是传统的键盘还是增强型键盘，其实说白了就是键盘的功能是不是扩充了，例如大家用新式的键盘，可能就会多一些按键，像我们的 Win 键等，都是后加的，甚至有些键盘上会有好多并不是传统键盘中的键，当然喽，那些键，除非人家专业的程序，咱也用不到，可我们要明白就是了，呵呵。

Ａｌｔ状态

这个状态表明 Alt 键有没有被按下，如果按下，则向系统发送的是

WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，而没有被按下，则发送 WM_KEYUP WM_KEYDOWN 消息，仅此而已。

所以如果大家响应 WM_KEYUP 消息，则不需要检测这个标志，肯定为 0，呵呵

键的先前状态与转换状态

大家有没有一直按住某个键的经历？如果一直按住，则按键会一直发送以下消息：

WM_KEYDOWN
WM_KEYDOWN
……
WM_KEYDOWN
WM_KEYUP

也就是如果我们一直按键，则这个键就会持续地发 WM_KEYDOWN 消息，直到接收到 WM_KEYUP 结束（WM_SYSKEYDOWN 与 WM_SYSKEYUP 也一样）。

那么转换状态，就是键有没有被按下，如果没按下则为 0，即正在被释放时，为 0，如果被按下，就为 1，也就是说，响应 WM_KEYDOWN WM_SYSKEYDOWN 这个值就为 0，而如果我们响应 WM_KEYUP WM_SYSKEYUP 时，这个值就为 1

然而，转换状态就不是这样，当响应 WM_KEYUP 时，转换状态为 0 但如果你一直按下键，则这个值就会不停地在 0 与 1 之间不断地变换，从而引起不断的 WM_KEYDOWN 的消息的产生。

·获得键盘状态

像我们如果要获得键盘上的按键状态，怎么办？用下面的函数：

GetKeySatate

这个函数是去测试某个状态，例如我们用下面的函数：

iState=GetKeyState(VK_SHIFT);

大家看如下程序：

……

```
HDC hdc;
char ch[20];
SHORT iState;
switch(uMsg)
{
case WM_KEYDOWN:
  hdc=GetDC(hwnd);
  iState=GetKeyState(VK_SHIFT);
  sprintf(ch,"Shift=%d",iState);
  TextOut(hdc,0,0,ch,9);
  ReleaseDC(hwnd,hdc);
  break;
case WM_KEYUP:
  hdc=GetDC(hwnd);
  iState=GetKeyState(VK_SHIFT);
```

```
  sprintf(ch,"Shift=%d",iState);
  TextOut(hdc,0,0,ch,9);
  ReleaseDC(hwnd,hdc);
  break;
……
```

当然，以上的函数 GetKeyState 只能在消息中响应按键消息的时候，去测试当时的状态，而这个函数却不会自主地去测试键盘的状态，例如大家如果想用 F2 功能键来暂停程序的执行，千万别用：

```
while (GetKeyState (VK_F2) >= 0) ;
```

会死机的，呵呵，程序会死在那儿，除非你在每次测试时，都保证你的消息队列中有了相应的按键消息，如果一旦是最后一个按键消息了，你的程序就进入了“死循环”，这可是真的死循环啊，呵呵。

当然，你如果真的要知道按键的目前状态，你可以使用 GetAsyncKeyState 函数。

有了以上的知识你可能很为你的成就感到高兴，例如你按下 3 键，如果此时检测到 Shift 键被按下，可能你觉得就可以处理成“#”字符了，可你千万不要这样做，因为假如你的用户使用的不是标准键盘，而键盘的第三个符号为“£”，那怎么办？

这一讲的内容可能比较枯燥，但说实话，如果真的用 MFC 去做程序的话，大家一般是想不到要响应 WM_KEYDOWN 和 WM_KEYUP 消息的，并且即使响应了，那些消息中也会将一些必要的参数给传进来了，我们不可能讨论得这么细

我们以后的 API 的章节，我们都会与大家详细地讨论每一个细节。

假如我们不了解细节，我们就只能跟在别人的后面，别人的 MFC 中给了我们什么，我们就只能使用什么。

说实话，我们讲的这些，其实就是那细细的细纱，我们的目的不是为了搭积木，我们的目的是为了“盖楼”，我们的“积木”也不是“积木”，而是盖楼用的砖瓦水泥。

我们以前是教大家盖楼，而这几章是教大家如何“烧砖”，我们虽然走过不同的路，可是我们的目标却是共同的，我们的目标就是，能够真正地使用自己所学到的知识去做出一款款我们自己的软件。

我们在这个道路上，当然还会有好多的路要走，希望大家跟着牛刀，一步步地走好，走踏实。

**微软积木〇九八：字符消息**

既然讲到“字符消息”了，下面我们做一款编辑器吧，当然功能不会很强大，我们尽量将一些细节的技术讲到。

首先大家打字时，后面都有一个插入符吧？（注：不是光标，是你打的字后面一闪一闪的那东东）

很简单：

CreateCaret　　建立插入符
SetCaretPos　　设置插入符位置
ShowCaret　　显示插入符
HideCaret　　隐藏插入符
DestroyCaret　　销毁插入符

好，我们去建一个瞧瞧（注：这一讲牛刀不打算贴图了，一个是动态的东东不好贴图，再一个就是贴图了就会减少你编程的快乐）。

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        switch(uMsg)
        {
        case WM_CREATE:
                CreateCaret(hwnd,NULL,2,20);
                SetCaretPos(10,10);
                ShowCaret(hwnd);
                break;
        case WM_CLOSE:
                DestroyWindow(hwnd);
                break;
        case WM_DESTROY:
                DestroyCaret();         //销毁插入符
                PostQuitMessage(0);
                break;
……
```

请大家习惯性地将创建与销毁同时编程，否则再高明的程序员也很容易将销毁工作忘掉的。

好，请大家将背景使用白色背景，而不要用灰色，用灰色看不到插入符，牛刀也不知道为什么。

有一个想法，我们应该在软件变成当前窗口时显示插入符，而不是当前窗口时就不显示才好，是也不是？大家说呢？于是，我们可以做如下的改动：

```
……
        switch(uMsg)
        {
```

```
        case WM_CREATE:
                CreateCaret(hwnd,NULL,2,20);
                SetCaretPos(10,10);
                break;
        case WM_SETFOCUS:
                ShowCaret(hwnd);
                break;
        case WM_KILLFOCUS:
                HideCaret(hwnd);
                break;
```

……

当然，如果我们插入符的高度如果能和当前字体的高度一样，那就好了，再说，假如我们的字符写到屏幕上，我们的插入符如果能向后移一个字符的距离，那有多好？

那么我们就需要得到当前字体的一些信息，怎么得到呢？

那就需要用到一个结构：

TEXTMETRIC

有一个函数 GetTextMetrics 可以用当前字体的信息去填充我们这个结构，就这样简单，当然我们如果想要将这些信息保存下来，那么有两个办法，一个办法就是用全局变量，以前牛刀说过，建议大家别那样做，那么另一个办法就是静态变量（当然，如果我们在 MFC 中也可以用成员变量），好那么我们就去做我们的程序：

……

```
        static int cxChar,cyChar;
        TEXTMETRIC tm;
        switch(uMsg)
        {
        case WM_CREATE:
                CreateCaret(hwnd,NULL,2,20);
                SetCaretPos(10,10);
                hdc=GetDC(hwnd);
                        GetTextMetrics(hdc,&tm;);
                        cxChar=tm.tmAveCharWidth;
                        cyChar=tm.tmHeight;
                ReleaseDC(hwnd,hdc);
                break;
```

……

我们为什么用 tmAveCharWidth 作为字体的宽度呢，原来啊，英文的字符是不等宽的，例如

iiiiii

wwwwww

在某些字体下，下面的 w 序列就会比上面的 i 序列要长很多，所以我们

一般使用平均值，当然喽，如果我们用中文字体，那么就无所谓了（当然无所谓了，呵呵，中国汉字，那可是方方正正）

对了，说一句啊，我们一般如果要使用函数对结构体填充的话，一般情况下，都是要传指针进去的，当然特殊的情况是如果函数用的是“引用”的办法去传值，那可以不用传指针，我们在合适的时候要合理地运用。

行了，那么我们下面就与大家详细地聊一聊有关字符消息吧：

其实消息中本没有字符消息，大家还记得那个消息循环吗？

```
while (GetMessage (&msg, NULL, 0, 0))
{
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
}
```

其中的 TranslateMessage 就是负责创建字符消息，其实是遇到一个 WM_KEYDOWN 就建立一个字符消息，也就是说，如果你按一下字母表上的 A 键，那么就会产生下列的消息：

| 消息 | 按键代码 |
|---|---|
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | a(0x61) |
| WM_KEYUP | A(0X41) |

当然，假如你按下 Shift+A 按键，则会产生如下消息：

| 消息 | 按键代码 |
|---|---|
| WM_KEYDOWN | VK_SHIFT(0X10) |
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | A(0x41) |
| WM_KEYUP | A(0X41) |
| WM_KEYUP | VK_SHIFT(0X10) |

事实就是这样，当然，我们先不考虑大小写，我们假如一直按下某个按键呢？会产生如下的消息：

| 消息 | 按键代码 |
|---|---|
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | a(0x61) |
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | a(0x61) |
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | a(0x61) |
| …… | |
| WM_KEYDOWN | A(0X41) |
| WM_CHAR | a(0x61) |
| WM_KEYUP | A(0X41) |

当然，关于怎么产生这种连续发送的消息的，我们上一讲也与大伙讲过了。

当然，如果大家按 Ctrl+A~Z 就会产生一些特殊的控制代码，我们一般可以用着快捷键或特殊用途，如：

| 按键 | 字符代码 | 产生方法 | ANSI C 控制字符 |
|---|---|---|---|
| Backspace | 0x08 | Ctrl+H | \b |
| Tab | 0x09 | Ctrl+I | \t |
| Ctrl+Enter | 0x0A | Ctrl+J | \n |
| Enter | 0x0D | Ctrl+M | \r |
| Esc | 0x1B | Ctrl+[ | |

当然没有给大伙列全，并且好多都可以作为快捷键，那就不会产生字符信息。

你说美国佬是不是有点恶心，大家看下面几个消息：

| | 字符 | 死字符 |
|---|---|---|
| 非系统字符 | WM_CHAR | WM_DEADCHAR |
| 系统字符 | WM_SYSCHAR | WM_SYSDEADCHAR |

什么叫“死字符”，挑明了，就是不是美国佬的字符，恶心就是恶心，呵呵。

什么意思呢？其他好多国家的字符可以也很少，也是拼音文字，那就用不着输入法了，而他们可能就在键盘上设置几个特殊的键，去打出他们自己的国家的相应的字符，就如我们中国的汉语拼音会有“āáǎà”一样，他们都会有自己的相应的字符，那种也就是说键盘上并没有显示出来的字符，被美国佬称之为“死字符”。

呵呵，我们有输入法，我们没必要去测试这种“死字符”了。

显然 WM_SYSCHAR 肯定也是按下 Alt 后产生的字符消息，我们一般用着选择菜单，如果没有菜单，我们当然也可以挪着他用，一样的，这儿挪用不犯法，呵呵。

说到这一点，建议大家千万不要挪用一些系统定义的功能，原因很简单，如果你的按键习惯与其他的软件按键习惯如果不同，你的软件就很可能根本就没人用。

行了，那么咱们就先来做一款编辑软件。

当然我们做得很简单喽，我们假设我们的编辑器只能处理 40 个字符，并且多数事情都简单化：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        static int cxChar,cyChar;
        TEXTMETRIC tm;
        HDC hdc;
        int i;
        int x;
        static int len=0;
        static char ch[40];
```

```
PAINTSTRUCT ps;
switch(uMsg)
{
case WM_CREATE:
        CreateCaret(hwnd,NULL,2,20);
        SetCaretPos(10,10);
        hdc=GetDC(hwnd);
                GetTextMetrics(hdc,&tm);
                cxChar=tm.tmAveCharWidth;
                cyChar=tm.tmHeight;
        ReleaseDC(hwnd,hdc);
        break;
case WM_CHAR:
        ch[len]=(char)(wParam);
        len++;
        InvalidateRect(hwnd,NULL,TRUE);
        break;
case WM_PAINT:
        hdc=BeginPaint(hwnd,&ps);
                TextOut(hdc,0,0,ch,len);
                x=len*cxChar;
                SetCaretPos(x,0);
        EndPaint(hwnd,&ps);
case WM_SETFOCUS:
        ShowCaret(hwnd);
        break;
case WM_KILLFOCUS:
        HideCaret(hwnd);
        break;
case WM_CLOSE:
        DestroyWindow(hwnd);
        break;
case WM_DESTROY:
        DestroyCaret();
        PostQuitMessage(0);
        break;
```

……

很厉害的是，我们的这个小型的编辑器竟然能够处理汉字而一点问题都没有大家自己去试试，还是那话，不贴图喽，呵呵。

当然，如果你真的想做一款理想的编辑器，那这款编辑器可能就显得很是不足喽，首先，回退，换行，等，都没有处理，如果你有兴趣的话，你可以加进去，其实如果你动了脑筋了，是很方便的哟。

显然，我们的这种软件也只有演示作用，大家如果想将好多的功能做得更好，那可真得下得一番功夫，呵呵。

下面我们简要地与大家讨论一下鼠标的消息（用不着单独一讲了，呵呵）

鼠标消息有二十一个，我们一个窗体运行起来，会有标题栏、有些还有菜单栏，还有边框等等，当然当中会有一块白色区域（当然，如果你用灰刷子那么就是灰的喽），我们称之为“客户区”，而其他区域我们称之为非客户区，客户区有对应的十条鼠标消息，非客户区也有十条。

我们的鼠标有左中右三键，每个键都有按下、抬起、双击，三种情况，于是形成九种消息，还有一种是移动，于是，形成了以下廿种消息：

| 区域 | 键 | 按下 | 释放 | 双击 |
|---|---|---|---|---|
| 客 | 左 | WM_LBUTTONDOWN | WM_LBUTTONUP | WM_LBUTTONDBLCLK |
| 户 | 中 | WM_MBUTTONDOWN | WM_MBUTTONUP | WM_MBUTTONDBLCLK |
| 区 | 右 | WM_RBUTTONDOWN | WM_RBUTTONUP | WM_RBUTTONDBLCLK |
| | 移动 | WM_MOUSEMOVE | | |
| 非 | 左 | WM_NCLBUTTONDOWN | WM_NCLBUTTONUP | WM_NCLBUTTONDBLCLK |
| 客 | 中 | WM_NCMBUTTONDOWN | WM_NCMBUTTONUP | WM_NCMBUTTONDBLCLK |
| 户 | 右 | WM_NCRBUTTONDOWN | WM_NCRBUTTONUP | WM_NCRBUTTONDBLCLK |
| 区 | 移动 | WM_NCMOUSEMOVE | | |

大家看，很有规律，左中右就是相应的字母变为 L M R，而后面按下、释放与双击分别为 DOWN UP DBLCLK 而移动则没有左中右之分，就只有一个，于是，形成了以上的廿条消息。

还有一种消息为 WM_NCHITTEST ，代表“非显示区域命中测试，这一个消息优先于其他所有的消息，其参数 lParam 参数中含有鼠标相对于屏幕的 x 和 y 坐标， wParam 这个消息没有用。

当然，我们对于这个消息，一般我们直接传递给 DefWindowProc 默认函数处理，处理结果会返回一个值，表明当前鼠标的位置，有以下几种情况：

| | |
|---|---|
| HTCLIENT | 显示区域 |
| NTNOWHERE | 不在窗口中 |
| HTTRANSPARENT | 窗口被另一个窗口所覆盖 |
| HTERROR | 产生警告 |

只有当传回的值为 HTCLIENT 时，才会产生鼠标消息，也就是说，如果我们这个消息如果不让默认函数处理的话：

```
case WM_NCHITTEST:
        return (LRESULT)HTNOWHERE;
```

那么结果就是：其他的鼠标消息，我们的窗口都接收不到。

大家可以去试试，但请大家注意了，如果你真的那样做了的话，那么你的鼠标在你的这个窗口上就什么用都没了，似乎你程序产生的窗口什么反应都没有了，挺好玩，请大家按键盘上的 Alt+F4 去关闭这个程序，呵呵。

当然，大家可以做以下的程序去测试一下我们这一个消息，请大家注意坐标的变化哟，坐标可是以屏幕左上角为坐标的噢：

……

```
HDC hdc;
char ch[40];
int x,y,z;
switch(uMsg)
{
case WM_NCHITTEST:
        hdc=GetDC(hwnd);
        x=LOWORD(lParam);
        y=HIWORD(lParam);
        z=DefWindowProc(hwnd,uMsg,wParam,lParam);
        sprintf(ch,"x=%d,y=%d,(%d)    ",x,y,z);
        TextOut(hdc,0,0,ch,40);
        ReleaseDC(hwnd,hdc);
        break;
```

……

值得一提的是，新型的鼠标中间可能会有一个滑轮，可以按下，相当于鼠标的中键，也可以滚动，从而产生一种新型的鼠标消息：

WM_MOUSEWHEEL

这个消息的 wParam 的高字组的低八位是一个整数，理论上从 -128~127 变化，（但可能能用的只有 0,120,-120）我们可以用这个滚轮的消息来滚动窗口上的一些东东，当然喽，牛刀不详细与大家举例子了，呵呵。

最后，值得一提的是，我们如果要响应鼠标的双击消息，那么我们必须在定义类时，加上 CS_DBLCLKS 属性，否则鼠标接收不到双击消息，我们一般使用如下的类型：

```
wndclass.style= CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
```

·字符消息与鼠标消息，是我们正常做软件中的常用消息，我们必须对其有一个熟练地掌握过程，加上前一讲我们讲的键盘消息，可以说没有这些消息，我们的电脑几乎就没有办法操作。

当然喽，如果是专业的电脑，也很有可能就没有键盘与鼠标，那我们另说，呵呵，像专业的控制软件、机器人等技术，但肯定会有其他的输入与输出的设备，总之，我们既然学习了 Windows 编程，我们就免不了与这些消息去打交道。

```
case WM_PAINT:
        hdc=BeginPaint(hwnd,&ps);
```

```
                    TextOut(hdc,0,0,ch,len);
                    x=len*cxChar;
                    SetCaretPos(x,0);
                EndPaint(hwnd,&ps);
```

牛刀大哥,这一句没加 break;是故意,还是?反正我试了加不加好像没有影响.令外在 WM_CREATE 中不用 SetCaretPos,好像在 WM_PAINT 中使用 SetCaretPos 就可以了.

呵呵，对不起，这是牛刀忘了，但这儿没有影响是因为：

```
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                        TextOut(hdc,0,0,ch,len);
                        x=len*cxChar;
                        SetCaretPos(x,0);
                EndPaint(hwnd,&ps);
        case WM_SETFOCUS:
                ShowCaret(hwnd);
                break;
```

后面响应消息 WM_SETFOCUS 后，在那个消息后面的 break; 那儿作为出口了，但在这儿因为 ShowCaret 函数没有运行次数的限制，所以也没有报错，但这儿的确应该加 break; 你的判断是正确的。

牛刀这儿的确没问题,但如果这个消息在 WM_CLOSE 消息之前可能就麻烦大了，你试着将消息颠倒看看，你可能就看不到这个窗口了。

你说的在 WM_PAINT 中使用 SetCaretPos 其实是一样的但建议放在字符消息或者按键消息中，因为我们如果放在 WM_PAINT 中那么每刷新一次屏幕就会调用一次这个函数，没必要。

事实上就是这样，我们做程序只要有思路了，只要思路正确了，哪些函数放在哪儿完全是我们乐意怎么去实现的事,牛刀给大家讲这些东东的目的其实就是为了让大家有一个基本的设计思路，而如果说实话作为做软件的话，牛刀整个教程中没有一款完整的软件,而牛刀每设计的一个单独的程序都会教给大家一个知识点，相信牛刀给大家的知识点应该不是枯燥的，那么牛刀就心满意足了，呵呵。

相信一点，牛刀的教程中会有很多很多的漏洞，但即使有这些漏洞，至少也是在电脑上运行了，实践过了的，呵呵，而不是像你想像的那样，牛刀是故意的，怎么能呢？牛刀做程序一般情况下还是比较严谨的，这一个 break; 肯定是在删除一些代码时误删了，牛刀一般做这种程序的过程是这样的：

```
case WM_PAINT:
break;
```

然后是：

```
case WM_PAINT:
     hdc=BeginPaint(hwnd,&ps)

     EndPaint(hwnd,&ps);
break;
```

然后再向其中加入一些代码，当然，如果你熟练了，上面的整个过程可以一气呵成，但思路一定要清晰，再者就是每创建一个句柄和销毁一个句柄也一般都成对地敲进电脑，那样不容易有遗漏。

总之，牛刀以前的教程中曾经说过，一个是编程风格，一个是编程习惯，自己做程序时，或者是学做程序的初期就一定要养成一个良好的习惯。

说实话，牛刀不喜欢那种整天地去看书本理论，或者将整个书本上的程序向电脑上搬，因为那不是正常的程序设计的思路，你将书本上程序搬上电脑可能很简单，如果你一直那样，那么一旦你自己去做一个自己独创的程序，那可能就不知道从何下手。

这也正是为什么好多人学习好多的编程，理论基础很扎实，可就是难上手，每一个程序都在编好后进行好多的调试，就是这个原因。

牛刀原来看有些书上会教大家画程序流程图，大家没有看到牛刀教大家画过吧？为什么？是牛刀不懂么？

不对，原因牛刀教大家的思路就是一种正常的编程思路，是符合人们编程习惯的，而程序流程图的作用是：

先将程序在脑中形成完整的一套形态，然后再用图表表示出来，然后再按照顺序输入电脑，那就是电脑的编程过程了。

而牛刀要告诉大家的是，其实没有必要，我们写程序的整个过程就是一个思考的过程，我们想到了哪里，我们的程序就已经写到了哪里了，我们脑袋中有的是对每一种情况的不同处理，就如牛刀以前教大家的那个“知识块”，想想牛刀教过大家贴图了，一共就是四步，只要做到贴图的工作就必然是这四步，难道这还需要流程图么？

随着学习的深入，我们常用的算法都已经在脑袋中定型，对于这些算法，我们几乎可以信手拈来，甚至一行代码：

```
#define max(x,y) ((x)>(y))?(x):(y)
```

你瞧瞧，就上面那一个求最大值的宏，如果我们展开，画个流程图就需要一张纸，如果你想要改成函数，并且能够处理各种类型的数据可能就需要若干个函数，每个函数都有七八行以上的代码，而我们熟练了，那就是一行代码，一个宏，就是这么干脆利落，并且还通吃一切，是一个万能的求最大值的宏，呵呵，方便就是方便啊。

**微软积木〇九九：点阵图片**

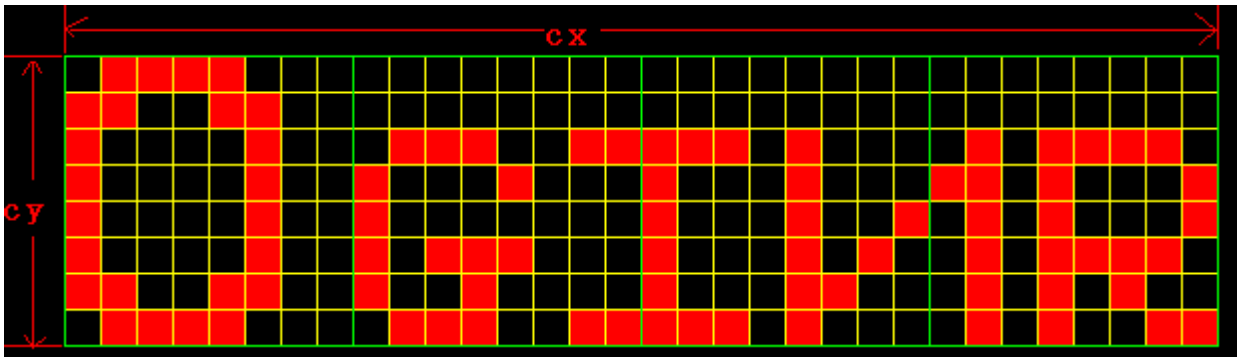
HBITMAP hBitmap;

很显然，前面有 H 打头的，就是一个句柄，我们的图片也是一种资源，作为资源就需要有句柄，当然，对于句柄，我们就需要在使用之前创建，使用之后销毁：

```
hBitmap = CreateBitmap (cx, cy, cPlanes, cBitsPixel, bits) ;
...
DeleteObject(hBitmap);
```

当然，上面建立的第一个为宽度，第二个为高度，第三个为颜色位面，第四个为单个像素的颜色位数，第五个是一个指针，就指向图片的图像区域，当然，那个区域是单字节的，但我们一般定义的时候，都是 32 的倍数。

显然我们一般作程序不太可能用这种办法去创建一幅图片，可既然谈到了，那我们就去创建一个试试吧，我们一般试着玩的话，第三个第四个参数都设置为 1，表示我们创建的是单色位图，下面我们看图，我们创建下面的图片：




好，我们看上面，横向的为 cx，纵向的为 cy，于是，我们定义一串二进制：（我们用 0 代表上面的红色，1 代表上面的黑色啊，于是就出现了下面的字符）

```
10000111111111111111111111111111
00110011111111111111111111111111
01111011100011000001011110100001
01111011011101110111011100101110
01111011011111111011011010101110
01111011010001110111010110100001
00110011011011110111001110101011
10000111100011000001011110101100
```

以上就是二进制了，呵呵，我们要用单色的，不然我们算不过来，呵呵，因为我们整个的宽度为 32 个位，也就是四个字节，共 8*4=32 字节，于是，我们定义则为：

宽度：cx=32

高度：cy=8

颜色面与像素位均为 1

当然，接下来就要构造这幅图片了，我们用下面的四位一组去构造：

```
10000111 11111111 11111111 11111111 =87 ff ff ff
00110011 11111111 11111111 11111111 =66 ff ff ff
01111011 10001100 00010111 10100001 =7b 8c 17 a1
01111011 01110111 01110111 00101110 =7b 77 77 2e
01111011 01111111 01110110 10101110 =7b 7f 76 ae
01111011 01000111 01110101 10100001 =7b 47 75 a1
00110011 01101111 01110011 10101011 =7b 6f 73 ab
10000111 10001100 00010111 10101100 =87 8c 17 ac
```

上面还看得懂吧？是将二进制转换为十六进制，仅此而已，当然喽，在我们 C 语言中，需要将其前面加上 0x 以来表示十六进制，于是就变成了如下形式，我们剩机就定义了吧：

```
        static BYTE bits[]={
                        0x87,0xff,0xff,0xff,
                        0x66,0xff,0xff,0xff,
                        0x7b,0x8c,0x17,0xa1,
                        0x7b,0x77,0x77,0x2e,
                        0x7b,0x7f,0x76,0xae,
                        0x7b,0x47,0x75,0xa1,
                        0x7b,0x6f,0x73,0xab,
                        0x87,0x8c,0x17,0xac
        };
```

想必大家能够明白是什么意思吧？下面我们就去字义我们的图片：

```
                hBitmap=CreateBitmap(32,8,1,1,bits);
```

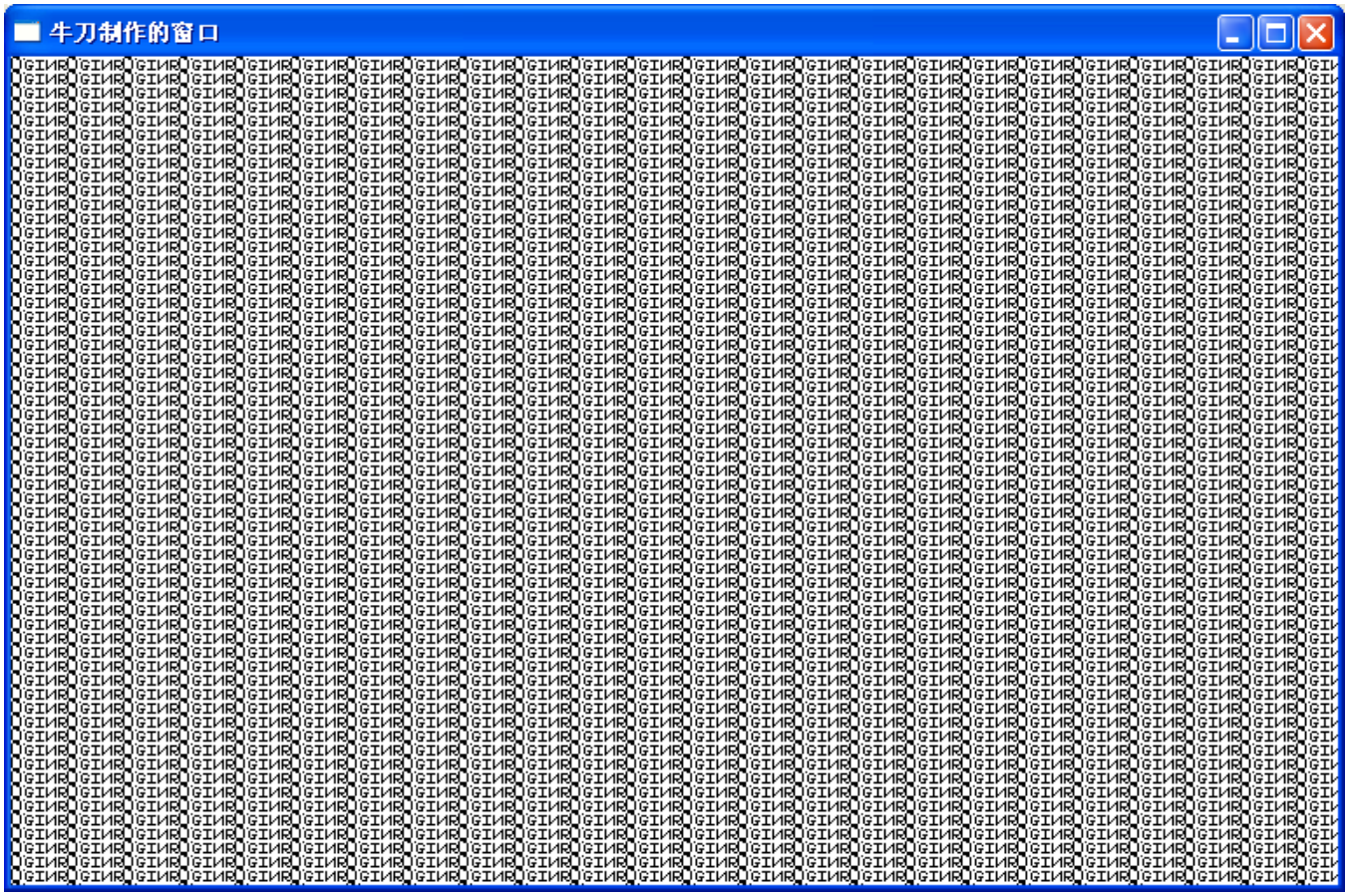
好了，既然句柄定义好了，下面就有两套方案去使用，一是创建一个兼容 DC ，选入位图，然后使用 Bitblt 去贴图，还有一个办法就是使用位图画刷，我们使用后一种：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        static BYTE bits[]={
                        0x87,0xff,0xff,0xff,
                        0x66,0xff,0xff,0xff,
                        0x7b,0x8c,0x17,0xa1,
                        0x7b,0x77,0x77,0x2e,
                        0x7b,0x7f,0x76,0xae,
                        0x7b,0x47,0x75,0xa1,
                        0x7b,0x6f,0x73,0xab,
                        0x87,0x8c,0x17,0xac
        };
        HBITMAP hBitmap;
        HBRUSH hbr;
        HDC hdc;
        PAINTSTRUCT ps;
        RECT rect;
```

```
        switch(uMsg)
        {
        case WM_PAINT:
                hBitmap=CreateBitmap(32,8,1,1,bits);
                hbr=CreatePatternBrush(hBitmap);
                GetClientRect(hwnd,&rect);
                hdc=BeginPaint(hwnd,&ps);
                FillRect(hdc,&rect,hbr);
                EndPaint(hwnd,&ps);
                DeleteObject(hBitmap);
                        break;
```

……

呵呵，难看死了：




不过请大家注意啊，我们上面可没有使用图片资源啊，完全是一个像素一个像素地做出来的，上面只是给大家演示一下，我们其实完全可以从程序中无中生有地创作出一幅图片来，可是真要用这种办法去创建出一幅廿四位的真色彩的位置来，那可就难了。

难了，就交给计算机去完成呗，这是我们通常的做法，于是，我们就出现了“位图资源”：

大家可以打开菜单中的 文件/新建.../[文件]/Resource Script
输入一个名子，按“确定”，就可以创建一个资源了：

创建之后，我们的工作区中就有了资源选项卡“ResourceView”，我们就可以将我们想要创建的资源一样地用 插入\资源... 去插入。

当然喽，我们去插入一幅位图，然后在上面画一些东东，我们下面继续编程，下面就是用 LoadBitmap 去载入这个资源就一切 OK 了：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        HBITMAP hBitmap;
        HBRUSH hbr;
        HDC hdc;
        PAINTSTRUCT ps;
        RECT rect;
        switch(uMsg)
        {
        case WM_PAINT:
                hBitmap=LoadBitmap(hIns,MAKEINTRESOURCE(IDB_BITMAP1));
                hbr=CreatePatternBrush(hBitmap);
                GetClientRect(hwnd,&rect);
                hdc=BeginPaint(hwnd,&ps);
                FillRect(hdc,&rect,hbr);
                EndPaint(hwnd,&ps);
                DeleteObject(hBitmap);
                        break;
```

……

当然，以上程序要顺利运行，需要首先定义几样东东：

·全局变量 hIns 将 WinMain 中传进的程序句柄拿过来，使用 hIns=hInstance; 就一切 OK 了

·包含头文件 #include "resource.h"

不管你加入资源的名称是什么，都是这个头文件，没个改，呵呵，大家包含进来就行了。

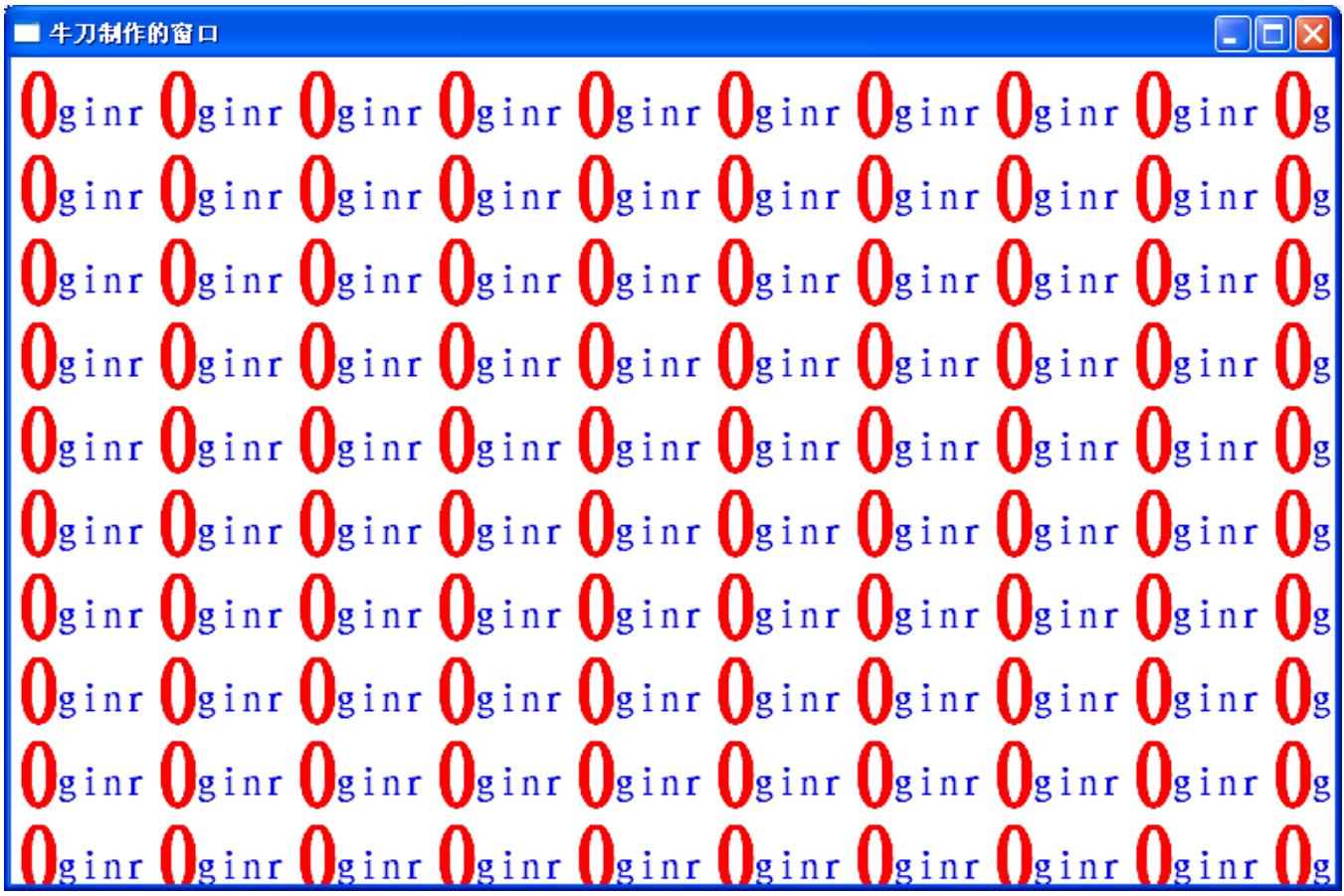
说一点，我们这儿使用了全局变量，这样方便啊，当然，严格地来说，不建议大家这样做，当然以后我们再聊到这儿，我们可以专门的函数，也可以取得当前程序的句柄，这儿大家先将就用着，我们这一讲讲的是位图，就不给大家跑题了，呵呵。

再说说 MAKEINTRESOURCE ，这是一个宏，我们知道，我们的 IDB_BITMAP1 其实是一个资源标识符，其实就仅仅是一个整数：

#define IDB_BITMAP1 101

那么我们总不能将整数取进来吧，没什么意义，于是这个宏，就将这个整数所代表的资源的地址送了过来，并且放到了句柄 hBitmap 中，于是，我们下面就可以顺利地使用此句柄了。

那好，我们看看效果：




这幅图的感觉、效果显然比上一幅图的效果要好多了，大家说是也不是？

嗯，好是好了，可如果我们想要得到图片的大小那怎么办呢？我们的一个结构体为：
BITMAP 我们下面看看这个结构体的定义：

```
typedef struct tagBITMAP
  {
    LONG        bmType;
    LONG        bmWidth;
    LONG        bmHeight;
    LONG        bmWidthBytes;
    WORD        bmPlanes;
    WORD        bmBitsPixel;
    LPVOID      bmBits;
```

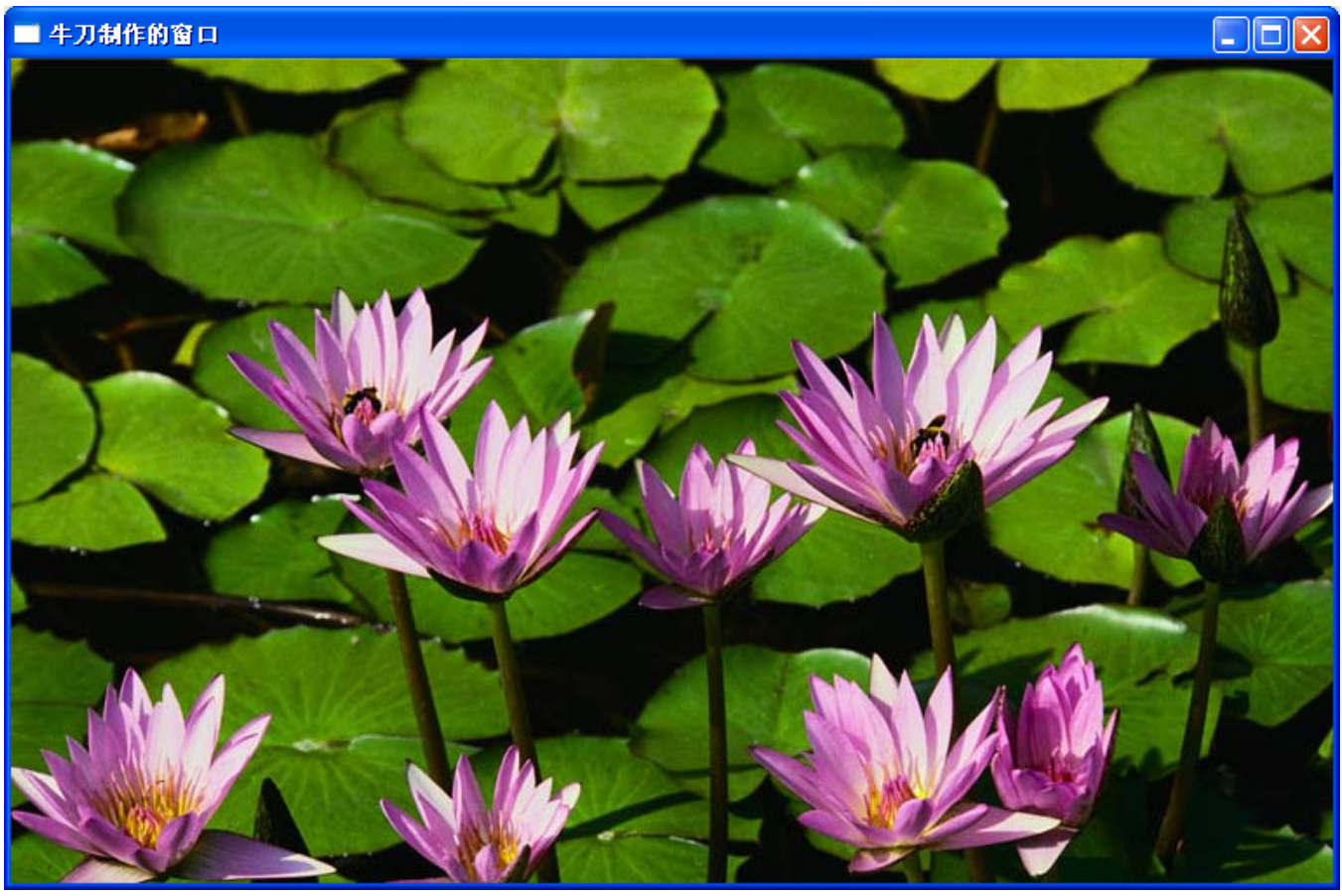
```
} BITMAP, *PBITMAP, NEAR *NPBITMAP, FAR *LPBITMAP;
```

看看什么意思估计能看个八九不离十吧，呵呵，下面我们就准备用这个结构体去得到图片的尺寸：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        static HBITMAP hBitmap;
        static int cxClient,cyClient,cxSource,cySource;
        BITMAP bmp;
        HDC hdc,hdcMem;
        PAINTSTRUCT ps;
        switch(uMsg)
        {
        case WM_CREATE:
                hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstanc
e,MAKEINTRESOURCE(IDB_BITMAP2));          //大家看牛刀是怎么得到程序句
柄的，建议大家用这种办法
                GetObject(hBitmap,sizeof(BITMAP),&bmp);          //这里
我们就用此函数对  bmp  结构进行填充
                cxSource=bmp.bmWidth;
                cySource=bmp.bmHeight;
                break;
        case WM_SIZE:          //当窗体变化时，响应此消息
                cxClient=LOWORD(lParam);
                 cyClient=HIWORD(lParam);          //传回窗口的宽度与高
度
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                        hdcMem=CreateCompatibleDC(hdc);          //创建
兼容 DC
                        SelectObject(hdcMem,hBitmap);          //选入位
图，准备贴图
                        BitBlt(hdc,0,0,cxSource,cySource,hdcMem,0,0,S
RCCOPY);
                        DeleteDC(hdcMem);
                EndPaint(hwnd,&ps);
                        break;
……
```

怎么样？有点感觉了吧？呵呵，这里的贴图不也和我们以前的贴图几乎一样？就只是多了一个  hdc  参数而已嘛。

是的，几乎一样，只不过  MFC  将这些细节给封装起来了而已，看效果：




图片很漂亮，可如果你看过这幅图片的话，你肯定会发现，这幅图并没有显示得全，而只是显示了一部分，那么我们用这么小的窗口想要显示全怎么办？

呵呵，可能你会想到了，很简单啊，滚动条。

好，我们下面就来定义一个“滚动窗体”：

也就是在窗体上试图加上滚动条呗，可你想到没有，给窗体加滚动条显得异常地简单，只需要在创建窗体时，加上 WS_VSCROLL 或者 WM_HSCROLL 就可以有效果：

```
hwnd=CreateWindow("MyWindow","牛刀制作的窗口",
        WS_OVERLAPPEDWINDOW |WS_VSCROLL |WS_HSCROLL,
        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAU
LT,
        NULL,NULL,hInstance,NULL);
```

很简单地一改，我们就可以看到效果了：

看到效果很简单，可是如果大家点一点就会发现，空架子，这个滚动条根本就没有用，动都不能动，这是为什么？

简单啊，我们还没有给滚动条编程呢，那么，下面我们就准备给滚动条编程，当然是使用 API 函数喽：

·滚动条的范围设置
SetScrollRange(hwnd,iBar,iMin,iMax,bRedraw);

这个函数设置滚动条的滚动范围，第一、三、四个参数可能大家一看就明白什么意思，第二个参数为 SB_VERT 或 SB_HORZ 分别为水平或垂直滚动条设置范围，而最后一个是布尔值，表示我们是不是重画滚动条，在这里可以设置为 FALSE,一般正常情况下不需要重画，呵呵。

·滚动条的位置设置
SetScrollPos(hwnd,iBar,iPos,bRedraw);

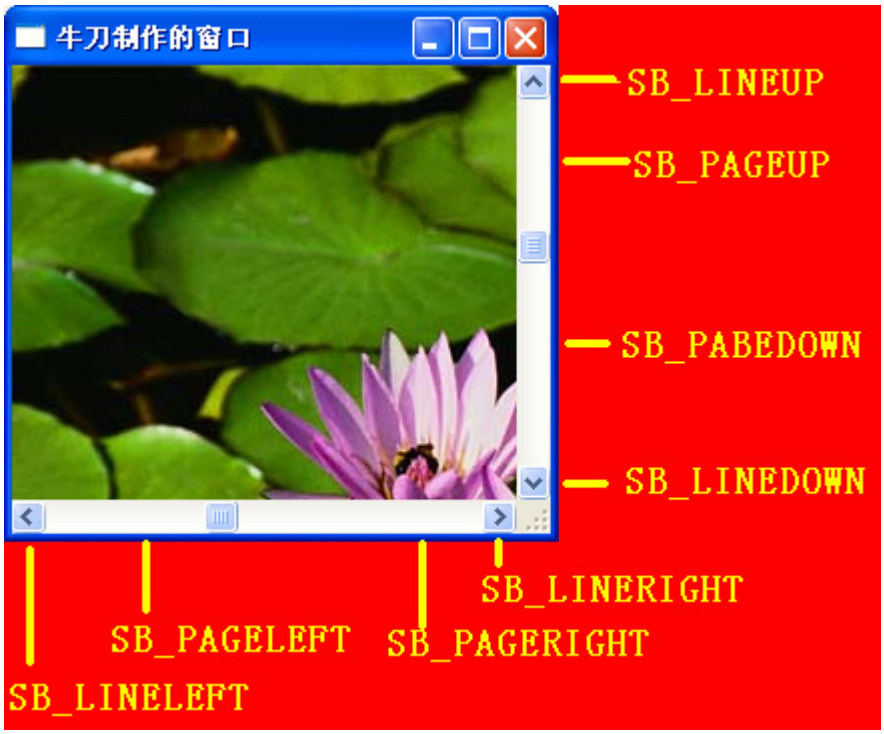
什么意思还看得明白吧？呵呵，这里最后一个参数一般使用 TRUE ，不然我们的显示就太呆板了，效果不好，大家具体使用时，可以改改看看效果，例如我们如果设置为 FALSE 时，如果我们按键头就不会动，直到松了才可以看

到效果，特不方便，所以一般用 TRUE。

当然，我们只要点到这两个滚动条，就会向窗体发送 WM_VSCROLL 或 WM_HSCROLL 消息，其中的 wParam 参数的低字体是一个通知码，有以下的一些取值：

```
#define SB_LINEUP           0
#define SB_LINELEFT         0
#define SB_LINEDOWN         1
#define SB_LINERIGHT        1
#define SB_PAGEUP           2
#define SB_PAGELEFT         2
#define SB_PAGEDOWN         3
#define SB_PAGERIGHT        3
#define SB_THUMBPOSITION    4
#define SB_THUMBTRACK       5
#define SB_TOP              6
#define SB_LEFT             6
#define SB_BOTTOM           7
#define SB_RIGHT            7
#define SB_ENDSCROLL        8
```

大家还能看得懂什么意思吧？呵呵，我们下面来看看下图，鼠标点的位置不同，效果就不同耶：




上图标明了部分通知码的意思，其他可能大家猜也能猜到什么意思了，好了，

那么我们下面就需要对其进行编程。

怎么编程呢？下面我们给出思路：

·响应　WM_CREATE　消息，对位图进行初始化，取得位图大小
·响应　WM_SIZE　　消息，取得窗体的大小，并且对垂直滚动条与水平滚动条进行范围设定（每改变一次都需要设定一次，并且取得得大值）
·响应　WM_HSCROLL WM_VSCROLL　消息，对图片的贴图位置进行设定
·响应　WM_PAINT　　消息，按照贴图位置进行贴图

以上就是基本思路，大家自己去试着做做这款软件看，我们下一讲给出具体实现代码。

## 微软积木一〇〇：滚动视窗

呵呵，一不小心一百讲了，牛刀将这一百讲的所有资源贴出来，有些朋友不能经常上网，可以直接将牛刀写的东东给下载在电脑中，慢慢地看，不过大家看牛刀的文本时，就不是那么图文并茂了，都是单个的文本，并且是笔字连天（呵呵，电脑中没有错字，只有笔字），大家海涵啊，同时也可以为大伙做电子书提供便捷。

下面是上一讲遗留下来的代码，实现了滚动窗体，并且可以很好地显示图片并且可以滚动到没有显示出来的部分。

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        static HBITMAP hBitmap;
        static int cxClient,cyClient,cxSource,cySource,x,y,xMax,yMax;
        BITMAP bmp;
        HDC hdc,hdcMem;
        PAINTSTRUCT ps;
        switch(uMsg)
        {
        case WM_CREATE:
                hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstanc
e,MAKEINTRESOURCE(IDB_BITMAP2));
                GetObject(hBitmap,sizeof(BITMAP),&bmp);
                cxSource=bmp.bmWidth;
                cySource=bmp.bmHeight;

                break;
        case WM_SIZE:
                cxClient=LOWORD(lParam);
                cyClient=HIWORD(lParam);
                xMax=cxSource-cxClient;
                yMax=cySource-cyClient;
                SetScrollRange(hwnd,SB_VERT,0,yMax,TRUE);
                SetScrollRange(hwnd,SB_HORZ,0,xMax,TRUE);
                x=0;
                y=0;
                break;
        case WM_VSCROLL:
                switch(LOWORD(wParam))
                {
                case SB_LINEUP:
                        if(y>0)y--;
                        break;
                case SB_LINEDOWN:
```

```
                    if(y<yMax)y++;
                    break;
                case SB_PAGEUP:
                    if(y>5)y-=5;
                    break;
                case SB_PAGEDOWN:
                    if(y<yMax-5)y+=5;
                    break;
                case SB_THUMBPOSITION:
                    y=HIWORD(wParam);
                    break;
                }
                SetScrollPos(hwnd,SB_VERT,y,TRUE);
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_HSCROLL:
                switch(LOWORD(wParam))
                {
                case SB_LINELEFT:
                    if(x>0)x--;
                    break;
                case SB_LINERIGHT:
                    if(x<xMax)x++;
                    break;
                case SB_PAGELEFT:
                    if(x>5)x-=5;
                    break;
                case SB_PAGERIGHT:
                    if(x<xMax-5)x+=5;
                    break;
                case SB_THUMBPOSITION:
                    x=HIWORD(wParam);
                    break;
                }
                SetScrollPos(hwnd,SB_HORZ,x,TRUE);
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                    hdcMem=CreateCompatibleDC(hdc);          //创建
兼容 DC
                    SelectObject(hdcMem,hBitmap);          //选入位
图，准备贴图
                    BitBlt(hdc,-x,-y,cxSource,cySource,hdcMem,0,0
```

```
,SRCCOPY);//请大家思考 x,y 前为什么要加一个负号
                    DeleteDC(hdcMem);
                EndPaint(hwnd,&ps);
                    break;
……
```

很可惜的是，这款软件的闪动特厉害，为什么呢？牛刀以前隐隐地说过，我们的纸是双面的，后面有一个灯在一闪一闪的，当然，牛刀也说过，窗户纸每被刷新一次，都换上一张白纸，那么，这种闪灯或换白纸的消息就为：WM_ERASEBKGND

不错，就是这个消息，换白纸，那么我们下面就不让他换了就一切 OK 了，那么怎么不让其换呢？打劫呗：

```
    case WM_ERASEBKGND:
            return 0;
```

请大家注意，这里一定用 return 0; 而不要用 break; ，从而使得这个消息不会又被交给系统去处理，这样一来，我们的程序效果就十分出色了。

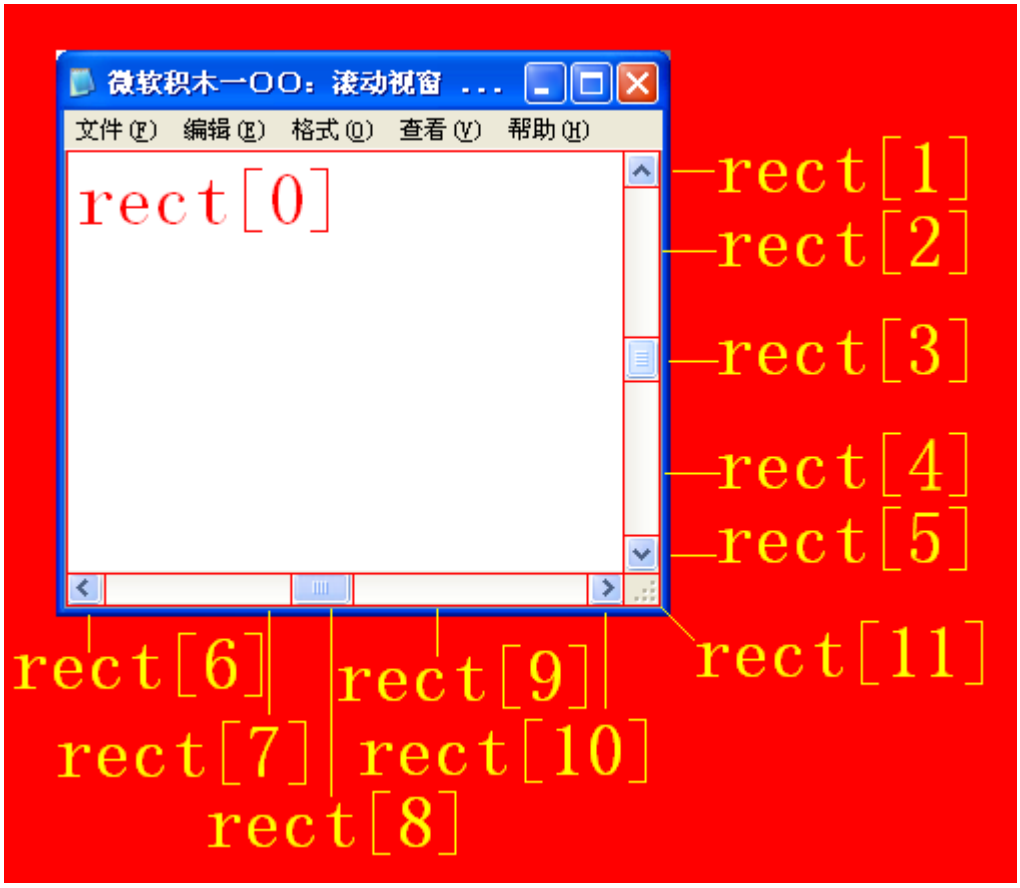
如果我们仅仅与大家谈滚动的窗体以及编程方法，那可能到这儿，这一讲也就结束了，那么我们下面准备干什么呢？

咱们自己去画两个滚动条。

什么？自己画？

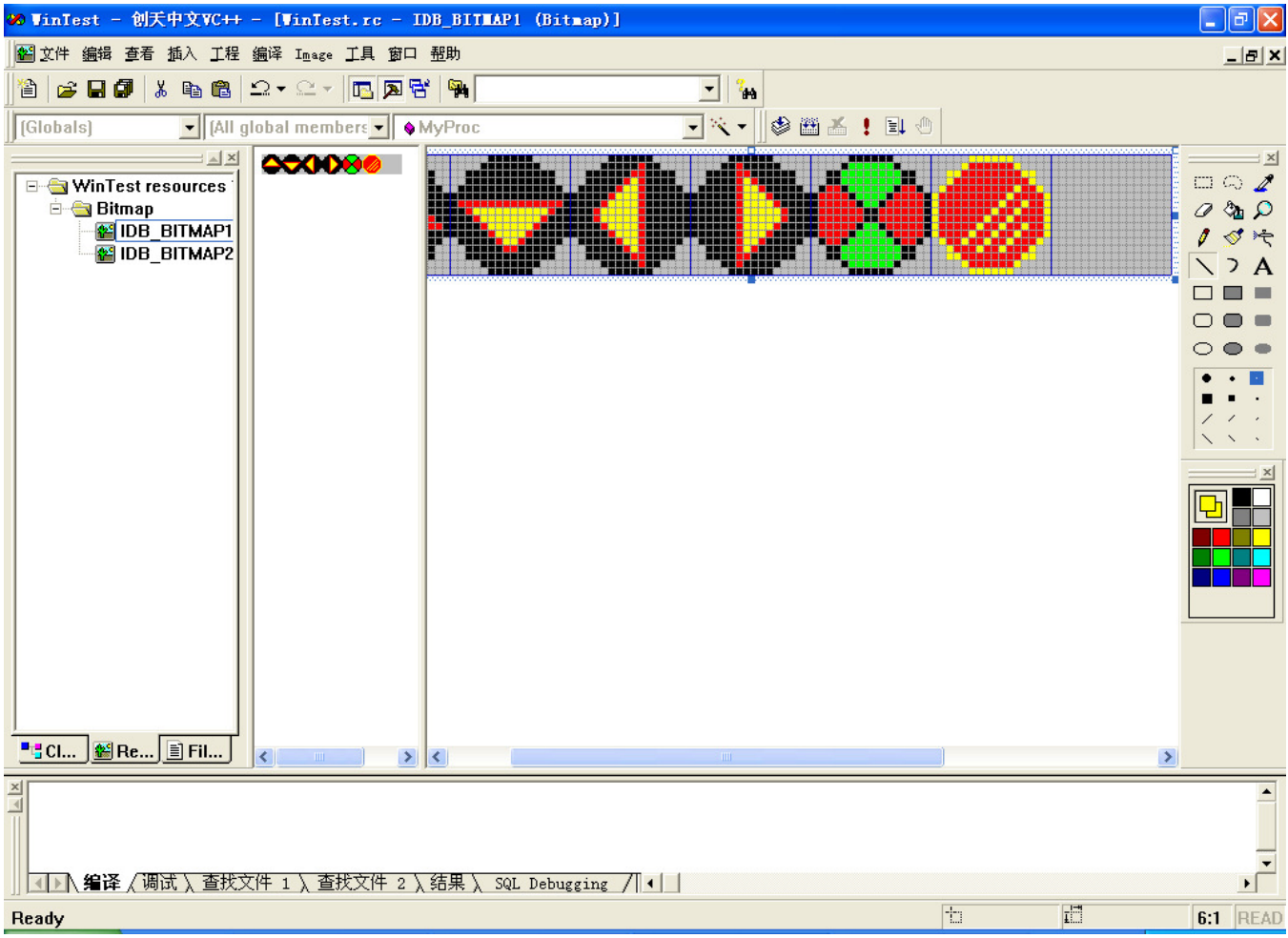
嗯，不错，咱们这次不用滚动窗体了，咱们在窗体内直接用贴图的办法将滚动条贴到窗体上，然后用鼠标的消息去对图片上的一些区域进行测试，在我们这里也可以称之为“命中测试”，这种技术可是在好多的软件中都需要用到的哟。

那么自己画窗体，就需要对窗体上的一些区域进行分区，以便我们了解到，鼠标现在到了哪一个区域了，那怎么分呢？一般我们滚动的窗体可以分为十二个区：




当然，这十二个区我们可以用一个 RECT 结构的数组来表示，然后我们就在鼠标按下消息中去看看我们的鼠标到底落在哪一个区域之中，举个例子如果是落在 rect[5] 区，那么我们就知道里面的图要向下滚动而如果落在 rect[2] 区，那么我们就需要向上翻页，基本的思路就是这样。

那么我们就要去做一幅图片，来表示滚动条喽，看下图：




大家看，图上就是我们的一些滚动条需要用到的资源了，那么我们下面要准备的是什么呢？准备数据，因为我们的滚动条上的滚动柄是有位置的，当然，我们演示作用的滚动条我们就不做得那么完美了，表示个意思，教大家如何滚动就行了，我们就假设滚动条被分成一百份，每份占三个像素，就是 100*3=300　个像素，再加上两头的按钮是　16*2=32　，再加上滚动条的宽度，为 16，一共是 300+32+16=348　个像素，

我们不打算在窗体的左上角就开始显示，于是，我们左上角坐标定为：（50,50），那么我们就可以算得每一个矩形区域的范围，当然喽，我们下面给程序时，大家就知道范围是什么了，我们这儿就先讲原理，呵呵。

好，我们有了范围后，我们就可以测试鼠标的位置了，当然首先要在 WM_LBUTTONDOWN　消息中取得鼠标的坐标：

```
x = LOWORD (lParam);
y = HIWORD (lParam);
```

然后测试在哪个区域中，当然可以用以下的方案：

```
if(x>rect.left && x<rect.righ && y>rect.top && y<rect.bottom)
```

```
{
}
else
{
}
```

但，其实没有必要，因为我们有函数： PtInRect：于是，就需要定义一个 POINT 结构，于是就有了：

```
pt.x=LOWORD(lParam);
pt.y=HIWORD(lParam);
```

然后使用：

```
if(PtInRect(&rect,pt))
{
}
else
{
}
```

当然，以上就是我们的基本构想，甚至贴图，那就需要准备图片喽，我们下面就沿着这条思路去做一个我们自己的“滚动窗体”：

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
        static HBITMAP hBitmap,hbp;
        HDC hdc,hdcMem;
        PAINTSTRUCT ps;
        static RECT rect[12];
        static int x,y;
        int i;
        POINT pt;
        switch(uMsg)
        {
        case WM_CREATE:
                hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstanc
e,MAKEINTRESOURCE(IDB_BITMAP2));
                hbp=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,MA
KEINTRESOURCE(IDB_BITMAP1));
                rect[0].left=rect[0].top=rect[6].left=rect[1].top=50;
                rect[0].right=rect[0].bottom=rect[1].left=rect[2].lef
t=rect[3].left=rect[4].left=rect[5].left=rect[11].left=rect[6].top=re
ct[7].top=rect[8].top=rect[9].top=rect[10].top=rect[11].top=rect[5].b
ottom=rect[10].right=382;
                rect[1].right=rect[2].right=rect[3].right=rect[4].rig
ht=rect[5].right=rect[11].right=rect[6].bottom=rect[7].bottom=rect[8]
.bottom=rect[9].bottom=rect[10].bottom=rect[11].bottom=398;
                rect[1].bottom=rect[2].top=rect[2].bottom=rect[3].top
```

```
=rect[6].right=rect[7].left=rect[7].right=rect[8].left=66;
                rect[3].bottom=rect[4].top=rect[8].right=rect[9].left
=82;
                rect[4].bottom=rect[5].top=rect[9].right=rect[10].lef
t=366;
                x=y=0;
                break;
        case WM_LBUTTONDOWN:
                pt.x=LOWORD(lParam);
                pt.y=HIWORD(lParam);
                if(PtInRect(&rect[1],pt))
                {
                        if(y>0)y--;
                }
                if(PtInRect(&rect[5],pt))
                {
                        if(y<100)y++;
                }
                if(PtInRect(&rect[6],pt))
                {
                        if(x>0)x--;
                }
                if(PtInRect(&rect[10],pt))
                {
                        if(x<100)x++;
                }
                if(PtInRect(&rect[2],pt))
                {
                        if(y>5)y-=5;
                }
                if(PtInRect(&rect[4],pt))
                {
                        if(y<95)y+=5;
                }
                if(PtInRect(&rect[7],pt))
                {
                        if(x>5)x-=5;
                }
                if(PtInRect(&rect[9],pt))
                {
                        if(x<95)x+=5;
                }
                rect[2].bottom=rect[3].top=66+y*3;
                rect[3].bottom=rect[4].top=82+y*3;
```

```
                rect[6].right=rect[7].left=66+x*3;
                rect[7].right=rect[8].left=82+x*3;
                InvalidateRect(hwnd,NULL,TRUE);
                break;
        case WM_PAINT:
                hdc=BeginPaint(hwnd,&ps);
                        hdcMem=CreateCompatibleDC(hdc);
                        SelectObject(hdcMem,hBitmap);
                        BitBlt(hdc,50,50,332,332,hdcMem,x,y,SRCCOPY);
        //画图片，以下以画滚动条

                        SelectObject(hdcMem,hbp);

                        BitBlt(hdc,rect[1].left,rect[1].top,16,16,hdc
Mem,0,0,SRCCOPY);
                        BitBlt(hdc,rect[5].left,rect[5].top,16,16,hdc
Mem,16,0,SRCCOPY);
                        BitBlt(hdc,rect[6].left,rect[6].top,16,16,hdc
Mem,32,0,SRCCOPY);
                        BitBlt(hdc,rect[10].left,rect[10].top,16,16,h
dcMem,48,0,SRCCOPY);

                        for(i=0;i<20;i++)//贴滚动条的滚动边
                        {
                                BitBlt(hdc,66+i*15,382,15,16,hdcMem,9
6,0,SRCCOPY);
                                BitBlt(hdc,382,66+i*15,16,15,hdcMem,9
6,0,SRCCOPY);
                        }
                        BitBlt(hdc,382,382,16,16,hdcMem,80,0,SRCCOPY)
;         //贴角上面的那个图，虽然不能改变大小，但还是贴上去好看，呵呵

                        BitBlt(hdc,382,66+y*3,16,15,hdcMem,64,0,SRCCO
PY);
                        BitBlt(hdc,66+x*3,382,16,15,hdcMem,64,0,SRCCO
PY);

                        DeleteDC(hdcMem);
                EndPaint(hwnd,&ps);
                        break;
……
```

牛刀好像还没有发过这样长的代码，呵呵，效果嘛，当然没有人家原有的滚动条的效果好，因为我们的滚动条根本不能连续滚动，那么想连续滚动怎么办呢？那就需要去响应 WM_TIMER 消息了，当然需要先创建一个计时器，然

后将一些事情交由计时器去做就行了，当然，牛刀这只讲原理，大家可以自己去构思如何实现自己的滚动条，可以使得自己的滚动窗体效果更好。

这里就是牛刀教大家用贴图的办法去制作的第一个控件了，其实我们的所有的控件其实都可以用这种方法制作，但控件中的数据的传递则就需要使用“命中测试”的办法去取得控件中的一些内容，仅此而已。

对了，创建计时器应该不需要牛刀作详细解释吧？其实仅仅是比 MFC 的相应函数中多了一个 hwnd 而已，根本没什么的，我们做程序就不要怕，可以作大胆的设想，因为我们的构思有多厉害，我们的程序做得就有多厉害，真的。

既然讲到构思了，下面就与大家不谈编程了，我们聊一聊大家可能感兴趣的话题——黑客。

我们曾经说过：品一段程序，可以得到一种思维，其实也是这样的，大家不

要以为电脑病毒是电脑中的病毒，其实不是，其实电脑病毒应该是黑客程序员们头脑中的思维。

其实黑客有两种，一种是用其他人的软件去当黑客的，我们称那种黑客为“伪黑客”，呵呵，黑客其实也有假冒伪劣；另一种黑客是自己写软件去当黑客的，牛刀只承认后一种黑客，也就是说，真正的黑客是用技术去干坏事，而不是用软件去干坏事。

虽然说牛刀不敢下断言，但牛刀相信，真正的被公安机关破案的，那都是“伪黑客”，真正的黑客干事，公安机关能破案？至少牛刀怀疑公安机关的能力。

如果说大家不相信吧，那么我们可以来谈谈病毒，为什么杀毒公司总称自己的杀毒能力有多强有多强，但却没有一个普适性的杀毒软件，能将所有病毒一网打尽呢？可以给大家一个答案，那就是：杀毒软件的公司根本没有那个能力。

为什么这样说呢？因为杀毒软件公司可以去分析代码，得出某些代码是病毒，但杀毒软件的公司却不能去分析每一个人头脑中的思维，其实真正的计算机病毒并不是存在于计算机中，而是存在于病毒编写程序员的头脑之中，的的确确是一种思维，是一种高等的思维，这种思维杀毒软件公司是根本拿不去的，这就是为什么没办法做一款普适性杀毒软件的原因。

虽然牛刀不建议大家去当黑客，但是技术还是要学的，其实任何技术都可以拿去为人类造福，也可以拿去毁灭人类的，只是看我们怎么去使用我们手中的技术。

那么我们这一讲就到这儿吧，希望大家在编程的路上走稳，走好，而尽量不要出歪歪心，呵呵。

技术是用来为人民造福的，而不是为了去满足掌握技术的人员的私欲的，我们掌握了技术，就需要合理地使用好我们手中的技术。

我们的思维，别人当然琢磨不透，但如果我们的思维用在了正路上了，那么我们的思维就不需要让别人去琢磨了。

**微软积木一〇一：假冒伪劣**

终于突破了一百讲了，以前讲的都是真的？怎么这一讲讲《假冒伪劣》的了？

呵呵，生活中假冒一般都可能伪劣，可在电脑中，假冒可不一定伪劣哟，都可以当真的用嘀。

还记得上一讲我们讲的那个《滚动视窗》吗？我们看其中的一段程序：
[cpp]
```
case WM_VSCROLL:
    switch(LOWORD(wParam))
    {
    case SB_LINEUP:
        if(y>0)y--;
        break;
    case SB_LINEDOWN:
        if(y<yMax)y++;
        break;
    case SB_PAGEUP:
        if(y>5)y-=5;
        break;
    case SB_PAGEDOWN:
        if(y<yMax-5)y+=5;
        break;
    case SB_THUMBPOSITION:
        y=HIWORD(wParam);
        break;
    }
    SetScrollPos(hwnd,SB_VERT,y,TRUE);
    InvalidateRect(hwnd,NULL,TRUE);
    break;
case WM_HSCROLL:
    switch(LOWORD(wParam))
    {
    case SB_LINELEFT:
        if(x>0)x--;
        break;
    case SB_LINERIGHT:
        if(x<xMax)x++;
        break;
……
```
[/cpp]　我们很想使用键盘，按向上向下键头或者向左向右箭头就可以滚动窗体了，可这怎么实现呢？大家一定想到如下的程序：[cpp]
```
case WM_KEYDOWN:
    switch(wParam)
    {
```

```
case VK_LEFT:
        if(x>0)x--;
        break;
case VK_RIGHT:
        if(x<xMax)x++;
        break;
case VK_UP:
        if(y>0)y--;
        break;
case VK_DOWN:
        if(y<yMax)y++;
        break;
}
SetScrollPos(hwnd,SB_VERT,y,TRUE);
SetScrollPos(hwnd,SB_HORZ,x,TRUE);
InvalidateRect(hwnd,NULL,TRUE);
 break;[/cpp]
```

做出以上的程序固然没有错，可是大家想到一点吗？如果我们一旦想要改变某项功能，例如如果我们单击向上的滚动条滚动时，我们需要每次移动两个值的位置，那么我们就需要改两处，当然，如果我们有更多的不同地方还需要使用这一段代码，可能又要做全部的改变，一步改变不到位，我们的程序就有可能出现问题。

对了，现在讲得相应地要有点深度了，每次给出大段大段的代码大伙还习惯吧？呵呵，这些代码也是牛刀一个字符一个字符地敲进去的哟，大家最好也能敲一遍。

当然，我们对于上面内容有两套方案可以解决，第一套方案就是：函数，我们将我们多次需要用到的程序段变成函数，到时间要用到就调用相应的共同函数就一切 OK，我们如果有改动就去改函数。

还有另一种方案，就是我们这一讲要重点讲的，咱们来一个“假冒消息”，可千万注意哟，再重申一遍，咱们这个假冒可不伪劣哟。

发送消息的函数为 SendMessage 里面的参数我们给大家做段程序大家就明白了：[cpp]

```
case WM_KEYDOWN:
switch(wParam)
{
case VK_LEFT:
        SendMessage(hwnd,WM_HSCROLL,SB_LINELEFT,0);
        break;
case VK_RIGHT:
        SendMessage(hwnd,WM_HSCROLL,SB_LINERIGHT,0);
        break;
case VK_UP:
```

```
            SendMessage(hwnd,WM_VSCROLL,SB_LINEUP,0);
            break;
        case VK_DOWN:
            SendMessage(hwnd,WM_VSCROLL,SB_LINEDOWN,0);
            break;
        }
        break;[/cpp]
```

大家看怎么样，很惊讶吧？不错，发送假消息，呵呵，并且还是立刻执行呢，这种消息并不放到消息队列中，其实说实话，就相当于调用函数，当然还有一个办法，就是想将消息发送到消息队列中，那么使用 PostMessage 就是将消息送到消息队列中，当然这种发送办法有延迟。

以上的就相当于什么呢？你老爸有一款手机，你老妈也有一款，你也有一款（如今生活条件好了，大家都来一款吧，呵呵），那么你老爸发送一条消息给你“孩子，早上好”，你一看号码，噢，老爸向我们好，一看号码，如果你是老妈的号码，那么是你老妈发的。

有一天，你老妈心血来潮，忽然拿你老爸的手机给你来一个“愚人节快乐”，那么你也只好当愚人了，呵呵，因为你一看号码，就会认为是你老爸发过来的，你永远也想不到，你老妈会用这种方式去调教你，是也不是？

如果不告诉你，你是不可能知道哪一个短信是老爸发的，或者是老妈发的。

我们的计算机也是一样，你发送了一条消息给你的程序，你的程序也没有办法判断这个消息到底是你的鼠标点击滚动条发出的，还是你的程序中的某个发送消息的函数发出的，于是，就出现我们上面的那种怪怪情况。

贾不假，白玉为堂金作马。

呵呵，事实就是如此，既然系统给了我们 SendMessage 这个函数，我们就要知道会利用，使得我们的程序显得更加地合理、紧凑。

不旦我们可以假冒消息，甚至消息还能造假怎么造假？自定义消息呗。

牛刀以前说过，什么消息啊，什么句柄啊，什么资源啊……
通通地都是一个整数，仅此而已。

所以啊，我们的消息，完全可以自己用 #define 去定义一个，甚至都可以不去定义，直接用一个整数，如，我们可以发送以下的消息：

```
SendMessage(hwnd,4001,200,3000);
```

这个消息可以发送吗？可以，可你如果这么发送的话，计算机根本就不知道你在干什么，当然喽，这种消息响应也很简单啊：

```
case 4001:
        ...
        break;
```
一样的，真的一样的，就没什么不可以，但为了一目了然，为了我们的消息有意义，还建议大家用 #define 去定义一个自定义的消息为好：

```
#define MY_MESSAGE  200
```

能不能这样定义？能，是肯定能，但是不建议大家这样定义。

为什么呢？因为我们这个整数很可能和系统已经定义的其他消息犯冲，例如假如你定义的数据与 WM_LBUTTONDOWN 数字一样的话，那么如果你响应自己的消息，那么鼠标的按下消息系统就接收不到了，如果你处理了鼠标按键消息，那么你自己的消息怎么办呢？如果你同时处理这两个消息，那么你的程序肯定会报错，那是显然的，你看过以下的程序吗？[cpp]switch(...)
{
case 21:
        ...
        break;
case 21:
        ...
        break;
}[/cpp]  你去试试看，不报错才怪，呵呵。

那么我们怎么定义自己的消息呢？系统有一个专门让我们定义消息的消息 WM_USER ，这个消息是系统用的最大消息，也就是说，小于这个消息的可能都被用了，大于这个消息的消息都没有被用，也就是说，我们可以这样定义：

```
#define MY_MESSAGE  WM_USER+1
```

后面 +1 +2 随便加几都可以，反正都没有被使用过就可以定义。

既然定义了，就可以用 SendMessage 去发送了。

好了，还是来看一看上一讲的程序：[cpp]
```
case WM_LBUTTONDOWN:
        pt.x=LOWORD(lParam);
        pt.y=HIWORD(lParam);
        if(PtInRect(&rect[1],pt))
        {
                if(y>0)y--;
        }
        if(PtInRect(&rect[5],pt))
        {
```

```
                if(y<100)y++;
        }
……
……
……
        if(PtInRect(&rect[9],pt))
        {
                if(x<95)x+=5;
        }
        rect[2].bottom=rect[3].top=66+y*3;
        rect[3].bottom=rect[4].top=82+y*3;
        rect[6].right=rect[7].left=66+x*3;
        rect[7].right=rect[8].left=82+x*3;
        InvalidateRect(hwnd,NULL,TRUE);
        break;
```
[/cpp]　　我们当然没有必要将所有的消息都换掉，我们仅仅换掉按向上键头与向下键头的两个“命中测试”。

当然，我们要先定义消息：

```
#define MY_MESSAGE  WM_USER+12
```

然后，我们要定义两个通知码，一个按向上，一个按向下，也是两个整数罢了

```
#define TEST_LINEUP 2
#define TEST_LINEDOWN 34
```

这个通知码的数据随便敲两个，只要保证不同就行了，那么下面我们就假设我们的 wParam 中传送这两个通知码，而 lParam 我们保持为 0 ，不用她，那么我们就可以顺利地发送消息：[cpp].....

```
        if(PtInRect(&rect[1],pt))
        {
                SendMessage(hwnd,MY_MESSAGE,TEST_LINEUP,0);
        }
        if(PtInRect(&rect[5],pt))
        {
                SendMessage(hwnd,MY_MESSAGE,TEST_LINEDOWN,0);
        }
```
.....[/cpp]　　当然，我们既然发送了这个消息，那么下面我们就去响应这个消息吧：[cpp]……

```
    case MY_MESSAGE:
        switch(wParam)
        {
        case TEST_LINEUP:
```

```
                if(y>0)y--;
                break;
        case TEST_LINEDOWN:
                if(y<100)y++;
                break;
        }
        break;
```
……[/cpp]　　假如你不能懂的话，那牛刀真不知道该怎么将你说懂，如果看牛刀教程看到这儿的话，相信计算机在你的脑袋中其实已经并不是很神秘的了，上面的所谓的消息，所谓的通知码，也只不过是一些整数而已，只不过不同的是，这些整数用宏去定义了一个宏而已，其实我们的计算机也就是这样，我们的计算机只能处理数据，哪里能处理什么消息，我们这所谓的消息只是我们自己一厢情愿罢了其实所谓的消息处理，其真正的意义是“循环”，其实所谓的消息队列，其实就是一个队列，其中放了若干的数据，我们的消息响应就是每次拿出三个数据来处理一来，我们的消息发送就是每次发送三个数据，仅此而已。

也就是说，我们看到的消息处理，其实都是海市蜃楼，真的。

但是事实上也说回来，如果我们编程真的用下面的办法：[cpp]swithc(uMsg)
```
{
case 0x123:
        ...
        break;
case 0x456:
        ...
        break;
case 0x987:
        ...
        ...
```
}[/cpp]　　如果是这样的话，肯定你会很……

呵呵，但是，如果改成下面的代码：[cpp]case WM_CREATE:
```
        ...
        break;
case WM_SIZE:
        ...
        break;
case WM_PAINT
        ...
        break;
        ...
```
...[/cpp]　　那么你看这种代码会舒服得多。

不错，你看下面的代码很舒服，但在计算机中其本质却果真是上面的一种，

其实我们做这些程序，仅仅是给不同的数字去定义不同的功能，仅此而已。

好了，不多说了，聊聊天吧，呵呵，我们学习编程，其实真的是学习编程的思路，牛刀上面告诉你的可能真的是消息的本质，或者正如我们以前所说的，其实根本就不存在着面向对象的技术，根本就不存在着消息机制，根本就不存在着消息循环，那一切的一切，都是我们主观的臆造，但我们却可以不断地用这些新技术去骗自己。

·时代一　汇编替代了机器码　当我们使用汇编了，我们认为我们可以很容易地编程了，可大家想到没有，我们看到的汇编代码是假的，我们的机器永远不可能去执行汇编代码
·时代二　高级语言替代了低级语言　　当我们使用高级语言了，我们会高兴地叫　我们的机器可以算　１＋１＝？　了，我们的机器可以直接使用人类的计算机式子了　　可大家想到没有，我们的机器使用的还是二进制，我们的机器还是只能去执行那枯燥的机器指令
·时代三　多线程替代了单线程　当我们使用了　Windows　了，我们很高兴，因为我们可以一边打字一边听音乐了，因为我们的电脑可以同时运行好多的任务天大的骗局啊，请大家注意，我们电脑中只有一个　IP　，我们的电脑永远只能同时执行一个程序，只不过电脑为我们各个程序分配了不同的时间片
·时代四　画面替代了字符　　大家相信你看到的真是图形操作界面吗？那只是显卡玩的一套小把戏，大家坚信一点，你看到的图片其实在电脑中根本不存在，你认为你看到的是一张二维的画面，甚至有三维的技术，可电脑的内存一直是一维的，永远都是一维的，我们看到的东东，只不过是显卡在我们眼前动了动手脚，从而使得我们感觉面前一片美好，其实不是这样的。
·时代五　面向对象替代了面向过程　不管你信不信，还是那句话，我们的不是面向对象的技术，而是一种“伪死循环”的技术，这个伪死循环使得我们可以利用宏去定义一个个的消息，从而实现消息机制，也正是这一套循环机制，使得我们的面向对向技术得以实现，从而使得我们感觉到，我们用类定义的一个一个的对象，都“活”了起来，其实不是这样的，你的类还是死的，还是没有生命的。你的对象，其实只是编译器玩的一套小小的把戏，在编译的过程中，偷偷地将你定义的类以及类中的所有的函数的名子都改了，从而实现了多态、重载等一系列的技术。
·时代六　电脑替代了人脑　　相信这个时代永远不会来临，原因很简单，我们现代的计算机全是一维的，全是单任务的，我们的电脑永远只能按顺序地去完成我们交给的任务，怎么能替代人脑呢？除非电脑硬件上有重大的革命，否则，至少我们目前的电脑最多能够实现思维模拟，而且能够模拟的思维仅仅是用数学能够证明了的思维，当然电脑还有一个法宝，那就是——穷举法，人工智能等技术用得特多，人们将一些情况全都举了，从而找到了结果，于是就说这是实现了人工智能了，这不是自欺欺人么？这只是用快速的运算去替代人们最愚笨思考办法。

## 微软积木一〇二：犹抱琵琶

千呼万唤屎出来，犹抱琵琶半遮面（注：噢，打错了，好像是“始出来”，哎发贴子打错应该属于正常啊，大家别介意啊）

更改经典不是牛刀的专长，咱们还是来谈谈一种新型的东东——琵琶。

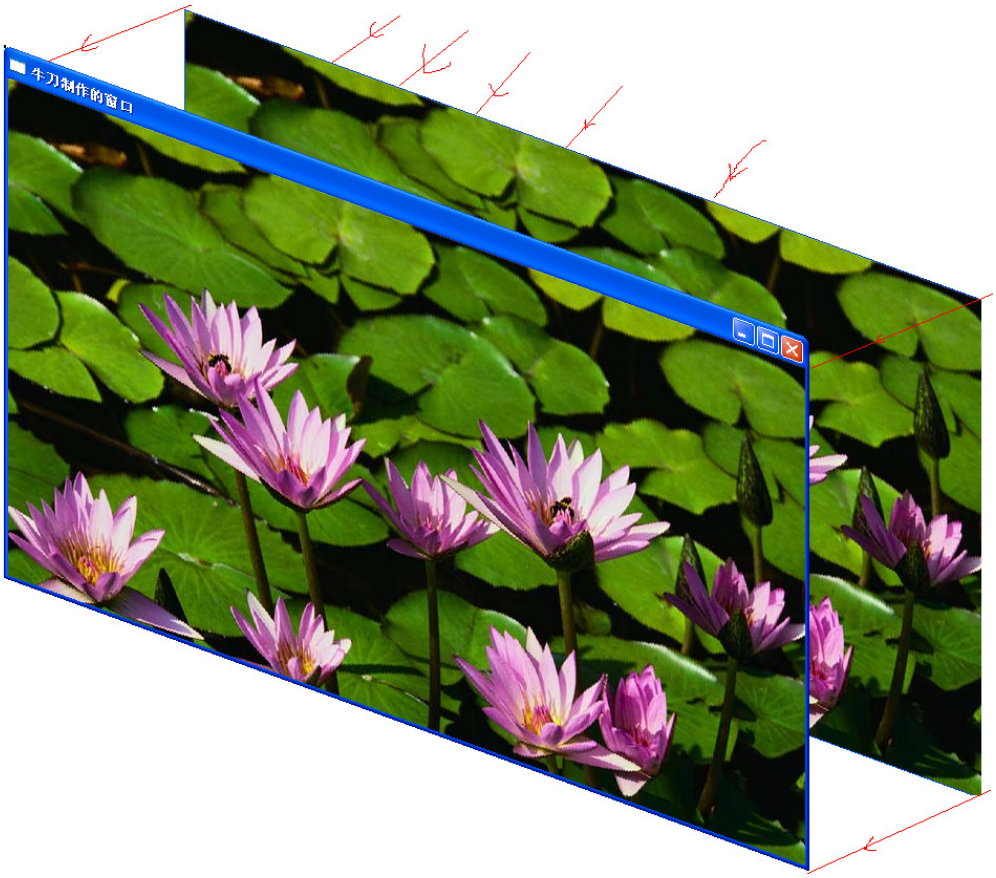
这里的琵琶有什么特点呢？能够遮面，这不是废话嘛，不管怎么说着吧，反正是肯定有半边脸看不见喽，在 PhotoShop 中，这种类似于琵琶的东东被叫做遮照层，在ＶＣ中，被称之为——剪裁区域，在牛刀这里被称之不——档板。

大家还记得牛刀以前给大家讲贴图吗？我们贴一张图可能很容易，也很容易通过 BitBlt 的最后一个参数，通过蒙板去实现透明，而这一讲，我们不给大家那样弄，这一讲咱们要弄一块档板档一下，什么意思呢？牛刀曾经说过，咱们 HDC 这一张纸是有正反面的，每画一次画后，我们的背面就会有一束灯光打到屏幕上，当然这束灯光也是矩形面，并且是都垂直于我们的窗口的，所以没有变形。

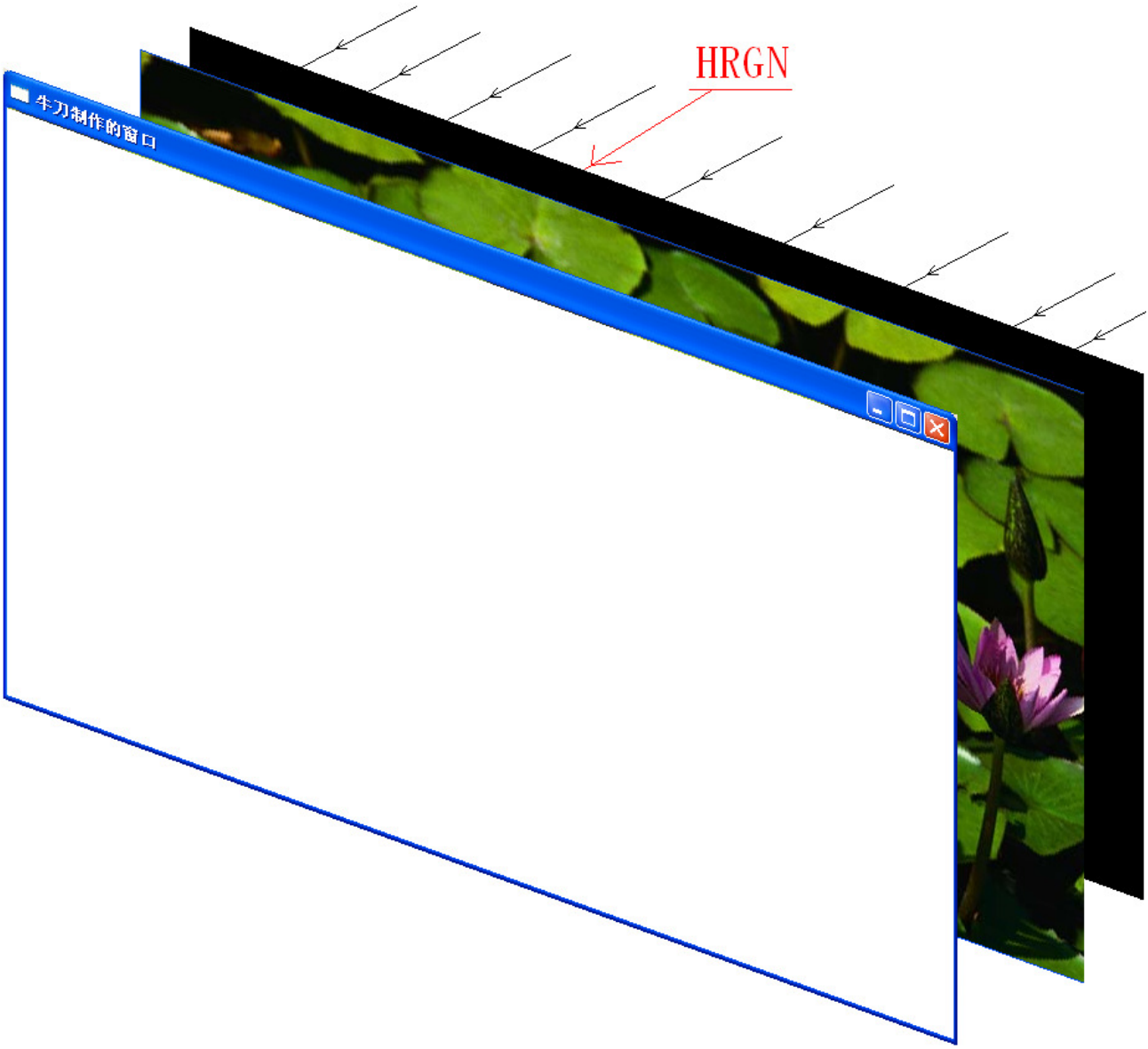
当然，我们可以使用非平行光，那就是我们以前与大家讲过的模式设计部分，我们可以将灯光设为不平行，于是，我们就可以将很小的一样东东变大，或者将很大的东东变小。

显然我们这一讲与大家讲的时候，我们都与大家使用平行光去讨论问题，我们教程中绝对不应该将简单的问题复杂化，否则就与我们的初衷背道而驰了，大家说是也不是？

那么既然讲到了这块档板，那么这块档板是什么呢？是一个句柄，称之为：HRGN



呵呵，牛刀就画向条光线表示平行光线吧，其实也不平行，那么 HRGN 呢？在下面，我们看：




HRGN 就是后面那一块，我们使用黑的，表示这是一块黑布，光透不过来了，于是，我们显示的就是一片白，那么我们如果将这块布掏开一个洞，会出现什么情况呢？

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
static HBITMAP hBitmap;
HDC hdc,hdcMem;
PAINTSTRUCT ps;
RECT rect;
HRGN hrgn; //琵琶
switch(uMsg)
{
```

```
case WM_CREATE:
  hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,MAKEINTRESOU
RCE(IDB_BITMAP2));
  break;
case WM_PAINT:
  hdc=BeginPaint(hwnd,&ps);
   hdcMem=CreateCompatibleDC(hdc);
   SelectObject(hdcMem,hBitmap);
   GetClientRect(hwnd,&rect);
   hrgn=CreateEllipticRgn(rect.left,rect.top,rect.right,rect.bottom);
//创建档板
   SelectObject(hdc,hrgn); //选入档板，这块档板也是一个句柄，和其他句
柄一样地使用
   BitBlt(hdc,0,0,rect.right,rect.bottom,hdcMem,0,0,SRCCOPY);
   DeleteObject(hrgn); //删除档板
   DeleteDC(hdcMem);
  EndPaint(hwnd,&ps);
  break;
……
```

大家看到，档板的使用其他与图片、画刷、画刷、字体等等等等句柄的使用并没有什么太大的区别，当然喽，我们这儿将档板上掏了一个圆形的洞，于是，我们看效果：

感觉挺新鲜吧？水中月、镜中花。

事实就是如此，这种档板的使用，更使得我们的程序做出来可以妙趣横生，甚至用一个函数，就可以对窗口进行切割：

SetWindowRgn

大家看程序：

```
case WM_CREATE:
  hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,MAKEINTRESOU
RCE(IDB_BITMAP2));
    GetClientRect(hwnd,&rect);
    hrgn=CreateEllipticRgn(0,0,rect.right,rect.bottom);
    SetWindowRgn(hwnd,hrgn,TRUE);
    DeleteObject(hrgn);
    return 0;
……
```

大家看效果：

人家将窗口沿着你画的那个圆给裁了，难道人家定义这个为“剪裁区域”呢。

不错，正是这个剪裁区域＋SetWindowRgn　就可以很方便地去实现不规则的窗体。

圆的、方的、三角形的，甚至像瑞星小狮子那样是根本就谈不上规则的特不规则的窗体。

可那些窗是到底是如何实现的呢？

实现的原理其实就是，我们如何去定义我们的剪裁区域，只要这个区域做得好，那么我们使用这块区域去干事就会得心应手了。
CreateRectRgn 创建一个矩形的剪裁区域
当然一定会有
CreateRoundRectRgn 去创建一个圆角矩形的剪裁区域
CreatePolygonRgn 去创建一个多边形的剪裁剪区域
多边形的剪裁区域如果做得好，就可以拿去创建不规则窗体了。
当然更有一个特别舒服的函数：
iRgnType = CombineRgn (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine) ;
这个剪裁区域的函数是将 hSrcRgn1 和 hSrcRgn2 进行组合，组成了新的剪裁区域，并将新剪裁区域送给 hDestRgn，而 iCombine 则是组合方式：

| iCombine 值 | 新剪裁区域 |
| --- | --- |
| RGN_AND | 两个剪裁区域的公共部分 |
| RGN_OR | 两个剪裁区域的全部 |
| RGN_XOR | 两个剪裁区域的全部除去公共部分 |
| RGN_DIFF | hSrcRgn1 不在 hSrcRgn2 中的部分 |
| RGN_COPY | hSrcRgn1 的全部（忽略 hSrcRgn2） |

有了上面的规则，我们就可以合理地使用这个函数，去自由地组合我们自己的剪裁区域了。
我们可以很容易地定义一些我们自己想要使用的剪裁区域，然后去实现一些图片的剪裁，或者窗口的剪裁。

```
LRESULT CALLBACK MyProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
static HBITMAP hBitmap;
HDC hdc,hdcMem;
PAINTSTRUCT ps;
RECT rect;
HRGN hrgn,hrgn1,hrgn2;
switch(uMsg)
{
case WM_CREATE:
  hBitmap=LoadBitmap(((LPCREATESTRUCT)lParam)->hInstance,MAKEINTRESOU
RCE(IDB_BITMAP2));
   return 0;
case WM_PAINT:
  hdc=BeginPaint(hwnd,&ps);
   hdcMem=CreateCompatibleDC(hdc);
   SelectObject(hdcMem,hBitmap);
   GetClientRect(hwnd,&rect);
   hrgn1=CreateEllipticRgn(0,0,rect.right*2/3,rect.bottom*2/3);
   hrgn2=CreateEllipticRgn(rect.right/3,rect.bottom/3,rect.right,rect
.bottom);
   hrgn=CreateRectRgn(0,0,1,1); //请大家注意， hrgn 虽然可以从下面函数得到，但必须创建，一般我们就使用 0,0,1,1 创建一个意思意思就行了，呵
```

呵，否则会出问题的。

```
    CombineRgn(hrgn,hrgn1,hrgn2,RGN_XOR);
    SelectObject(hdc,hrgn);
    BitBlt(hdc,0,0,rect.right,rect.bottom,hdcMem,0,0,SRCCOPY);
    DeleteObject(hrgn1);
    DeleteObject(hrgn2);
    DeleteObject(hrgn);
    DeleteDC(hdcMem);
  EndPaint(hwnd,&ps);
  break;
```

我们这里使用了两个区域的异或集，大家看效果：

看到了吧，就使用这种技术，就可以创造出不规则的显示效果，当然喽，如果我们想去剪裁一个特不规则的显示效果，我们显然可以用一个双重循环，去一个点一个点，一个区域一个区域地往里加，从而使得我们的剪裁区域可以变得任意。

当然喽，剪裁区域还可以使用以下四个函数去直接使用：

```
FillRgn (hdc, hRgn, hBrush) ;
FrameRgn (hdc, hRgn, hBrush, xFrame, yFrame) ;
InvertRgn (hdc, hRgn) ;
PaintRgn (hdc, hRgn) ;
```

大家去用用吧，用用就知道是怎么回事了，想要知道梨子的滋味，就去尝尝。

好，既然说到了剪裁区域，我们就不得不说说一种新概念——路径。

在 PhotoShop 中，这个概念显得比较突出，那么什么是路径呢？路径其实就是大家走过一条路，地上本来就没有路，走的人多了便成了路……哪挨哪呀，嘻嘻。

在电脑中就这么回事，你在地上转个圈，那不叫路，可在电脑中转个圈，还就真的能算数，行了，这次咱们再次到 MFC 中去玩一玩儿，咱们当两栖型的啊，呵呵。

说实话，有时候封装会使得我们的程序结构不明显，但有时候封装也的确为我们提供了便利。

我们还是在 OnDraw 函数中去，去干什么呢？

路径有二个函数：

BeginPath

路径的开始，开始一个路径

EndPath

路径结束，结束一个路径。

什么意思呢？也就是说，当我们一旦执行了 BeginPath 并且没有执行 EndPath 时，我们说我们现在做的事情是在路径中。

一旦在路径中画了图，屏幕上就会什么反应都没有，因为我们是画在路径中，或者说，我们是在路径层上作图，那么我们下面就来看一段程序：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
pDC->BeginPath(); //创建路径层
  //在路径层中
pDC->EndPath();  //结束路径层
}
```

不贴图了，什么效果都看不到，那么我们在路径层中去写几个字吧，当然，最好能选入一个字体：

```
void CMFCTestView::OnDraw(CDC* pDC)
{
pDC->BeginPath(); //创建路径层
CFont font;
font.CreatePointFont(1300,"宋体");
CFont *pOldFont=pDC->SelectObject(&font); //选入字体
pDC->TextOut(0,0,"路径测试");
pDC->SelectObject(pOldFont); //选入老字体
DeleteObject(font);    //删除字体
pDC->EndPath();  //结束路径层
}
```

当然喽，上面这个也不贴图了，仍旧一片空白，为什么呢？因为我们其实并没有在我们的设备描述表上作图，而是在设备描述表中的路径上作图。

也正因为这个图是设备描述表中的，所以基本唯一，所以也用不着单独地使用一个路径句柄的东东来保存路径喽。

那么路径我们怎么使用呢？有以下五种方案：

·假如我们真的要保存路径，可以用以下的函数，将路径转换为剪裁区域：

PathToRegion
·假如我们不需要保存，直接使用，就可以使用以下四个函数：
StrokePath 用画笔沿路径描边
FillPath 用画刷填充路径内部
StrokeAndFillPath 什么功能应该能够想像吧？呵呵
SelectClipPath 将路径用于剪裁区域

值得一提的是，上面五个函数，任意一个函数一用，这个路径就自然消息，不存在了，也就是说，我们不能同时使用两个函数，如果使用两个函数，那么后一个函数就自然失效。

大家想一想就明白了，如果可以使用两次函数，那么既然有了 StrokePath 和 FillPath 函数，那么 StrokeAndFillPath 这个函数不就是多余的了么？

也正因为有这种性质，所以路径不需要句柄，用了就自然消失。

那么有什么用呢？

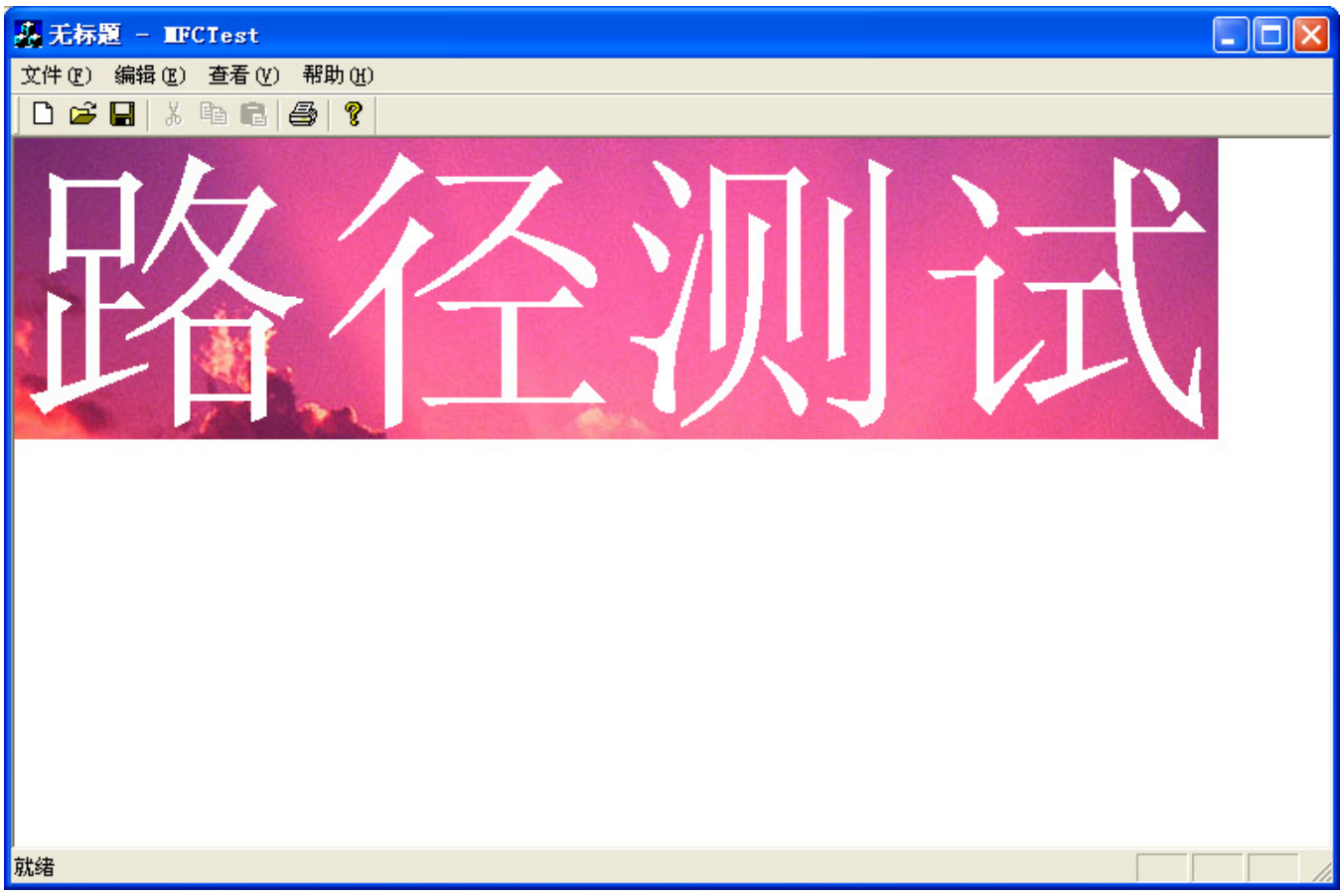
我们先假假地用这个路径变成剪裁区看看效果，我们贴一幅图：

……

```
pDC->SelectClipPath(RGN_AND); //大家看参数啊，也和剪裁区一样吧，呵呵。
CBitmap bmp;
bmp.LoadBitmap(IDB_BITMAP1);
CBrush brush;
brush.CreatePatternBrush(&bmp);
CRect rect;
GetClientRect(&rect);
pDC->FillRect(&rect,&brush);
```

……

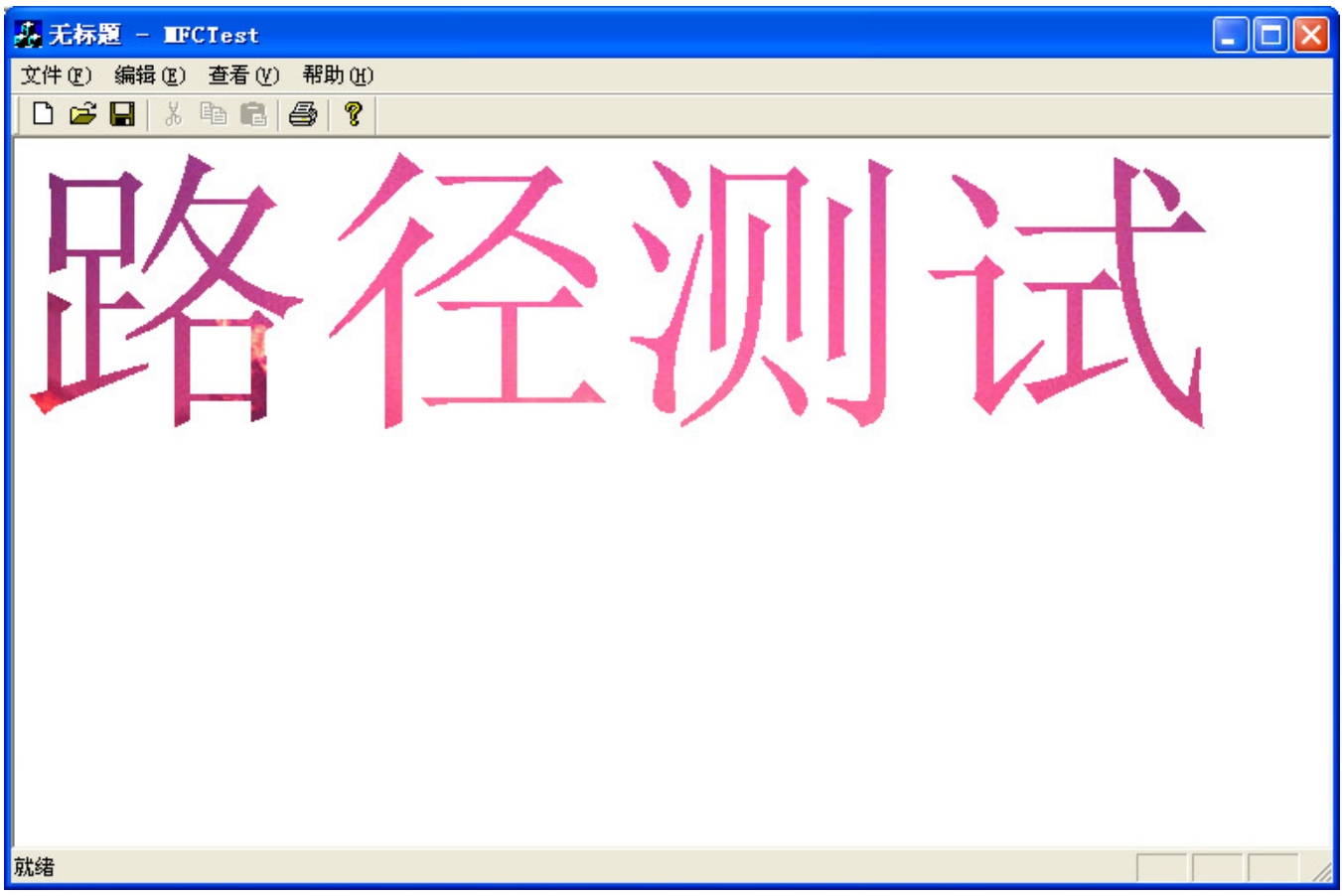
上面的那一段代码大家很熟悉吧，那么一旦有了 SelectClipPath 后，后面那段代码贴上来的图会有什么效果呢？大家请看：




怎么贴外面去了？什么原因，大家还记得我们前面讲的字符吗？是由字符的模式决定的，我们的字符有背景，而我们改成没背景的：

```
pDC->SetBkMode(TRANSPARENT);
```

大家看效果：



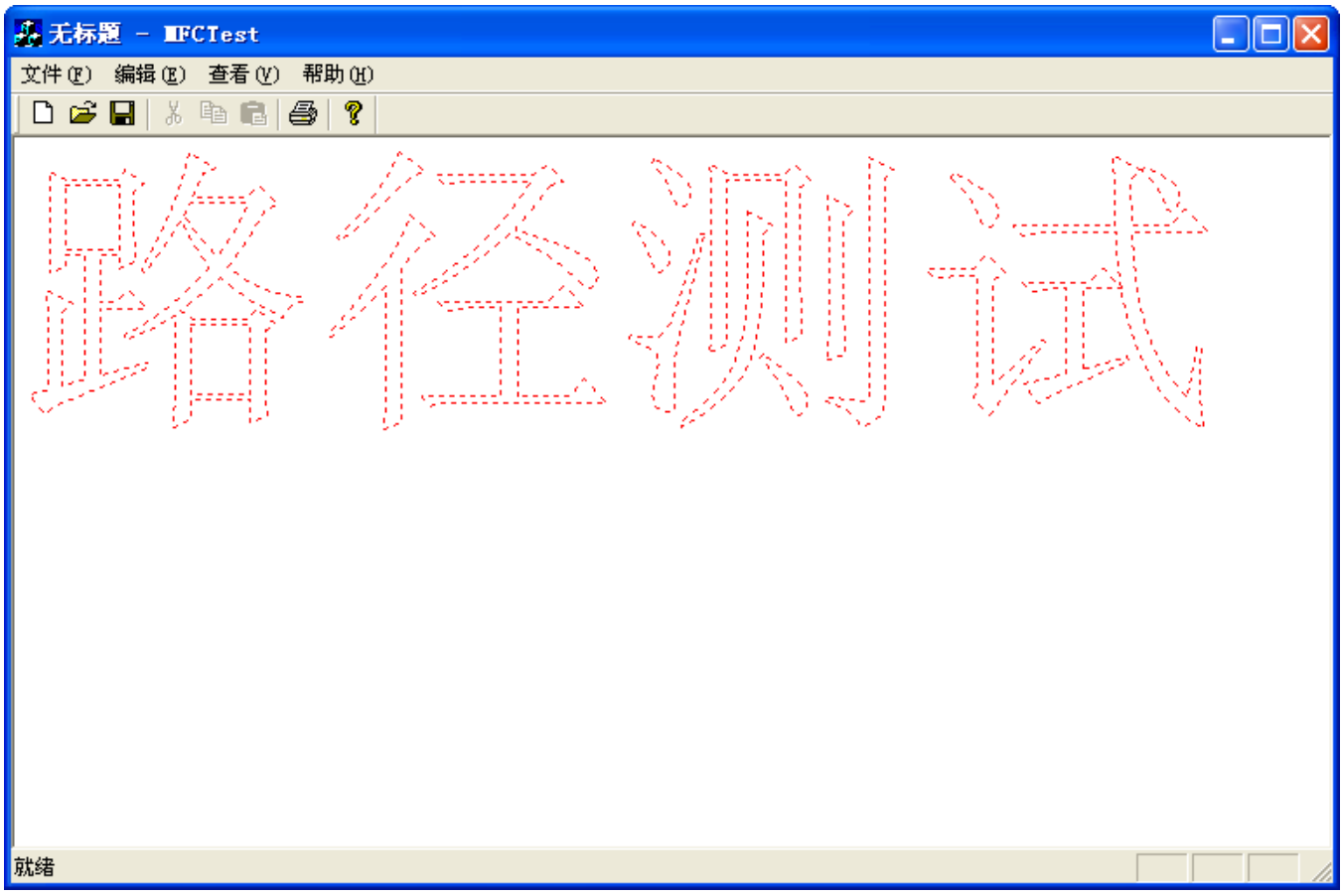

还真就是那么回事，呵呵，那么好，大家删去上面的程序，我们创作一个画笔，去描描边看看：

……

```
CPen pen(PS_DOT,1,RGB(255,0,0));
CPen *pOldPen=pDC->SelectObject(&pen);
pDC->StrokePath(); //这一句最重要，呵呵，效果都在这一句上呢。
pDC->SelectObject(pOldPen);
DeleteObject(pen);
```

……

大家看效果：




当然，大家可以再去试一试刷子，或者将画笔刷子一起试。

当然，你也可以将路径转换成剪裁区域去保存，以便以后使用

总之，路径与剪裁区域有异曲同工之处，都是可以很好地实现我们脑袋中的构想的工具。

对了，从 API 的角度讲了这么多，大家对控件的那些通知消息还会有什么不明白的地方吗？

每一个控件都是会向父窗体发通知的，只不过有些通知里面包含的内容多一些，有些通知中包含的内容少一些，甚至有些通过发过来的消息中包含的是一个指针，这个指针可能指向一个字符串，也可能指向一个结构体，从而去控制控件中一些数据的技术。

其主要的构思还是消息机制，当控件需要与父窗体交流时，当然就需要告诉父窗体，我干什么了，当然告诉老爸的东西如果少了，那很简单，如果需要告诉的多了，两个三十二位的数据包含不了了，那也只有用结构体或者字符数组了，那是没有办法的事。

## 周边编程一〇三：替换技术

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月六日
*/

　　呵呵，牛刀不是小心眼，转载多了，其实是好事，但牛刀将版权也放在上面，如果大家需要转载牛刀文章的话，请不要删除版权信息，这样一是保护了牛刀的版权，二一个，也算是牛刀为本论坛扬扬名，作作贡献吧，相信论坛上的人气会越来越旺的，嘻嘻。

　　好久没写文章了，为什么呢？因为牛刀出文章的速度来快，想必好多的人会赶不上，除非你有很深厚的基本功，呵呵。

　　那么这么久了，我们的《微软积木》结束了吗？没有，这几章是牛刀插进来的，目的嘛，开阔大家的眼界，并且专门地讨论一些牛刀其实早就想告诉大家的技术——替换技术与编译技术。

　　当然，还是那句话，一口不能吃一个胖，其实替换技术在我们日常用的好多软件中就存在，大家随时可以用记事本打开一篇文章，例如大家可以打开《红楼梦》，然后按菜单——编辑/替换(R)...

　　然后在查找内容中输入——贾宝玉
　　在替换中输入——张飞

　　然后按——全部替换，慢慢等吧，呵呵。

　　然后你再耐心读这部小读，你就会发现了新大陆，原来创造一部小说竟然是这么简单，又是一部新版——《关公战秦琼》，呵呵。

　　牛刀找些片段给大家欣赏欣赏——
　　张飞不信，还要听去。婆子们赶上说道：“二爷快回去罢。天已晚了，别处我们还敢走走，只是这里路又隐僻，又听得人说这里林姑娘死后常听见有哭声，所以人都不敢走的。”张飞袭人听说，都吃了一惊。张飞道：“可不是。”说着，便滴下泪来，说：“林妹妹，林妹妹，好好儿的是我害了你了！你别怨我，只是父母作主，并不是我负心。”愈说愈痛，便大哭起来。
　　但愿曹雪芹别来找我的麻烦，呵呵。

　　当然，这种替换技术，在编程中，有专门的定义方法——

```
#define 宝玉 张飞
```

呵呵，如果你真在程序中这样做了，那么很高兴地告诉你，你的编译通不过，什么原因呢？因为我们编程中的替换必须是——标识符，并且是英文方式的，不允许有中文，哎，这没办法，牛刀曾经想通过以下的手段将英语编程做成中文编程，结果失败了——

```
#define 主 void main(){
#define 完 }
#define 印 printf
...
```

当然，如果不是中文，而是英文标识，那就完全可以实现：

```
#include <stdio.h>

#define m void main(){
#define e }
#define p printf

m
p("宏的示例\n");
e
```

大家不要以为不可能，上面的程序完全可以正确运行，为什么呢？因为那下面的定义完全是一种替换，我们分成两部分给大家看：

```
m                   void main(){
p("宏的示例\n");        printf("宏的示例\n");
e                   }
```

明白是什么意思吗？其实通过 define 的定义，左边的，就相当于右边的，甚至于右边的一模一样。

但为什么我们这一讲开始，题目会叫——《周边编程》呢？因为我们的这种替换技术，其实完全不是编程中的内容，什么意思，如果你学过汇编，你就会知道，汇编中有一种指令，叫做——伪指令，也就是说，这些指令，计算机是不会去编译与执行的，而仅仅是一种替换而已，于是，人们为这种替换技术定义一个名称，叫做——宏。

不明白宏，对于我们编程没有丝毫的影响，因为如果你一直看牛刀前面的教程，其实牛刀就是这样走过来的，我们的思路并不受任何拘束，当然，牛刀以前也曾经与大家讨论过一些宏的基本知识，当然，我们这一讲开始与大家详细讲解，当然喽，我们不会使用系统已经定义好的一些宏去给大家讲解，我们努力地让大家能够去理解系统定义的一些宏。

在讲解宏之前，我们先与大家讲一讲一个符号——“\”。

这个符号，在C语言中，被称之为——转义符，为什么会有这个符号的定义与概念呢？没办法，因为好多的字符是显示不出来的，例如回车符，换行符，跳格符.....

适当扯远些吧，那么为什么会有这些不可显示的符号呢？原来啊，这还来源于打印机，或者说是打字机，那是很遥远的年代了，打字机现时，我们每打一个字符，打字机就显然要往后面移动一格，而一行结束后，就要向下移动一行，并且将打字机恢复到一行的开始，当然，这种动作在一开始是手工进行的，因为那时候是打字机初期阶段，并且可能没有电动机吧。

当然，后来技术先进了后，就直接可以用电机啊，等装备直接实现换行，于是，就在键盘上设置了这么一个回车键，当然，为了制表的需要，以后又相继添加了跳格键——Ｔａｂ，一按就可以跳到一个标准格，这样，就可以很容易地用打字机做表格了，很方便。

当电脑出现后，这些键都要占空间，当然喽，什么上下左右键，Home 键,End 键，退格键……

一系列的键就出现了，很特别的是一个字符很特殊，叫响铃，ASCII 码为 7，一运行会触动小扬声器报警，当然，为了让打印机能够顺利地去打印内容，又加进了换页码，等好多好多的码，不要小看这些看不见的字符哟，在老式的 DOS 时代，我们显示内容可就需要通过这些来显示呢？因为那时候根本没有显示器，也只有通过打印机甚至打孔机来取得结果。

当然喽，如果你对这些感兴趣的话，可以用下面的程序来获得一个并不好听的计算机报警声：

```
#include <stdio.h>

void main()
{
        for(;;)
        {
                printf("\7");
        }
}
```

好恐怖，停不下来了，大家做的时候，不要用“死循环”，声音不是很好听的，呵呵。

牛刀很纳闷的一件事就是，其实完全可以使用人们根本使用不到的 @ 字符去作为转义字符啊，当然，一种可能就是，这个字符在那个年代根本不存在，那么这个字符是什么时间出现的呢？应该是出现了“电子邮件”之后，呵呵，虽然牛刀不用电子邮件，但不代码牛刀就不知道有这么个东东啊，呵呵。

不管怎么，我们知道的一点就是，只要出现 \ 那么后面的所有代码的意义就全变了，如：

\n　　回车
\t　　制表
……

当然，如果我们要显示这个字符，怎么办呢？使用两个就行：　　\\

双引号表示字符串的开始，那么我们要显示双引号怎么办呢？也用转义就行：　　\"

好像跑题了，怎么又岔到了转义符上来了呢？

原来啊，我们的宏，只能是一行，然而，我们有时候当定义一个很长的宏的时候，放一行上，看起来会很费力，于是在行未用转义符　\　将最后的行结尾的标志给转义掉，于是，我们就可以将一行的内容放在多行上面了，为了证明这一点，大家一般的　ＶＣ编译器中，注释通常与一般文本颜色不同，而　//　又是单行注释，我们可以注释完一行后，在行未使用一个　\　，　于是，下一行的颜色也变了，大家一试便知。

也正因为如此，我们好多的宏，如果太长，一般末尾会使用　\　来注释掉行尾，从而使这得这宏能跨越多行。我们来看几个系统的宏定义：

```
#define WS_POPUPWINDOW      (WS_POPUP           | \
                             WS_BORDER          | \
                             WS_SYSMENU)
```

怎么样，其实上面的代码应该是：

```
#define WS_POPUPWINDOW      (WS_POPUP | WS_BORDER | WS_SYSMENU)
```

但我们在制作宏的时候，一个定义被分成了三行了，而在行未就使用了转义符　\

甚至在新的标准下，支持双字节转义符，大家可以将下面的代码输进电脑，看看运行出来的效果：

```
cout <<"??!"<<endl;
```

当然，关于转义符，我们就先谈这么多了，否则太跑题了，呵呵。

既然说到了宏，我们就无形中将宏分成了三类——替换宏、参数宏与模板（注：这是牛刀称法，正规教材上估计不会将模板当成一种宏的一般教材上会将模板当成一个函数或类，当然，我们谈到那儿时，我们再说）我们先来说说这个替换宏吧：

其实严格地说来，上面三种其实都可能称之为替换宏，只不过替换的规则不同，所以我们就细分出了后两种宏，什么意思呢？如果大家看上面的一些代码，肯定会明白，所谓的宏，不就是作一些语句的替换吗？

可这些替换是有规则的，而不是什么都可以替换，否则，我们如果做下面的宏：

```
#define if printf
```

那么我们的程序可就麻烦了，例如下面的程序：

```
if(a>b)
        c=a;
else
        c=b;
```

就被无情地替换成了如下代码

```
printf(a>b)
        c=a;
else
        c=b;
```

那还不报错啊？呵呵。

假如我们定义下面的宏：

```
#define ( {
#define ) }
```

那么我们的程序还不乱套了？

于是，我们规定——宏必须是系统可用的标识符。

有一个不成文的规定——宏名要全部字母大写。

当然，你可以不尊重这个规定，当然，我们也不会完全遵照的，例如下面的宏：

```
#define max(x,y) ((x)>(y)?(x):(y))
```

这个宏，如果大写反而感觉到很别扭，因为我们很乐意将其理解成一个函数。

那么上面带括号的宏什么意思呢？括号中可以是宏列表，于是，我们可以将括号中的所有参数都用一定的规则按顺序替换，这就使得宏可以有其很强的发挥基地。

当然这种替换也是会出现问题的，因为这个替换完全是字符替换，而没有任何意义的，而替换以后，需要编译器去编译，所以我们可以用替换的规则去理解一些东东，例如，我们写一个求平方的宏：

```
#define s(x) x*x
```

上面牛刀自己定义的一个宏，目的是为了求某一个数字的平方，那么我们做程序时，如果这样做：

```
c=s(5);
```

那么经过替换后，就变成了：

```
c=5*5;
```

结果肯定没有错，可是如果我们这样做呢？

c=s(2+3)

不要以为有什么不妥，替换出来的结果为：
c=2+3*2+3

结果就不是 25 了，用脚指头都可以算出来，等于 11（用脚指头真能算吗？牛刀怀疑，嘻嘻）

所以我们在做这种宏时，一般都在每一个传进参数外面加上一个限定的括号，定义成如下形式：
#define s(x) (x)*(x)
这样一来，就什么问题都没有了。

请大家注意的，严格的一条就是，这种定义仅仅是一种替换，没有任何其他的意义，并且这种替换是编译系统在编译之前进行的，如果没有相应的替换，这种宏就是一个空架子。

行了，作为一个新的元素——宏的开始，牛刀暂时不与大家详细深入了，下面通过一段程序，来结束我们这一段的讲解，在这段代码中，你甚至不会觉得我们在做程序——

```
#define MAIN void main(){
#define END }
#define PRINT(x) printf(x);
#define CR printf("\n");
#define MAX(T) T max(T a,T b) \
                        {       \
                                if(a>b)       \
                                        return a;       \
                                else       \
                                        return b;       \
                        }

MAX(int)

#include <stdio.h>

MAIN
PRINT("宏竟然可以这样用");
CR

        printf("max(2,3)=%d",max(2,3));
CR
END
```

思考：我们的 max(2,3) 是宏还是函数？
如果是宏？那么这个宏是怎么使用的？
如果是函数，那么这个函数是怎么产生的？

宏 CR 在程序中的作用，果真能过一个 CR 就可以实现换行吗？ CR 是我们编程中的代码，还是其他的，或者仅仅是一种替换而已。

似乎我们的程序没有了入口函数，我们的入口函数哪儿去了？我们的语言还是函数语言吗？

你能否做出一些其他的宏，实现更超级功能？去方便我们的编程？

作业：

C 语言中虽然可以使用一些特殊的进制形式，例如八进制十六进制等等，但却不支持二进制，请大家做宏的方法使得我们的编程能够支持二进制，以便方便我们的编程。

**周边编程一〇四：宏的表白**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月七日
*/
```

我叫宏
生日：不详
爱号：喜欢多管闲事，属于狗拿耗子类型的
功能：什么功能都没有，就是喜欢传话，并且传的过程中，一定要经过改变，不变不行，呵呵
职业：专门干那种捉弄人的事，喜欢改变别人，当然，是在编译器同意的情况下，当然，这种改变是不可以逆转的，所以一般的编程人员不太喜欢我，但如果用得恰当，并且好的话，我的出现可以使得好多的技术变得一劳永逸，并且好多库都是使用我的家庭成员去实现的。
家庭生员：
    替换宏
    参数宏
    模板（分为函数模板与类模板）

成员介绍，详见下面的牛刀文章：
    可能绝大多数的书上不会将模板当成宏，但牛刀这样说了，既然说了，当然有其道理，大家还记得上一讲最后我们给大家做的一段程序吗？

```
#define MAIN void main(){
#define END }
#define PRINT(x) printf(x);
#define CR printf("\n");
#define MAX(T) T max(T a,T b) \
                        {       \
                                if(a>b)       \
                                        return a;       \
                                else       \
                                        return b;       \
                        }

MAX(int)

#include <stdio.h>

MAIN
PRINT("宏竟然可以这样用");
CR
```

```
        printf("max(2,3)=%d",max(2,3));
CR
END
```

当然，对于替换，我们上面的程序与下面的程序则是等价的：

```
int max(int a,int b){if(a>b)return a;else return b;}
```

```
#include <stdio.h>

void main(){
        printf("宏竟然可以这样用");
        printf("\n");
        printf("max(2,3)=%d",max(2,3));
        printf("\n");
        }
```

对于上面那个求最大值的函数，可能大家也觉得很好玩吧？其实那就是一个函数，只不过，不同的是，我们将这个函数的所有语句都放在一行上了，虽然可以这样做，但这种函数看起来的确有些别扭，并且不符合人们的习惯，于是，C＋＋的设计者就产生了一个符合人们习惯的想法，将这种特殊的宏，用一种特殊的形式成定义形式去定义，从而引出了一个新的概念——模板：（当然可以看一下模板函数的定义：）

```
template <class T>
返回值类型  模板函数名(参数列表)
{
        //函数的实现
};
```

什么意思？我们还来实现上面的那一个求最值的模板吧，看程序：

```
#include <stdio.h>

template <class T>
T max(T x,T y)
{
        if(x>y)return x;
        return y;
};
void main()
{
        printf("max(2,3)=%d\n",max(2,3));
}
```

请大家先看我们模板的定义部分：

```
template <class T>
T max(T x,T y)
{
        if(x>y)return x;
```

```
        return y;
};
```

如果我们将下面函数的 T 去直接用 int 给替换了，那么是什么呢？看程序：

```
int max(int x,int y)
{
        if(x>y)return x;
        return y;
}
```

不就是一个函数吗？所以说，模板其实就是一个宏，只不过不同的是，模板可以放在不同行上，而宏必须放在同一行上，仅此而已，所以牛刀接下来对模板的讨论就放在宏里面了，大家不要误解，当然，如果你要考试的话，那还是以教科书为准，呵呵，否则的话，牛刀的这种理解一定适合于你。

当然，有两点需要说明的：

·一就是我们的模板中的 class 不是类，不要混为一谈，那只是个定义，表示后面的数据是一种类型，甚至这种类型由编译器自动地根据需要去用这个模板自动地生成相应地函数，而不需要我们显式地定义了，于是，大家看上面的程序中，我们并没有去定义一个 max 函数，但随时就可以使用了，甚至可以这样用，很直接，很舒服：

```
#include <stdio.h>

template <class T>
T max(T x,T y)
{
        if(x>y)return x;
        return y;
};
void main()
{
        printf("max(2,3)=%d\n",max(2,3));
        printf("max(5.5,6.7)=%f\n",max(5.5,6.7));
}
```

让人瞠目了吧，呵呵，我们只需要定义一个模板，就什么事情都不要做，就可以直接地使用，这也是模板给我们的很特别的地方。

·第二点就是：我们的函数竟然不需要去定义，也就是说，这个模板既有了宏的替换功能，同时又有了函数的功能，只不过不同的是，这个函数是通吃一切类型，呵呵。（当然需要能够比较大小的两个类型喽）

好了，很容易地想到，什么叫类模板吧？呵呵：

```
template <class T>
class 类模板名
{
```

```
        //类成员与函数
};
```

可能你会认为小题大作了是吧？好，那么我们这一讲先将宏与模板的知识先放一放，我们下面与大家来定义一个类：

目标：使用类去实现整数的集合

集合，是大家在高中就接触到了的一个内容，但在编程语言中，基础语言是不支持集合类型的，当然喽，后来我们做面向对象的程序以后，多数语言都根据一些自己的规则去编写一些集合类，或者集合函数，以便去处理集合内容喽。

可我们为什么说语言不支持集合呢？因为我们的集合并不像数组那样，是一个数据类型，当然，一旦我们使用了类以后，我们就可以将集合做成一个数据类型喽。

当然，我们做集合可以有两种方案，一种方案是用指针去实现，可以实现任意大小的集合，另一种方案是数组，我们集合的大小有限制，假设我们的集合中允许最多有十个数据吧（当然可以有好多，但如果太多的话，我们的好多效果就很难在一小段程序中体现出来了。

那么我们首先要知道我们集合是什么，干什么用的，怎么用，那不在于我们的定义吗？

牛刀集合定义：

集合相当于一个动态的数组，但又不是数组，其大小可以随着加入的数据多少而变化（当然，我们实验的集合是有限制的喽，呵呵，十个）

集合可以向其中加入一个整数或者删除一个整数，如果我们增加一个整数的话，集合的数据量就多一个，否则数据量少一个，但如果集合满了，就不能向其中增加数据了。

集合中加入两个相同的数，表示不同的集合元素（当然这是与高中数学不同的地方，当然，你也可以做一个表示相同的集合，一样的，那只在于我们怎么去设计。

行了，有什么需要，我们到时间再说吧，我们首先来定义这么一个集合类：

```
#include <stdio.h>
class IntArray
{
        int a[10];
        int n;
public:
        IntArray()
        {
                n=0;
        }
        void Add(int x)         //向集合的末尾加入一个数据
        {
```

```
                a[n]=x;
                n++;
        }
        void List()         //为了看集合中的内容，我们加入了这个函数，
来显示集合中的内容
        {
                for(int i=0;i<n;i++)
                {
                                printf("集合中的第 %d 个数据
为:%d\n",i+1,a[i]);
                }
        }
};

void main()
{
        IntArray A;
        A.Add(2);
        A.Add(5);
        A.Add(7);
        A.Add(3);
        A.List();
}
```

上面的集合定义中，我们加入了四个数据，为了显示，我们为集合增加了显示的能力（List），当然喽，这里会有些问题，我们一般会这样改，看程序：

```
        void Add(int x)         //向集合的末尾加入一个数据
        {
                if(n>=9)
                {
                        printf("对不起，集合满了\n");
                        return ;
                }
                a[n]=x;
                n++;
        }
```

这样一来，我们的集合就不会出问题了，当然，我们还可以按位置去添加一个数据，那当然只要重载 Add 函数就行了，呵呵：

```
        void Add(int x,int Index)
        {
                if(n>=9)
                {
                        printf("对不起,集合满了,不能继续加载数据\n");
                        return;
                }
```

```
                if((Index>n+1) || (Index<0))
                {
                        printf("对不起，你插入的位置不正确!!! \n");
                        return;
                }
                for(int i=n+1;i>Index;i--)
                {
                        a[i]=a[i-1];
                }
                a[Index]=x;
                n++;
        }
```

……

```
void main()
{
        IntArray A;
        A.Add(2);
        A.Add(5);
        A.Add(7);
        A.Add(3);
        A.Add(10,2);
        A.List();
}
```

显示的结果为：

```
集合中的第 1 个数据为:2
集合中的第 2 个数据为:5
集合中的第 3 个数据为:10
集合中的第 4 个数据为:7
集合中的第 5 个数据为:3
```

好，我们下面来删除数据，其实删除数据会显得异常地简单，看程序：

```
        void RemoveItem()
        {
                if(n>0)n--;
        }
        void RemoveItem(int Index)
        {
                if(n>0)
                {
                        if((Index<0) || (Index>n))
                        {
                                printf("删除非法!!! \n");
                                return;
```

```
                        }
                        for(int i=Index;i<n;i++)
                        {
                                a[i]=a[i+1];
                        }
                        n--;
                }
        }
……
void main()
{
        IntArray A;
        A.Add(2);
        A.Add(5);
        A.Add(7);
        A.Add(3);
        A.RemoveItem(2);
        A.List();
}
```

行了，这就基本能够实现一些简单的集合类型了，呵呵，当然你可以增加一些特殊的功能，例如集合的排序等，当然，用上面的这个很简单的集合，我们就可以很容易地实现——队列、链表、栈等一系列特殊的结构，呵呵，其实那些结构也不过如此嘛？

例如，栈就是在头上插，头上删，或者尾插尾删就一切 OK 喽，而队列的删除与插入位置正好是一头一尾，仅此而已。上面集合的功能我们就不再扩充了，并且删去 List 函数：

```
include <stdio.h>
class IntArray
{
        int a[10];
        int n;
public:
        IntArray()
        {
                n=0;
        }
        void Add(int x)         //向集合的末尾加入一个数据
        {
                if(n>=9)
                {
                        printf("对不起，集合满了\n");
                        return ;
                }
```

```
        a[n]=x;
        n++;
    }
    void Add(int x,int Index)
    {
        if(n>=9)
        {
            printf("对不起,集合满了,不能继续加载数据\n");
            return;
        }
        if((Index>n+1) || (Index<0))
        {
            printf("对不起, 你插入的位置不正确!!! \n");
            return;
        }
        for(int i=n+1;i>Index;i--)
        {
            a[i]=a[i-1];
        }
        a[Index]=x;
        n++;
    }
    void RemoveItem()
    {
        if(n>0)n--;
    }
    void RemoveItem(int Index)
    {
        if(n>0)
        {
            if((Index<0) || (Index>n))
            {
                printf("删除非法!!! \n");
                return;
            }
            for(int i=Index;i<n;i++)
            {
                a[i]=a[i+1];
            }
            n--;
        }
    }
};
```

```
void main()
{
        IntArray A;
}
```

行了，好容易做成的一个集合的类，大家不想将其做成一个模板吗？当然，接下来，我们就将其中的一些东东给改了，呵呵

```
#include <stdio.h>
template <class T>
class ObArray
{
        T a[10];
        int n;
public:
        ObArray()
        {
                n=0;
        }
        void Add(T x)
        {
                if(n>=9)
                {
                        printf("对不起，集合满了\n");
                        return ;
                }
                a[n]=x;
                n++;
        }
        void Add(T x,int Index)
        {
                if(n>=9)
                {
                        printf("对不起,集合满了,不能继续加载数据\n");
                        return;
                }
                if((Index>n+1) || (Index<0))
                {
                        printf("对不起，你插入的位置不正确!!! \n");
                        return;
                }
                for(int i=n+1;i>Index;i--)
                {
                        a[i]=a[i-1];
                }
                a[Index]=x;
```

```
                n++;
        }
        void RemoveItem()
        {
                if(n>0)n--;
        }
        void RemoveItem(int Index)
        {
                if(n>0)
                {
                        if((Index<0) || (Index>n))
                        {
                                printf("删除非法!!! \n");
                                return;
                        }
                        for(int i=Index;i<n;i++)
                        {
                                a[i]=a[i+1];
                        }
                        n--;
                }
        }
};

void main()
{
        ObArray<double> A;
        A.Add(2.3);
        A.Add(6.5);
        A.Add(4.6);
}
```

不同的是，我们类中的一些类型需要通过三角符号告诉系统，因为我们在定义这个数据时，是需要初始化的，例如系统怎么为其分配空间啊，什么的，都需要系统去做。

大家是不是想来个这样的定义？

```
struct Point
{
        int x;
        int y;
}
```

……

```
void main()
{
        ObArray<Point> A;
}
```

当然喽，这样我们的系统就要作一些适当的改动喽，呵呵，大家可以自己去八仙过海——各显神通，呵呵，这样一改，我们这个集合类就可以通吃一切喽，甚至可以将一些类，一些对象都加进我们的集合，而这一点，如果没有模板技术，是肯定不可能被办到的。

原因很简单，因为我们系统可以为大家定义一个类，但并不知道大家是使用的什么数据类型啊，甚至有些数据类型是大家自己去定义的，于是，我们这些想法，就非模板莫属喽。

当然，以后我们讲到有关模板的应用的话还会与大家讲到，这一讲到此为止，关于模板的事情，我们就先放一放啦，不是说不想与大家讲，而是不能讲，一讲就要写书，本身在Ｃ＋＋中，模板库就是一个相当巨大的库，说实话，想全面地掌握都是很困难的，不要说使用了，我们只能使用到哪儿需要使用了，我们才会与大家去交待一下就行，我们为什么这几讲要讲一讲周边编程也是这个原因，如果不讲，我们的很多的技术其实是深入不下去的。

那么这一讲就到此结束了，下一讲见。

**周边编程一〇五：歪打正着**

/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月八日
*/

上一讲与大伙讲到了模板，感觉似乎有点跑题，不过既然牛刀将其定义到宏的范围之内，肯定不会平白无故，如果大家仔细体会一下，那么我们的模板仅仅是一个特殊的宏，只不定义的手段不是用 #define 而是使用的 template 而已 template <class T>

但是不同的是什么呢？不同的是，这里的 T 不是我们传进参数或者直接用宏的办法去替换，而是由编译器自动分析下面程序的数据类型，从而用相应的数据类型去替换，仅此而已，我们下面看两种不同的定义：
定义一：
```
template <class T>
T max(T a,T b);
```

定义二：
```
#define T int
T max(T a,T b);
```

大家知道，如果我们只是使用 int 类型的话，那么两种定义是一模一样，可惜的是，下一种宏定义就特别地死板，只能产生一个整型的求最大值函数，而定义一就不同了，定义一中的 T 其实我们没有去定义，而是让编译器在编译的过程中去发现我们需要定义什么样的数据类型，再根据相应的数据类型去生成相应的最值函数，仅此而已。

那么大家动动脚趾头想想，定义一的替换是在编译之前还在编译之后呢？

显然编译器在没有编译的时候是不可能知道我们一个数据的数据类型的，那么第一种定义就肯定在编译之后喽，那么就怪了，编译器既然编译过了，那么这种替换还有意思吗？不是说编译过后就全变成二进制代码了吗？

非也非也，原来啊，我们编译器在编译程序时，不是一次成功的，而是要进行若干次，每编译一次称着一“遍”，一般的编译器的编译，少则两“遍”多则需要好几遍。

而我们这种识别类型的工作，部分空间分配的工作，就在第一遍完成，而需要编译器扫描后再替换的一些工作，就需要放在第一遍扫描过后去进行喽。

于是，我们看下面的程序：

```
#include <stdio.h>

template <class T>
T max(T a,T b)
{
        if(a>b)return a;
        return b;
}

void main()
{
        printf("max(2,3)=%d\n",max(2,3));
}
```

程序在第一遍编译时，遇到了 T ，发现了这个 T，编译器也不知道什么类型，但后来发现使用的 max 函数，使用到了这个 T ，那么好，编译器就会继续编译，去找这个函数，于是在 printf 函数中找到了 max(2,3) ，于是，根据其中的数据分析出其中数据为整型，于是，编译器就确定了 T 的类型为 int ，于是在第一遍结束之后，上面的 template<class T> 与 #define T int 就没有区别了。

第二遍开始之前，程序首先启动替换系统，将 max 函数中的 T 全部用 int 去替换掉，然后再重新编译，于是，程序顺利编译通过，过程就是这样的。

什么意思想必大家也明白了吧？计算机是不会回头的，所以一遇到这种不确定的东东，则必须进过多遍的编译，才能编译完成，仅此而已。

从这一点来说，大家应该明白，为什么模板的定义要用 template ，而不用 #define 了吧？因为 #define 是告诉编译器，我是一个普通的宏，编译器你只需要作相应替换就行了，而 template 则告诉编译器，编译器你行行好，你先编译一遍，确定我定义的数据类型，然后来来替换我，好吗？

于是，同样的宏，就有了两种不同的替换方案，歪打正着，都实现了同一个目的——替换。

对于类的替换也是一样，同样是通过多遍扫描才能编译通过的，于是，与正常程序不同的是，这种编译可能需要的时间要长一些，但对于编程者来说，无疑是一个福音。

我们编程的目的就是尽量让能够交给计算机去计算的工作去交给计算机去完成，最大化地做到我们人为的计算机减少到最少，从这一点上来说，如果不用宏，我们亏啊，呵呵，于是，我们下面就要专门地讨论讨论宏在我们编程中的作用——

作用一：提高通用性

首先来说，我们做程序不可能一成不变的，宏的最直接的一个例子就是：

```
#define PI 3.14
……
```

好处一：偷懒，这无疑是最大的好处，一个 PI 要少打好多的数字，呵呵。

好处二：整体的精度改变，我们一旦程序做好了，想要改变精度，我们只需要做一件事，那就是作如下改变：

```
#define PI 3.141592654
```

好处三：使得程序变得通用，例如我们需要在其他部分需要一个求圆面积的代码，正好这儿有一份，那么复制过去就行了，那么 PI 的精度一改，整个代码就可以满足我们的要求喽。

通用性好的最大的好处就是可能只知道程序的功能，就可以自由复制，并且只需要通过几个宏，就可以改变整个程序以满足我们的需要，从而使得我们编程变成了一个不是经常地敲字母的事喽，而变成了一个粘粘贴贴的东东喽，呵呵，需要改动的很少。

可以想象，只要宏定义得足够丰富，那么我们做程序可能就来得格外地方便。

作用二：跨平台性

例如有朋友跟牛刀讨论过，ＶＣ下， void main(); 可以使用，而其他编译环境下，通不过，那么我们就可以做如下的定义：

```
#define Main void main(){
#define EndMain }
```

如果在 GCC 的编译环境下，我们可以作如下的定义：

```
#define Main int main(){
#define EndMain return 0;}
```

那么，我们下面的程序：

```
Main
        //程序代码
EndMain
```

这样的程序就很通用了，当然，你可能会觉得，那多费事？还要到相应的平台下面，改变相应的宏，真麻烦。

标准考虑到了我们的尴尬，于是，为我们的宏，做了专业的条件编译指令，这一讲先告诉两个，大家先用着，呵呵

```
#include <stdio.h>
```

```
#define VC 1

#ifdef VC
        #define Main void main(){
        #define EndMain }
#endif

#ifdef GCC
        #define Main int main(){
        #define EndMain return 0;}
#endif

Main

EndMain
```

大家看，#ifdef 就是：如果定义了后面的宏的话，就执行下面的宏定义，直到 #endif 结束

当然，如果你改变了平台，你只需要 #define GCC 2 就一切可以了。

对了，就到这儿提一句啊，我们一直在强调，宏不是我们的C语言编程部分，也就是说，不是语句，所以行未是不需要分号“;”作为结束标志的，这一点尤为重要，如果有分号的话，也会一并被替换掉，切记，切记。

对了，如果你改变了别人的 VC 变成了你的 GCC ，那么请你也要将别人的 VC 的定义给删掉哟，否则出问题的，呵呵，就是宏定义重复了。

其实啊，在现实的编程中，我们完全没有必要担心这种情况，因为平台的定义都是在专业文件中，并且随着专业的平台发布的，所以我们只要像这样定义：

```
#include <stdio.h>

#ifdef VC
        #define Main void main(){
        #define EndMain }
#endif

#ifdef GCC
        #define Main int main(){
        #define EndMain return 0;}
#endif

Main

EndMain
```

那么这个代码就拿到哪个平台下就都通用喽。

当然，我们这儿是为了与大家讲明白，可以通过这种宏的定义去使得我们的代码有平台通用性，其实这种通用性不需要我们自己去考虑，每个平台下都会有大段的代码，去专门处理这种事情，从而会使得我们的同样的 C 或 C++ 程序，拿到哪个平台上都能够顺利地编译通过。

当然，你如果有机会去看看相应的代码，你可能就会觉得那些代码怪怪的了，很可能是一头雾水，不过也没有必要，我们对待这些特殊的代码，我们唯一的方案就是——不要去看她。

说真的，牛刀原来为名人机做一些程序的时候，过后再看看这些程序，都会怀疑这些程序是不是我自己写的，也就是说，编程序是一个很享受的过程，而看（读别人的）程序则必然是一个很枯燥的过程。

所以牛刀一再建议大家，写程序时，一定要加注释，为了人类的编程事业，建议大家要舍得花一点时间。

作用三：方便性

经典的宏，往往会被传抄，拷贝，C语言不支持直接的二进制编程，于是，下面的代码就成了经典之举：

```
#define b00000000 0x00
#define b00000001 0x01
……
#define b11111110 0xfe
#define b11111111 0xff
```

如果你用到位运算，像这样的代码，有一份，那么编程真是太方便了，有时候牛刀在处理二进制的时候，就会想啊想（二进制到十六进制的表记不住啊，牛刀喜欢掰手指头，呵呵，别笑啊）。

就像上面这样的宏，可以说什么任何的技术含量，可惜这种宏是最珍贵的。

还有一种宏，是人们用计算机凭空生成一个图片用的，当然，牛刀以前也曾经在《点阵图片》那一章与大家做过一幅，可那真的是牛刀一个一个的十六进制算出来的，可牛刀在网上看过一个就相当经典了，人家这样定义：

```
#define X )*2+1
#define _ )*2
#define s ((((((((((((((((0

static unsigned short stopwatch[] =
{
s _ _ _ _ _ X X X X X _ _ _ X X _ ,
s _ _ _ X X X X X X X X X _ X X X ,
```

```
s _ _ X X X _ _ _ _ _ X X X _ X X ,
s _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
s _ X X _ _ _ _ _ _ _ _ _ X X _ _ ,
s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ _ _ _ _ _ _ _ _ _ X X _ ,
s X X _ X X X X X _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ X X _ ,
s X X _ _ _ _ _ X _ _ _ _ X X _ ,
s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
s _ X X _ _ _ _ X _ _ _ _ X X _ _ ,
s _ _ X X X _ _ _ _ _ X X X _ _ _ ,
s _ _ _ X X X X X X X X X _ _ _ _ ,
s _ _ _ _ _ X X X X X _ _ _ _ _ _ ,
s _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
};
```

上面的一段代码是牛刀复制过来的，至少牛刀觉得这段代码相当精彩。

作用四：空间效率高

宏与我们正常编程之间最大的不同就是，宏仅仅是一个替换，并且永远留在程序文件中，在编译完成的 exe 文件中，没有宏的概念，于是，当一段宏没有被替换的时候，那仅仅是一段宏，一段没有用的废代码而已，而一旦被替换，这种宏就可以变成数据，变成代码，变成函数，变成程序，甚至变成一些恐怖的恶魔。

曾经在于大家讲过的 MFC 中的消息响应机制的时候，就曾经与大家说过，我们的 MFC 类中，响应消息，没有使用消息响应机制，向父类中传递消息，也就是说，那不是虚函数，那是什么？

那就是宏：

```
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
        ON_WM_CREATE()
        ON_WM_NCLBUTTONDOWN()
        ON_WM_NCMOUSEMOVE()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

大家看，其中的响应到的部分宏，就会变成程序，而没有响应到的部分，那段代码，那段宏就与我们没有关系，这样，微软就做到了，响应多少的消息，就做多大的程序，而不会使得程序因为我们的后台库的原因而变得很巨大。

在这一点上，ＶＣ做得就比ＶＢ要优秀的多，ＶＢ不管你做多么小的程序，都要将一整套的ＶＢ库带着，打包发行，才能够真正的使用ＶＢ做开发出来的一

套软件。

提到这儿，牛刀再多说两句，我们的 API 编程是直接调用 API 函数产生的，没有库的概念，所以我们 API 编出的程序可以拿到任何一台装有 Windows 的电脑上直接运行，而 MFC 程序就不同了， MFC 有 DEBUG 模式 和 Release 模式，也就是调试模式与发行模式，为什么会有这两种模式呢？

这就在于，调试模式是没有带有 MFC 库的，所以我们的调试模式产生的 exe 文件如果想要发行的话，就要将 MFC 的相应的动态链接库带在一起去发行才行，然而，没有必须，一般我们在发行之前，使用 Release 模式，重新编译一下就行了，因为发行模式是将所有用到的 MFC 库都打包到了同一个程序中了，所以在任何机上都可以运行。

所以理论上说，Release 模式文件应该比 Debug 模式文件要大，然而事实不是，牛刀刚刚看了一下，发行模式的文件要比调试模式文件要小得多，这就矛盾了，为什么库越多，程序反而越小呢？

原来啊，调试模式中是有好多好多的废信息的，而发行模式将这些东东全都给删了，就是这个原因，简单吧？呵呵，也就是说，发行模式是将调试模式给简化喽。

那么是怎么简化的呢？我们调试模式大家还知道吗？我们可以将程序用调试方式一句一句地运行，这是怎么实现的呢？我们的断点是怎么设置的呢？这些信息，exe 文件中必须都有，让编译器知道运行到哪儿喽，因为 VC 和ＶＢ不同，ＶＢ是解释执行的程序，而ＶＣ是编译执行的程序，所以说，我们的ＶＢ是可以输入一行就执行一行的，从而，有用过 ＶＢ经历的人应该都知道，如果我们某一行语法如果有毛病，ＶＢ编译器立马就给报错了，而这一点，ＶＣ就做不到，所以说，ＶＣ的调试模式下，必须加入好多好多的冗余信息，去将我们在程序运行中的一些数据给导出来，甚至调试模式下会给一些特殊的宏，或者是函数，例如断言，反正牛刀也没用过那些，就不与大家说了，说也是空谈，你说是不，呵呵。

而这一系列的冗余信息，在编译成发行模式时，全部删去。

对了，大家知道，我们在调试模式下定义一个字符串，如果不赋值，就是会一串“烫”是吗？而发行模式就不会给我们的内存进行统一地设置了，于是，在那些模式中，我们的所有内存一下子就都暴露在了我们面前。做个程序试试：

```
#include <stdio.h>

void main()
{
        char ch[20];
        printf("%s\n",ch);
```

}

调试模式下，显然是：

烫烫烫烫烫烫烫烫烫?↕

那么我们按　编译/批构件.../全部重建

然后我们到相应的目录下，就会有一个　Release　目录，这个目录下，就是发行版的程序，我们进去当然需要用　DOS　形式进去，我们去试试：

```
D:\PVc\Test\Release>TEST



D:\PVc\Test\Release>
```

怎么样？什么都没有，牛刀不服，去　Debug　目录再执行一下看一看：

```
D:\PVc\Test\Release>cd ..\Debug

D:\PVc\Test\Debug>test
烫烫烫烫烫烫烫烫烫?↕

D:\PVc\Test\Debug>
```

看出区别来了吧？

严重跑题。

**周边编程一〇六：文件系统**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月九日
*/
```

大家如果看其他的一些软件，就会发现，那些软件中，会有好多好多的文件，有头文件，有 CPP 文件，有 C 文件，甚至有 DLL 文件，当然还有其他一些好多的文件，对于这些文件来说，我们可能会产生一些纳闷，这当程序编译时，这些文件是怎么组织的呢？下面，我们就具体来讨论讨论这个问题。

显然，我们所有的程序都可以放在一个文件中，而没有任何问题，然而，我们的程序基本做不到这一点，当然，如果你想做到这一点，也不是太难，下面牛刀就来做一个：

```
void main()
{
}
```

行了，上面的文件，就是一个单文件程序，当然，在其中，我们可以定义任何函数与使用任何运算符，只要你不使用其他的函数，我们就可以在单文件系统中做，看程序：

```
int max(int x,int y)
{
        if(x>y)return x;
        return y;
}

void main()
{
}
```

当然，上面的函数，我们没用，当然可以用起来，看程序：

```
……
void main()
{
int a=3;
int b=4;
int c;
c=max(a,b);
}
```

是喽，我们只要不用到其他的函数，我们就可以使用单文档，当然，如果你要用 printf 或者其他函数将结果显示出来，这种单文档就解决不了问题了，不过我们为了让大家明白是怎么回事，我们大家就来理解编程吧，而不要去看结果喽，呵呵，有时候，我们做程序是需要理解能力的。

为了说明的方便，我们下面用注释的方法来表示我们的文件，当然，创建一个文件，很简单喽，文件/新建... 然后选择相应的文件类型，再命一个名子，就确定就成，呵呵，我们现在其实很想将上面的程序分成两个文件，但为什么可以那么分呢？我们先再试验一下单文件系统下，我们的程序会怎么样，看程序：

```
int max(int,int);

void main()
{
        int a=2;
        int b=3;
        int c;
        c=max(a,b);
}
```

请大家看，上面的程序，我们并没有具体定义 max 函数，而仅仅定义了一个函数原型，奇怪的是，这种程序在编译时，竟然不报错，而是在链接时，才出现了两个错误：

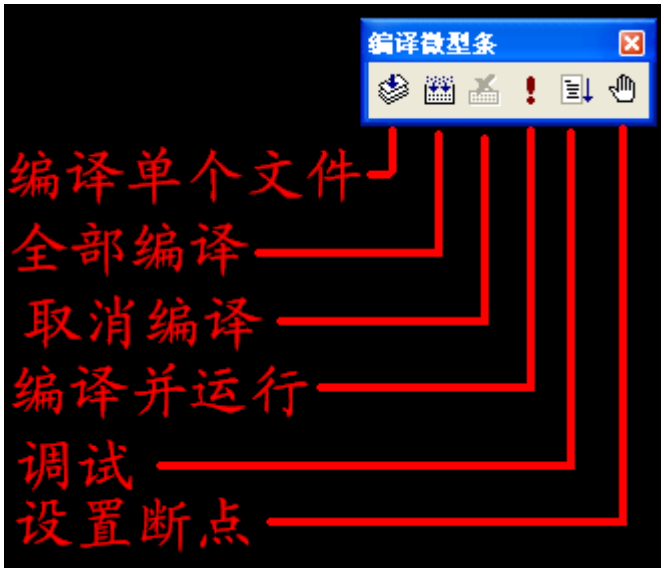
```
--------------------Configuration: Test - Win32 Debug--------------------
Compiling...
Test.cpp
Linking...
Test.obj : error LNK2001: unresolved external symbol "int __cdecl max(int,int)" (?max@@YAHHH@Z)
Debug/Test.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.

Test.exe - 2 error(s), 0 warning(s)
```

不详细解释了，一个是找不到函数，一个是程序不能运行，仅此而已。

那么为什么可以编译呢？这其实就是——函数原型 的作用。

如果大家看过牛刀以前的教程，牛刀曾经说过，函数原型相当于一面小镜子，那么既然现在深入到这儿了，我们的思路要改一改，那么什么是函数原型呢？

函数原型是告诉我们编译器，这个函数就像这样子了，你尽管编译吧，千万不要报错哟，呵呵，什么原因，原来啊，我们的程序是单个编译的，请大家看下面的图片：




我们可以编译单个文件，可以全部编译或者停止编译，当然，可以编译完后立刻运行程序，或者设置断点去调试程序。

那么什么概念呢？我们一般使用全部编译，而全部编译必须有 main 入口函数，而单个编译不需要有。

大家建立一个新的工程，取名 max ，建立程序 max.cpp ，然后在其中加入如下代码：

```
int max(int a,int b)
{
        if(a>b)return a;
        return b;
}
```

请注意，编译时，一定要“单个编译”的按钮，这样就编译就不会报错了，当然，不但不能全部编译，而且是不能运行的，因为没有入口函数 main ，好了，大家这下到相应的 debug 目录下看看，会产生一个 max.obj 文件，这就是编译好的文件了，我们将其复制到我们自己原来的需要调用 max 函数的相应目录下，大家再编译，好像还有问题，是也不是，什么原因呢？因为我们的程序并不知道 max.obj 中的相应的函数啊，因为我没有告诉我们的编译器，在 max.obj 中有相应的函数在那儿，那怎么办呢？

我们在 max 那个工程中加入一个 max.h 然后在其中加入

```
int max(int,int);
```

当然，我们需要在 max.cpp 中加入一个 #include "max.h"

将 max.h 包含进我们的 max.cpp 文件，然后单个编译，到程序下一看，多了一个 max.pdb 文件，我们再次复制过来，再次编译：

还是没有用啊，什么原因，因为我们的编译器在找相应文件时，必须用 #include 将相应的文件包含进来，才能找得到，于是，我们在我们原有的文件中，加入一个 max.h 头文件，并且将头包含移进主程序，看代码（注：我们以注释的方法告诉大家是什么文件：

```
//max.h
int max(int,int);
//test.cpp
#include "max.h"

void main()
{
        int a=2;
        int b=3;
        int c;
        c=max(a,b);
}
```

结果很是失败，还是找不到 max 函数，什么原因？难道是我们的理论错了吗？

似乎程序的编译过程应该这样的——
注：此过程是牛刀想当然，没有科学依据，但可以帮助大家理解程序的运行过程：

- 当我们按——全部编译时，我们的编译器首先进行宏替换，这一点不容置疑。（并且每找到一个新文件，都会进行这项工作的。
- 我们遇到 #include 包含时，如果是尖括号（注：就是大于号小于号），包含进来的头文件，就直接到系统目录中去找相应的头文件（注：这里的系统不是指 Windows 系统，而是指 ＶＣ编译系统与规定的几个相关目录），如果是分号，则先到当前目录已经子目录下找文件，如果没有再到系统目录下去找相应的文件。
- 当然，如果找不到头文件，则报错，但如果找到了头文件，就根据头文件名，找相应的 cpp 文件或者是 dll 文件，如果找到 cpp 文件则对其进行宏替换后编译，如果找到 dll 文件，则对其进行加载（当然，也不是随意加载的，我们下面会给大家做一个程序给大家看一看）。
- 编译完成后，我们的程序就可以直接运行，否则运行报错，这就是我们上面的情况。

当然，我们再去新建一个工程，大家按 文件/新建.../Win32 Static Library
然后在工程下写入 max ，选一个位置，确定 完成

再建一个 max.cpp

```
int max(int a,int b)
{
        if(a>b)return a;
        return b;
}
```

这次我们可以按——全部编译，因为我们的静态链接库不需要有 main 函数，呵呵。

好了，我们编译后，可以去目录下看一看，有一个 max.lib 文件，那就是我们的静态链接库文件了，请大家注释，这个文件是真正编译过的，二进制文件，不需要再次编译就可以用了。

好了，我们将 lib 文件复制到我们的程序目录下：（注意，不是 Debug 目录，而是 Debug 目录外面一层目录。）

然后我们输入下面的程序：

```
#pragma comment(lib,"max.lib")
int max(int,int);
void main()
{
        int a=2;
        int b=3;
        int c;
        c=max(a,b);
}
```

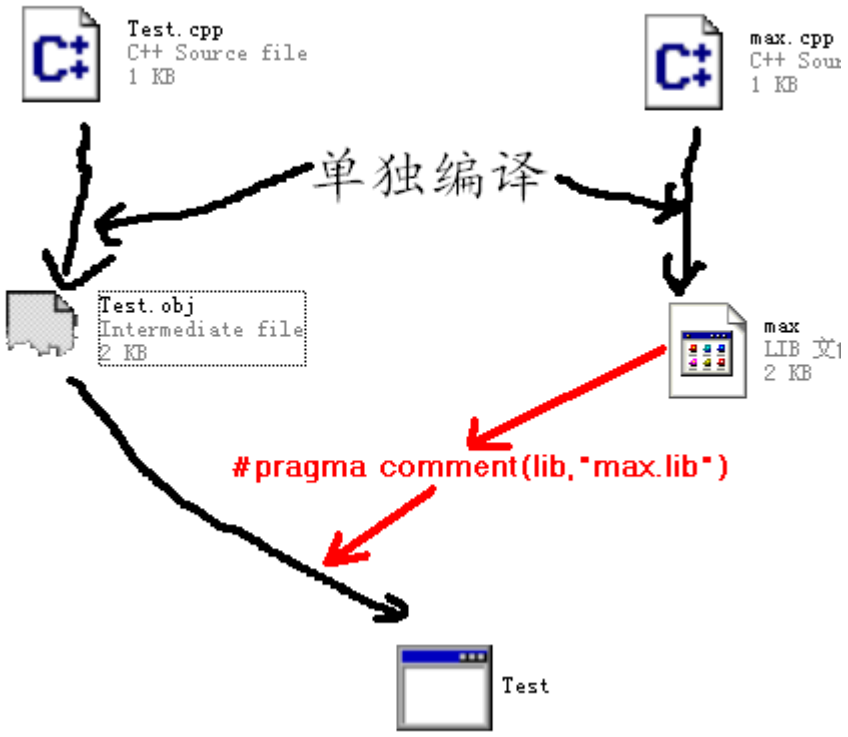
然后编译，终于成功了，耶～～～～

说实话，以上的过程是我们一个探索的过程，牛刀当然可以直接给大家建立一个静态库，然后用一个程序去调用，但我们这种探索的乐趣就全没了，大家说呢，呵呵。

好了，我们上面的过程告诉大家一个事实， lib 文件是一个静态库，但在动态链接库中，这个文件就是一个导出表，当然，我们在以后详细给大家介绍。

我们这一讲不打算详细与大家讲链接库的知识。

也就是说，我们的多文件系统其实是这样编译与链接的：




也就是说，每个文件，都是可以由每个自主的编译器单独编译的，然后遇到了 #pragma comment(lib,"max.lib") 则将一个静态链接库加进来，形成了一个 test.exe 文件。

当然，以后我们讲到动态链接库就不是这样了，动态链接库是一个单独的文件，并不随着编译器编进我们的 exe 文件中，所以有动态链接库的软件，如果相应的 DLL 文件丢失了，则会报错。

这一点大家注意了，删文件时，动态链接库文件一定不要乱删，否则好多的软件就可能运行不起来。

当然喽，现在估计不可能出现这种情况，因为大家的硬盘都大了，在以前这问题可就严重喽，呵呵，删文件当然拣大的文件删啊？什么文件大呢？动态链接库。什么动态链接库大？系统的动态链接库大。

呵呵，结果可想而知吧？很可能你删掉了一个文件，你的电脑就完了，重装系统吧，呵呵。

还是那话，上面只是一个小插曲，一般我们自己做输入，用不着使用静态库，除非别人的静态库已经编译成了二进制文件了，并且没有源代码提供给我们，我们才不得已地去使用这种静态库技术。

当然，现在我们正常编程，我们分成多文件系统就行了。

好家伙，讲了半大天才点题，呵呵，我们还来做上面的 max 程序吧。

```
//test.cpp
#include <max.h>
int max(int,int);
void main()
{
        int a=2;
        int b=3;
        int c;
        c=max(a,b);
}

//max.h

int max(int,int);

//max.cpp

#include "max.h"

int max(int a,int b)
{
        if(a>b)return a;
        return b;
}
```

上面的文件是我们最常用的结构，是不是大家觉得与我们宏贴不上边啊？其实我们可以将 #include 也当成宏来看。

瞧，牛刀宏的概念可广喽，那么我们为什么可以将此概念当成宏来看呢？我们看牛刀将上面的程序转换一下：

```
#include "max.h"        //---  int max(int,int);

int max(int a,int b)
{
        if(a>b)return a;
        return b;
}
```

这是不是一种替换呢？是啊，也是一种替换啊，只不过不同的是，我们替换的是另一个文件中的内容，正因为这种替换，才使得我们的程序能够找到函数原型，从而编译不报错，这难道不是宏？我们再来做一个例子给大家——

```
#define MAX int max(int,int);

MAX

int max(int a,int b)
{
        if(a>b)return a;
        return b;
}
```

大家感觉是不是似乎也很像近？但正如我们的单行替换中的一种特殊情况一下，我们提出了“模板”的概念，模板是一种编译一遍过再替换的技术，那么我们的 #include 当然也必然会有其特殊的用途喽。

那是当然，包含语句除了可以替换掉原来的头文件以外，还告诉了我们的编译器，在链接的时候，和哪几个 obj 文件进行链接，并且在编译的时候，我们需要编译哪几个 cpp 文件或者 c 文件。

这种功能是宏所没有的。

当然，还是那句话，如果你的语言底蕴很深了，也可以认为牛刀的说法不合理，原因也很简单，上面的一些理论与理念，只是牛刀的独特的想法罢了，呵呵。

那么我们为什么需要将每个文件单独编译呢？有什么好处吗？

当然有，那就是：我们每个文件只需要编译一遍就行了，如果直接就是二进制文件，甚至不需要编译，这就是好处。

什么，难道还有文件需要引用很多遍的？

是，我们做个程序看一看：

```
// max.cpp
#include <stdio.h>
int max(int a,int b)
{
        printf("进入了求最值的函数\n");
        if(a>b)return a;
        return b;
}

// test.cpp

#include "max.h"
#include <stdio.h>
void main()
```

```
{
        int a=2;
        int b=3;
        int c;
        c=max(a,b);
        printf("最大值为:%d\n",c);
}
```

请大家看，我们在两个文件中，都使用了 stdio.h 头包含。为什么要这样做？因为我们两个文件中都使用了 printf 函数，所以我们需要都使用 stdio.h 库函数，所以都要包含，那么如果我们需要编译的话，那么理论上 stdio.h 库的编译就需要两遍，大家说是吗？

如果那一个库单独编译的话，那么一遍就够喽。

当然，对于这种技术，我们可以用下面的办法来实现，定义一遍就够喽，看程序：

```
// max.h
#include <stdio.h>

int max(int,int);
//test.cpp
#include "max.h"

……
//max.cpp
#include "max.h"
……
```

这样我们就可以很容易将一些必要的东东放在一个头文件中，从而达到包含此文件的都可以使用。

所以我们一般在同名 CPP 文件中，通常包含同名头文件，并且将需要定义的一些变量、常量、宏、头包含都放进自己的头文件中去，从而使得我们的程序显得格外地清爽舒适。

当然，具体怎么使用，我们下回分解（啪!!）

]

**周边编程一〇七：文件使用**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月十日
*/
```

上一讲与大伙通过实践的办法摸索了文件系统的概况，得到了管理员的特许——深入下去，呵呵，牛刀敢不深入吗？

嘻嘻，开个玩笑，当然要深入下去，如果不深入下去，那么我们以后的文件组织又从何谈起呢？大家说是吧？

这一讲我们编写三个文件，看程序：

```
// max.h
int max(int,int);

//max.cpp

int max(int a,int b)
{
        if(a>b)return a;
        return b;
}

// max3.h
int max(int,int,int);

//max3.cpp

#include "max.h"

int max(int a,int b,int c)
{
        int d=max(a,b);
        return max(c,d);
}
//test.cpp

#include "max3.h"

void main()
{
        int a=2;
```

```
        int b=3;
        int c=4;
        int d=max(a,b,c);
}
```

上面的程序看起来费劲吗？下如果费劲的话，我们将所有的定义部分全拿了，我们来看纯的 #include 部分，看下面的定义，（因为我们不打算与大家讲程序喽，呵呵，既然是周边编程，我们突出重点——

```
// max.h
int max(int,int);

//max.cpp

// max3.h
int max(int,int,int);

//max3.cpp
#include "max.h"

//test.cpp

#include "max3.h"
```

当然，我们知道，我们的替换功能其实是这样的，看仔细喽，呵呵

```
// max.h
int max(int,int);

//max.cpp

// max3.h
int max(int,int,int);

//max3.cpp
#include "max.h"        // int max(int,int);

//test.cpp
#include "max3.h"        // int max(int,int,int);
```

上面的注释大家还看得懂吧？如果看得懂的话，其实上面的定义，我们知道，我们在 test.cpp 中是不能使用 int max(int,int); 函数的，为什么呢？因为我们的 test.cpp 中没有 int max(int,int); 的函数原型啊，大家可以从上面的例子看出来，那么我们怎么在 test.cpp 中也包含进我们需要的函数原型呢？当然是这样喽：

```
//test.cpp
#include "max.h"
#include "max3.h"
```

于是，形成了如下的文件包含关系——

```
// max.h
int max(int,int);

//max.cpp

// max3.h
int max(int,int,int);

//max3.cpp
#include "max.h"        // int max(int,int);

//test.cpp
#include "max.h"        //int max(int,int)
#include "max3.h"        // int max(int,int,int);
```

可是，根据我们上面一讲的原则，我们建议对于其他文件的头包含，最好能够放在头文件中，然后在 CPP 文件中，包含一个自己的头文件，就一切 OK 了，是吧？于是，我们的程序变成了如下的形式：

```
// max.h
int max(int,int);

//max.cpp
#include "max.h"

// max3.h
#include "max.h"        // int max(int,int);
int max(int,int,int);

//max3.cpp
#include "max3.h"

//test.cpp
#include "max.h"        //int max(int,int)
#include "max3.h"        // #include "max.h"---int max(int,int);
                         //int max(int,int,int);
```

以上的程序无意间就出现了一个新的问题，当然，我们的函数原型是可以多次定义的，于是，我们的程序问题不大，可是我们突然心血来潮，我们改变了我们的 max 函数的编程方案了，我们需要定义一个全局变量，并且还希望这个全局变量能够在多文件中传递下去，而我们定义的这个变量就是取得最小值，看程序：

```
// max.h
int min;
int max(int,int);

//max.cpp
```

```
#include "max.h"
int max(int a,int b)
{
        min=(a<b)?a:b;
        if(a>b)return a;
        return b;
}
```

程序大家还明白什么意思吧？我们希望求最大值的同时，也将最小值从程序中给带出来，于是，我们在 max.h 头文件中定义了一个求最小值的函数，于是，我们的程序结构形成了如下的情况——

```
// max.h
int min;
int max(int,int);

//max.cpp
#include "max.h"

// max3.h
#include "max.h"        // int min;
                        //int max(int,int);
int max(int,int,int);

//max3.cpp
#include "max3.h"

//test.cpp
#include "max.h"        //int min;
                        //int max(int,int)
#include "max3.h"        // #include "max.h"---int min;
                                    //int max(int,int);
                        //int max(int,int,int);
```

编译上面的程序，我们就会得到一个报错——

```
--------------------Configuration:        Test        -        Win32
Debug--------------------
Compiling...
max.cpp
Test.cpp
d:\pvc\test\max.h(2) : error C2086: 'min' : redefinition
max3.cpp
Error executing cl.exe.

Test.exe - 1 error(s), 0 warning(s)
```

这个报错可能是我们意料之中，因为我们通过来面可以看到：

```
//test.cpp
```

```
#include "max.h"          //int min;--定义了一次最小值
                          //int max(int,int)
#include "max3.h"          // #include "max.h"---int min;又定义了一次最小值
                                                  //int max(int,int);
                          //int max(int,int,int);
```

我们的确定义了两次最小值，从而形成了重复定义，于是乎，我们的程序就出现了报错。

牛刀曾经在关于程序的变量的作用范围的部分曾经说过，我们在不同层次的作用域中，可以定义同一个变量，这种作用域是靠大括号——"{}" 来限定的，而在同一个作用范围是永远不允许同名变量的，这一点来说，我们经常会出现下面的程序：

```
for(int i=0;i<10;i++){...};
for(int i=0;i<20;i++){...};
```

这种结构的重复定义其实很容易出现，但奇怪的是，好像新版的 ＶＣ是允许这样定义了，不属于重复定义，当然，牛刀用的是ＶＣ６．０，至少这个版本还是基本满足牛刀的理论的，呵呵。

再一个重复定义的出现就是多文件系统中，因为我们一特别是多个人做共同开发的时候，我们是不知道别人的头文件中是否用到了我们也需要包含进的头文件，多而达到一种协调的目的，再说，这种协调也是不切合实际的，好，你与你的团队能够协调了，你能够和微软协调吗？你能够与第三方 DLL 文件提供者协调吗？显然是做不到，是也不是。

既然做不到，我们就需要想办法去解决这个问题，牛刀上面其实是有意地定义 int min; 这个变量，以便提出我们新型的错误和解决方案的，牛刀说过，不建议大家做全局变量的，现在也不建议。

那么既然说到解决方案了，那肯定是有解决方案的喽，那当然，那就是——条件编译。

又回到了我们的宏上面来了。

当然，为了让大家清楚怎么使用，我们还是用单文件系统来讲：

```
//test.cpp

int min;

int min;

void main()
{
}
```

就上面的情况，就可以模拟出我们重复定义了，当然，单文件中，我们很容易删掉一个，然而，我们的目的不是用删除的办法去解决问题，怎么办？条件编译——

条件编译：是一种编译技术，让编译器根据一定的条件去选择，哪些代码需要编译，哪些代码不需要编译了。

#ifdef 宏

如果宏被定义了，就编译下面代码，直到 #endif 结束。

#ifndef 宏

如果宏没有被定义，就编译下面的代码，直到 #endif 结束。

#endif

不用说了吧？条件编译结束标志，呵呵。

下面我们就来用条件编译的方法来做如下定义，大家看看效果：

```
//test.cpp
#ifndef MIN
#define MIN 1
int min;
#endif

#ifndef MIN
#define MIN 1
int min;
#endif
void main()
{
}
```

这样一来，我们的重复定义问题就解决了，那么怎么解决的呢？我们给我们的程序加进注释——

```
//test.cpp
#ifndef MIN          //如果 MIN 宏没有被定义，则执行下面的条件编译代码（当然，此时，MIN 宏并没有被定义，所以下面语句可以得到执行。
#define MIN 1        //这一句代码可就损了，呵呵，进来就定义了这一个宏，下面斥千万别认为没有定义了。
int min;             //于是，这一句 int min; 的定义就顺利地被编译器给编译了。
#endif               //直到结束，这一段代码就被编译器认可了。

#ifndef MIN          //下面几句代码是不是和上面四句一模一样？一样是一样，但是由于上面的条件编译执行完了，一执行完了，那么 MIN 就已经被定
```

义了，所以下面的所有（注到 #endif 之前）的条件编译代码，编译器就拒绝编译了，于是，下面的代码就是——形同虚设。

```
#define MIN 1
int min;
#endif                    //直到这儿，这部分条件编译，就仅仅是一个空架子。呵呵。

void main()
{
}
```

大家看出道道来了吗？我们的宏与条件编译整个加在一块儿，实现了我们有些代码不需要编译，而有些代码需要编译，从而就很好地限制了多次定义的风险。

也就是上面那四句，以后不管再出现几次，都是形同虚设，所以说，通过这种条件编译的手段，我们解决了多文件系统中的重复定义的风险。

嘘!!! ～～～————————@＃￥％……＆×（

总想息一口气，呵呵，想将这一部分讲明白，真是不容易啊，不知大伙明白了吗？（注：没明白发贴子来问，关键知识点，牛刀是一定要给大家说明白，说透切的，这些东东可不能囫囵吞枣哟）。

好，解决方案找到了，于是，我们上面的定义就可以变成这样：

```
// max.h
#ifndef _MAX_H_H_H_____
#define _MAX_H_H_H_____ 1
int min;
int max(int,int);
#endif
```

呵呵，是不是大家觉得上面的定义有点怪怪呀？这不是说定义的怪，而是一定要怪，为什么呢？如果我们真的在这儿定义一个 MAX ，当然可以，并且一点问题都没有，可是如果一旦别人在其他的地方也因为需要，定义了一个 MAX 宏呢？那就形成了一个十分特异的奇怪现象，你的程序一点毛病都没有，他的代码也任何毛病都没有，但是好多的代码相互一拼，就会出现某些变量没有被定义。

什么意思呢？你的宏如果被别人定义了，那么我们的 int min; 的定义，永远也不会被定义，于是，我们以后再去用这个变量，岂不就是没有被定义吗？于是，对于这种文件防重复的宏定义，我们有一些规则——当然是牛刀规则啦，呵呵。

- 一：宏定义得越怪异越好，越长越好，因为我们的宏在其他的地方根本用不到，而目的仅仅是为了防止重复。
- 二：如果没有经验，可以按如下规则—— _文件名_H__

注：这只是习惯问题，因为好多的头文件都是这样规定的，我们这种特异性的规定，就是为了避免我们定义时与其他定义重复。

• 三：定义时，可以不需要定义出具体的数据，例如上面的定义，我们可以简单地定义：　　#define _MAX_H_H_H_____
　　而不需要定义出　1　，因为这种数据定义没有任何意义。
• 四：每做一个头文件，都需要这样处理，只有这样处理了，我们以后做程序时，才可以放心大胆地包含进任何的文件。
　当然喽，ＶＣ内部的规定是在前面加两道下划线，我们一般定义前面不加下划线，用大写就可以代表宏了，而这里在防重复定义时，建议前面加一道下划线，这样不会与系统定义重复，也不会与我们自己的定义或他人的定义重复

　例如我们以前曾经说过，系统就有一些宏定义，用起来可能大家觉得特别，例如：　__TIME__ 用这个宏就可以直接显示出时间，看程序：

```
//test.cpp
#include <iostream.h>
void main()
{
        cout<<__TIME__<<endl;
}
```

　呵呵，是不是很好玩？很简单？很舒服？

　嗯，舒服的确是很舒服，咱们再来看看一些系统为我们定义好的防重复定义的宏吧，呵呵：

```
// B.h: interface for the B class.
//
//////////////////////////////////////////////////////////////////////
/

#if !defined(AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_)
#define AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class B
{
public:
        B();
        virtual ~B();

};

#endif
// !defined(AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_)
```

　　好　家　伙　，　看　见　了　吗　？　这　么　长　：

AFX_B_H__CCC43B76_4F1D_43BE_A150_E8F1AC25FD11__INCLUDED_

呵呵，讲到这儿，我们讲的已经比较深入了，呵呵，至少我们认识了宏有以下几种功能——

·一、简单替换功能：实现一些程序的通用性。

·二、条件替换功能：实现一些简单的函数，一些简单的模板

·三、文件替换功能：实现文件的包含、文件间的协调与为编译器链接打下基础

·四、程序生成功能：实现一些程序的自动生成，可大可小，动态改变我们程序的结构

·五、防重复定义功能：防止数据、类、函数的东西的重复定义，从而使得我们的文件结构更加合理，更加安全。

**周边编程一〇八：文件组织**

/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月二十四日
*/

大家安好？

其实说实话，牛刀这一讲以及下一讲其实早就写好了，但看惯了牛刀文章的朋友肯定会知道牛刀的风格，所以牛刀上面写好的一些文章不发了，牛刀将在以后若干讲的文章中，改一改写作风格，还记得今年的五月十二日？不说了。

当然还不忍说两句，其实这种事情的发生，是可以避免这么大的损失的，但为什么损失还如此地大呢？因为如今的人活得太舒服了。

曾听以前的一些有经验的人谈起过，其实这种撤离可以做得很迅速的，经过训练，只需半分钟不到，房屋中的人就都可以撤离的干干净净。

然而，人们头脑中的这根弦都放松了，大家可知道，人们面对危险，才会有防范意识。曾经记得几个很有名的案例——

·淹死的，都是会水的。
·汽车撞死的，都是好人（听说过几个瞎子被汽车撞死的？）
·青蛙用开水烫，肯定会跑了，而如果放在凉水中用小火，可以将其煮了。
……

当然，这几天在网上看了看，其实也有一些迹象，例如有些地方的青蛙爬得满地都是。

忽然，牛刀想到了一点，咱们搞软件为什么不可以做一款类似于报警类的软件呢？（有关这一点，牛刀将在合适的时候在水区与大伙讨论）

行了，当然，关于默哀三天啊，等等，这些就是国家的一些行政职能了，国家发挥这些职能，也类似于我们的操作系统。既然讲到这儿了，我们就适当深入一点。

国家一个命令（一般以发文的等式或媒体传播的进行）就使得我们的五月十九到廿一日定为了国家的致哀日，我们的操作系统其实也有同样的功能。

谈到这一点，我们不免就想再来说说我们 Windows 系统与 Dos 系统的不同与关系。

Dos 就像一个人，这个人管理着整个电脑的一切，可惜的是，他每一个时刻只能做一件事情，但比较专业

Windows 就像一个国家，由国家的政府机关管理着国家的一切，并且有着各自不同的分工，所以可以同时做好多的事，而国家领导者只需要下发指令即可。

我们 Windows 这个国家的指令就是 API ，而国家领导的集体就是整个 Windows 系统。

那么我们这个国家有哪些机构呢？文件是如何组织的呢？这就需要使用到我们的一些机构了。

我们的文件系统就是国家的最底层系统，这个系统很像国家的下水道，或者是电网，电信等机构，专业负责数据的管理工作。

我们知道，我们在 Dos 时代，一切的数据都是由我们一个 CS:IP 结构的指针去管理的，所以在 DOS 环境下，我们只需要充分调动 CS:IP 指针的积极性，我们几乎可以通吃一切。然而，在图形操作系统下，我们不行。

我们就假设我们的图形操作系统是一块大画布，那么我们这个画布上的一切就都运动不起来，因为我们的画布毕竟仅仅是画布，那是静止的东西那么我们怎么使得这些画布能够运动起来呢？图片当然不能动，那么我们就只有将画布裁成小块，每小块都画一个图片，然后去移动画布，我们的整个图形系统就运动起来了。

可是一个问题又来了，那么我们如果让大家任意在画布上涂抹的话，那么我们的电脑就会五彩缤纷，然而，人们却不乐意那样，因为如果是那样的话，那么我们玩程序的确是一件很好玩的事，然而，我们做程序就变成了一件很痛苦的差事。

举个例子吧，一个按钮，我们就需要绘制弹起的图片，按下的效果，选中的效果等等等等效果都要考虑好，然后建立一些画布，然后去不断地换。

当然还有一个办法就是，我们将这些画布都按某种方式建好，甚至可以用印刷的办法去批量地生产，然后在需要按钮那个画布时，拿出来用就行了，于是，这些准备好的画布就单独形成了一个盒子，我们在计算机中称之为——库。

其实我们的计算机中的库很多，图片库、图标库、光标库、字体库……
还有一些专业的例如指纹库、声音库、音色库……

这些库由于类别的众多于是就形成了种种的文件，为了区别，我们给这些文件定义了不同的后缀名，例如我们这一讲准备与大家详细讲的动态链接库就是这样的一个库。

大家真的相信使用一个 CreateWindow 就可以创建一个窗口吗？嗯，不对。

其实那个函数就是调用动态链接库中的一个画这种图片的程序，去为我们画出我们的窗口。

当然，作为计算机这一个国家来说，每画出一个东东，那就需要为其作一个编号，以便日后可以认识，就像我们国家为我们每个公民准备了的居民身份证一样，也只有那样，我们才能挑出每一个人来。

举个例子吧，我们江苏可以有一个人叫王平，四川也有一个人叫王平，当地震来临之后，如果有报道，一个叫王平的人在地震中受伤，那么我们绝对对江苏的这一个王平放心。
但如果我们的汶川如果有两人叫王平，但其中有一个人不幸，那么两家人就都会画魂了。

牛刀说的这个现象，其实在电脑中也是一个很严肃的问题——重名问题。

Dos 下的同一路径，也就是说，Windows 下的同一文件夹，不允许有同名的文件，这样的规则保证了我们系统访问时的安全性。

正如人们正因为有重名现象，所以我们需要身份证号一样，系统就是通过一个十六位无符号数据来管理系统的资源的，所以说，理论上，我们的系统能够使用的资源，如 HWND HDC HPEN HBRUSH ... 这些东东的个数不能超过 65536 个，所以我们以前与大家讲过，请大家使用过的资源及时释放，就是这个原因。

也就是说，这个国家最多只能有 65535 个人，那么这个国家的就需要通过一些手段去限制人口的增长，所以像 ReleaseDC EndPaint 等函数就是计算机系统的计划生育委员会，如果没有这个机构了，那么我们的系统早迟要崩溃。

好了，我们不与大家将这些东东扩展开去讲了，我们这一讲主要是讲文件的组织，那么我们首先要明白一点的是什么？ Dos 是单任务操作系统，其实 Windows 也是单任务操作系统，所以说，Windows 其实是一个由操作系统包裹的整体软件，而我们在其中运行的所有的软件都是这个系统的一分子。

那么就有一个问题，我们的操作系统是怎么管理计算机中的各个软件的呢？
那就是——消息机制。

而投递消息的通道就是——动态链接库。

为了说明问题，我们先来做一个动态链接库的例子：

大家按 文件/新建.../[工程]/Win32 Dynamic-Link Library

然后输入一个文件名，如 DllTest，当然需要再建立一个 .cpp 文件喽，

呵呵，好，这下我们进去编程：

```
int max(int a,int b)
{
        return a>b?a:b;
}
```

多么简洁的代码，呵呵（牛刀是不是有点孤芳自赏啊？噢，对了，默哀……）

大家编译后到 Debug 目录下看看，怎么没有 exe 文件？却只有一个 .dll 文件，是么？

其实这个文件我们根本没办法用，为什么？

假如你家想贡献出来开博物馆，但游人来了，却找不到门，那么怎么进去观光呢？是吧？也就是说，我们上面的程序其实是一个根本就没有用的程序，因为我们没有对外开放。

那么我们怎么开放我们的函数呢？前面加上一个特殊的定义就行了，而这个开放的文件就是 .lib 文件，大家现在看 Debug 目录下是不是没有，我们开个门就有了：

```
_declspec(dllexport) int max(int a,int b)
{
        return a>b?a:b;
}
```

不错，就是前面的 _declspec(dllexport) 将一窗通往外界的门打开

而外界如果想使用这个函数，那么就要用 _declspec(dllimport) 将这个函数包含进来，就可以使用喽，于是，无意中，我们将函数从程序中分离了出来。

那么既然分离出来了，肯定有分离出来的好处，那么好处就是——动态链接库不属于程序本身。

可能大伙有些纳闷，怎么我做的程序，却不属于程序本身。

这就相当于我们的国家，不能都是农民。

什么意思呢？农民种出来粮食自己食用，我们称之为自产自销，我们以前曾经做过的程序基本都属于这一种，但国家需要生存也需要粮食，怎么办呢？在过去，农民也有农业税，两上交，就是为了解决这个问题，交上去的粮食就是国有粮食，由国家统一管理了。当然农民自己不够吃了，也可以去领，或者你家中有一位是国家公务员及国家工作人员，于是这一部分的公粮也就了所有程序所共有的命脉。

什么意思呢？也就是你只要有能力，就可以直接向国家中央机构进行渗入，取得一些特权，甚至也是国家的命令发布者，只要你挥手一呼，大家都为了汶川的白姓，献出自己的每份爱心，于是，其他人就随风而动。

有这么悬吗？其实就是这样的，什么原因呢？

原来啊，我们以前做的程序，是单任务程序，我们只在做一件事，而我们现在做的事，需要有两个文件去完成，就是一个源文件（程序文件）与一个动态链接库文件，而一旦动用到动态链接库了，那么就必须使用操作系统提供的服务喽，那么也就是说，我们的链接库其实是由系统为我们加载的，那么系统一加载，就无疑中要触动链接库的核心，于是，其实与其说我们调用动态链接库，还不如说，其实是操作系统去调用动态链接库，然后为我们服务的呢。

有一个问题来了，既然是系统先调用了动态链接库，那么我们用动态链接库去打劫系统的构思不禁油然而生。

怎么感觉做人都有一种恶心的感觉，呵呵，好玩似的；但如果不这样说吧，牛刀再也想不出更好的解释了。

就好像警察做了好事回来却发现枪没有了的那种感觉，不过你别说，还真就是那么回事。

大家还记得我们以前曾经与大家说过关于输入法的编程吗？我们想要打劫系统的按键消息从而产生新的消息，那么就必须通过这种打劫的手段去实现。

当然，关于链接库的一些具体技术我们就放到下一讲去讲，我们这一讲就与专门与大伙讲有关于文件的事。

当我们的一些程序在系统中运行起来时，有两种形式——一种是二进制形式，另一种是解释平台的形式，也就是说，我们的程序都是看得见的，而整个过程，都有某一个软件去一步一步地解释，然后运行，像我们的 HTML 语言，JAVA 语言，其实就是这类语言，其实像 Basic 等语言也是解释性语言，解释性语言允许我们直接用语言去写一些脚本，然后通过脚本去干一些事情。而系统提供的正是对这种脚本解释的支持，使得好多的病毒得到系统的一些东东显得异常地方便与简单。

还回到我们的 DLL 文件上来吧，我们的 DLL 文件其实不是解释性文件，而是真正意义上的二进制文件，但这个文件有其特殊性，就是这个文件一旦运行起来，就不属于任何一个程序本身，而属于操作系统，也正因为如此，我们的操作系统就要对其进行管理，一旦管理，我们的程序就无形之中在操作系统中活跃起来，或者说，给操作系统开了一扇后门。

打一个不太恰当的比方，一个农民，子女考上了公务员，于是，子女的职能就由家庭成员转变成了国家干部的一份子，于是，这些子女就有了进入国家机器的通行证，于是，就可以通过一些手段去控制国家的一些机器运转，有点绕吧，我们下一讲与大家具体讲明白，当然讲明白，不免就要多说几句，于是，可能这些问题就可能需要分成好多讲，这也是没办法的事情，哎～～～

其实也就是无形之中将家庭成员公开出来作为人民的公仆了，于是，其他老

百姓有问题，就有可能需要通过这些具有公务员资格的农民子女之手，于是就形成了一种特殊的关系，我们称之为——公有化。

注：这一讲，我们就不给大家贴图片了，我们用箭头来表示一些东东：

a.dll *->b.exe
*--->c.exe
*--->d.exe
*--->e.exe
*...
...

大家看上面，我们制作一个文件， a.dll 这个链接库不属于 b.exe 所有，其他所有的 c.exe d.exe ... 都可以与其发生一个联系，这种联系是公开的，公平的，公正的，甚至我们的操作系统也都可以与其发生联系。

就像过去的两上交，如果某人当了村长村支部书记，那么他们就有权力代收两上交，于是，钱就都到了自己的腰包。

当然，人类社会因为有法律的约束，这部分资金是不能挪着自己使用的，可惜的是，计算机系统可不吃不一套，拿来就是自己的，于是，我们的程序无形之中就通过 DLL 的手段对其他的程序展开了侵略。

注：牛刀不想培养黑客，但是谈到了技术，是没有黑客与非黑客的界限的，我们不能因为这些技术会被人们用来干坏事就不讨论她。
何况，我们做程序就是做人，社会固然存在着不协调的一面，但牛刀希望看牛刀教程过来的朋友都能本着一种友善的心，因为这以后几讲的内容可能会使得你侵略到其他的程序，那么再过后我们的网络技术那一部分可能无疑就会使得你有能力去侵范别人的电脑，我们能因为其有负面影响就不说吗？那岂不是怕噎着就不吃饭了？

大家还记得打劫吗？牛刀教过大家用好多办法去打劫自己做的程序的消息，并且我们的消息机制也是通过打劫的手段来实现的，那么，如何打劫其他进程的消息呢？那就需要使用 DLL 了。

说实话，好多打劫行为真的是严重地违反了计算机内部系统的稳定的，然而有这种技术——HOOK 技术，也就是做程序的人通常所说的——钩子。

呵呵，这儿是一个垫笔，那么具体怎么深入这些技术呢？请听下回分解。

**周边编程一〇九：周边体验**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月二十五日
*/

　　上一讲好像就顾和大家闲聊天了，一点实质性的东东都没触及到，这一讲我们就来触及一些系统的东东，当然，还是那句话，我们学的是技术，请大家善待技术。

　　好了，当然是和大家从最基础的讲起，请大家像上一讲那样做一个动态链接库工程　DllTest　然后输入下面的代码——

**代码:**

```
_declspec(dllexport) int max(int a,int b)
{
        return a>b?a:b;
}
```

　　然后编译，但不要运行，因为我们的动态链接库是不能直接运行的，但的确是一个二进制文件，什么意思呢？因为我们每一个程序都要有一个入口函数，没有入口函数的程序是没办法运行的，也就是说，我们必须想办法将动态链接库运行起来，我们编的一些代码才能够起作用，那么怎么运行呢？

　　好，我们建一个新工程，名称就瞎起一个，如　DllMax.cpp　吧，呵呵，我们起什么名称都无所谓的，这次我们写一个控制台应用程序：

```
#pragma comment (lib,"DllTest.lib")

//请注意，上面是一个宏，所以我们最后不需要分号

#include <iostream.h>

_declspec(dllimport) int max(int a,int b);
//请注意，上面是　dllimport　表示包含进我们的动态库中的函数
//其实是一个函数原型，包动态库的函数我们就需要这样做

void main()
{
        cout<<"max(2,4)＝"<<max(2,4)<<endl;
}
```

　　我们编译会有一个错误，什么原因？找不到　DllTest.lib　文件与链接库文件。我们去复制过来，放在我们的工程目录下面：（注意不要放在　Debug　目

录下，再次编译，可以得到结果：）

```
max(2,4)=4
```

好，下面牛刀与大家来个逆向思维，我们将这个工程下的 DllMax.exe 复制到 DllTest 工程下，然后去调试我们的 DLL 文件：

这次大家运行 DLL 文件，不是不能运行吗？不错，需要我们给一个可以运行的文件，调试时， DLL 编译器自动先去运行可运行文件，然后就可以调试我们的链接库了，大家按后面的小三角尖头，按 Browse... 然后加入我们上面编好的 DllMax.exe，然后确定：

注意请大家将文件放到 DllTest 的 Debug 目录下，否则是找不到 DllTest.dll 文件的。

于是，我们就可以顺利调试我们的 DLL 文件了，这就是我们 DLL 文件的调试技术——

我们改一改我们的 DLL 程序，然后直接按运行——

```
#include <stdio.h>

_declspec(dllexport) int max(int a,int b)
{
        printf("进入了 max 函数\n");
        return a>b?a:b;
}
```

于是，屏幕上就出现了以下的结果——

```
进入了 max 函数
max(2,4)=4
```

我们当然可以将函数的功能给改了，再看——

```
#include <stdio.h>

_declspec(dllexport) int max(int a,int b)
{
        printf("这个 max 函数的功能被我们改了，改成了求最小值\n");
        if(a>b)return b;
        return a;
}
```

看结果：

```
这个 max 函数的功能被我们改了，改成了求最小值
max(2,4)=2
```

这个结果证明，我们可以通过改变 DLL 文件而不用去改变程序本身，从而改变我们程序的运行结果，从而给出了我们 DLL 文件的一个独有的功能——

中间件技术。

大家知道好多的软件可以网上升级吧？那是怎么实现的呢？其实很简单，只需要改变一些特有的 DLL 文件即可，当然还有一些软件可以做成脚本解析技术，然后将需要升级的东东做成脚本，于是，他们就可以使用脚本去达到升级的目的。

一些网络游戏软件如——《传奇》等，就是使用的脚本技术，当然，有一些外挂，也是使用脚本技术，他们只需要载入一定的脚本，就可以完成不同的任务目标，如购物、打钱、拣钱、拣东西……

好像又有点跑题了。

既然跑了，就多跑两句吧，珍惜生命，善待生命。

游戏中，我们的角色的生命消失了，可以重生，而事实上是不可以的，请大家能够分清游戏与现实，善待生命。

听说过网吧中的血案，某一小朋友向其他一个小朋友要钱，没给，拿一把刀就将别人戳了。

虽然现代社会的知识结构告诉人们，人没有重生，但游戏却告诉人们，死亡是一件不过如此的事。所以从这一点上来说，游戏给人们的影响基本都是负面的，或者说是负面影响要大一些，总之，我们要善待生命，当然，也希望游戏从打打杀杀的怪圈中脱离出来。

其实从这一点上来说，还是感觉棋类游戏比较好。

好了，还是回到我们的 DLL 文件上来吧。大家一定要明白一点的就是，DLL 文件不属于任何软件本身，而属于操作系统的一部分。

既然属于操作系统，我们就有一些很特别的、很有意思的事——

a.dll *->b.exe

*--->c.exe

*--->d.exe

*--->e.exe

*...

...

大家看，我们的 a.dll 同时属于 b.exe c.exe e.exe ... 那么，我们作为一个多任务操作系统，假如 b.exe c.exe e.exe 同时运行了，那么 a.dll 是属于哪一个可执行文件所有呢？是只运行一个 dll 文件，还是出现了多个 dll 文件的运行版本呢？

这就是 Windows 做得比较绝的地方，其实运行的还就是一个 dll 文件，而同时给不同的文件提供服务，于是，就出现了专业的库。

如上面我们讲的，我们的库一旦改变，那么软件不需要改，就自然地有了新库所定义的所有的特性。

如果大家玩过以前版本的 Windows 那么大家一定知道，新的 Xp 版的程序标题栏上的最大化、最小化和关闭按钮就比以前的版本的关闭按钮要好看得多，并且在新的版本下运行的程序界面都统一，这是为什么？

这就是因为我们所有的软件都用到了三个基本的库——kernel32.lib user32.lib gdi32.lib

而如果想改变界面风格，就只需要将这三个库中的一些库中的代码作一次改变，然后重新编译一下，我们的就可以将所有程序的界面风格就都改变了。

为什么这样做呢？这样的技术，使得微软想发行新版本的 Windows 就显得异常地方便也正因为这样的技术，使得我们的 Windows 软件风格显得异常地统一。

但是有一个问题来了，如果 a.dll 同时为几个软件服务，那么 a.dll 中如果有共用数据怎么办呢？如果每一个软件都可以去改变公有数据，那么我们软件的安全不就成问题了吗？

于是，微软用的技术是，每一个公有数据在遭遇不同软件的第二次更改时，都将公有数据重新备份，从而使得我们的 DLL 显得异常地安全。

也仅仅是异常安全，而不是非常安全，如果是非常安全，那么我们的好多非常恶心的目的就没有实现的可能了。

当然，链接库的另一个功能就是实现软件的接口技术。

大家知道，一个函数，如果我们的入口参数与返回值不变，那么这个函数本身就不会变，也就是说，函数的调用就不会变，那么函数内部的程序不论如何变化，都不会去改变软件本身，但是软件的功能却可以通过函数本身去任意改变。

大家还记得吗？我们 Windows 的入口函数一定要是 WinMain 为什么呢？因为我们的 exe 文件就相当是系统的一个 DLL，所以我们的入口函数不能变，其中的参数也不变，那么我们就可以在我们的程序中去随意地实现我们自己的功能。

于是，操作系统成为了我们ＶＣ＋＋开发图形操作系统的一个平台。

当然，如果有其他的操作系统，当然也可以用同样的办法去开发出自己的一套平台，自己的一套 DLL ，自己的一套 API ，去让别人用自己平台提供的一些功能去做一个新的开发平台。

名人机就有这样的一套平台，并且其入口函数是 GuiMain 仅此而已。当然，我们也可以提供我们自己的全新的开发平台，使用全新的函数接口，从而使得别人能够顺利地使用我们提供的平台接口去开发新的程序，一样的。牛刀这里就举一些例子跟大家玩玩，作为这一讲的结束吧：

在 DllMax.cpp 中，将程序改了：

```
#pragma comment (lib,"DllTest.lib")

#include <iostream.h>

_declspec(dllimport) void MyMain();

void main()
{
        MyMain();
}
```

再将 DllTest 下的程序改了：

```
#include <stdio.h>

_declspec(dllexport) void MyMain()
{
        printf("新的入口函数\n");
}
```

将新的入口函数的编译 dll 与 lib 文件复制到 DllMax 目录下，然后编译 DllMax 程序：

再将 DllMax 的 Debug 目录下的 DllMax.exe 移到 DllTest 目录下 Debug 目录下，有点绕吧？？？

其实我们这是为了让大家理解，我们这是两套程序，其实真正地做起工程来，是通过在工程中设置一些参数来达到目的的，不过我们这一讲不与大家讲那些东东了，如果那些与大家讲明白了，可能又需要用一讲甚至更多的内容，大家还是跟着牛刀绕绕点好，并且每做一件事，自己要思考的一件事——自己是在干什么的，弄明白了这一点，可能你就彻底地明白了 DLL 文件是怎么回事了（如果不还绕明白，发贴子上来问）。

好了，这下子我们进到了 DllTest.cpp 下，我们就有了一个新的入口点函数 MyMain 了，以后只要有我们编译好的 DllMax.exe 文件在，我们随时随地都可以使用这个文件与新的入口点函数去做我们自己的程序。

这就是以前梦影与我讨论了好久的问题，是否能用 void main(); 作为函数入口的问题，你瞧，这次，我们竟然连 void main();都不用了，我们使用一个 void MyMain(); 函数竟然也能行。

于是，我们似乎明白了，微软的编程环境在后台给我们做了些什么，大家说是吗？

当然，我们可以在我们新的以 void Main(); 为入口的程序中，做一些我们自己想做的任何事情了，好了，下面我们就给大家做一个经典的程序吧，弄一段程序，给大家显示 π 的 800 位，看程序——

```
#include <stdio.h>
int a=10000,b,c=2800,d,e,f[2801],g;
_declspec(dllexport) void MyMain()
{
        for(;b-c;)f[b++]=a/5;
        for(;d=0,g=c*2;c-=14,printf("%.4d",e+d/a),e=d%a)
                for(b=c;d+=f[b]*a,f[b]=d%--g,d/=g--,--b;d*=b);
}
```

这是在网上看到的一款很经典的程序，计算效率异常地高，结果是——

```
31415926535897932384626433832795028841971693993751058209749445923078164062862089
98628034825342117067982148086513282306647093844609550582231725359408128481117450
28410270193852110555964462294895493038196442881097566593344612847564823378678316
52712019091456485669234603486104543266482133936072602491412737245870066063155881
74881520920962829254091715364367892590360011330530548820466521384146951941511609
43305727036575959195309218611738193261179310511854807446237996274956735188575272
48912279381830119491298336733624406566430860213949463952247371907021798609437027
70539217176293176752384674818467669405132000568127145263560827785771342757789609
17363717872146844090122495343014654958537105079227968925892354201995611212902196
08640344181598136297747713099605187072113499999983729780499510597317328160963185
```

好了，接下来我们讨论一个很好玩的问题——

```
//DllTest.cpp
_declspec(dllexport) void MyMain()
……
//DllMax.cpp
_declspec(dllimport) void MyMain();
……
```

大家看，前面一部分其实只有括号里面的不同，如果能用一种办法去替换一下，那么——

怪怪的想法又开始了。大家还记得条件编译吗？于是乎——

```
#ifndef DLL_TEST
#define DLL_TEST _declspec(dllimport)
#endif
```

```
DLL_TEST void MyMain();
```

一个很奇怪的想法是不是？假如你还没有回过味来，牛刀将全部程序解决方案奉上，牛刀不打算与大家详细解释，大家自己去咂咂滋味——

```
//DllTest.h

#ifndef DLL_TEST
#define DLL_TEST _declspec(dllimport)
#endif

DLL_TEST void MyMain();

//DllTest.cpp

#define DLL_TEST _declspec(dllexport)
#include "DllTest.h"
...

//DllMax.cpp
#include "DllTest.h"
...
```

这儿需要大家思考了，我们关于这里的奥妙，我们下一讲揭晓，希望大家自己动动脑筋，于是，牛刀贴上来一些代码，大家看前面的一些特别的，怪怪的东西——

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow);

LRESULT CALLBACK WindowProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);

OS_DLL MView*       ViewCreate(MView *pOwner, MView *pView, UINT32 dwObjectID, MRect *pRect, UINT16 byOption);
```

以上的一些代码，都是从一些经典的程序中复制过来的，大家仔细体会其中的一些代码，如 WINAPI CALLBACK OS_DLL 等东东，在其中的作用。

子曰：思而不学则罔，学而不思则殆。

希望大家在学习的时候，多多思考。（啪!!!）

下回分解。

**周边编程一一〇：深入浅出**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年五月二十六日
*/
```

上一讲咱们讲了一段有关于条件编译的问题，我们将代码复制过来瞅瞅——

```
//DllTest.h

#ifndef DLL_TEST
#define DLL_TEST _declspec(dllimport)
#endif

DLL_TEST void MyMain();

//DllTest.cpp

#define DLL_TEST _declspec(dllexport)
#include "DllTest.h"
...

//DllMax.cpp
#include "DllTest.h"
...
```

我们先将上面的代码的 #include 部分扩展一下，我们以前曾经与大家说过，我们可以将 #include 看成是一个宏，只不过其替换部分不是一行代码，而是一个文件，看替换结果——

```
//DllTest.cpp

#define DLL_TEST _declspec(dllexport)
//#include "DllTest.h"
//以下是替换部分
#ifndef DLL_TEST
#define DLL_TEST _declspec(dllimport)
#endif

DLL_TEST void MyMain();
//以上是替换部分
...

//DllMax.cpp
//#include "DllTest.h"
```

```
//以下是替换部分
#ifndef DLL_TEST
#define DLL_TEST _declspec(dllimport)
#endif

DLL_TEST void MyMain();
//以上是替换部分
...
```

我们先看看下面的 DllMax.cpp 部分吧——

```
#ifndef DLL_TEST          //在 DllMax.cpp 中，由于我们这个没定义，所以下面部分是经过编译的
#define DLL_TEST _declspec(dllimport)          //此处定义了 DLL_TEST 请注意，此时编译了宏的值为什么
#endif                    //结束条件编译

DLL_TEST void MyMain();          //由于条件编译了上面的句，所以这一句就相当于 _declspec(dllimport) void Main();
```

再来看看 DllTest.cpp 部分的代码——

```
#define DLL_TEST _declspec(dllexport)          //定义了一个宏
#ifndef DLL_TEST                               //由于宏被定义了，所以下面的编译就跳过不编译了。
#define DLL_TEST _declspec(dllimport)
#endif                                         //由于上一句被跳过，所以大家知道 DLL_TEST 是什么定义了吧？

DLL_TEST void MyMain();                        //于是这一句就相当于：_declspec(dllexport) void MyMain();
```

由是，同一个头文件，在使用者中被编译成了 _declspec(dllimport) void Main();，而在定义者中则变成了 _declspec(dllexport) void MyMain(); 于是，一下子，我们只需要定义一个头文件，我们的定义就一下子变得具有了通用性，结果我们使用链接库时，只需要包含进一个头文件就一切 OK 了。

于是，我们应该发现了微软的良苦用心了吧？为什么我们要有函数原型？为什么要有头文件？为什么要有多文件系统，为什么头文件中要用条件编译.....

这么多的为什么，好像一下子就找到了答案。

但说实话，虽然上面的结构相当好，但是如果叫你去做一个链接库并且就导出一个文件，可能你就果真就懒得使用这么复杂的条件编译，但我们与大家说这一点是告诉大家，有这样形式的代码，并且我们要知道这种代码是在干什么，以便我们以后在看别人程序时，要知道所以然。

再者，上一讲就与大家说过，我们学习一定要善于思考，而不能满足于老师

的传授，我们自己甚至应该有自己的创新的想法，然后试图使用我们现有的工具去将其实现才好。

好了，为了与大家讲解有关系统控制的一些细节，我们下面先与大家讲一讲一种新的函数——钩子。

当然，我们讲解动态链接库也仅局限于此，不能再深入下去了，如果再深入下去好像就会脱离牛刀教程的本质了。

那么我们先来回顾一下我们以前对消息的几种打劫方案——

·方案一：消息循环函数体中，用 switch 打劫——

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT message,WPARAM  wparam,LPARAM lparam)
{
        switch(message)
        {
        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }
        return DefWindowProc(hwnd,message,wparam,lparam);
}
```

我们正常的消息响应都是使用的这个办法去打劫消息，使得我们能够实现对某个消息的响应。

·方案二：在消息循环体中打劫，一般用 if 去打劫

```
        while(GetMessage(&msg,NULL,0,0))
        {
                if(打劫消息==msg.message)
                {
                        //打劫程序
                }
                else
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
```

当然，如果我们要打劫多个消息，也可以使用 switch 语句：

```
        while(GetMessage(&msg,NULL,0,0))
        {
                switch(msg.message)
                {
                case WM_...:
                        ...
```

```
                break;
        case WM_...:
                ...
                break;
        ...
        default:
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
    }
```

能够理解吧？其实是一样的，但不同的是什么呢？方案一是取得消息队列中的最后一条消息进行响应，而方案二是直接响应系统消息，不经过消息队列就响应，所以介于这种情况，如果我们需要响应速度特快的消息，则可以使用第二种方案，如果我们要求响应速度要求不是很高，我们就使用第一套方案，当然，我们一般是使用第一种方案。

以上两种方案，一种是取消息队列的最后一条消息，一种是不经过消息队列，直接将消息打劫，而消息队列是我们 Windows 操作系统的基本的一套时间分片的功能分解。什么意思呢？大家还记得牛刀经常与大家说过的，我们的 Windows 操作系统其实还是单任务操作系统吗？只不过操作系统将时间分成了若干片，好多的程序分片使用时间，不过因为我们正常使用的片的替换速度特迅速，所以我们的肉眼分辨不清耳朵分辩不出，所以我们感觉上好像我们的计算机既能打游戏，又能听音乐，甚至还有其他重要程序在上面运行。

好了，一头一尾都可以打劫了，那么对于消息队列呢？因为那是系统的东西，所以我们没有办法直接在程序上做什么手脚，还好，微软给我们开了一扇后门，那就是——Hook。

大家看过古装的一些电视吧，过去那种窗户，一捅一个窟窿，我们就可以看到里面的情况了，其实这个钩子也是这样的一个窟窿，其实牛刀感觉与其翻译成钩子，还不如翻译成窟窿来得形象，那么我们就去我们的程序中安装一个窟窿吧——

好，下面我们安装一个窟窿，通过函数——SetWindowsHookEx 就可以，下面我们就做一个对话框的程序，然后我们在其中的 OnInitDialog 函数中加入一个鼠标钩子——

```
SetWindowsHookEx(WH_MOUSE,MouseProc,NULL,GetCurrentThreadId());
```

第二个参数是一个函数指针，我们所说的安装窟窿，其实就是用上面的函数使得操作系统能够根据我们所定义的函数指针去将消息队列中的消息导入到相应的函数中去处理，仅此而已，其实我们在做窗口类时，那个函数指针也是这个作用，说法不同罢了，用的是同一种技术。

第一个参数是指示我们导出什么样的消息，第三个参数我们这里就设置为 NULL，而第是个参数是线程 ID ，我们用一个函数去取得即可。好了，下面我们就去捕捉我们的鼠标消息吧——当然是通过定义的函数去取得——

```
LRESULT CALLBACK MouseProc(int nCode,WPARAM wParam,LPARAM lParam)
{
        return 1;
}
```

好，我们的对话框“死了”，鼠标点不动了。在其上的所有鼠标消息都被我们上面的窟窿函数给吃了。

亲爱的朋友们（是不是感觉老不舒服？别扭？），请不要同时将鼠标消息与键盘消息都吃掉，否则你要哭。

上面的程序，大家可以按空格或者 Esc 退出，如果你真的也将键盘钩子给屏闭了，那么，闲言少叙，咱们说干就干：

```
LRESULT CALLBACK KeyProc(int nCode,WPARAM wParam,LPARAM lParam)
{
        return 1;
}
……
        SetWindowsHookEx(WH_KEYBOARD,KeyProc,NULL,GetCurrentThreadId(
));
……
```

以上两个钩子千万别同时放在自己的程序上，否则你要哭，牛刀不教你退出方法，嘀嘀哒

当然，我们可以通过 CallNextHookEx 将我们不需要屏蔽的消息给传递出去，这个函数就好像我们以前消息处理中的 DefWindowsProc 函数一样，将我们不处理的消息放出去。

那么我们想打劫屏幕上所有窗口的钩子，那么我们就需要使用 DLL 文件了，因为我们的链接库必定为系统所调用，那么链接库中打劫的自然就是系统的消息循环啦（什么？系统还有消息循环？你自己去想想，子曰：思而不学则罔，学而不思则殆）。

好，我们还在我们上一次做的那个链接库中，不是有一个 MyMain 函数吗？我们在那里去建立一个钩子瞧瞧——

```
#include <windows.h>          //我们需要使用其中一些些东东，所以包含进来
#include <stdio.h>          //我们需要显示信息，所以包含进来
LRESULT CALLBACK KeyProc(int nCode,WPARAM wParam,LPARAM lParam)
{
        return 1;
}
_declspec(dllexport) void MyMain()
{
        printf("钩子测试动态链接库\n");
        SetWindowsHookEx(WH_KEYBOARD,KeyProc,GetModuleHandle("DllTest
```

″),0);//最后一个参数设置为 0 ，就和所有线程相关了。
Sleep(100000); //睡觉若干时间，程序不能退出，一退出就一切完完。
}

你运行一下，看一看效果，你到其他软件中去用键盘试试，你回去砸键盘吧，你的键盘成了废品喽。

当然，由于我们上面做的动态链接库是在单任务系统中建立的，也就是说在控制台应用程序下建立的单任务文件，所以我们一旦使用 sleep 那么程序就钉在那儿不动了，如果大家是使用多线程任务的话，我们就可以方便地实现类似于输入法的效果的程序了。

当然，输入法有其自身的一套独门的 API 函数，学起来也挺复杂，关系这些种种的效果，我们就不详细展开了，展开又是一个没完，还是那句话，可以写书喽。

当然，链接库还有其他的好多技术，牛刀无法做到面面俱到，原因很简单，牛刀不想没有培养黑客的心，结果却培养出一批黑客来，与我们的初始精神就背道而驰了，你说是不？

另一个原因，那也就是，牛刀对于这方面，其实也并不十分内行，好多都是现学现卖，如果大家觉得有必要，大家可以相互讨论切磋下去，也不适为一个很好的办法。

再说，我们的《周边编程》部分，其实是《微软积木》的一个小插曲，因为我们好多的问题需要跳出微软积木的部分去看，去试，去玩，才能够真正地弄明白，否则我们对于微软的一套 MFC 好多东西我们会很纳闷。我们这部分《周边编程》也是围绕着这样的精神展开的，所以不能每涉及到一个问题都展的太开，否则我们每一个问题都可以写上一本书。

就像这种钩子吧，类型就多得不得了，一个一个地去与大家试显然是不现实。我们只有将这些问题尽量简单化地介绍，让大家明白这是怎么回事就行了，因为这一部分知识纯粹是从宏——文件——链接库，按照这样的思路展开的。我们下一讲开始，就要往起收了，否则会连牛刀也不知道这个教程以后的发展方向了。

这一讲有关钩子牛刀讲得很粗糙，但意思到了，我们的目的是告诉大家，我们的程序如果想去管理或控制其他程序，就避免不了与链接库打交道。

当然，有一些函数可以得到其他窗口的句柄，从而对其他窗口进行写写画画，不过那种控制仅仅是主动控制，而不能被动触发，什么意思挺绕人吧，那么如果果真觉得绕我们就结束这一讲吧，下一讲我们将重新讨论一个新的专题——文件及文件的操作。

**周边编程一一一：文件读写**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：　二〇〇八年六月二十四日
备注：其实这篇文章早在廿多天前就写完了，那天突然不能上网了，于是，这篇文章写好没来得及发，还记得上次的汶川大地震吗？于是，这篇文章还是略带伤感，当然，在这段时间内，没能够来得及处理大伙的贴子还请见谅。
*/

　　今天咱们不默哀了，咱们先回顾一下下我们周边编程所走过来的一些历程——

咱们初期是从宏那儿开头的，然后从宏的一些机理、作用、技术，以及一些特殊的宏，如模板、头包含走过来的，然后从头包含又走到了多文件系统，提到了我们文件的组织，文件的使用，一直到动态链接库，咱们动态链接库那一章讲的不是十分详细到位，因为我们不想与大家做过多的深入如果有机会咱们专题讨论，否则就脱离了我们教程的初衷了。

　　既然讲到了文件，我们就不免要说到——文件的读写。

　　原因其实很简单，一个文件是放在那儿，可是我们不读这个文件，那么这个文件至少对于我们这个程序来说，没有什么太大的作用，那么我们怎么去读写文件呢？咱们还是先从库函数谈起，然后讲到封装以及数据流。

　　好，这一讲开始，咱们教程中继续给大伙插一些图片，不过统一的，我们都用黑色背景，以表示对死难者的默哀，看图——

　　在电脑上如果没有文件系统，那么我们的程序所有的东东都保存不下来，就好像牛刀以前在小霸王学习机上做程序似的，那些程序可以说，有时候一个程序能做好半天，忽然停电——完蛋喽，机一关——全没了。

　　后来有了电脑，一切全好了，咱做的程序也能够保存起来了，可是一个程序运行的数据却每次都是一样的，并保存不了，为什么？没学到保存那一章呗。学到了就一切都会好起来嘀，这一讲开始，咱们目的也是如此。

　　咱们在《跟前牛刀一步一步制作防杀衣》上曾经与大家讨论过库函数的一些文件操作——

```
HFILE fp=fopen();
fread()
fwrite()
fseek()
```

ftell()
...

下面我们还是来做一些这样的事，不过这次牛刀与大家用我们以前所学的一些加密技术，去加密一些数据，看看我们到底是如何真正地做一款有实际意义的软件的。

数据传输中，会出现一些特有的现象，例如某一款网络传输软件如果只能传输大写字母，怎么办？如果只可传输可见字符，怎么办？于是，我们不可见字符的传输就成了一个问题，那怎么解决呢？一个办法就是——我们将字符给打碎了，然后变成适合的字符形式，就可以传输了。那么，这一讲开始，我们就做这样一款软件——MakeUUe

UUe 就是这样的一款软件格式，现在正常不用了，但 WinRar 软件现在还支持这种格式，所以我们当做好这种格式的文件后，还可以使用 WinRar 这款经典的压缩软件去解压，当然从这一点上来说，大家应该明白了吧，WinRar 并不是一款压缩软件，而真正的身份应该是一款解码软件，而解什么码，那就看需要了，如果编码规则的压缩规则，那么就可以去解压缩，仅此而已。

好了，我们这文件扩展名就为 uue ，那么编程规则呢？

开始： begin 后面是一个数字，什么功能，牛刀也不明白，大家随便写一些吧，然后后面是文件名，这就是开始规则

结束——以符号“`”结束，就是“Esc”下面那一个符号，然后后面是 end，那么好，我们就先去做这一部分——

```
#include <stdio.h>
void main()
{
        FILE *fp=fopen("MyTest.uue","w");

        fwrite("begin 123 测试.txt\n",19,1,fp);




        fwrite("`\n",2,1,fp);
        fwrite("end\n",3,1,fp);
        fclose(fp);
}
```

上面的程序就可以很好地在你的磁盘中出现一个 MyTest.uue 文件喽，大家有兴趣可以去用解码软件打开，看看其中就有一个文件，是吧？

那么我们上面一大段是干什么用的呢？那部分就是我们将要在其中写入我们需要写入的内容，好，下面我们就讲一讲——编码规则

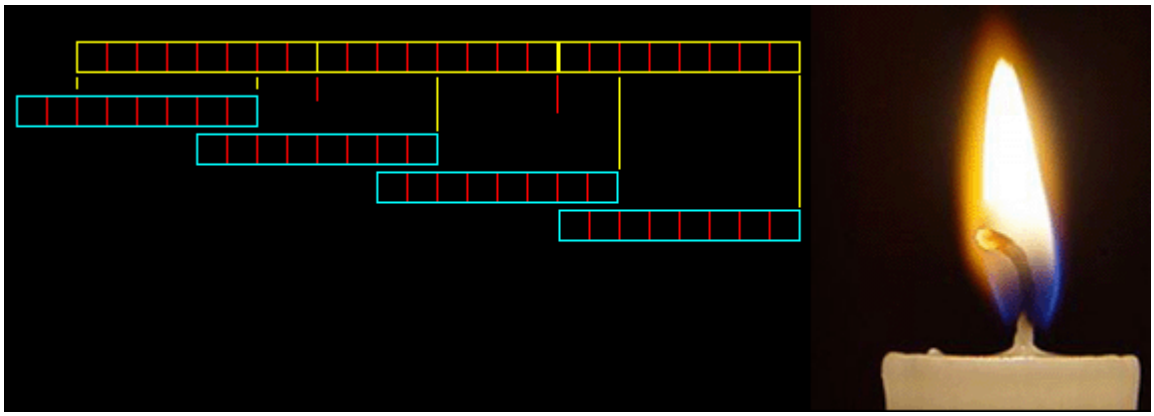
将你的文件，四十五个一组，四十五个一组分若干组，不足 4 5 个的，用 0 补足为 3 个字节的倍数，并且统计个数，并且将个数+32　作为一行的行首（注：一个字节），然后后面三位一组三位一组分开，按以下规则进行切割（好恐怖哟）——

由于 3 个字节，即 24 位，按 6 位一组，6 位一组，可以分为四组，然后每一组由于是 6 位（即可以表示 2^6=64 个字符，用以下 64 个字符重新表示——

`!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_

于是，就形成了新的编码，位操作，显然是最佳的解决方案，那么我们做一个函数去做做看——

大家看下面的图（注：这一讲开始，我们重新使用图片喽，黑底）



```
void MakeUUe(unsigned char *in,unsigned char *out)
{
                                                    unsigned          char
ch[]="`!\"#$%&'()*+,-./0123456789:\;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_";
        //看图一，有时候我们做程序还就得看图说话
        out[0]=(in[0]>>2) & 0x3f;          //0x3f 是什么大家能明白吧？屏
蔽字，不懂可以发贴问牛刀
        out[1]=((in[0]<<4) & 0x30) | ((in[1]>>4) & 0x0f);          //假
如你不能够确定运算符的优先级，
        out[2]=((in[1]<<2) & 0x3c) | ((in[2]>>6) & 0x03);          //那
么最好的办法是——加括号
        out[3]=in[2] & 0x3f;
        for(int i=0;i<4;i++)
        {
                out[i]=ch[out[i]];
        }
}
```

这不就好了？这个函数，那么我们下面就是去调用喽，好了，我们先写了三个数据　"abc"因为是三个数据，所以 3+32=35，那好，我们看程序——

```
void main()
{
```

```
        FILE *fp=fopen("MyTest.uue","w");
        unsigned char chin[3],chout[4];
        fwrite("begin 123 测试.txt\n",19,1,fp);

        chin[0]='a';chin[1]='b';chin[2]='c';
        chout[0]=35;
        fwrite(chout,1,1,fp);
        MakeUUe(chin,chout);
        fwrite(chout,4,1,fp);

        fwrite("\n`\n",3,1,fp);
        fwrite("end\n",4,1,fp);
        fclose(fp);
}
```

这样可好，我们的程序就可以顺利地将 abc 三个数据写进我们的文本了，大家如果计算机中有 WinRar 软件（一般电脑估计都有），直接打开就可以得到结果了，那么好，如果是一段文字呢？那么有几个问题需要注意——

·一、分段
·二、计算字符个数
·三、三个一组地编码。

那么好，我们来做一套，先看看我们怎么将一段文字做进来吧——

```
void main()
{
        FILE *fp=fopen("MyTest.uue","w");
        unsigned char chout[62];
        unsigned char ch[]="牛刀教你做编码软件——《周边编程一一一：文件读写》";
        unsigned char *pchout=chout;
        unsigned char *pch=ch;
        pchout++;

        fwrite("begin 123 测试.txt\n",19,1,fp);


        int i=0;
        while(!i ||( pch[0] && pch[-1] && pch[-2]))        //这儿的程序怪怪吧，大家思考为什么可以这样做
        {//注：参考牛刀以前的一些教程
                MakeUUe(pch,pchout);
                i+=3;
                pch+=3;
                pchout+=4;
```

```
                if(45==i)
                {
                        chout[0]=32+i;
                        chout[i*4/3+1]='\n';
                        chout[i*4/3+2]='\0';
                        fwrite(chout,62,1,fp);
                        pchout=chout;
                        pchout++;
                        i=0;
                }
        }
        int j=i;
        if(0==pch[0])j=i;
        if(0==pch[-1])j=i-1;
        if(0==pch[-2])j=i-2;
        chout[0]=j+32;
        chout[i*4/3]='\n';
        chout[i*4/3+1]='\0';
        fwrite(chout,i*4/3+2,1,fp);


        fwrite("\n`\n",3,1,fp);
        fwrite("end\n",4,1,fp);
        fclose(fp);
}
```

当然，更好的办法就是，如果我们能够将一个文件中的内容都写进来那该多好？好吧，那我们只要将程序改一改即可——

```
void main()
{
        FILE *fp=fopen("MyTest.uue","w"),*fpr;
        unsigned char chout[62];
        unsigned char *ch;
        unsigned char *pchout=chout;
        unsigned char *pch;
        pchout++;
        int len;
        fpr=fopen("测试.txt","r");
        if(NULL==fpr)
        {
                printf("对不起，文件没有找到，退出程序\n");
                return;
        }
        fseek(fpr,0,SEEK_END);
        len=ftell(fpr);
```

```
        fseek(fpr,0,SEEK_SET);
        ch=new unsigned char[len+1];
        fread(ch,len,1,fpr);
        ch[len]=0;
        pch=ch;
        fclose(fpr);
        fwrite("begin 123 测试.txt\n",19,1,fp);


        int i=0;
        while(!i ||( pch[0] && pch[-1] && pch[-2]))
        {
                MakeUUe(pch,pchout);
                i+=3;
                pch+=3;
                pchout+=4;
                if(45==i)
                {
                        chout[0]=32+i;
                        chout[i*4/3+1]='\n';
                        chout[i*4/3+2]='\0';
                        fwrite(chout,62,1,fp);
                        pchout=chout;
                        pchout++;
                        i=0;
                }
        }
        int j=i;
        if(0==pch[0])j=i;
        if(0==pch[-1])j=i-1;
        if(0==pch[-2])j=i-2;
        chout[0]=j+32;
        chout[i*4/3]='\n';
        chout[i*4/3+1]='\0';
        fwrite(chout,i*4/3+2,1,fp);


        fwrite("\n`\n",3,1,fp);
        fwrite("end\n",4,1,fp);
        fclose(fp);
}
```

是不是考虑到一个问题了，是否能够将文件名通过参数的方法传递进来，然后我们就可以用我们制作的这个小软件去随意地做一个 UUe 编码程序了。

当然，那种东东，你也可以去试验，牛刀在这里就不给大家做整个程序了，做得好就是一款很不错的软件啊。

大家可以用自己做出来的软件将任意的一个文件去做编码，一样的，效果应该很是不错。

显然如果上面的代码你如果都能顺利地读懂了，那么你就对使用库函数去操作文件有了很好的了解。

既然了解了，那么下一步的目的就是对其进行“封装”，系统当然给我们做好了一个土封装好的类 CFile 类

我们这一讲先与大家开一个头，当然喽，建一个 MFC 程序吧，我们又回到了微软的积木上来喽。

```
CFileView::CFileView()
{
        CFile file;
        file.Open("测试.txt",CFile::modeCreate | CFile::modeWrite);
        //  既然是类了，那么好多类中的东东就可以用了
        CString str("我用 CFile 试一试");
        file.Write(str,str.GetLength());
        file.Close();
}
```

封装过的东东就是很厉害，我们不免想去看看这个类中的一些函数——

```
Seek
SeekToBegin
SeekToEnd
GetLength
SetLength
```

厉害吧，我们原来的什么移动指针啊，到人家这就一个函数就替代了，很方便的哟。

关于这个类，我们几乎上机试试就能够基本掌握， MFC 中还有一个更厉害的文件操作类——CArchive

这个类将这们的文件读写方法变成了一个数据流，并且有超强的管理功能，下面我们就来看一看：

CArchive 的定义是以 CFile 类为基础的，在初始化时，必须使用 CFile 将 CArchive 与一个文件相关联，然后我们就可以直接使用了，看一段代码——

```
CFileView::CFileView()
{
        CFile file;
        file.Open("测试.txt",CFile::modeCreate | CFile::modeWrite);
```

```
        CArchive ar(&file,CArchive::store);
        int i=123;
        char ch='D';
        float f=3.1415926f;
        CString str("http://0ginr.com/bbs/");
        double d=3.333;
        ar<<i<<ch<<f<<str<<d;
        ar.Close();
        file.Close();
}
```

怎么样？感觉很新鲜吧？那么我们再用程序将其读出来瞧瞧：

```
CFileView::CFileView()
{
        CFile file;
        file.Open("测试.txt",CFile::modeRead);
        CArchive ar(&file,CArchive::load);
        int i;
        char ch;
        float f;
        CString str;
        double d;
        ar>>i>>ch>>f>>str>>d;
        ar.Close();
        file.Close();
        CString st;
        st.Format("i=%d,ch=%c,f=%f,str=%s,d=%f",i,ch,f,str,d);
        MessageBox(st);
}
```

这就很舒服了，舒服的牛刀几乎用不着过多地解释，可怪事就是，为什么我们的 str 可以直接读出来？是不是这种数据流的存储过程中被做了什么手脚？为了验证一下下，牛刀做了一些特别的程序：

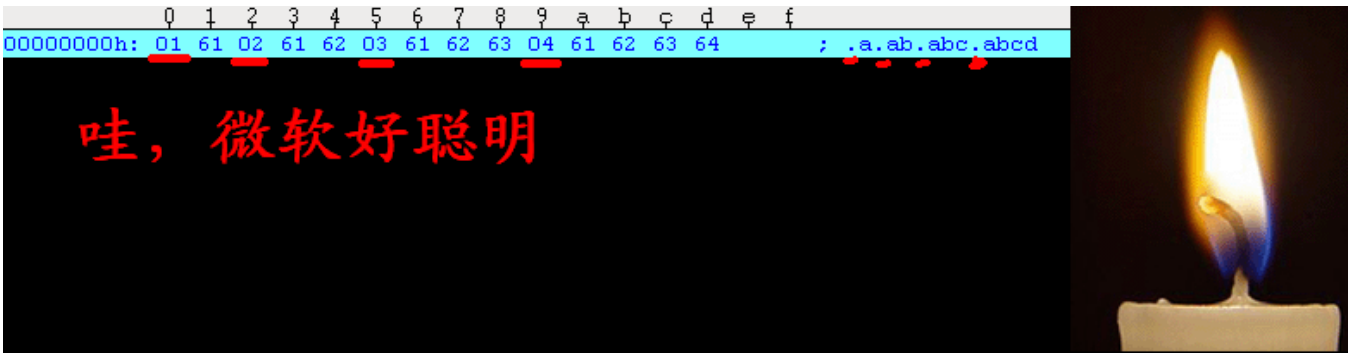
```
CFileView::CFileView()
{
        CFile file;
        file.Open("测试.txt",CFile::modeCreate | CFile::modeWrite);
        CArchive ar(&file,CArchive::store);

        CString str1("a");
        CString str2("ab");
        CString str3("abc");
        CString str4("abcd");

        ar<<str1<<str2<<str3<<str4;
```

```
        ar.Close();
        file.Close();
}
```

我们到目录下将我们写入的数据打开——

┌a┐ab└abc┘abcd

怪怪的是：每一个数据前面都多了一些怪怪的东东，肯定是类给我们额外加的一些东东，好，那么我们用一些软件将数据弄出来瞅瞅——




哇，微软好聪明，微软正是用这种方法，可以将类也流式化，能够用磁盘给保存起来，以后再用到相同的类是，使用同样的技术将类从文件中恢复出来，这就是微软大名鼎鼎的——文件流技术。

啪!!!（我们的《周边编程》系列到此告一段落，下一讲开始，我们继续《微软积木》，当然可能还需要大家等待一段时间，牛刀做做准备，大家再会汶川的死难者们做做祈祷，希望大家安好、牛刀安好、希望 0Ginr 论坛永远伴随大家一路走好）

**微软积木一一二：文档开篇**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年六月二十九日
*/

Doc　我们一直避免触及，其实说实话，也真的比较难以一下子用很简洁的语言表述清。

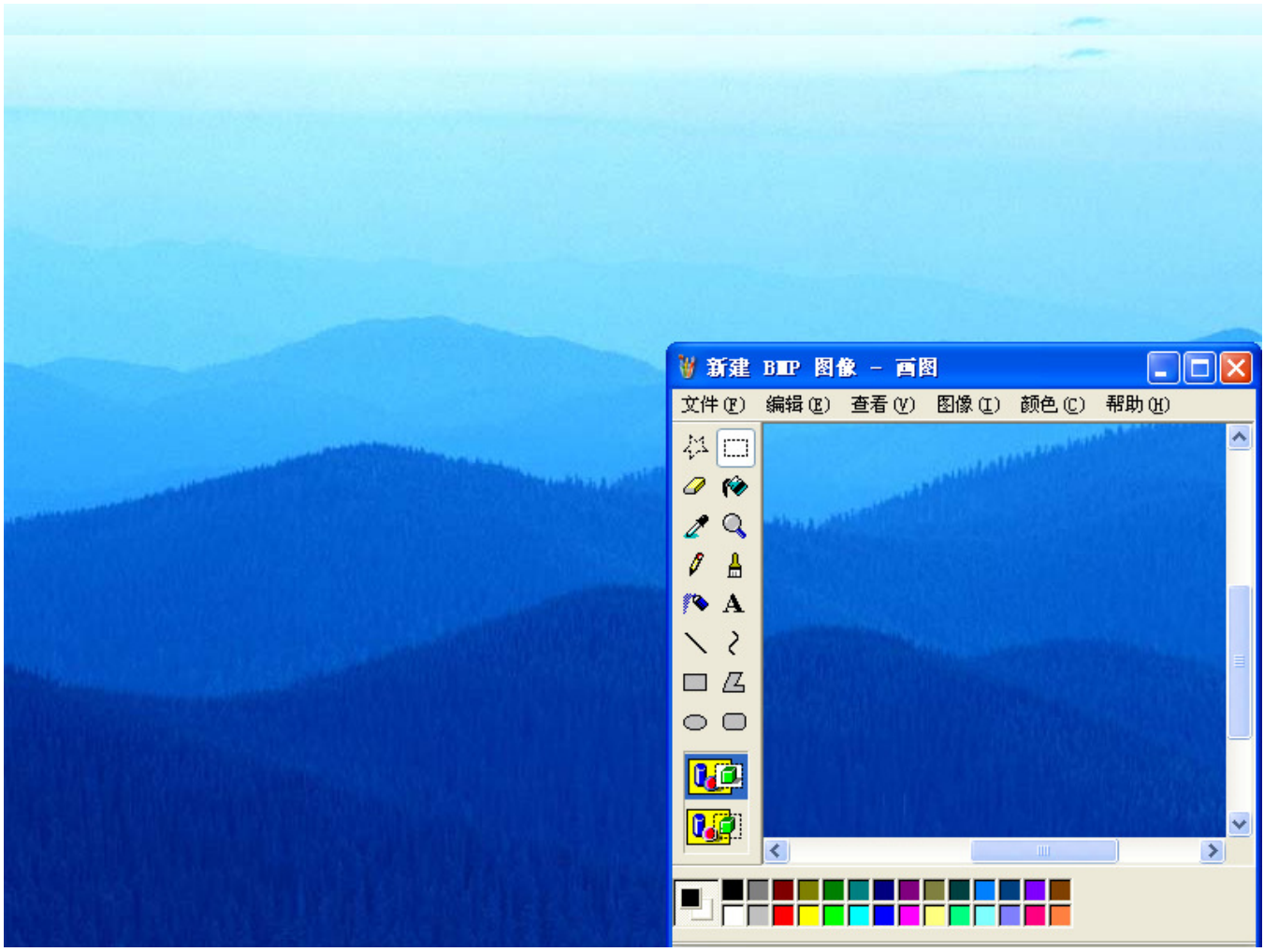
　　这下子好了，我们有了上面几讲的《周边编程》的基础，我们可以来讨论与触及这个问题了。

　　我们所讲的这种结构，通常被称之为　文档——视结构。

　　这没办法，因为我们的电脑显示器是一个有限的方块，而人们却想在其中做无限的事情，虽然有些不太现实，但人们做到了，这就象祖国的大好河山，不可能装在一部相机中，但相机中毕竟也装了一部分，当然我们想装进另一部分，只需要移动我们的相机即可。

　　于是，电脑中的数据就好像我们的大好河山，而我们的显示器就像一架相机，而我们正是通过这部相机，去观察着数据的每一个细节。

　　而　View　中就封装了视类的对象，　Doc　中则封装着我们的数据，并且封装地特别带劲，以至于我们想弄明白个子丑寅卯都不太容易，既然不太容易，也就预视着我们这个问题不是一讲两讲能够说得清楚，先看一幅图吧——




大家看，一幅图片，其实　那么大，每我们右下角的那款软件中，就只能显示其中的一小部分，那么这幅图片，我们就称之为——文档，在软件中显示的这一部分，我们称之为——视，而我们看到的这一部分，我们称之为——视口。

当然，那都是一些名词喽，我们没有必要对这些名词去斤斤计较，我们去假假地做一个软件吧，当然也是画图，当然喽，我们没必要去做一个完整的画图软件，因为如果是那样的话，我们就需要深入好多的细节喽，并且绝大多数的细节都是一样的，没有必要弄得那么玄大家说是？

那么我们怎么画呢？我们就画两种东东，左键画圆，左键画方，你能吗？呵呵，别上当，不是叫你同时画，好，我们先建立一个　MFC　工程，用单文档，名称就叫　Draw　吧，当然，最后一步，也就是第六步，大家选择　CDrawView　下面的基类选择　CScrollView，然后按“完成”，生成基本的框架。

又来了，呵呵，四个类——
CDrawApp

CDrawDoc
CDrawView
CMainFrame

当然还有一个对话框类—— CAboutDlg

那么这一讲开始，我们专门讨论 CDrawDoc 和 CDrawView 两个类。

当然，这一讲是不可能讲到 CDrawDoc 的，大家应该有所准备，我们先将软件搞定，想必这种软件大家一定很不费事就可以拿下，只需要映射鼠标的左键右键的按下和抬起消息就一切 OK ，下面的程序就当是为大家做复习吧——

```
CPoint m_ptBeg;          //保存起点坐标
CPoint m_ptEnd; //保存终点坐标
BOOL m_bldown ; //左键是否按下
BOOL m_brdown ; //右键是否按下
```

这里我们使用了两个坐标，一个起点，一个终点，并且使用了两个布尔量，表示是否左键或右键按下，那么我们定义了，就必须为其赋初值，这是必须做的，那么这种事情当然在构造函数中干喽。

```
CDrawView::CDrawView()
{
        m_ptBeg=0;
        m_ptEnd=0;
        m_bldown=FALSE;
        m_brdown=FALSE;
}
```

好，下面我们就将对我们的一些鼠标消息做响应

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_ptBeg=m_ptEnd=point;
        m_bldown=TRUE;
        CScrollView::OnLButtonDown(nFlags, point);
}

void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
        m_ptEnd=point;
        CClientDC dc(this);
        dc.SelectStockObject(NULL_BRUSH);
        dc.Rectangle(CRect(m_ptBeg,m_ptEnd));
        m_bldown=FALSE;
        CScrollView::OnLButtonUp(nFlags, point);
```

```
}

void CDrawView::OnRButtonDown(UINT nFlags, CPoint point)
{
        m_ptBeg=m_ptEnd=point;
        m_brdown=TRUE;
        CScrollView::OnRButtonDown(nFlags, point);
}

void CDrawView::OnRButtonUp(UINT nFlags, CPoint point)
{
        m_ptEnd=point;
        CClientDC dc(this);
        dc.SelectStockObject(NULL_BRUSH);
        dc.Ellipse(CRect(m_ptBeg,m_ptEnd));
        m_brdown=FALSE;
        CScrollView::OnRButtonUp(nFlags, point);
}
```

上面这段程序已经可以完美地运行喽，聪明的你可能发现，怎么定义的两个布尔变量根本没有使用嘛，呵呵，只能说你很聪明，牛刀服了，牛刀想用这两个布尔变量来实现一个很好玩的构想——橡筋线，也就是说这线像橡皮筋似得，很有弹性，当然喽，我们这儿不是画橡筋线了，而是橡盘矩形了，不过同样地好玩。

我们先说说我们的思路，一种思路是，在每画一次时，将原先的线条擦除，另一思路是，画新线前，将整个背景擦除，其实思路都很好，那我们用第一条思路。

当然这一条思路需要使用到我们的异或运算，什么，什么叫异或不懂了吗？

就是两个相同结果为 0，两个不同结果为 1　啊，我们一般用　XOR 表示，而我们这当然不是直接用这种运算，直接用多落后？我们用函数啊，有一款专用的函数叫，叫什么来着的？牛刀去做程序时，给大家注释好喽，看代码——（当然需要映射一个鼠标移动的消息）

```
void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        //注意喽，下面要设置模式喽——
        dc.SetROP2(R2_NOTXORPEN);
        //简单吧？一个函数即可，下面画图
        dc.Rectangle(CRect(m_ptBeg,m_ptEnd));
        dc.Rectangle(CRect(m_ptBeg,point));
        m_ptEnd=point;
        CScrollView::OnMouseMove(nFlags, point);
}
```

慢着慢着，怎么那么别扭啊？啊，怎么个别扭啊，你去将代码敲进电脑你就知道怎么个别扭了，牛刀就是不说，嘻嘻。

原来我们没做任何的判断啊，当然别扭喽，重新看代码，下面我们将注释去掉，看大家读代码的能力——

```
void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
        CClientDC dc(this);
        dc.SetROP2(R2_NOTXORPEN);
        if(m_bldown)
        {
                dc.Rectangle(CRect(m_ptBeg,m_ptEnd));
                dc.Rectangle(CRect(m_ptBeg,point));
        }
        if(m_brdown)
        {
                dc.Ellipse(CRect(m_ptBeg,m_ptEnd));
                dc.Ellipse(CRect(m_ptBeg,point));
        }
                m_ptEnd=point;
        CScrollView::OnMouseMove(nFlags, point);
}
```

这个效果，这个感觉啊，一个字——好，你说呢？

当然我们这里黑框框，黑圈圈，如果是彩色的，那就一团糟喽，当然如果真的是彩色的，那就必须用第二套方案喽。

第二套方案，大家自己去摸索摸索，牛刀至少暂时在这儿是不讲的，对了，由于牛刀这套教材的实践性特强，牛刀就不给大家贴图了，尽量少贴，牛刀发现这个论坛现在贴图太多打开得慢，而如果你真的按牛刀上面代码一个一个地敲进电脑，你很快就会看到效果的。

有一点遗憾，那就是：一旦我们的窗口有所变化，或者隐藏过，那么我们在窗口上所作的一切都会前功尽弃，这当然是我们不愿意看到的，于是，我们有了一个想法，就是将这些图形能够在我们画图的同时，能够保存下来，以便以后的重绘。

既然提到保存数据，那么问题就多了，怎么保存？如何保存？（好像一样啊）保存在哪里？（磁盘里还是内存中）怎么读取？如何读取？读取出来怎么处理？等等等等一系列要考虑的问题。

当然，如果你深入地接触过 MFC 一定会知道什么叫做——流式化？但我们假设你还不明白这些，那么你如何地去将一个数据保存下来呢？

当然有两套方案——数组与链表。

那么好，既然讲到这数组与链表了，我们不免来深入一下，如果接触过牛刀以前的教程或者学得相当棒，那么应该知道，我们如果需要保存三个不同类型的数据，那么就需要使用一种数据结构，被我们称之为——结构体，当然免不了去定义结构体，那么好，我们在 CDrawView 的类定义前面去定义一个结构体好了——

```
typedef struct DRAWSTRUCT
{
        BOOL bDrawType;         //我们表示我们需要画圆还是画方，假设我们真表示画圆，否则表示画方
        CPoint ptBeg;
        CPoint ptEnd;
} DrawStruct;

DrawStruct ds[100];
```

好，这就是定义了一个数组了，当然，这里我们是用的全局的，我们还需要定义一个全局的变量，来表示已经画入多少个元素了，那么好，我们也定义一个全局的变量（注：这里讲解原理，原则上不建议大家定义过多的全局变量）。

```
int nCount;
```

我们这是全局变量，我们在构造函数中将其初始化为 0，（什么？全局变量也能在构造函数中初始化？）

那么我们知道，圆上面的布尔值为真，否则为方，那么我们就可以在鼠标的抬起函数中将我们的数据写入结构体了，看程序：

```
        //画方的部分：
        ds[nCount].bDrawType=FALSE;
        ds[nCount].ptBeg=m_ptBeg;
        ds[nCount].ptEnd=m_ptEnd;
        if(nCount<99)nCount++;



        //画圆的部分：
        ds[nCount].bDrawType=TRUE;
        ds[nCount].ptBeg=m_ptBeg;
        ds[nCount].ptEnd=m_ptEnd;
        if(nCount<99)nCount++;
```

好，接下来，我们在 OnDraw 函数中去重画，看程序：

```
void CDrawView::OnDraw(CDC* pDC)
{
        CDrawDoc* pDoc = GetDocument();
```

```
        ASSERT_VALID(pDoc);

        for(int i=0;i<nCount;i++)
        {
                if(ds[i].bDrawType)
                {
                        pDC->Ellipse(CRect(ds[i].ptBeg,ds[i].ptEnd));
                }
                else
                {
                        pDC->Rectangle(CRect(ds[i].ptBeg,ds[i].ptEnd)
);
                }
        }
}
```

为了显示出区别，牛刀在 OnDraw 函数中，没有使用空画刷，大家看，如果画过之后，再次进入到 OnDraw 中，我们的确是重画了，并且保存了下来，如果你想使得程序更完美，你就可以再过设一个空画刷，简单吧？不过如果你用过空画刷后，你的程序运行起来就会显得天衣无缝喽。

以上就是数组方案，还有一种方案，我们称之为链表方案，不过这次我们设置链表时，我们就直接使用 CRect 结构作为链表中存储数据的部分喽，那么什么叫做链表？就是一个包含自己的指针，仅此而已，这次我们定义成成员变量。

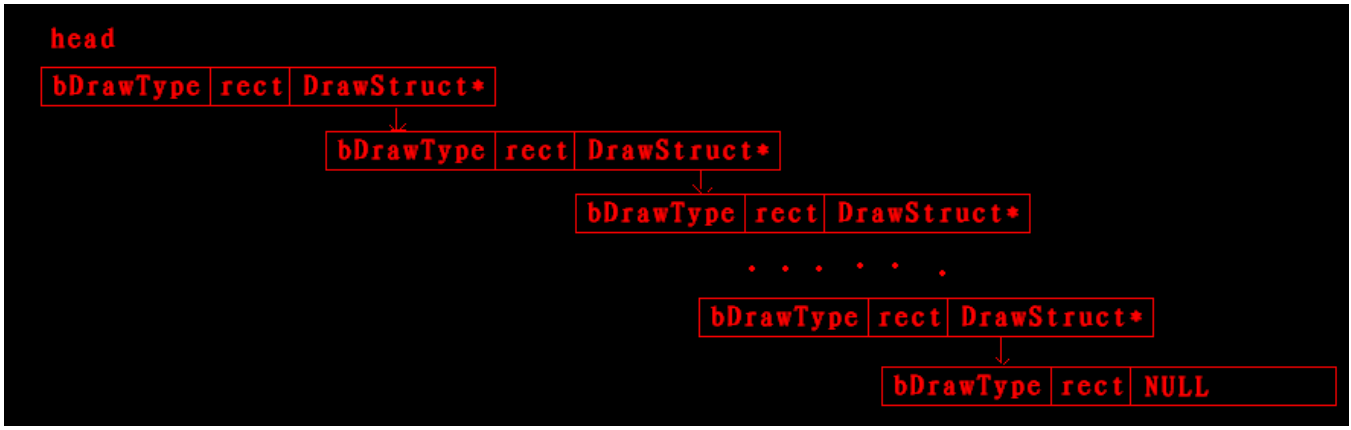
对了，说一下啊，上面数组的程序如果有 99 号的数据，是重画不到的，大家可以去完善一下，再一个缺点就是，我们只能画一百个图形，多于一百个就无能为力了，这就是数组的缺点。好了，我们这就去做程序：

```
struct DrawStruct;
typedef struct DrawStruct
{
        BOOL bDrawType;
        CRect rect;
        struct DrawStruct *next;        //请注意，这是一个指针
}DrawStru;
class CDrawView : public CScrollView
{
        DrawStru *m_head;
```

当然喽，这次我们也还需要在构造函数中对其进行初始化：

```
m_head=NULL:
```

那么什么叫链表呢？看图：




看，就像图上那样，用链串起来的数据，我们称之为链表，牛刀在《中国算盘》那一篇文章中与大家曾经说过，链表、队列、栈等数据结构其实是一种数据结构，牛刀这次就将这链表作为栈来用，也就是说，一切的事情，都要在链表头操作请大家看代码——

```
//画方部分:(首先申定义一个指针 p 好像这样定义是编程人的习惯吧，反正牛刀都是这样干的)
DrawStru *p=new DrawStru;          //多方便，定义时就新建了一个，那么下面就对其赋值吧，我们这画方：
p->bDrawType=FALSE;
p->rect=CRect(m_ptBeg,m_ptEnd);          //请注意，这里结构中的 rect  可是  MFC  类哟，可以直接赋值嘀
 p->next=m_head;          //这样一来，我们的下一个就指向了 m_head  头部，头部就可以放心地放手喽，看下面
m_head=p;          //这样就将数据插入到了头部，下面画圆部分相同。



//画圆部分：
DrawStru *p=new DrawStru;
p->bDrawType=TRUE;
p->rect=CRect(m_ptBeg,m_ptEnd);
p->next=m_head;
m_head=p;
```

好了，这下我们就可以顺利地到我们的  OnDraw  函数中去使用这一款链表了——

```
pDC->SelectStockObject(NULL_BRUSH);          //不用这一句你就看出滑子来了，大家可以先将这一句去了，看看什么叫做贱（噢，不是这个贱，是那个栈啊，哪挨哪儿啊，呵呵）
DrawStru *p=m_head;          //多方便，一开始就指过去：
while(p)          //循环不一定就是 for 啊，呵呵
{
```

```
            if(p->bDrawType)
            {
                    pDC->Ellipse(p->rect);          //瞧,这多方便?
            }
            else
            {
                    pDC->Rectangle(p->rect);
            }
            p=p->next;          //这一句挺关键，没有就进入了死循环
    }
```

行了，有关图形的保存与重绘，我们下讲继续吧——啪（又来了)!!!

**微软积木一一三：图形重绘**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年六月三十日
*/

上一讲，我们使用了种方法，将图形从鼠标抬起操作中移到了 OnDraw 函数中，从而使得我们的图形能够一直持久在保存在屏幕上了，有 VB 操作经验的朋友可能不屑一顾，不就是在 Picture 控件上设置一个属性的事情吗？费了那么大的劲。

从这一点上来说，牛刀也不得不服气，人家的确做得好就得承认，你说是吧，不过牛刀还是觉得 VC 相对而言比较自由。

上一讲我们讲了两个结构——数组与链表，这两个结构体各有千秋，数组以其不可比拟的速度优势占着上风，可链表结构又以其得天独厚的长度随意性而别具一格，那么我们到底用哪种好呢？ MFC 为我们做好了这两种结构操作的所有模板，当然还为我们的操作做了另一种结构——集合。

集合是什么呢？上过高中的朋友一定不陌生，但我们这里的集合与高中学的集合有一些区别。

当然，在集合的使用上，我们可以将其看作是链表，或者是数组，都没问题，看作是链表吧，集合有速度上的优势，看作数组吧，集合又不需要为其定义一个数据上限，我们使用起来感觉好像是数组，但却是一个可以无限大的数组，这就是集合。

当然喽，牛刀一般喜欢用程序去模拟一些这些数据结构，不过这次咱们不给大家模拟了，大家可以自己去试着模拟一下看一看，牛刀曾经在前面的《周边编程》部分给大家提及过，但那里用的类型就比较单一了，而 MFC 为我们做的库中有两种方案——

·一种方案是使用一些无类型的指针，这样一来，我们的所有数据，不管是结构也好普通类型也好，只需要将这些指针强制转换成无类型的指针型数据，就可以直接放到集合中去了（以前牛刀曾经与大家说过，所有的指针型数据的指针都是卅二位的，所以我们可以随便，以及下面的类，都可以用集合去放（对了，说到了类，我们下面用类来保存数据吧，但用结构也是一样，大家可以自己去试）
·另一种方法就是使用模板，将所有的类型放到模板函数中去，不管你用什么办法，去定义什么类型，只需要用一个像宏一样的模板参数，就可以将自己定义的任何类型的数据全都放在我们的集合。

下面我们使用第一种方式，当然，这次我们定义一个类 CDraw，插入一个

就行了，不需要用 MFC 的类啊，一个基本的类就行了，下面我们就去里面定义一些成员变量——

```
class CDraw
{
        BOOL m_bDrawType;
        CRect m_rect;
public:
        CDraw();
        virtual ~CDraw();

};
```

当然，需要在构造函数中对其进行初始化，那么初始化为什么呢？为了实现构造初始化，我们再定义一个重载的有两个变量的构造函数，看代码——

```
CDraw::CDraw(BOOL bDrawType, CPoint ptBeg, CPoint ptEnd)
{
        m_bDrawType=bDrawType;
        m_rect=CRect(ptBeg,ptEnd);
}
```

注意了，上面的那个默认的无参数的代码我们也不删，下面有用。

对了，乘机也定义一个 OnDraw 函数吧，咱们将那一部分代码拿到这里来，（注意了，牛刀所说的 C++ 语言的自由也体现在这些地方，的确很自由，很舒服，基本上是想怎么干就怎么干，想怎么定义就怎么样定义）

```
void CDraw::OnDraw(CDC *pDC)
{
        pDC->SelectStockObject(NULL_BRUSH);
        if(m_bDrawType)
        {
                pDC->Ellipse(m_rect);
        }
        else
        {
                pDC->Rectangle(m_rect);
        }
}
```

这种方便自不用说了，咱们下面就在 View 文件中去包含进我们新定义的类的头文件——

```
#include "Draw1.h"
```

（注：请大家注意，我们的 CDraw 文件的头文件如果按传统的命名方案 Draw.h 那么就与我们的 CDrawApp 文件的头文件重复了，所以系统自动为我们重新定义了一个头文件的文件名，为 Draw1.h ，当然，相应的 CPP 文件就为 Draw1.cpp 喽，大家注意了就是，在 MFC 中的 C***App 和 CMainFrame 两个文件的头文件名称用的的简写，大家去看看就明白喽，偷懒嘛，电脑自动偷懒，呵呵，所以我们以后再定义相关类时注意尽量不要与这些文

件名重复，牛刀在上面定义这个类时没有考虑到这个问题，大家以后注意就行了。）

然后我们就可以去使用我们的新类了。慢着，我们到底放在哪里呢？放在集合里面啊，我们使用一个 MFC 的无符号型集合，为 CPtrArray ，先去定义一个成员变量：

CPtrArray m_ptrarray;

然后我们去在我们的鼠标的左键与右键的抬起事件中，去定义这个指针就行了，传参就行了，甚至用不着去判断是圆是方喽，方便啊——

```
//画方部分：
CClientDC dc(this);
CDraw *pdraw=new CDraw(FALSE,m_ptBeg,m_ptEnd);
pdraw->OnDraw(&dc);          //方便啊
m_ptrarray.Add(pdraw);          //看出好的感觉来了吗？
m_bldown=FALSE;          //不是为了鼠标的移动消息部分，牛刀这部分都想扔了，呵呵。
```

```
//画圆部分：
CClientDC dc(this);
CDraw *pdraw=new CDraw(TRUE,m_ptBeg,m_ptEnd);
pdraw->OnDraw(&dc);
m_ptrarray.Add(pdraw);
m_brdown=FALSE;
```

好了，这下我们去试试结果——

当然，在牛刀的感觉中还是比较完美的，除非，除非你在使用时，不小收打开其他软件了，于是我们的程序重画了，我们的东东就没有了，于是，必须响应 OnDraw

```
int nCount=m_ptrarray.GetSize();
for(int i=0;i<nCount;i++)
{
        ((CDraw*)m_ptrarray.GetAt(i))->OnDraw(pDC);//像是数组吧？但用的不是 [] 而是 ()，这很像 VB 中的数组，舒服啊
}
```

你看，在 CDraw 中定义了一个 OnDraw ，我们的所有绘制工作就都移去过了，而使得我们在这里的编程就显得格外地简单，这就是我们“封装”的基本思想。

说明一下，因为我们的 CPtrArray 这种集合类使用的是无符号指针 LPVOID 也就是 void * 类型吧，所以我们在使用后，应该使用相应的强制类型转换（像上面的 CDraw* ），将其转换为我们需要使用的指针类型，这样我

们就可以拿出来顺利地使用了。

牛刀发一些感慨啊，呵呵。

像这种指针之间的相互通用的现象，在其他的语言中实现起来真的是很难，难就难在其他的语言没有指针这样的一个类型，从而使得我们做程序能够做得这样完美，可以想像，没有指针的日子怎么过（当然，人家 VB DELPHI 还不是一样地过来了，呵呵）。

VC 完美之处就是其使用了统一长度大小的 32 位的指针变量，从而使得我们的数据只需要按照某个方式存进来，就可以相应地按照某种方式读出来。

但反过来说，也正因为这种完美，使得我们如果想做 VC 去干坏事，那真是——手到擒来，创建一个控制台应用程序，咱们用这种手段去干一些事（当然不是见不得人的勾当，但也请看牛刀教程的朋友们也是同样怀着一颗友善的心。）

```
#include <stdio.h>

void main()
{
        char *ch="abcd";
        int *a=(int*)ch;
        printf("%x\n",*a);
}
```

请大家看这种数字的转换，如果不明白指针的道理，简单就会觉得不可思议，a 的 ASCII 码的值可不就是 0x61 嘛？可是结果却有些出乎我们的意料——

64636261

为什么结果是颠倒的呢？因为在我们的卅二位的机器中，数字是逆着存放的，如果看过牛刀的《汇编实践》，看这个问题那就是小菜一碟喽，那好，我们反过来试试：

```
#include <stdio.h>

void main()
{
        int i=0x31323334;
        char *ch=(char *)&i;
        printf("%s\n",ch);
}
```

大家看结果：

4321?↕

后面多了两个，当然如果你在电脑上运行，最后一个不是我们复制上来看到的那样的，大家自己去试试，为什么多了两个数据呢？仔细体会什么叫做——字符串，再次体会我们的整型数据的低位优先存放的规则。

要说明的是，我们的电脑不是全部都是高位存高低地址的，也有些电脑上不

是这样的，于是就出现了两个问题，一个是文件兼容问题，如果那些电脑上的文件拿到了我们的电脑上，能不能使用了呢？ MIDI 文件就是一款相当优秀的例子，可以说，我们现在在电脑上播放 MIDI 音乐用的全是类似于模拟器的技术(什么叫模拟器相信大家应该不陌生吧？玩这么长时间的电脑谁没玩过游戏啊，呵呵)。

另一个问题就是，我们的网络传输的问题，如果不是这个问题，那么我们的网络编程就会简单得多，然而也正因为这个问题，所以编程时，就存在着一个网络字节的问题，于是，我们的所有的数据，都要变成网络字节进行传输（什么原因很简单，我们的的网络传输只能一位一位地传输，到目的地址后进行安装，如果没有相应的安装方案，那么很可能我们传给别人的数据是 0x12131415 ，而别人接收到的数据则为：0x15141312 ，那么如果对方是接你的一个软件，那么这款软件就可以因为其中的一个数据的接收不正确从而引起整个软件的崩溃。（这不是耸人听闻）

打住打住，好像扯得远喽。

其实就是这样，学习不能学得满脑子浆糊，什么问题一定要去搞懂，弄明白，知其然还要知其所以然，也只有这样，我们真正做起事情来才能够游刃有余。

行了，我们从上一讲到这一讲，同样的问题，用数组、链表、集合三种办法给大家模拟了，当然，我们这里使用了一个布尔变量 BOOL 来表示我们使用了什么样的图形，如果我们要画的图形不是两种，而是很多，那么我们就不能使用布尔变量喽，当然用什么办法，想必大家也会明白。

当我们的数据保存下来，有时候我们还是觉得有所不满意，那么我们就想到了另一个问题——数据的保存喽。

我们写一篇文章，都不希望随着我们关闭我们的软件就消失，最好的办法就是我们软件再重新打开始，能够通过某种办法使得我们的文件再次打开，供我们继续使用，那最容易的办法就是将其保存在磁盘上了，当然，在《周边编程》中，我们说过了三种办法——

·FILE 文件，使用 C 语言的标准函数的办法去保存数据
·CFile 文件，使用 MFC 的类去保存与使用文件
·CArchive 文件，也就是我们以后将要介绍到的数据流的使用方案，我们当然还是从最基本的来，当然，我们先用数组与 FILE 的办法吧，链表的方案由大家去实现。

我们假设大家按键盘上的 S 按键就是需要保存数据，而按 L 按键就是需要载入数据，那么我们去映射一个 WM_CHAR 消息：

```
void CDrawView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        //这儿才好，有 nChar 可用，呵呵：
        FILE *fp;
        int i;
        switch(nChar)
```

```
        {
        case 's':
        case 'S':
                fp=fopen("我的图形保存.txt","wb");//为了便于我们观察，我们使用 txt  格式的文件
                        fwrite(&nCount,4,1,fp);
                        for(i=0;i<nCount;i++)
                        {
                       fwrite(&ds[i].bDrawType,sizeof(BOOL),1,fp);
    //这儿牛刀也不知道  BOOL  是多少位的，估计是卅二位的吧，不管了，这样做就永远不会错，呵呵
                            fwrite(&ds[i].ptBeg,sizeof(CPoint),1,fp);
                        fwrite(&ds[i].ptEnd,sizeof(CPoint),1,fp);
                        }
                fclose(fp);
                nCount=0;//这儿我们乘机将图形关了，以便再次显示时知道是重新载入的。
                break;
        case 'L':
        case 'l':
                fp=fopen("我的图形保存.txt","rb");
                fread(&nCount,4,1,fp);
                for(i=0;i<nCount;i++)
                {
                        fread(&ds[i].bDrawType,sizeof(BOOL),1,fp);
                        fread(&ds[i].ptBeg,sizeof(CPoint),1,fp);
                        fread(&ds[i].ptEnd,sizeof(CPoint),1,fp);
                }
                break;
        }
        Invalidate();
        CScrollView::OnChar(nChar, nRepCnt, nFlags);
```

这就好，大家随便画几样东东，然后按  S  没了，再按  L  ，又出来了，挺好玩吧，呵呵，下面我们去看看我们生成的文本文件——

```
└       ☼┌  j   ?  a┌      j┌     !┐  !┌  ┌   ?     ┘┐
```

乱糟糟一团糟，哪挨着哪儿啊？呵呵，可是你还别说，还真就是这么回事，我们的图形已经被保存在这里喽，不信，你将这一段数据复制到你的相应文件中去，然后按  L  ，看见没有看见牛刀画出来的图形？

（注意喽，别试喽，咱们的数据被 Windows  系统给转换过了，一些数据并不是原来的数据了，大家还记得前面我们提到的文本与二进制的区别吗？而现在的系统保存文件的时候，不是  Unicode  竟然自动转换了过去，所以我们的数据就不是原来的数据了，所以大家别试，肯定看不到牛刀的图形的，嘻嘻，自己去试试就行了）

当然，下一讲，我们就重点讨论我们的文件保存与数据流技术，期待牛刀

**微软积木一一四：文件保存**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年七月一日
*/
```

上一讲讲到了用 FILE 去保存数据，当然大家很容易想到使用 CFile 去保存，当然，大家明白吗？上一讲的那一段保存程序完全可以这样做：

```
void CDrawView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        //这儿才好，有  nChar  可用，呵呵：
        FILE *fp;
        int i;
        switch(nChar)
        {
        case 's':
        case 'S':
                fp=fopen("我的图形保存.txt","wb");//为了便于我们观察，我们使用 txt  格式的文件
                        fwrite(&nCount,4,1,fp);
                        for(i=0;i<nCount;i++)
                        {
                                fwrite(&ds[i],sizeof(DrawStruct),1,fp);
                        }
                fclose(fp);
                nCount=0;//这儿我们乘机将图形关了，以便再次显示时知道是重新载入的。
                break;
        case 'L':
        case 'l':
                fp=fopen("我的图形保存.txt","rb");
                fread(&nCount,4,1,fp);
                for(i=0;i<nCount;i++)
                {
                        fread(&ds[i],sizeof(DrawStruct),1,fp);
                }
                break;
        }
        Invalidate();
```

```
        CScrollView::OnChar(nChar, nRepCnt, nFlags);
}
```

是不是感觉要舒服多了？当然，我们这种保存不错是将整个结构作为一个整体去保存了，然而毕竟我们一次只能保存一个结构，并且整个的结构都需要我们自己去管理，那么能不能将这些结构的保存放到结构里面去完成呢？

想要将程序放到结构中去，这当然就是一个对象了，牛刀以前曾经说过，class 与 struct 定义的结构或类没有本质的区别，唯一的不同就是 class 默认的数据是私有的，而 struct 默认的成员数据是公有的，仅此而已，因此，我们还是回到我们定义的那个类上来 CDraw，我们希望的不是使用一些循环去检测 CDraw 有几个，而是想将一个文件指针传过去，让类去自己完成保存工作，当然，这个类中如果还有子类，同样将指针传递过去，从而使得我们的类中的数据像用一条线一个一个地串起来一样，我们称这种过程称之为——数据的串行化。

当然，想要串行化一个数据也不是那么容易，我们可以做一个控制台应用程序去看看其难度：

```
#include <stdio.h>

class A
{
        int a;
};

void main()
{
        printf("类 A 的大小为%d\n",sizeof(A));
}
```

牛刀的结果为 4 ，为什么牛刀要做这个问题呢？我们再来看一个程序，然后说明问题：

```
#include <stdio.h>
class B
{
        int a;
        int b;
};

class A
{
        int a;
        B b;
};

void main()
```

```
{
        printf("类 A 的大小为%d\n",sizeof(A));
}
```

这时候的结果是什么呢？如果你能够上机实践一下的话，那么结果就为——类 A 的大小为 12

这很好很好理解，那么一旦我们使用了下面的定义——

```
class A
{
        int a;
        B *b;
};
```

那么这个结果就完全变了，结果变成了 8 ，也就是说，我们如果想用 sizeof(A) 的方法去取得 A 中的数据大小，然后奖所有的 A B 数据全部保存下来是不可能的了，也就是说，我们将类或者结构当成一个没有指针变量的结构体用时，我们的程序不用出现任何毛病，而如果一旦我们的结构中出现了指针，我们即使保存，我们也只能保存下我们类 B 的指针，而这个指针是一个内存指针，而我们下次再从硬盘上想通过保存的指针将 B 恢复出来，现实吗？显然是恢复不出来的了，原因很简单，因为我们没有办法在类 A 中去恢复出 B ，因为 A 中并没有 B 的结构，而仅仅有一个指向 B 的指针，我们怎么去保存 B？

当然，如果这个结构是我们自己去做的，那么我们就已经知道了 B 是什么，我们只需要将 A 中的一些参数保存下来后，再根据一些参数，将 B 再保存一下，然后还需要指明 B 与 A 的关系，总之，太复杂了，如果你有兴趣，就上面的一个指针，一个数据，你是怎么去保存这两个类的呢？作为思考题，让大家自己去思考。。

而我们的另一种办法是——这种我们做不了的事，就让类自己去做，于是，我们必须在 A 中定义一个函数，传进一个指针，让 A 自己去用文件指针保存下 B 的内容，然后再向系统放掉指针，当然，如果我们想从保存的数据中恢复出 A 和 B 那么我们就从 A 定义的这个类中恢复出 B ，当然还是用 A 中的函数，有点绕人是也不是？下面我们就去做这样的一个函数：

```
#include <stdio.h>
class B
{
        int a;
        int b;
public:
        B(int x,int y)
        {
                a=x;
                b=y;
        }
        void print()
```

```
        {
                printf("B 类中的数据 a=%d,b=%d\n",a,b);
        }
};

class A
{
        int a;
        B *b;
public:
        A(int x,int y,int z)        //我们用这个函数去创建  a  和 b
        {
                a=x;
                b=new B(y,z);
        }
        void print()
        {
                printf("A 类中的数据 a=%d\n",a);
                b->print();
        }
};

void main()
{
        A a(5,6,7);
        a.print();
}
```

好了，我们这不错是顺利地定义了一个 a 并且由 a 又定义了一个指针 b，那么你是如何打算去保存这三个数据的呢？噢，对了，并且这三个数据还很特别，是私有成员，外面是不能够访问的，那我们怎么办呢？

当然，一个办法是，将两类的所有变量设为公有的，可以倒是可以，可你如何保存 A 类与 B 类之间的关系呢？这种种的问题一股脑地交给了我们，于是，就有着另一个方法产生了，让类自己去做，行，我们下面就为其做一个保存测试：

```
#include <stdio.h>
class B
{
        int a;
        int b;
public:
        B(int x,int y)
        {
                a=x;
                b=y;
```

```
        }
        void print()
        {
                printf("B类中的数据a=%d,b=%d\n",a,b);
        }
        void Save(FILE *fp)
        {
                fwrite(&a,4,1,fp);
                fwrite(&b,4,1,fp);
        }
};

class A
{
        int a;
        B *b;
public:
        A(int x,int y,int z)        //我们用这个函数去创建  a  和 b
        {
                a=x;
                b=new B(y,z);
        }
        void print()
        {
                printf("A类中的数据a=%d\n",a);
                b->print();
        }
        void Save(FILE *fp)
        {
                fwrite(&a,4,1,fp);
                b->Save(fp);
        }
};

void main()
{
        A a(5,6,7);
        a.print();
        FILE *fp=fopen("我的文件存储测试.txt","wb");
        a.Save(fp);
        fclose(fp);
}
```

这程序够经典吧？我们使用 A 中的参数 Save 去保存了自己类中的数据 a 并且顺便又调用了 B 类中的 Save 去让 B 也去保存一下自

己的数据，这种保存的方式，我们称之为——流式化。

这样一来，我们就可以使用一个 Save 就可以顺利地将 A 与 B 类中的数据全都保存起来了，并且以后可以从存储器中将这两个类给恢复出来？

什么？类还能恢复？

对，其实是保存的对象，恢复的对象，那么我们通过流式化的方法，就可以顺利地将我们对象的生存期与软件相同变成了与我们的硬盘相同，当然喽，如果对象保存的文件被删了，那也就算了，还有一种情况就是你用锤子将硬盘给砸了，也恢复不出来了，呵呵。

好，那我们就去定义一个 Load 去恢复这两个对象。

慢着，恢复对象是有条件的，首先，我们的对象必须有一个无参数的构造函数，这是为什么呢？因为我们在恢复对象的时候，我们当然是先去定义一个变量，然后再用这变量中的 Load 函数去恢复这个变量，如果没有一个无参构造函数的话，那么我们就必须先这这个类去传参，我们又拿什么去向参数中传呢？注意喽，我们不是在 Load 之前就将对象的数据给取出来的啊。（注：当然，上面的程序，我们可以先将三个数据取出来，然后再通过有参的方法去用参数来构造这个对象，但有两个缺点——

·一：你这个软件是一成不变的吗？如果你以后需要增加一个或者几个参数，那么你就需要将程序所有的恢复对象的程序部分统统做一次更改，如果你在程序中多处使用这个对象，那么这种改法显然是不划算的，并且也是不实惠的，还容易错，告诉你一个现实，一个地方错了，那就全错了，所以从这个方面来看，这种做法不可取。
·二：你这个类不打算扩充了吗？如果 B 类中又增加了一个 C 类的指针呢？于是我们就需要到外部的代码去重新修改整个调用 B 类、恢复 B 类的代码。

于是乎，与其传参有着这么多的不确定性，那么我们还不如不传参，而参数都是在类中的相应函数中自动恢复。

另一个问题，就是，我们 A 类中，必须知道有 B 这么一个对象。

咳，这不是废话吗？怎么能不知道呢？不是自己定义的吗？

那好，不错你这是自己定义的，那如果你这部分代码是为别人做的呢？别人需要用你的 Save 去保存自己定义的类呢？于是，我们就需要让 A 知道我们新定义了哪个类，是也不是？

于是，又要有一定的手段将自己定义的那个类要告诉 A ，使得 A 能够处理我们新的类（这一点我们以后再交待，牛刀的一张嘴表不了两家的事啊，呵呵）。

我们先处理这个 Load ，当然，处理之初是为这两个类定分别定义一个无参的构造函数——看程序：

```
#include <stdio.h>
class B
{
        int a;
        int b;
public:
        B()         //大家看，我们只需要定义在这儿就行了，其中一行代码都
        {         //用不着，因为我们压根就没有打算使用这个构造函数去恢复
        }         //对象，但缺了还就不行，呵呵，真是鸡肋啊!!!!
        void Load(FILE *fp)
        {         //看好喽，我们是怎么恢复数据的，很简单的嘛，读出来就一切 OK 喽
                fread(&a,4,1,fp);
                fread(&b,4,1,fp);
        }
        B(int x,int y)
        {
                a=x;
                b=y;
        }
        void print()
        {
                printf("B 类中的数据 a=%d,b=%d\n",a,b);
        }
        void Save(FILE *fp)
        {
                fwrite(&a,4,1,fp);
                fwrite(&b,4,1,fp);
        }
};

class A
{
        int a;
        B *b;
public:
        A()
        {
        }
```

```
        Load(FILE *fp)
        {
                fread(&a,4,1,fp);
                b=new B();          //请注意喽，这里我们是作恢复  B  的
工作，大家这下知道为什么  B  中一定要使用
                                                //没有参数的构造函数的原因了
吧？下面我们就去做  B  的恢复工作，一个函数即可
                                                //当然这儿的括号没有也可以,这
儿牛刀仅仅是为了告诉你为什么要使用无参构造函数
                b->Load(fp);
        }
        A(int x,int y,int z)          //我们用这个函数去创建  a  和 b
        {
                a=x;
                b=new B(y,z);
        }
        void print()
        {
                printf("A 类中的数据 a=%d\n",a);
                b->print();
        }
        void Save(FILE *fp)
        {
                fwrite(&a,4,1,fp);
                b->Save(fp);
        }
};
void main()
{
        A a(5,6,7);
        a.print();
        FILE *fp=fopen("我的文件存储测试.txt","wb");
        a.Save(fp);
        fclose(fp);
        fp=fopen("我的文件存储测试.txt","rb");
        A b;    //注意喽，我们这儿使用的无参函数，下面就准备恢复数据喽
        b.Load(fp);
        fclose(fp);
        printf("下面是恢复后的  b  的数据,是根据以上的数据恢复的\n");
        b.print();
}
```

上面的程序挺完整，仅仅说明了一个问题，我们可以使用  A  中的  Save 去保存自己的数据与自己所定义类的数据，那么就有一个问题了，我们假如定义了另一个类  C  ，并且也要使用  A  类的  Save  进行保存，也需要使用  A

的 Load 去恢复，怎么办？如果我们在 B 中也定义了一个类 E 也需要用 A 中的 Save 去保存，也要用 A 中的 Load 去恢复，那怎么办？

当然有一种办法，就是我们去改上面的程序将父类中的相应的函数给改了，定义一个就加一个，定义一个就加一个，如果定义在 B 中就在 B 中加，如果定义在 A 中就在 A 中加。

当然，你想想，这会是微软的风格吗？牛刀前面与大伙谈那么多的《周边编程》是干什么的？这不，这就需要用到了。这儿先与大家一点点的提示，我们下一讲会详细与大家继续深入——

```
class A
{
        int a;
public:
        A();
        void Save(FILE *fp);
        void Load(FILE *fp);
        void print();
        A(int a,int b,int c);
};

A::A()
{
}

A::A(int a,int b,int c)
{
        ...
}
void A::Save(FILE *fp)
{
        ...
}

void A::Load(FILE *fp)
{
        ...
}
void A::print()
{
        ...
}
```

看明白其中的奥妙了吗？微软定义的东东不会是没有用的，瞎定义的无用的东东，一定有其特有的作用与合理的解释的，那么我们能从这些微软给我们定义好的东东中发现什么样的端倪呢？我们下一讲再见——啪!!!（怎么没人鼓掌？）

**微软积木一一五：流式文件**

/*
版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年七月二日
*/

这一讲开始需要大家阅读慢一些，仔细地去体会喽，说实话，这些写出来的东西都是牛刀自己的理解，书本上一定是不会与你说得这样地深入的，原因很简单，任何一本 MFC 的书都假定你有很深厚的 C++ 的功底了，所以那些书上介绍这些肯定都是一带而过，顶多告诉你一些新名词，例如称这种过程为流式化，称那种过程为运算符重载，或者我们将文件流式化需要加入哪些宏，需要加入哪些定义，仅此而已，我们如果看一套教你怎样将自己定义的类流式化的书籍，你就只会看到这样的一段说明——

要让用户定义的类支持串行化，一般分为五步：

1.从 CObject 或其派生类派生出用户的类

2.重载 Serialize()成员函数，加入必要的代码，用以保存对象的数据成员到 CArchive 对象以及从 CArchive 对象载入对象的数据成员状态。

3.在类声明文件中，加入 DECLARE_SERIAL 宏。编译时，编译器将扩充该宏，这是串行化对象所必需的。

4.定义一个不带参数的构造函数。

5.在实现文件中加入 IMPLEMENT_SERIAL 宏。

顶多下面再给大家举个例子，仅此而已，人家的目的算是达到了，那么其中的 CObject 是什么东东呢？ Serialize 又是什么样的函数呢？两个宏又是什么怎么呢？

上面的也就是第四点牛刀上一讲与大家说过为什么，而其他几点就需要我们这一讲慢慢与大家详细道来喽。

当然，我们这一讲还是不能讲得那么全面，还是那句话，牛刀一张嘴，表不了两家的事，何况我们的编程是思绪万千，错综复杂呢？是不是到这儿大家感觉玩微软的 MFC 不像玩积木了？

如果你有这种感觉，那么赶快去再补一补前面的 C++ 部分的知识，我们一定不能让我们的知识落后于我们的思路，还是那句话，微软的一切定义都必然有其登台的机会的。

还是先看看牛刀上一讲最后给大家做的那一套的东东——

```
class A
{
        int a;
public:
        A();
        void Save(FILE *fp);
        void Load(FILE *fp);
        void print();
        A(int a,int b,int c);
};

A::A()
{
}

A::A(int a,int b,int c)
{
        ...
}

void A::Save(FILE *fp)
{
        ...
}

void A::Load(FILE *fp)
{
        ...
}

void A::print()
{
        ...
}
```

牛刀让大家看上面的东东有什么意思，也就是说，上面的东东，我们可以拿出来，那么既然能够拿出来，并且如果我们的定义已经程式化的话，我们就可以为其定义为一个宏、一个模板甚至一个头包含，也就是说是一个库。

再一个就是，这个定义可以拿到程序的任何地方，以便能够合理地组织，合理地使用。

那么怎么能够真正地实现这种目的呢？其实现的手段又怎样呢？说实话，这一讲是肯定讲不了了，并且我们准备先抛开上面的 A 类与 B 类，去先定义两个类 TArchive 和 TObject ，怎么？牛刀想干什么啊？呵呵，咱们就是

要和微软对着干，并且干就干点名堂出来，呵呵。

当然，我们不可能也没有必要去将微软的所有过程都去实现一遍，那是没有必要的，我们要的过程是——理解：

那好，朋友们注意喽，我们坚决不用 MFC 库（仅限于下面的一部分以及以后可能的几节对台戏部分，呵呵，不唱对台戏能不用吗？牛刀说的意思是，一定不使用 MFC 库的头包含，仅此而已，当然功能是不可能很全的罢了），

好了，下面我们加入两个类，一个为 TArchive，一个为 TObject 那么好，直接插入就行了，插入后，我们加入一个有两个参数的构造函数（在 TArchive 里面）

（注：当然，我们加入一个头文件：tinclude.h 表示我们所有的头包含：

```
//---------tinclude.h------

#include <stdio.h>
#include "TArchive.h"
#include "TObject.h"
```

什么意思是司马昭之心——路人皆知吧？我们就是要在以后加进来的一些类中，只要加入这个头文件就不需要加入任何其他头文件了，仅此而已，所以我们以后如果有必要，那些头文件一样向里面加，不怕的，我们的类中都有保护机制的，也就是牛刀在宏那一节与大家说过的，防重复定义的宏可以保证我们定义不重复，那么好，我们就进入主函数去包含进这个头文件就一切 OK 喽：

好，下面我们就进入 TArchive 类，去定义一大堆的符号重载函数，我们先重载“<<”符号，用来保存文件啊，呵呵，（当然，我们先去定义构造函数吧：

```
//当然先得定义一个成员变量，对了，定义下面变量可要包含 stdio.h 头文件啊，不然要报错的，这可不是 MFC 库哟，呵呵
        FILE *m_fp;
//下面这个不多见吧？
        enum Mode{load,store};
//上面的变量应该在哪儿定义知道吧？下面就定义构造函数喽：
TArchive::TArchive(FILE *fp, int nMode)
{
        //第二个字符传进来，让我们明白了怎么读写，当然，我们的类要知道读写当然还要定义一个布尔变量
        //我们就假设定义了一个成员变量 bool m_bIsRead 吧，当然，我们下面就可以对这个变量初始化喽
        if(load==nMode)
        {
                m_bIsRead=true;
        }
        else
```

```
        {
                m_bIsRead=false;
        }//上面是将我们的成员变量赋了值，接下来就可以对我们的文件指针的成员变量赋值喽：
        m_fp=fp;        //简单吧？其实最简单的就是最好的，注意喽，在 MFC  库中，这里传递的可是 CFile
                                //的类哟,别弄混了,我们一切是为了简单，为了方便，而  MFC  一切是为了封装
}

//下面我们的  m_  开头的就是成员变量，我们最多交待一下其类型，大家和牛刀跟这么长时间来了，在哪儿定义变量应该是得心应手了吧？
//好，下面就们就要去疯狂地重载运算符喽，当然，我们这就重载两三个，重载过多没意思，我们的目的是为了和大家说明一些总是而已，
//牛刀的这些例子是不求面面俱到的，只是希望大家能够练就一双慧眼，去看穿微软在  MFC  的把戏（对了，听说盖兹退休了，呵呵）

TArchive & TArchive::operator <<(int i)
{       //这儿重载的是整型，下面再来一个字符型，我们就算完成任务

}

TArchive & TArchive::operator <<(char ch)
{       //行了，这就行了，什么叫运算符重载大家还记得吧？

}
```

行了,一切都够了,怎么个运算符重载大家还记得吧？我们下面来复习复习，当然喽，我们下面去定义，我们这个简单啊，在重载函数中只需要保存就行了，然后返回就一切  OK  了：

```
TArchive & TArchive::operator <<(int i)
{
        if(m_bIsRead)
        {
                printf("对不起，写模式不对");
        }
        else
        {
                fwrite(&i,4,1,m_fp);
        }
        return *this;        //注意了，这儿要返回的是引用
}

TArchive & TArchive::operator <<(char ch)
{
```

```
        if(m_bIsRead)
        {
                printf("对不起，写模式不对");
        }
        else
        {
                fwrite(&ch,1,1,m_fp);
        }
        return *this;          //编程没这么舒服的了，你觉得呢？下面就编译看看，有没有错。
}
```

这个能行吗？牛刀也有点不信，真得打算做和段程序看一看，那么来吧——

```
#include "tinclude.h"

void main()
{
        FILE *fp=fopen("我的测试文档.txt","wb");
        TArchive ar(fp,TArchive::store);
        ar<<96<<97<<98<<'z';
}
```

你还别说，效果还就不错，相当地不错，我们进入文件看一看：

`  a  b  z

不免有一种冲动，为什么不将“>>操作符顺便重载一下呢？这就来喽——

```
TArchive & TArchive::operator >>(int &i)
{
        if(m_bIsRead)
        {
                fread(&i,4,1,m_fp);
        }
        else
        {
                printf("对不起，读模式不对");
        }
        return *this;
}

TArchive & TArchive::operator >>(char &ch)
{
        if(m_bIsRead)
        {
                fread(&ch,1,1,m_fp);
        }
        else
        {
```

```
                printf("对不起，读模式不对");
        }
        return *this;
}
```

请大家注意喽，读模式中的参数全是引用，这一点与写不同，因为读模式需要改变这些引用的参数，下面我们来定义三个整型数据与一个字符型数据，去读出我们写进文件的数据，当然，写那一部分我们先删喽——

```
#include "tinclude.h"

void main()
{
        FILE *fp=fopen("我的测试文档.txt","rb");
        TArchive ar(fp,TArchive::load);
        int x,y,z;
        char ch;
        ar>>x>>y>>z>>ch;
        printf("x=%d  ,  y=%d  ,  z=%d  ,  ch=%c\n",x,y,z,ch);
}
```

运行结果——

```
x=96  ,  y=97  ,  z=98  ,  ch=z
```

耶，我们成功喽。

当然，这多技术上仅仅是一个小成功，因为这其实就是一种正常的，甚至是普通的运算符重载而已嘛，我们这里仅仅是将上一讲的 Load 和 Save 函数变成了两个运算符而已嘛，你不要小看这一变，我们上面的 ar>>x>>y>>z>>ch; 中的 “>>”多么像传送带啊，将 ar 中的数据源源不断地从文件中传了出来，我们称之为——流文件，或者流化文件，或者将其看成一串，所以成称之为——串行化。

也正是这种串行化，使得我们的类的保存与恢复，进入了工厂化、流水线的时代——

大家还记得我们上面还定义一个 TObject 吗？下面我们在这里将其也串行化为友元（什么？什么叫友元？牛刀怎么以前没有提到过？）

那好，这一讲剩下的时间，我们先与大家将什么叫友元弄明白，其他的一些东东，我们放到下一讲再与大伙详细讨论：（新建一个新的控制台应用程序）

```
#include <stdio.h>


class A
{
        int a,b;
        A()
```

```
        {
                a=4;
                b=5;
        }
};

void fun()
{
        A a;
        printf("A::a=%d\nA::b=%d\n",a.a,a.b);
}

void main()
{
        fun();
}
```

大家看上面的程序，编译肯定报错，那是肯定的，因为我们的类 A 中所有的变量与函数甚至构造函数都是私有的，也就是说，这个类对于外界来说，是一个根本没有用的类，好像一个房子没有门，然后有一种函数，或者一种类，可以在上面开一个天窗，那就是友元，就是朋友嘛，那么怎么申请朋友呢？很简单，在类中将友元的函数原型再定义一下，前面再放一个 friend 即可，看程序：

```
#include <stdio.h>


class A
{
        friend void fun();
        int a,b;
        A()
        {
                a=4;
                b=5;
        }
};

void fun()
{
        A a;
        printf("A::a=%d\nA::b=%d\n",a.a,a.b);
        a.a=123;
        a.b=456;
        printf("经过改变后，新的 A 中变为\n");
        printf("A::a=%d\nA::b=%d\n",a.a,a.b);
}
```

```
void main()
{
        fun();
}
```

你相信上面的程序能够运行吗？并且能够得到正确的结果，可以说， C++中的这种朋友可真是肝胆相照啊，可这仅仅是单向的可能一方是赤胆忠心，另一方却是口蜜腹箭，所以说，这种朋友既没有传递性，又没有双向性，都是一方的一厢情原，将自己的类中的一切都贡献出来了。

可能你有担心，这会不会破坏类的封装性？其实没有这个担心的必要，因为我们的朋友是类自身确认的，与父子，与其他都无关，所以说，如果你不能进入别人的类中去将自己申请为朋友的话，对方绝对不会认你这个朋友的。

为什么到现在才与大家说这个，因为现在与大家可以三言两语就将这个定义说得清楚，如果在一百讲之前，可能和你说上好久你也不一定弄得明白，呵呵。

那么我们将一个类声明为朋友，那么这个类就称为“友元类”，说实话，友元类比子类的权限还要大，说实话，做儿子没有做朋友强哟。

我们这一讲不打算与大家深入了，留一点聊天的时间吧，不知大家体会到没有，微软的 C++封装的强大，我们几乎可以封装一切我们想要封装的东东，像我们上面的那种将文件向磁盘上存储与在磁盘上读取文件这样的操作，竟然可以这样被完美地封装成一个运算符，当然，这并没有增加软件的开发难度，因为微软的 C++ 有重载技术，在没有重载时，这种流技术可以说几乎是一个梦想，或者即使能实现可以编程也会相当复杂。

当然，方便的代码就是文件的增加，库的增加，编译器大小的增加，大家应该知道，编译器给我们的优化是有限的，而我们的思路却是无限的，我们要把这有限的编译器的优化用到无限的思路之中去（怎么那么别扭？呵呵）。

说实话，现在这种面向对象的思路的确是一种相当优秀的思路，说实话你不信，在没有这些面向对象技术之前，做一款程序要画好多的程序流程图，并且先将问题整体考虑，然后再逐步地细化、细化，直到我们的思路能够变成代码，然后用手敲上机器。

类、封装、重载……

真是伟大的创举～～～

啪，咱们下讲再见。

**微软积木一一六：串行文件**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇七年七月三日
备注：今天论坛好奇怪，附件不能插入，好在只有一个图，图片放在下面，文章中的 <图一> 便是应该放置图片的位置，请管理员看一下什么原因，谢谢。
*/
```

串行化的技术，我们一直在用，例如我们以前用的 cout<<"0Ginr"<<endl; 之类的，不就是串行化吗？我们上一讲也与大家讲了串行化的一些例子。

那么我们这一讲要与大家说什么呢？如何使得我们的类能够实现串行化，说实话，如果真的想将所有的类都实现串行化啊，那还真的不现实，不很现实，原因很简单啊，我们的类大小不一样啊，我们怎么能够知道我们的类怎么去存数据，存多少数据呢？一切都无从谈起，于是，
就需要用一些特殊的手段，当然喽，这一讲开始，我们暂时先不要将我们的手段得那么明显，一开始我们的目的是为了让大家明白怎么去实现这种串行化技术。

看我们假设我们的 TObject 已经实现了串行化，那么为了在 TObject 中能够使用 TArchive，中的一些数据，那么我们就必须使用我们上一讲最后讲到的——友元。

但是在我们这儿也用不着，因为我们的 TArchive 中的所有参数与函数全部都是公有的，我们试着跟大家弄着玩吗。不过如果大家看到微软的源代码前面有 friend 不要奇怪，那也就是说，对于 CObject 来说，CArchive 的全部的数据都可以访问，那么我们来定义这个 CObject 的重载的例子——

```
TArchive & TArchive::operator >>(TObject &object)
{
        object.Load(*this);
        return *this;
}
TArchive & TArchive::operator <<(TObject &object)
{
        object.Save(*this);
        return *this;
}
```

有点明白的意思吗？我们是在类中使用 Save 和 Load 去实现这种串行化的，那么我们下面去定义一下这两个成员变量：

```
void TObject::Load(TArchive &ar)
{
       ar>>m_a>>m_b;//看明白了什么意思了吧？串行化过程放这里来了
}
```

```
void TObject::Save(TArchive &ar)
{
        ar<<m_a<<m_b;          //方便啊，太方便了
}
```

对了，这里说一下啊，在 CAchive 中如果要用要 CObject 对象，或者在 CObject 中要使用 CAchive 对象，请大家在本类前面分别定义一下另一个类的函数原型，看牛刀的一段代码——

```
class TArchive;
class TObject
{
        int m_a,m_b;
public:
        void print();
        void Save(TArchive & ar);
        void Load(TArchive &ar);
……
```

什么意思懂吧？我们在编译文件时，并不是所有的头文件都打开的，而我们这里使用了一个统一的头文件 tinclude ，所以分另打开这个头文件后，一个类在这个头文件中看不到另一个类的定义，而另一个类也一样，而文件的编译是分别编译，也就是说生成不同的 obj 文件，并且互相独立编译的，每个文件只打开相关的头包含，而不是将全部头文件全部打开（你想这也不其道理啊，如果编译器去见到 #include 就将头文件打开，那么可能在编译时就会进入递归包含，好像不很好理解吧？我们看一个示例——）

```
//        a.h
#include "b.h"
//        b.h
#include "a.h"
```

像上面如果我们打开 a 后，发现有 b 头文件，又去打开，然后又看到 a ，又去打开，于是就出现了文件包含上的递归，虽然每个包含都会有相应的防重复措施，但编译器却编译不下去，是也不是？于是，我们的编译器一般仅打开 cpp 中的头包含，进而编译成相应的 OBJ 文件。

那么好，我们的 tinclude.h 头文件中并没有任何的类的定义，所以就会出现找不到类的编译错误，所以需要在类前面加上一个类原型（注意哟，可不是函数原型哟），我们下面看看编译出来的三个不同的 obj 文件——

好了，那么下面我们就去定义那个 Save 和 Load 成员函数吧——

```
void TObject::Load(TArchive &ar)
{
        ar>>m_a>>m_b;          //看明白了什么意思了吧？串行化过程放这里来了
}
void TObject::Save(TArchive &ar)
{
```

```
        ar<<m_a<<m_b;          //方便啊，太方便了
}

void TObject::print()
{
        printf("在 TObject  中，  m_a=%d,m_b=%d\n",m_a,m_b);
}
```

这种方便性让人瞠目，牛刀在上面顺便就定义了一个 print 函数，去打印数据，方便啊，呵呵。那么好，我们去主程序中去做一些程序瞧一瞧——

```
#include "tinclude.h"
void main()
{
        FILE *fp=fopen("我的测试文档.txt","wb");
        TArchive ar(fp,TArchive::store);
        printf("构造一个类 a  ，参数为  3,8\n");
        TObject a(3,8);
        a.print();
        printf("准备串行化喽\n");
        ar<<a;          //请注意，这儿的 a  可是一个类哟
        printf("串行化成功\n");
        fclose(fp);
}
```

有点感觉了吧？

那么好，我们去看看文件——

└ ◘

真的看不出什么来，那么好，我们下面去再写一个串行化的例子，去读数据，当然，这次咱们用的可就是无参的构造函数喽——

```
#include "tinclude.h"
void main()
{
        FILE *fp=fopen("我的测试文档.txt","rb");
        TArchive ar(fp,TArchive::load);
        printf("构造一个无参类\n");
        TObject a;
        printf("准备串行化喽\n");
        ar>>a;          //还是那么舒服，下面我们打印出结果看看效果——
        a.print();
        printf("串行化成功\n");
        fclose(fp);
}
```

好了，结果很舒服——

```
构造一个无参类
准备串行化喽
```

在 TObject 中， m_a=3,m_b=8
串行化成功

好了，我们 TObject 的类的串行化问题不错是解决了，那么这毕竟是万里长征第一步，咱们总不能每次都这么复杂的地在 CArchive 中加入一种一种的新类，去定义一个一个新的串行化的重载运算符吧？

微软很聪明，其实我们上边是给大家介绍如何串行化，其实我们的 TObject 中也没有成员变量，也没有有参数的构造函数，而只有两个 成员函数——Save 和 Load ，并且都是虚函数，也就是前面加上了标志 virtual 看程序——

```
#include "tinclude.h"
class TArchive;
class TObject
{
public:
        virtual void print(){printf("这是基类的 print 函数，你没有覆盖");}
        virtual void Save(TArchive & ar){printf("请您去覆盖这个 Save 函数，就可以实现流式化喽\n");}
        virtual void Load(TArchive & ar){printf("请您去覆盖这个 Load 函数，就可以实现流式化喽\n");}

        TObject();
        virtual ~TObject();

};
```

想干什么？

呵呵，作一个基类啊，将想流式化的子类都定义为这个类的儿子，再覆盖这些函数，就可以顺利实现流式化，不信咱们来定义一个类，这假设这个类为一个鼠标指针坐标 TPoint 结构，看程序——

```
#include "tinclude.h"
class TPoint :public TObject
{
        int x,y;
public:
        TPoint(){};
        TPoint(int a,int b){x=a;y=b;}
        void Save(TArchive & ar){ar<<x<<y;}
        void Load(TArchive & ar){ar>>x>>y;}
        void print(){printf("x=%d,y=%d\n",x,y);}
};
void main()
{
        FILE *fp=fopen("我的测试文档.txt","wb");
        TArchive ar(fp,TArchive::store);
```

```
        TPoint point(20,31);
        point.print();
        ar<<point;
        fclose(fp);
}
```

怎么样？就需要将父类定义为 TObject，然后去将成员函数覆盖一下，就可以顺利地实现流式化。

也就是说，我们想要在我们的流式文件中保存文件或者提取文件就显得格外地简单了，当然，上面的程序，我们再读出来看一看——

```
void main()
{
        FILE *fp=fopen("我的测试文档.txt","rb");
        TArchive ar(fp,TArchive::load);
        TPoint point;          //无参
        ar>>point;          //提取
        point.print();
        fclose(fp);
}
```

妙不可言啊，呵呵。（注：如果你叫牛刀不要崇洋媚外请你去想一个比微软的这种技术更先进的技术出来，牛刀只能说——技术是没有国界的，有时候别人比我们厉害，我们只有承认，但是我们同样可以去创新，那就要看我们有没有那种创新的脑袋了，从这一点上来说，中国的龙蕊的制作未免在牛刀看来说是一个失败，其实有本事真的不需要去模仿别人，花那么大的精力才做一个兼容的 CPU ，如果是牛刀搞龙蕊，那么首先搞技术的一趟班子至少应该保证其安全性，先进性，在此基础上设置出的机器应该是六十四位，全中文汇编，指令数据分离机制。当然这样的电脑做出来，是任何用处都没有的，因为没有任何一款现有的电脑软件能在上面跑，那么芯片出来了，先做个模拟器啊，呵呵，将现有的电脑系统所有的软件都看成一条数据流，保存到磁盘中的也仅仅是一串数据，从这个角度看问题，病毒的基本生存空间就没有了，呵呵，当然，这种硬件上的安全，估计是谁都不会乐意去做的，因为没有任何效益；看过牛刀汇编教程的朋友应该还记得，我们的电脑为什么会出现病毒，其实其主要的就是其结构是指令数据存储结构，并且指针与数据系统是不分的，而这种结构是最不稳定的，原因很简单，只需要设置几个数据就可以使得 IP 地址乱串，能有好吗？）

呵呵，有点扯远了，总之，没有硬件安全就不可能有软件安全，你家的墙上如果有窟窿那么防盗门再厚再好也无济于事，因为盗贼就没有必要从防盗门进来。

说实话，其实汇编语言是有语言规则的，如果都按规则来，那么就不会有病毒，然而，好奇之心，人皆有之，这也是没有办法的事。

好了，上面的过程已经足以证明了我们可以通过父亲虚函数的办法使得子类可以实现串行化，可惜的一点是——子类之间的关系却不能保存下来，那么假如

一个父类有好多子类，怎么让父类知道我们当前应该调用哪一个子类呢？也就是说，必须让父类知道有这样一个子类，并且还要让父类知道什么时候调用这样的子类，于是，父类就需要向子类汇报工作，当然，这个过程我们完全可以用以下的结构来实现——

```
A
{
public:
        void Fun();
...
}
void A::Fun()
{
        ...
}
...

B
{
public:
        B();
        ...
}
...
```

上面我们仅仅说明一个结构，大家看，一般情况下，我们定义一个 A 类，那么实现部分则在 A 类的后面，而 B 类的实现部分则在 B 类的后面。

请问，有谁规定必须这样了吗？

难道不能像这样吗？

```
A
{
public:
        void Fun();
...
}
...

B
{
public:
        B();
        ...
}
...
void A::Fun()
```

```
{
        ...
}

B::B()
{
        ...
}
...
```

看，我们将 A 的类的定义部分拿到了 B 的后面，应该是没有问题吧，在哪儿定义都是一样的。

不错，是一样的，但效果不同，下面的一种方案告诉我们，这意味着，我们在 B 类定义之后，可以继续去定义 A 类的成员函数，而此时，我们的 B 类已经有了雏形了，也就是说，可以通知 A 类的某个函数去在 A 类中注册我们 B 类的一些信息，以便 A 类在串行化时做一些识别，当然，这是一个函数，而这一个函数已经形成了固定的模式，从而在我们的类的设置中，可以将其设置成为一个宏，到了时间，调用一下宏就可以了，甚至可以注册版本号，注册类名，等等。

也正是基于这种基本的思想，我们就可能使用几个简单的宏与一些基本的规则，来使得我们的某个类实现对象的串行化，其基本规则为——

1.从 CObject 或其派生类派生出用户的类

//注：CObject 是串行化的基类

2.重载 Serialize()成员函数，加入必要的代码，用以保存对象的数据成员到 CArchive 对象以及从 CArchive 对象载入对象的数据成员状态。

//这个成员相当于我们前面定义的 Load 和 Save ，这个函数结整合在一起了，用起来很方便

3.在类声明文件中，加入 DECLARE_SERIAL 宏。编译时，编译器将扩充该宏，这是串行化对象所必需的。

//请注意位置，这个宏使得 CArchive 能够识别这个类（请大家注意了，我们上面自己做程序举的例子其实是先用无参构造函数恢复一个类，然后通过运算符重载实现这个类中的数据恢复，然而，有了这个宏，我们的对象可以直接从类中恢复出来感觉上是无中生有，而其新建对象的过程是由基类完成，而新建的对象的类名则由这个宏给顺利地传递过去。

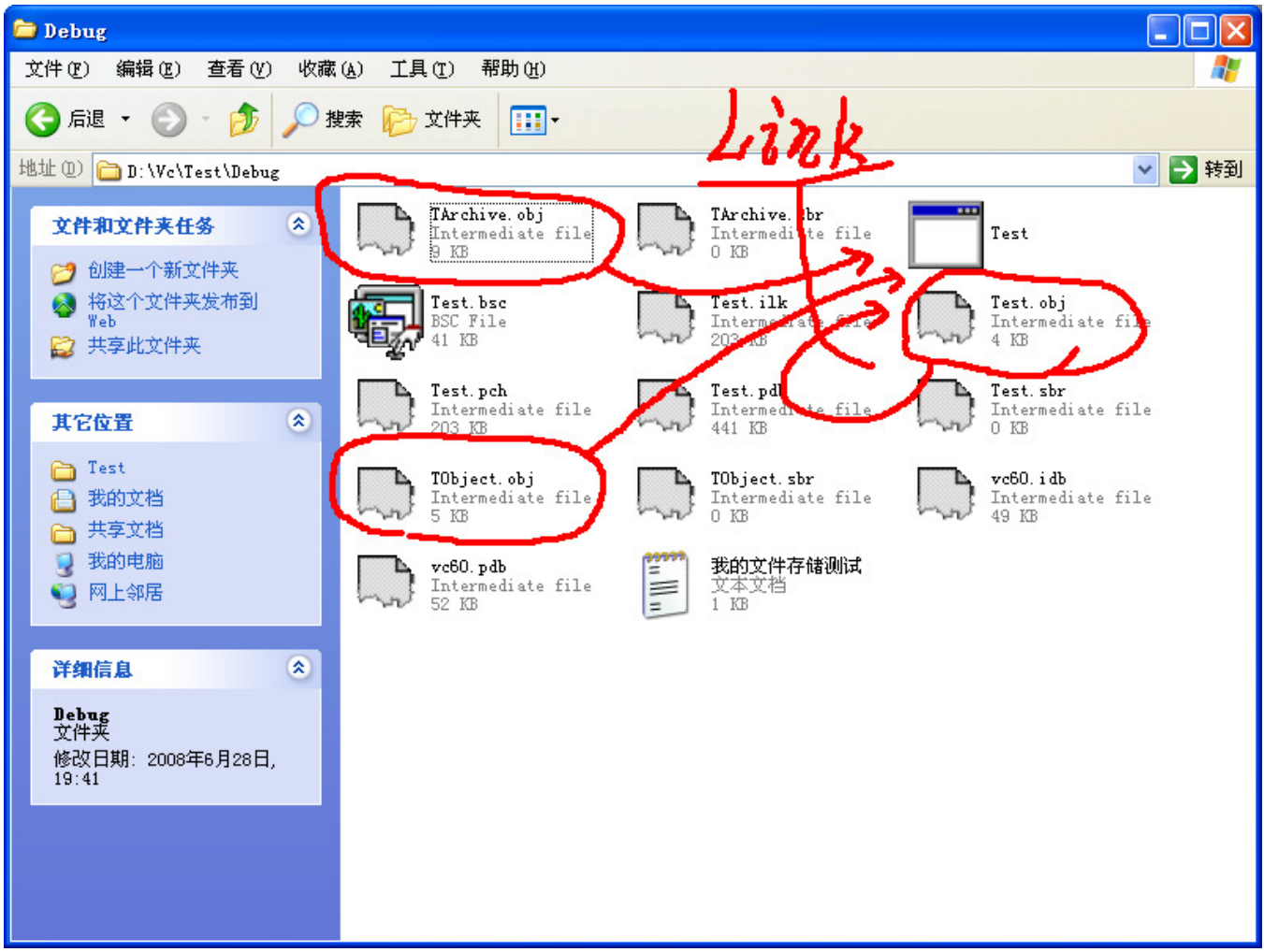
4.定义一个不带参数的构造函数。

//什么意思应该明白吧，这是鸡肋，标准的鸡肋

5.在实现文件中加入 IMPLEMENT_SERIAL 宏。

//实现类名与版本信息的保存，以后做到再与大家详细详解。

啪!!!!!! OVER


Debug
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退 搜索 文件夹
地址(D) D:\Vc\Test\Debug
转到
Link
文件和文件夹任务
创建一个新文件夹
将这个文件夹发布到Web
共享此文件夹
其它位置
Test
我的文档
共享文档
我的电脑
网上邻居
详细信息
Debug
文件夹
修改日期: 2008年6月28日, 19:41
TArchive.obj
Intermediate file
9 KB
TArchive.sbr
Intermediate file
0 KB
Test
Test.bsc
BSC File
41 KB
Test.ilk
Test.obj
Intermediate file
4 KB
Test.pch
Intermediate file
203 KB
Test.pdb
Test.sbr
Intermediate file
0 KB
TObject.obj
Intermediate file
5 KB
TObject.sbr
Intermediate file
0 KB
vc60.idb
Intermediate file
49 KB
vc60.pdb
Intermediate file
52 KB
我的文件存储测试
文本文档
1 KB

**微软积木一一七：串行对象**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年七月四日
注：还昨天情况，插不了图片
*/

　　这一讲的开始，我们已经准备去接近微软的对象串行化的核心了，还回到我们的软件——Draw　去串行化我们的新类　CDraw　。

　　当然，在串行化那个对象之前，我们先去看一个我们以前不曾经亲密接触过的类——CDrawDoc

　　大家打开这个类，看其中有一个成员函数——

```
void CDrawDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                // TODO: add storing code here
        }
        else
        {
                // TODO: add loading code here
        }
}
```

　　看，微软将我们的　Load　和 Save　结合到一起了，成了一个成员函数，这样有一个好处，那就是——我们只需要将这一个成员函数定义下来，我们的串行化的主导部分就结束了，于是，串行化中有一点就是——覆盖这个成员函数。

　　那么也就是说，我们的这个　Doc　类就是已经被串行化好了的，并且有一个好处，这个串行化的过程已经与文件的打开与保存容为了一体，不信，我们去定义几个数据，看看效果——

```
void CDrawDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                CPoint pt(48,49);
                CRect rect(50,51,52,53);
                CString str("牛刀也来试试串行化类，看效果");
                ar<<pt<<rect<<str;
        }
        else
```

```
        {
                CPoint pt;
                CRect rect;
                CString str;
                CString strtmp;
                ar>>pt>>rect>>str;
                strtmp.Format("pt.x=%d,pt.y=%d\nCRect(%d,%d,%d,%d)\ns
tr=%s",pt.x,pt.y,rect.left,rect.top,rect.right,rect.bottom,str);
                AfxMessageBox(strtmp);          //请注意，我们的 Doc
类不是 CWnd 的子类，所以就没有窗口句柄，所以我们必须用全局的消息函数
        }
}
```

请大家注意，我们的 Doc 不是 CWnd 的成员，也就是说其没有窗口句柄（本来就不是窗口类嘛），所以不能使用 MessageBox 函数来显示消息，所以我们用了一个全局的消息处理函数—— AfxMessageBox 只要前面有 Afx 的，全是全局的函数，在 MFC 编程的任何地方，都可以使用，很是方便，好了,大家去保存一个数据(注——当然用文件/保存命令喽),然后按文件/打开),没反应，为什么？

微软的程序在偷懒，说实话，跟你借一百元钱，你身上正好有，你会回去取吗？回去取显然是得不偿失，没想微软的软件也来这一手，那么好，关掉程序，重新运行文件，再次用打开命令（注意喽，这次不要用保存喽），于是，我们打开了文件，看效果——

<图一>

你不要想打开第二次，微软不干，呵呵，下面我们去准备干点恶心的事——

```
void CDrawDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                CPoint pt(48,49);
                CRect rect(50,51,52,53);
                CString str("牛刀也来试试串行化类，看效果");
                ar<<pt<<rect<<str;
        }
        else
        {
                CPoint pt;
                CRect rect;
                CString str;
                CString strtmp;
                ar>>rect>>str>>pt;          //看牛刀串行化文件与上面的
顺序不同，要注意区别啊
```

```
                strtmp.Format("pt.x=%d,pt.y=%d\nCRect(%d,%d,%d,%d)\ns
tr=%s",pt.x,pt.y,rect.left,rect.top,rect.right,rect.bottom,str);
                AfxMessageBox(strtmp);
        }
}
```

这次，这次微软给了我们一个教训——

<图二>

这个教训够深刻，这是怎么回事呢？原来啊，微软在保存数据的时候，做了一些版本管理，对了，大家看到了没有，我们上面用的 CPoint CRect CString 都是微软的类库中的类啊，也就是说，这些类都已经实现了串行化，可以直接使用喽。

那么好，我们下面就去将我们的程序实现串行化，先看我们的 CDraw 类

·空构造函数——有了，我们就没有删，也没有改

·从 CObject 派生，我们在类的定义后面加上一段就行：

```
class CDraw  :public CObject        //后面的是我们后加的
{
        BOOL m_bDrawType;
        CRect m_rect;
public:
        void OnDraw(CDC *pDC);
……
```

·Serialize 成员函数，我们去定义一个就行喽——（注意，因为其中的程序格外地简单，我们也一并定义喽）

```
void CDraw::Serialize(CArchive &ar)
{
        if(ar.IsStoring())
        {
                ar<<m_bDrawType<<m_rect;
        }
        else
        {
                ar>>m_bDrawType>>m_rect;
        }
}
```

那么好，还有两个宏，那就更简单了，在上面的定义部分加入一个宏——

```
class CDraw  :public CObject        //后面的是我们后加的
{
        DECLARE_SERIAL(CDraw)         //这儿就是宏，千万要注意，这儿是
宏，所以最后一定不要加分号
        BOOL m_bDrawType;
        CRect m_rect;
public:
```

……

**那个宏后面不要加分号啊，否则就要报错喽**

```
//////////////////////////////////////////////////////////////////////
/
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
/
IMPLEMENT_SERIAL(CDraw,CObject,1)          //这个宏需要三个宏参数
CDraw::CDraw()
{

}
```

……

上面的一个宏需要三个宏参数，第一个是类名，第二个是基类名，也就是我们的 CObject 第三个参数是版本号，如果我们要发行新的版本，并且想与以前的版本不兼容了，那将版本号一改变万事大吉，就这行简单，也就是说，我们这里的串行化其实加入了版本管理的功能，那么好，我们的这个类就已经成功地实现了串行化了。

现在就特别地想去保存我们画的图形，是不是有这么个冲动？

那好，我们还记得我们以前使用的 CPtrArray m_ptrarray;

记得了吗？将其挪到 CDrawDoc 类中去，怎么？我们想用人家 Doc 类去管理我们的数据，我们不交权怎么能行呢？

那好，相应的程序就都得根着改哟。

这可难办了，怎么改啊？如果数据在 Doc 类，那么在 View 类中就首先要得到 Doc 类的类定义本身，难啊难，对了，我们来看看我们这段代码——

```
void CDrawView::OnDraw(CDC* pDC)
{
        CDrawDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
```

……

这段代码似乎在以前的什么地方见过吧，是啊，我们一直没有用啊，这就是得到文档类啊，相当相当嘀简单，我们下面直接用好了，我们在这儿就用这个指针就可以顺利地得到相应的数据了，教大家一个讨巧的办法，定义一个引用，方便啊，那就是给起一个小名，呵呵，就不用每次都写一大段代码喽——

```
void CDrawView::OnDraw(CDC* pDC)
{
        CDrawDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        CPtrArray &ptr=pDoc->m_ptrarray;
        int nCount=ptr.GetSize();
        for(int i=0;i<nCount;i++)
        {
                ((CDraw*)ptr.GetAt(i))->OnDraw(pDC);
        }
}
```

怎么，难道不是很方便吗？

下面当然就要在鼠标抬起的消息上做一些手脚喽——（注意喽，得到文档指针还是上面一段代码，复制过去就行了——

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
        m_ptEnd=point;
        CClientDC dc(this);
        CDrawDoc* pDoc = GetDocument();
        CDraw *pdraw=new CDraw(FALSE,m_ptBeg,m_ptEnd);
        pdraw->OnDraw(&dc);
        CPtrArray &ptr=pDoc->m_ptrarray;
        ptr.Add(pdraw);
        m_bldown=FALSE;

        CScrollView::OnLButtonUp(nFlags, point);
}

void CDrawView::OnRButtonUp(UINT nFlags, CPoint point)
{
        m_ptEnd=point;
        CClientDC dc(this);
        CDrawDoc* pDoc = GetDocument();
        CDraw *pdraw=new CDraw(TRUE,m_ptBeg,m_ptEnd);
        pdraw->OnDraw(&dc);
        CPtrArray &ptr=pDoc->m_ptrarray;
        ptr.Add(pdraw);
        m_brdown=FALSE;
        CScrollView::OnRButtonUp(nFlags, point);
}
```

如果你仔细地去体会，其实程序都很简单，好了，这下我们的数据是保存到 Doc 类中了（对了，说明一下，这个成员变量一定要用公有的哟，我们在 View 类中要直接访问，当然，你也可以将 View 类设成是 Doc 类的友元，也可以，总之，我们做程序的手段千千万，就要看我们怎么用喽）。

既然保存到了 Doc 类中，就不会是没有有的，我们很想将数据保存下来，是也不是？那么好，我们首先要在 Doc 文件中包含着我们的头文件 Draw1.h 注意喽，是 Draw1.h 哟，后面的 1 可不要忘喽。

（对了，上面我们不是没有使用吗？怎么也在 View 类中使用的很好呢？因为我们的 CPtrArray 是无符号指针的集合啊，所以你保存进来了，我根本不需要知道你是什么东东，但是要串行化了，你不知道人家是什么类怎么行呢？）

好，看程序——

```
void CDrawDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		//注：请注意了，我们如何保存呢？那么多的数据，我们怎么知道我们的数据有几条呢？
		//于是，我们先去定义一个整型变量，去保存我们集合中数据的多少，看程序——
		int nCount=m_ptrarray.GetSize();
		ar<<nCount;
		for(int i=0;i<nCount;i++)
		{
			CDraw *pdraw=(CDraw*)m_ptrarray.GetAt(i);
			ar<<pdraw;
		}
	}
	else
	{
		//这儿当然是需要取出个数喽：
		int nCount;
		ar>>nCount;
		for(int i=0;i<nCount;i++)
		{
			CDraw *pdraw;
			ar>>pdraw;		//你瞧，我们并没有对这个类作任何初始化，然而直接就可以产生了
			m_ptrarray.Add(pdraw);
		}
	}
```

我们上面并没有对类进行任何初始化，然而还能够顺利地产生这个类，不觉得好玩吗？绝对是无中生有啊，呵呵，所以我们这种串行化的感觉啊，嗯，咳～～～

好，画几个道道，保存一下，然后去打开，看看效果——

对了，我们以前不是保存的类引用吗？怎么这儿可以保存指针？

我们去看看微软的定义——

```
friend CArchive& operator <<( CArchive& ar, const CObject* pOb );
```

人家就是这样定义的啊，你管人家，人家怎么舒服怎么来，是也不是？

那好，好奇之心人皆有之，我们去看看我们保存进来的文件——

```
|                    ┌    |    CDraw                          ?        ┌
€       R   6┐  i┌  ┌€    ┌     ?  0┌  ┌€┌   ?  6   ?     ┌€┌   ?  ?
 ?  ?
```

怎么样？看见文件中有我们的类名了，是吧？不但有类名，还有版本管理呢，大家将上面的宏定义部分——

IMPLEMENT_SERIAL(CDraw,CObject,1)

改了，改成下面的——

IMPLEMENT_SERIAL(CDraw,CObject,2)

再次运行软件，打开我们以前保存的数据，看到什么了？又是上面的——“非预期格式”是不是？这就是我们串行化对类的版本管理，人家的文件连格式都包含进去了，难怪怎么老是弄错不了呢。

上面的软件已经近乎完美了，果真如此吗？请画两幅不同的图片，例如一个图上全是圆圈，一个图上全是方块，然后分别保存，接下来分两次打开，发现什么了？圆圈与方块是不是混在一块儿啦？

这就是因为我们重新打开一个数据时，没有做好数据的清除工作，那么我们到哪里去做这项工作呢？

Doc 提供了虚函数 DeleteContents 我们去覆盖这个虚函数，然后去做一些清除工作，以下是基本代码——

```
void CDrawDoc::DeleteContents()
{
        //请注意了，由于我们的集合类是在销毁之时，集合后面的数据是向前移的，所以我们一般
        //使用从后向前的销毁办法：
        int nCount=m_ptrarray.GetSize();
        while(nCount--)
        {
                CDraw *pdraw=(CDraw*)m_ptrarray.GetAt(nCount);
                delete pdraw;
                m_ptrarray.RemoveAt(nCount);
        }
                        CDocument::DeleteContents();
```

}

以上是一般的处理方案，我们取出数据后，必须将指针销毁，资源归还系统，这是一个良好的编程习惯，而不要仅仅习惯于用一个 RemoveAll 了事，我们编程应该考虑和需要考虑到的地方，我们还是应该去考虑一点，我们不应该去做一个不负责的人，你说是吗？

**微软积木一一八：视口处理**

/*
　　版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇八年七月五日
*/

　　有没有发现，牛刀以前喜欢用一些比较好玩的词来做标题，现在的文章标题越来越俗了，呵呵，没办法啊，如果弄得与讲得不太搭边的话，也有人提过类似的建议，说看标题不能知道牛刀讲的内容，呵呵，真难。

　　我们前面花了好多的时间与大家讲了文档的串行化问题，可以这样说，一旦实现了串行化，我们就几乎用不着为我们文档的保存费太大的脑筋，因为微软的 Doc 的功能太强了，什么都为我们做了，我们还有什么交给 MFC 有不放心的呢？

　　我们曾经说过，我们这里的结构为 文档——视 类结构，那么文档的问题我们讲完了，那么视类的结构呢？这一篇开始，我们主要与大家讲一讲有关视口的问题。

　　当然，我们前面讲的什么左键画方右键画圆啊，都是在我们的视口中处理的，那么这个视口还有什么特别的吗？当然有，大家还记得开始叫大家做程序的时候，牛刀是叫大家将 View 的基类设置成 CScrollView 类吗？这个类其实是一个滚动视窗类，既然滚动，就应该有滚动条啊，为什么我们没有发现？

　　因为我们操作的范围大于视窗范围（注：我们看到的那一部分叫视口哟），那么我们如何设置视窗的范围呢？有一个函数——

SetScrollSizes

　　大家到 View 类的 OnInitialUpdate 成员函数中去找找，牛刀找到的结果是：

```
CSize sizeTotal;
// TODO: calculate the total size of this view
sizeTotal.cx = sizeTotal.cy = 100;
SetScrollSizes(MM_TEXT, sizeTotal);
```

　　微软真扣门，就定义了一个 100X100 的，难怪没有滚动条呢？我们改，改为 2000 ，呵呵，想必你的显示器的宽度不会超过 2000 吧？肯定会有滚动条喽。

<图一>

　　大家画两幅图，拖动拖动滚动条看看，我们画的东东竟然会跑，呵呵，可想微软给我们在背后做了多少的工作？

你拖到没有图的地方，再画一个圆，再拖回来，一看，怎么回事？我们在其他的地方画的圆怎么跑上来喽？（牛刀不贴图了，希望大家去实践，不然牛刀说一些问题，你必然会莫名其妙的，呵呵。）

当然，你如果真的不去实践，那么看牛刀的教程其实是很痛苦的，真的，牛刀喜欢调人味口，又不告诉你结果。

有了问题，就要去找原因什么原因呢？看图——（哎，说实话，有时候图片的表述力真的比文学强，没办法）

<图二>

我们看，那幅风景画，我们称之为视图，也就是一幅图片吧，而右边与下面两幅图片上的画，分别是这个视图中的一部分，也就是只显示了一部分，我们看到的这一部分，我们称之为——视口。

需要说明的是，在 OnDraw 函数中，绘图是在视图上绘制的，而在鼠标消息响应的时候，我们的图形是在视口上绘制的。

并且不管是视口还是视图，左上角的坐标总是 0,0 。

什么意思大家应该明白吧，我们鼠标只能得到视口的坐标，而得不到视图的坐标，那么假如视口与视图的左上角重合了，一般这种差别就显示不出来，而一旦这视图与视口的左上角坐标不一样，那就很问题了。

例如我们看下面那个视口，假如我们在这个视口的小太阳（注意，在山头上还没出来呢，呵呵）上点一个点，那么假设这个坐标刚好是 (100,100) （注：相对于视口），那么假设我们这个视口左上角对应于视图上的云的那一块的坐标刚好是 (300,300) ，那么，理论上说，我们的小太阳那个位置的的山头上的坐标就应该是 (400,400) 是也不是？然而，你知道，我知道，电脑却不知道，可惜了的电脑，你用程序告诉她的事情，那办得可是异常完全，但如果你不告诉电脑，那么电脑也就是一台电脑，永远是一台机器，聪明的傻瓜，呵呵。

别抬杠，可能你也发现有漏洞了，那么怎么在 OnDraw 函数中就是在视图上画，并且画得坐标还就不出错呢？照牛刀上面的理论， OnDraw 中画应该有什么地方会改变我们的坐标？

你还别说，有服咱们不在 OnDraw 中画，咱们将代码拿到 WM_PAINT 函数中去，咱们去响应 OnPaint 消息——

```
void CDrawView::OnPaint()
{
        CPaintDC dc(this);
        //当然喽，将上面代码复制过来也可以，咱们偷个懒
```

```
        OnDraw(&dc);
}
```

改过以后，你再画上两幅画，去拖动滚动条，看看效果，可能是一塌糊涂，显示得怎么那么别扭？

这里我们就要说了，我们这里的 OnPaint 也是在视口上操作的，而没有真正地在视图上操作，你滚动条不管拖到哪里，图形怎么不好，你将窗口先最小化，然后再还原，看到什么了？你又看到了先前画的图了，是也不是？

那就奇了怪了，谁在里面做了手脚？

父亲，也就是 CScrollView 类在我们这儿做了手脚。

好，那我们就来复习一下以前学习的一些内容——

父亲定义的虚函数，如果子类中没有同名函数，则执行父类中的函数，如果子类中有相应的函数，就遵守同名覆盖的原则。

也就是说，父类中一定响应过 OnPaint 消息，并且给我们的程序做了手脚，如果不信，我们在子类去调用父类——

```
void CDrawView::OnPaint()
{
        CScrollView::OnPaint();
}
```

我们的软件又正常了，那么，我们的父类到底在干什么呢？

```
void CView::OnPaint()
{
        // standard paint routine
        CPaintDC dc(this);
        OnPrepareDC(&dc);
        OnDraw(&dc);
}
```

牛刀跟了过去， CScrollView 竟然是 CView 的子类，呵呵，调用了 CView 中的 OnPaint ，那么好，我们看到了这里， CPaintDC 定义 DC 没有问题 ，调用 OnDraw 函数没有问题，那么有问题的一定是 OnPrepareDC(&dc);

并且这个函数将我们的 dc 给改了，是不是这么回事呢？我们在我们自己的 OnPaint 函数中也去调用看看——

```
void CDrawView::OnPaint()
{
        CPaintDC dc(this);
        OnPrepareDC(&dc);
        OnDraw(&dc);
}
```

永远不要不想信自己的眼睛，不相信自己的思路，我们的判断非常准确，准确是准备，但就是有问题，并且问题很大，我们如果将自己的图能够画在视图的正确位置呢？

当然，首先要调用一下上面的函数，去设置一下视口，但视口是设置了，而我们 OnDraw 函数是在视口上用已经保存好的数据画的图像啊，而我们新画的坐标如果不经过变换，那么我们上面的 100,100 坐标永远不会变成 400,400 ，原因很简单，我们的坐标不是保存在视口中的，而是保存在我们的 m_ptrarray 集合类中。

注意一点了，我们的 OnPrepareDC(&dc); 函数不管怎么能够变视口，视口的左上角坐标为 0,0 永远也变不了，因为我们视口的坐标被我们称之为物理坐标，而视图中的坐标才被我们称之为逻辑坐标，也就是说，我们看到的窗口左上角的坐标，那才是真的 0,0 点，而相对于这个点来说的我们鼠标取得的永远是物理坐标，而我们显示的区域用的坐标系才是可以变化的呢，而我们的鼠标却不是保存在那里的，于是，我们的鼠标坐标需要单独处理，怎么处理？

我们有两个函数是作物理坐标与逻辑坐标的转换的 DP 为物理坐标 而 LP 为逻辑坐标，于是，我们可以使用下面两个函数——
DPtoLP(&point);
LPtoDP(&point);

去作物理与逻辑坐标之间的转换，于是，我们加入以下的代码，于是，我们以前的程序就变成了如下样式——

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
        m_ptEnd=point;
        CClientDC dc(this);
        CDrawDoc* pDoc = GetDocument();

        OnPrepareDC(&dc);
        dc.DPtoLP(&m_ptBeg);
        dc.DPtoLP(&m_ptEnd);

        CDraw *pdraw=new CDraw(FALSE,m_ptBeg,m_ptEnd);
        pdraw->OnDraw(&dc);
        CPtrArray &ptr=pDoc->m_ptrarray;
        ptr.Add(pdraw);
        m_bldown=FALSE;

        CScrollView::OnLButtonUp(nFlags, point);
}

void CDrawView::OnRButtonUp(UINT nFlags, CPoint point)
```

```
{
    m_ptEnd=point;
    CClientDC dc(this);
    CDrawDoc* pDoc = GetDocument();

    OnPrepareDC(&dc);
    dc.DPtoLP(&m_ptBeg);
    dc.DPtoLP(&m_ptEnd);

    CDraw *pdraw=new CDraw(TRUE,m_ptBeg,m_ptEnd);
    pdraw->OnDraw(&dc);
    CPtrArray &ptr=pDoc->m_ptrarray;
    ptr.Add(pdraw);
    m_brdown=FALSE;
    CScrollView::OnRButtonUp(nFlags, point);
}
```

大家看，我们在保存之前，当然要先调用 OnPrepareDC 去设置 DC ，设置后，我们的 DC 才知道如何去调整我们的坐标到新的值，是也不是？

以上的技术让我们可以将我们的思路在视图上操作，而不要将精力花费在坐标变换上，坐标的变换只需要使用一个函数就一切 OK 了。

如果大家看过牛刀以前的教程，还记得牛刀以前曾经用 API 也与大家做过一个滚动视窗的软件吗？在那里，我们的视窗的所有滚动的设置都需要我们自己去做，视图多大，视口在哪儿，我们的图片坐标应该是加还是减，显示不显示滚动条，显示哪一个滚动条，响应滚动条上的若干个滚动消息等等等等，都是我们必须考虑的一部分。

而微软的滚动视窗类 CScrollView 却仅仅通过一个函数 OnPrepareDC 与几个转换函数就全部解决了。

当然，在这几个函数的背后，微软给我们做了什么，我们却全然不知，我们编程的舒服可是来之不易的啊，呵呵。

其实说实话，滚动视窗部分，被牛刀这样的三言五语，也就算是说明白了吧？牛刀不喜欢说那一大套一大套的理论，说实话，这里的理论其实很多，什么视窗转换的公式啊，什么世界坐标系啊、设备坐标系啊、逻辑坐标系啊，什么坐标转换啊……

甚至我们可以在程序中自己定义坐标空间，在这一点上 VB 做得就特棒，从这一点上说，牛刀比较喜欢 VB 。VB 中的坐标是四角定义的，也就是定义左上角与右下角分别是什么点，然后整个坐标空间 VB 就自

动给我们硬算出来了，并且 VB 中的坐标可以用小数，VB 做完转换后自动取整，真是太太太太太太太太方便了。

而 VC 不行，好多东西都需要自己去做定义与转换。

其实我们使用的编程工具就是与我们的杠杆一样，不可能同时省力与省距离的，我们只能做个折中，说实话，不管你使用什么语言，精通了都可以做出相当棒的程序来。听说中国的 WPS 软件的 DOS 版本竟然是汇编软件做出来的，当然同样是风靡全国，并且使得五笔走入了人们的视线，在中国产生了一股电脑热与打字热。

说实话，我们现在做程序真的是太方便了，什么都没做，微软就给我们提供了一套标准的 MFC 库，直接出一个有用的窗体，菜单、工具栏到常用功能一应俱全，并且还为我们提供了那样好的类的理论，使得我们编程竟然不需要流程图，只需要定义函数，定义类，定义成员函数定义功能直到定义好了我们整个的软件。

我们再来总结一下视口与视图的处理过程——

OnPrepareDC(&dc);

先用上面的函数去设置设备描述表，使得我们看到的部分是视图的正确部分，然后用：

DPtoLP 或者 LPtoDP 去设置视图或者视口的坐标，使得我们保存下去的坐标或者使用的坐标满足我们的需求。

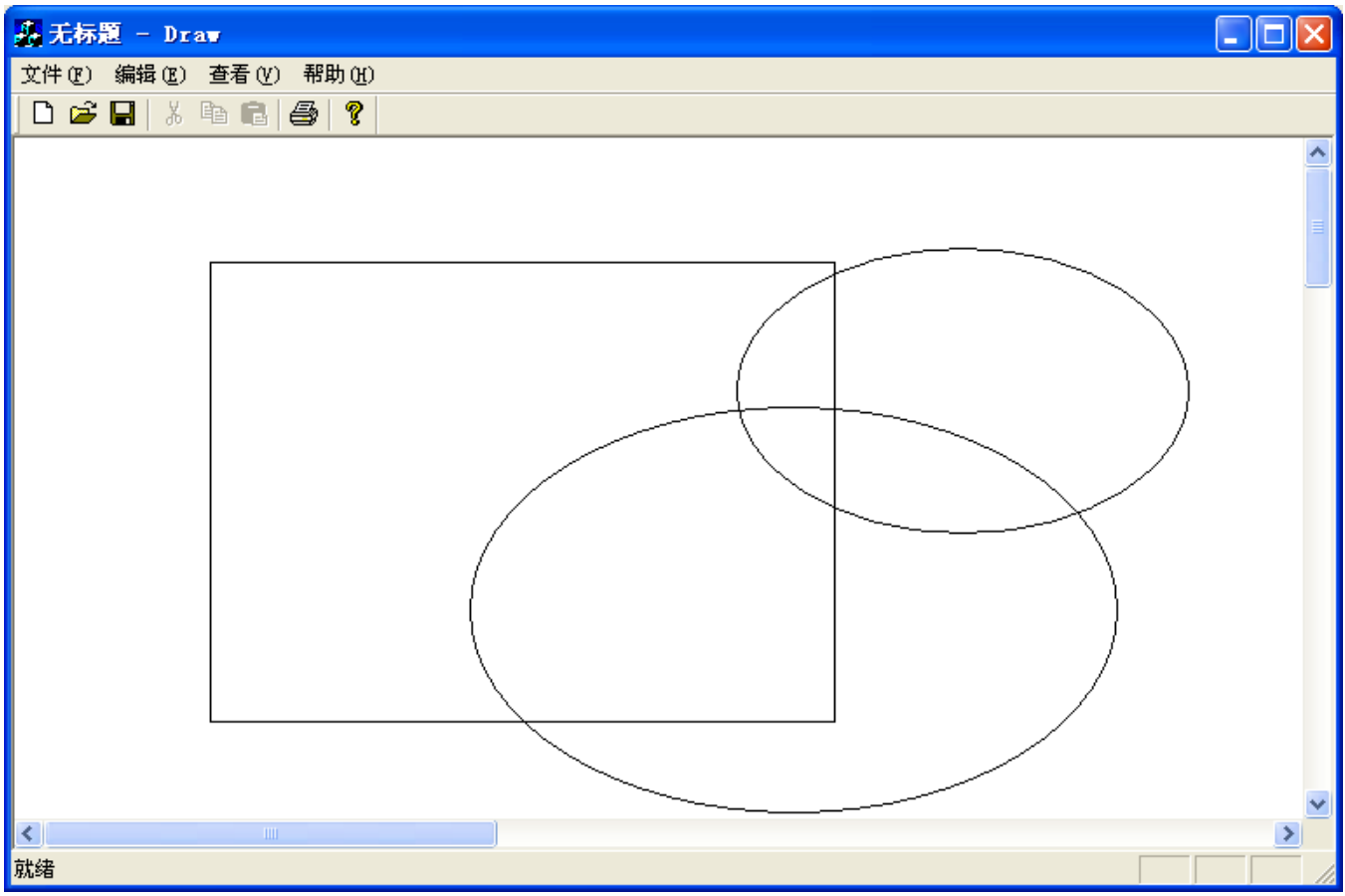
仅此而已，当然，一旦我们的视口与视图的映射模式不一下，就完全不是那么一回事喽。

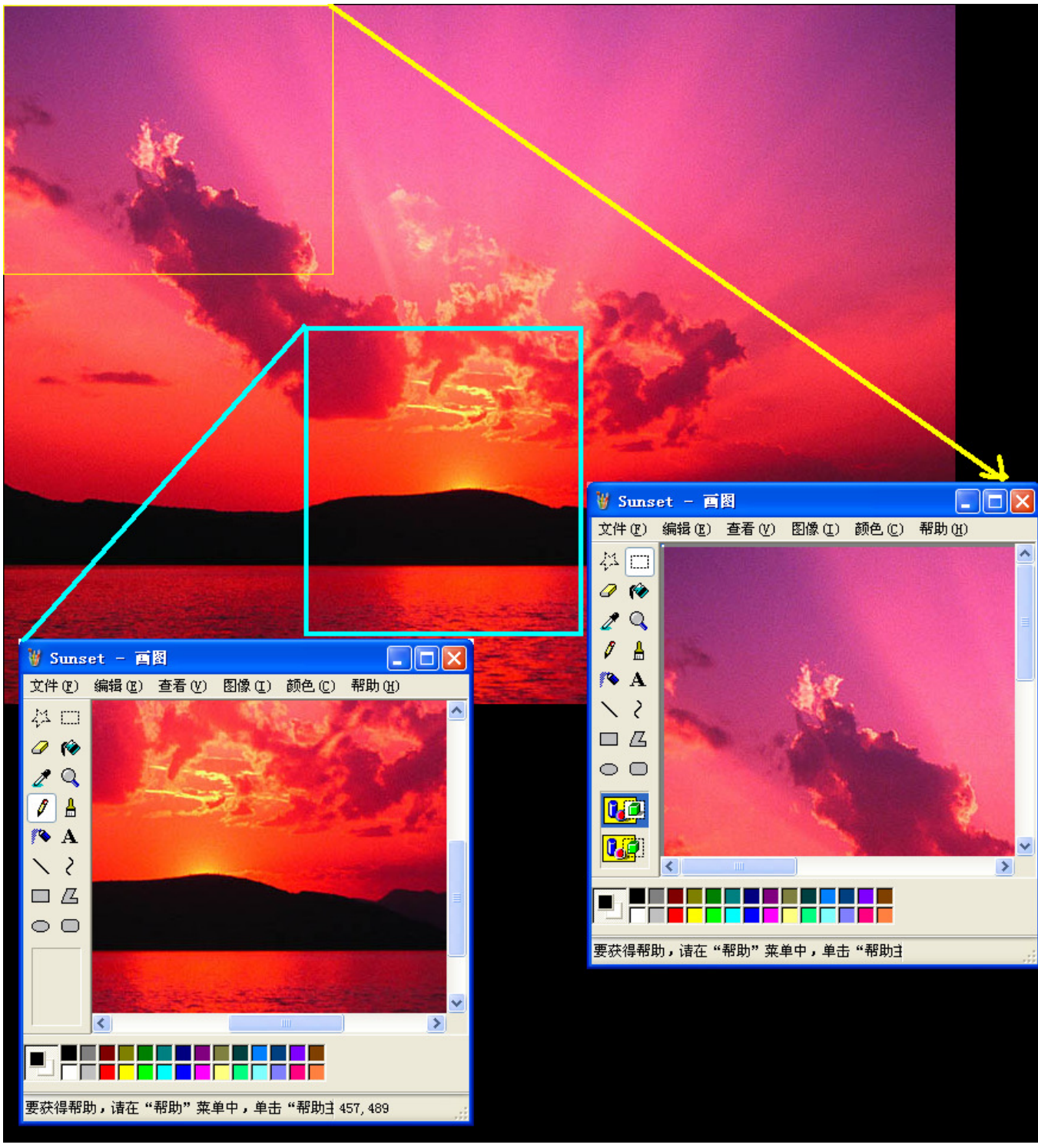
也就是说，我们的软件的映射模式如果不同，那么可能我们看到的结果就有所不同，我们这一讲也不会与大家详细地讲，当我们用 MM_TEXT 时，视图与视口的坐标之间是等距的关系，我们可以调整视口之间的坐标，或者视图之间的坐标，使得之间不同。

如果要看到相应的情况，我们只需要在我们的软件中画几个图，然后看打印预览，就可以看到效果了。

是不是图形变小了？哪是因为激光打印机的像素点比我们的屏幕上的像素点要小得多的缘故，当我们以 MM_TEXT 去画图时，其实是一个以点的形式去画图，那么就看这个点在现实的设备中的大小了，可以想像，这一定是不数学中的理想的点，如果用那种理想的点去画图的话，那我们将什么也看不到，因为理想的点是没有宽度与高度的，而现实的点却有。

那么我们就在下一章开始，于大家真实在我们的ＶＣ环境下去实践一些东东，看看这些不同模式之间的关系


无标题 - Draw
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪


Sunset - 画图
文件(F) 编辑(E) 查看(V) 图像(I) 颜色(C) 帮助(H)
要获得帮助，请在“帮助”菜单中，单击“帮助
Sunset - 画图
文件(F) 编辑(E) 查看(V) 图像(I) 颜色(C) 帮助(H)
要获得帮助，请在“帮助”菜单中，单击“帮助
457, 489

**微软积木一一九：模式设置**

/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：
*/
    啪!!!

    这一块啊，在皇上那儿叫龙胆（注：牛刀也不知是不是这个胆啊，管不了那么多了），在药铺这叫压方，在老师那，这叫戒尺（比我们这长，窄），在说书的那儿叫穷摔，也就醒木，在我们这啊，啪!!!!
    叫废话，呵呵，叫鼠标。

    今年的大事就是多，有好的也有坏的，事情就是这样，没有人知道谁打彩票中得不中不了五百万，也没有谁知道谁喝了凉水塞了牙，人世间，有好人也有坏人，故事里的事说是也是说不是也不是（哪挨着哪儿啊）

    闲话少叙，咱们书归正传（又来了）

    其实说实话，这一讲本打算大量贴图，因为这一讲的图形很好看，就是不把秃子看，秃子头上的炸弹，出了人命怎么办（哎，实在打不住，大家不要笑哟，大家有什么记忆犹新的顺嘴段子，也可以贴出来玩玩嘛，有些东西都快变成儿时的记忆了，因为现在的人都不太在乎这些历史的精华，而主要事情，可能就是上网，电脑，玩游戏，有谁还记得过去还有跳坊，警察抓小偷，地雷快爆炸与打陀螺，玩竹蜻蜓）。

    嗯，好像上面的话有点伤人，不管怎么都是不由自主，想别大家也不会介意的，牛刀就是脚脖子扣铃铛，响到哪里脚到哪里。

    牛刀的本意不是说大家是秃子（哎，说实话，这种事情啊，是越解释越糟，不解释了），而是说，大家一定要上机实践，把所有的知识领会贯通，因为我们这一讲的内容，主要需要了解，而不是能够死记硬背就运用的东西。

    那么好，我们不在 Draw 软件中做事了，我们新建一个 MFC 工程，牛刀将之定为 ModeTest ，你想定为什么名子定什么名子，那牛刀管不着，呵呵（对了，以上的串行化部分就已经结束了，以后我们想要保存就提一下，串行化一下就行了，微软的 MFC 会为你的保存工作做得相当完美）。

我们先来看看几种映射模式——
固定比例映射模式

映射模式　　逻辑单位

MM_LOENGLISH　　0.01 英寸
MM_HIENGLISH　　0.001 英寸
MM_LOMETRIC　　0.1mm
MM_HIMETRIC　　0.01mm
MM_TWIPS　　1/1440 英寸

可变比例映射模式
MM_ISOTROPIC
MM_ANISOTROPIC

我们先谈谈这种映射模式吧，其实最原始的映射模式就是 MM_TEXT 映射模式，就是我们常用的，并且是唯一的像素映射模式（对了，大家上面新建一个工程时，第六步还是 CScrollView 啊，别弄错了，弄错了自己去改（注意啊，千万不要以为不能改，根牛刀看了这么些讲了，难道还对微软的环境有神秘感？呵呵，当然喽，改的时候，在头文件中，只有一处，改过来就行，在 CPP 文件中，有好多处，但最好使用查找/替换，（注意了，在 CPP 文件中 按 Ctrl+H，不要按上面那个，我们只在当前页找，找 CView 改为 CScrollView 请注意，一定要全字匹配，不然你改后的乱子可就大了，呵呵）。

当然，改过以后，我们的程序可以运行，但一运行就会报错，大家可以为 View 类添加一个 OnInitialUpdate 的虚函数（注意了，不是消息响应，如果有那当然更好），然后在其中定义一个 CSize ，设置一下我们的视图大小，使用函数——

SetScrollSizes

其实这个函数有四个参数，第一个参数是映射模式，后面分别是——大小，按箭头移动的大小，按箭头与滑柄之间的空区移动的大小，那最后两个都有默认值，那么我们还设置为 2000

```
void CModeTestView::OnInitialUpdate()
{
	CScrollView::OnInitialUpdate();

	CSize sizeTotal;
	sizeTotal.cx = sizeTotal.cy = 2000;
	SetScrollSizes(MM_TEXT, sizeTotal);
}
```

于是，我们的程序一运行，就有滚动条喽。

当然，我们的程序中什么都没有，那么既然没有，我们就去画一些线（在 OnDraw 中）

```
void CModeTestView::OnDraw(CDC* pDC)
{
	CModeTestDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);
```

```
        pDC->MoveTo(-2000,0);
        pDC->LineTo(2000,0);
        pDC->MoveTo(0,-2000);
        pDC->LineTo(0,2000);
}
```

请大家注意，由于我们这一讲专门讲——视，所以我们上面的文档获得的那个函数和下面一句都可以删了，和我们没关系。

你可能纳闷了，为什么牛刀用了 -2000 呢？你一会儿就明白了，你运行程序，边上的线可能不太清楚，没关系，我们再加上一些坐标显示——

```
void CModeTestView::OnDraw(CDC* pDC)
{
        int i;
        CString str;
        pDC->MoveTo(-2000,0);
        pDC->LineTo(2000,0);
        pDC->MoveTo(0,-2000);
        pDC->LineTo(0,2000);
        for(i=-2000;i<2000;i+=200)
        {
                str.Format("%d",i);
                pDC->TextOut(0,i,str);
                pDC->TextOut(i,0,str);
        }
}
```

当然，如果你觉得有必要，你去画格子也可以我们马上看看有没有必要，（对了，大家千万不要复制牛刀的代码，请你一个一个地敲上电脑，并且在敲的过程地不断地理解，牛刀一般加添加的方法，并且每次都贴完整的代码，其实你的打字量很少的。）

```
void CModeTestView::OnDraw(CDC* pDC)
{
        int i;
        CString str;
        //牛刀来一个怪的：
        pDC->SetWindowOrg(-1000,-1000);
        pDC->MoveTo(-2000,0);
        pDC->LineTo(2000,0);
        pDC->MoveTo(0,-2000);
        pDC->LineTo(0,2000);
        for(i=-2000;i<2000;i+=200)
        {
                str.Format("%d",i);
                pDC->TextOut(0,i,str);
```

```
                pDC->TextOut(i,0,str);
        }
}
```

大家再去看看效果，怎么样？这就证明着我们的 SetWindowOrg 是设定我们逻辑坐标系的左上角坐标，看图——

<图一>

我们的视口坐标原点（注意，我们说的坐标原点总是指左上角，而不是坐标系原点），也可以使用 SetViewportOrg 去设置，那好，我们为了与视图区别，我们响应一个鼠标移动的消息映射——

```
void CModeTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        int i;
        CString str;
        CClientDC dc(this);         //取得  dc
        CRect rect;
        CPen pen(0,1,RGB(255,0,0));//为了显示区别，我们使用红笔在视口中画
        CPen *pOldPen=dc.SelectObject(&pen);
        GetClientRect(&rect);
        //注：为了设置视口原点，我们定义一个成员变量 m_ptOrg，并且在构造函数中初始化为 0

        dc.SetViewportOrg(m_ptOrg);
        dc.MoveTo(rect.left,0);
        dc.LineTo(rect.right,0);
        dc.MoveTo(0,rect.top);
        dc.LineTo(0,rect.bottom);
        for(i=min(rect.left,rect.top);i<max(rect.right,rect.bottom);i+=200)
        {
                str.Format("%d",i);
                dc.TextOut(i,0,str);
                dc.TextOut(0,i,str);
        }
        str.Format("x=%d,y=%d    ",point.x,point.y);
        dc.TextOut(0,0,str);
        dc.SelectObject(pOldPen);
        DeleteObject(pen);
        CScrollView::OnMouseMove(nFlags, point);
}

void CModeTestView::OnLButtonUp(UINT nFlags, CPoint point)
```

```
{
        //在这里，我们改变视口原点
        m_ptOrg=point;
        CScrollView::OnLButtonUp(nFlags, point);
}
```

大家去试试看，我们不管怎么，可以改变视图原点，也可以改变视口原点，但大家看我们鼠标在屏幕上移动的设备坐标永远是右上角为(0,0)，这一点永远改变不了。

是不是似乎明白了，似乎又更糊涂了，似乎我们的视口与我们的鼠标之间也隔着一层玻璃，而我们问题在玻璃外面比比划划？还有一个就是，我们视口的变换似乎与视图的变换正好相反，这是怎么回事？

牛刀不服，在视口原点定义下面去打点——

```
dc.SetPixel(point,RGB(255,0,0));
```

然后去体会体会效果，（当然是鼠标单击屏幕之后喽）

于是，我们似乎体会到一些什么，随着我们鼠标的移动，我们鼠标打出的点不我们鼠标原来的位置了，出现了鼠标坐标的漂移。为什么会这样？

原来啊，我们还有一套坐标系——叫做世界坐标系，我们知道，我们的设备坐标系是不可移动的，而我们的世界坐标系同样是不可移动的，为什么我们下面再说。

那么我们的世界坐标系在哪里呢？我们怎么使用呢？

其实我们一直是在使用这个世界坐标系（注：牛刀有些观点与能与你或者某些书本不同，考试以书本为准，理解以牛刀为准，呵呵），大家上面明明定义的逻辑坐标系是从 (0,0) 到 (2000,2000) 那大家为什么能在 (-2000,-2000) 的位置画图呢？为什么不报错呢？因为我们其实就是在世界坐标系中画图，世界坐标系与普通坐标系一样，也是四个方向无限延伸的，只不过我们的世界坐标系 y 轴是向下递增的，仅此而已。

于是，我们的世界坐标系其实是不能动的，当然，设备坐标系也是不能动的了，可惜的是，我们只能在设备坐标系中观察，于是，我们设置视口时，是设置视口在窗口中的位置，例如我们将其移动到 (100,100) 处，我们就看到坐标原点在我们屏幕上的 (100,100) 处了，然而，我们的视图设置的坐标却是视图在世界坐标系上的坐标原点，于是，视图中想看到世界坐标系的原点，则必须将自己的左上角调到世界坐标系的左上角那一大块区域，而那一大块区域却只能是负的，于是我们在定义那个视图坐标时，就只能用负值了。

什么意思呢？也就是说，我们如果拿尺去量物体的长度，物体不动，那么我们的尺每测量一次就要向右移动，而如果我们的尺子放那儿不动，但用想要量的物体向尺子上挨，那么特别就需要向左移动，这其中的关系有点微妙，大家用心

仔细去思考一番，也就是说，我们的设备坐标与我们的世界坐标是不能移动的，所以只有移动视图或视口，而视图是世界坐标系的观察窗口，所以我们如果想看到世界坐标系的原点，我们的视口左上角就要设置为世界坐标系的负值，而我们的视口对于我们的窗口来说，窗口却不能移动，于是，我们如果移动视口就只能向窗口的右下角移动，才能看见视口的原点，于是乎，我们设置视口的值就必须是正值。

如上图所示，我们的窗口与视口发生了不同步，于是，我们就必须将鼠标的值从窗口转换到视口，怎么转换？很简单啊——

DPtoLP

不要告诉我们忘了，呵呵，于是，在我们鼠标移动前面的打点处，加上一个函数——

```
dc.DPtoLP(&point);
dc.SetPixel(point,RGB(255,0,0));
```

其实说实话挺简单，但只可惜很容易被绕进去，那么视口与视图之间当然也是通过那个函数去转换的喽，世界坐标系呢？显然也是一样。

于是乎，我们可以放弃一切的理论与条条框框，在合适的时候，使用 DPtoLP 或者 LPtoDP 去转换我们的坐标，当然，只要在使用 SetWindwOrg 或者使用 SetViewportOrg 后，都需要做相应的设置。

说实话，如果你要在做软件中将这些关系绕清楚，那还真的需要不断地去调试自己的程序，否则你就必须对各个坐标系之间的关系的理论要万分熟悉。

行了，这一讲后半部分与大伙聊天，因为这一讲如果真的想理解，真的需要费时间，不断地上机，不断地理解，不断地调试，否则很难吃透。

大家做程序其实就是一种思路的过程，我们做程序的得心应手就是建立在我们对我们需要操作对象的充分了解的基础上的。

说实话，坐标变换不是窗口与视口的专利，窗口与视口的变换只是坐标变换的皮毛，甚至只用什么 DPtoLP 或者 LPtoDP 就解决了，而事实上的坐标变换十分复杂——

坐标变换分为二维变换与三维变换，二维变换中有伸缩变换，错切变换，旋转变换，如果有必要还可以进行一些特殊的变形。

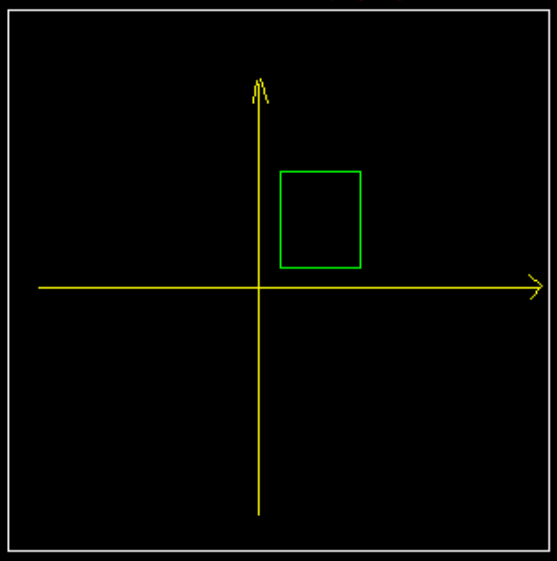
三维坐标变换也是伸缩变换，错切变换，旋转变换（注：都是三维上的喽），三维的还的投影变换，透视变换等等，将三维的图形投影到二维的空间上去。

呵呵，说实话，真的三维的东东，我们是没办法在电脑上看的，必须作投影变换，如果我们我们工程技术上用，则需要使用正投影，方便测量与计算，如果我们需要视觉上的享受，则透视投影是最佳的了。

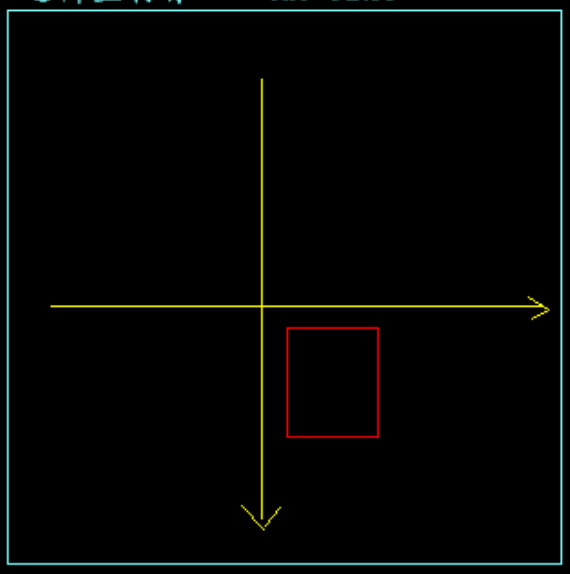
其实很想与大家将这些三维技术聊一聊，不过如果真的那样聊下来，相信我们这一专题的结束将是遥遥无期了，呵呵，我们就此打往，以后再与大伙闲聊。

啪!!!

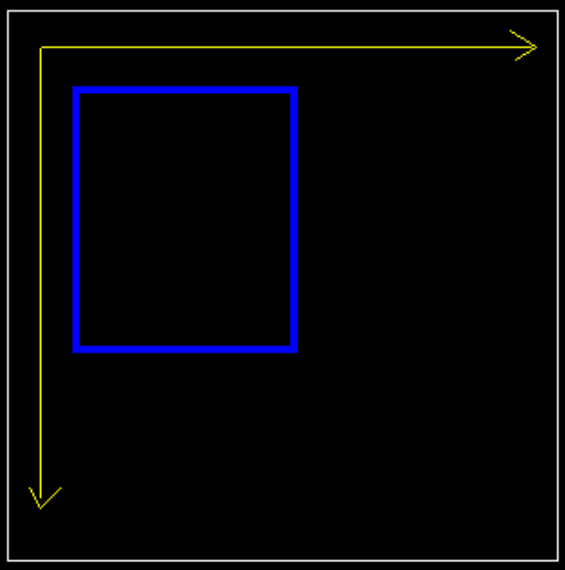
常规坐标系——自然科学中使用



逻辑坐标系——MM_TEXT



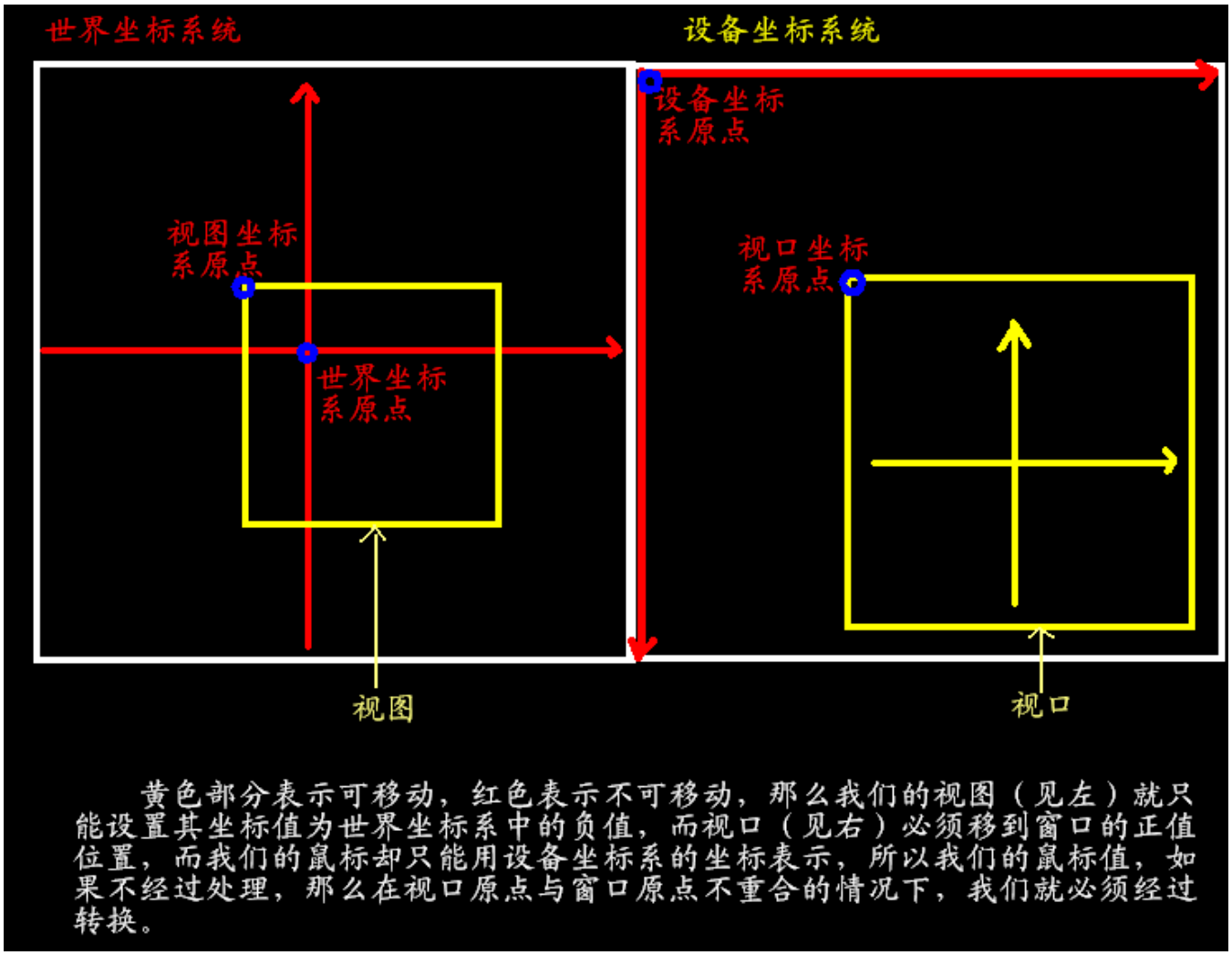
设备坐标系——窗口显示MM_TEXT



请大家看上面与左边的坐标系，我们左上角的 常规坐标系 在电脑中必须首先变成上面的视图形式，然后由左边的设备坐标系在视图中截取一块区域显示至于获取哪一块呢？那就需要设置视口在视图上的坐标喽，于是，我们的 OnPrepareDC 就做了这样的工作，根据我们的滚动条去设置视口所截取的视图部分的位置。

一般情况下，我们的视图的左上角也是（0,0），上面是我们用 SetWindowOrg 改变以后的情况，我们由于视图与视口都是向右向下增加的，所以左上角的位置都是负的，所以我们使用了 pDC->SetWindowOrg(-1000,-1000)

从而使得我们可以显示出坐标，当然喽，我们的视口的坐标也可以改变，那就是使用 SetViewportOrg，我们下面会与大家一试便知




**微软积木一二〇：固定模式**

```
/*
    版权信息
作者：牛刀小试
首发论坛：http://0ginr.com/bbs/
首发时间：二〇〇七年七月七日
```

*/

可能你已经觉得有些眩晕了吧？呵呵，这没办法，该叫你思考的地方，真的是回避不了，该叫你体验的地方，你不去亲自地体验一把，那也很难理解，牛刀总觉得，去实践比什么都重要，拿牛刀写这教程来说吧，说实话，不写这东西，牛刀也是很喜欢偷懒的，但既然写了，好多的东西搞得就比原来要明白得多。

再一个重要的事实，那就是——好书不厌百回读，真正的一本好书，你每读一遍都会学到比以前更多的知识，真的，人的知识结构不同了，理解起问题来是有差别的。

上一讲，我们主要与大家讲了 MM_TEXT 模式下的一些映射方法，说实话，那是最简单的，也是最初级的，我们一般在不设置模式的情况下，都是用的那一种模式，那么这一讲，我们来看几个固定模式——
固定比例映射模式

| 映射模式 | 逻辑单位 |
| --- | --- |
| MM_LOENGLISH | 0.01 英寸 |
| MM_HIENGLISH | 0.001 英寸 |
| MM_LOMETRIC | 0.1mm |
| MM_HIMETRIC | 0.01mm |
| MM_TWIPS | 1/1440 英寸 |

什么意思呢？大家上还记得牛刀和大家说过，打显示与打印预览的问题？打印机的点小，我们显示器的像素点大，所以我们同样的一个幅图，在屏幕上显示的效果与打印机上的显示效果是不同的，于是，人们一直在找到一种折中的方案，后来这种方案竟然被人找到了，后来被称之为——所见即所得，可以理解啊，在屏幕上看到的是什么样，在任何一个输出端输出的就是什么样，不管是激光打印机，还是针式打印机。

那是怎么做到呢？那就是坐标的模式不同。

什么意思呢？虽然我们的显示器显示效果与打印机打印效果不同，但如果规定一个单位，不同相同了？话不好说，咱们举个例子吧。

我们假如有一个映射模式是以 0.1mm 做为单位的，即上面的 MM_LOMETRIC 那么我们的 1 就是代表 0.1 mm ，用这样的坐标。

可能你觉得，不是一样的吗？当然有不同，不同点就是各种输出端对此的种模式的技持。

举个例子（注：牛刀为了说明方便，不是事实中的例子啊）：

假如我们显示器的像素点是 0.01 mm 而激光打印机的像素点是 0.001mm ，那么当在这种映射模式下，我们的屏幕如果要显示一个为 5 单位的效果，那么在屏幕上就用 50 个点去显示， 而打印机如果要显示 5 个

单位则用 500 个点去显示，什么意思明白了吧？也就是说，每台输出设备是肯定知道自己的一个像素能够显示的的大小的，这是每一台设备的固有值，永远也改不了的，所以得到一些数据时，进行相应的点扩充，这样一来，如果在屏幕上显示，我们拿尺一量，也是 0.5mm 如果在打印机打出来，也就 0.5mm 这样一来，我们的屏幕上看到的效果就与打印机上打印机来的效果一样了，这就是固定比例映射模式的概念。

当然喽，也正因为如此，我们上面几种映射模式其实效果是一个样，我们先来用两个试一试，然后我们就其中的一个来重点介绍，其他的大家知道了就行，这是根据需要来用的，你做一款软件只能有一个映射模式。

好，我们先在设置滚动大小的函数中将映射模式改了，先改为 MM_LOMETRIC 吧：

```
SetScrollSizes(MM_LOMETRIC, sizeTotal);
```

第二个，我们还设置为 2000,2000 ，于是，我们再在 OnDraw 中画些图：（请注意喽，固定比例映射模式与我们普通坐标系一样，x 轴向右为正，y 轴向上为正，所以我们画图就需要注意喽——（我们需要在第四象限画图，因为我们坐标原点还是在右上角，呵呵）

```
void CModeTestView::OnDraw(CDC* pDC)
{
        pDC->Rectangle(CRect(20,-20,1000,-1000));          //必须在第四象限画图，映射模式与我们数学坐标系一样
}
```

行，大家先看着效果，牛刀去改一改映射模式——

```
SetScrollSizes(MM_HIMETRIC, sizeTotal);
```

大家看两种不同映射模式的效果——

<图一>

效果明显不同，右边一个每一个单位是左边的十分之一，当然那个矩形看上去就要小得多，那么大家在左边那种大的矩形时，去看看打印预览的效果——

注意了，打印预览一打开是缩小预览的，将其放大，与我们在屏幕上看到的效果几乎差不多了吧？

<图二>

所以我们一般正常情况下，如果是为了日常应用，那么 MM_TEXT 就够了，那个与我们屏幕上的像素点是一一对应的，而如果大家想将其显示效果统一，那么就使用上面的固定比较模式的效果，看着情况用，估计大家不会用英寸，呵呵，别扭啊，那么也就是说，我们正常都是用 0.1mm 或者需要显示精细的话，用 0.01mm 就行。

```
void CModeTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CString str;
```

```
        str.Format("x=%d,y=%d    ",point.x,point.y);
        CClientDC dc(this);
        dc.TextOut(0,0,str);


        CScrollView::OnMouseMove(nFlags, point);
}
```

大家将上面的代码敲进去，看看我们鼠标得到的坐标，似乎还是按 MM_TEXT 显示的，牛刀不服，再用打点函数去打点——

```
        dc.SetPixel(point,RGB(255,0,0));
```

不错，没有改变，为什么？呵呵，缺函数啊，再看——

```
void CModeTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CString str;
        str.Format("x=%d,y=%d    ",point.x,point.y);
        CClientDC dc(this);
        OnPrepareDC(&dc);
        dc.TextOut(0,0,str);
        dc.SetPixel(point,RGB(255,0,0));

        CScrollView::OnMouseMove(nFlags, point);
}
```

呵呵，一加 OnPrepareDC(&dc); 牛刀发现，牛刀打的点不见了，哪儿去了？聪明的你能想像得到吗？如果用一个转换函数呢？

```
void CModeTestView::OnMouseMove(UINT nFlags, CPoint point)
{
        CString str;
        str.Format("x=%d,y=%d    ",point.x,point.y);
        CClientDC dc(this);
        OnPrepareDC(&dc);
        dc.DPtoLP(&point);
        dc.TextOut(0,0,str);
        dc.SetPixel(point,RGB(255,0,0));
        str.Format("dx=%d,y=%d    ",point.x,point.y);
        dc.TextOut(800,0,str);
        CScrollView::OnMouseMove(nFlags, point);
}
```

这下就正常了，我们转换后的坐标显示变成正常的了，可惜有一点遗憾，那就是中文显示不好，不知道是什么原因？

于是，我们知道，我们的坐标其实是一个与数学坐标系一样的坐标系，于是，我们去画一些函数曲线试试，肯定很好玩的——

```
void CModeTestView::OnDraw(CDC* pDC)
{
```

```
        pDC->SetWindowOrg(-1000,1000);//注意映射模式，别要脑筋在这儿打滑，呵呵
        pDC->MoveTo(-1000,0);
        pDC->LineTo(1000,0);
        pDC->MoveTo(0,-1000);
        pDC->LineTo(0,1000);
        pDC->MoveTo(-1000,1000);
        for(float i=-1000;i<1000;i++)
        {
                pDC->LineTo((int)i,(int)(((i/1000)*(i/1000))*1000));
        }
}
```

这是一个标准的二次函数啊，呵呵，看效果——
<图三>

当然，你可以很容易地修改函数公式，看看不同函数的不同效果。

注意了，如果用到了一些库函数，需要使用数学库 math.h 别忘了包含头文件。

·思考：如果我们需要将视口原点调到屏幕的上可以看得见的地方，我们的调整坐标值应该是什么情况？
A 两个全正
B x 正，y 负
C x 向,y 正
D 两个全负

呵呵，牛刀还不容易这样出题目给大家做啊，如果上面的一题你做对了，那么说明你对牛刀以前说的就都理解了，如果你不能说对，那证明牛刀以前说得还不好，需要改进，当然，答案不告诉大家，自己上机去试。

说实话，知识是需要理解的，你理解了，那知识是你的，你不理解，那么那种知识永远与你无缘，可能牛刀的说法有点片面，但理解在学习的过程中的确至关重要。

好了，大家已经将上面的固定模式理解了吗？如果理解了，牛刀也乘机将可变比例的映射模式也讲了吧，呵呵，原来准备单独列一讲的，如果现在这样看，完全没必要了，原因是，我们并没有将问题搞复杂。

很可惜的是，比例模式是不能在 CScrollView 视图中设置的，呵呵，于是，我们新建一个工程（注：这次使用普通视图了，CView 类）

可变比例映射模式有两种 MM_ISOTROPIC 和 MM_ANISOTROPIC

前一种 纵横比是 1:1 的，也就是说，在这种情况下，我们画一个圆，那就是一个圆，而后一种是不固定纵横比的，可以随意，也就是说，在后一种来说，可能一个圆就变成椭圆了，如果一处理一张圆脸，就会变成一张长脸了（想说驴脸的，太太太太不文雅了，呵呵）。

那么前面一种，大家自己去玩玩好了，牛刀带大家做一个后面一种的程序，去体会体会（请注意，这里因为比例不固定，所以我们要设置世界坐标系在视图中的范围，与窗口在视口中的范围，因为这次我们的图像是要做变形的，没有范围，变形则无从谈起，好了，做程序时具体与大家再说）

```
void CMFCTestView::OnDraw(CDC* pDC)
{
        CRect rect;
        GetClientRect(rect);//得到客户区的矩形，我们设置的范围将充满客户区
        pDC->SetMapMode(MM_ANISOTROPIC);          //这是设置我们的可变比例的模式，在 CScollView 中，父类为我们做了，我们传个参数过去就行
        pDC->SetWindowExt(1000,1000);          //设置世界坐标系的范围，也就是说，我们如果在其他范围中画图，我们的窗口就显示不出来了，固定比例就没必要了，因为那所用的坐标长度一样，这儿不一样了
        pDC->SetViewportExt(rect.right,-rect.bottom);//设置视口的范围我们在世界坐标系中的 1000,1000 投射到这儿就是 rect 的宽度与高度，这样一来，就不是固定比例了
        pDC->SetViewportOrg(rect.right/2,rect.bottom/2);          //将窗口的原点设置在正中，准备画图
        pDC->Ellipse(-500,-500,500,500);          //我们在 (-500,-500) 到 (500,500) 两点构成的矩形范围内画圆
}
```

这一个代码我们不贴图了，大家将窗口变化看看，仔细体会上面每一个代码是什么意思。

GetClientRect 获得当前窗口客户区的大小
SetMapMode 设置当前模式
SetWindowExt 设置世界坐标系在视图中的投影区域（范围）
SetViewportExt 设置视口在窗口上的投影范围

大家注意了，我们视口与窗口投影与视图与世界坐标系之间投影的不同。经过上面设置后，我们的视图大小与视口大小也就固定了，然后所画的那一个矩形是在世界坐标系中画的，但根据视图与视口的大小进行放缩，然后在窗口中投影上来，也就是说，我们的窗口相对于世界坐标系来说，不管大小如何变化，从窗口左边到右边大小都被设置为 1000 ，从上面到下边也是 1000 于是，我们用 500 为半径画个圆，当然就充满着整个窗口喽

<图四>

图有点绕是不是？原因就是，我们的可变比例的映射模式下， x 轴正方向与 y 轴正方向都是可以设置的，也就是说，可以用负号将主轴方向给反过来，一旦反过来了，我们下面的视口坐标是如何设置的就显而易见了，大家绕看看，能绕过来当然更好，绕不过来的话就别去想了，呵呵，有些技术可能是你这一辈子都用不到的，理解得了便好，不能理解也没关系。

当然，上图的理解只是牛刀个人的见解，如果牛刀的理解错了，或者你有什么其他好的见解，还希望与牛刀详加讨论，理越辩越明，呵呵。

别看以上的代码不起眼，真的理解好了，做程序事半功倍，理解得不好，做程序会两全事倍功半的，呵呵。（对了，这两个成语哪个好啊？是不是牛刀搞混了，呵呵）

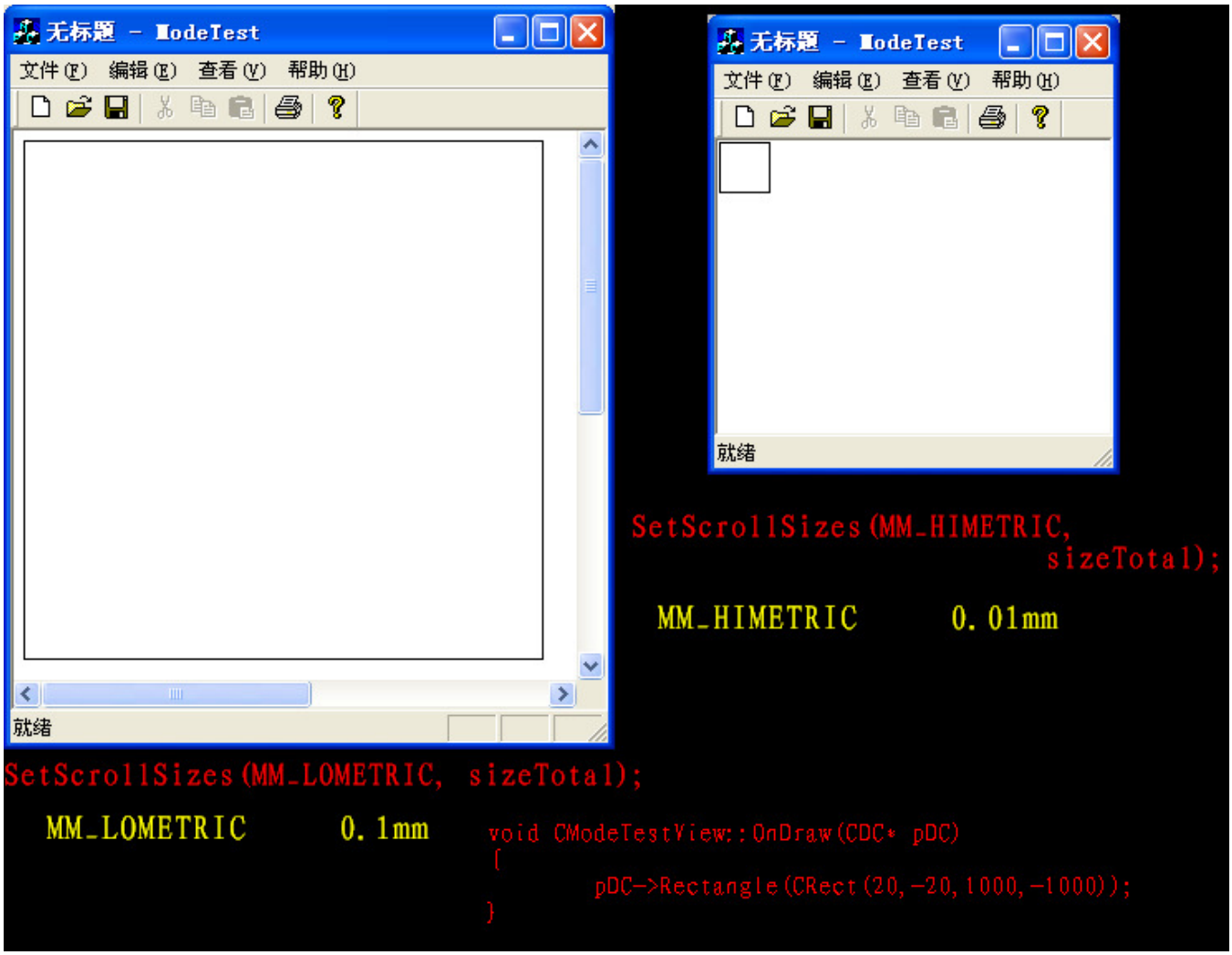
大家试着将模式改为 MM_ISOTROPIC 看看，怎么样？你的圆一直是圆了吧，并且能够在窗口中一直显示，这就是 x=y 的不等比例变换的模式，在这种模式下操作图像不变形，这是最大的优势，其他与我们上面的效果一样。

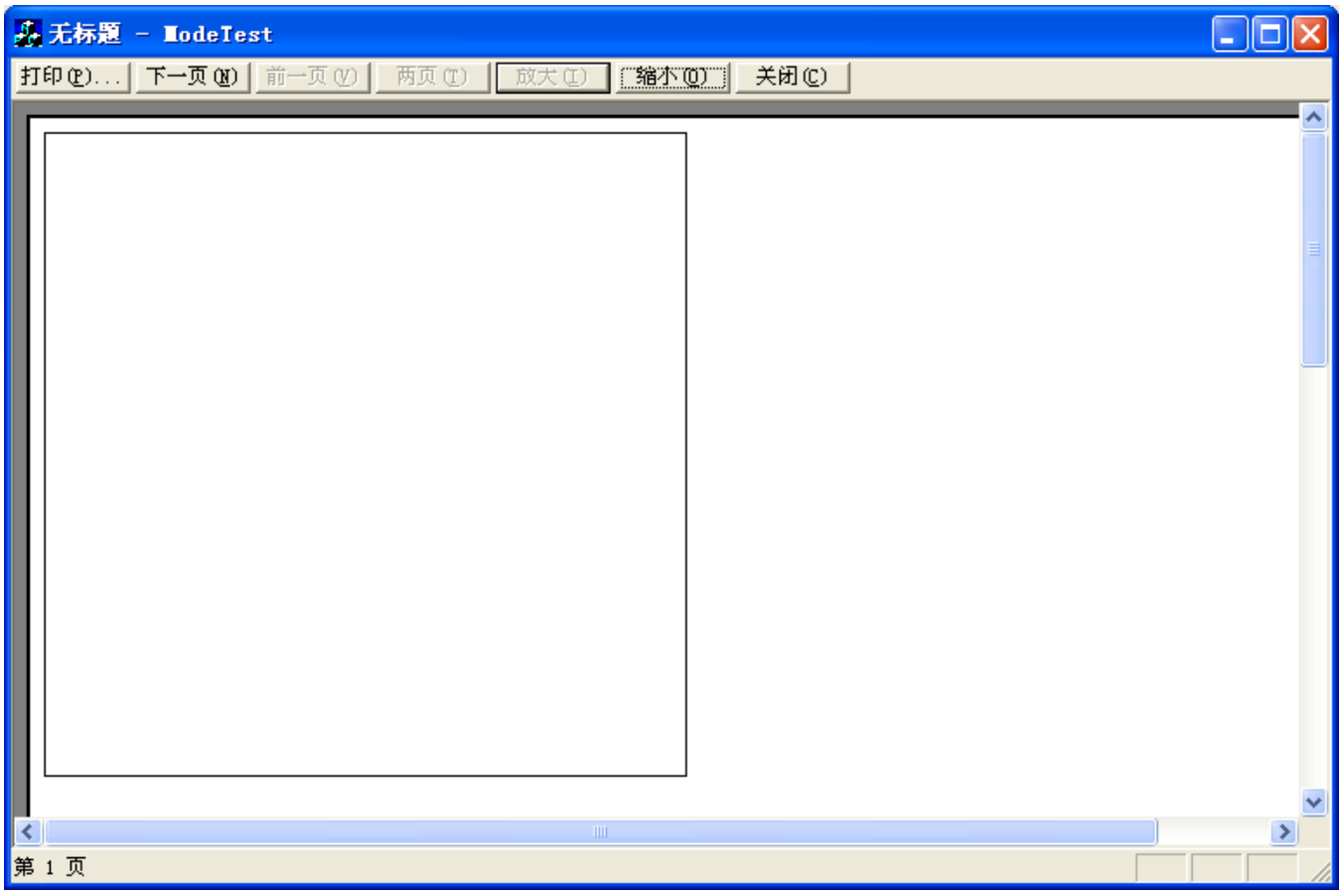
坐标变换与模式设置部分我们到此就结束了，其实如果这东东展开来说，一年半载也不见得说得透，模式设置得如果好再加上一些特殊效果，其实可以做出很厉害的效果出来的，我们与大家做一个小软件试试啊，当然这种效果如果有 3D 技术的技术，那效果就相当棒了，下面我们就用一段程序作为我们的结束——

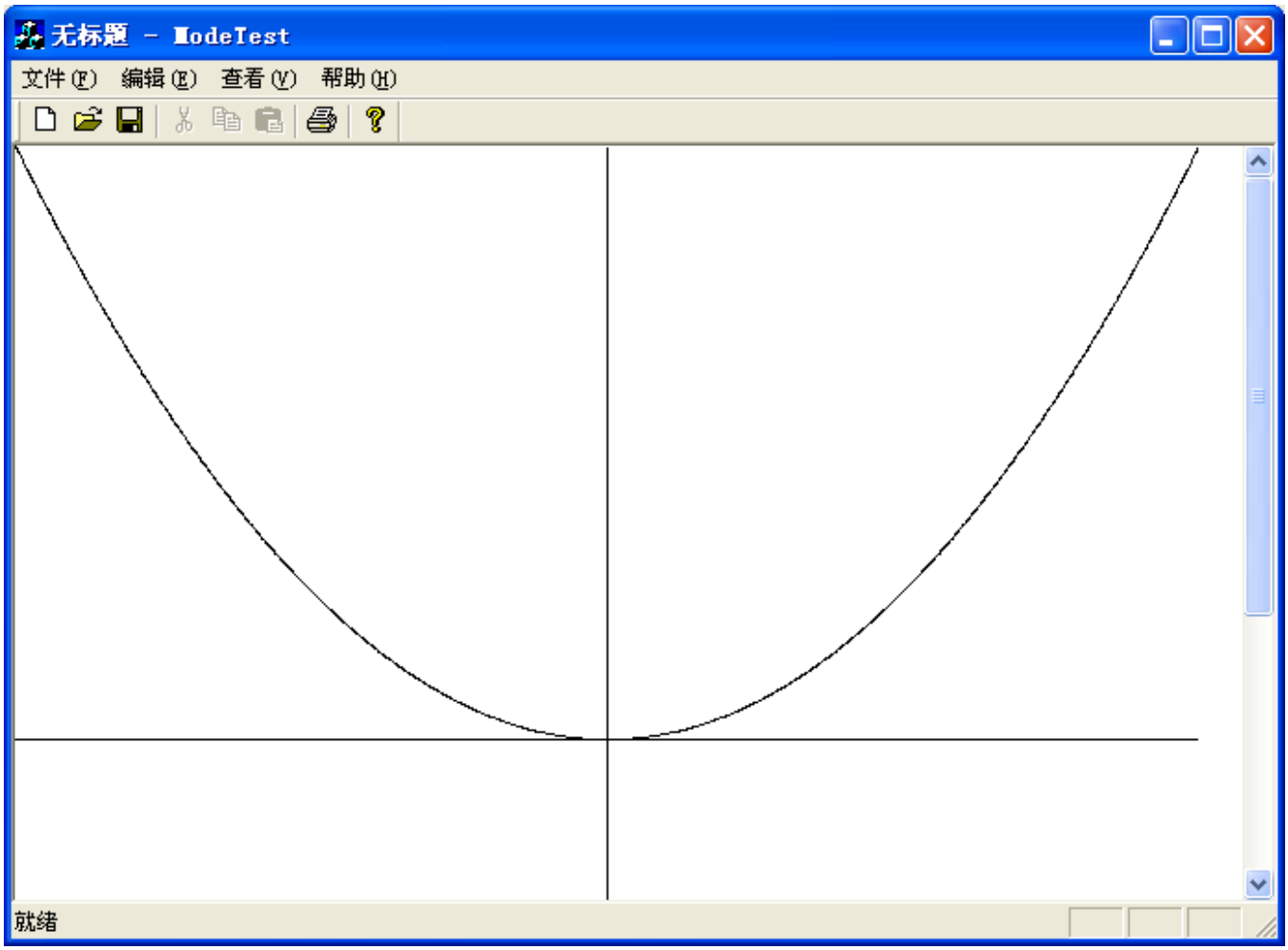
```
float a,l,x=200,step=10;
for(int i=-500;i<500;i++)
{
        for(int j=-500;j<500;j++)
        {
                l=sqrt((i-x)*(i-x)+j*j);
                a=sin(l/step);
                l=sqrt((i+x)*(i+x)+j*j);
                a+=sin(l/step);
                a+=2;
                a*=64;
                pDC->SetPixel(i,j,RGB((int)a,0,0));
        }
}
```

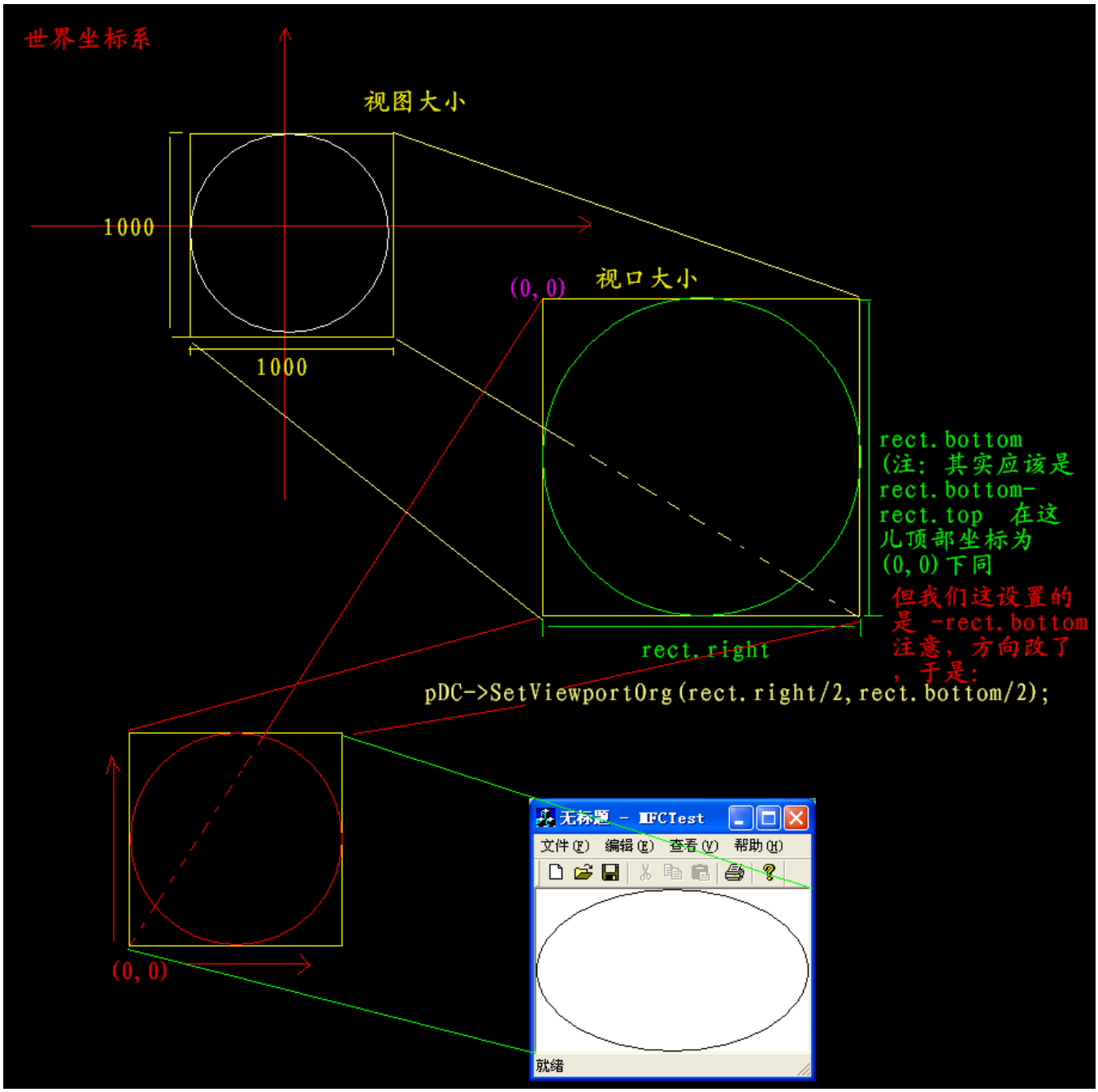
注意了，放在上面的那个可变比例模式下面去做。

啪!!! 咱们这一讲就到此结束。


无标题 - ModeTest
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪
SetScrollSizes(MM_LOMETRIC, sizeTotal);
MM_LOMETRIC 0.1mm
void CModeTestView::OnDraw(CDC* pDC)
{
pDC->Rectangle(CRect(20,-20,1000,-1000));
}
无标题 - ModeTest
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪
SetScrollSizes(MM_HIMETRIC, sizeTotal);
MM_HIMETRIC 0.01mm


无标题 - ModeTest
打印(P)...
下一页(N)
前一页(V)
两页(T)
放大(I)
缩小(O)
关闭(C)
第 1 页


无标题 - NodeTest
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪


世界坐标系
视图大小
1000
1000
(0, 0)
视口大小
rect.bottom
(注：其实应该是
rect.bottom-
rect.top 在这
儿顶部坐标为
(0,0)下同
但我们这设置的
是 -rect.bottom
注意，方向改了
，于是：
rect.right
pDC->SetViewportOrg(rect.right/2,rect.bottom/2);
(0, 0)
无标题 - MFCTest
文件(F) 编辑(E) 查看(V) 帮助(H)
就绪