第一章　从 DOS 轻松转到 Windows

本章分成三部分。

1. 什么是 EasyWin，介绍 EasyWin 的概念。从一个 EasyWin 的 DOS 风格程序介绍 DOS 和 Windows 的不同，作为对 DOS 时代的简单介绍。
2. 介绍如何用 EasyWin 作为你进步的阶梯。如何从 Windows 中获得各种句柄参数，添加菜单，处理消息。这些高级技术你可以轻松地应用于其它方面。
3. 程序的调试是比较重要的一方面，一般比较理想的方法是用两个屏幕来调试，一个用作操作，另一个用来显示各种调试信息，显示程序运行中各种变量的值。这部分介绍的如何使用 EasyWin。从而可以只用一个屏幕达到同样的效果。

1.1　在 Windows 下运行 DOS 程序

1.1.1　Windows 与 DOS 应用程序的比较

一个程序员刚转到 Windows 下时，会对 Windows 的消息驱动风格感到难以掌握。在 DOS 下编程序时，一个程序设计者完全决定了什么时候接收用户的输入，当等待用户输入时，显示一串信息，然后等待，用户输完后，程序接着向下执行。当程序要显示信息时，则向屏幕发送一字符串，根本不必管字符在屏幕上的显示位置。程序的执行基本上是顺序执行的。而在 Windows 下，程序接收外部的消息不再是在自己想接收的时候，程序可以在任意的时候接到任意的消息，这样，就要求程序要保留自己执行状态的信息，以便在接到下一条消息时知道自己处在程序的什么地方。

从 DOS 的简单输入输出接口、顺序执行的程序方式过渡到 Windows 下的图形用户接口、消息驱动方式，实际上也反映了计算机科学的发展和应用范围的扩大。在计算机应用的早期，计算机的主要用途是数值计算机，用户接口用于输入用户要计算的数，输出计算后的结果，这样简单输入输出接口即可满足要求。顺序执行的程序设计方式也恰好反映了数值计算的程序设计方法。但随着计算机的发展，计算机的应用范围不断扩大，计算机的应用逐渐扩大到事务处理等一些非数值计算领域，在这些领域中，用户与计算机的操作方式是交互式操作方式，计算机在处理过程中要不断与用户打交道，根据使用者的指令去执行，在这种与用户的交互作用日益重要的情况下，必然要过渡到消息驱动方式。这也是时代发展的趋势。

1.1.2　使用 EasyWin

即使在 Windows 下，有时也要编一些相对简单的程序，使用 DOS 编程的一般方法即可。如希望它能以 Windows 程序的方式运行，换句话说，就是能尽量简便地将 DOS 程序移植到 Windows 下，使用 EasyWin 可以简单地做到这一点。

EasyWin 是 Borland 公司推出的 Borland C++中用来将 DOS 程序移植成 Windows 程序的一个工具。编本书时用的是 Borland C++ 3.1。在 BC++3.1 的 Borland C++ for

Windows 程序开发环境下(简称 BCW)只要打开一文件,按 DOS 下 C++程序的一般编写方法去编写,然后选菜单中的 Run 命令即可。BCW 在链接程序时,如果没有找到 Windows 应用程序特有的入口点 WinMain()函数,便会寻找 main()函数并连接 EasyWin 库。这样,几乎一行专用于 Windows 的代码都不必编写,就可以得到一个可在 Windows 下运行的程序。这样,在编一些普通的 Winodws 程序时,可以很容易地用 EasyWin 达到目的。

值得注意的是,EasyWin 只能移植使用字符接口的程序。如果你的 DOS 程序涉及直接操作硬件,使用 DOS 的图形模式,那么,你的程序是不可移植的。因为在 Windows 这些部分是必须由操作系统控制的部分,你的程序与它们直接打交道将和操作系统相冲突,导致程序不可再运行。所以,如果你的程序中有上面所说的成分,如果想移植到 Windows 下,那么,就应考虑编写一正规的 Windows 函数代替 DOS 下图形库中的函数。

1.2 实例分析

下面,我们仔细看一个标准的 EasyWin 程序,作为对 DOS 的一个回顾,了解 EasyWin 能干什么。

程序的名称是 DOSSTYLE.CPP,在 Borland C++的 BCW 环境下,选择 New,创建一新文件,将程序输入,然后选择 Run 命令,即可以 Windows 应用程序的方式运行。在运行完毕后,按标准 Windows 程序关闭方法打开系统菜单,然后按 Close 命令关闭。

应用程序代码

```
//filename: dosstyle.cpp, demo dos io.
// author TSINGHUA ZHANGJIAN

#include <iostream.h>
#include <conio.h>
main()
{
    cout<<"This program accept two number"<<endl;
    cout<<"Please Input first number:";
    float m,n;
    cin>>m;
    cout<<"\nPlease Input sceond number:";
    cin>>n;

    clrscr();

    cout<<m<<" add "<<n<<" Get "<<m+n<<endl;
    cout<<m<<" multiple "<<n<<" Get "<<m*n<<endl;

    cout<<"now will delete the first line,press any key start."<<endl;

    getch();

    gotoxy(1,1);
    clreol();

    cout<<"now will clean screen,press any key start."<<endl;
    getch();

    clrscr();
```

```
        cout<<"This is a demo of Dos time screen control style."<<endl;
        gotoxy(8,2);
        cout<<"Author TSINGHUA ZHANGJIAN,1/16,1994"<<endl;
        return 0;
    }
    //file dosstyle is end;
```

程序的前两行是预处理包含文件命令，包含了 iostream. h 和 conio. h 两个标准头文件。包含 iostream. h 是因为我们使用了 C++中的流，利用流的插入算符"<<"和提取算符">>"来进行输入和输出。conio. h 用来进行控制屏幕输出的一些特性，对屏幕输出进行简单的控制。

流的介绍

在这里值得说一说的是 C++的流，使用 C++中的流进行输入和输出比用标准 C 语言库中的输入输出函数 printf()和 scanf()要方便得多。这是因为在 C++中进行了函数重载，C++语言能够自动根据参数的不同类型而去调用不同的函数。对一个通常的算符调用，比如：

```
int i,j ;
cout<<i<<j<<endL;
```

C++语言实际上将调用语句转换为如下的函数调用(假设"<<"算符是 C++流的成员函数)((cout. <<(i))、<<(j))；由于"<<"的返回值仍然是输出流，因此可以接着进行输出调用，如果"<<"算符是输出流的友元，那么表达式可以被转换成如下形式：

```
<<(<<(cout,i)),j)
```

自定义流操作符

使用 C++的流的优点是不必去记忆繁杂的操纵控制符，在程序设计中，不必去翻看语言手册，即可直接写出输出表达式。

流的另一个特点是你可以写出你自己的类的输出表达式，而仅需同样调用 C++中的输出算符，下面看一个例子。

```
Class TMyClass {
        public:int i,j;
            TMyClass(int l,int m) { i=l; j=m}
            friend class ostream;}
Ostream perator <<(ostream &os,TMyClass cl)
    {
        Os<<"\n,now out class TMyClass,"<<cl. i<<" "<<cl. j<<endl;
            return os;
    }
  TMyClass AClass(3,5);
Cout<<AClass;
```

那么，运行上面程序的结果将是如下：

```
now outClass TMyClass 3 5
```

流的优点

从上面和程序中可以明显地看到流的优点。对于一个你自定义的类，你可以用形式统一的流输出算符来操纵它，只要你已重载了运算符。使用形式统一的流输出算符来进行输入输出的好处是巨大的。首先形式方便，流输出都有统一的格式。另外，当你希望改变输出形式时，只要修改重载函数的定义即可。

利用 conio.h 中提供的函数可以对屏幕输出进行简单的控制。可以进行清屏，这将导致光标移到屏幕左上角。在字符型终端上进行输入输出时，有一个光标的概念。在字符型应用程序中，当你敲入要输入的内容时，你输入的内容也同时在屏幕上显示出来，称之为回显。所谓光标就是决定下一个待显示字符位置的一种闪烁标记。你可以通过控制光标的位置来达到控制下一个字符输出位置的功能。在正宗的 Windows 程序中，由于鼠标的指示已被称为光标，所以对于字符型终端的这种概念在 Windows 中称为插入符号。

程序外观

在 DOSSTYLE.CPP 的 main()函数中，开始显示一提示信息，请你输入两个数。然后程序利用 C++中的流操作符来接收你输入的两个数。这里，你再次看到了 C++流的方便与易用性。

为了显示普通 DOS 类程序的特点，在取得了两个数后，程序首先应用 ClrScr()函数来清屏。调用此函数将清除屏幕上的显示内容，将光标定位于屏幕左上角。这是说，这时，下一个显示的字符，除非你特别控制，将显示在屏幕的左上角。

这时，程序计算出你输入的两个数的和与积，分别显示出来。

这个程序的主要目的是为了演示 DOS 风格程序的特点，所以，这时程序请你按下任意一键以便清除屏幕显示的第一行。利用 getch()来等待用户按下任意一键以便进行下一个操作是 DOS 程序设计中常用的方法。getch()用来取出一个用户输入的字符，该函数将等待，直至用户按下一键时才返回，所以只有当用户按下一键后该函数才会返回。

光标与屏幕坐标系

你按下一键后，程序接着往下执行。程序首先调用 gotoxy()来将光标移到第一行第一个字符的位置，这使该位置成为当前显示位置。然后调用 clreol()从当前位置开始，清掉该行的内容。在这里，值得说一说的是字符显示中的坐标。计算机显示中，屏幕显示坐标的规定和人看书时的习惯是一致的。即从左至右、从上到下。所以，屏幕上 x 坐标增长方向是从左至右，而 y 轴的方向是从上到下。而在普通的迪卡尔坐标系中，y 轴方向是从下至上增长的。这是值得注意的地方。在通常用的屏幕坐标中，屏幕的原点在屏幕的左上角。一般将该点定为(1,1)，即 x 方向和 y 方向的坐标都为 1 的地方。一般屏幕是 40 列，25 行，即 x 的变化范围是从 1 至 40，y 轴是 1 到 25。

程序在清掉第一行后，在该行上显示一提示信息，告诉你，按下任意一键后将清屏，你按下任意一键后，屏幕清屏后显示一行提示信息，然后在第二行的中间显示版权信息。

这个程序在运行完毕返回后，程序的窗口将加上一个前缀“inActive”，意思是该程序已从 main()主函数返回。在此之前，EasyWin 程序的窗口边框不可以被缩放。变为不活动后，则可改变大小。

如果这个程序在 Borland C++的 DOS 环境下编译，则生成一个 DOS 程序。建议你这样做，以便比较 DOS 环境和 EasyWin 环境的异同。

通过上面这个程序，你应该对 DOS 下屏幕控制有较多的了解。在 DOS 下，你可以在屏幕的任一点开始显示，这使用函数 gotoxy()将光标移到那一点即可。也可以将屏幕上的显示内容清掉，清掉一行或整屏都可以。使用 ClrScr()和 Clreol()这两个函数即可达到目的。

1.3　Windows 编程分析

1.3.1　Windows 的尴尬

当你进入 Windows 进行编程时，感到惊叹的不仅是近七百个函数（普通 C 语言库只有一百多个函数），而且对调用这些函数时所需的参数个数感到惊奇，调一个函数，经常要传递四、五个参数，有的甚至十几个。令人难以记忆，难以掌握。所以，几乎所有的 Windows 程序员都要在桌面上摆一本程序员参考手册，当在计算机上编制程序时，还要随时查看联机帮助文件，以便寻找某个函数的具体用法。有时候，当你对某个 Windows 函数的特性不十分了解时，希望先编一个小程序测试它一下。如果是在 DOS 下，只要包含有这个函数定义的头文件，然后在 main()函数中直接调用该函数即可。代码一般不超过十行。这是掌握函数用法，提高自己水平的很有效的方法。但是，在 Windows 下，即使编一个什么功能都不实现，只具有一个可以关闭的简单窗口的程序，其代码恐怕也已超出了四五十行。有谁会愿意为试验一个函数而去对着屏幕敲四五十行源代码呢？这恐怕只会使人对 Windows 编程兴趣索然。

1.3.2　Windows 的事件驱动与函数调用特点

那么，为什么会造成这种状况呢？这是因为，Windows 是一个消息驱动式的程序设计环境，你编的程序必须具有 Windows 程序所共有的那些部分。同时，很多 Windows 函数都要有一个句柄参数，以便操作系统对此函数的行为进行控制。比如，对窗口操作的函数都要有一个窗口句柄作为头一个参数，以便操作系统知道是对哪个窗口进行操作，避免对其它窗口的操作，以便实现多任务。几乎所有的 GDI 函数都有一个 HDC 句柄参数，以便 Windows 利用此句柄对该函数的行为加以控制，比如，剪裁，以免绘制操作影响到别的区域等。在 Windows 中，窗口句柄、实例句柄都是在程序创建过程中得到的，似乎只有按正规的 Windows 程序来设计，才能获得这些句柄，并把这些句柄参数用在各种函数调用中。

那么，难道就没有什么像 DOS 下实验函数那样方便的方法吗？希望首先获得各种句柄参数，在数十行代码内，然后用这些句柄来实验函数。

这是能做到的，答案就是利用 EasyWin。

EasyWin 中已为你做好了消息处理机构，Borland C++在连接你的程序时，如果没有找到 WinMain()入口，则会寻找 main()入口并自动连接 EasyWin 库。那么，在 EasyWin 中如何获得各项句柄呢？

1.3.3　获得 Windows 中的句柄

Windows 是一个非占先式的操作系统，意思是说，只有应用程序主动让出执行权，程序才有可能切换到另一个程序。而应用程序是以窗口为基础的。一个应用程序拥有一个主窗

口，用户和此窗口进行交互操作，操作系统向此窗口发送各种消息。在操作系统中，保存有一个窗口句柄表，其中包含有各个窗口句柄、每个窗口句柄实际上是操作系统数据段中的地址。窗口的数据包括实例句柄，程序窗口函数处理地址等各种数据。可见，只要有了窗口句柄，其它各种参数就都能获得。Windows 保存当前活动窗口的窗口句柄，这样，我们就可以获得活动窗口句柄。

当一个 EasyWin 程序开始执行时，它所创建的窗口成为当前活动窗口，因为这个窗口接收键盘输入等消息。这时，如果用 Windows 函数 GetActiveWindow()将获得当前活动窗口句柄，这个窗口活动句柄就是应用程序产生的 EasyWin 窗口，这样，我们所有的问题就都迎刃而解。

1.4 Windows 与 DOS 混合的应用程序

我们来研究下面这个 demogdi.cpp 程序。这个程序有一个菜单，你可以在窗口上产生一个钟表，一个沿窗口四个边沿碰撞的小球，以及用随机颜色在窗口的随机区域位置上画矩形方块。你也许感到奇怪，菜单是怎么加上的呢？Windows 是基于消息的操作系统，你自己的消息处理函数是怎么加上的呢？别急，让我们一步一步来看。这个程序虽然很不完善，但它展示了 Windows 编程中的各种技巧，值得研究。

程序代码

```
//filename: get.h,get the EasyWin handle;
// author TSINGHUA ZHANGJIAN
#include <windows.h>
class TGet{
    public:
        TGet();
        HWND hWnd;
        HINSTANCE hInst;
};

inline TGet::TGet()
{
    hWnd=GetActiveWindow();
    if(IsWindow(hWnd))
        hInst=GetWindowWord(hWnd,GWW_HINSTANCE);
    else hInst=0;
}

//define class TDC,get the window DC,
class TDC{
    public:
        TDC(HWND hWnd){
            this->hWnd=hWnd;
            hDC=GetDC(hWnd);
            }
        TDC(TGet AGet){
            hWnd=AGet.hWnd;
            hDC=GetDC(hWnd);
```

```
        }
        ~TDC(){
            ReleaseDC(hWnd,hDC);
        }
        HWND hWnd;
        HDC hDC;
};
//file get.h is end;
//////////////////////
//file name:menu.h;use menu in EasyWin;
//author TSINGHUA ZHANGJIAN
#include <windows.h>
class TMenu{
    public:
        TMenu(){
            hMenu=CreateMenu();
            hPopup=CreatePopupMenu();
            AppendMenu(hMenu,MF_POPUP|MF_STRING,hPopup,"&Menu");
            }
        Add(int nID,char * string){
            AppendMenu(hPopup,MF_STRING,nID,string);
            AppendMenu(hPopup,MF_SEPARATOR,0,0);
        Check(int nID,BOOL MF){
            CheckMenuItem(hPopup,nID,MF? MF_CHECKED:MF_UNCHECKED
);
            }
        HMENU hMenu,hPopup;
};
//file menu is end;
//filename demogdi.cpp,used demo gdi and message control in EasyWin;
//author TSINGHUA ZHANGJIAN
#include <windows.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "get.h"
#include "menu.h"
#define IDM_RANDOM   10
#define IDM_CLOCK    20
#define IDM_BALL     30
#define IDM_CLEAN    32
//timer ID define
#define IDT_RANDOM   35
#define IDT_CLOCK    38
#define IDT_BALL     40
#define TIME         1000
#define TIMERANDOM   1000
#define TIMECLOCK    1000
```

```
#define TIMEBALL        100

FARPROC lpfnOldProc;
void BeginRandom(HWND hWnd);
void EndRandom(HWND hWnd);
void BeginClock(HWND hWnd);
void EndClock(HWND hWnd);
void BeginBall(HWND hWnd);
void EndBall(HWND hWnd);
void DoRandom(HWND hWnd);
void DoClock(HWND hWnd);
void DoBall(HWND hWnd);

TMenu Menu;

LRESULT WINAPI_export DealCommand(HWND hWnd,WORD Message,WPARAM
wP,LPARAM lP)
{
    switch(Message){
        case WM_COMMAND:{
        switch(wP){
        case IDM_RANDOM:
            if(LOBYTE(GetMenuState(Menu.hMenu,IDM_RANDOM,MF_BYCOMMAND))
              &MF_CHECKED)
                {
                Menu.Check(IDM_RANDOM,FALSE);
                EndRandom( hWnd);
                }
             else {
                Menu.Check(IDM_RANDOM,TRUE);
                BeginRandom( hWnd);
                }
            DrawMenuBar(hWnd);
            break;
       case IDM_CLOCK:
           if(LOBYTE(GetMenuState(Menu.hMenu,IDM_CLOCK,MF_BYCOMMAND))
             &MF_CHECKED)
                {
                Menu.Check(IDM_CLOCK,FALSE);
                EndClock( hWnd);
                }
            else {
                Menu.Check(IDM_CLOCK,TRUE);
                BeginClock( hWnd);
                }
           DrawMenuBar(hWnd);
           break;
      case IDM_BALL:
                if(LOBYTE(GetMenuState(Menu.hMenu,IDM_BALL,MF_BYCOMMAND))
                  &MF_CHECKED)
                {
                Menu.Check(IDM_BALL,FALSE);
```

```
                    EndBall( hWnd);
                    }
                else {
                    Menu. Check(IDM_BALL,TRUE);
                    BeginBall( hWnd);
                    }
                DrawMenuBar(hWnd);
                break;
            case IDM_CLEAN:{
                if(IsWindow(hWnd)){
                  TDC DC(hWnd);
                  RECT rt;
                  GetClientRect(hWnd,&rt);
FillRect(DC. hDC,&rt,GetClassWord(hWnd,GCW_HBRBACKGROUND));
                  }
                  }
                  break;
            default: return CallWindowProc(lpfnOldProc,hWnd,Message,wP,lP);
            }
          }
          break;
      case WM_TIMER:
      if(wP==IDT_RANDOM){
if(LOBYTE(GetMenuState(Menu. hMenu,IDM_RANDOM,MF_BYCOMMAND))
  &MF_CHECKED)
              DoRandom( hWnd);
      }
      else if(wP==IDT_CLOCK){
      if(LOBYTE(GetMenuState(Menu. hMenu,IDM_CLOCK,MF_BYCOMMAND))
        &MF_CHECKED)
              DoClock( hWnd);
              }
          else if(wP==IDT_BALL){
          if(LOBYTE(GetMenuState(Menu. hMenu,IDM_BALL,MF_BYCOMMAND))
            &MF_CHECKED)
              DoBall( hWnd);
          }
      break;
      case WM_DESTROY:
          if(lpfnOldProc)
              SetWindowLong(hWnd,GWL_WNDPROC,(long)lpfnOldProc);
      default: return CallWindowProc(lpfnOldProc,hWnd,Message,wP,lP);
      }
}

main()
{     cout<<"This is EasyWin gdi demo"<<endl;
      TGet AGet;
      Menu. Add(IDM_RANDOM,"&Random");
```

```
        Menu.Add(IDM_CLOCK,"&Clock");
        Menu.Add(IDM_BALL,"&Ball");
        Menu.Add(IDM_CLEAN,"C&lean Screen");
        SetMenu(AGet.hWnd,Menu.hMenu);
        lpfnOldProc=(FARPROC)SetWindowLong(AGet.hWnd,GWL_WNDPROC,(long)Deal
Command);
        return 0;
}
void BeginRandom(HWND hWnd)
{
        if(IsWindow(hWnd)){
        SetTimer(hWnd,IDT_RANDOM ,TIMERANDOM,NULL);
        randomize();
        }
}
void EndRandom(HWND hWnd)
{       if(IsWindow(hWnd)){
        KillTimer(hWnd,IDT_RANDOM);
        }
}
void BeginClock(HWND hWnd)
{
        if(IsWindow(hWnd)){
        SetTimer(hWnd,IDT_CLOCK ,TIMECLOCK,NULL);
        randomize();
        }
}
void EndClock(HWND hWnd)
{       if(IsWindow(hWnd)){
        KillTimer(hWnd,IDT_CLOCK);
        }
}
void BeginBall(HWND hWnd)
{
        if(IsWindow(hWnd)){
        SetTimer(hWnd,IDT_BALL ,TIMEBALL,NULL);
        randomize();
        }
}
void EndBall(HWND hWnd)
{
        if(IsWindow(hWnd)){
      KillTimer(hWnd,IDT_BALL);
      }
}
void DoRandom(HWND hWnd)
{
     RECT rt;
          GetClientRect(hWnd,&rt);
     rt.right=random(rt.right);
```

```
        rt.bottom=random(rt.bottom);

        char r,g,b;
        r=random(255);
        g=random(255);
        b=random(255);
        COLORREF c=RGB(r,g,b);

        if(IsWindow(hWnd)){
        TDC DC(hWnd);
        HBRUSH old=SelectObject(DC.hDC,CreateSolidBrush(c));
        Rectangle(DC.hDC,rt.right-10,rt.bottom-10,rt.right+10,rt.bottom+10);
        DeleteObject(SelectObject(DC.hDC,old));

          }
}
#define direction 60
#define step          20
void DoBall(HWND hWnd)
{
        static int x,y,x0,y0;//store last position.
        static BOOL xd,yd,Init;
        static RECT rt;
        if(IsWindow(hWnd)){
            TDC DC(hWnd);
            SetROP2(DC.hDC,R2_NOTXORPEN);

            if(! Init) {
            GetClientRect(hWnd,&rt);
            x0=step * cos(M_PI * direction/180.0);
            y0=step * sin(M_PI * direction/180.0);
            xd=yd=TRUE;
              Init=TRUE;//indicate initiate have done;
                    }
            if(Init){
            HBRUSH old=SelectObject(DC.hDC,GetStockObject(BLACK_BRUSH));
            Ellipse(DC.hDC,x-10,y-10,x+10,y+10);
            SelectObject(DC.hDC,old);
            }

            if(xd){
            x+=x0;
            if(x>rt.right)xd=! xd;
            }
            else{
                x-=x0;
            if(x<0)xd=! xd;
            }
            if(yd){
                y+=y0;
                if(y>rt.bottom)yd=! yd;
            }
            else {
                y-=y0;
```

```
            if(y<0)yd=! yd;
          }
        //have justified position;
        HBRUSH old=SelectObject(DC.hDC,GetStockObject(BLACK_BRUSH));
        Ellipse(DC.hDC,x-10,y-10,x+10,y+10);
        SelectObject(DC.hDC,old);
    }
}
void DoClock(HWND hWnd)
{
    static int nLastAngle,x,y;
    static int r0;
    static RECT rt;
    if((nLastAngle+=10)>=360)nLastAngle=0;
    if(IsWindow(hWnd)){
        TDC DC(hWnd);
        SetROP2(DC.hDC,R2_NOTXORPEN);
        if(! x&&! y) {
        GetClientRect(hWnd,&rt);
            r0=(rt.right<rt.bottom? rt.right:rt.bottom);
            r0/=2;
              }
            MoveTo(DC.hDC,rt.right/2,rt.bottom/2);
            if(x||y)LineTo(DC.hDC,x+rt.right/2,y+rt.bottom/2);
        x=r0*cos(M_PI*nLastAngle/180.0);
        y=r0*sin(M_PI*nLastAngle/180.0);
        MoveTo(DC.hDC,rt.right/2,rt.bottom/2);
        LineTo(DC.hDC,x+rt.right/2,y+rt.bottom/2);
    }
}
//file demogdi is end;
```

代码分析

文件的开头包含了四个系统文件，其中包含文件 math.h 是因为为了计算时钟秒钟的屏幕位置而使用了正弦和余弦函数。

接下来，文件包含了自定义的一个文件头 get.h。它是用来获得 EasyWin 的活动窗口的。在类 TGet 的构造函数中，利用 GetActiveWindow()函数获得了当前活动窗口，也就是我们程序的窗口，把它放在一个公用数据成员 hWnd，这样，当你一旦生成一个 TGet 类对象，也就同时获得了当前的窗口句柄。你应该还记得，当生成一个类的实例，即一个该类的对象时，C++编译器将隐含地为这个对象调用构造函数。从而，一些必要的初始化代码只要放在构造函数中，就可以完成初始化，使代码简洁，避免遗忘。

取得句柄

类 TGet 的另一个数据成员是该窗口的实例句柄 hInst，一个窗口的 hInst 实际上就是

该程序的数据段地址。那么，实例句柄是如何获得的呢？在 Windows 中，操作系统为窗口维护的数据表中，就有实例句柄。利用 GetWindowWord(nWnd,nIndex)，其中，nIndex 可以为 GWW_HINSTANCE，这样，你就可以获得实例句柄。如果 nIndex 值为 GWW_HWNDPARENT 或 GWW_ID 就将获得该窗口的父窗口或窗口标识值。从这里，你可以看到，如果在你的程序中用到了类似的值，你就可以用上面的方法来获取这些值，而不必在程序中自己保存这些值。

用 GWW_前缀获得的是该窗口的 WORD 型值。窗口还有另外三个 long 型变量，分别是 ExStyle，Style，WndProc，即扩展的窗口式样，窗口式样和窗口的消息处理函数地址。用 GetWindowLong 函数即可获得。

这样，当你生成一个类 TGet 的实例时，你就同时获得了该窗口的句柄和实例句柄，这样，你就可以操作该窗口了。

如何防止头文件重复包含

在编写稍微复杂一些的软件时，一个软件程序通常由几个.CPP 文件构成，同时还要有相应的几个头文件。这样，就产生了如何防止重复包含的问题。如果一个头文件被包含两次，则会产生符号名冲突的错误。举一个通常的例子，几乎所有的 Windows 程序都要包含<windows.h>头文件。如果你有两个头文件，一个称为"A.h"，另一个称为"B.h"，它们为了应用 Windows 函数，同时包含了<windows.h>，如果没有预防措施，那么当你必须在一个文件中包含"A.h"和"B.h"时，就会产生重复包含错误。但实际上这种错误是不会发生的。因为采用了预防措施。一般地，为了让编译器知道已经包含了头文件，在头文件的开头一般进行宏名检测，如果标记该文件的宏已被定义，则不包含该文件，否则，定义该宏，同时包含主体部分。

举个例子，以头文件 Get.h 为例。

```
#ifndef__GET_H
#define__GET_H
//other Line
#endif //ifndef__GET_H
```

在头文件的开头定义一个根据该文件名产生的宏__GET_QH，先检测该宏，如果没定义过，那么，ifndeft endif 语句之间的部分将被包含使用。如果已定义过该宏，则说明了该头文件已被包含过，可以不必包含 ifndef 和 endif 语句之间的语句块。使用这种技术，就可以避免重复包含，从而避免引起错误。

给宏取名

有一个小问题是如何根据头文件名来取宏的名字，虽然从理论上说你可以取任意的名字，但与文件名有联系将使人容易看懂。所以应从文件名出发来取名字。该规则应与你所用编译器一致。在中国当前广泛使用的是 Borland 公司的 Borland C++3.1 和 Microsoft 的 MSVC 1.0 或 Microsoft C++7.0。如果你用的是 Borland 公司的产品，应使用 Borland 公司的规则，即将头文件名全部变为大写，中间区分后缀的点变为下划线，同时，在文件名的前面加两个下划线字符。举个例子，比如 Get.h，利用 Borland 公司的规则就得到了__GET_H 这个宏名字。另一方面，如果你用的是 Microsoft 公司的产品，则应按如下规则：文件名全变为

大写，中间点为下划线后，在文件名的开头和结尾各加入两个连字符，仍用前面的例子，由 Get.h 产生的宏名为__GET_H__。具体采用哪种规则根据你用的产品自己定义。因为 EasyWin 是 Borland 公司的产品，所以用的是 Borland 公司的规则。

窗口中的菜单

另一个被包含的自定义头文件是 menu.h。它定义了一个类 TMenu 用来操作菜单。因为 EasyWin 不能用通常的方法加入菜单，所以必须用自己生成菜单的方法再加进去。

菜单是很值得谈一谈的话题。菜单是 Windows 和用户交互作用的手段之一。有时，需要修改菜单，另一种是普通的菜单项，当选中它时，系统将退出菜单选择状态，并向应用程序改善菜单选择消息，因为这种状态代表不能被使用。另外两种状态是检查状态和非检查状态。一般在菜单左边有个对号表示该菜单项已被设置为检查状态。你可以通过调用函数 CheckMenuItem()来改变菜单项状态。

在我们的程序中，由于开始时主窗口没有菜单，所以我们自己创建一个空菜单句柄，然后再依次加入菜单项，最后将该菜单连到主窗口中。供用户选择使用。

添加菜单

在 menu.h 中定义的类 TMenu 用来提供添加菜单的接口。TMenu 类的构造函数中首先调用函数 GreateMenu()创建一空菜单项句柄，用来添加各种菜单项。然后再创建一空菜单项以便加入供用户选择的菜单项。然后，将该弹出式菜单以"menu"的名字加入到我们创建的空菜单中。从构造函数中可以看到"Menu"前面有个"&"符号，这代表以它后面的一个字母为加速键。在本例中，如果你按下"ALT"键和字母 M，将进入菜单选择状态，显示出弹出式菜单项"Menu"下的各种菜单选择项。如果该"&"所在的项是在普通菜单项中，那么，当在菜单中时，该字母下有一下划线，按下该字母将结束菜单状态并发送命令消息。

有时，在有的应用程序中，你会注意到有一个菜单项在菜单栏的最右边，一般的 Windows 文件并未记录有如何达到该功能的方法。你可以采用在给要放在最右边的菜单字符串上加一个前缀"\a"的方法来做到这一点。比如字符串"\a&Help"将导致在菜单栏的最右边出现"Help"菜单。如果你喜欢这种方式的菜单条，就可以使用这里说的方法。

类 TMenu 提供了两个简单的接口函数 Add()和 Check()，Add()用来将一菜单项加入到弹出式菜单"Menu"中。Check()则用来在菜单栏前面加上检查标志或去掉检查标志。

下面接着的是常数定义语句和函数声明语句。如果你愿意的话，这些本来可以放在一个头文件中，这里将它直接放在.CPP 文件中了。

使用宏的优点

几个菜单用常数定义在这里，使用宏来定义的优点是意义清楚，使人很容易看懂该常数的用途。另一个重要的方面是修改的方便。一个常数有时要在程序中出现好几次，如果直接以它的数值代替，则每次要修改好几处。而用常数定义则只需改变定义该常数宏即可，只需改变一处。你在编程时应注意这一点。要记住，编程的首要目的是给人阅读，而非计算机编译。

以 IDM 开头表示是菜单项(M 是 Menu 之意)。IDM_RANDOM 用来在屏幕上的随机位置以随机颜色显示的矩形。DIT_RANDOM 是时钟标识符，TIMERANDOM 决定了两个

矩形出现时间的间隔。

Windows 中的定时

有必要说明的是在 Windows 中如何处理定时。有时，有很多操作要以时间为依据，比如在我们的程序中小球的运动，秒针的转动，都需要定时。在过去的 DOS 程序中，采用的接收时钟中断的方法，一个程序设置 DOS 的时钟中断向量，在处理该中断的程序中完成定时操作。在 Windows 中，采用的方法是设置时钟，你用函数 SetTimer()来设置时钟，在该函数中，你要指定时钟标识符，时间间隔，间隔以毫秒计。你设置时钟成功后，Windows 将按你设定的时钟间隔发送 WM_TIMER 消息，其 WPARAM 参数是时钟标识符，用来区分用户设置的多个时钟。当用完时钟后，调用函数 KillTimer()来终止这些定时器。在我们的程序中，设置了三个时钟。

修改窗口地址

接下来是几个函数声明，另外，声明了一个变量用来保存旧的窗口函数处理地址。为了能使窗口能处理我们自己的消息，采用的是替换窗口函数的方法。即将原来的窗口函数地址获得后，将其保存在 lpfnOldProc 中，将窗口函数的地址为我们自己的处理消息的函数地址。因为 Windows 产生一条消息时，首先寻找消息处理函数的地址。但这时候地址已被换成我们自己处理的地址，所以我们自己的处理函数首先被调用。在我们的处理函数中，对于我们不处理的消息，则再传送给原来的函数处理。这样，既保持了原有窗口的特征，又加进了我们自己的处理。

SubClass 技术

这种改变窗口处理函数的方法称为“SubClass”技术，它对于创建既继承原有窗口特点，又具有新特征的窗口具有重要的意义。比如，对于 Windows 已有的一些窗口类，当你希望对它加进新的特征时，采用的就是这种技术。在 Borland 公司的 ObjectWindows 和 Microsoft 的 mfc 类库中都有这种技术的使用。

处理发送来的消息

在 demogdi.cpp 中，DealCommand 是我们新的消息处理函数的地址，它的形式必须是 DealCommand(HWND,WORD,WPARAM,LPARAM)的形式，因为 Windows 假设窗口函数具有此种形式并用此种形式来调用该函数，说明为别的形式将引起不可预料的错误。

DealCommand()中首先处理几个菜单命令消息，对每个消息来说，首先判断菜单状态，如果已被设置为检查状态，则改变菜单为非检查状态，并终止该命令所引起的动作。对不处理的消息利用 CallWindowProc 传给原来的处理函数。该函数的作用是用参数中的消息去调用消息处理函数。

该程序中另一值得注意的是绘制方式。屏幕上小球的运动看起来好象真的在四处弹跳一样。实际上，从源程序中可以看到，只不过是从屏幕上消掉原来的小球，然后再计算新的位置，并在新位置上重新画出原来的小球。由于人眼的视觉暂留及心理作用，看起来就好象小球真的在运动一样。实际上，所有的图像重现设备，包括电影和电视，采用的都是这种原理。

移动图像绘制

在这种消去原有绘制图形及显示新的图像中，值得注意的是绘制方式的选择。你可能觉得简单，消去原有图像只要把它用背景刷子刷一遍就行了，绘制新图像只要将图像拷贝过去就行了。实际上，这种方式只适用于背景单调的情况。如果屏幕上原来有一些别的图像，用上面这种方法将破坏原有图像，是不可取的。

怎样使原有的背景色不受影响呢？方法是利用逻辑操作。众所周知，对于一个布尔型变量，和另一个布尔型变量异或两次将回复原值。绘制时，采用这种原理。对于待绘制图像，取绘制方式为异或方式，这可以通过调用函数 SetRop 来做到。先在原有图像位置上再绘制一次，由于这相当于两次异或，所以，将消去该图像，而背景则不受破坏。然后，在新的位置上再绘制一次，于是图像就在新的位置上显示出来。

程序中别的部分可以自己看懂，无需多言。

上面讨论了 Windows 中的 SubClass 技术，定时器设置，动画技术的实现等 Windows 应用的较高级技术。

1.5 用单机调试 Windows 程序

调试工具的使用

下面讲一讲如何利用 EasyWin 作调试终端。

如果你曾经编过程序，主持过软件项目的话，一定会同意如下的话：编软件，80%的时间是用在调试上。这句话一点也不过份地说明了调试的重要性。人总有考虑不到的情况，所以，当你把一个软件编出来后，几乎可以肯定地说这里面必然有错误。但是，怎样找出错误呢？

对于一般查错式的调试，如果你是在 Borland 环境下，可以利用 Turbo Debugger 来调试你的程序。Turbo Debugger 可以在语句上设断点，检查变量的值，而做这一切只不过需要使用鼠标的左右两键即可。并且它最大的优点是易学、易用，特别适合初学者使用。

Borland 的调试工具与 Microsoft 的 CodeView 一样也是字符界面。但 CodeView 的易用性就不如 turbo debugger，使用它需要记忆一些命令，不太方便。

但是，使用调试工具的特点是繁杂、缓慢，改变了程序的执行环境。比如，在程序执行中有时需要知道某变量的值，你可以使用 Turbo Debugger 来检查该值，或者采用一些初学者使用的另一种方法，使用函数 MessageBox()来告诉自己该值是多少。但这种方法的缺点是显而易见的，第一，麻烦，当你准备把一个程序变成一个无调试的版本时，必须再重新修改这些语句。如果程序较长，达万行规模的话，这将是一个很烦人的问题。第二，影响了程序的执行，因为你必须使用键盘或鼠标来关掉那些消息框。

如何解决上述的问题呢？Borland 公司和 Microsoft 公司都推荐双机调试模式，即一台机器运行程序，另一台机器显示调试中的各种数据和信息，运行 Turbo Debugger 或 CodeView 等调试程序。这是一种比较理想的方式。尤其是 Microsoft 公司的 mfc 类库，为了增强类库的易调试性，在运行程序时，如果用的是调试版本，将产生大量信息送往辅助监视器。

但是，考虑到中国的实际情况，绝大多数人不具备使用双机调试的条件，怎样使用一台机器达到两台机器调试所具有的优点呢？

用 EasyWin 作调试终端

方法是使用EasyWin产生的窗口来做调试终端，利用该窗口来显示调试信息。

你也许还记得，EasyWin是用来转换DOS程序的，它怎么能用于一般的Windows程序中呢？当你在普通程序中使用函数InitEasywin()时，如果以前没产生过EasyWin窗口，将产生一个EasyWin窗口，这时，你调用printf，cout等各种DOS下屏幕输出语句的内容都将出现在该EasyWin窗口上。如果以前已产生了EasyWin窗口，则没有影响。

现在，我们的思想你应该明白了。当运行程序时，除了产生一个主窗口外，该窗口是我们程序的窗口，同时，还产生一个EasyWin窗口。这时，你可以调整这两个窗口的位置和大小，使它们恰好互不覆盖。此时，你就可以对子窗口进行操作，各种信息将源源不断地产生在屏幕上，既不影响你执行主程序，又可以看到各种数据，尤其是可以用卷滚条翻看以前产生的信息，进行比较。使用这种方法来调试，收到了在一个屏幕上达到了两个显示器的效果，可以说既经济又实惠，符合国情，何乐而不为呢？

下面，我们先讲明与调试有关的几个问题。

首先，如何比较方便地在调试版本和正式程序版本之间转换。因为调试产生的信息主要用于自己检查错误时使用，在正式版本中当然不能产生这些信息，因为这些信息不但无用，更影响机器的速度。

在集成开发环境下，对于生成可执行文件，一般有去除调试信息这一项功能。生成正式版本时，只要选择去除调试信息这一项即可。

但是，如何去除你程序中的调试信息呢？比较笨的方法是在生成正式版本时用编辑器删除那些与调试有关的语句，但这种方法不但麻烦，而且当你再次修改程序时，需要再加上那些语句，太不方便。

要想比较理想地解决上述问题，应该使用宏。

调试准备

我们先看一看程序，调试中应当产生哪些信息才能方便调试。

第一种是某个变量或某个表达式的值，你需要在调试时知道这些值，以便知道你的程序是否已正确地给变量赋值，并且从变量的值，你可以大体估计出程序该向哪儿执行，观察执行结果是否正确。在正式版本中，当然不能产生这种信息。一般称这种调试语句为dump，即出该值。

第二种是确定某个前提条件是否确定，比如一变量，它的值永远不能小于0，比如数组下标变量。如果你希望确定该表达式，就需在这里测试一下，看条件是否满足。在程序调试完毕后，这种语句也是多余的，也应去掉。一般称这种为assert。

为了能容易地在调试版本与正式版本间转换，采用定义宏的方法。对于定义该宏时，程序调试语句成为空语句，不产生代码，从而达到转换命令。

几个宏的定义

要想能轻松地调试程序，还应明白几个编译器定义的宏及宏命令。

编译器在编译一个文件时，首先定义了__FILE__这个字符串，它是当前正被编译的文件的文件名，你的程序中可以直接使用它来作为文件名。

另一个是__LINE__，它是当前正被编译的行的行号，利用这个宏，你可以知道在源程序的第几行。