用哪个符号。

在wndclass被初始化之后，RegisterClass被调用，将其发送给Windows。特别地，在该调用之后Windows就会知道当各种事件发生时要调用哪个过程。下一个调用CreateWindow为窗口的数据结构分配内存并且返回一个句柄以便以后引用它。然后，程序做了另外两个调用，将窗口轮廓置于屏幕之上，并且最终完全地填充窗口。

此刻我们到达了程序的主循环，它包括获取消息，对消息做一定的转换，然后将其传回Windows以便让Windows调用WndProc来处理它。要回答这一完整的机制是否能够得到化简的问题，答案是肯定的，但是这样做是由于历史的缘故，并且我们现在坚持这样做。

主循环之后是过程WndProc，它处理发送给窗口的各种消息。此处CALLBACK的使用与上面的WINAPI相类似，为参数指明要使用的调用序列。第一个参数是要使用的窗口的句柄。第二个参数是消息类型。第三和第四个参数可以用来在需要的时候提供附加的信息。

消息类型WM_CREATE和WM_DESTROY分别在程序的开始和结束时发送。它们给程序机会为数据结构分配内存，并且将其返回。

第三个消息类型WM_PAINT是一条指令，让程序填充窗口。它不仅当窗口第一次绘制时被调用，而且在程序执行期间也经常被调用。与基于文本的系统相反，在Windows中程序不能够假定它在屏幕上画的东西将一直保持在那里直到将其删除。其他窗口可能会被拖拉到该窗口的上面，菜单可能会在窗口上被拉下，对话框和工具提示可能会覆盖窗口的某一部分，如此等等。当这些项目被移开后，窗口必须重绘。Windows告知一个程序重绘窗口的方法是发送WM_PAINT消息。作为一种友好的姿态，它还会提供窗口的哪一部分曾经被覆盖的信息，这样程序就更加容易重新生成窗口的那一部分而不必重绘整个窗口。

Windows有两种方法可以让一个程序做某些事情。一种方法是投递一条消息到其消息队列。这种方法用于键盘输入、鼠标输入以及定时器到时。另一种方法是发送一条消息到窗口，从而使Windows直接调用WndProc本身。这一方法用于所有其他事件。由于当一条消息完全被处理后Windows会得到通报，这样Windows就能够避免在前一个调用完成前产生新的调用，由此可以避免竞争条件。

还有许多其他消息类型。当一个不期望的消息到达时为了避免异常行为，最好在WndProc的结尾处调用DefWindowProc，让默认处理过程处理其他情形。

总之，Windows程序通常创建一个或多个窗口，每个窗口具有一个类对象。与每个程序相关联的是一个消息队列和一组处理过程。最终，程序的行为由到来的事件驱动，这些事件由处理过程来处理。与UNIX采用的过程化观点相比，这是一个完全不同的世界观模型。

对屏幕的实际绘图是由包含几百个过程的程序包处理的，这些过程捆在一起形成了GDI（Graphics Device Interface，图形设备接口）。它能够处理文本和各种类型的图形，并且被设计成与平台和设备无关的。在一个程序可以在窗口中绘图（即绘画）之前，它需要获取一个设备上下文（device context）；设备上下文是一个内部数据结构，包含窗口的属性，诸如当前字体、文本颜色、背景颜色等。大多数GDI调用使用设备上下文，不管是为了绘图，还是为了获取或设置属性。

有许许多多的方法可用来获取设备上下文。下面是一个获取并使用设备上下文的简单例子：

```
hdc=GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

第一条语句获取一个设备上下文的句柄hdc。第二条语句使用设备上下文在屏幕上写一行文本，该语句设定了字符串开始处的（x, y）坐标、一个指向字符串本身的指针以及字符串的长度。第三个调用释放设备上下文，表明程序在当时已通过了绘图操作。注意，hdc的使用方式与UNIX的文件描述符相类似。还需要注意的是，ReleaseDC包含冗余的信息（使用hdc就可以惟一地指定一个窗口）。使用不具有实际价值的冗余信息在Windows中是很常见的。

另一个有趣的注意事项是，当hdc以这样的方式被获取时，程序只能够写窗口的客户区，而不能写标题条和窗口的其他部分。在内部，在设备上下文的数据结构中，维护着一个修剪区域。在修剪区域之外的任何绘图操作都将被忽略。然而，存在着另一种获取设备上下文的方法GetWindowDC，它将修剪