附录D　开发中注意事项

还剩下一些问题没有讨论，下面这些问题虽然与 VC++或MFC类库无关，但在应用创建一个应用程序时，也是同样重要的。

设置Developer Studio　Developer Studio可以轻松地设置为符合个人开发应用程序的标准。这里我们将讨论其中的一些选项。

调试　有一些资源可以用来调试应用程序，除了在运行应用程序的调试版时明显可用的交互调试器外，还可以调用框架、 DEVELOPER STUDIO，甚至MFC类函数以及第三方运行调试器。

组织和建立工程　如果应用程序只是一个执行文件，而没有附加的 DLL文件时，则你可能停留在工程的工作空间范围内。然而，如果应用程序涉及多个库和多个执行文件时，则需要考虑建立和维护工程的策略问题。

Microsoft **命名习惯**　这里讨论Microsoft坚持的某些非正式的和完全自愿的变量命名习惯。

Spying　SPY.EXE是潜在的工具，它不仅可调试自身的应用程序，而且也作为一种“监视”其他应用程序的方法，以推算出他人是怎样执行他们的界面的。

其他例子资源　除了本书例子以外，我们还介绍可以参考的其他例子资源，这些资源可能有一个或多个打算在应用程序中用 MFC VC++执行的例子。

D.1　设置Developer Studio

D.1.1　激活其他应用程序

Developer Studio可在Tools菜单中添加定制菜单命令，在环境内运行其他应用程序，方法如下：

1) 单击Tools/Customize菜单命令，以打开Customize属性表。

2) 选择Tools选项卡，并滚动到菜单列表的底部 (Developer Studio应该已经在该菜单中预装了几个命令)。

3) 选中底部的空白焦点矩形，并输入需要在菜单中显示的名字；按回车键将打开三个编辑框，分别是Command、Arguments和Initial Directory。

4) 在Command中，输入要运行的执行文件的路径名。

5) 在Arguments中，输入传递给执行文件的参数。

6) 在Initial Directory中，输入运行该命令前Developer Studio应该改变的目录。

可把下面的命令添加到Tools菜单：

Explorer**命令**　使Windows Explorers最初在工程目录中打开。菜单名字输入到Explorer中。命令输入C:\Windows\explorer.exe或C:\Winnt\explorer.exe(根据操作系统而定)，参数输入/e，并使初始目录(Initial Directory)为空白。

DOS**命令**　打开一个初始在工程目录下MS-DOS窗口。菜单命令输入MS-DOS，命令输入

C:\Winnt\System32\cmd.exe或C:\Windows\command.exe(根据操作系统而定)，没有参数，在初始目录中输入$(当前目录)。

Editor命令 打开一个第三方文本编辑器，并传递当前选中的文件名到该编辑器；虽然环境提供的文本编辑器是相当有效的，但市场上还有一些更加好用的编辑器。在菜单命令中输入Editor，在命令中输入编辑器的路径，在参数中输入 $ (FileName) $ (FileExt)，在初始目录中输入$ (WkspDir)。当执行任何一个这样的菜单命令时， Developer Studio保存所有的文件，以便即使在调用该外部编辑器前修改了一个文件，打开该文件时，还是可以编辑它的最新版本。

D.1.2 工具栏按钮

Developer Studio允许配置工具栏中的按钮。

1) 单击Tools/Customize菜单命令打开Customize属性表，并选择Commands选项卡。

2) 可以把按钮组框中所示的按钮拖到一个已有的工具栏中，或者拖到一个空白空间以创建一个新的工具栏。

3) Category组合框改变可选的按钮选项。

4) 要删除一个按钮，把它拖回到 Property Sheet中。

单击Command选项页上的每个图标将显示它所代表的命令的说明。通过查看菜单本身，用户可以发现哪个预定义的图标与哪个预定义的菜单命令对应；显示在菜单命令边上的图标是它们的工具栏按钮。

工具栏按钮可能代表的命令如下：

- 添加到Tools菜单的任何命令，通常以 7开始。因此，在该页的 Tools目录中找到一个按钮图标，它具有一个小铁锤和一个 7在第一个命令的右下角。
- ClassWizard，该命令可以在View目录下找到，一个具有巫士魔术棒的三角形。
- Open Workspace…命令是一个最常使用的命令，但没有默认的工具栏按钮；可以通过选择All Commands目录为该命令创建一个按钮；在 All Commands目录中，可以看到所有Developer Studio命令的一个详细列表；滚动目录直到接近底部找到 Workspace为止；拖动该选项到工具栏上的一个位置。因为该命令没有一个默认的工具栏图标与之关联，Studio会提示一个可以分配给该按钮的常规图标的列表；还可以选择在按钮中包括菜单命令的名字。

注意 按自己的意愿定制Studio后，应立即退出，因为对工具栏和菜单所做的改变直到退出Studio时才能保存，如果Studio在正常退出前崩溃，则所有改变将无效；此外，还要注意一次运行多个 Studio实例的情况，确保最后一个退出的 Studio是需要改变的 Studio。因为，最后退出的Studio决定最终改变Studio配置。

D.2 调试

大多数调试工作可以通过 Studio的Text Editor界面完成，参见MFC文档可获得更详细的内容。然而，还有其他一些资源可以用来调试代码，下面介绍一些。

D.2.1 TRACE ( )

如果应用程序是时间或序列敏感的，可能需要打印出调试消息，而不用交互式调试器单

步执行通过一个关键部位；过去，程序员用 printf ( )语句把调试消息打印到打印机或磁盘文件上；现在应用程序框架提供了一个与 printf ( )命令相当的TRACE ( )函数，但是TRACE ( )直接打印到Studio的Debug窗口。它要求打印字符不超过 512个，一个例子如下：

注意 在语句结束时一定要用换行符(\n)！

```
TRACE("Integer=%d, String=%s\n", i,sz);
AfxTraceEnabled=FALSE; //turns off all TRACE() statements
```

注意 TRACE ( )语句在应用程序发行版中有问题时，如果问题只出现在发行版中，则必须改用printf ( )，还可以用下面即将讨论的AfxMessageBox ( )。

D.2.2 AfxMessageBox ( )

如果问题只出现在没有安装 Developer Studio的操作系统上的应用程序中，或只出现在应用程序的发行版中时，除了在软件中用 printf ( )外，还可以考虑使用 AfxMessageBox ( )；可以格式化一个消息或创建一个静态消息，用以指示应用程序通过一个关键部位时的过程，并用 AfxMessageBox ( )显示这些消息； AfxMessageBox ( )甚至可以不必在一个控件窗口的类中，或者根本不必在一个类中。

D.2.3 ASSERT ( )和VERIFY ( )

另外两个用来检查工作的全局函数是 ASSERT ( )和VERIFY ( )，这些宏通常用来检查传送到一个函数的参数是否合法。如果这些参数来自用户的输入，可能还要用 AfxMessageBox ( )显示某种错误消息；但是，如果该参数源于应用程序的其他部分，则可以使用具有更小系统开销的ASSERT ( )和VERIFY ( )，MFC库用ASSERT ( )和VERIFY ( )防止任何愚蠢的错误导致奇异的结果。

在应用程序的调试版中， ASSERT ( )和VERIFY ( )核查一个合法的C或C++表达式，如果不是TRUE，则显示一个错误消息并退出应用程序；在应用程序的发行版中， ASSERT ( )不做任何事情 —— 甚至不核查括弧内的表达式， VERIFY ( )虽然核查表达式，但如果发现错误，还是不能停止应用程序的发行版。这两个宏的例子如下：

```
ASSERT(this)            // if false, debug application aborts
                        // statement not evaluate on release
                        // version
VERIFY(GetFile (...))   // if false, debug application aborts
                        // statement still evaluated in release
                        // version
```

D.2.4 Dump ( )

每个从CObject派生的类都继承一个Dump成员函数，该函数在概念上与 Serialize ( )相似，但是，它可以输出类中所有可读格式的成员变量，可以转储文档中的一个或所有对象，方法与用Serialize ( )串行输出所有数据对象到磁盘 (见例67)一样；但是，与Serialize ( )不一样的是，不是把存档类的实例传递给 Dump ( )，而是把一个转储环境的实例传递给它。可以用默认的afxDump转储环境，它将输出显示在 Studio的调试窗口中，或者可以用一个文件：

```
CFile file;
file.Open("dump.txt", CFile::modeCreate | CFile::modeWrite);
```

```
CDumpContext dc(&file);
```

然后，用下面的语句转储一个对象：

```
pObject->Dump(&dc);
```

如果要实现一个完全转储系统，可以用菜单命令调用第一个 Dump ( )函数。然后，文档中的所有其他Dump ( )函数以递减的模式被调用。

一个Dump ( )例子如下：

```
void CMyData::Dump(CDumpContext &dc) const
{
   CObject::Dump(dc);
   dc << "Item 1 = " << m_nItem1;
   dc << "Item 2 = " << m_nItem2;
   dc << "Item 3 = " << m_nItem3;
   :  :  :
}
```

强调一下，该函数只在应用程序的调试版可用。

D.2.5 例外

在Studio的调试窗口中可能会偶尔看到下面类型的错误：

```
First-chance exception in xxx.exe(KERNEL32.DLL):
        0xE06D7363:Microsoft C++ Exception.
```

这表明一个例外插入MFC库的try { } catch { }配置中，并最终被处理了。如果想自己查看什么导致了例外时，则可以告诉调试器停在该类型的例外处，而不是让它在catch { }中自动处理。

1) 开始时，在调试模式下运行应用程序，以显示 Debug菜单命令。

2) 然后，单击Debug / Exceptions菜单命令以打开Exceptions对话框。

3) 在Exceptions对话框中，可以找到所有例外的一个列表，选择列表底部的 Microsoft C++ Exceptions，然后单击Stop Always 。现在，每当发生一个MFC C++例外时，调试器都将停止应用程序，并定位在例外之处。

1. GetLastError ( )

MFC使用的或你直接使用的一些 Windows API调用除了不能产生期望的结果外，在失败时也没有解释。但是可以通过调用 ::GetLastError ( )访问少数API调用的错误(具体哪一个参考MFC文档)，返回的错误代码可以在MFC文档中找到。

2. AfxCheckMemory ( )

对于偶尔发生的破坏堆内存的问题，可以用 AfxCheckMemory ( ) 正确指出该问题。AfxCheckMemory ( )检查堆内存的合法性，如果没有问题，该函数返回 TRUE，习惯上可用下面的语句：

```
ASSERT(AfxCheckMemory())
```

把该行插到有问题的地方，并期望它捕获一些。

3. CMemoryState

MFC的CMemoryState类可摄下全局堆的系列快照供以后比较，帮助跟踪内存泄漏，一个典型的用法如下：

```
CMemoryState msBefore, msAfter, msDif;
```

```
msBefore.Checkpoint();

   // allocate and deallocate memory using "new" and "delete"

msAfter.Checkpoint();
msDif.Difference(msBefore, msAfter);
msDif.DumpStatistics();   // displays leaks
```

注意，我们需要该类的三个实例来完成比较。

D.2.6 第三方错误检查程序

Developer Studio的调试器可以做一些有限的内存泄漏检查，但是不能进行资源泄漏检查，资源泄漏发生在创建一个设备环境或画笔对象，并在使用后没有销毁时。要真正地彻底检查程序泄漏，应该考虑使用一个第三方泄漏检查程序，如 BoundsChecker™或Purify™。这些应用程序不仅可以检查资源泄漏，而且还可以指出其他类型的问题，否则，这些问题可能要几天的时间才能发现，如memcpy ( )在它不应该写的地方写了。

BoundsChecker使用简单，我个人喜欢用。只要在该框外运行程序，它便会输出一些错误消息，其中的一个可能是真正的问题，其余消息可能是假的。用这种类型的应用程序的一个内在问题是：它不断地提供错误线索。例如， BoundsChecker的一个主要任务是过滤出所有来自不能改变的 MFC代码的已知错误消息；关于这一点，在使用 Purify时是相同的。但是，Purify确实能处理一些BoundsChecker不能处理的事情，两个程序似乎都能发现对方不能发现的问题，因此，理想的情况是最好两者都使用。

D.2.7 Dr. Watson(华生医生)

有关调试应用程序的最后一个且最不重要的资源是内建在操作系统中的程序错误调试器，这是显示在屏幕上的小窗口，在一个应用程序有一个页错误或试图访问禁用内存时显示，并要求必须单击Yes销毁前面的工作；在一个没有安装Developer Studio的系统，只显示注册或堆栈错误。在大多数情况下，这些值无助于分析任何事情。当系统安装了 Developer Studio时，则有机会把应用程序调入到一个调试会话中。然而，如果应用程序是一个发行版时，这样做近乎无用，除非懂得汇编语言并花上几个小时。

D.3 规划并建立工程

如果应用程序是一个单一的执行文件，则可以忽略这部分的讨论，因为建立应用程序只需单击Build All按钮；然而，如果应用程序包括一些可执行文件，怎样建立应用程序则需要规划，例如，如何使应用程序或库A在库B前建立，以及库C可以找到D产生的包含文件和库文件。

当规划一个包括多个工程目录的大型应用程序时，则将面临怎样使一个工程访问另一个工程的包含文件和库的逻辑问题。一些冒失的人会把所需的文件简单地拷贝到所需的目录下，这只会提出另一个逻辑难题，即确定目录中的哪个文件属于哪个工程或反之。如果不属于，它们是最近的版本吗？此外，如果几个工程需要访问同一个库时，会有每个文件的几种可能版本存在吗？

D.3.1 使用\Include和\Lib目录

一种更合理的方法是创建两个工程可以共享的通用目录： \include和\lib。显然， \include

目录有包含文件(.h文件)，\lib目录，除了有库外，还有任何 DLL文件和调试库的调试特征表(.lib、.dll和.pdb文件)。

当运行一个要求 .dll文件的执行程序时，操作系统将在与执行文件相同的目录中寻找这些.dll文件；如在那里没有找到，则在系统的标准执行路径中寻找；因此，还必须把 \lib目录添加到系统的标准路径下；在 Windows 95和98系统中，要在系统的根目录下编辑 autoexec.bat文件，并添加\lib目录到PATH＝语句中。对于Windows NT系统，要在控制面板上运行系统配置文件，并把它添加到PATH＝语句中。

注意 在两种情况下，确保重新启动系统，否则，改变不起作用。

D.3.2 重命名调试版

这将引起下面的问题：因为库现在共享相同的目录，则必须给予调试库一个与发行库不同的名字；通常可以在调试库文件名后面加 d；例如，使mylib.lib调试版为mylibd.lib，这可以在Project Settings对话框中完成；用户还需在工程中创建一个附加的 .def文件以包含 .dll调试版的名字，只要把已有的 .def文件拷贝到添加了 d的新文件(如mydll.def拷贝到mydlld.def)，然后添加一个d到包含在新的.def文件中的名字；当把该新的 .def文件包括到工程中时， Studio将声明，在一个工程中不能有多个 .def文件；使用Project Settings，选取旧的.def文件，并把它从工程的调试版中去掉，然后，把该新的 .def文件包括在调试版中。

D.3.3 Post Build Step

通过添加一个DOS命令到Project Settings对话框的Post Build Step选项卡中，可以使合适的文件自动加入到 \include和\lib目录中。该命令可以是简单的 COPY语句，拷贝工程的\Release或\Debug目录到\include或\lib目录中。对于DLL工程，拷贝.lib和.dll文件；对于调试版，还要包括.pdb文件，它包含调试器所需的与 DLL进行源级调试会话的内容。

D.3.4 Developer Studio目录

通过Options属性表可以告诉Developer Studio在哪里找\include和\lib文件，要打开Options属性表，单击Tools/Options，然后，选择Directories选项卡，在Show Directories组合框下面可以看到Include文件和Library文件的输入项；在这些列表中添加 \include和\lib目录，然后退出Developer Studio(先关闭Developer Studio的所有其他实例)以保存这些添加的内容；现在，当建立应用程序时， Developer Studio的编译器和连接器也将搜索这些目录；当然，假定已把这些库添加到工程的设置中(见例84)。

D.3.5 版本控制软件

笔者大力推荐使用版本控制应用程序，这不仅仅为了保护资源，还为了轻松地创建更早版本的应用程序。两个好用的版本控制应用程序是 Visual Source Safe和PVCS。

D.3.6 Build Machine和Batch File

一台公平的机器通常被设计为一台“建造”机器，这通常也是一台文件服务器和版本控制应用程序驻留的机器；因为一个建造可以永远占据，尤其当应用程序完善并呈现更多的特

征时，极力推荐使用一个批处理文件建造整个应用程序、库和可执行文件。当一个较长的处理过程要求一些简单的但是关键的步骤时，一个批处理文件能消除人为错误；一个批处理文件保证每次应用程序按相同的方法创建，并在应用程序行动异常时自动取消建造。建造批处理文件通常有三个步骤：

1) 消除旧的建造——删除\include和\lib目录下的所有文件。

2) 用版本控制应用程序抽取正确的软件版本；通过 Visual Source Safe，可以用下面的方法来完成；也可以取出其他版本以创建应用程序的更早的版本。

```
cd \develop\project1
ss Get $/Project1 -R
ss Get $/Project1 -R -V2.3 -- gets version 2.3
```

3) 按下面要求的顺序建造每个工程，输出任何一个错误到以后可以观看的文本中，注意赋予cfg=“ ”声明的字符串必须与.mak文件中的相同。

```
nmake -fProject1.mak -xProject1.err cfg="Project1 - Win32
   Release">>err.err
```

这将引起质问，从哪里得到一个 .mak文件?更早版本的环境自动产生它们，然而，最新版本使它成为用户必须设置的一个选项。

4) 单击Developer Studio的Tools/Options命令，并选择Build选项卡。

5) 然后，在保存工程时，确保设置了 Export makefile选项。

你可能不想为大型工程做这样工作，因为 makefile输出减慢了单个工程的保存；而更希望每当工程改变时，简单地创建一个新的 makefile；要用命令创建一个 makefile，只需单击Project/Update Makefile…。

当在Developer Studio外面，以批处理模式建造时，则在 Developer Studio之内设置的目录不再适用；编译器和连接器现在将使用定义在操作系统环境中的路径；因此，要再次使编译器和连接器知道公共的\include和\lib目录在哪里，必须把下面的语句添加到Developer Studio中。

```
lib=\lib
include=\lib
path=\lib (so that dll files can be found)
```

对于Windows NT，必须用控制面版上的System应用程序改变系统变量，并应重新启动计算机。

D.3.7 Build Machine和Developer Workspace

前面已提过，更新版本的Developer Studio不能自动产生一个.mak文件，但是，它们使用两个新文件：一个 .dsp文件(它跟踪单个工程)，和一个.dsw文件(它跟踪工作空间)。一个工作空间可能包含几个工程，这些工程之间甚至相互具有依存；因此，如果使用更新版本的Developer Studio的话，可能还想用这些应用程序按下面的步骤创建一个大工程。

1) 假设已有一些.dll或.exe工程子目录，用Developer Studio打开主.exe工程的.dsw文件。

2) 在Developer Studio的Project菜单中，用Insert Project into Workspace命令添加所有其他应用程序到该工程的工作空间中。

3) 再次在Project菜单下，用Dependencies…命令设置整个工程的所有依存。例如，选择一个.exe工程，然后单击所有必须在该工程建立前要建立的 .dll工程。

4) 可以用Project/Set Active命令确定要求Developer Studio Build或Build All时建立哪一个工

程，如果想使主.exe工程成为主动的工程，则可以只通过单击Build或Build all重建整个工程。

5) 如果想建立整个工程的调试版和发行版，而在完成以后不必返回，可以用 Developer Studio 的Build菜单，选择Batch Build命令并选择一切以建造一切。

6) 该工作空间配置保存在 .dsw文件中，对于一个特别大的工程，可以考虑在 Build Machine上拥有一个包含整个工程的.dsw文件。

D.3.8 Build Kit

规划一个大型应用程序的最后一步是创建一个 Build Kit。一个Build Kit只是文件的一个选项，任何人可以用来创建一个新版的应用程序，这包括 \include和\lib目录下的文件，以及建库所用的源文件(可选)。Build Kit应当驻留在Build Machine中，并在建造下的一个单独的目录中，使Kit单独设置一个目录允许被新的开发使用，即使在一个新建造中间也可以使用。并且，如果新的建造不能正确地起作用时，在修复该建造时不延误任何他人的新工作；一旦一个建造被证实起作用后，另一个批处理文件可以用来把它拷贝到 Build Kit目录中。

D.4 Microsoft 命名习惯

Microsoft已为MFC C++编程时使用的程序变量设立了一种非正式的自愿的命名习惯。尽管这种习惯并不强求在创建一个 MFC应用程序时使用，也许并不总是实用的，但它确实有一些优点。除了能给一个变量常规的描述名以外，用这种习惯还可以分辨一个变量的类型以及是否一个类的成员变量。

以下面符号开始的变量含义：

m_	一个类的成员变量
m_b	布尔变量
m_n	整型变量
m_p	指针变量
m_dw	DWORD变量
m_classname	在classname中指定的类类型

批评这种习惯的人指出，这有悖于语言的灵活性，例如在命名变量 m_bFlag以后，现在要将它转变为整型，而不编辑所有出现的地方或不管它，是非常困难的，并有可能迷惑未来的软件开发者。然而，据笔者个人经验，很少需要改变变量类型，而且能够迅速区别一个变量做什么用，远远胜过它具有的不足。

D.5 Spying

在VC++开发包中可能被忽略的应用程序是 SPY.EXE，这个小应用程序不仅可以帮助你指出应用程序正在进行什么，而且正如其名字的含义一样，它能指出他人怎样在应用程序界面中操作窗口。

D.5.1 发现消息

可以读到想知道的所有关于哪一个窗口消息对于应用程序中的哪一个窗口是可用的，但是，当所有应用程序的界面成份都被叠在一起时，每个人都会猜测什么消息是可用的。要确

切地发现这些消息，用SPY.EXE应用程序截获并显示每个可视窗口的所有消息。

1) 运行应用程序开始。

2) 然后执行SPY.EXE(这可以在MFC C++的\bin目录下找到)。

3) 单击Spy的Spy/Message菜单命令，打开Message Options属性表。

4) 找到Finder Tool，并把它拖到想要监视的窗口。

一个新的窗口将在Spy内打开，并开始显示进出那个窗口的窗口消息；从那里，用户可能会看到一个未曾想过要处理的窗口消息。

D.5.2 发现窗口

用那个相同的Finder Tool，有时可以猜出他人怎样用他们的界面实现一个作用；例如，可以确定该效果是否是以前不曾用过的窗口风格，还是其他应用程序开发者自己开发的。如前面一样抓住Finder Tool，但是现在把它拖到当前考虑效果的窗口中，下面窗口的大小将被突出显示，并显示出它的类和风格。有时，你可能为所发现的效果而惊讶！

D.6 其他实例资源

如果不能在本书中找到一个需要的例子，在本书之外还有其他例子资源，如你的 VC++发布工具CD带有它自己的例子目录；另外一个优秀的，但是不常用的例子资源是创建 MFC类库的源文件集，可以在VC++的\mfc\src目录下找到，这里也是查找非文档化虚拟 MFC函数的地方。

另一个例子资源可以在Internet上找到：

ftp://ftp.microsoft.com/Softlib/MSLFILES/

但是必须在3A.M.时间访问它，并且它只限2500个用户访问。