第二部分

Visual C++与数据库管理

返回总目录

目　　录

第 4 章　ADO 与 ODBC

有很多种使用数据库的方法，对大多数数据库来说，选择 C++这种产品也许并不适宜。我们知道，像 dBASE IV，FoxPro，Oracle 和 Access 这样的产品是完全以数据库管理为中心的。事实上，这些产品非常善于创建数据库管理器，以至于它们确实并不善于做太多其它的工作。即使要用更通用化而非更专用化的数据库产品来执行一些类型的工作，在使程序设计更容易这一方面，像 Visual Basic 和 Delphi 这样的 RAD 环境也要比 Visual C++强很多。

你是不是对我的说法感到很奇怪？下面我就要谈一谈，在谈到使用数据库管理系统(DBMS)这个话题时，用 Visual C++实际上可以做些什么。虽然上述其它语言使得编写成熟的包括用户界面和高速搜索能力的 DBMS 就像孩子做游戏一样容易，但是，它们缺少 Visual C++可以提供的某些重要东西。你不能为使用 Access 的数据库轻松地编写出实用程序。正像实用程序的定义所说的，实用程序应该很小并且具备可移植性——Access 应用程序却不是这样。即使用 Access 这样的产品创建的程序可以很小并且可以移植，你仍有其它方面的需求：底层的功能。

注：编写数据库实用程序及驱动程序时，可以选择 Visual C++语言。

想像一下，使用像 Visual Basic 这样的语言来与实时数据采集设备打交道的情况。在进行底层访问时，RAD 的保护环境常常使程序员不能进行有效的处理。当然，数据采集设备几乎不依赖于简明的连接。你打算如何把 Visual Basic 和外

部的数据源连接起来呢？数据源甚至可能不了解 Windows，DOS 或类似的成熟的操作系统。

只要使用得当，很容易看到 Visual C++是一种不可或缺的数据库管理工具。针对大规模的应用程序，即使你仍想依赖于 Visual Basic 这样的 RAD 语言，也请考虑一下 Visual C++，它创建的程序规模小、提供底层访问并能提供实时访问。事实上，你可能还没有想到，Visual C++数据库应用程序的市场是很有潜力的。随着人们在旅途中越来越多地使用膝上型和掌上型电脑，这两类电脑上的数据库应用程序也变得越来越普通。你也许能够适应今天的膝上型电脑上的 Access 应用程序，但谈到硬盘大小或内存需求时，公司里较老的膝上型电脑可能就达不到要求。运行 Windows CE 的掌上型电脑在运行这个 Access 应用程序时，肯定会发生故障。在这一数据库市场的新领域，Visual C++提供了无价无限的工具。

Web 链接 谈到使用 Visual C++和数据库，其实你并不孤单。从一开始就有数据库专用新闻组提供有关数据库创建技巧的帮助，比如 microsoft.public.access。不过，这些新闻组提供的是通用信息，对实际编写应用程序并非全都那么有用。专门针对 Visual C++问题的新闻组是 microsoft.public.vc.database 和 microsoft.public.vc.mfcdatabase。如果你决定用 ODBC 访问数据库，可能还要查看一下 microsoft.public.odbc.sdk 新闻组，它讨论的不仅仅是 SDK。对最新技术感兴趣的程序员可以查阅 microsoft.public.ado 新闻组，或者 microsoft.public.oledb（对

象链接和嵌入数据库）新闻组，前者讨论 ADO，后者讨论 ADO 的基础技术。在 microsoft.public.ado.rds 有一个 ADO 子组，它讨论远程数据访问。

既然所有的疑惑都消除了，大多数人的信心也就增强了，下面我们就介绍两种使 C++访问数据库中的数据的主要方法：ODBC（开放数据库互连）和 ADO（ActiveX 数据对象）。在本章中，将介绍这两种类型的访问方法，但我想你会发现，ADO 方法是针对新的程序设计情形而采用的。它克服了早期技术的诸多限制，依赖于 Microsoft 新的底层访问方法 OLE-DB（对象链接和嵌入数据库）。在本书的后面我们会看到，用 ADO 和 Visual C++提供的各种向导来汇集数据库工程，其速度有多快。

注 ODBC 通常用来访问不具备 OLE-DB 特性的非 Microsoft 数据库中的数据；16 位的 ODBC 驱动程序工作起来可能非常缓慢。

ODBC 素以最慢的数据访问方法而著称，但是很可惜，当 ADO 或 DAO 都不支持某个数据库管理器而 ODBC 支持这个数据库管理器时，在这种特定的情形下，你仍然需要使用 ODBC。在大多数情况下，这意味着要从数据库厂商那里获得所需的驱动程序，虽然 Visual C++确实附带了一些产品的驱动程序（如果你正在使用数据库管理器的某些神秘功能，那么就需要建立自己的接口——这并不是一件十分困难的事）。本质上讲，你总是要使用 ODBC 来访问 Microsoft 产品之外的其它 DBMS 产品所创建的数据库，这些数据库并不具备 OLE-DB 功能。ODBC 还要求做一些额外的工作——为 ADO 调整 Visual C++中的大部分向导。

高级技巧

除了使用 ADO 和 ODBC 外，你还可以使用像 DAO（数据访问对象）这样的早期技术，该技术包含在像 Access 这样的 Microsoft 产品中。DAO 依赖于用 Microsoft Access 自动获得的 Microsoft Jet 数据库引擎。DAO 还是较早版的 Visual Basic 所使用的引擎（最新版的 Visual Basic 和 Visual C++依赖于相同的 ADO/OLE-DB 组合），所以如果需要支持较早的 Visual Basic 应用程序，那么 DAO 仍是一个不错的选择。

尽管 Microsoft 文件声明，可以用 DAO 访问非 Microsoft 产品建立的数据库，但你仍会发现，在这种情况下，使用 ADO 和 ODBC 要好得多。这样的话，不但兼容性问题会少一些，速度也将有所提高，因为数据请求经过的接口层减少了。有一条经验要记住，DAO 是设计用来处理 MDB 文件的。

DAO 的一个问题是，它不支持远程通信。这是 Microsoft 提出 RDO（远程数据对象）的原因之一。这种特别技术在 Visual Basic 应用程序中的使用，要比在 Visual C++中的使用多得多，所以我猜想，你们中有很多人都在使用它。但是，记住 RDO 仍是一种生命力很强的技术，这一点很重要。ADO 确实具有替代 RDO 的远程数据服务(RDS)特征。换言之，ADO 在一个软件包中提供了 DAO 和 RDO 两种功能性。

4.1 什么是 ODBC

开放数据库互连（ODBC)是 Microsoft 引进的一种早期数据库接口技术。它实际上是我们要在本章后面加以讨论的 ADO 的前身。Microsoft 引进这种技术的一个主要原因是，以非语言专用的方式，提供给程序员一种访问数据库内容的简单方法。换句话说，访问 DBF 文件或 Access Basic 以得到 MDB 文件中的数据时，无需懂得 Xbase 程序设计语言。事实上，Visual C++就是这样一个程序设计平台，即 Microsoft 最初是以 ODBC 为目标的。

你会发现，ODBC 工作起来和 Windows 一样——它用包含在 DLL 内的驱动程序完成任务。其实，ODBC 提供一套两个驱动程序：一个是数据库管理器的语言，另一个为程序设计语言提供公用接口。允许 Visual C++用标准的函数调用经公用接口访问数据库的内容，是这两个驱动程序的汇合点。当然，还有其它和 ODBC 有关的实用程序类型的 DLL。例如，一个这样的 DLL 允许你管理 ODBC 数据源。ODBC 的实际管理接口出现在 SYSTEM 文件夹中的某个 CPL(控制面板）文件中——我们在后面要谈到这方面的问题。

ODBC 的确能履行承诺，提供对数据库内容的访问，并且没有太多的问题。它没有提供数据库管理器和 C 之间尽可能最好的数据转换，这种情况是有的，但它多半能像广告所说的那样去工作。唯一影响 ODBC 前程的是，它的速度极低——至少较早版本的产品是这样。ODBC 最初面世时，一些开发者曾说，因为速度问题，ODBC 永远也不会在数据库领域产生太大的影响。然而，以 Microsoft 的市场影响力，ODBC 毫无疑问是成功了。今天，只要有两种 ODBC 驱动程序

的一种，那么几乎每一个数据库管理器的表现都会很卓越。

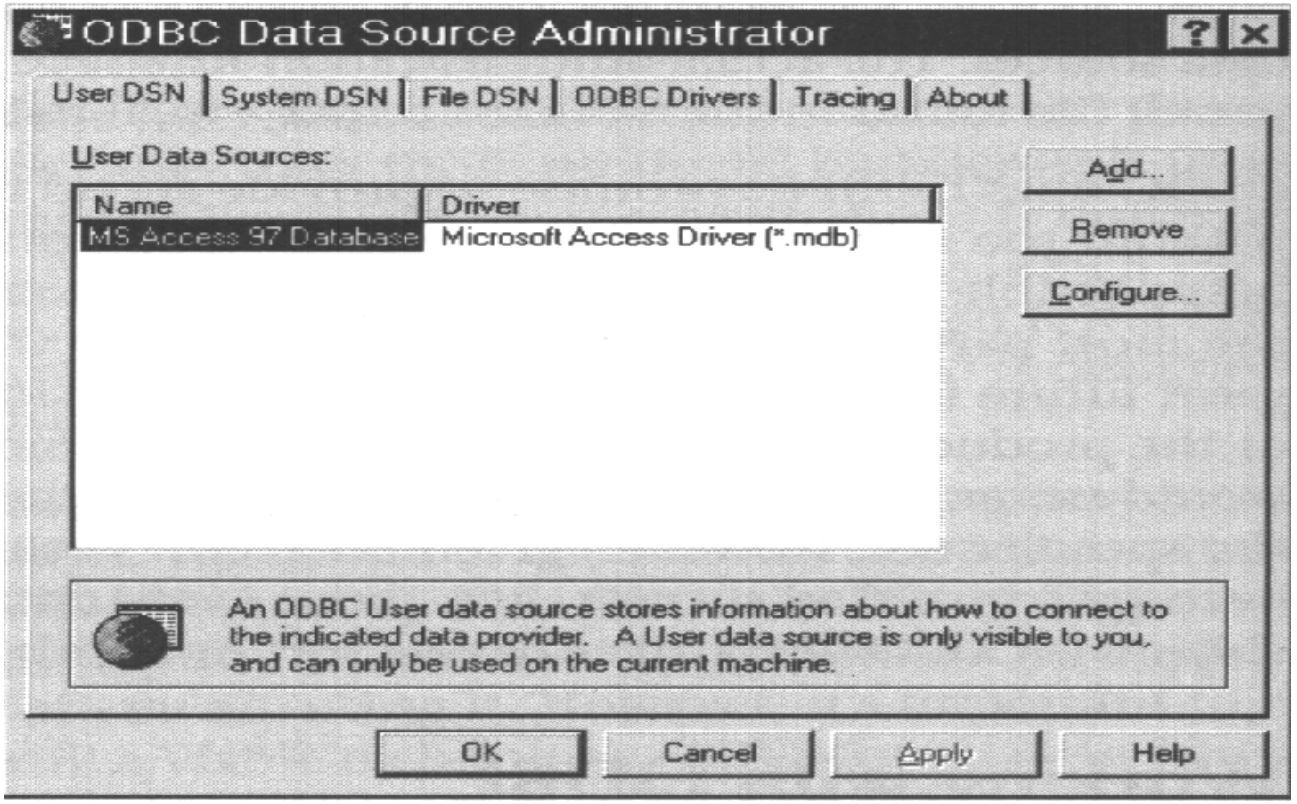
使用 ODBC

在可以用 ODBC 做任何事之前，必须有一个数据库——至少在脑子里。在一个像 Access 这样的应用程序中创建数据库框架通常会更容易一些，因为 Access 可以非常轻松地提供完成任务所需的各种功能。但是，如果你至少安装了所需的 ODBC 驱动程序，那就可以在 C++中创建程序框架。一设计了数据库，就要为其创建 ODBC 数据源，我们会在本节中看到这一点。下面的过程并没有严格地说明活动的经过，只是说明了一种配置数据源的技术。

注释 我会在第 5 章中说明如何创建一个 Access 数据库。现在，我们只是看一看，为了访问那个数据库，我们下一步要做些什么。

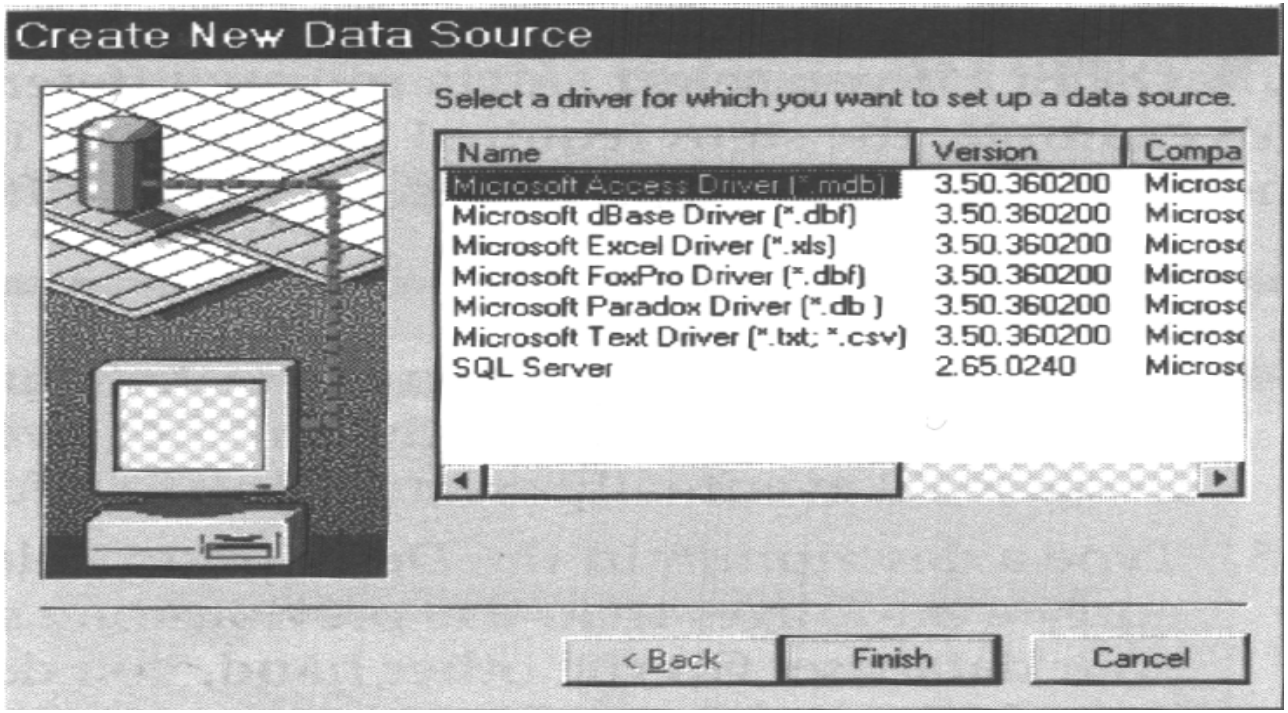
---

1. 双击 Control Panel（控制面板）中的 32 位 ODBC 小程序（如果当前系统上未安装 16 位驱动程序，那么一些版本的 Windows 使用简单的 ODBC 小程序）。你会看到 ODBC Data Source Administrator（ODBC 数据源管理员）对话框，如下图所示。请注意，该对话框中当前显示的选项卡是 User DSN（用户数据源名）选项卡。对系统级数据库来说，使用 System DSN（系统数据源名）选项卡；对文件级数据源（从严格意义上说，它不是数据库），使用 File DSN（文件数据源名）选项卡。你还会看到一个数据库的缺省选项。如果想从 Visual C++内部创建数据库，就会用到这个选项。



警告 对本地数据库来说，通常要在 User DSN（用户 DSN）选项卡上创建一个项；对远程数据库，则在 System DSN（系统 DSN）选项卡上创建。任何情况下，都不能在 User DSN（用户 DSN）和 System DSN（系统 DSN）选项卡上创建同名的项。通常会出现的问题是，你试图访问远程数据库，但从 Web 服务器获得了非常奇怪和矛盾的错误消息。事实上，如果在远程数据库访问期间得到了奇怪的错误消息，那么 32 位 ODBC 小程序是首先应该查看的地方之一。

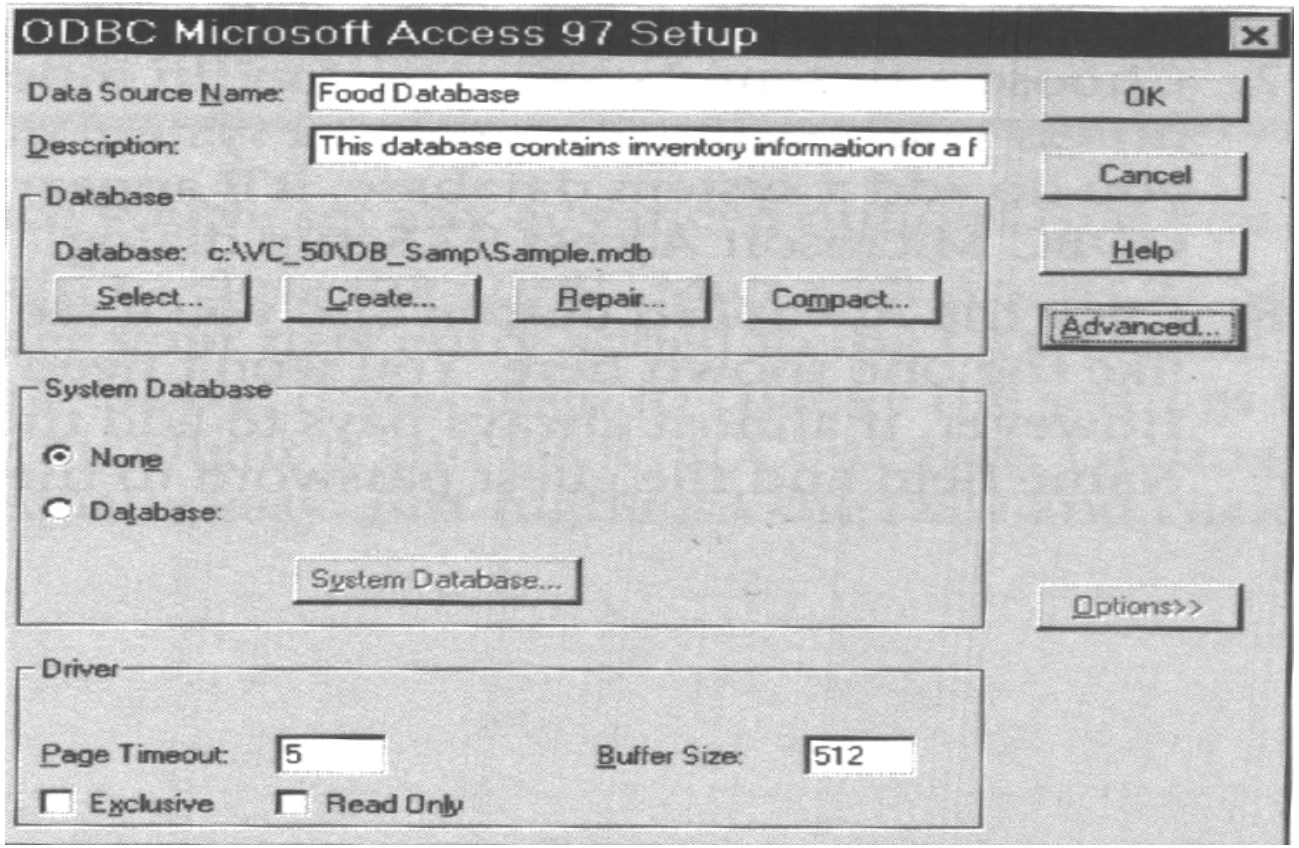
2. 单击 Add 按钮。会看到 Create New Data Source（创建新的数据源）对话框，如下图所示。



技巧 检查 ODBC Data Source Administrator（ODBC 数据源管理员）对话框的 About 选项卡，可以确定你正在使用的 ODBC 驱动程序的最新版本。这个选项卡包含了各种 ODBC DLL 的版本号、生产厂商的名称以及出现在 SYSTEM 文件夹中的文件名。大多数情况下，通过查看版本号可以验证 ODBC 驱动程序是否是最新的版本。

3. 选择一个数据源。对本练习来说，我选择了 Access 数据源。单击 Finish（完

成），会看到某种类型的配置对话框，如下面的 ODBC Microsoft Access 97 Setup（设置）对话框所示。



注释 如果你选择的数据源和我在本实例中选择的不同，那么所需的配置步骤也和这里说明的不同——每个 ODBC 驱动程序都要求不同类型的配置。

4. 在 Data Source Name（数据源名）域内输入数据源名称。一定要选择意

义明确但又不过于冗长的名称。我选择 Food Database（食品数据库），因为我最终要创建一个与食品库存有关的数据库的链接。

5. 在 Description（描述）域内输入一段说明性文字。可以让这个项比上一个项稍长一些，因为它描述数据库的用途。另一方面，也不要写入像《战争与和平》那样大的小说。对本练习，我输入了：This database contains inventory information for a food store（本数据库包含食品存储的库存信息）。

6. 单击 Select（选择）按钮。你会看到一个 File Open-type（文件打开类型）对话框，可以在那里选择一个现有的数据库。ODBC 驱动程序会自动选择正确的文件扩展名。

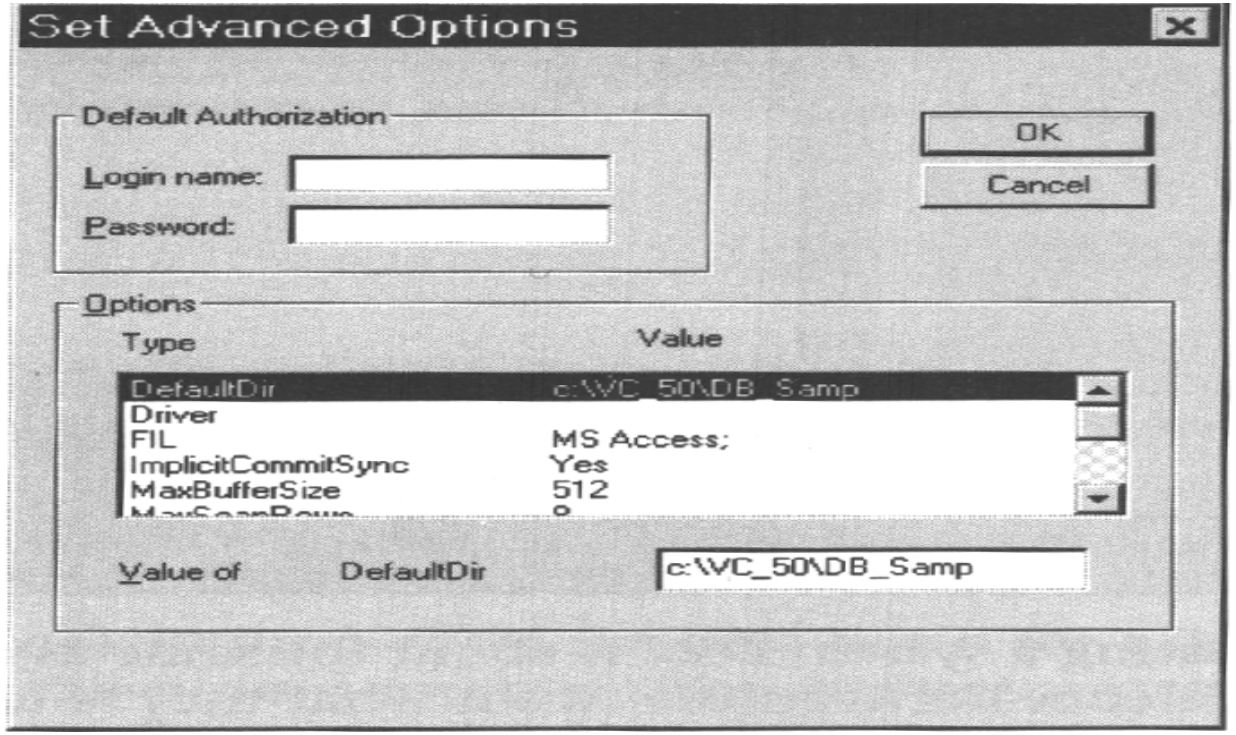
技巧 并不是一定要提前设计数据库。请注意，Access ODBC 驱动程序还包括一个创建新数据库的按钮。很多 ODBC 驱动程序都提供了这种功能，但并不是全都这样。单击这个按钮会启动数据库管理器应用程序，并允许你设计数据库。注意有一点很有意思，Access ODBC 驱动程序还会允许你使用这个对话框压缩或修补数据库。

---

7. 选择系统数据库选项。在大多数情况下要选择 None（无），除非你为应用程序特别创建了一个系统数据库。如果确实添加了系统数据库，它会出现在 ODBC Microsoft Access 97 Setup（设置）对话框的 System DSN（系统 DSN）选项卡上。

8. 单击 Advanced（高级）按钮，会看到 Set Advanced Options（设置高级选项）对话框，如下图所示。无需对很多项做修改。但是，要把客户名添加到 Login Name（注册名）域中，把客户口令添加到 Password（口令）域中。这允许客户

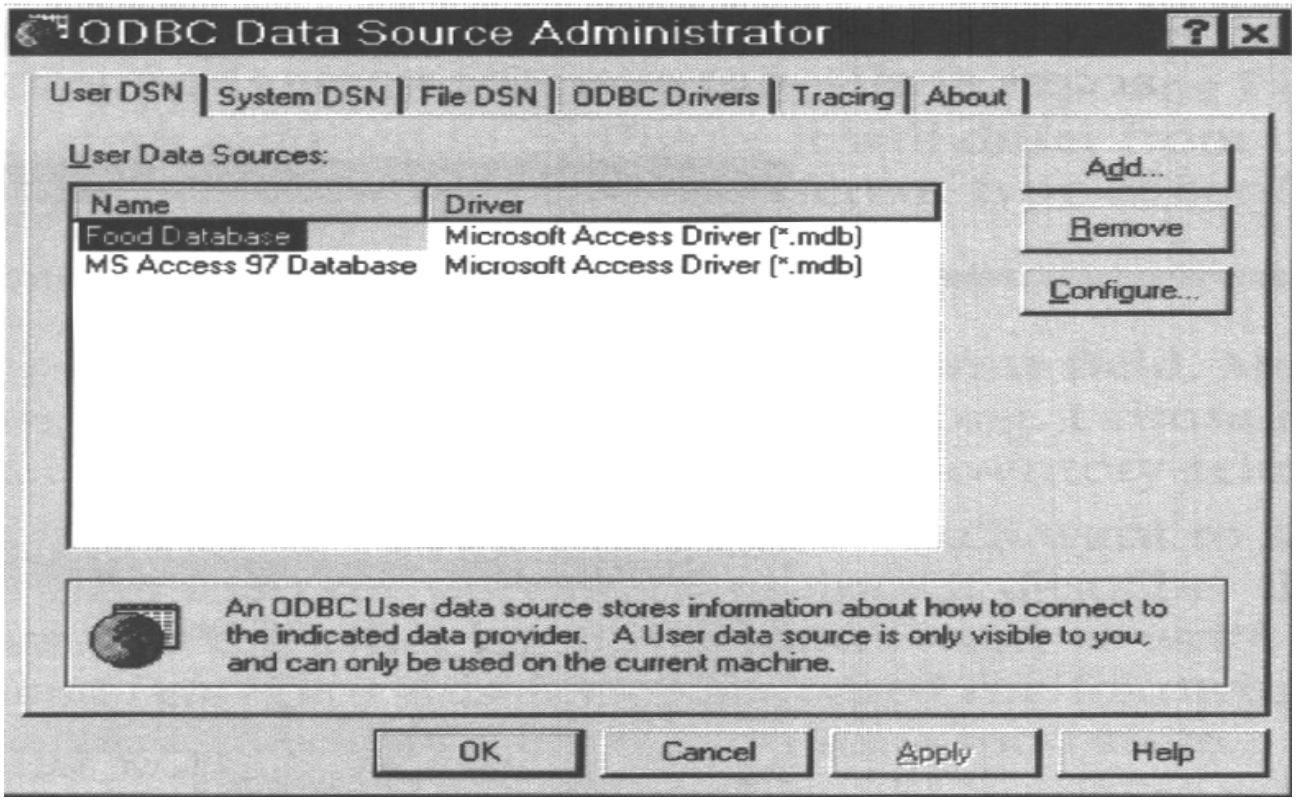
在访问你的数据库时，根本不用了解访问的细节——甚至不用了解被记录的客户名。



技巧 为最优化潜在的区域，可以浏览一下 ODBC 驱动程序提供的一系列高级选项。例如，Access ODBC 允许你更改 DBMS 所用的线程数量。缺省设置 3 通常提供了不错的性能，但是你会发现，复杂程序中线程多一些的话，可以提高前台任务的速度。由于 Windows 使用了一些处理器循环对线程实施管理，所以使用了过多的线程又会降低应用程序的速度。

9. 设置完所需的高级选项后单击 OK。

10. 再次单击 OK 关闭 ODBC Microsoft Access 97 Setup（设置）对话框。应该看到，新的设置项已经添加到 ODBC Data Source Administrator（ODBC 数据源管理员）对话框中。如果今后要为数据库更改这些设置，只要简单地加亮它并单击 Configure（配置）。删除数据库配置也很容易，只要加亮 DSN 并单击 Remove（删除）即可。



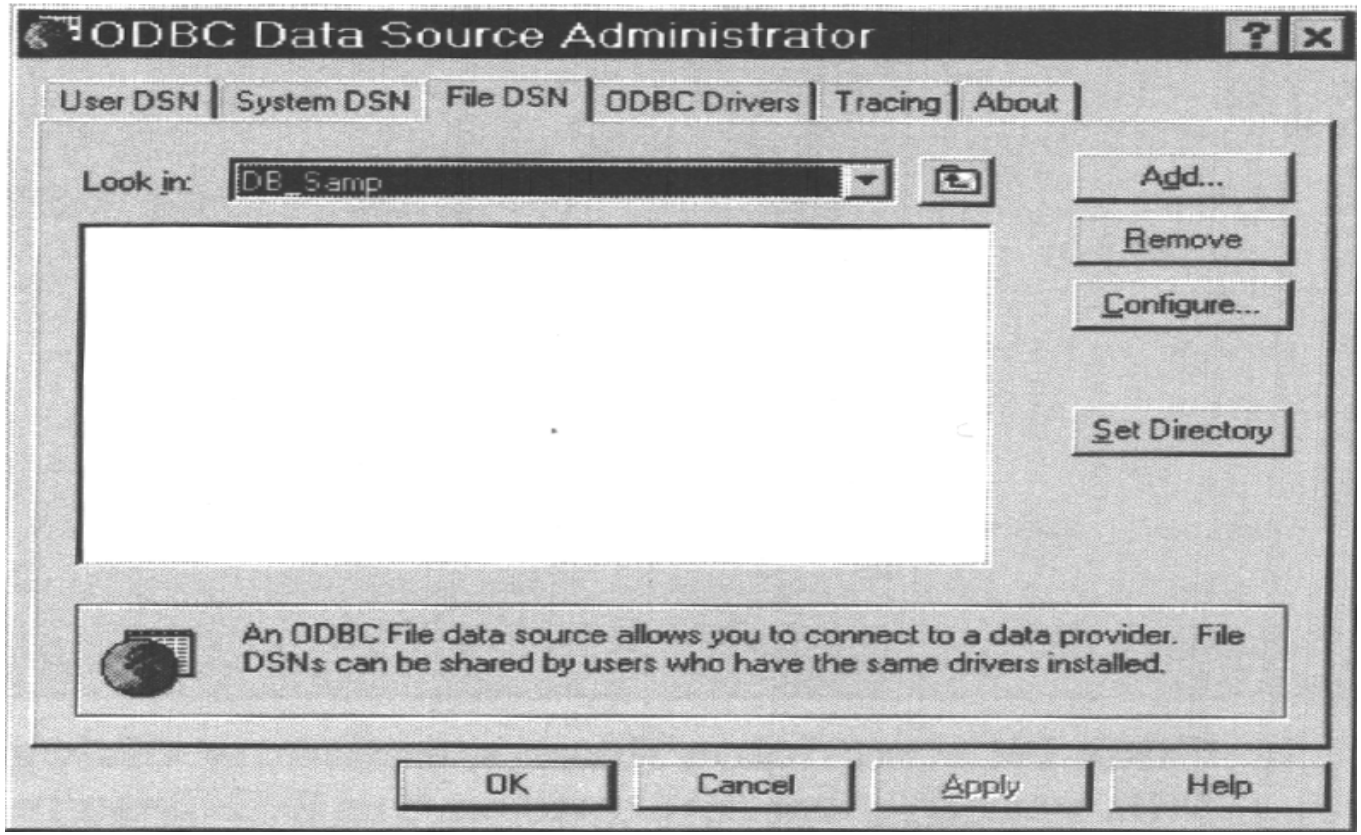
创建系统 DSN 和制作用户 DSN 差不多。两者间一个显著的差别是，使用

它们的目的不同。系统 DSN 告诉应用程序如何与数据库相连，在一些情况下如何与之交互作用。系统 DSN 不包含数据库所需的任何数据——它包含连接标准，其中可以包括从用户列表到重要文件位置的一切信息。

文件 DSN

你可能已经注意到了，上一节中的实例有问题。如果想单独配置网络上的每一台机器，它会工作得很好，但这可能不是一个好主意。还有一种存储创建数据源所需信息的方式：文件 DSN。这正是我们要在本节中讨论的。下面的过程说明了设置文件 DSN 的一般方法。

1. 双击 Control Panel（控制面板）中的 32 位 ODBC 小程序，会看到 ODBC Data Source Administrator（ODBC 数据源管理员）对话框。选择 File DSN（文件 DSN）选项卡，会看到如下图所示的对话框。要做的第一件事是，选择存储 DSN 信息的地方。



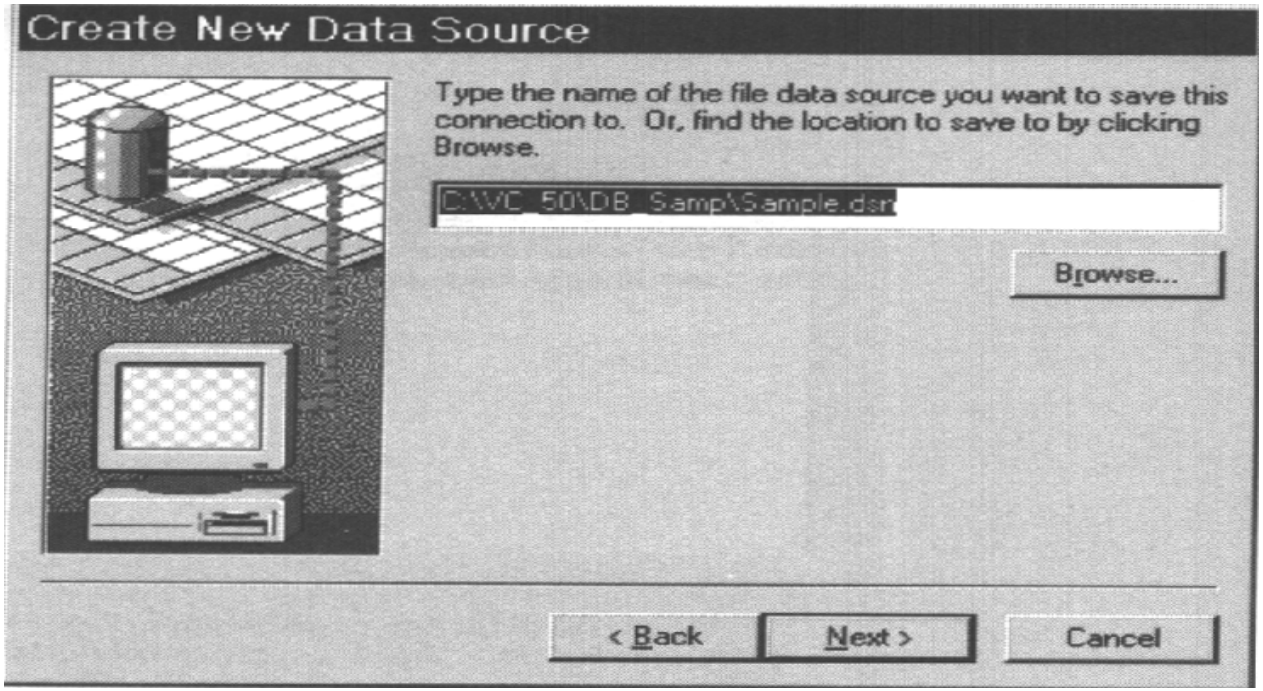
2. 单击 Look In（观察）下拉列表框，会看到一系列当前机器的目录和驱动器。可以为 DSN 使用任何存储位置。我通常在网络上选择数据库存储目录。使用 UNC（通用命名标准）意味着，每个人都会用相同的路径来访问 DSN 文件。

技巧 Up One Level（上一级）按钮（在 Look In 下拉列表框旁边）的工作方式和在 Explorer 中是一样的。可以用该按钮返回到上一级目录。最后，你会在 My Computer（我的电脑）处结束，看到机器上所有驱动器的清单。

---

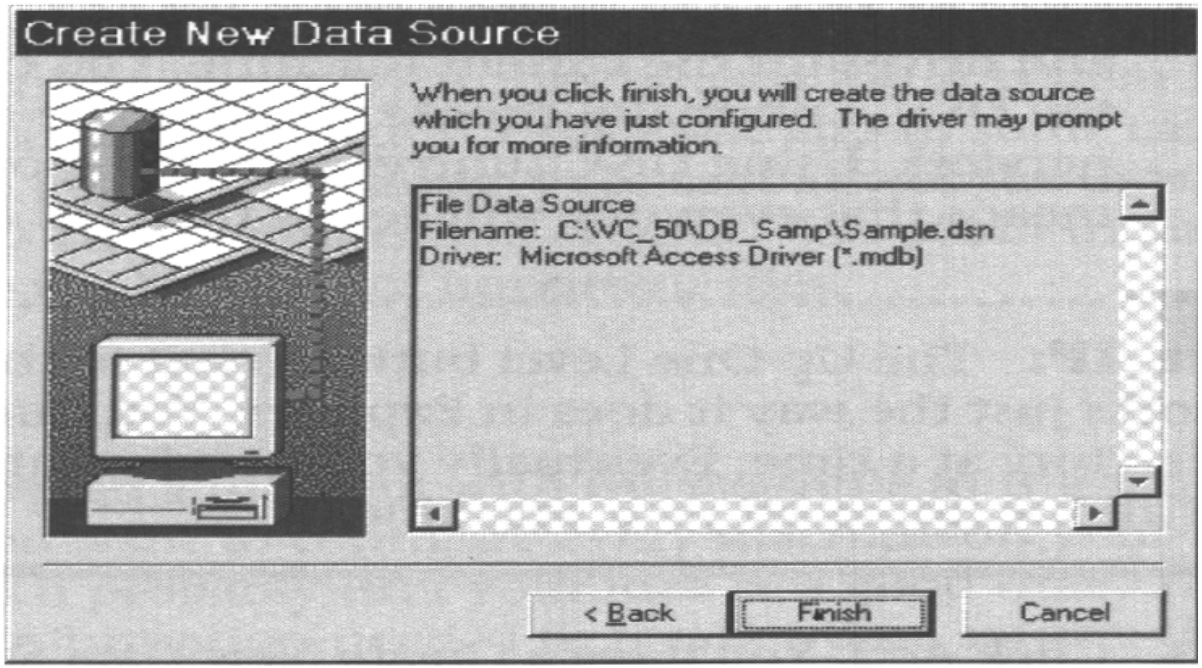
3. 单击 Add（添加），会看到 Create New Data Source（创建新的数据源）对话框。

4. 在列表中选择一个 ODBC 驱动程序，然后单击 Next（下一个）。对本示例来说，我再次选择 Access。你会看到 Create New Data Source（创建新的数据源）对话框的下一页显示出来。在这里选择数据源的名称和存储位置。单击 Browse（浏览）会看到 File Open-type（文件打开类型）对话框，在这里选择存储位置。输入一个文件名，ODBC 向导自动添加 DSN 作为扩展名。在本示例中，我选择 SAMPLE.DSN 作为 DSN 文件的名称。

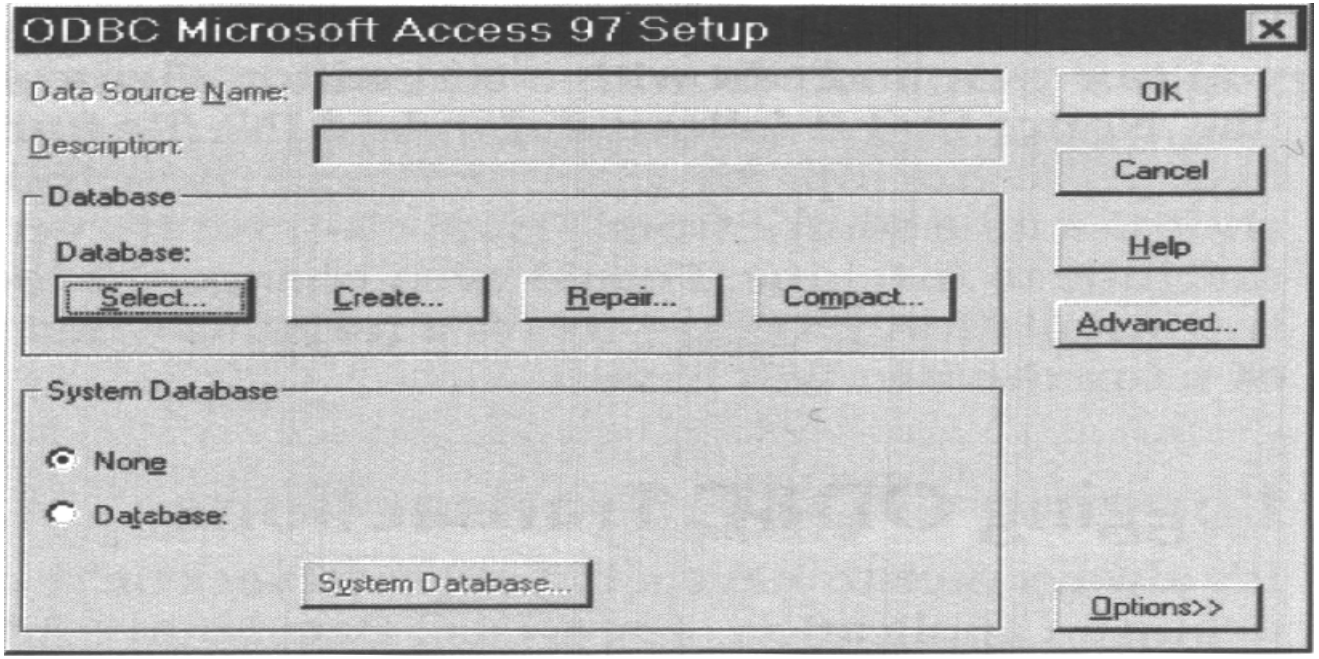


5. 单击 Next（下一个）会看到一个摘要对话框，如下图所示。它说明正准

备创建的 DSN 的参数。

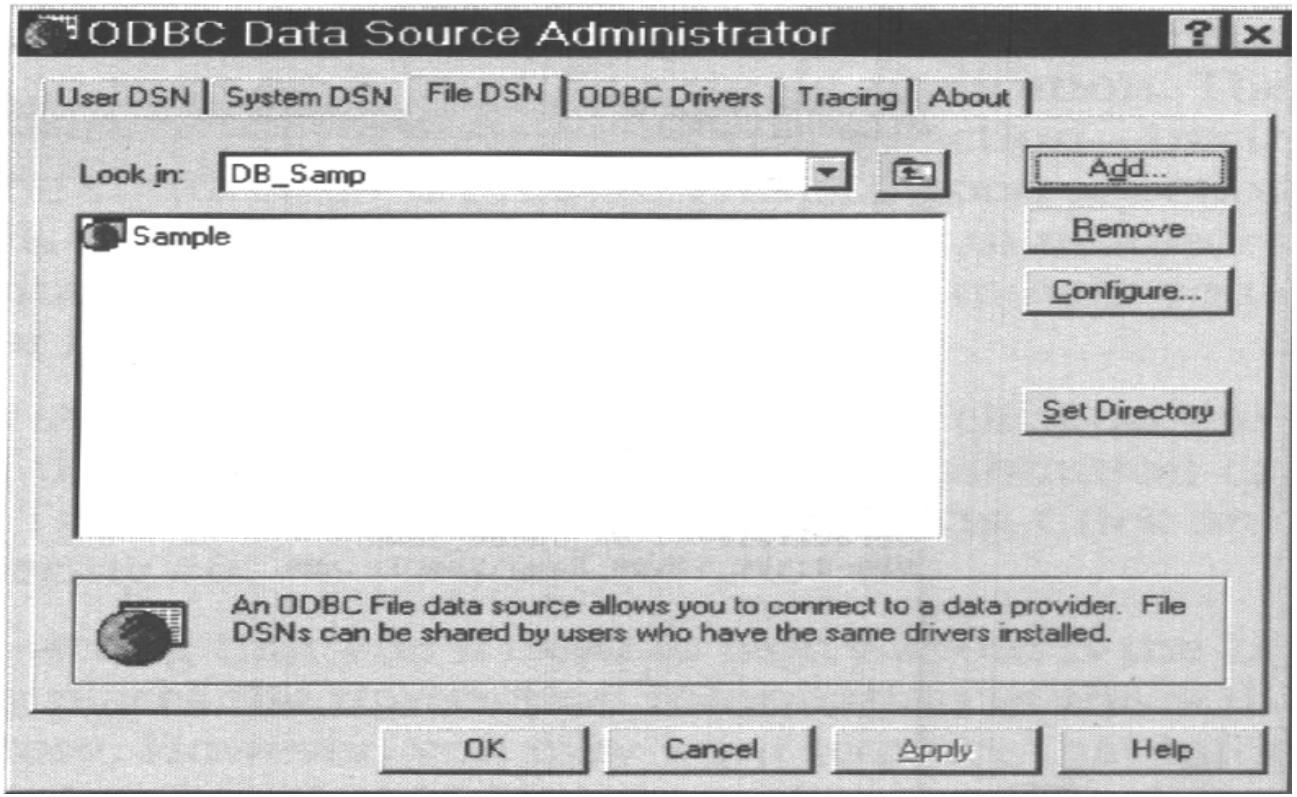


6. 单击 Finish（完成）。这时，会看到 ODBC Microsoft Access 97 Setup（设置）对话框的修改版。不能像我们在上一节所做的那样，在 Data Source Name（数据源名）或 Description（描述）字段中添加信息了，不过，其它的都和以前一样。



7. 一定要单击 Select（选择）按钮输入数据库的名称，然后选择想要使用的数据库（如果想创建一个新数据库，也可以单击 Create（创建））。

8. 完成配置过程时单击 OK，你会在 ODBC Data Source Administrator（ODBC 数据源管理员）对话框中看到一个新的 DSN 文件项。



和我们创建的前一个 DSN 不同，这个 DSN 实际上创建可以用文本编辑器来查看和编辑的文件。图 4.1 显示了我的文件的外观。请注意，它符合标准的 INI 文件格式。你可以在顶端看到[ODBC]标题。下面是我选择的全部设置。这个文件允许我从 Visual C++选择数据源，而机器间的传输也是非常容易的。我甚至能在安装过程中按要求更改位置——在你不知道用户会有什么样的设置时，这一点确实很不错。

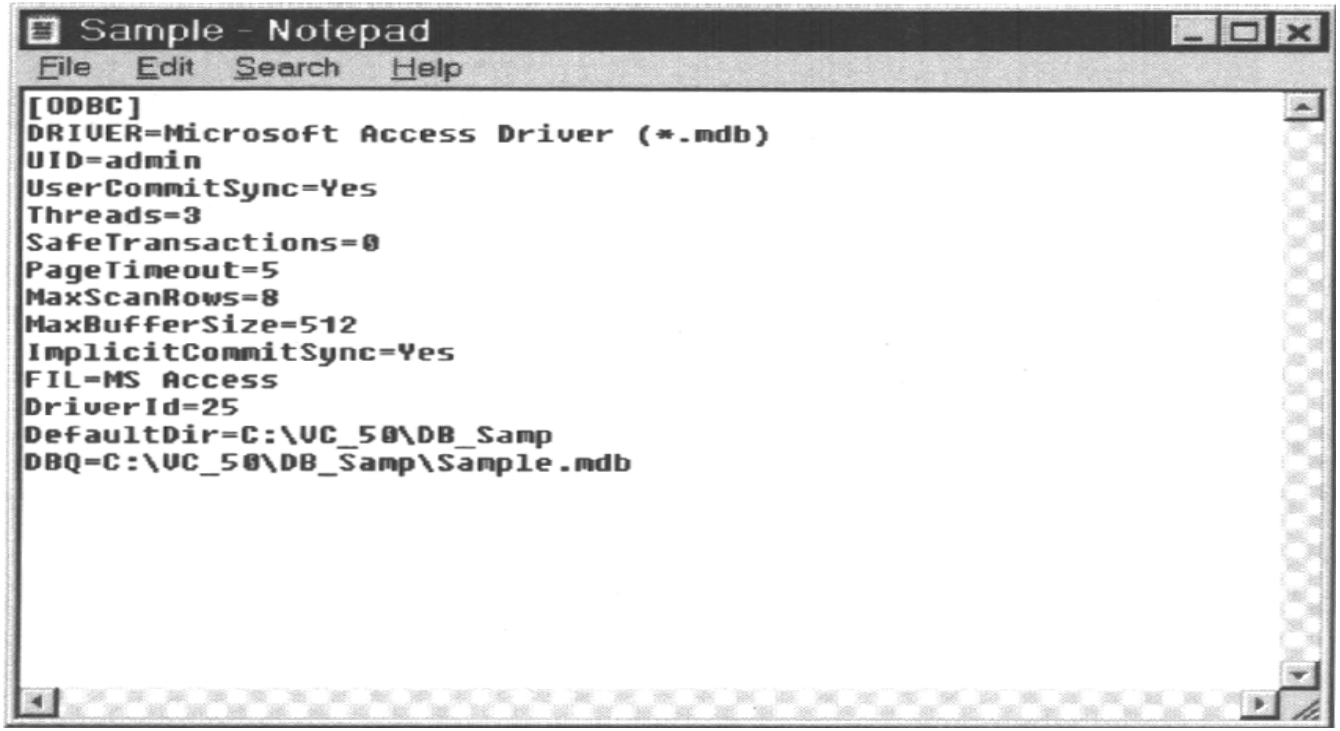


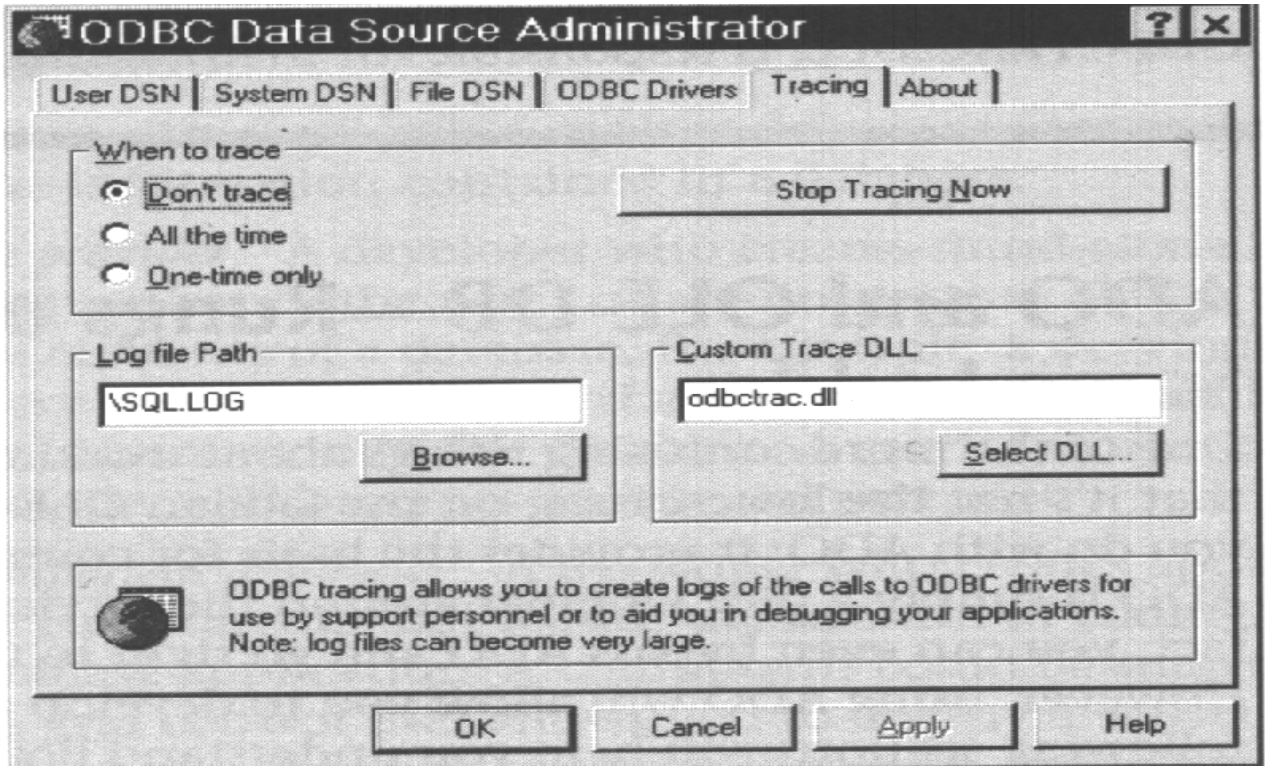
图 4.1　SAMPLE.DSN 文件包含了从 Visual C++内部使用我的数据库时所需的全部设置

记录 ODBC 事务

调试应用程序时加以记录，这一点很重要。ODBC Data Source Administrator（ODBC 数据源管理员）对话框也提供了这样的功能。可以选择跟踪你通过 ODBC 对数据库进行的各种事务。当然，这些记录可能会变得相当大，但我们并不是总要使用它们。

开始记录事务时，只要双击控制面板(Control Panel）中的 32 位 ODBC 小程

序，打开 ODBC Data Source Administrator（ODBC 数据源管理员）对话框。选择 Tracing（跟踪）选项卡，你会看到如下图所示的对话框（请注意，该对话框的 Windows 98 版本略有不同）。



如你所见，有三个单选按钮确定何时跟踪 ODBC 调用。缺省设置是 Don't Trace（不跟踪）。如果打算调试单个应用程序，应该选择 All the Time（全部时间）。One-Time Only（只一次）在下一次连接期间跟踪 ODBC 调用——只要连接一拆除，跟踪就关闭。当用户带着特定的问题调用时，这是一个不错的选择。可以监视一次会话期间的连接，然后用该信息帮助创建排除错误的计划。

跟踪不会自动开始。还要单击对话框右侧的 Start Tracing Now（现在开始跟踪）。只要跟踪一开始，按钮标题就变成 Stop Tracing Now（现在停止跟踪）。再次单击按钮关闭跟踪过程。

唯一要担心的其它设置是 Log File Path（记录文件路径）。ODBC 通常在根目录下的 SQL.LOG 文件中放入事务信息。但是，也可以将该信息放在网络驱动器上，或者用户看不到的地方。在调试过程中，缺省位置通常就比较适宜。

注释 除非你想创建自己的日志记录 DLL，否则不要更改 Custom Trace DLL（自定义跟踪 DLL）域内的设置。在这里列出的 DLL——ODBCTRAC.DLL——负责维持事务记录。

4.2 ADO 和 OLE-DB——同一问题不同层次的解决方案

使用 ADO 时一件让人困惑的事是，需要理解它不是数据库访问的底层解决方案。OLE-DB 是使用 ADO 的基础；它也提供与数据库通信的基础。ADO 只是 OLE-DB 提供的服务的漂亮包装。事实上，如果你愿意，甚至可以绕过 ADO 而直接使用 OLE-DB。但是，使用 ADO 开发应用程序的速度要快得多。下面的章节会帮助你理解 OLE-DB 和 ADO。

理解 OLE-DB

那么，OLE-DB 是什么呢？顾名思义，它用 OLE（或者更具体一点，是部

件对象模型——COM）为数据访问提供一套接口。就像任何其它的 COM 对象一样，可以查询、创建和撤销一个 OLE-DB 对象。OLE-DB 对象的来源被称为提供者。你会得到各种 OLE-DB 提供者，它们是 Visual C++软件包的一部分，也有可能在厂商升级其数据库产品时提供。OLE-DB 的好处是，同一个提供者可以和任何 Visual Studio 产品协作：Visual C++，Visual Basic，Visual InterDev 和 Visual J++。

OLE-DB 还依赖于事件，这和任何 COM 对象一样。这些事件说明什么时候要求更新数据库表，以显示其它用户生成的新项，或者你所请求的表什么时候准备好供查看。还会看到这样的事件，它们用来发送信号通知各种数据库错误和其它要求马上轮询的活动。

Microsoft 定义了 OLE-DB 用户的四个主要类型。下面将各组予以分解，从总体上说明它们是如何帮助使用 OLE-DB 的。

- **数据提供者**　用 OLE-DB SDK（软件开发工具箱）创建 OLE-DB 提供者的人。提供者用户与数据库和事件交互作用，发信号通知特殊事情的发生。
- **数据消费者**　要求访问包含在数据库中的信息的应用程序、系统驱动程序或用户。
- **数据服务提供者**　创建用以增强用户或管理员使用或管理数据库内容能力的独立实用程序（服务）的开发者。例如，开发者可以创建查询引擎，允许用户用自然语言请求数据库中的信息。服务和 OLE-DB 提供者互相协作，成为一个整体。

- 商务部件开发者　创建应用程序模块或部件，以减少创建数据库应用程序所需的编码量的开发者。部件可能和网格控件一样通用，可以快速地显示数据库中记录的子集，或者显示正被访问的数据库类型的专用信息。

那么，OLE-DB 和 ODBC 有什么不同呢？表 4.1 说明了这两个产品间的主要差别。我们会在本章的“应该在什么时候使用 OLE-DB、ADO 或 ODBC？”一节中讨论这些差别是如何影响你的使用决定的。

不要认为 OLE-DB 和 ODBC 是两种完全分离、可以互相替换的技术。ODBC OLE-DB 提供者允许你通过 OLE-DB 或 ADO 访问 ODBC 提供的所有功能。换言之，这两种技术是互补的，不能完全互相替换。可以用 ADO 或 OLE-DB 替换 ODBC 吗？可以，但如果这样做了，将得不到最佳的应用程序性能。OLE-DB 的整体作用是，拓宽可以用 Visual C++应用程序访问的数据库类型的范围。很明显，如果确实要用单个应用程序访问 ODBC 和列表数据，那么 OLE-DB 为此提供了较好的解决办法。

理解 ADO

现在你对 OLE-DB 比较熟悉了，那么 ADO 适合在哪里使用呢？如前所述，ADO 为访问 OLE-DB 提供者的功能提供了一种容易的方法。换言之，ADO 允许你快速地创建应用程序，允许 Visual C++处理你在直接使用 OLE-DB 时通常应考虑的一些细节。我已经在第一章关于 ADO 数据连接的章节中提供了关于 ADO 特征的概述，这里就不再讨论了。

表 4.1 OLE-DB 与 ODBC 的技术比较

元素	OLE-DE	ODBC	描述
Access 类型	部件	直接	OLE-DB 提供与数据交互作用的接口，用户访问数据要通过被设计成与 OLE-DB 交互作用的部件来完成
数据访问特殊化	任何列表数据	SQL	ODBC 总被设计成把 SQL 用作数据事务的基础。在一些情况下，这意味着程序员必须做出让步，强制数据符合 SQL 标准
驱动程序访问方法	部件	本机	如前所述，所有对 OLE-DB 提供者的访问都要通过使用各种类型部件的 COM 接口。ODBC 通常要求某类直接程序设计，且特别依赖数据库厂商所提供的 SQL 兼容性等级
程序设计模型	COM	C	OLE-DB 依赖于 COM 向程序员提供的对提供者的访问。这意味着 OLE-DB 是语言无关的，而 ODBC 则使用专用语言
技术标准	COM	SQL	OLE-DB 坚持 Microsoft 的 COM 标准，这意味着厂商和专用平台要比 ODBC 所使用的 SQL 技术标准多得多

与 OLE-DB 一样，ADO 也是基于 COM 的。它提供了一个双重接口：对本地操作是 ADODB 的程序 ID，对远程操作是 ADOR 的程序 ID。虽然注册表像

使用分室线程模型那样显示 ADO 线程，但 ADO 库本身是种自由线程。ADO 的线程安全依赖于你所使用的 OLE-DB 提供者。换言之，如果使用 Microsoft 的 ODBC OLE-DB 提供者，不会有任何问题。如果使用第三方的 OLE-DB 提供者，要在假定 ADO 线程安全之前检查厂商的文档（这是在 Internet 或企业网连接上使用 ADO 时的要求）。

使用 ADO 时会用到七个不同的对象。表 4.2 列出了这些对象，并描述了它们的使用方法。虽然 ADO 对象的功能层次要比先前技术提供的层次高得多，但其中多数对象类型是 Microsoft 已引进的其它技术的复制品。我们会在本章的概述一节更为详细地讨论 ADO 的类。

注释 一些 ADO 对象由接口而不是由实际类来表示。表 4.2 还说明了对象的关联关系，它帮助你理解如何导出并非由 Visual C++类直接表示的对象。

表 4.2 ADO 对象概述

对象	类	描述
Command	CADOCommand	命令对象使用连接或记录集对象执行任务。即使可以把执行命令作为连接或记录集对象的一部分，命令对象也要灵活得多，而且允许定义输出参数
Connection	CADOConnection	定义与 OLE-DB 提供者的连接。可以用这个对象来执行任务，如开始提交和回滚事务。它还提供了打开或关闭连接以及执行命令的方法

续表

Error		错误对象作为连接对象的一部分而创建。它提供关于 OLE-DB 提供者引起的错误的额外信息。单个错误对象可以包含多个错误的信息。每个错误对象都与特定的事件相关，如提交事务
Field		字段对象包含记录集对象内的单列数据。换言之，字段可以被视为表中的一列，它包含了某类与记录集相关的所有记录的数据
Parameter	CADOParameter	为命令对象定义单个参数。参数修改存储过程或查询的结果。参数对象提供输入、输出，或同时提供输入输出
Property		某些 OLE-DB 提供者需要扩充标准的 ADO 对象。属性对象代表了完成这一任务的一条途径。属性对象包含属性、名称、类型以及值信息
Recordset	CADORecordset	它包含了返回表中对所选独立元素查询和光标的返回结果。Visual C++让你选择用单个记录集对象创建连接和记录集，或者用现有的连接对象支持多个记录集对象

4.3 什么时候应该使用 OLE-DB，ADO 或 ODBC

要在访问数据时判断出应该使用哪一种技术，这并不容易。可能需要公用

实用程序来处理多个数据库类型；部分数据可能出现在本地硬盘驱动器上，部分在网络上，还有一部分在主机上。甚至客户安装在设备上的产品也会使这种选择更加困难。例如，你所期待的 ODBC 支持级别也许依赖于所安装的 Microsoft Office 的版本，因为这个产品不提供 ODBC 支持。你还会发现，ADO 类提供的对象和方法要比 ODBC 类多。ADO 可以提供程序中绝对必须具有的一些特性——例如，你会发现 OLE-DB 和 ADO 两者都支持 DFX_Currency，但在 ODBC 中没有对应的功能——但你要想掌握它们也必须付出一定的努力。

注 使用 ADO 而不是 ODBC 的一个原因是，ADO 提供的对象和方法比 ODBC 多。

选择 OLE-DB 或 ODBC 时，有几条一般的规则。因为 ADO 实际上只是 OLE-DB 的包装，所以这些规则也适用于它。下面提供一些基本的原则，可以用来帮助你决定选择 OLE-DB 还是 ODBC。

- **非 OLE 环境** 如果要访问支持 ODBC 的数据库，而该数据库又在不支持 OLE 的服务器上，那么 ODBC 是最好的选择。
- **非 SQL 环境** ODBC 在处理 SQL 时非常出众。处理非 SQL 数据库时，OLE-DB 则具有非常明显的优势。
- **OLE 环境** 对支持 OLE 的服务器来说，选择 OLE-DB 还是 ODBC 也许是希望各半。如果有 ODBC 驱动程序可供利用，那么使用 ODBC 是一个好主意；否则，就只有选择 OLE-DB 了。
- **所需的互操作性** 如果需要可互操作的数据库部件，那么只有选择 OLE-DB。

其它的问题往往是复合问题，或者至少是失去了可以用来区别这两种技术

的方便的规则。例如，有一些特征是 ADO 和 ODBC 所共有的。其中一个特征是，Visual C++允许你直接访问任一种技术。这意味着，可以完全访问 ADO 和 ODBC 都提供的每一个特征（是的，这确实不错，但也有缺陷，因为谈到访问，不能说一种技术明显地优于另一种技术）。

如果做了错误的决定，一些熟悉的技术实际上会帮助你把应用程序从 ODBC 转移到 ADO，或反方向转移。两种技术都依靠数据库对象来管理基础的 DBMS，而记录集对象则包含针对 DBMS 的查询结果。另外，ODBC 和 ADO 都以相似的成员来使用数据库和记录集对象。即使要对类和成员名称做一些更改，你仍会发现，ODBC 和 ADO 程序设计的代码非常相似。

ODBC 和 ADO 都依靠外部驱动程序来完成任务。但是，用 ADO 还可以从 Microsoft Jet 引擎获得额外的帮助。这个额外的支持是如何使程序设计更加容易的呢？考虑一下，要在哪里执行两个依赖不同引擎的数据库的外部连接。在 ODBC 下，必须人工执行全部所需的设置。使用 ADO 意味着，Microsoft Jet 引擎会为你执行这些设置。可惜的是，让 ADO 做这个工作意味着会失去控制级别和一些灵活性，而这正是你从一开始就想用 C++获取数据的原因。

注 ODBC 和 ADO 具有相似的特征，如记录集和数据库对象，尽管 MFC 利用这些对象的方式完全不同。

有一个地方绝对不能使用 ADO。如果要访问 16 位数据，就不能用 ADO。不管你怎么想，都必须用 ODBC。然而，几乎没有人还在使用 16 位数据库了。大多数的新工程都会使用 32 位接口，这意味着会有一个选择。老的工程已经有了嵌入代码的数据访问方法，所以你也实在没有必要在这里做决定。

ODBC 的一点不足是，不能像以前使用 ADO 那样跟踪事务。以 Microsoft Jet 引擎使用 ADO 时，应该得到事务的工作区级的支持。ODBC 只提供数据库级的事务支持，这就是说，可以从几个不同的工作区跟踪事务（这使得调试非常困难，还可能导致其它类型的问题）。

注释 处理 ODBC 数据库时，ADO 提供数据库级的支持；这和 ODBC 提供的支持层次是相同的。除非你使用 Microsoft Jet 引擎，否则得不到工作区级事务支持的好处。

4.4 Visual C++ ODBC，OLE-DB 及 ADO 类概述

用 C++创建数据库应用程序的讨论终于涉及到了类，为了进行实际的工作，你需要了解类。每个类都执行非常特殊的任务，所以，知道在哪里使用哪一个类至关重要。所有类的总目标都是从服务器获得特定的数据，并在屏幕或打印机上显示出来。

注 这确实可以帮助你把数据库对象看作是将数据从数据源装入你的计算机的一种管道。

我们要了解的第一个类是 Cdatabase(ODBC)或 CADOConnection(ADO)。用这些类创建的对象让你得到对数据库内数据的访问。既可以创建指向特定记录的指针，也可以下载整个查询；数据库对象创建你需要的连接。获得的数据访问的类型依赖于是创建了窗体视图（所有数据的单记录显示）还是记录视图（与

特定标准相匹配的所有记录的网格显示）。

一旦连接了数据库，就要有某种容器来保存它所包含的内容。CRecordset(ODBC)和 CADORecordset(ADO)类在这里发挥作用。记录集保存最终要在屏幕上显示的数据。这有助于把记录集看作是保存数据库中数据的容器。如下所述，有三种类型的记录集：

- 表　表型记录集代表了数据库某个表内的数据。你可以对这样的单个表进行任何处理，包括添加、删除或编辑记录。
- 动态集　需要用查询从数据库的一个或多个表中提取信息时，会用到动态集型记录集。与表型记录集一样，可以添加、删除或编辑动态集型记录集中的记录。
- 快照　这是数据库一个或多个表中所包含的数据的静态拷贝。与动态集一样，要通过查询来提取信息。与动态集不一样的是，你不能修改快照中记录的内容。但是，可以用快照型记录集查找数据或产生报告。

你可能已经注意到了，我在谈论记录集时提到了术语“查询”。查询就是一个简单的提问。查询时所做的一切工作就是要求数据库提供一批符合特定标准的记录。使用 ODBC 时，要依靠 CRecordset 类数据成员来更改想要看的记录的查询。ADO 有一点不同。可以用 CADOCommand 类创建特殊的查询，或用 CADOConnection 执行标准的查询。CADOCommand 要比 CADOConnection 灵活得多。可以用它来执行多种和数据库有关的任务。例如，可以用它操纵数据库的结构或者执行其它的管理任务。

本章中我们要考察的最后一组类是实际浏览数据时使用的类。

CRecordView(ODBC)和 CADORecordView(ADO)类允许你实际查看检索到的数据。本质上说，这些类要完成的任务就是将数据从记录集对象移动到对话框或窗口上的控件中。你还可以使用这些类来检测何时到达记录集的开始或结尾。

第 5 章　数据库创建概述

绝大多数的应用程序由于某种实在的需求而开始设计。假定你的硬盘需要某种特殊的维护。下一步就是描述你所需要的功能。例如，一个实用程序可能需要访问磁盘资源，确定是否有问题，并最终解决所发现的问题。一考虑到某种功能，就可以开始查看用户接口，最后创建使设备正常运转所需的代码。

所有这些步骤，数据库应用程序都要求。你依然要有已知的应用程序需求，然后决定那个应用程序要做什么。不过，数据库应用程序还要增加一步。数据库应用程序的主要功能是访问数据，并有条件地允许用户操纵它（当然，这是一个非常简化了的说法，实际过程要复杂得多）。在真正进行数据库应用程序设计之前——明确地说，在创建任何用户界面元素之前——必须首先设计数据库本身。

决定如何设计你的数据库的部分过程是规划出你计划如何访问数据库以及计划在数据库中放入多少数据。例如，如果唯一的目标只是存储一些个人姓名和地址，那么就不会需要基于服务器的 DBMS（数据库管理系统）。你还要清晰地知道所用的程序设计语言是否能够操作所使用的 DBMS。在第 4 章中，我们研究了三种从 Visual C++访问数据库的方法：ODBC，OLE-DB 和 ADO。我甚至说明了如何设置 ODBC 的数据源名称(DSN)。最后，我们讨论了不用获得

额外的驱动程序即能访问的 DBMS 的类型，以及在需要时获取驱动程序的途径。不过，知道如何访问数据库并不是说你已经知道数据库都包含些什么。

注 在可以设计数据库之前，要知道如何访问它以及数据库的内部结构的工作方式。

本章中，我们将学习数据库的内部知识——并探查与 DBMS 相关的数据。我们将简单地介绍一下数据库的基础知识。例如，学完本章后，你会知道什么是表以及如何定义表。我们还要学习查询和索引，还要考察几个界面元素，比如窗体等，但这只是最基本的知识。最重要的部分是，你会理解数据库是如何运行的，这是用 C++访问它们的先决条件。

由于篇幅所限，还有一些问题本章中没有讨论。我不想花很多时间去探讨关系数据库理论——市面上这样的书很全。本章结束时，你可能不会全面地掌握 SQL（结构化查询语言）或完全理解用户界面设计。总而言之，通过本章，你会掌握创建后续章节中用到的简单数据库所需的一切知识——应该把本章看作是数据库的入门资料，仅此而已。

Web 链接 如果需要更多的关于特定数据库问题的信息，而不是仅限于大多数书籍所提供的或某个人所谈到的信息，可以试一试新闻组。例如，你会在 comp.theory.info-retrieval newsgroup 中找到很多理论信息。一系列完整的 comp.database.<product name>新闻组，比如 comp.database.ms-access，也提供了许多帮助。你会看到，这些新闻组可以帮助你发现隐藏在数据库设计之后的理论问题。它们还可以指导你克服设计自己的数据库时所面临的障碍。

5.1 构造块概述

每个数据库都是由模块化的片组成的。有存储数据的片、组织数据的片，还有的片允许你查找数据库中特定片的信息。数据库还包含接口元素和允许你操纵所包含数据的代码片。然而，在设计 C++应用程序时，你多半会忽略后两种模块化的片。事实上，我们在本章中不讨论编码，因为编写 C++应用程序不需要它（我们只简单地讨论窗体，因为它们在你从 Visual C++访问数据库结构之前，提供了测试 DBMS 中你的数据库结构的便利方式）。

注释 我们要为本章中所有的示例使用 Microsoft Access。并不是因为 Access 更好一些，而是因为它提供了大量的工具，可以使数据库设计过程更加容易。Access 还是 Microsoft Office 专业版软件包的一部分，所以在阅读本书时，很可能你们中许多人的手头都有 Access 的拷贝。请记住，你应该会有所收获，并把它们应用到其它的 DBMS 产品中去。

图 5.1 概述了 Visual C++、数据库管理器以及数据库是如何交互作用的。如果你现在实在不理解图中的一切是如何进行的，那也无须担心。我们将在随后的章节中讨论所有的元素。现在，这张图只是大体说明各种部件是如何连接到一起的。

可移植性 开始设计过程之前，一定要确保数据库和你选择的 DBMS 实际上可以和 Visual C++相协作。使用 ODBC 之外的 ADO，OLE-DB 和 DAO 的能力应该能够极大地增加找到正确解决办法的机会。另外，如果决定从多个平台访问数据库的话，需要保证有办法以想为其编写应用程序的所有平台上使用数据库，并保证 DBMS 支持文件和记录锁定的标准方法。简言之，在最后决定使用哪一个 DBMS 之前，必须检查每一个可计算性问题，尤其是计划在多种机器上使用 DBMS 的情况下更应如此。

数据库

最好把数据库看成是一个容器（见图 5.1）。数据库用来存储要用于存储和操纵数据的全部元素。大多数的数据库是单个的文件，比如 Access 用的 MDB 文件。也可以使用有分离文件的产品，文件都存储在一个像目录的地方。在这种情况下，目录就像数据库——它是把所有其它元素保持在一起的容器。

注释：Ashton-Tate（dBASE 的发明者）首次在单个 Xbase 表格（DBF 文件）中使用了“数据库”这一术语。事实上，这个文件的扩展名 DBF 的意思就是“数据库文件”。这一销售策略带来的后果是，一些人仍然对什么是表以及什么是数据库感到迷惑。数据库是一个把所有其它数据元素保存在一起的容器，其中包括代码、窗体、表、查询、索引和报告。不管你现在在想什么，遍及全书的术语“数据库”总是指这样一个容器，它将一个数据集的所有数据元素都保存在一起。

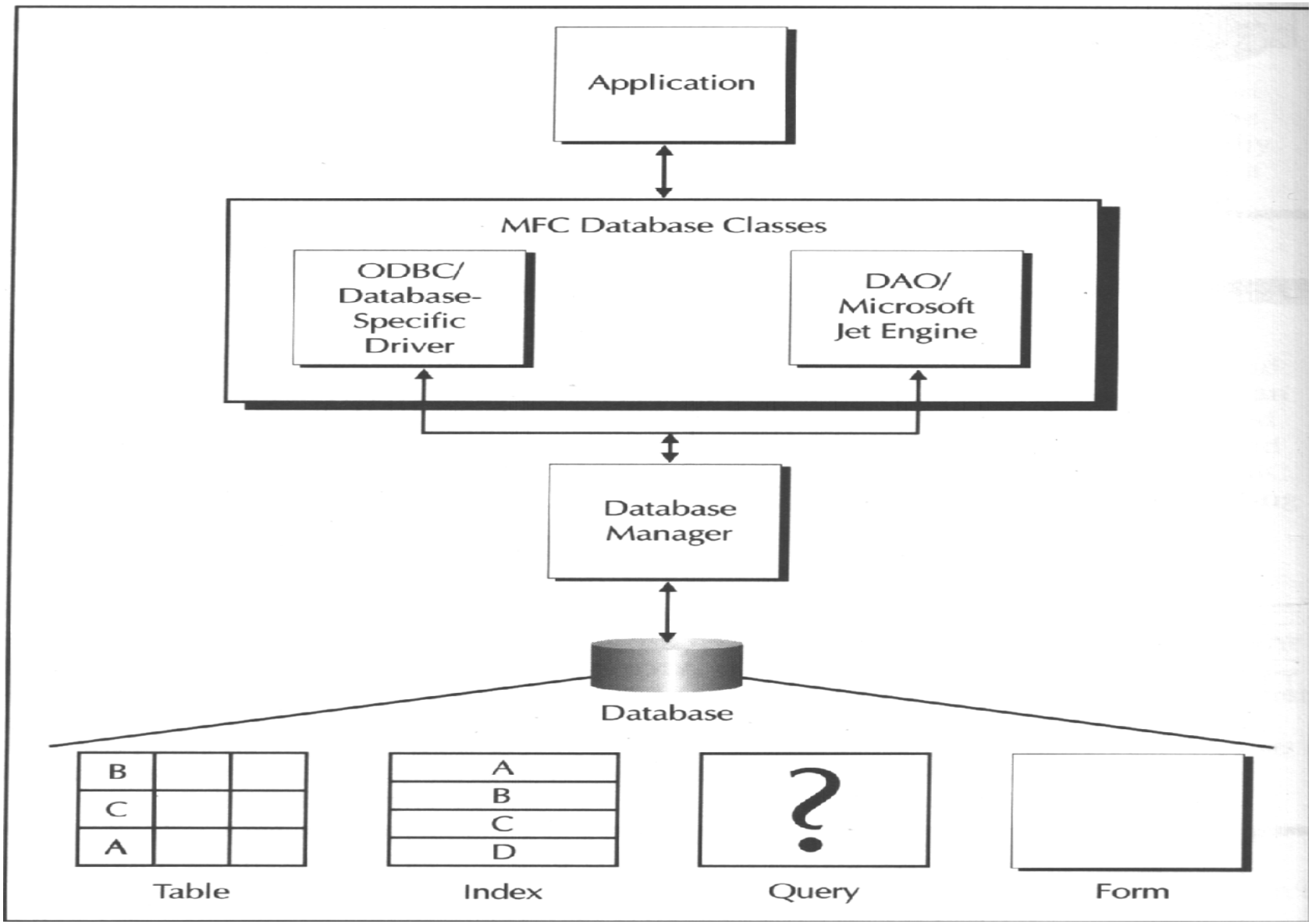


图 5.1　Visual C++、数据库管理器和数据库本身之间的关系

数据库并不只是端坐在那里，看上去很可爱。一个被称为数据库管理器的特殊应用程序可以在文件内部查看并提取所需的一切。如果你正在建立数据库，可能想看看已经定义的查询或者窗体。另一方面，作为一个用户，你唯一的兴趣可能是包含在表内的一些难以理解的比特。不管你需要什么，数据库管理器正是这样的应用程序，它在数据库内查找并为你提取所需要的东西。本质上讲，数据库管理器像一个正在碗柜这个容器中找东西的人那样，寻找某个玻璃器具或者什么吃的东西。

注 和任何用户一样，Visual C++请求数据管理器从数据库获取它所需要的东西。

把数据库管理器和数据库放在一起，就形成了数据库管理系统(DBMS)。在Visual C++的情况下，用来处理数据的数据库管理器和你编写的程序共用数据管理器的规则。换言之，用户要提出请求，必须和应用程序交互作用。反过来，应用程序请求数据管理器提供用户所请求的数据。数据管理器会返回数据，应用程序将其显示出来。这听起来好像有很多工作要做，实际上并非如此。对C++程序员来说，大部分工作是自动完成的（至少用MFC AppWizard创建应用程序时是这样）。

表

表是数据库内唯一可以保存数据的结构。其它每种结构都被设计成以某种方式处理数据：窗体用于显示数据、索引用于排序数据、查询寻找特定的数据片。实际存储数据的唯一结构就是表。

那么，表是如何拼装起来的呢？图 5.1 已很好地说明了这一点。请回想一下你用笔和纸创建的其它表（或者是你在类似本书的书籍中看见过的表）。表通常包含行和列。同样，数据库中的表也包含行和列。

行就是记录。每个记录代表了要收集的所有数据的一次出现。例如，如果你在创建姓名和地址表，那么行（记录）应该包含一个人的所有信息（至少应该包含设计表时要求保存的信息——我们会在后面看到，表是以节省数据库服务器空间的方式组合起来的，你仍然可以看到正在过目的任何数据的完整图片）。

注　行（记录）总是包含要存储在数据库中的数据实例。

每一个表列（或称做字段）都包含一类数据。例如，可能有名为 Last_Name 的表列（字段）。这个字段应包含你的姓名与地址数据库中所有人的姓名。字段只保存一类数据。你不能用单个字段在一个记录中保存地址信息，而在另一个记录中保存年龄信息。如果既要保持地址信息又要保持年龄信息，就必须构造两个字段。

注　列（字段）包含一类数据，如姓氏或者地址。

从表中获取特定的一段数据或者数据元素很容易。你所需要完成的任务就是请求特定的行（记录）和列（字段）。从数据库中获取数据的过程——不管它是包括一个数据元素还是包括表提供的所有记录——都称为查询。本质上讲，在你向数据库管理器提出一个问题后，它就提供与该问题相匹配的数据予以响应。Visual C++在提出问题时，很大程度上要依赖一种特殊的语言，这就是 SQL（结构化查询语言）。但是你会发现，在使用 ADO，OLE-DB 或 DAO 时，也

可以利用其它的机制。随着课程的进展，我们会越来越熟悉 SQL，但是如果想设计复杂工程的话，就要花一些额外的时间来学习这门语言。

注 创建查询就像是询问数据库关于表内容的问题。

索引

如果再看一下图 5.1，你就会注意到，表第一列中的项并没有按字母顺序排序。没错，数据库不存储你以特别的顺序发送给表的项——事实上，如果数据库不排序的话，其效率可能会更高。如果每次某个人添加或删除记录时，数据库管理器都要重新排序表中所有的记录，你可能就做不了太多的工作了。不用说，你并非总是需要数据的顺序相同，所以两个人没有必要去争论数据的顺序是什么。一个人可能想要地址表以姓氏来排序，另一个人可能又需要表按 ZIP 代码排序。

注 数据库把数据放在表的末尾，不能保证会以什么样的顺序来查找未编索引的表中的数据。

那么，如何以某种顺序获得所需的信息呢？答案是索引。图 5.1 说明了每个记录都要考虑的两个原则。第一，索引不能包含整个表的所有项——它只包含一个子集。在这里，它只包含第一个字段（列）内的项。第二，索引是有序的。索引包含了指向表中的记录的指针，不过图 5.1 没有显示出来。换言之，在选择记录时所看到的数据，和数据库管理器用来查找整个记录的指针是成对的。排序索引允许你快速地查找想要的那类数据。指针允许你查找表中的整个记录。使用索引还允许两个人以两种不同的顺序访问一个表中的数据。因为索引有序

而表无序，所以可以为用户所需的每一种数据排序方式定义一个索引。

没必要把索引限制在单个表内使用。很多数据库管理器允许你创建跨越多个表的索引，只要那些表是相关的即可。例如，为了掌握客户的采购情况，可以创建两个表。第一个表包含客户列表，而第二个表包含每次采购的细节。索引可以用消费者 ID 字段加采购编号来排序两个表，首先是以客户为序，然后是以采购编号为序。

注 只要表以某种方式相关，索引就可以跨越多个表。

我们来看一看这个多表索引概念实际上是如何工作的。图 5.2 说明了多表索引的一个示例。请注意，Clients（客户）和 Purchases（采购）表间是有关系的。两表中编号为 0001 的 Client ID 识别名叫 Ann 的客户。但是，Clients（客户）表只有 Client ID 的一个拷贝，因为一次只有一个客户可以使用 Client ID。因为客户可以多次采购，所以 Purchases（采购）表包含每个 Client ID 的多个拷贝。请注意，在 Purchases（采购）表中记载 Ann 采购了 $20。这次采购由编号为 97001 的 Purchase ID 识别。此外，每一个 Purchase ID 都是唯一的；否则，无法将一次采购与另一次采购区分开来。现在看一下索引。它包含两个值，Client ID 和 Purchase ID。它先由 Client ID 排序，然后由 Purchase ID 排序。虽然表是无序的，但是该索引却有序。索引中的指针允许你在两个表中查找信息。

不要认为索引也被限制在表内的直接项上。很多数据库程序员都发明了组合表项的方法，甚至可以用特殊等式产生非表的索引数据。很明显，数据库管理器制定的规则限制了把什么放入索引的选择。换言之，如果数据库管理器不允许，就不能把某种数字计算添加到索引中去。同时，如果以最初预想的顺序

在屏幕上显示数据时有麻烦，可以先做一下试验。

注 一些 DBMS 不限制你在索引中使用哪些字段名；你也可以使用复杂的表达式。

可惜的是，Access 没有提供在数据库设计视图中创建复杂索引的简明方式。可以创建包含多个字段的索引，可以分别控制那些字段排序的顺序，但是不能直接使用等式或者其它复杂的指令。在一些情况下，可以用 SQL 语句创建复杂的索引，但其本身就很复杂。幸运地是，你可以把复杂的索引作为查询的一部分来提供，我们会在本章的后面讨论这一点。

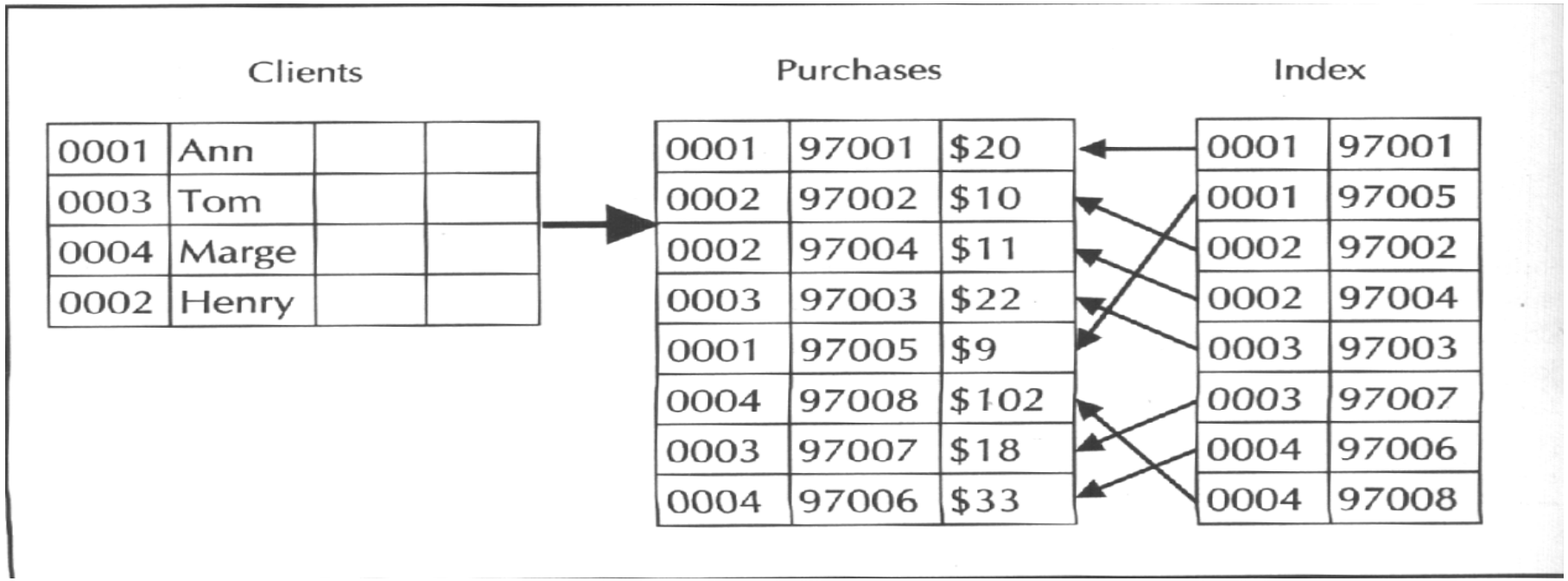


图 5.2 索引可以跨越多个表

技巧 DBMS 不允许直接向索引中添加等式或者计算值，但是，你可以通过向表中添加特殊的计算字段来实现这一点。该字段可以包含计算值，而索引又包含这个特殊的字段。例如，可以用 Access 完成这个工作。它允许用 Expression Builder（表达式生成器）给一个字段赋缺省值。如果用户从来没有见过这个字段，那么结果就是你可以为了编索引而使用的特殊字段。如你所见，结果是一样的，只不过完成的方法稍稍复杂了一点。

查询

再看一下图 5.1。我用来定义查询项的图标表示这是一个大概的看法——查询是向数据库提出的有关它所包含数据的提问。可能会有人向你展示难以理解的复杂的 SQL 语句，试图让你相信，查询并不是一个简单的过程。一个简单的事实是，查询就是一个提问。

注 始终要记住，查询就是向数据库提出的有关它所包含数据的提问。

每个查询都有特定的部件，但这个信息只是帮助你以 DBMS 可以理解的方式来简述这个提问（请记住，DBMS 实际上对操纵数据库的内容负责）。记住这一点，我们来看一看比较简单的 SQL 语句，它允许你查看两个相关表中的所有记录。

```
SELECT Foods.*, Orders.*
    FROM Foods INNER JOIN Orders ON Foods.Food_ID = Orders.Food_ID
```

```
WHERE ((("Food_ID")<>"VEG-00001-SPL"))
ORDER BY Foods.Food_ID;
```

首先我要指出，在使用 Access 时，没有必要使用像这样的 SQL 语句，除非你实在想用。Access 会用 Select Query（选择查询）对话框帮助你创建 SQL 语句。现在让我们看一看这条查询语句。

我们要做的第一件事是，选择两个表以及这两个表内的所有字段。SELECT 语句要 DBMS 选择一些数据。Foods（食品）和 Orders（订货）是两张表。出现在表名后的星号(*)要 DBMS 选择两张表中所有的字段。如果想选择特定的字段，应该以<table name>.<field name>格式输入字段名。

SQL 语句的下一行看起来确实很复杂，但它所要做的，只是要 DBMS 建立两张表的关系。FROM 关键字告诉 DBMS 主表是 Foods（食品）。INNER JOIN 说明如何创建与从表 Orders（订货）的关系。ON 关键字说明我们准备建立关系的字段。在这里，两张表都有一个同名的字段，我们就用它来创建关系。创建关系的标准是，有两个包含相同类型信息的字段，就像我在图 5.2 中说明的那样。在这种情况下，两个相关的字段都是 ClientID。对主表中的每一个项来说，从表通常都包含多个项，虽然这并不是绝对要求的（对主表中的每一个记录，从表不是非得有一个项与之对应）。

可以用 SQL 来限制实际获取的记录数。在这里，我们告诉 DBMS，不想看到 Food_ID 字段等于 VEG-00001-SPL 的任何记录。很明显，可以用任何标准来限制取回的记录数，选择标准可能变得非常复杂。

最后，ORDER BY 关键字告诉 DBMS 你想如何排序记录。在这里，我们想

基于 Food_ID 字段的内容以升序次序来排序记录。因为 SQL 允许完全利用表达式，所以你可以使 ORDER BY 子句要多复杂、就多复杂。可以用等式或其它任何所需的标准，以正好需要的顺序获取想要的数据。

Web 链接 如果你确实想熟练掌握 SQL，请看 http://www.opengroup.org./ot/search.htm 处的 Open Group Web 站点（Open Group 是 OSF 和 X/Open 的组合）。输入 SQL，然后在询问想要寻找的内容时单击 Search（搜索）按钮。你会发现一系列的 SQL 规范。

与 Access 不同，Visual C++在大多数情况下会要求你手工创建这些 SQL 语句。时间不长你就会发现，在 Access（或你正在使用的任何 DBMS）中创建所需要的查询是提高开发速度的一种方式，尤其在你对编写 SQL 语句了解不多的时候更是如此。

注 窗体在屏幕上表示用查询从数据库收集到的数据。

窗体和报告

到现在为止，我们谈论了很多管理和存储问题。不错，在可以使用数据库之前，它要求内部管理结构全都正确，但是，如果不能以某种方式使用收集到的信息，那么确实没有任何理由说工作已经完成了。窗体和报告是实际使用数据库信息的两种最普通的方法。

注 报告通过打印表示用查询从数据库收集到的数据。

通常，窗体表示在屏幕上看到的数据，而报告表示发送到打印机的相同数

据。窗体常常显示为已请求的所有记录的网格，或者一次只显示一条记录。报告总是显示已收集到的一系列完整的数据，但它通常不使用网格格式。典型情况下，报告提供已请求数据的某种结构化视图。

深入窗体

我们再来谈一谈窗体。如前所述，通常用两种方式来表现窗体中的数据：网格中的所有记录或者一次显示一条记录。图 5.3 显示了典型的网格视图，图 5.4 则显示了典型的单记录细节视图。网格法的主要优点是可以得到数据的整体印象。使用这种视图时，正确地排序数据一定会有好处——当记录数超出一屏信息时，可以容易地看到记录之间的关系。例如，在以网格显示之前，可以用产品类型来排序库存数据库，这样就可以看到库存中有多少相似项。

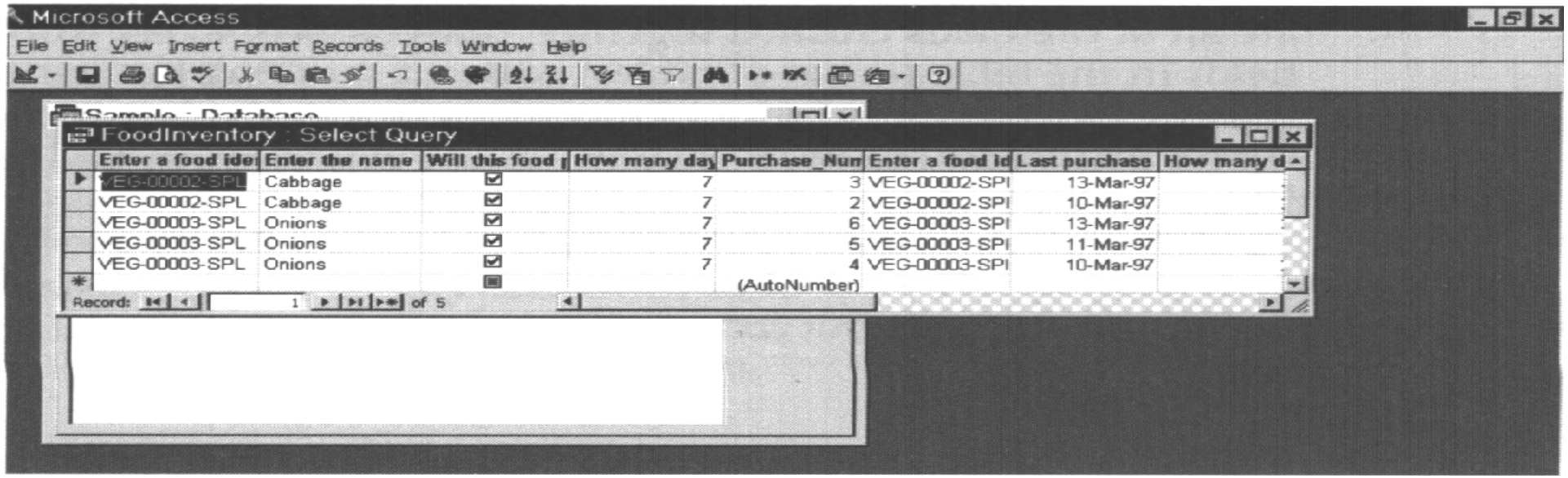


图 5.3　网格视图允许你一次看到多条记录

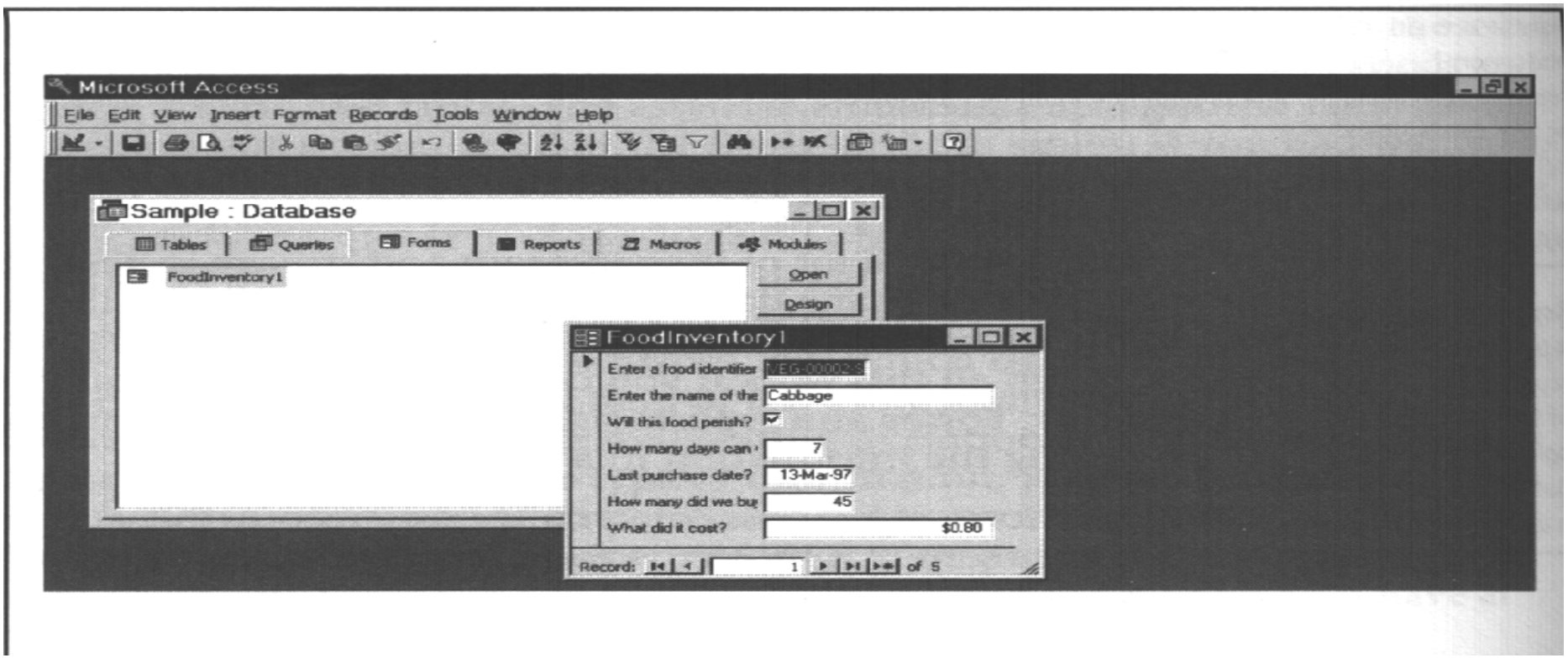


图 5.4　单记录细节视图允许你看到一个记录的所有信息

使用数据网格视图的缺点也十分明显。不但所有的字段都挤在一起，而且如果不水平滚动的话，很可能看不到表（或表集）中全部的字段。当然，可以很快地获得许多记录的概述，但是看不到个别记录的完整情况。一些数据库应用程序克服了这种限制，允许从网格视图的一组记录中选择单个记录，然后再转换到单记录细节视图方式。

技巧 一定要放置你认为用户在设计网格视图窗体时最需要的字段。这样，才能极大地减少用户查找重要数据时必须做的水平移动次数。一旦根据重要性对数据进行了排序后，就可以实现第二步，把相似的数据聚集到一起。例如，把所有的电话号码保存在一个地址数据库中是非常有用的。这样，试图查找特定人的住址时，只要逐个调用号码就可以了。可以通过事件监视用户的使用，或者用应用程序实际地观察它们，以此来测试网格的排列。用户查找数据时所必须做的移动越少越好。

单记录视图有精确表示一个记录的优点。你可以看到记录所提供的一切。当然，一看到图 5.4，这种表示的缺点立刻就明显了。不像网格视图，单记录视图不提供任何种类的概述。不能用它来分析趋势，或者得到整个表内容的更完整的情况。由于这个原因，单记录视图通常被保留作表数据的详细视图。一旦用网格视图找到了想要的东西，就可以用它来编辑或者验证要删除的记录。

注 用网格视图向用户提供可以用来选择记录的概述，然后用单记录视图让用户一眼就看到记录的全部细节。

第三种窗体视图你可能也会经常使用。它由添加了网格的单记录视图窗体组成。单个记录显示主表的内容。例如，它可能显示如姓名、地址和电话号码这样的客户信息。网格包含与主记录相匹配的从表内的记录。例如，网格可能包含一年中某客户进行的所有采购。

注 始终要避免数据过多而使得窗体杂乱无章。如果有必要的话，可使用多重窗体来避免屏幕杂乱。

可惜的是，要做出一个出色的组合型窗体是十分困难的。除非严格限制网

格显示的字段数，否则网格会占据很多的屏幕资源。结果，很多组合窗体看起来都很拥挤，而且就是因为有太多的东西要看，反而很难看得见数据了。问题是，你想使查找数据很容易，而这限制了用户在任一给定窗体上的位移量。

单记录视图的另一种窗体使用多重窗体来避免屏幕杂乱的问题。假定实际上需要 50 个字段来描述才能了解库存中的一个项目。视图在一个屏幕上显示那么多的字段，就会产生极大的杂乱，没有人可以找到所需要的东西。在这里，应该创建一个概述记录中的数据的主窗体，然后由辅窗体提供详细的资料。例如，可以在一个辅窗体上放入库存注释，因为注释往往要占据很多的空间。还可以在分离的窗体上放入特定项目的厂商数据，因为在很多情况下，一般来说你不会再排序新的项目了。当然，添加辅窗体就是向主窗体中添加细节按钮，这会进一步减少可用的屏幕空间量。要权衡屏幕杂乱和一个窗体上按钮过多的混乱因素。在一些情况下，最好的答案是，除非确实需要某条信息，否则不要显示它。

深入报告

最初，人们设想计算机可以产生无纸办公室。现在好像并没有实现这一点。用计算机产生的报告比任何用打字机的办公室产生的报告还要多。事实上，创建报告是很容易的，以至于还没有真正地去思考时，报告纸就已经开始产生了。

传统的报告赋予你这样的能力，将信息存储在数据库中，并以纸的形式输出。但是，报告不必要再出现在纸上了。例如，如果你有一个 Web 站点，需要使雇员及时了解公司当前的状况，就可以发表能在 Internet 浏览器中阅读的形式的报告。电子邮件也在通信中发挥了更大的作用；你可能会发现，报告必须是

电子格式，这样才能把它邮寄给别人。

不论报告采取哪一种形式，它都具有一些确定的特性，需要在创建过程中加以考虑，这些特性是：组织、目的和接受对象。下面各段就讨论要着重考虑的这些问题。

组织 在可以组织报告之前需要考虑，想在报告中看到哪种信息，该信息有多少是重复的。除去不需要的数据，为你想要查看的数据腾出空间——当使用已打印的媒体时，这种考虑非常重要，因为在使用数据的时候就不能再用上它了。腾出空间的另一种方式是，使所有重复数据都在报告的同一行上。这个过程称为分组。创建分组不但节省了空间，还可以帮助报告的阅读者更快地查找到所需的信息。

目的 报告的目的确定，应该包括多少信息、摘要是否会替代详细的资料清单。既要给读者提供所需的全部信息，又不能让无用的信息使报告显得杂乱，这一点很重要。例如，没有必要把客户数据库的每一个细节都告诉经理；他们真正感兴趣的是最后一行：每个客户采购了多少，或者是每次采购的金额有多大？

我们再来看一看我在图 5.2 中介绍过的 Clients 和 Purchases 表。如果你为此表创建了采购报告，那么肯定想包括 Purchases（采购）表中的全部摘要信息。但是，为了节省空间，最好把所有像 Client ID 和 Name 字段这样的客户信息用作标题，分组采购信息。在像这样的报告中，可能不需要所有的客户信息。例如，如果这个报告的目的是产生一个电话号码表，就可能无须添加地址信息。

如果想创建一个关于客户的一般报告该怎么办呢？在这种情况下，可能要

包括 Clients(客户)表中的所有客户信息,而排除 Purchases(采购)表中像 Purchase ID 这样的信息。另外,可能只要简单地合计每个客户的采购值,并在报告中列入单个数字,而不是创建每次采购的详细清单。

接受对象(观众) 观众会影响编制报告的类型。例如,你可能会发现,给同事看的报告可以采用非常简单易读的格式。你可能需要这样的格式,因为这样的报告包括了很多详细的信息。另一方面,从管理角度讲,又要使报告更精致一些才行。

技巧 不要低估你作为程序员对报告的需求。当然,可以试着用屏幕上的窗体排除所遇到的数据库设置问题。但是,报告可以帮助你查找隐藏更深的错误。例如,你可能会发现,数据库有微妙的数学错误,你只有在打印数据库内容时才会看到它。如果只是在屏幕上看数据库内容,这个错误可能不是很明显。一定要把你设计的报告实际打印出来,才能看到它们在显示数据库的内容方面同样非常有效。虽然你是在用 C++设计应用程序,但是了解 DBMS 本身能干些什么,会对在 Visual C++环境内设计更好的报告有所帮助。

5.2 创建数据库

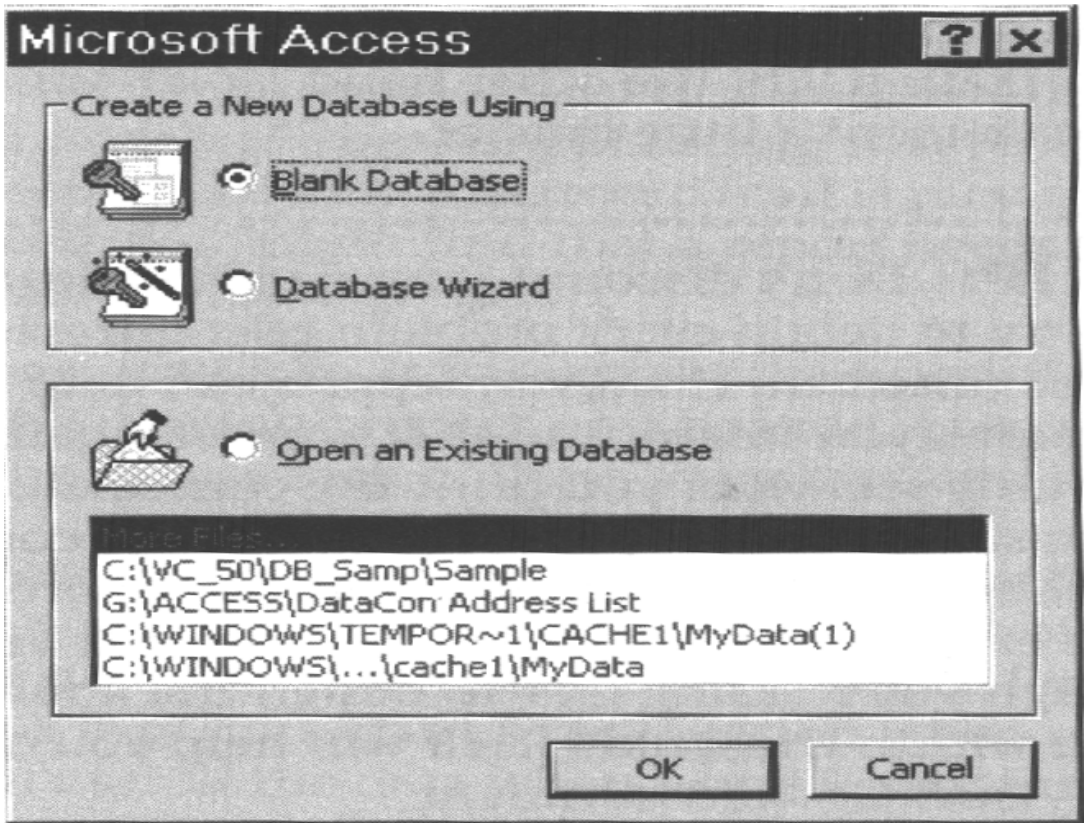
现在看一看数据库。这儿的数据库不会是你曾经见过的最复杂的数据库。实际上,开始时它只包括两个表。本节的整个目的就是让你从基础开始。虽然

我们要为这个示例使用 Access，你当然也可以把学到的基础知识应用到其它数据库产品中去。

注释 我们在全书中都要用到这个示例数据库,所以即使你不打算把 Access 用作你的 DBMS 理解它的创建过程也相当重要。后面数据库章节中的所有示例代码都是以这个示例表为基础的。你看到的演示原则上同样适用于其它 DBMS。请记住，Visual C++使用 ODBC 和 DAO 两种类，它们都不依赖于特定的数据库实现。

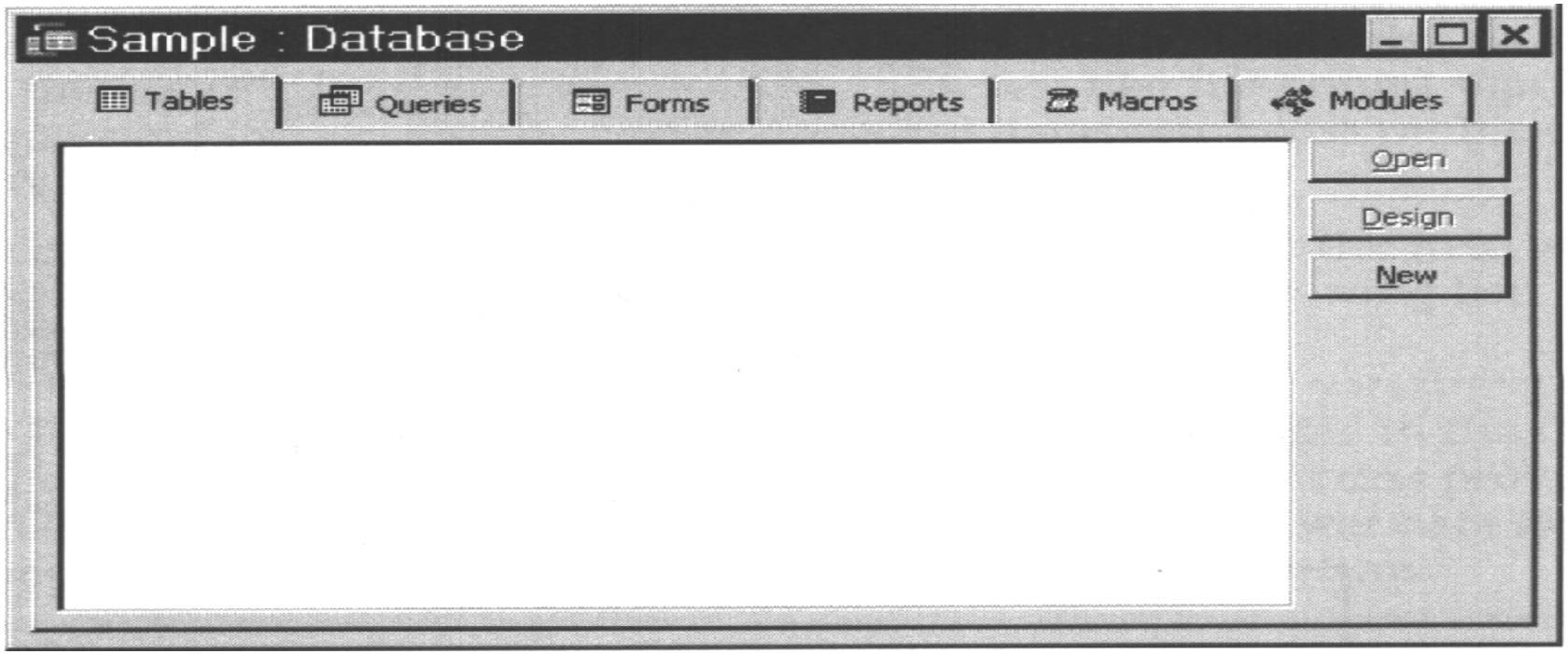
你需要完成的第一件工作是创建一个新的数据库。下面的过程帮助你从头开始。所有的示例程序都假定你将数据库命名为 Sample,并把它存储在 DB_Samp 目录中，当然任何名字和位置也都是可以的。

1. 启动 Microsoft Access。你会看到如下图所示的 Microsoft Access 对话框，它询问你想要做些什么。



2. 选择 Blank Database（空白数据库）选项，然后单击 OK。会看到 File New Database（文件新数据库）对话框，它看起来与 File Save（文件保存）对话框十分相似。

3. 输入数据库名（本示例使用 Sample），选择保存数据库的目录（本示例使用 DB_Samp），然后单击 Create（创建）。这时，会出现如下图所示的空白 Database 对话框。



现在有了可以任意使用的空白数据库，可以开始用表、查询和数据库具有的其它部件来填充这个数据库了。下面的章节会说明，如何创建本书中示例程序所需的元素。当然，你也可以根据需要添加任意多的数据库元素。为进行实验而建立一个示例数据库是个不错的想法。

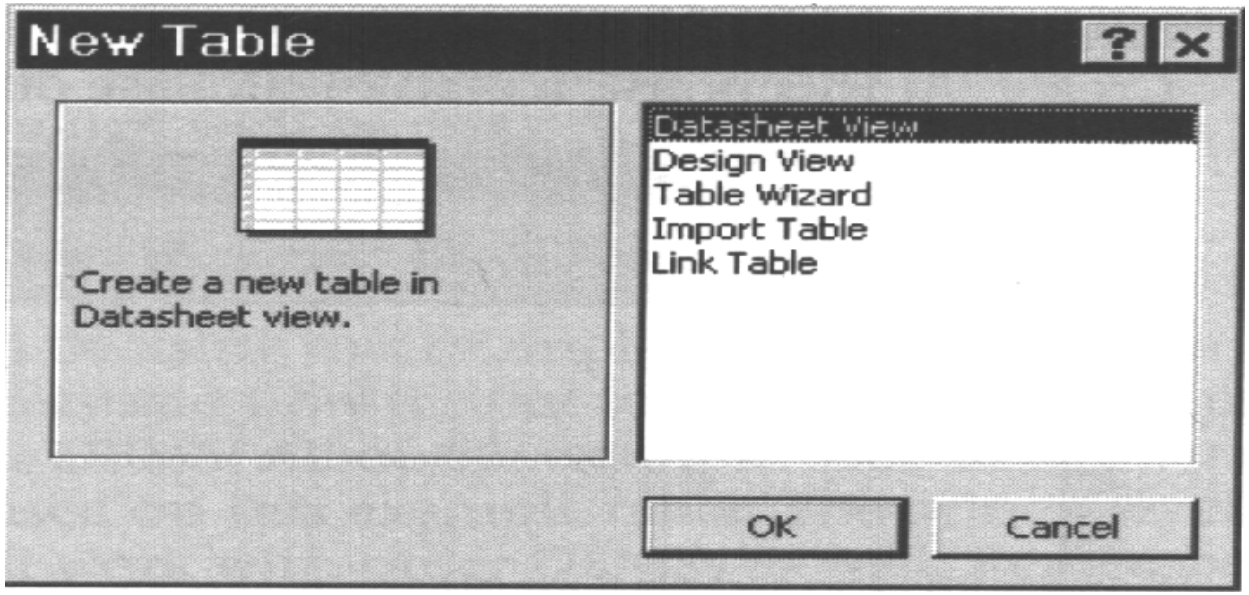
注释 下面章节中的所有过程都假定，你打开了示例数据库并准备使用。如果在前面的步骤中都遵守了示例的约定，应该会看到 Sample: Database 对话框。

5.3 添 加 表

在示例数据库中，我们需要使用两个表。第一个表 Foods（食品）包含一系列食品库存项。第二个表 Orders（订货）包含食品项的订货信息。我们从创建 Foods（食品）表开始。

1. 选择 Sample: Database（示例：数据库）对话框的 Tables（表）选项卡。

2. 单击 New（新建）。会看到如下图所示的 New Table（新建表）对话框。



3. 加亮 Design View（设计视图）选项，然后单击 OK。会看到如图 5.5 所示的 Table（表）设计对话框。

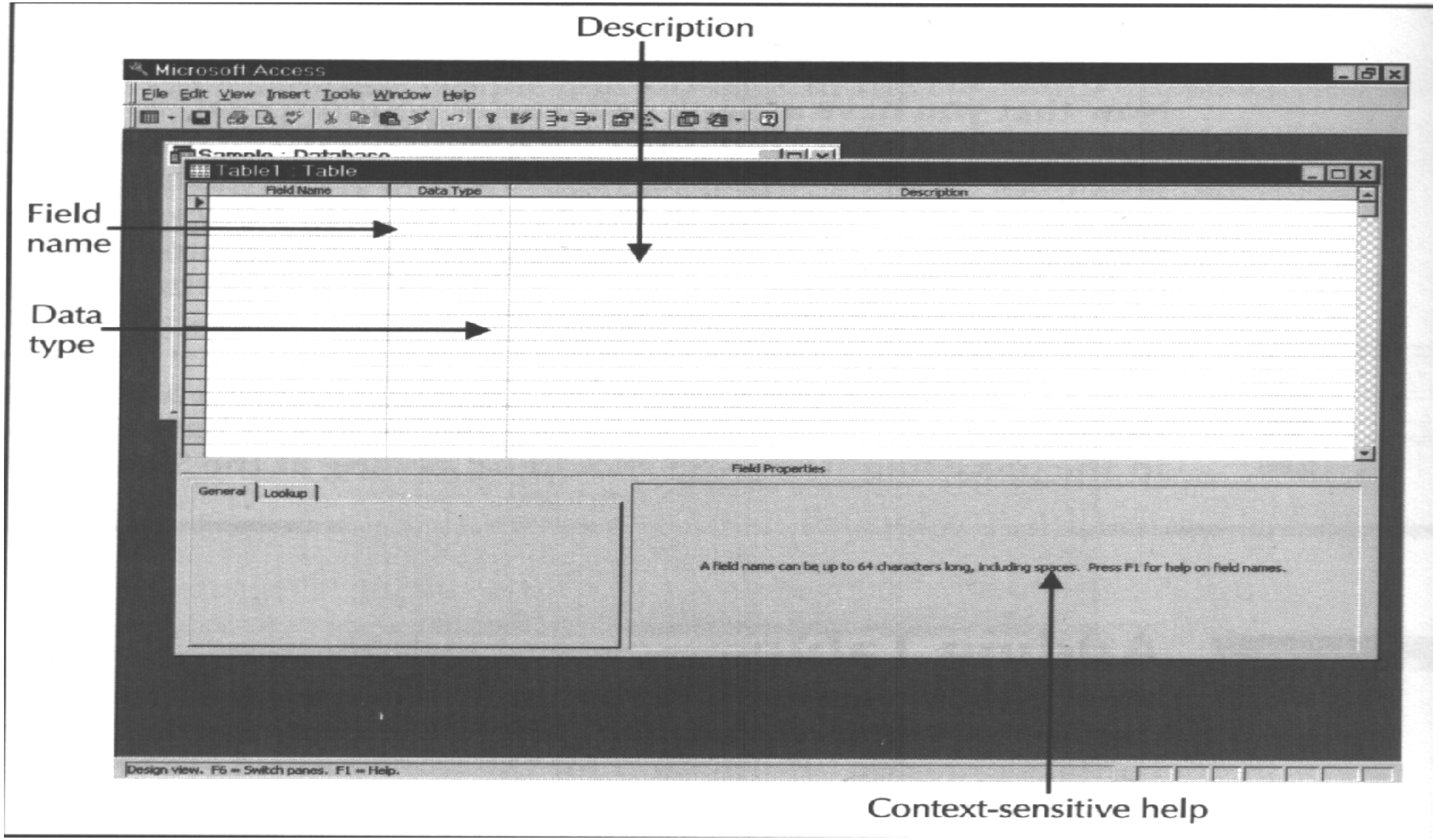


图 5.5　Table（表）设计对话框让你创建示例数据库所需的表

我们花几分钟讨论一下 Table（表）设计对话框。在这个表中有三列。第一

列 Field Name（字段名）包含字段的名称。在这里需要填入的只是描述这个字段将保持信息类型的名称。虽然 Access 允许在字段名中使用空格，但要避免这样去做，以便 Visual C++能够更加容易地访问该字段。在字段名中单词间添加空格的最佳方式是，使用像这样的下划线：Field_Name。还要放弃使用任何非字母数字字符，如磅符号(#)或和符号(&)——使用这些字符肯定会导致 C++程序发生故障。

第二列 Data Type（数据类型）定义字段将要保存数据的数据类型。Access 定义了大量的数据类型，各数据类型如下所示。

注释 下列数据类型反映了 Office 97 版的 Access 中所有的数据类型。你会发现，较早版的 Access 提供的可供选择的数据类型要少一些。我们不会在示例数据库中使用任何更新的数据类型，所以较早版的 Access 应该可以良好地运行。

- Text（文本） 包含 255 个或更少字符的短字符串。
- Memo（备忘录） 用于存储注释或其它大段文本的长字符串。
- Number（数字） 任何数值值，包括 Byte（字节），Integer（整数），Long Integer（长整数），Single（单精度实数），Double（双精度实数）和 Replication ID（复制 ID）。

注释 C++程序员会通过术语 GUID（全局唯一标识）了解 Replication ID（复制 ID）。在 C++中，Replication ID 用于识别 Access 对象。

- Date/Time（日期/时间） Access 中几种日期和/或时间格式中的一种。

- Currency（货币） 用来存储货币值的特殊数字格式。
- AutoNumber Access 可以提供一个计数器作为表的主键。主键确保每一个记录都有唯一的可以用来访问它的值，所以创建主键是必要的。
- Yes/No C++程序员会把它作为 Boolean（布尔）值来识别。可以把它格式化为 Access 中的 Yes/No，On/Off 或 True/False。但是，格式化该值的能力并不影响它仍是一个 Boolean（布尔）值这一事实。
- OLEObject（OLE 对象） 一种由主机支持的 OLE 对象。
- Hyperlink（超链接） 这种数据类型最近才被添加到 Access 中。它允许你存储去往网络资源的 UNC（统一命名协定）路径或去往 Internet 资源的 URL（统一资源定位器）。Access 甚至支持在 Web 网页上使用锚地。关于 URL 和锚地的详尽讨论，请参见第 8 章。
- Lookup Wizard（查找向导） 在有些情况下，你想让用户从特定的选择列表中选择，而不是把值输入到表中。Lookup Wizard（查找向导）允许你创建对数据库中另一个表的引用，用来填充这个字段。用户会在屏幕上看到一个列表框或组合框，可以在第二个表的预定义选择列表中选择。

Table（表）设计对话框的第三列 Description（描述），允许你向表中添加注释，以便说明字段的作用。当数据库变得很大时你就会发现，以文档化方式说明数据库每个表中各个项的准确目的是极为必要的。有时候你会发现，起初设计数据库时合理的字段不再需要了。你会发现，如果没有正确的文档，要定位这些消耗额外空间的字段就困难多了。

在 Table（表）设计对话框的左下角，你会看到一个被称为 Field Properties（字段属性）的带选项卡的对话框。在这里输入表的 Access 专用格式化信息。在很多情况下，会看到极小化了的 Caption（标题）属性。Caption（标题）属性定义了 Access 如何在窗体和报告中说明相应的字段。你还会发现，设置标题使设计应用程序变得稍微容易了一些，因为它促使你在设计工程的时候就考虑了用户界面问题。

就在 Table（表）设计对象中 Field Properties（字段属性）的右边，有一个凹下的方框。这个框通常包含了有用的提示信息。例如，如果单击 Field Name（字段名）列，这个框会告诉你，包括空格在内字段名最多可以有 64 个字符长。

现在我们已经熟悉了 Table（表）设计对话框的工作方式，那就来创建两个表把。下图是 Foods（食品）表中包含的字段（简单地输入图中所示的各项，图中还显示了应该使用哪一个数据类型）。

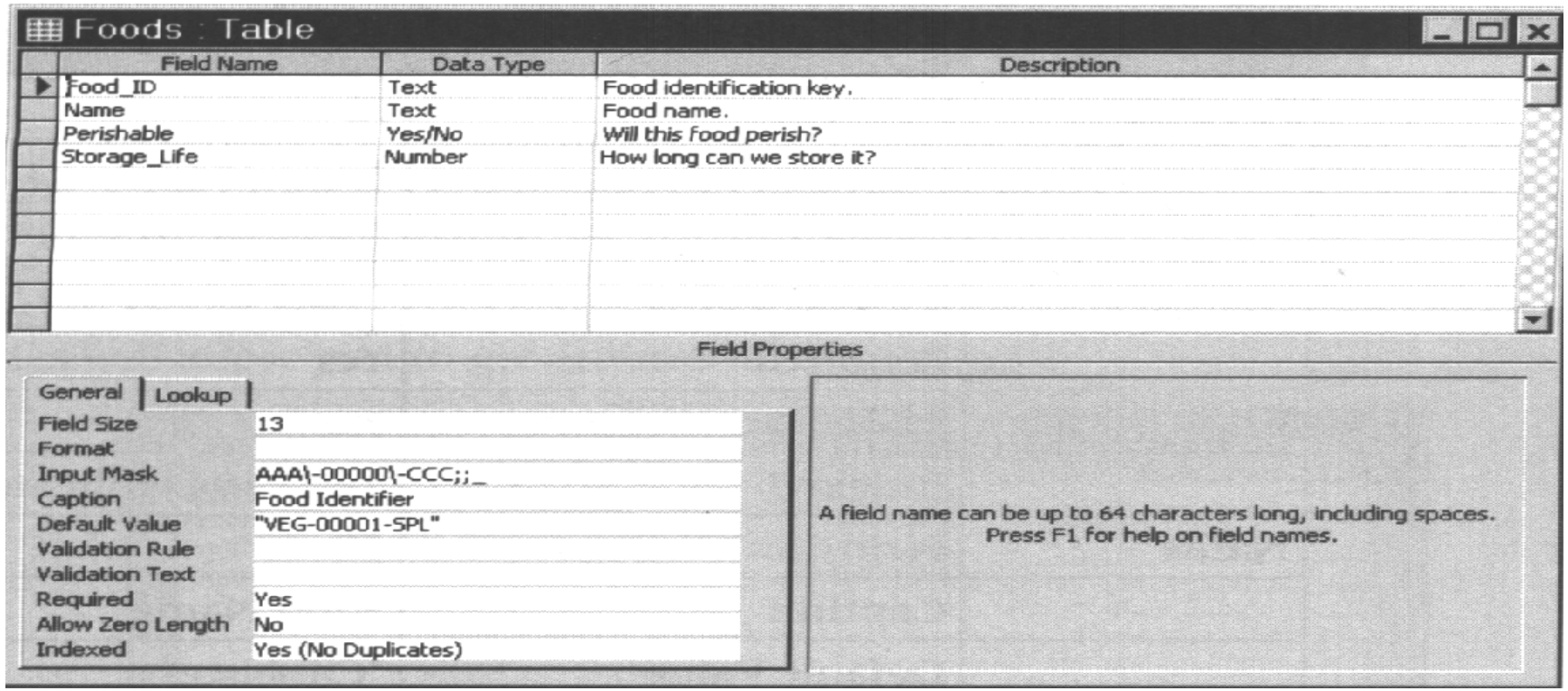


表 5.1 包含了需要为每一个字段输入的属性值。

在完成这张表之前，我们还要做上两件事。要做的第一件事是定义主键。我们要用 Food_ID 字段来建立 Foods（食品）表和 Orders（订货）表的关系，所以把它作为我们的主键。采用下面的过程定义表的主键。

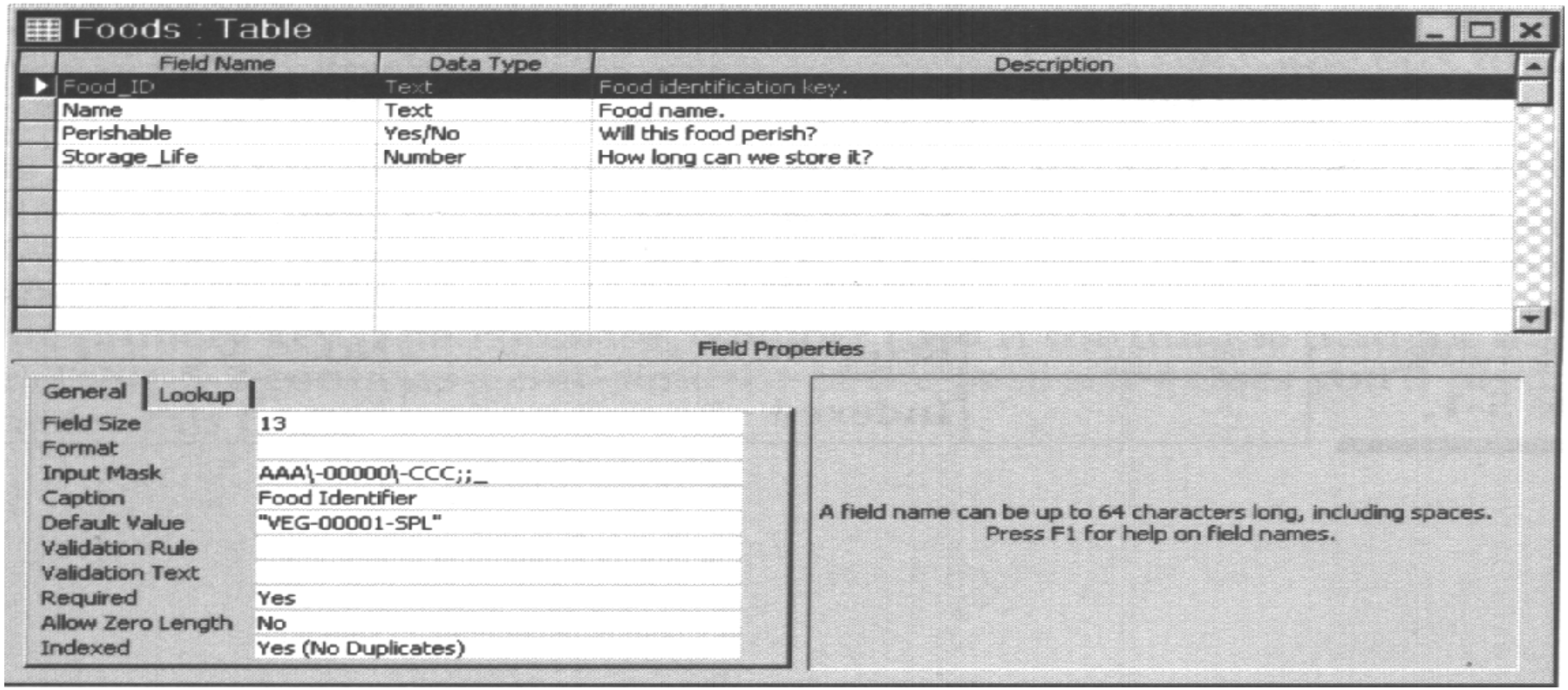
表 5.1　Foods 表的属性值

字段	属性名	值
Food-ID	Field Size	13
	Input Mask	AAA\-00000\-CCC;;_
	Caption	食品标识符
	Default Value	"VEG-00001-SPL"
	Required	Yes（不复制）
	Allow Zero Length	No
	Indexed	Yes（不复制）
Name	Field Size	50
	Caption	食品名称
	Default Value	"卷心菜"
	Required	Yes
	Allow Zero Length	No
	Indexed	Yes（不复制）
Perishable	Caption	易腐坏吗？
	Default Value	True
	Validation Text	必须输入 Yes 或 No
	Required	Yes
	Indexed	Yes（复制完成）
Storage_Life	Caption	保存期限

续表

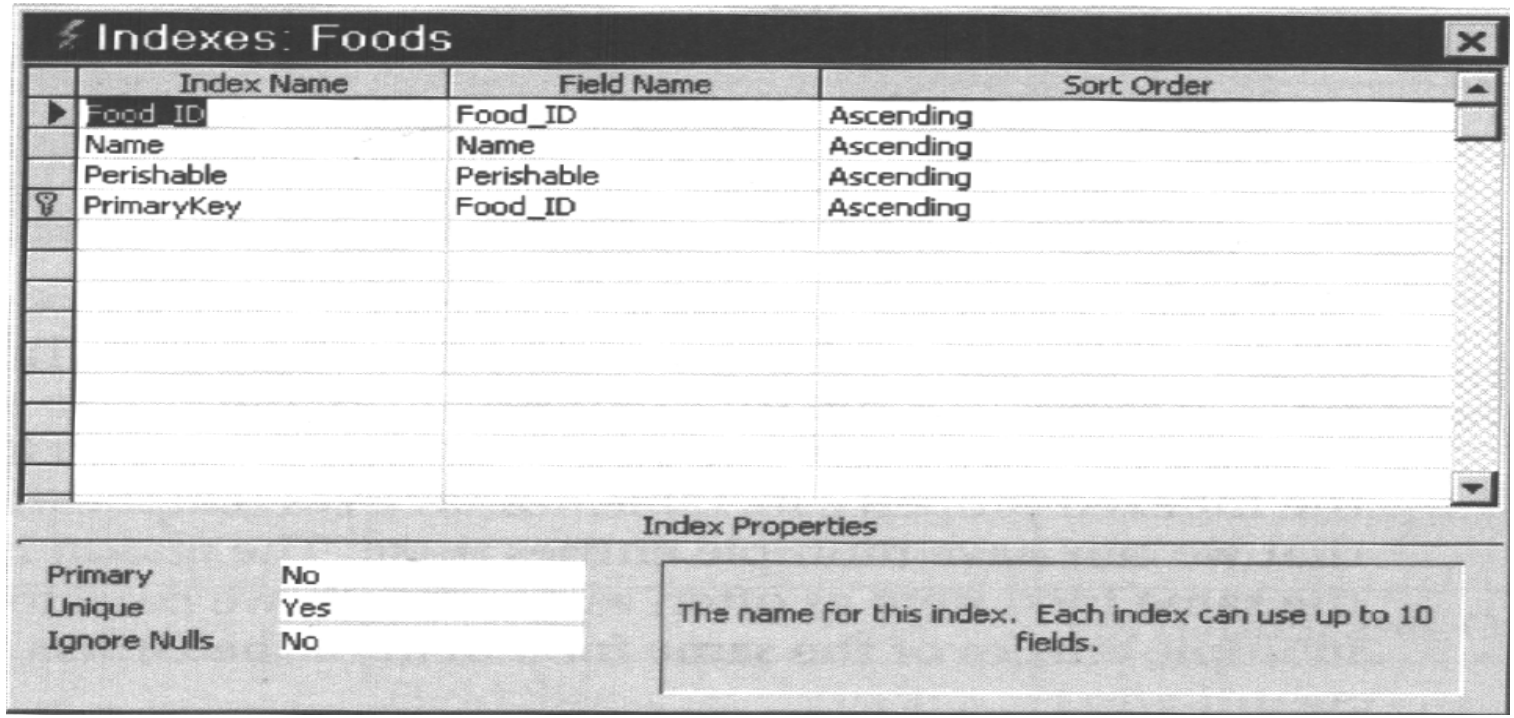
	Default Value	7
	Validation Rule	>0
	Validation Text	值必须大于 0
	Required	No
	Indexed	No

1. 单击 Food_ID 字段左边的方框，整个字段被加亮，如下图所示。



2. 用 Edit|Primary Key（编辑|主键）命令创建主键。Access 会在 Food_ID 字段的左侧添加一个像钥匙一样的小图标。

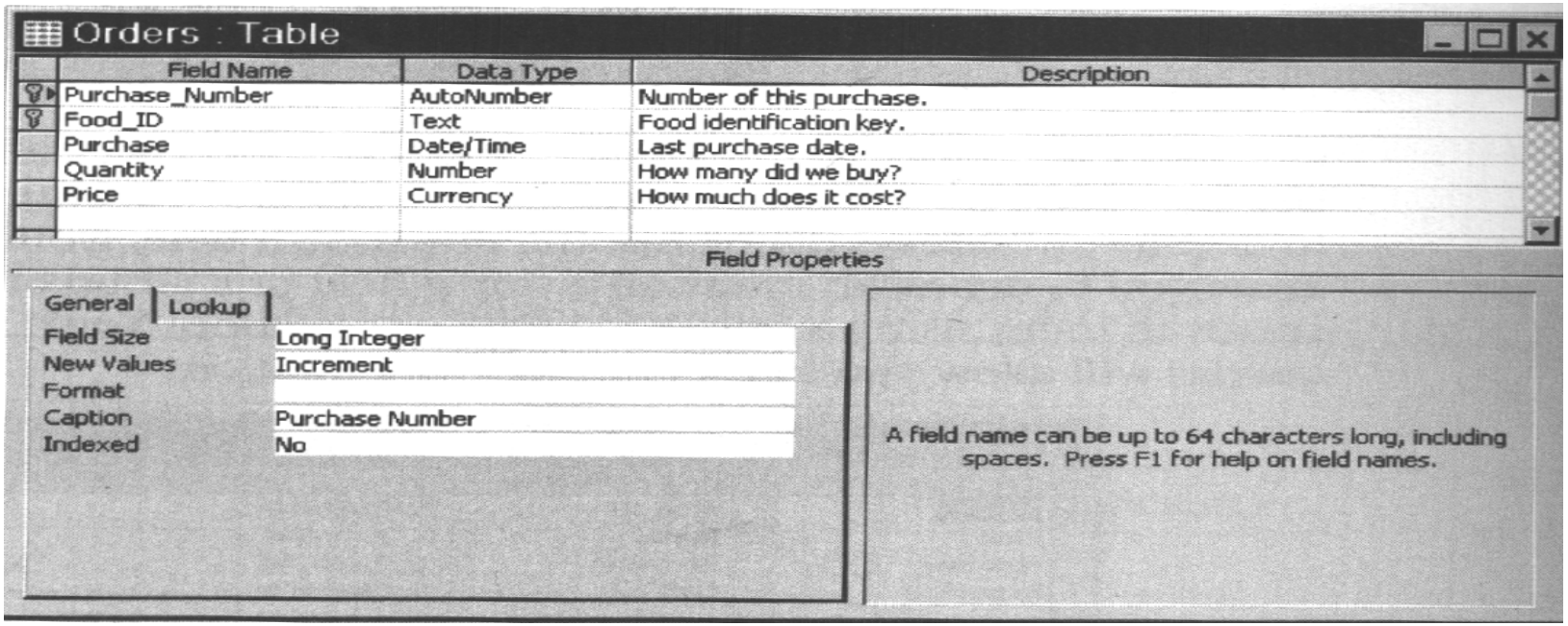
定义了主键后，你会想到要看一看所有索引。用 View|Indexes（视图|索引）命令或单击工具条上的 Indexes（索引）按钮显示 Indexes（索引）对话框，如上图所示。请注意，Access 允许定义单个字段或一组字段的索引；但不能用计算域或等式来创建索引。你会在后面看到，使用查询可以突破这个限制。



单击 Indexes（索引）对话框上的 Close（关闭）框将其关闭。现在保存我们的第一张表。单击工具条上的 Save（保存）按钮。Access 显示一个对话框，要求输入表名。输入 Foods，然后单击 OK。表名会出现在 Sample:Database 对

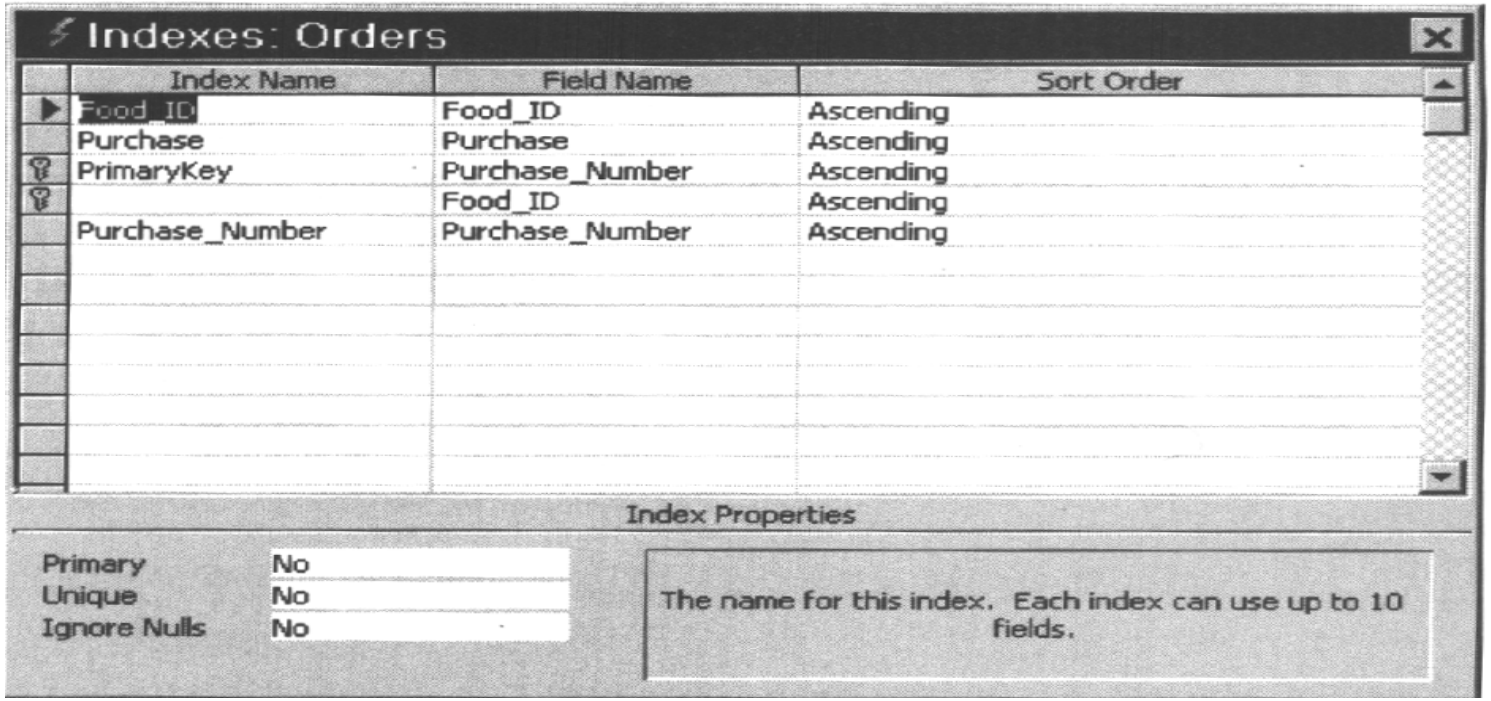
话框中。单击 Close（关闭）框关闭 Table（表）设计对话框。

我们用相同的过程创建第二张表。在此情况下，表名是 Orders。下面是 Orders 表所需的全部字段项（图中还包含了每个字段的数据类型）。请注意，Purchase_Number 和 Food_ID 字段都被用作主键。Purchase_Number 是这张表中取唯一值的字段。表 5.2 包含了完整的需要为这张表定义的属性值。



你应该注意到，表 5.1 中 Food_ID 字段的属性和表 5.2 中相同项的属性存在一个主要的差别。表 5.1 在 Food_ID 字段上建立了索引（参看 Indexed 属性），并且不允许有重复值。原因在于对数据库中的每一个食品来说，我们只想要一个数据项。但是，表 5.2 中的 Indexed 属性值为 Yes，这就是说我们在这里可以有多个重复项。原因很简单：你可能经常需要订购相同的食品，所以我们允许

用户在 Orders 表中使用相同食品的多个重复项。



我们再检查一下 Indexes（索引）对话框。单击工具条上的 Indexes（索引）按钮。应该看到如下图所示的画面。请注意，我们定义了一个多字段索引(PrimaryKey)，方法是，在同一行上输入索引名和第一个字段名，第二行及后续行只包含一个字段名。

像前面一样，关闭 Indexes 对话框，然后保存新建的表。表名一定要是 Orders。

现在已经有两张表可以使用了，下面开始填充一些数据。图 5.6 显示了 Foods 表的数据，图 5.7 显示了 Orders 表的数据。如果你喜欢的话，可以再添加一些其它数据；只是要保证，对 Foods（食品）表中创建的每一个项，在 Orders（订

货）表中都要有一个或多个对应的项。同样地，要保证不把任何在 Foods（食品）表中没有对应项的记录输入 Orders 表中。

表 5.2 Orders 表的属性值

字段	属性名	值
Purchase_Number	Field Length	长整数
	Caption	采购编号
	Indexed	Yes
Food_ID	Field Size	13
	Input Mask	AAA\-00000\-CCC;;_
	Caption	食品标识符
	Default Value	"VEG-00001-SPL"
	Required	Yes
	Allow Zero Length	No
	Indexed	Yes
Purchase	Format	中间日期格式
	Caption	最近一次采购的日期
	Default Value	Now()
	Required	Yes
	Indexed	Yes
Quantity	Field Size	长整数
	Caption	我们买了多少？

续 表

	Default Value	1
	Validation Rule	>0
	Validation Text	必须大于 0
	Required	Yes
	Indexed	No
Price	Format	货币格式
	Caption	以美元为单位的价格
	Default Value	0
	Validation Rule	>0
	Validation Text	必须大于 0
	Required	Yes
	Indexed	No

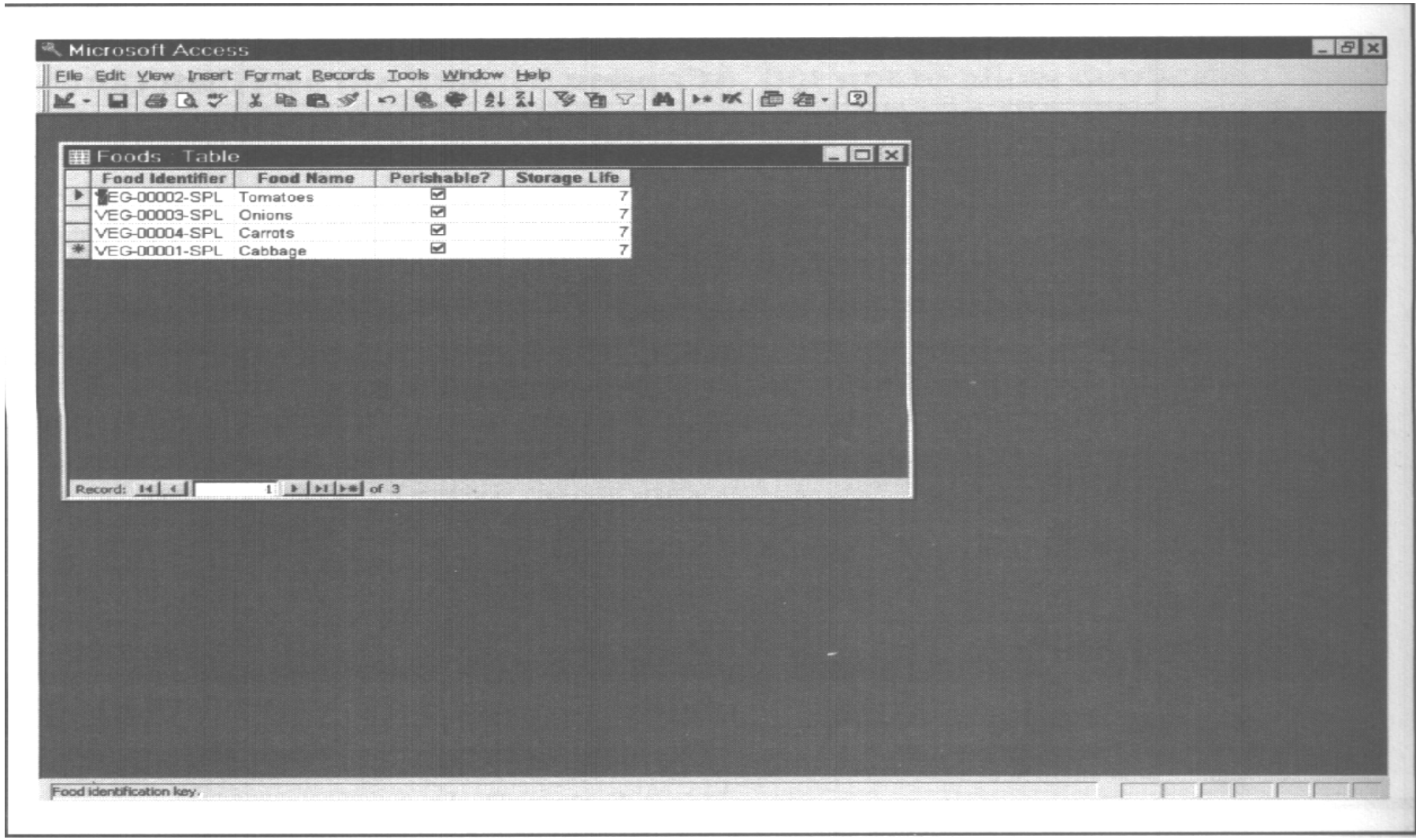


图 5.6　对 Foods（食品）表中的每一个项，一定要在 Orders（订货）表中也有一个对应项

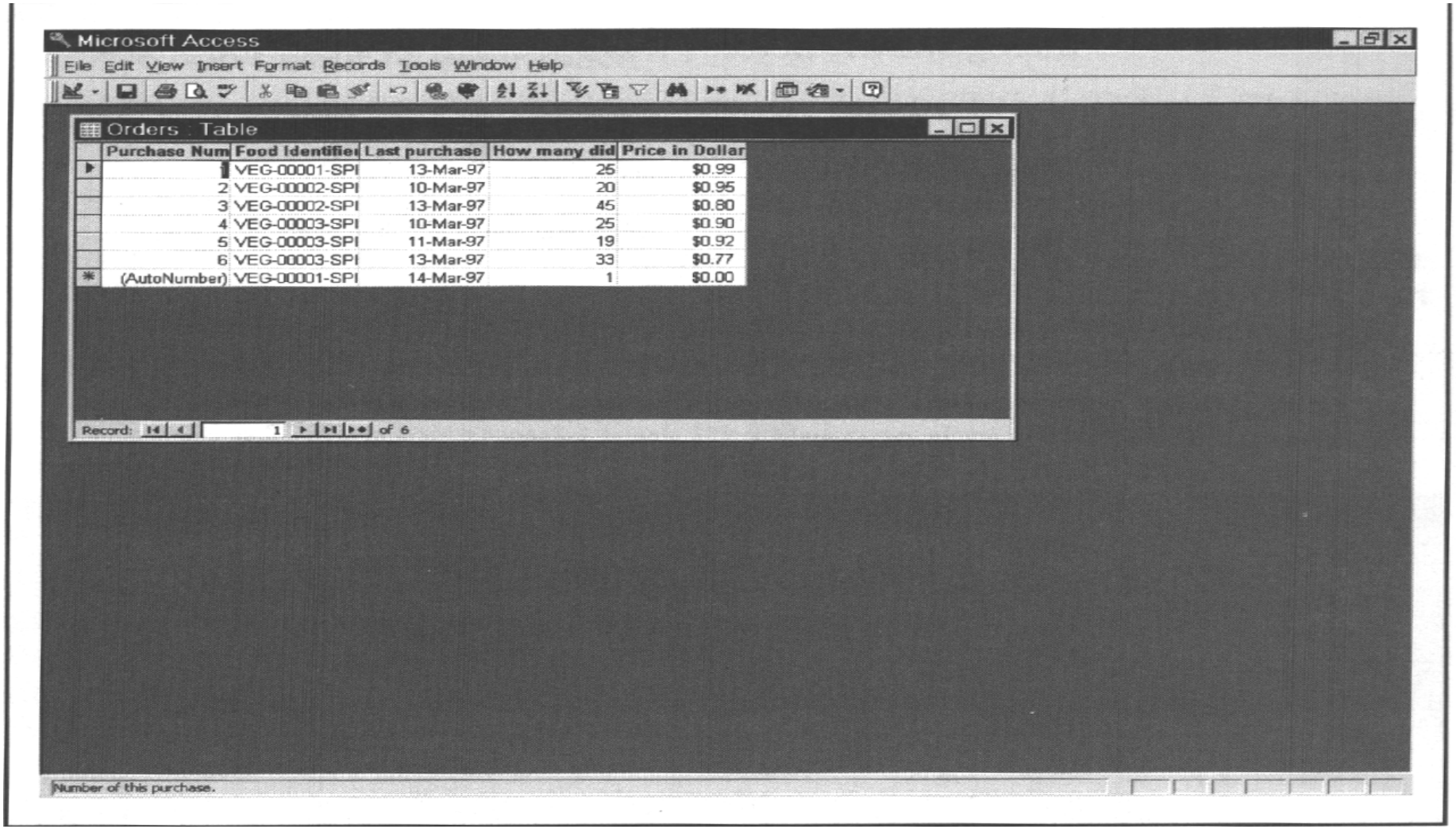


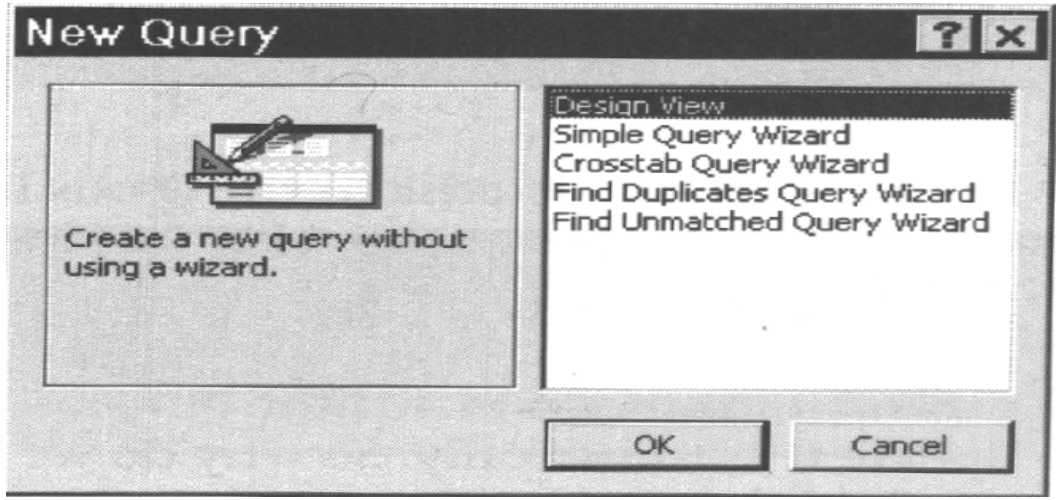
图 5.7　不要在 Orders（订货）表中创建任何在 Foods（食品）表中无匹配 Food_ID 的记录

5.4 使用查询排序数据

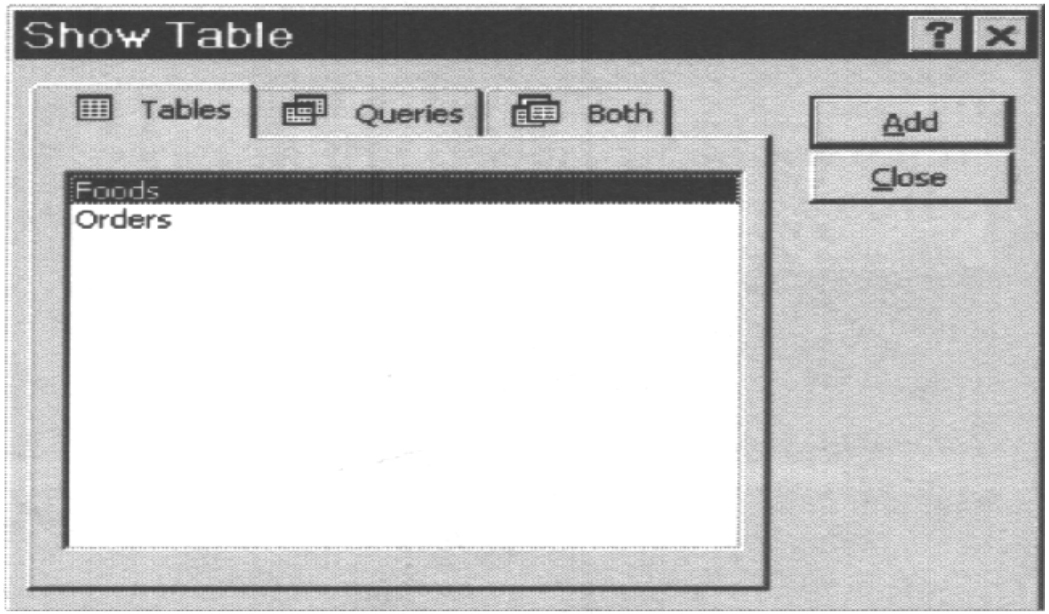
查询使精确查找所需数据的过程变得令人难以置信的容易。Access 利用可视查询界面使创建查询的过程变得更加容易。下面我们来看一看，为刚刚在上一节创建的两张表创建查询的过程。

1. 单击 Database（数据库）对话框的 Queries（查询）选项卡。

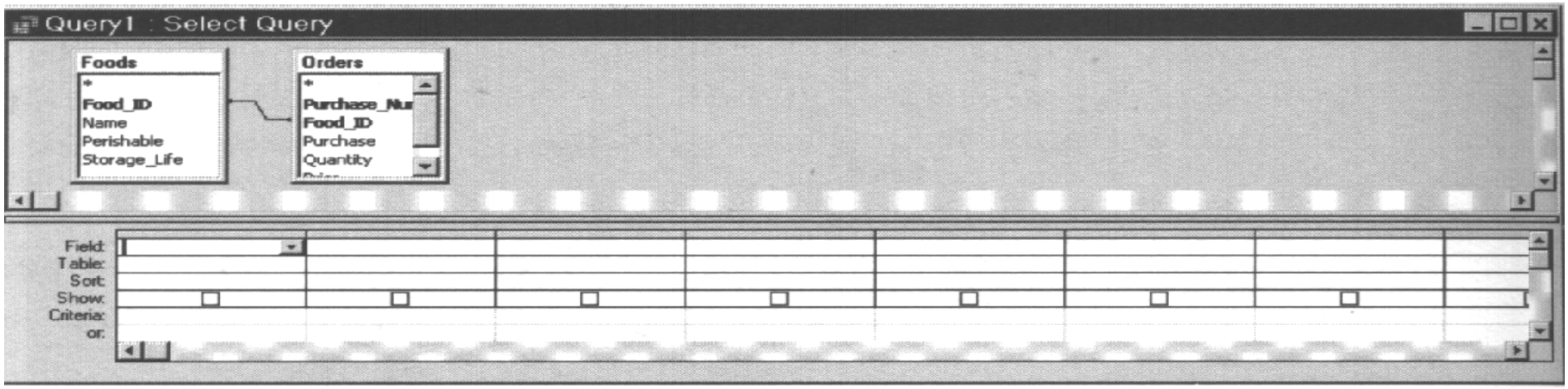
2. 单击 New（新建），会看到如下图所示的 New Query（新查询）对话框。



3. 选择 Design View（设计视图），然后单击 OK。你会看到如下图所示的 Show Table（显示表）对话框。



4. 加亮这两个表（Foods 和 Orders），然后单击 Add（添加）。单击 Close（关闭）。系统显示 Show Table（显示表）对话框，并且你会看到如下图所示的 Select Query（选择查询）对话框。请注意，Access 自动创建两个表之间的逻辑关系，主键为粗体字体。现在，我们要决定在查询中显示哪些字段。



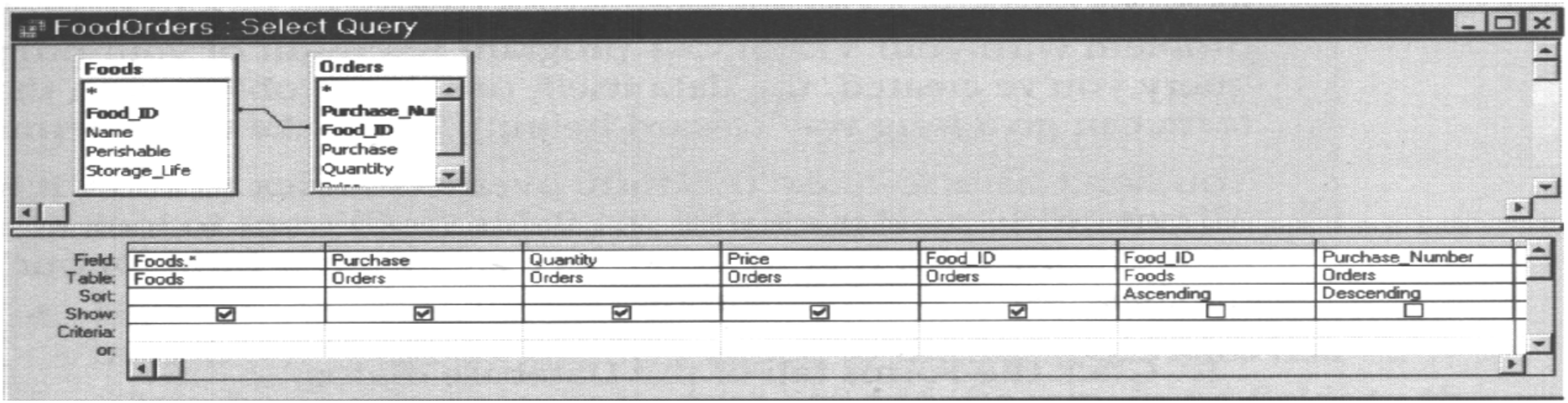
5. 双击 Foods 列表框中的星号。这是告诉 Access，查询中想使用这个表的所有字段。

6. 双击 Orders 表中的 Foods_ID，Purchase，Quantity 和 Price 字段。这是告诉 Access，查询中我们只想使用这个表中的三个字段。不需要选择 Purchase_Number 字段，因为 Access 会自动更新它。那么，为什么我们需要两个 Food_ID 字段呢？我们要用这个字段来建立两个表之间的关系，所以这样做是很重要的。否则，就不能向 Orders 表中添加记录。现在，我们要决定查询的显示顺序。

7. 双击 Foods 表中的 Food_ID 字段。在对话框的下半区中，去掉 Show（显示）属性行中该字段对应的复选标记。我们不想把这个字段显示两次。

8. 在 Food_ID 字段的 Sort（排序）属性中选择 Ascending（升序）。这是告诉 Access 我们想如何排序信息。光用 Food_ID 字段排序还不够，因为我们在 Orders（订货）表中有这个字段的多个拷贝。

9. 双击 Orders 表中的 Purchase_Number，并去掉 Show（显示）属性的复选标记。在 Sort（排序）属性中选择 Descending（降序）。现在，我们有了一种以特别的顺序来显示 Orders 表中所有项的方式。在本例的情况下，首先以最近一次订货的升序方式来显示食品。此时，Select Query（选择查询）对话框如下图所示。



可以测试一下所创建的查询，以确定它们是否按预期方式运行。此时，完成测试的一种最容易的方式是，单击工具条上的 Run（运行）按钮。你应该看到如下图所示的画面。

FoodOrders : Select Query

Food Identifier	Food Name	Perishable?	Storage Life	Last purchase	How many did	Price in Dollar	Food Identifier
VEG-00001-SPL	Carrots	☑	7	13-Mar-97	25	$0.99	VEG-00001-SPI
VEG-00002-SPL	Tomatoes	☑	7	13-Mar-97	45	$0.80	VEG-00002-SPI
VEG-00002-SPL	Tomatoes	☑	7	10-Mar-97	20	$0.95	VEG-00002-SPI
VEG-00003-SPL	Onions	☑	7	11-Mar-97	19	$0.92	VEG-00003-SPI
VEG-00003-SPL	Onions	☑	7	13-Mar-97	33	$0.77	VEG-00003-SPI
VEG-00003-SPL	Onions	☑	7	10-Mar-97	25	$0.90	VEG-00003-SPI

Record: 1 of 6

请注意，这次查询中显示了 Orders 表的全部六个数据项，并且只有我们已选择的字段被显示出来。正如我们所要求的那样，所有的食品标识符都以字母数字排序。另外，每项的最新定单显示在前面而不是后面。

现在保存查询。单击 Save（保存）按钮，然后在 Save As（另存为）对话框中输入 FoodOrders。单击 OK 完成这个过程。单击 Close（关闭）按钮关闭这个查询。

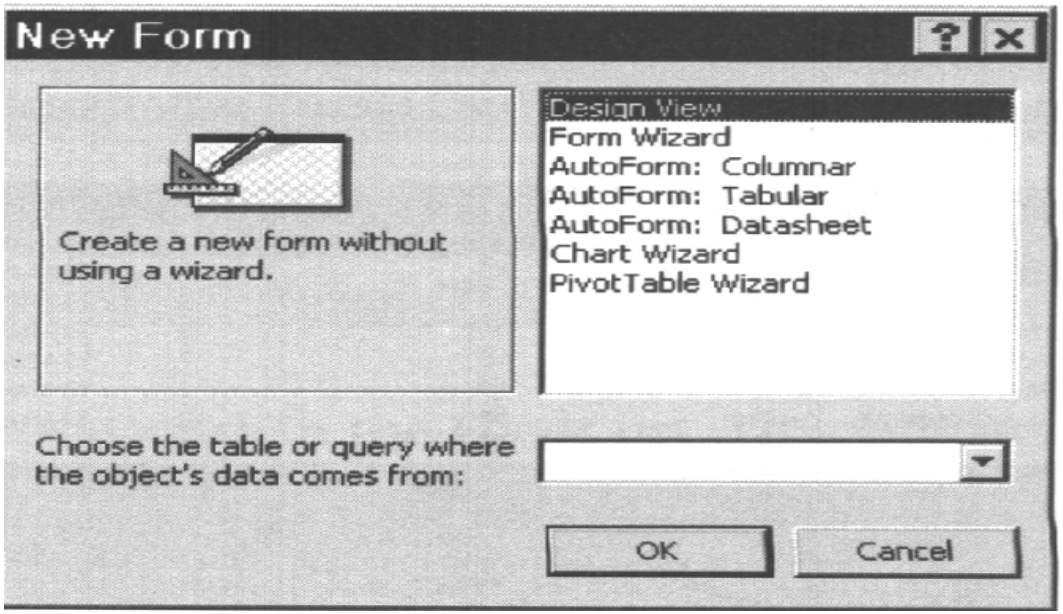
5.5 创建测试窗体

对 Visual C++程序员来说，在 DBMS 内创建窗体有两个重要的原因。第一，可以在产品类型的环境中再测试一次查询，这个环境和你在 Visual C++下测试的环境是一样的。第二，测试窗体允许你不必创建特殊的应用程序即能查看数

据。无论是程序代码、已创建的查询、数据本身或者 DBMS 的故障导致了 Visual C++程序出现错误，你都可以很方便地把错误找出来。

要让窗体完成查询任务，无须把它做得过于复杂。真正需要显示的字段是想在用户应用程序中看到的哪些字段。记住了这一点后，我们用下面的过程为示例数据库创建一个测试窗体。

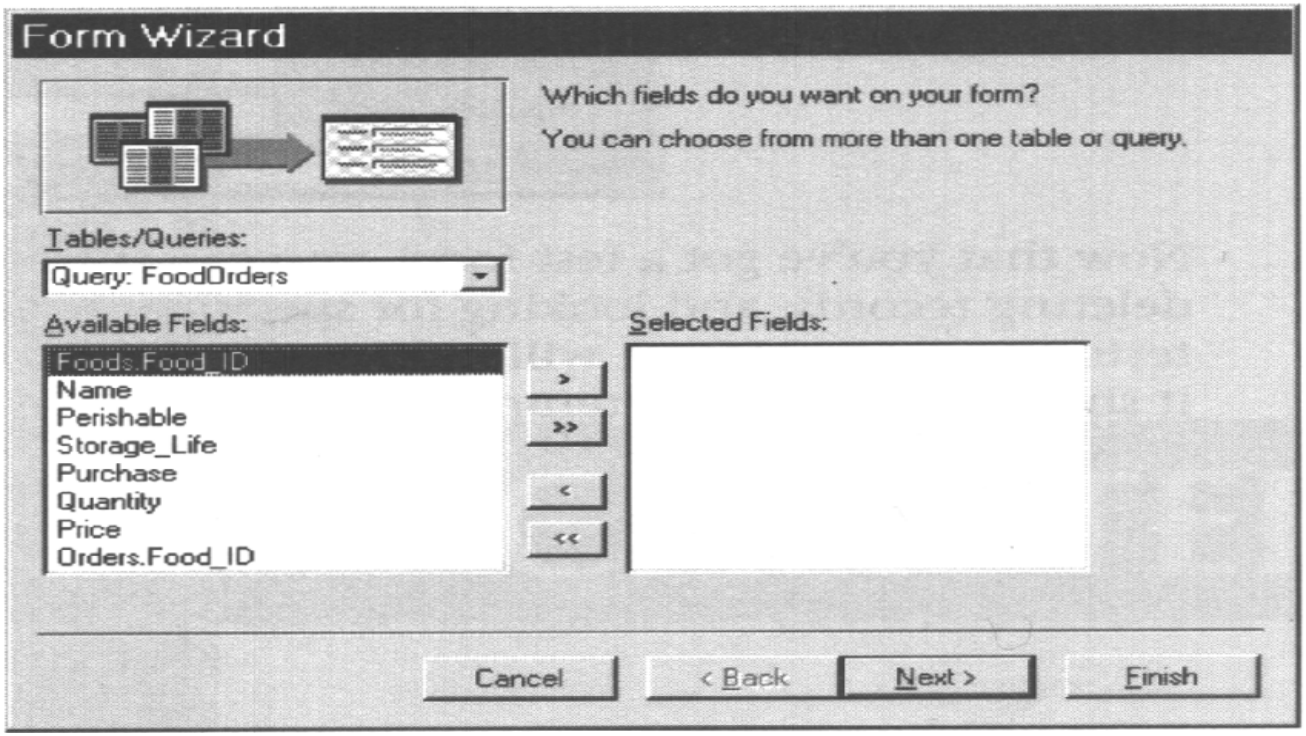
1. 单击 Database（数据库）对话框的 Forms（窗体）选项卡。



2. 单击 New（新建），会看到如下图所示的 New Form（新窗体）对话框。

3. 加亮 Form Wizard（窗体向导），在下拉列表框中选择 FoodOrders 查询，然后单击 OK，会看到如下图所示的 Form Wizard（窗体向导）对话框。仔细看一看字段列表。验证一下 Form Wizard（窗体向导）是否从查询中挑选了你所期

望的全部字段，这一点也很重要。

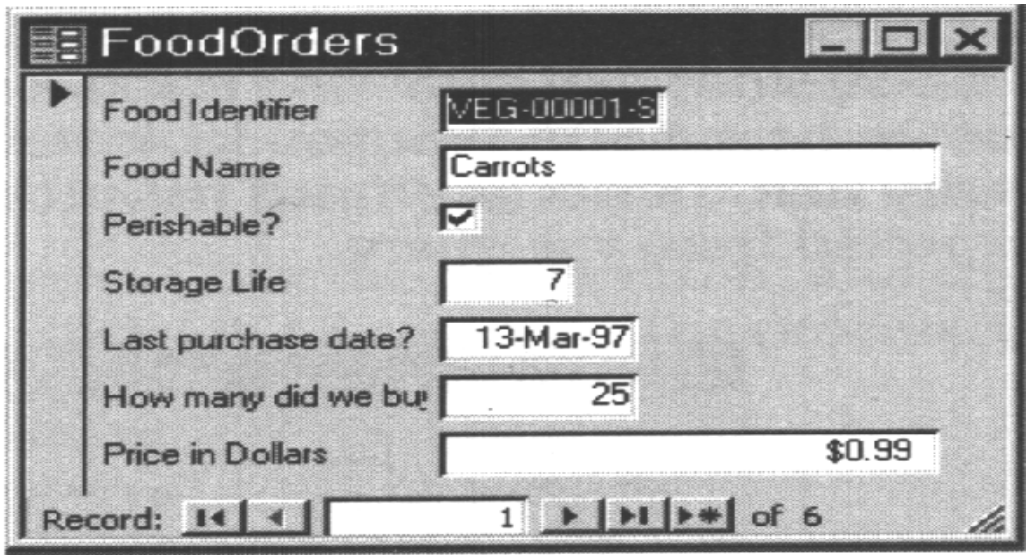


4. 加亮 Orders.Food_ID 字段，然后单击右箭头按钮(>)，会看到该字段被添加到 Selected Fields（已选定字段）列表中。

5. 单击双箭头按钮(>>)，把剩下的字段添加到 Selected Fields（已选定的字段）列表中。由于我们并不需要 Foods.Food_ID 字段，所以加亮该字段，然后选择左箭头按钮(<)，把它从 Selected Fields（已选定字段）列表中删除。

6. 由于我们对窗体的美学特性还不感兴趣，所以单击 Finish（完成）。你会看到数据的单记录视图，如下图所示。你应该再次看一看字段列表，确认所

需字段都已经出现在窗体上。应该做的第二件事是，验证所选字段的标题都正确无误。最后，验证屏幕上显示的数据是否正确。



现在我们已经有了测试窗体，你也可以试着用窗体添加新记录、删除记录和查找特定的信息，等等。在 Visual C++应用程序使用数据库之前，在实际窗体上进行上述类型的测试，可以帮助你验证是否一切以正常的方式运行。

第 6 章　使用 ODBC 建立数据库应用程序

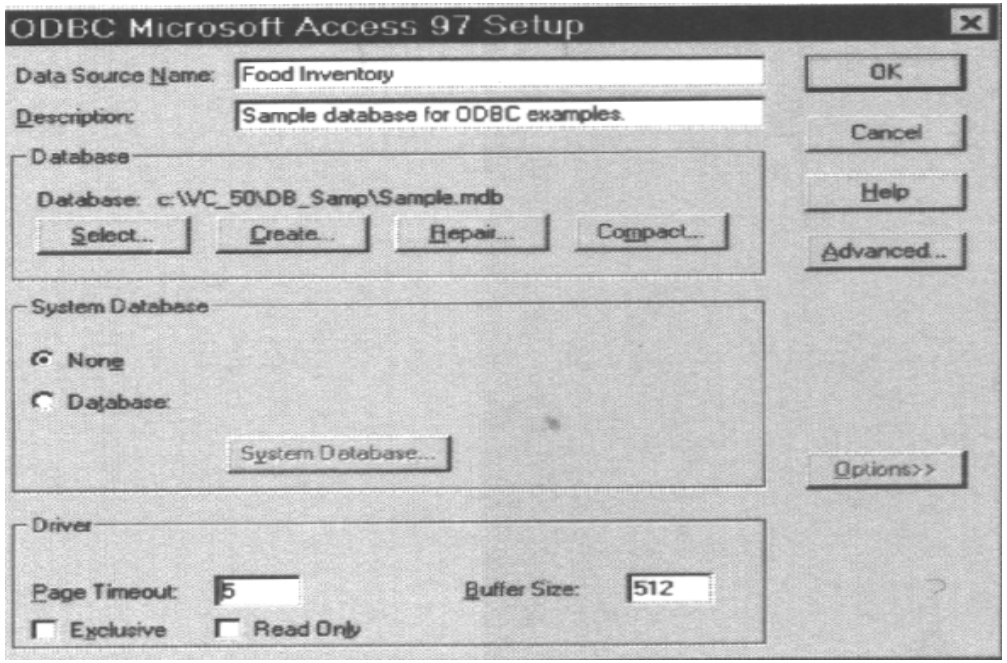
我在前面两章介绍了一些你会在本章中实际用到的概念。我们做的第一件事是，学习一些 ADO，OLE-DB 和 ODBC 理论。此时，假设你决定用 ODBC 来编写应用程序，而不是用 ADO 或者 OLE-DB。如果没有读过第 4 章，现在可能需要补读，这样才有利于下一步的学习。

我们在第 5 章花了一些时间学习如何创建数据库。实际上，本章假定你在那一章中构造了示例数据库，这里就不再讨论示例数据库了，但是我们要在示例代码中使用这个示例数据库。本章的大部分内容只和那两个表和查询有关；如果你确实想跳过示例窗体（第 5 章中说明的窗体）的话，也是完全可以的（但是，示例窗体可以帮助克服表或查询存在的问题）。

Web 链接　谈到使用 ODBC 时，可以花一些时间查阅一下新闻组。以 Access 使用 ODBC 时，一个更为流行的 Microsoft 发起的新闻组是 microsoft.public.access.odbcclientsvr。如果使用了 SQL Server，还可以查阅一下 microsoft.public.sqlserver.odbc 新闻组。当然，除了我们在第 5 章中谈到的更为一般的数据库新闻组外，还有其它许多专门讨论 ODBC 的新闻组。

在开始看一些示例 ODBC 数据库应用程序之前，还有最后一件事要做。回

忆一下第 4 章，在 Visual C++能够实际处理示例数据库之前，我们需要一个 ODBC 源。可以用第 4 章“使用 ODBC”一节中的过程创建数据源名称(DSN)。下面讲述本章所用的 DSN 参数。



技巧 在开始编写使用 ODBC 的 Visual C++程序之前，要设置所有的 DSN。否则的话，以后你将不得不重新编写某些代码。甚至在启动 Visual C++之前就可以完成全部所需的设置，这样可以让 MFC AppWizard 为你做尽可能多的工作。

请注意，我把 Data Source Name（数据源名称）字段设置为"Food Inventory"，Description（描述）字段设置为"Sample database for ODBC examples."。还要注

意在本章示例中使用的数据库的位置和名称。为了适应系统的设置，可能需要修改这些设置；用第 4 章阐明的 Select（选择）按钮选择示例数据库。最后，把选项设置成缺省值，不要更改 Set Advanced Options（设置高级选项）对话框上的任何设置。除非设置了数据库安全保护，否则不需要添加缺省用户名或口令——我们在第 5 章中没有设置安全保护。

技巧 一定要以计划使用的 DBMS 测试 ODBC 应用程序，然后使客户知道可能存在的应用程序限制。一些程序员用一个版本的 DBMS 开发应用程序，得到了有趣的结果，一些客户在相同 DBMS 兼容版本的产品环境中也做了一些尝试。例如，一些 Office 97 的用户突然发现，其应用程序的 ODBC 访问次数增加了四倍之多，这使得应用程序事实上不可用（并不是每个人都会碰到这种问题，但它确实给出了一个很好的例子，即在正确的情况下可能发生什么）。即使应用程序是按预想方式工作的，把 Access 的 Office 95 版更新成 Office 97 版也会产生这样预想不到的副作用，即增加了访问次数。这个应用程序的开发者可能是这样避免潜在的问题的，他保证使客户理解其应用程序只是在测试状态下使用 Office 95，或者他用 Office 97 来测试应用程序以寻找存在的副作用。

6.1 创建简单的窗体视图应用程序

我们从创建最容易的数据库应用程序类型开始，用 Visual C++创建在第 5 章中创建的 FoodOrders 查询的窗体视图。使用查询是启动应用程序并使其运行的最快的方式，因为查询允许你选择想要查看的字段并给数据排序，根本用不着任何代码。下面是创建基本数据库的过程。我们在第 2 章中讲了用 Visual C++创建基本单文档应用程序的要点，这里就不再详细研究了。

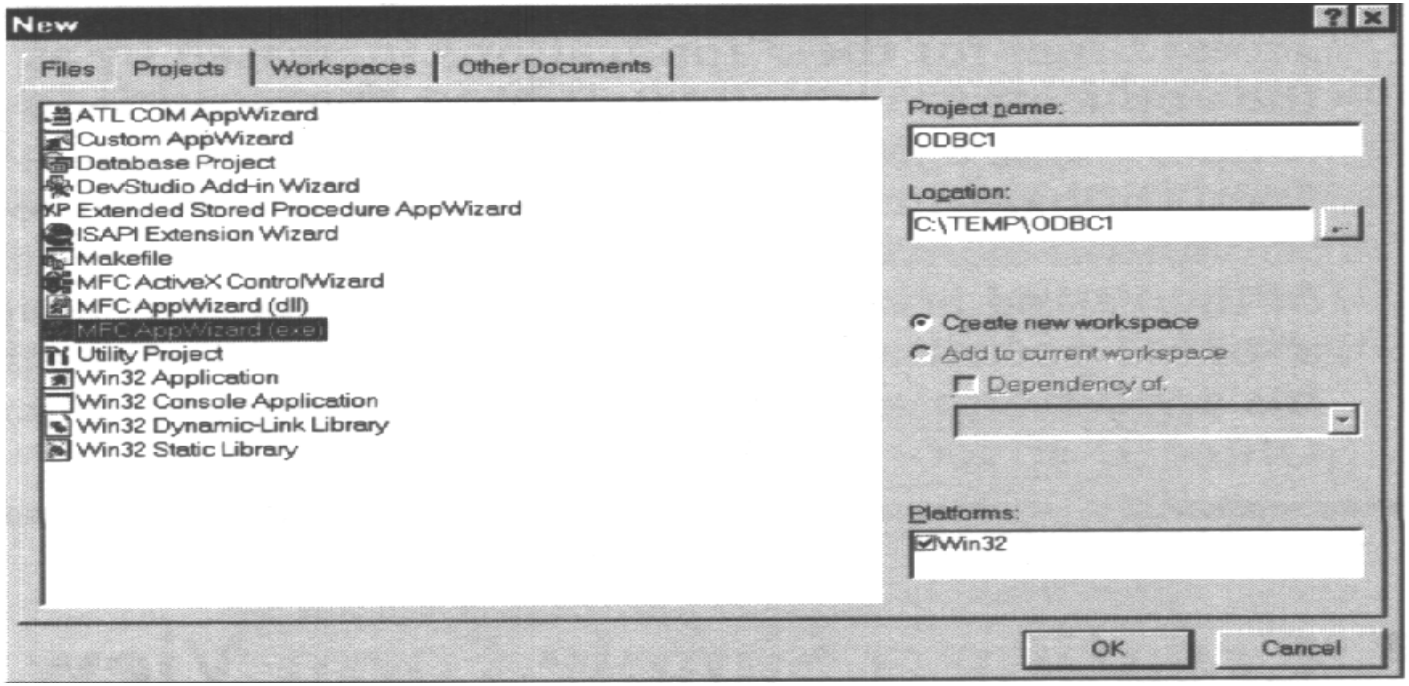
技巧 有一种情况应该使用表而不是查询，这一点十分重要。如果想创建一个实用程序来修复用户以某种方式损坏的数据库，可能需要进行某种单表访问。否则，即使发现了故障，Access（或任何其它好的 DBMS）所执行的引用完整性规则也可能要阻止你实际修复故障。尽管如此，通常情况下，应该避免更改单独的表，原因在于当应用程序正在运行且引用完整性规则生效时，单表操作的后果难以预料。

---

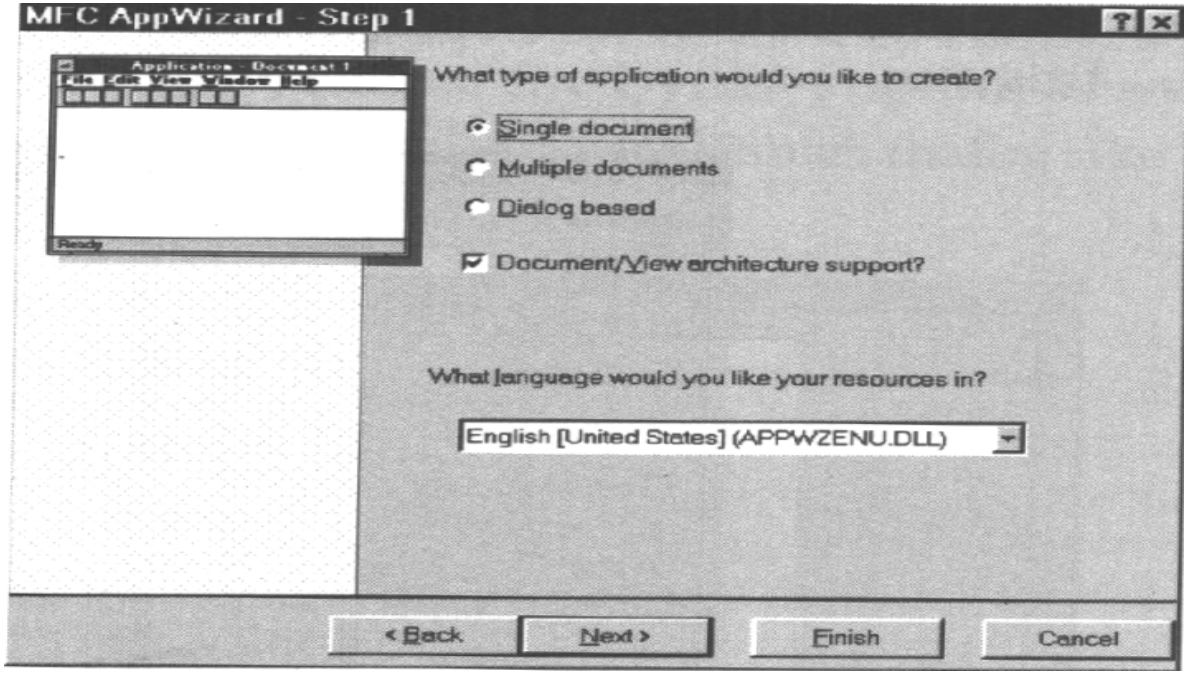
1. 启动 Visual C++。

2. 用 File|New（文件|新建）命令显示 New（新建）对话框。选择 Projects（工程）选项卡。

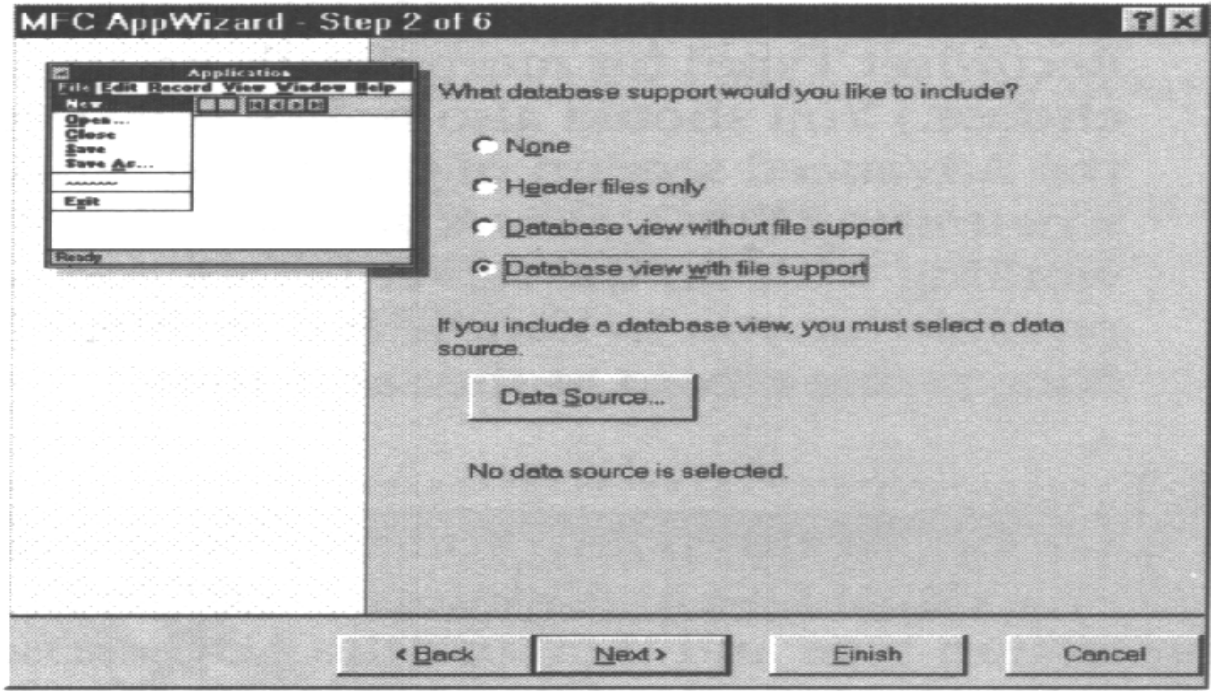
3. 在 Project Name（工程名称）域中输入新的应用程序名（本示例使用 ODBC1），然后加亮 MFC AppWizard(EXE)项，如下图所示。



4. 单击 OK。会看到如下图所示的 MFC AppWizard-Step1 对话框。

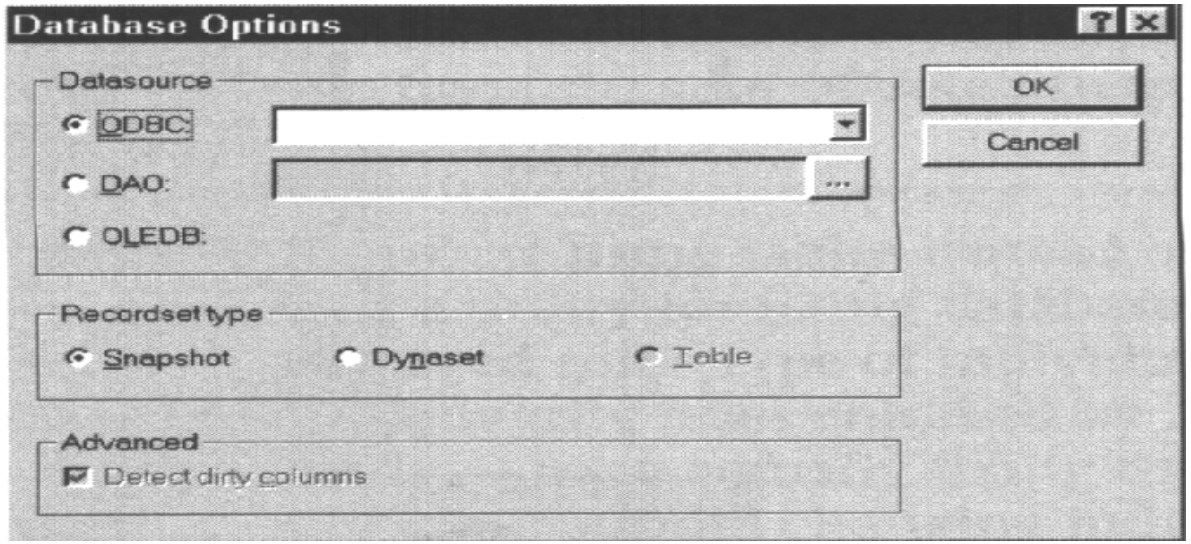


5. 选择 Single Document（单文档），然后单击 Next（下一个）。会看到如下图所示的 MFC AppWizard-Step 2 of 6 对话框。此时，要选择数据库选项（关于使用 Visual C++时数据库选择的详细资料，参见题为“选择数据库选项”的高级技巧框）。



6. 选择 Database View with File Support（带文件支持的数据库视图）选项。现在需要选择数据源。请注意，无论选择 Database View with File Support（有文件支持的数据库视图）选项还是 Database View without File Support（无文件支持的数据库视图）选项，Data Source（数据源）按钮都是可以使用的。

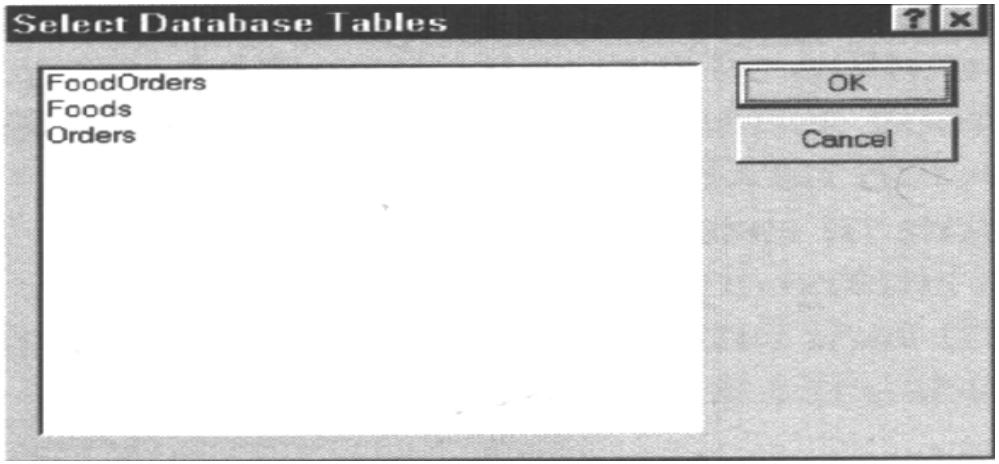
7. 单击 Data Source（数据源）按钮。会看到如下图所示的 Database Options（数据库选项）对话框。



这又是那样一种情形，即在用 MFC AppWizard 设计应用程序的时候，必须确实注意所进行的操作。请注意，这个对话框不但允许你选择数据源，还允许你选择所需的访问类型。我已经在第 4 章中谈过了 ODBC、DAO 和 OLE-DB 之间的差别。但是请注意，在选择 ODBC 或 DAO 时还可以选择三类 Recordset（记录集）（关于应该选择的记录集选项的详细资料，参见题为“选择记录集类型”的高级技巧框）。还应该注意对话框的 Advanced（高级）组框中的 Detect Dirty Columns（检测脏列）复选框。这个选项仅在使用 DAO 时是有效的，它允许你在写入更改之前，检测服务器上数据的变化。这就能够处理两个用户试图在同一时间更新同一个记录中数据的问题了。

注释 此时，你可能想知道，如何在 Visual C++应用程序内使用 ADO 呢？你应该选 OLE-DB 作为连接类型，然后在应用程序中使用 ADO 类。这可能有一点让人费解，但当你意识到 OLE-DB 是 ADO 的基础访问技术时，也就明白其中的奥妙了。

8. 选择 ODBC 选项，因为我们会在本章中使用 ODBC 访问。应该在下拉列表框中选择 Food Inventory（食品库存）列表项——这是我们在本章开始时创建的 DSN。要想马上看到对数据库所做的更改，还要选择 Dynaset（动态集）选项。



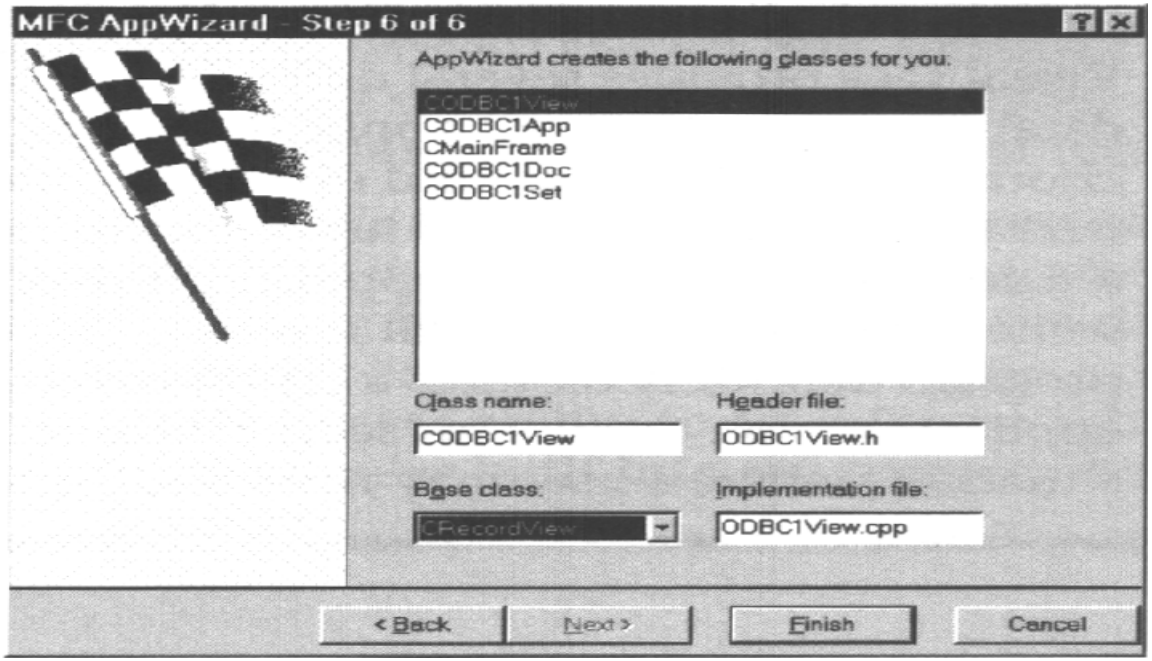
9. 单击 OK。会看到如下图所示的 Select Database Tables（选择数据库表）对话框。

10. 加亮 FoodOrders（食品订货）选项。这是我们在第 5 章中创建的查询。请注意，你也可能选择两个表中的一个，但通常会因为我们在前面谈到的理由

而选择查询。一定要在这里选择正确的条目，因为今后不会再有第二次选择机会了。和简单地从头编写应用程序相比，因选择错误而产生的工作量更大，如果不仔细的话，会导致代价昂贵的时间延误。

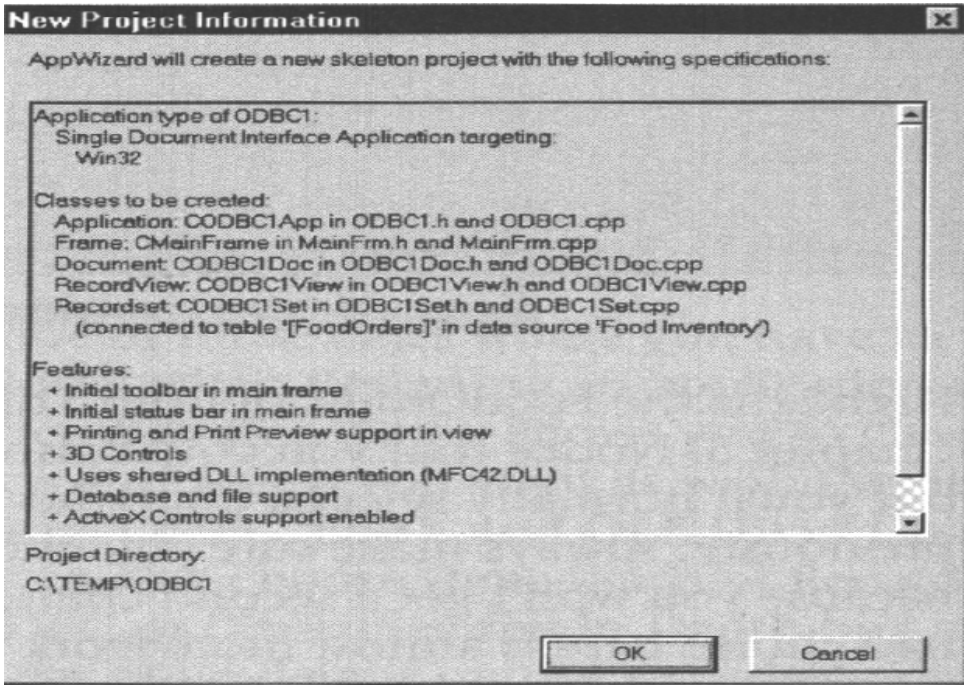
11. 单击 OK 返回 MFC AppWizard-Step2 of 6。此时，已经用我们在第 5 章中创建的示例数据库配置完应用程序，可以使用了。

12. 单击 Next（下一个）四次。应该看到如下图所示的 MFC AppWizard-Step 6 of 6。



13. 单击 CODBC1View。请注意，CRecordView 是应用程序的基类。在后续章节中会发现，无须接受这个缺省值，就像我们在第 2 章单文档应用程序中没有接受缺省值一样。

14. 单击 Finish(完成)。Visual C++显示如下图所示的 New Project Information(新建工程信息)对话框。



15. 单击 OK。Visual C++生成应用程序。

高级技巧

选择数据库选项

Visual C++提供了几个不同的选项，允许或禁止应用程序中的数据库支持。所选择的数据库支持层次依赖于正创建的应用程序的类型，虽然这并不是需要考虑的唯一标准。例如，你可能需要数据库是应用程序的一部分（像游戏程序），而不需要用户直接和它交互作用。同样地，就所要求的支持层次而言，实用程序肯定和你为安全数据任务所创建的应用程序不同。但是，用户可能需要访问同一数据库的几个不同版本，所以要允许用户从应用程序内把它打开。下面的一览表可以帮助你理解可用的选项。

- None（无） 选择这个选项意味着不想要任何数据库支持。今后再添加它是很烦琐的，所以如果有应用程序需要某种数据库支持的可能性，就请仔细考虑一下这个选项。甚至，非数据库应用程序都要经常利用数据库来存储配置设置或者其它信息。众所周知，游戏程序内含扩展数据库，你还会发现，几乎没有帮助桌面应用程序可以不使用数据库。

- Header Files Only（只要头文件） 如果认为需要数据库支持但确实对其细节不清楚的话，这是应该选择的最好选项。Visual C++向应用程序中添加所要求的头文件，但会要求你自己创建数据库类。实际上，这项工作比想像得要少，因为 Class Wizard 会为你完成创建类的最困难部分的工作。你要做的，只是添加实际检验数据库内容所需的代码。
- Database View without File Support（无文件支持的数据库视图） 有时候，你不想用户在可用文件的外部对文件做任何处理。这个选项要 Visual C++创建访问 ODBC 数据源所需的全部类，你已经提供了类，但不准备制作打开其它文件所需的菜单条目。自定义数据库应用程序的建立者们可能用这个选项用得最多。例如，如果这个应用程序被设计成用来编辑公司的工资表，就不想要用户私自打开文件，破坏安全措施。消除文件支持可能不会使数据库完全安全，但它确实排除了又一个潜在的安全漏洞。
- Database View with File Support（带文件支持的数据库视图） 实用程序总是使用这个选项，因为它为用户提供了最大限度的自由。不但你的应用程序会访问已被设计到应用程序中的数据源，而且你还可以打开外部文件。当然，用户实际使用这个特征的能力要依赖于已建立程序本身的能力。

高级技巧

选择记录集类型

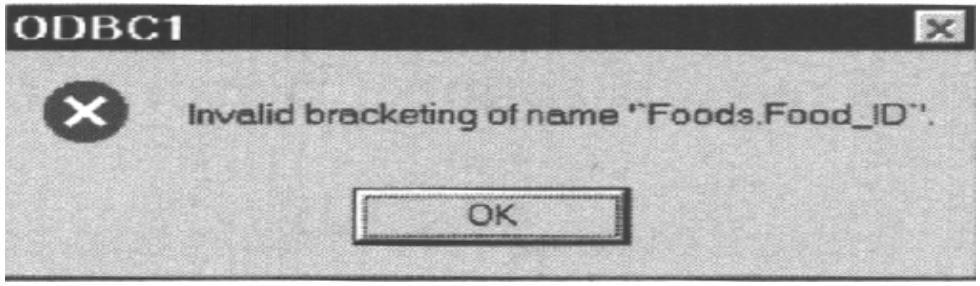
Visual C++提供了几种记录集，可以用来定制应用程序的工作方式。查看这些不同选项的最快方式要兼顾速度和特征。你会发现，在很多情况下，如果想添加特征，就必须付出程序执行速度降低的代价。下面告诉你一些可以自由支配的记录集选项。更重要的是，要告诉你从这个选项可以获得更快的速度还是更多的特征。

- Snapshot（快照） 这个选项要 Visual C++在一次快照中下载整个查询。换言之，及时快速地给数据库内容拍照，并把它作为未来工作的基础。这种方法有三个缺点。第一，你看不到别人在网络上做的更新，这可能意味着你的决定是建立在老信息的基础上。第二，一次就下载所有这些记录，这意味着在下载期间给网络增加了沉重的负担。第三，记录下载时用户会结束等待，这意味着网络的呼叫性能变得更低。然而这种方法也有两个优点。第一，记录一旦被下载，该工作站所需的网络活动几乎就没有了——这为其它请求释放了带宽。总之，你会看到网络的吞吐量增大了。第二，因为所有被申请的记录都在用户的机器上，所以用户实际上会得到应用程序更佳的总体性能。你可能想把快照的方法限制在较小的数据库上使用，原因在于快照适用于用户请求信息而不适用于数据编辑会话。

- Dynaset（动态集） 使用这个选项时，Visual C++创建指向所请求的每个记录的实际指针。另外，只有填充屏幕时实际需要的记录是从服务器上下载来的。这种方法的好处很明显。几乎马上就能在屏幕上看到记录。而且还会看到其它用户对数据库所做的更改。最后，其它用户也会看到你做的更改，因为动态集在你更改记录时被上载到服务器上。很明显，这种方法要求对服务器的实时访问，它减小了网络总吞吐量并降低了应用程序的性能。这个选项适合于创建用户要花费很多时间来编辑数据的应用程序。同时，它也是大型数据库的最佳选择，原因在于只需下载用户实际需要的信息。
- Table（表） 我们已经考察了记录级的其它两个选项。表方法（仅当使用 DAO 时可用）把所做查询的内容放进一个临时表。这样做不但减小了从服务器下载的信息量，还意味着程序员有了更大的灵活性，因为可以直接操纵临时表字段和记录。你必须为这个选项付出多大代价呢？如果不仔细的话，从服务器下载的过程实际上可能比记录子集花的时间还要长。另外，还有更新方面的问题——你看不到别人所做的更改，他们也看不到你所做的更改。当你正在使用 DAO 且用户会执行同等数量的数据查找和数据编辑时，这可能是最佳的选择。

修复 MFC AppWizard 产生的错误

现在已经有了应用程序的框架，可以编译它了（尽管你没有看到任何数据，但示例应用程序确实发挥作用了）。运行应用程序，你可能会对出现下图所示的错误对话框窗体感到惊奇（你的对话框看上去可能和我的略有不同，这要看 MFC 是连接应用程序的方式以及正在使用的 Visual C++的版本）。



如果没有看到这个消息，意味着 Microsoft 修复了 MFC AppWizard 中的这个问题，你可以跳过本节的剩余部分。错误消息实际上并不是来自应用程序；而是来自 Access，因为 MFC AppWizard 不能正确地处理复杂的字段引用。我们来看一下这个问题并找到最快的解决办法。

双击 CODBC1Set 类的 DoFieldExchange()函数。程序列表 6.1 显示了你将看到的示例。RFX_Text()函数正在把数据从数据库传输到我们的 Visual C++应用程序。请注意，第一个调用操作 Food_ID 字段。你需要把这个字段引用从"[Foods.Food_ID]"（即有一个名为 Foods.Food_ID 的字段）修改为"Foods.[Food_ID]"（即在 Foods 表中有一个名为 Food_ID 的字段）。要在最后一个 RFX_Text 调用中做相似的更改。把"[Orders.Food_ID]"引用修改为

"Orders.[Food_ID]"。

程序列表 6.1

```
void CODBC1Set::DoFieldExchange(CFieldExchange* pFX)
 {
 //{{AFX_FIELD_MAP(CODBC1Set)
 pFX->SetFieldType(CFieldExchange::outputColumn);
 RFX_Text(pFX, _T("[Foods.Food_ID]"), m_Food_ID);
 RFX_Text(pFX, _T("[Name]"), m_Name);
 RFX_Bool(pFX, _T("[Perishable]"), m_Perishable);
 RFX_Int(pFX, _T("[ Storage_Life]"), m_Storage_Life);
 RFX_Date(pFX, _T("[Purchase]"), m_Purchase);
 RFX_Long(pFX, _T("[Quantity]"), m_Quantity);
 RFX_Text(pFX, _T("[Price]"), m_Price);
 RFX_Text(pFX, _T("[Orders.Food_ID]"), m_Food_ID2);
 //}}AFX_FIELD_MAP
}
```

重编译应用程序。这一次运行时，就看不到任何错误了。当然，也看不到任何数据。我们会在下一节讨论这个问题。

添加一些数据显示代码

在我们准备向这个示例中添加代码之前，要执行几个步骤。必须做的第一件事是，创建用于显示数据的窗体。图 6.1 显示了我们这次要使用的窗体（可以根据个人爱好来修改它；只要保证所有字段都齐全就行了）。

对窗体上的各种控件，我使用了下面的 ID（从上到下）——它们与表所使用的字段值相对应：IDC_FOOD_ID，IDC_NAME，IDC_PERISHABLE，IDC_PRICE，IDC_PURCHASE，IDC_QUANTITY 和 IDC_STORAGE_LIFE。需要把 IDC_PRICE 控件的 Align Text（对齐文本）属性设置为 Right（右对齐），并选中 Multiline（多行）属性。对另外两个数字域，你也可以把它们的 Align Text（对齐文本）属性设置为 Right（右对齐），并选中 Multiline（多行）和 Number（数字）属性。

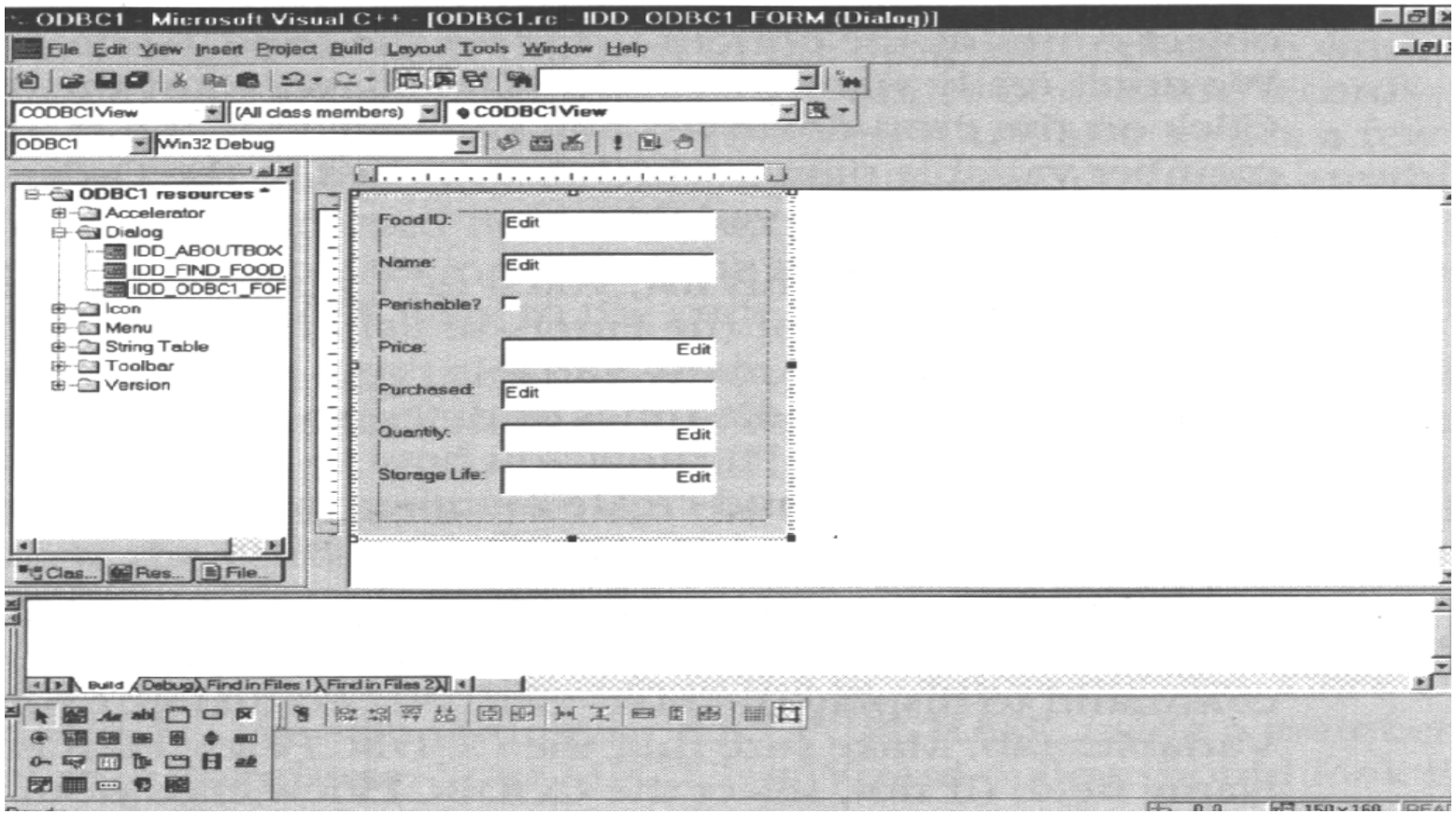
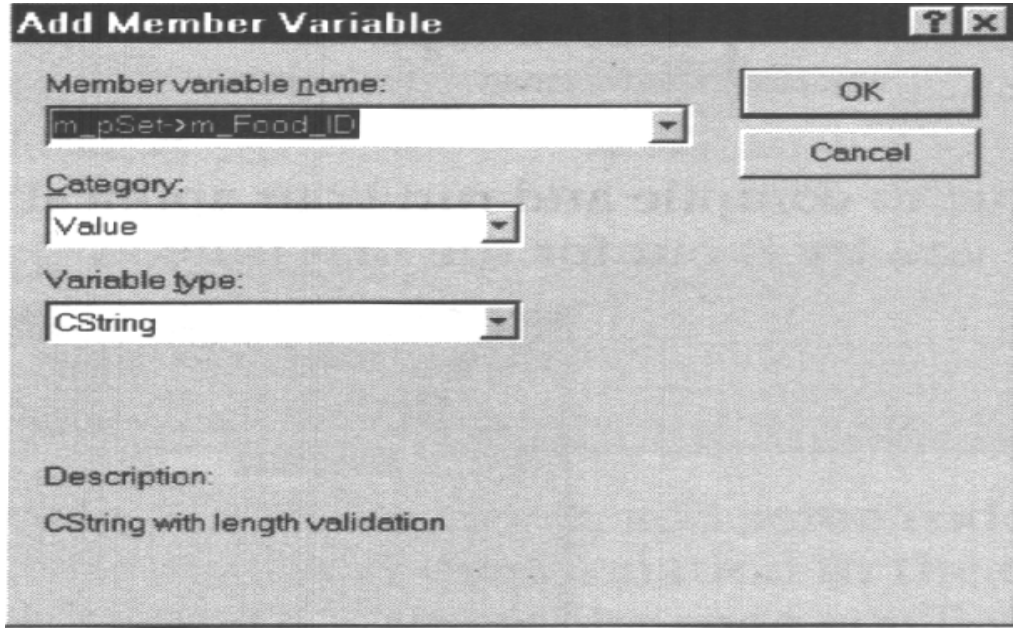


图 6.1　我们要构造一个窗体来显示数据库中的数据

现在我要说明为什么那些控件 ID 值非常重要。按住 CTRL 键并双击 IDC_FOOD_ID 控件，会看到一个如下图所示的 Add Member Variable（添加成员变量）对话框。



请注意，Visual C++已经在 Member Variable Name（成员变量名）字段中添加了一个值，但它不是我们需要的这个对话框中的正常值。通常你得到的变量以 m_开头，而不再带其它杂七杂八的东西。此时，如果你熟悉代码的话，应该知道 m_pSet 是指向包含当前数据库记录的对象的指针。那个对象中的 m_Food_ID 变量包含 Foods（食品）表中 Food_ID 字段的当前值。我们并不是真正想要那个 Food_ID 字段，而是想要 Orders（订货）表中的 Food_ID 字段。单击下拉列表框，你会看到 m_pSet->m_Food_ID2 成员变量名，它附属于 Orders（订货）表。选择 Orders（订货）表成员变量。

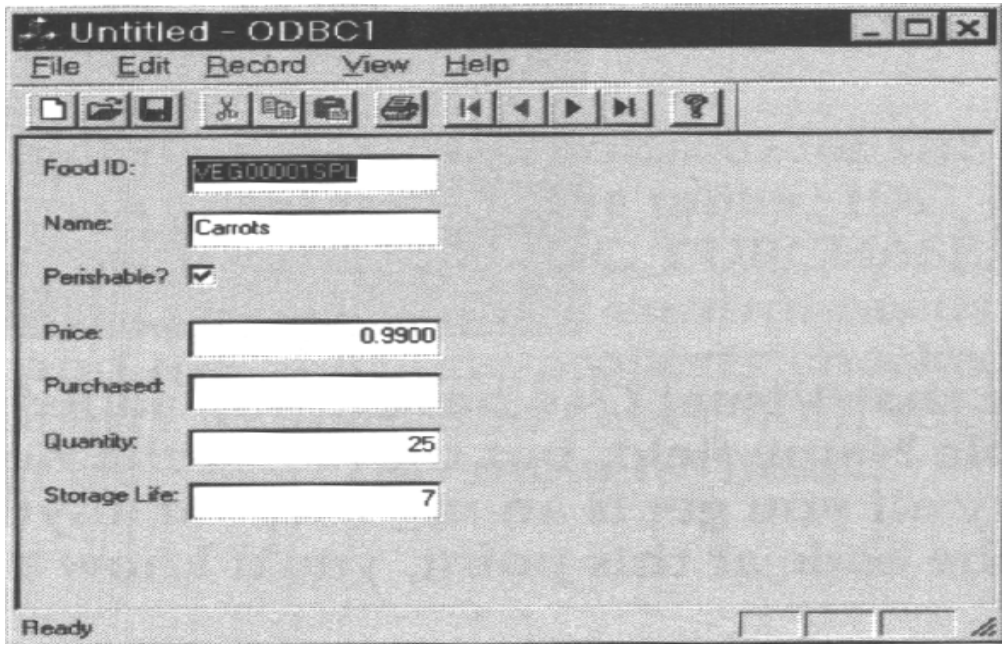
浏览这个列表，你会发现指向已构造查询中每一个字段的指针，Orders（订货）表中的 Purchase（采购）字段除外。Visual C++可以自动处理很多变量类型，

包括大多数的成员。但是，它不能处理时间或日期，所以必须编写一些检索那些值的代码。稍后我会说明如何完成这个任务。现在，就像我们为 IDC_FOOD_ID 控件所做的那样，为剩下的控件创建成员变量。

一旦创建了所有的成员变量，请再复查一下，确保所做的一切都正确无误。用 View|ClassWizard（视图|类向导）命令显示 MFC ClassWizard 对话框。选择 Member Variables（成员变量）选项卡。确保 CODBC1View 类出现在 Class Name（类名）域中（如果不是这样，从下拉列表框中选择 CODBC1View 类）。成员变量列表应该和下图所示的列表相一致。

IDC_FOOD_ID2	CString	->m_Food_ID2
IDC_NAME	CString	->m_Name
IDC_PERISHABLE	BOOL	->m_Perishable
IDC_PRICE	CString	->m_Price
IDC_PURCHASE		
IDC_QUANTITY	long	->m_Quantity
IDC_STORAGE_LIFE	int	->m_Storage_Life

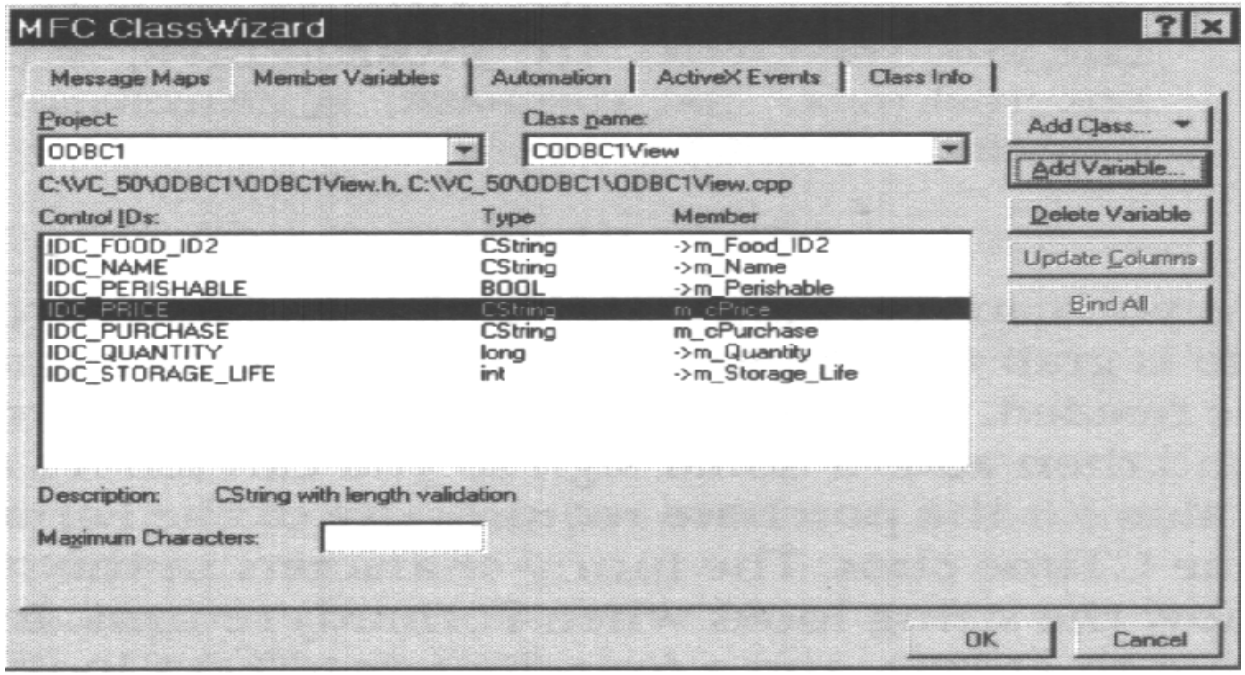
现在编译并运行示例应用程序。刚开始时应该能够看到下图所示的窗体。



运行应用程序后，就请试着在数据库中的记录间移动。你会注意到，这一部分程序运行得很好。程序还允许更改现有记录的值——至少为那些包含了值的字段。但是，你还会注意到一些不足。第一点你已经知道了——我们不在Purchased（采购）字段中显示任何内容。第二个问题是，Price（价格）字段包含四个零位置，而不是通常和货币值有关的两个零位置。我们必须用代码解决这两个问题。

我们在做代码更改之前，要更改有关类成员变量的一些事。用View|ClassWizard（视图|类向导）显示 MFC ClassWizard 对话框。象上一次所做的那样，选择 Member Variables（成员变量）选项卡和 CODBC1View 类。我们需要做的第一件事是，删除当前赋值给 IDC_PRICE 的变量。加亮它并单击

Delete Variable（删除变量）按钮。现在单击 Add Variable（添加变量）按钮，会看到一个 Add Member Variable（添加成员变量）对话框。为成员变量名输入 m_cPrice（Visual C++会自动供给 m_）。单击 OK 添加变量。为 IDC_PURCHASE 做相同的操作，这次用成员变量名 m_cPurchase。MFC ClassWizard 对话框看上去应该如下图所示。



现在我们有了一些变量，可以添加一些代码了。当然，我们想做的第一件事是显示数据库中的信息，这样在后面才能看到变化。程序列表 6.2 显示了需要添加到 CODBC1View 类的 DoDataExchange()函数的代码。要绝对确定，在函数的开始添加新的代码，而不是在函数的末尾添加。否则，会得到奇怪的结果。

程序列表 6.2

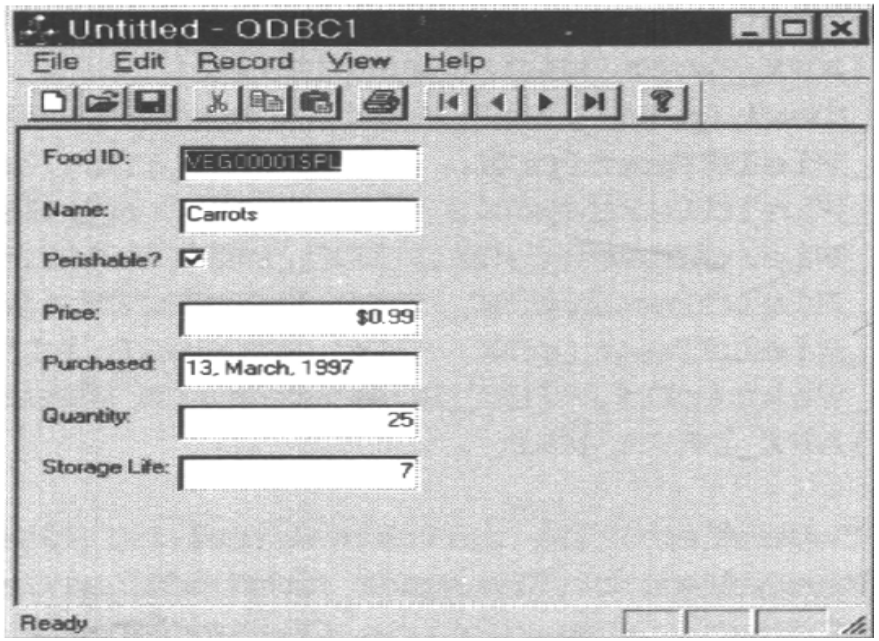
```
void CODBC1View::DoDataExchange(CDataExchange* pDX)
 {
 int                iStrLength;              // Length of the string.

 // Format the price for display by adding a $ and removing the
 // trailing zeroes.
 iStrLength = m_pSet->m_Price.GetLength();
 m_cPrice = "$" + m_pSet->m_Price.Left(iStrLength - 2);
 // Get the date value and convert it to a string.
 m_cPurchase = m_pSet->m_Purchase.Format("%d %B, %Y");

 CRecordView::DoDataExchange(pDX);
 //{{AFX_DATA_MAP(CODBC1View)
 DDX_Text(pDX, TDC_PRICE, m_cPrice);
 DDX_FieldText(pDX, IDC_FOOD_ID2, m_pSet->m_food_ID2, m_pSet);
 DDX_FieldText(pDX, IDC_NAME, m_pSet->m_Name, m_pSet);
 DDX_FieldCheck(pDX, IDC_PERISHABLE, m_pSet->m_Perishable, m_pSet);
 DDX_FieldText(pDX, IDC_QUANTITY, m_pSet->m_Quantity, m_pSet);
 DDX_FieldText(pDX, IDC_STORAGE_LIFE, m_pSet->mStorage_Life, m_pSet);
 DDX_Text(pDX, IDC_PURCHASE, m_cPurchase);
```

```
    //}}AFX_DATA_MAP
}
```

如你所见，解决显示问题相对比较容易。只要人工从 m_pSet 对象抓取值，然后按需要将其格式化。新代码从 Price（价格）字段删除两个额外的零，然后添加一个美元符号，这样就可以告诉用户，它是一个货币数量。获取采购值要求使用 Ctime 类提供的 Format()函数。字符串中有趣的字符("%d %B，%Y")确定 Format()运行时字符串的外观。在这里，我想把日、月当作文本，还有 4 位数字的年。现在我们来看看程序是如何运行的。编译并运行应用程序，它应该如下图所示。



尽管现在可以从数据库读取 Price（价格）和 Purchased（采购）字段了，但

我们还不能更新它们（确保在添加代码以更新字段之前读取字段，这一点很重要；否则，如果应用程序不按预期运行，你就不知道代码在哪里出错了）。这一次，要添加一些代码到 DoDataExchange()函数的末尾，因为我们想在和窗体上的控件交换数据之后再更新数据库。程序列表 6.3 显示了 DoDataExchange()函数的最终形式。

程序列表 6.3

```
void CODBC1View::DoDataExchange(CDataExchange* pDX)
 {
 int                iStrLength;          // Length of the string.
 int                iDay;                // Purchase day.
 int                iMonth;              // Purchase month.
 int                iYear;               // Purchase year.
 CTime              oTime;               // CTime object;

// Format the price for display by adding a $ and removing the
// trailing zeroes.
 iStrLength = m_pSet->m_Price.GetLength();
 m_cPrice = "$" + m_pSet->m_Price.Left(iStrLength - 2);

// Get the date value and convert it to a string.
 m_cPurchase =m_pSet- >m_Purchase.Format("%d %B, %Y");
```

```
CRecordView::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CODBC1View)
DDX_Text(pDX, IDC_PRICE,m_cPrice);
DDX_FieldText(PDx,IDC_FOOD_ID2, m_pSet->m_Food_ID2, m_pSet);
DDX_FieldText(pDX, IDC_NAME, m_pSet->m_Name, m_pSet);
DDX_FieldCheck(pDX, IDC_PERISHABLE, m_pSet->m_Perishable, m_pSet);
DDX_FieldText(pDX, IDC_QUANTITY, m_pSet->m_Quantity, m_pSet);
DDX_FieldText(pDX, IDC_STORAGE_LIFE, m_pSet->mStorage_Life, m_pSet);
DDX_Text(pDX, IDC_PURCHASE, m_cPurchase);
//}}AFX_DATA_MAP

// Transfer the information to the m_Price variable.
// Remember to remove the dollar sign, if necessary, first.
iStrLength = m_cPrice.GetLength();
if (m_cPrice.Find("$") != -1)
{
        m_pSet->m_Price = m_cPrice.Right(iStrLength - 1);
}
else
{
```

```
        m_pSet->m_Price = m_cPrice;
}

// Get the updated year.
iYeat = atoi(m_cPurchase.Right(4));

// Get the updated day.
iDay = atoi(m_cPurchase..Left(m_cPurchase.Find(" ") + 1));

// Get the updated month.
if (m_cPurchase.Find("January") != -1)
        iMonth = 1;
if (m_cPurchase.Find("February") != -1)
        iMonth =2;
if (m_cPurchase.Find("March") != -1)
        iMonth =3;
if (m_cPurchase.Find("April") != -1)
        iMonth =4;
if (m_cPurchase.Find("May") != -1)
        iMonth =5;
if (m_cPurchase.Find("June") != -1)
```

```
        iMonth =6;
    if (m_cPurchase.Find("July") != -1)
        iMonth =7;
    if (m_cPurchase.Find("August") != -1)
        iMonth =8;
    if (m_cPurchase.Find("September") != -1)
        iMonth =9;
    if (m_cPurchase.Find("October") != -1)
        iMonth =10;
    if (m_cPurchase.Find("November") != -1)
        iMonth =11;
    if (m_cPurchase.Find("December") != -1)
        iMonth =12;

  // Transfer the date information to m_Purchase.
  oTime = CTime(iYear iMonth, iDay, 0, 0,0);
  m_pSet->m_Purchase = oTime;
  }
```

上述程序看上去代码可不少，但是你可以相当容易地把它们分成几部分。我们要做的第一件事是添加一些新的变量来处理 Purchase 字段。还有几个变量表示日、月和年。我们还需要一个 Ctime 对象，它允许我们创建一个要放回到

数据库中的日期。

请留意，Price 字段的更新需要进行简单的变换。但是，如果 Price（价格）字段包含了美元符号的话，就不能使用它，所以代码用 IF 语句来执行两个层次的处理：在 Price（价格）字段中，一个有美元符号，而另一个没有。请注意，这是第一次使用 Cstring 对象的 Find()函数来确定是否存在美元符号。你会在代码中相当多地使用 Find()函数，所以在寻找解决问题的办法时记住这个方法是十分方便的。

Purchase 字段在应用程序中是 Cstring 类型，除非我们花一点时间进行分析，否则它不会轻易地转换成 Ctime 对象。我们的代码假定用户总是以这样的格式输入日期：1 December 1998。可惜的是，创建完全成熟的数据库应用程序时，你可能需要考虑比我们的示例程序更多的偶然性。我们故意使示例的代码很简单。用 Cstring 提供的函数和简单地调用 atoi()函数来转换年和日是非常简单的。月份的表达倒是有点挑战性的味道，它需要一个个地实际编写在代码中。一系列的 IF 语句都是为了处理月份。最后，我们采用日期的三个整数表示方法，并把它转换到 Ctime 对象中。但我们有了 Ctime 对象后，数据库更新操作也就只需要一次简单的传输了。

如果现在运行应用程序，可以检索并编辑我们原始查询中的全部字段。正如你所看到的，在 Access 中只需要 4，5 分钟的窗体创建过程，在 Visual C++中却需要花上数小时的时间来实现。在 Visual C++赋予你众多额外灵活性的同时，就要付出编写额外代码的代价，对这一点应该有个清醒的认识。

6.2 操纵数据库的内容

要真正编写数据库应用程序的话，前面的示例确实派不上大用场。甚至，一个实用程序的功能都不只是限于阅读数据库的内容，而这正是目前我们所能全部做到的。例如，我们甚至还没有在数据库中执行过添加新记录或者查找数据的方法。可以说，这两种特征是任何功能数据库应用程序甚至是实用程序的一部分。在本章节中，我们要向示例数据库中添加这两种功能。

向表中添加记录

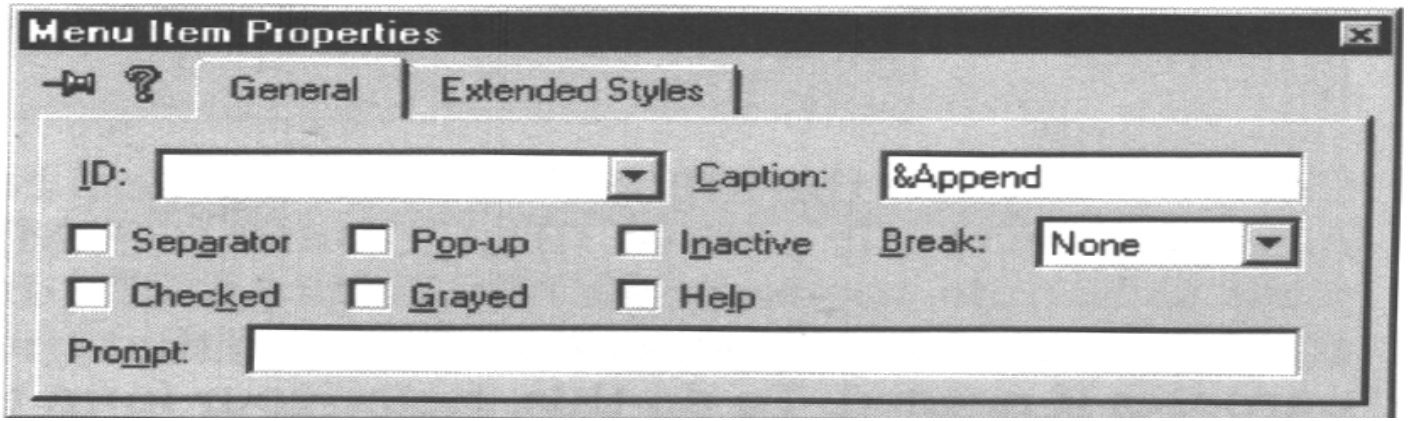
只要在设计阶段做出了正确的选择，那么向表中添加新记录的实际过程就不会困难。在大多数情况下，DBMS 会为你处理一切。你只需要以正确的方式提供正确的信息。例如，你可能想知道，为什么在本章上一节设计的窗体中，我选择使用 Orders 表的 Food_ID 字段，而不是 Foods 表的相应字段。一旦你考虑到了这一点，原因相对就简单了。使用表的 Food_ID 字段允许 DBMS 做出决定。如果在附加记录以后输入一个现有的 Food_ID 值，DBMS 会简单地把新记录添加到 Orders 表中。但是，如果输入一个新的 Food_ID 值，DBMS 就会把新记录添加到 Foods 和 Orders 两个表中。如果使用 Foods 表的 Food_ID 字段，就不可能有这个决定——DBMS 每次都会指示把新记录添加到两个表中。如果用户输入现有 Food_ID 值的话，会引起主键值重复的结果，并最终导致新记录被拒绝。

还有其它要考虑的原则。例如，你的应用程序处理数据有效性了吗？或者DBMS 会为你处理数据有效性吗？在多数情况下，如果处理数据有效性的话，你可以提供一个更为友好的用户界面，但这样做也意味着应用程序会增大。我们的应用程序并没有执行任何数据有效性检查工作，原因有两个：第一，我想让这个示例足够简单，你可以看到所发生的一切。第二，它是一个实用型的程序，即大多数使用这个应用程序的人是像你一样的专业人员。

在向应用程序中添加任何代码之前，必须执行一些设置。要做的第一件事是，添加一个增加记录的菜单项。下面的过程说明了如何完成这项任务（我们已经在第 3 章中学习了添加菜单项的方法，所以这里就不再讨论每一个详细步骤了）。

1. 在 Visual C++中选择 ResourceView（资源视图）。双击 Menu（菜单），然后双击 IDR_MAINFRAME 会看到如下图如示的菜单栏。

2. 单击 Record（记录）菜单，然后单击菜单底部的空白条目。输入&Append（添加），会看到如下图所示的 Menu Item Properties（菜单项属性）对话框。



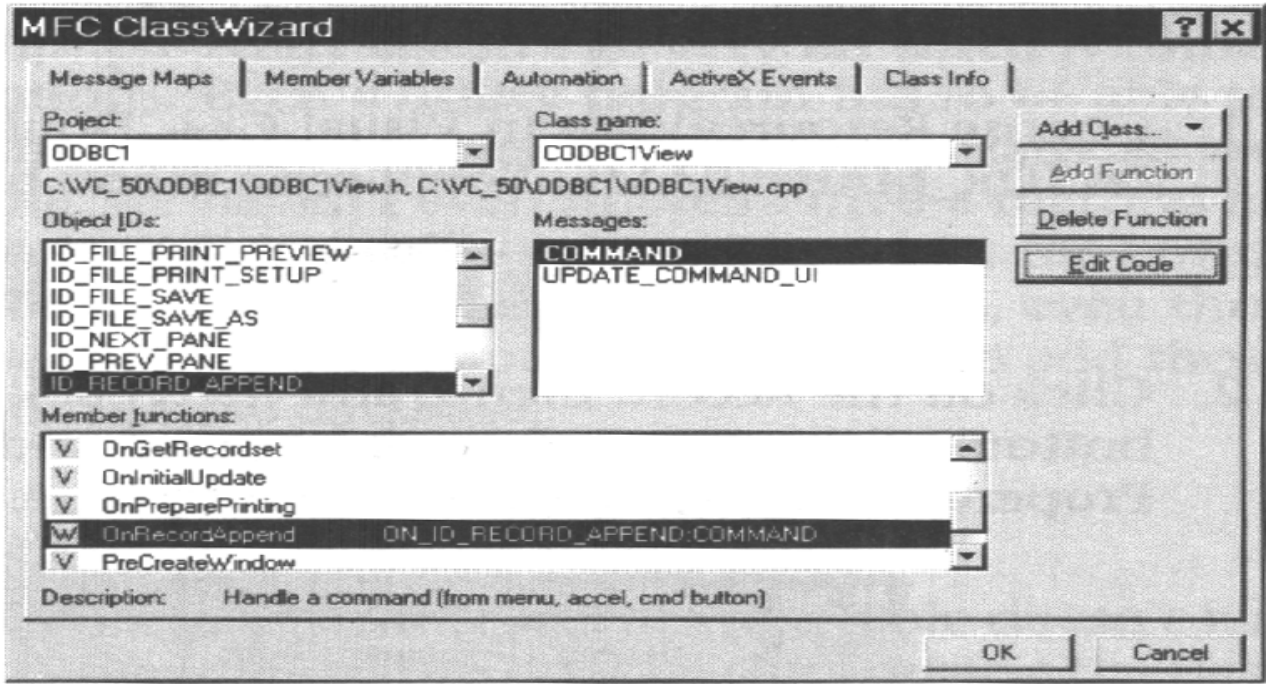
3. 在 Prompt（提示）域中输入 Append a new record to the table.\nAppend Record。

4. 单击 Menu Item Properties（菜单项属性）对话框中的 Close（关闭）框。Visual C++自动把 ID_RECORD_APPEND ID 的值赋给新的菜单项（双击的话，可以看到实际完成的情况）。

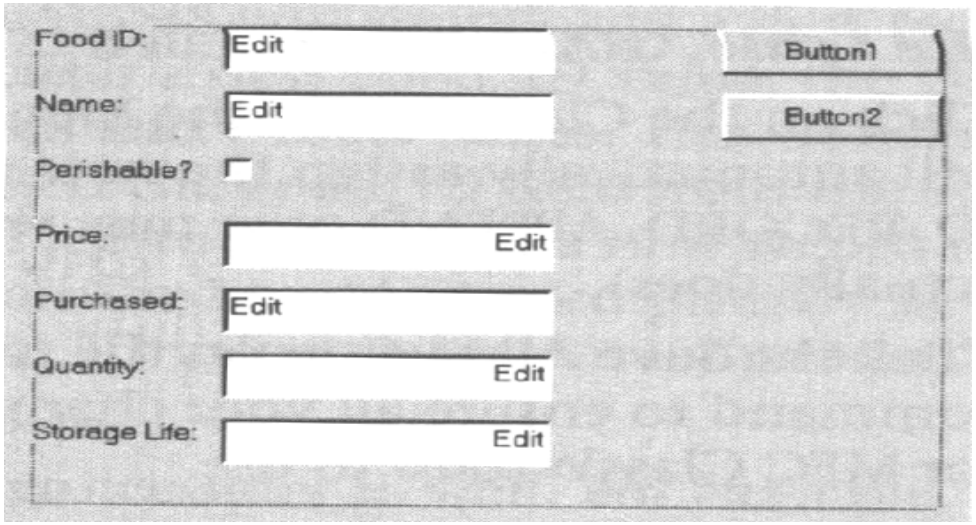
5. 单击工具条上的 Save All（全部保存）按钮或使用 File|Save All（文件|全部保存）命令，确定已保存了全部的更改并进行了正常的登记，以便 MFC ClassWizard 能够使用。

现在我们有了可供使用的菜单项，下面要在 ClassWizard 中添加一些条目，以使其发挥作用。右击菜单条目，从上下文菜单中选择 ClassWizard（类向导）。在 MFC ClassWizard 对话框中，应该看到 Object IDs 列表中已选择了 ID_RECORD_APPEND 条目（否则请选择它）。在 Class Name（类名）下拉列表框中选择 CODBC1View——这个类允许你查看数据库的内容，所以还要在这里为添加记录增加一些代码。加亮 Messages（消息）列表中的 COMMAND（命

令）条目，然后单击 Add Function（添加函数），你会看到 Add Member Function（添加成员函数）对话框。单击 OK 接受缺省的函数名。MFC ClassWizard 对话框看上去如下图所示。



单击 OK 关闭 MFC ClassWizard 对话框。现在我们要修改先前创建的窗体(IDD_ODBC1_FORM)。有几种处理向数据库中添加新记录的方式，但是我已经选择了添加按钮——实际上这是个一般按钮，如果需要的话，也可以把这个按钮用于其它目的。这时，把窗体大小调整为 220×160，然后添加两个按钮。窗体的外观如下图所示。

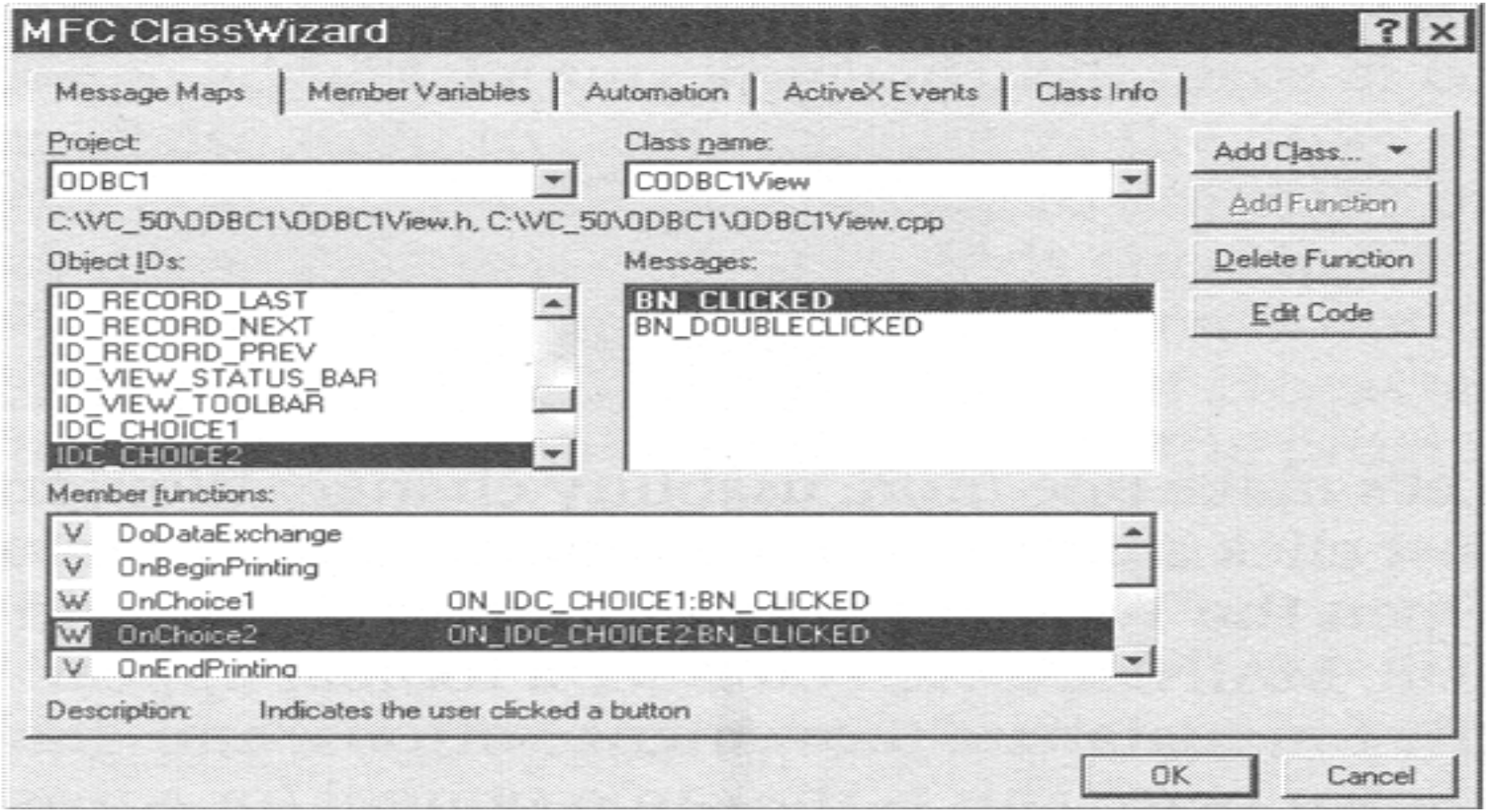


我们始终不想让按钮处于可见状态（因此可访问），所以需要修改几个属性。右击第一个按钮，然后从上下文菜单中选择 Properties（属性），打开 Properties（属性）对话框。清除 Visible（可见）属性的选中标记，把 ID 修改为 IDC_CHOICE1。对第二个按钮执行相同的操作，把它的 ID 修改为 IDC_CHOICE2。

注 如果想在对话框或窗口刚显示时隐藏某个控件，那么请不选中该控件的 Visible 属性。

添加一些函数来处理用户在按钮上的单击操作，这一点也很重要。你只要右击第一个按钮，从上下文菜单中选择 ClassWizard（类向导）选项。选择 Message Maps（消息映射）选项卡。在 Class Name（类名）字段中选择 CODBC1View。在 Object IDs（对象 ID）列表中找到 IDC_CHOICE1 条目并加亮它。现在加亮 Messages（消息）字段中的 BN_CLICKED 条目，然后单击 Add Function（添加

函数）创建处理用户单击操作的函数框架。要为 IDC_CHOICE2 Object ID 条目进行相同的操作。下面是 Member Functions（成员函数）列表的外观。

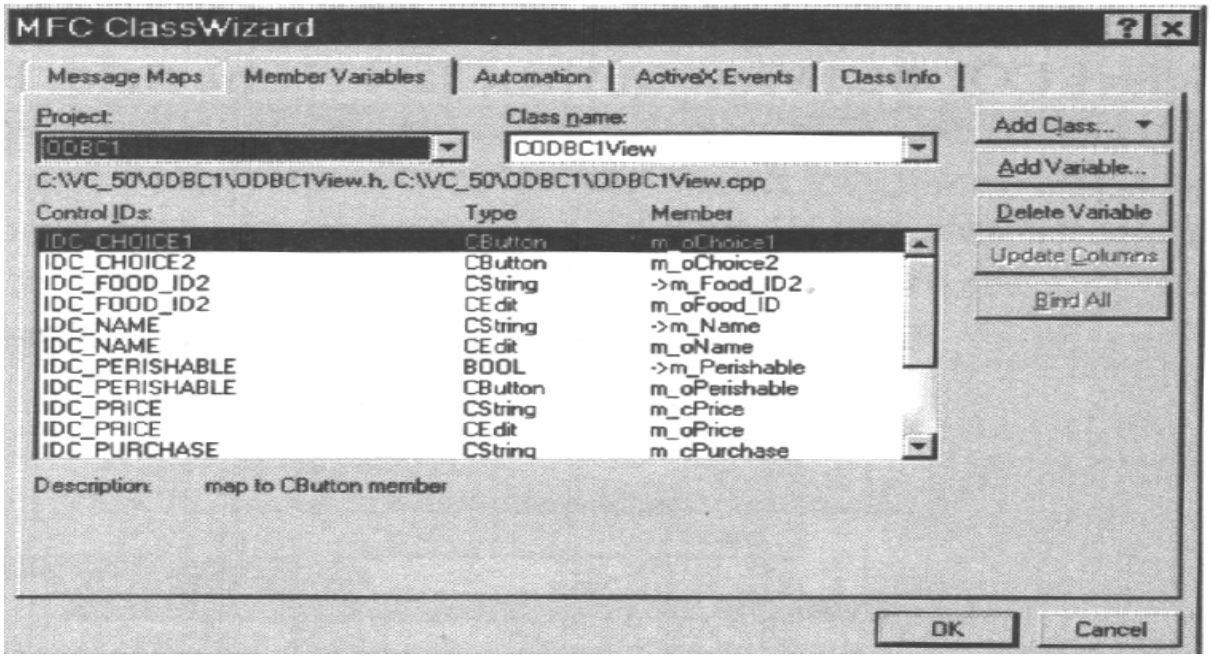


单击 OK 关闭 MFC ClassWizard 对话框。我们以前已经为窗体上的每一个数据项字段都添加了一些成员变量，那些成员变量允许程序和数据库交换数据。但是，现在我们需要直接访问那些数据项字段，所以要再创建一些成员变量。

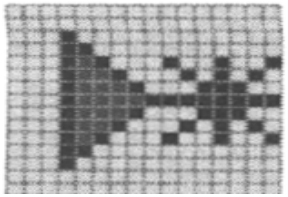
注 Visual C++允许你把多个成员变量赋给单个对象。

按住 CTRL 并双击窗体上的 IDC_FOOD_ID 控件。会看到 Add Member Variable（添加成员变量）对话框。在 Member Variable Name（成员变量名）域中输入 m_oFood_ID，然后在 Category（分类）域中选择 Control（控件）。单击 OK 完成这个过程。刚刚完成的是创建一个成员变量，它可以直接访问窗体上的控件。用下面的成员变量名为其它数据项控件执行相同的任务：IDC_NAME(m_oName)，IDC_PERISHABLE(m_oPerishable) IDC_PRICE(m_oPrice)，IDC_PURCHASE(m_oPurchase)，IDC_QUANTITY(m_oQuantity)和 IDC_STORAGE_LIFE(m_oStorage_Life)。

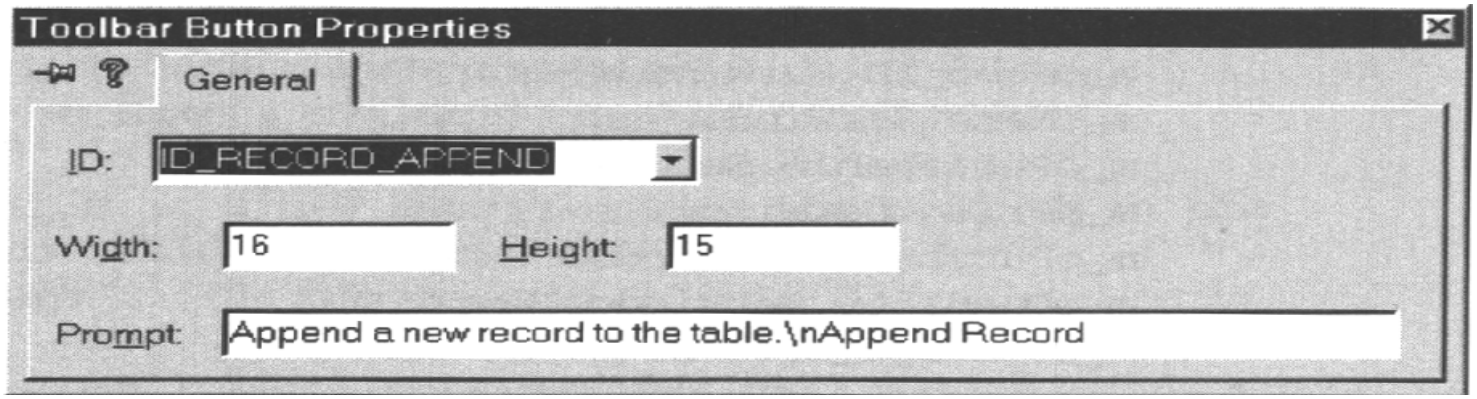
我们来检查一下添加新变量的结果（这个想法不错，因为你要确保可以实际访问它们）。右击任何地方，然后从上下文菜单选择 ClassWizard（类向导）。选择 MFC ClassWizard 对话框的 Member Variables（成员变量）选项卡。应该在窗体上看到每个数据项控件的两个变量，如下图所示。



我们再做一点提供程序可用性的修改。如果每次用户只是单击工具条上的按钮而不使用菜单的话，结果不是更好吗？打开 ResourceView（资源视图）中的 Toolbar（工具条）文件夹，然后双击 IDR_MAINFRAME。你会看到系统显示一个工具条。在四个数据库移动按钮后面添加一个新按钮，并在上面画一个如下图所示的符号。这是和 Access 中 New Record（新记录）按钮图案相同的符号（或者是合理的模仿符号）。



要这个按钮发挥作用并不太困难。双击工具条上新的 Append Record（添加记录）按钮（不是刚才你在绘图区所画的按钮，而是在工具条上的按钮），你就会看到 Toolbar Button Properties（工具条按钮属性）对话框。从 ID 下拉列表框中选择 ID_RECORD_APPEND。这把按钮和菜单命令联系起来。Prompt（提示）字段应该自动以正确的文本值填充，如下图所示。



单击 Close（关闭）框完成本次操作。这些就是把按钮添加到工具条上并使其发挥作用的全部操作步骤。

现在向这一部分示例中添加一些代码。程序列表 6.4 说明了需要添加的代码。一定要把每一段代码都添加到正确的函数中。因为一些操作必须以特定的

顺序进行，所以还要仔细地观察任务完成的顺序。最后，这个代码很明显不做任何类型的边界检查，所以在测试它时必须仔细地输入。Food ID 字段一定要包含足够多的类型正确的字符。如果犯了某个错误而程序不正常恢复的话，请使用 Access 手工编辑表。

程序列表 6.4

```
void CODBC1View::OnRecordAppend()
 {
      // See if the database was opened in read-only mode. If so,
      // display a message, and then exit.
      if (m_pSet->CanAppend() == 0)
           MessageBox("Cannot Append Records",
                "Database Opening Error",
                MB_OK | MB_ICONERROR);
    else
   {
          // Set the option button captions.
          m_oChoice1.SetWindowText("Submit Data");
          m_oChoice2.SetWindowText("Cancel");

         // Display the option buttons.
         m_oChoice1.Showwindow(SW_SHOW);
```

```
                m_oChoice2.Showwindow(SW_SHOW);

            // Create a blank record for the user to edit.
            m_oFood_ID.SetWindowText(" ");
            m_oName.SetWindowText(" ");
            m_oPerishable.SetCheck(0);
            m_oPrice, SetWindowText(" ");
            m_oPurchase.SetWindowText(" ");
            m_oQuantity.SetWindowText(" ");
            m_oStorage_Life.SetWindowText(" ");
        }
 }

 void CODBC1View::OnChoice1()
{
        Cstring         cValue;             // Current value of edit field.
        int             iCheck;             // Current check value of Perishable field.
        int             iStrLength;         // Length of the string.
        int             iQuantity;          // Value of Quantity field;
        int             iStorage;           // Value of Storage field;
        int             iDay;               // Purchase day.
```

```
    int             iMonth;                 // Purchase month.
    int             iYear;              // Purchase year.
    CTime           oTime;              // CTime object;
    BOOL            lNew = TRUE;        // New Food ID?

    // Initialize the search value.
    m_oFood_ID.GetWindowText(cValue);

   // Go to the beginning of the query and search
   // for the Food ID entered by the user.
   m_pSet->MoveFirst();
   while (!m_pSet->IsEOF()   ^ !lNew)
  {
        // Check if the value is equal.
        if (m_pSet->m_Food_ID   == cValue)
              lNew = FALSE;

        // Go to the next record.
        m_pSet->MoveNext();
}
```

```
// Add a new record.
m_pSet->AddNew();

// Save the new data values into the record.
m_oFood_ID.GetWindowText(cValue);
m_pSet->m_Food_ID2 = cValue;
m_oFood_ID.GetWindowText(cValue);
m_pSet->m_Name = cValue;

// If this is a new record, place the Food ID in the
// Foods table field as well.
if (lNew )
{
	m_oFood_ID.GetWindowText(cValue);
	m_pSet->m_Food_ID = cValue;
}

//Convert Perishable control value from int to BOOL.
iCheck = m_oPerishable.GetCheck();
if (iCheck == 0)
	m_pSet->m-Perishable = FALSE;
```

```
  else
        m_pSet->m-Perishable = TRUE;

// Convert and then save the Quantity and Storage fields.
  m_oQuantity.GetWindowText(cValue);
  iQuantity = atoi(cValue);
  m_oStorage_Life.GetWindowText(cValue);
  iStorage = atoi(cValue);
  m_pSet->m_Quantity = iQuantity;
  m_pSet->m_Storage_Life = iStorage;

  // Transfer the information to the m_Price variable.
  // Remember to remove the dollar sign, if necessary, first.
  iStrLength = m_cPrice.GetLength();
  if (m_cPrice.Find("$") != -1)
  {
        m_pSet->m_Price = m_cPrice.Right(iStrLength - 1);
  }
  else
  {
        m_pSet->m_Price = m_cPrice;
```

```
}

// Get the updated year.
iYear = atoi(m_cPurchase.Right(4));

// Get the updated day.
iDay = atoi(m_cPurchase.Left(m_cPurchase.Find(" ") + 1)),
// Get the updated month.
if (m_cPurchase.find("January") != -1)
      iMonth = 1;
if (m_cPurchase.Find("February") != -1)
      iMonth = 2;
if (m_cPurchase.Find("March") != -1)
      iMonth = 3;
if (m_cPurchase.Find("April") != -1)
      iMonth = 4;
if (m_cPurchase.Find("May") != -1)
      iMonth = 5;
if (m_cPurchase.Find("June") != -1)
      iMonth = 6;
if (m_cPurchase.Find("July") != -1)
```

```
        iMonth = 7;
if (m_cPurchase.Find("August") != -1)
        iMonth = 8;
if (m_cPurchase.Find("September") != -1)
        iMonth = 9;
if (m_cPurchase.Find("October") != -1)
        iMonth = 10;
if (m_cPurchase.Find("November") != -1)
        iMonth = 11;
if (m_cPurchase.Find("December") != -1)
        iMonth = 12;

// Transfer the date information to m_Purchase.
oTime = CTime(iYear, iMonth, iDay, 0, 0, 0);
m_pSet->m_Purchase = oTime;

// Once the user completes the record, update the database.
m_pSet->Update();

// Go to the last record so that the user can see the addition.
m_pSet->MoveLast();
```

```
	// Hide the option buttons.
	m_oChoice1.ShowWindow(SW_HIDE);
	m_oChoice2.ShowWindow(SW_HIDE);
}

void CODBC1View::OnChoice2()
{
	int		iStrLength;		// Length of the string.
	char*		cQuantity = "Empty1";	// Text value of Quantity field.
	char*		cStorage = "Empty2";	// Text value of Storage Life field.

	// Hide the option buttons.
	m_oChoice1.showWindow(SW_HIDE);
	m_oChoice2.showWindow(SW_HIDE);

	// Restore the display data.
	m_oFood_ID.SetWindowText(m_pSet->m_Food_ID2);
	m_oName.SetWindowText(m_pSet->m_Name);
	m_oPerishable.SetCheck(m_pSet->m_Perishable);
```

```
    // Process the Quantity and Storage Life fields.
    itoa(m_pSet->m_Quantity,cQuantity, 10);
    itoa(m_pSet->m_Storage_Life, cStorage, 10);
    m_oQuantity.SetWindowText(cQuantity);
    m_oStorage_Life.SetWindowText(cStorage);

    // Format the price for display by adding a $--the trailing
    // zeroes were removed during the data exchange.
    iStrLength = m_pSet->m_Price.GetLength();
    m_oPrice.SetWindowText("$" + m_pSet->m_Price);

    // Get the date value and convert it to a string.
    m_oPurchase.SetWindowText(m_pSet->m_Purchase.Format("%d %B, %Y"));
}
```

看上去上述代码可真不少，但如果把它们分开来看的话，也就不难理解了。我们从 OnRecordAppend()函数开始。应该做的第一件事是，检查数据库是否允许你添加记录。可以用 CRecordset 类的 CanAppend()函数来完成这个工作。如果可以添加记录，这个函数返回 TRUE。如果应用程序不能添加记录，它就显示一条出错消息，并像示例代码中那样退出。此时，你可以以只读方式之外的方式重新打开数据库。

OnRecordAppend()函数的第二部分完成三个任务。第一，我们用与 CWnd

类相关联的 SetWindowsText()给添加到窗体的两个通用按钮赋上一个标题。类总是继承其导出的基类的特征，认识这一点很重要。如果你查阅一下 CButton 类的文档，就会发现不能像 Visual Basic 语言那样设置 Caption（标题）属性，所以完成这项工作需要一个像 SetWindowsText()这样的函数（尽管函数名好像与其功能不相配）。第二，我们用 CWnd 类的 ShowWindow()函数使按钮可见。同样，这一点可能与直觉有些不符，但它确实工作得很好。第三，我们要创建空白编辑框让用户编辑。你还可以提供建议值或者其它适合的值来填充 CEdit 控件。

我们暂时先跳过 OnChoice1()函数（用户单击 Submit（提交）按钮时将调用这个函数），看看如果用户单击 Cancel（取消）（它调用 OnChoice2()函数）时会发生什么。OnChoice2()函数有两部分。第一部分应该很熟悉了。这一次我们不是让 CEdit 控件可见，而是把它隐藏起来。

第二部分检索当前的查询值，并把它放回到窗体上的 CEdit 控件中。对用户来说，就好像是你刚刚恢复了记录，而实际完成的是恢复窗体中的值。请注意，我们必须用四种不同的技术来恢复前面的值。第一种技术是直接数据传输。第二种技术是把数据库提供的数值转换成字符串，然后把字符串放入窗体中。第三种技术重复我们在上一节所做的工作——它只是格式化现有的字符串，所以实际显示成货币值。第四种技术使用 Format()函数把日期转换成字符串。

现在我们来看一看，如果用户填充窗体并单击 Submit Data（提交数据）按钮的话，会发生什么。请记住，为了演示方便，这段代码被简化了——可以添加一些代码来验证每个字段都包含了数据库可能实际使用的值。OnChoice1()函

数需要做的第一件事是，看看我们是否正在向 Foods（食品）表中的现有 Food ID 添加新条目，或者是否我们正在添加全新的 Food ID。这个差别是很重要的，你只会在几个段落中看到。查找这个信息的过程分成如下四个步骤。

1. 获得用户提供的 Food ID 值。在本例的情况下，我们用 GetWindowText()函数来完成。

2. 使用 CRecordset 类的 MoveFirst()函数移动到查询的开始位置。

3. 将查询的 Food ID 字段中的每一个值与用户提供的值进行比较。请注意，我们用 MoveNext()函数在记录间移动。

4. 如果找到了一个匹配值，那么把 1New 设置成 FALSE 并退出搜索循环。

一旦你决定了要添加哪种类型的记录，你就需要使用 AddNew()函数向数据库中添加一个空白记录。此时，这个空白记录只是保存在内存中。如果用户的机器出现电源故障，那么数据库中便不会有空白记录。此时，空白记录只是作为一个模板（如果你喜欢，也可以是空白窗体）来使用，以保存用户提供的数据值。

现在向我们的空白记录中添加数据。请注意，从窗体获取数据的过程不像最初想像得那样简单。最简单的传输过程是从 CEdit 控件到查询中的字符串字段之间的传输。这个过程分成两个步骤。要做的第一件事是，用 GetWindowText()函数把 CEdit 控件中当前的文本放入一个临时变量。填充了临时变量后，就可以把信息传输到数据库字段中了。

复选框很有意思。Visual C++支持的复选框可以包含三个值中的一个：选中，未选中或者不确定。结果，CButton 类的 GetCheck()函数返回一个整数，而不是

Boolean（布尔）值。你需要把这个整数值转换成 Access 使用的 Boolean（布尔）值。

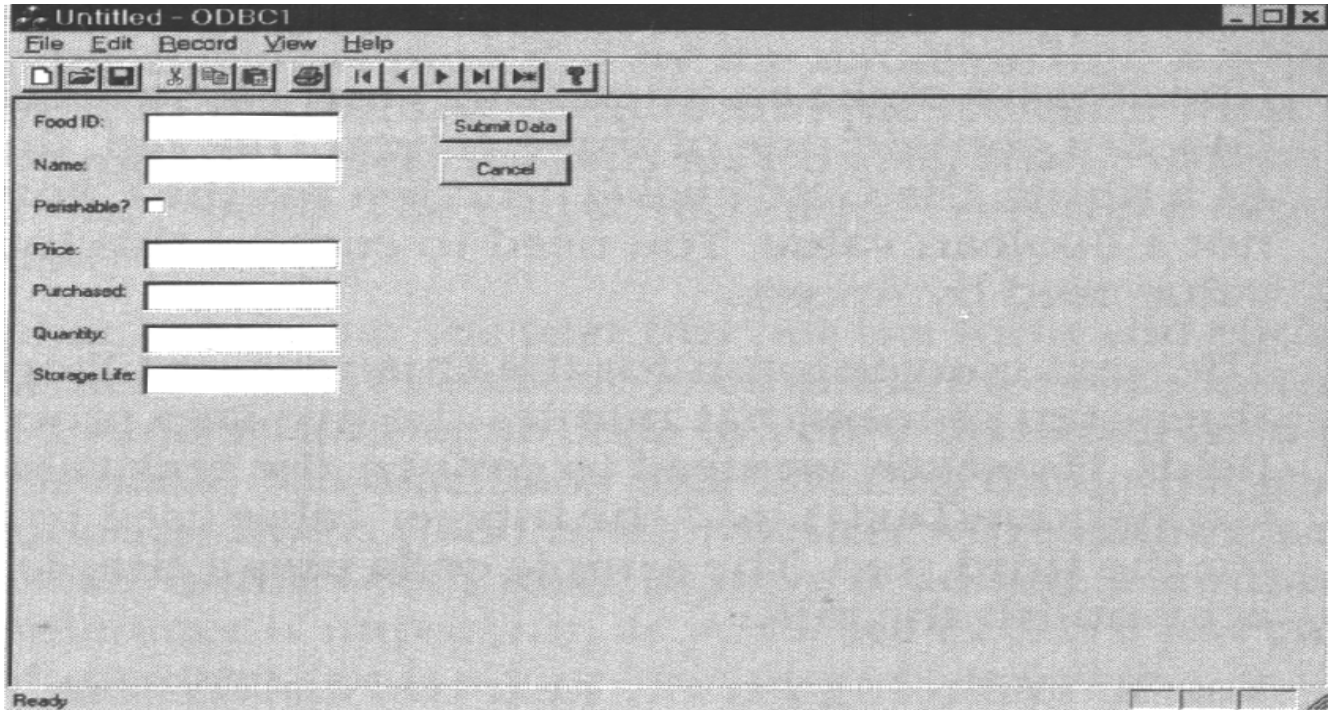
下一个转换发生在 Quantity 和 Storage Life 字段中。这个过程分成三个步骤，模仿了我们为文本字段使用的二步骤过程。但是，我们要把用 GetWindowText()获得的文本值转换成字段使用的整数值——这算是第三步。示例代码用简单的 atoi()函数来完成这项任务。

你应该见过下面的两个转换。它们都以相对未变的形式出现在 DoDataExchange()函数中。你仍然需要得到窗体上控件提供的 Purchased（购买）字段的文本值，并把它转换成日期。这一点没有任何变化。同样地，要从 Price（价格）字段删除美元符号。否则，Access 不会接受字段值。

现在我们有了填满了新数据值的空白窗体，可以用 Update()函数来更新数据库了。这是数据库唯一真正脆弱的时刻。如果在更新的很短暂的时间内出现了电源故障，可能会得到不可预知的后果。必须说，虽然这种方法不安全，但它确实要比长时间打开记录的老技术的数据库好。

OnChoice1()函数与数据库相关的最后一个动作是移动到最后一个条记录。你必须完成这个操作，这样用户才能实际看到新的记录。否则，用户可能会认为记录没有放到数据库中，并试图再添加一个。请注意，OnChoice1()函数以隐藏两个通用按钮来结束这个函数。除非用户实际上有新的记录要编辑，否则他们不能访问这些按钮，这一点很重要。所以，要尽可能地把它们隐藏起来。

那么，空白记录看上去是什么样子呢？下图是我们的示例程序显示的空白记录。

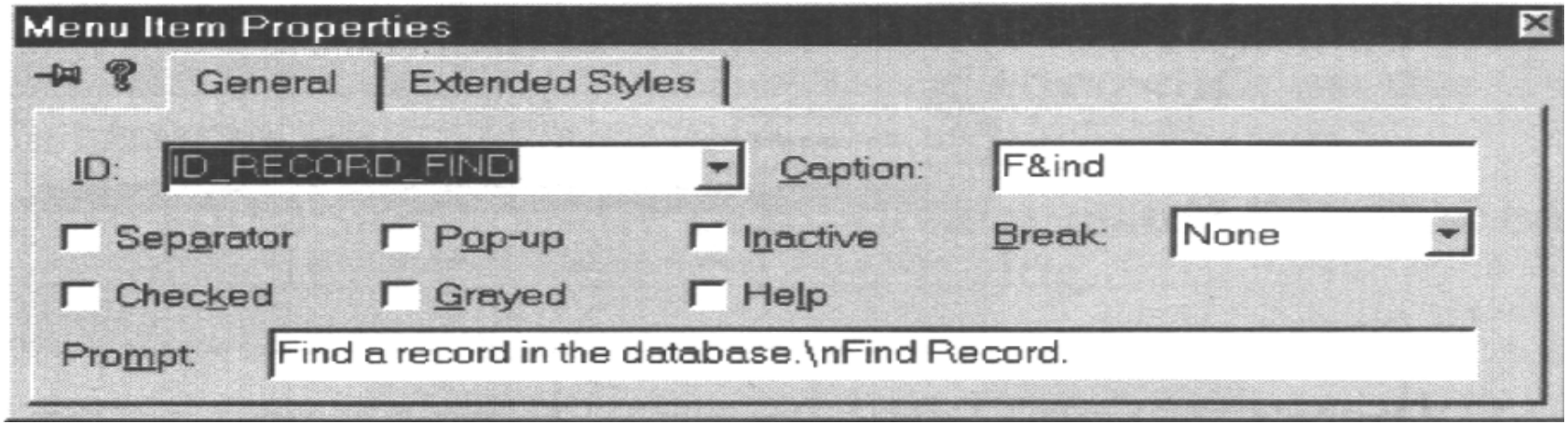


查找数据库中的数据

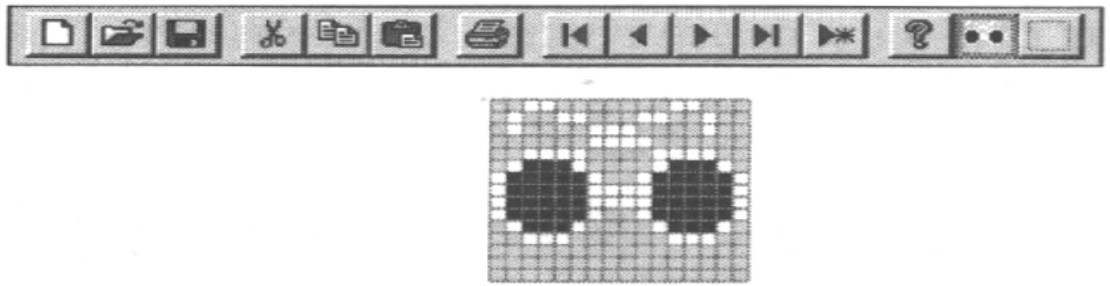
不管应用程序看上去有多好，如果用户找不到他们所需要的东西，那么它也不会有太大的价值。构造出使用户在数据库中移动的最佳方式一直是众说纷纭的主题。有一些书，它们整章的主题都是关于以尽可能有效的方式查找所需要的东西。我们应该花很多的时间来学习搜索技术，但最后也并不会为你提供太多的帮助。使用 ODBC 时，有一种简单的查询方式；它也许不是最快或者最有效，但肯定能找到你所需要的东西。我们要在本节中集中讨论这项技术。

很明显，在我们进一步深入探讨之前，还要再在菜单和工具条上花点时间。

再次打开 IDR_MAINFRAME 菜单，这一次我们准备在 Record（记录）菜单下添加 Find(查找)菜单项。选择最后的空白并输入 F&ind(注意，已经为 First Record（第一个记录）选项在 F 下加了下划线）。还要在 Prompt（提示）域中输入下述提示：Find a record in the database.\nFind Record。此时的 Menu Item Properties（菜单项属性）对话框应该如下图所示。

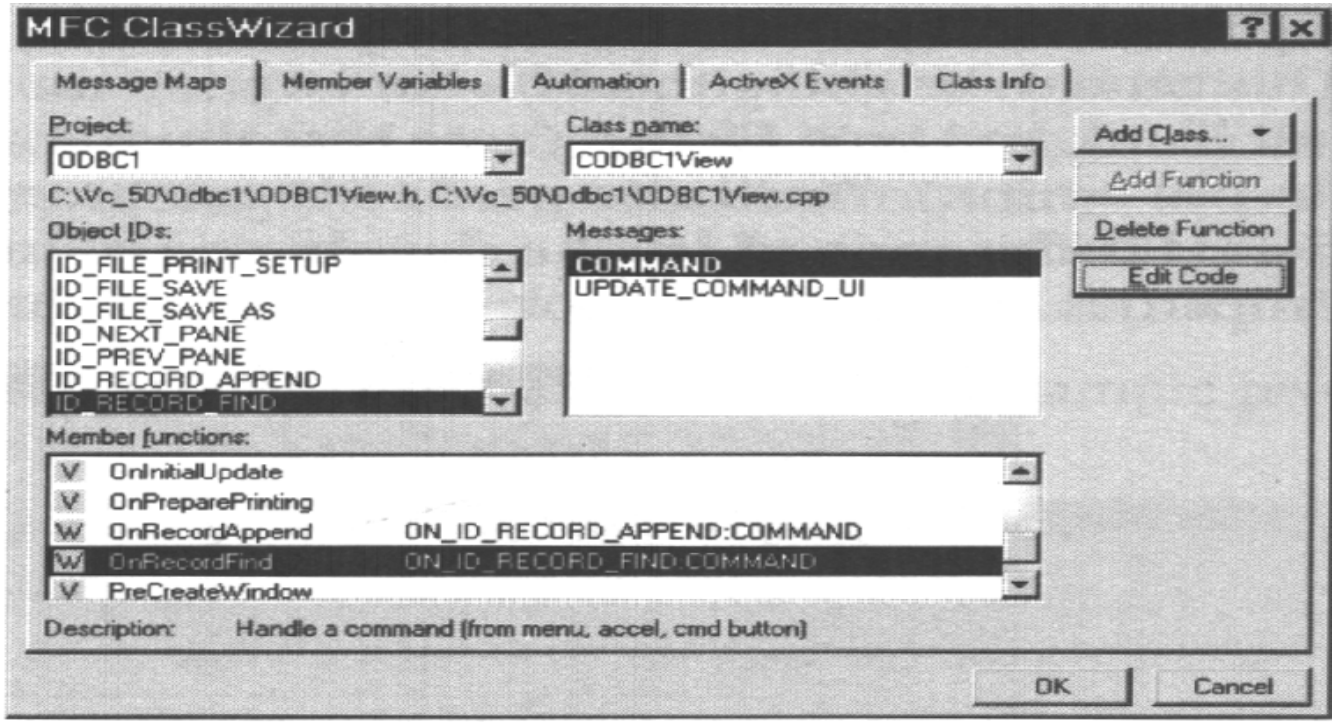


单击 Close（关闭）框完成新菜单项的创建工作。打开 IDR_MAINFRAME 工具条。下面是我添加的新的 Find（查找）按钮，你当然也可以使用任何一种想要的按钮（用不着奇怪——它确实是一副眼镜）。

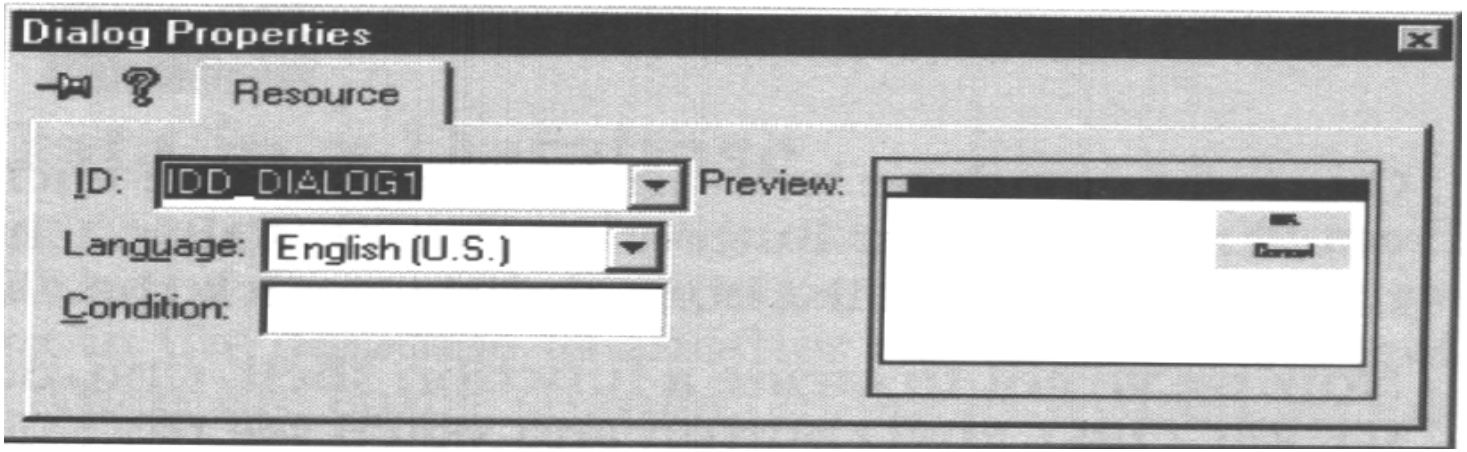


制作了按钮后，双击工具条上该按钮，就会看到一个 Toolbar Button Properties（工具条按钮属性）对话框。从 ID 下拉列表框中选择 ID_RECORD_FIND，然后单击 Close（关闭）按钮。

现在，我们已经创建了一个函数框架。按住 CTRL 并双击 ID_RECORD_FIND 按钮，会看到 MFC ClassWizard。应该选择 ID_RECORD_FIND 条目。一定要在 Class Name 字段中选择 CODBC1View。加亮 Messages（消息）列表中的 COMMAND（命令）条目，然后单击 Add Function（添加函数）。单击 OK 接受 Add Member Function（添加成员函数）对话框中的缺省函数名 OnRecordFind（我们要在本节的后面向这个函数中添加代码）。此时，MFC ClassWizard 对话框看上去应该如下图所示。



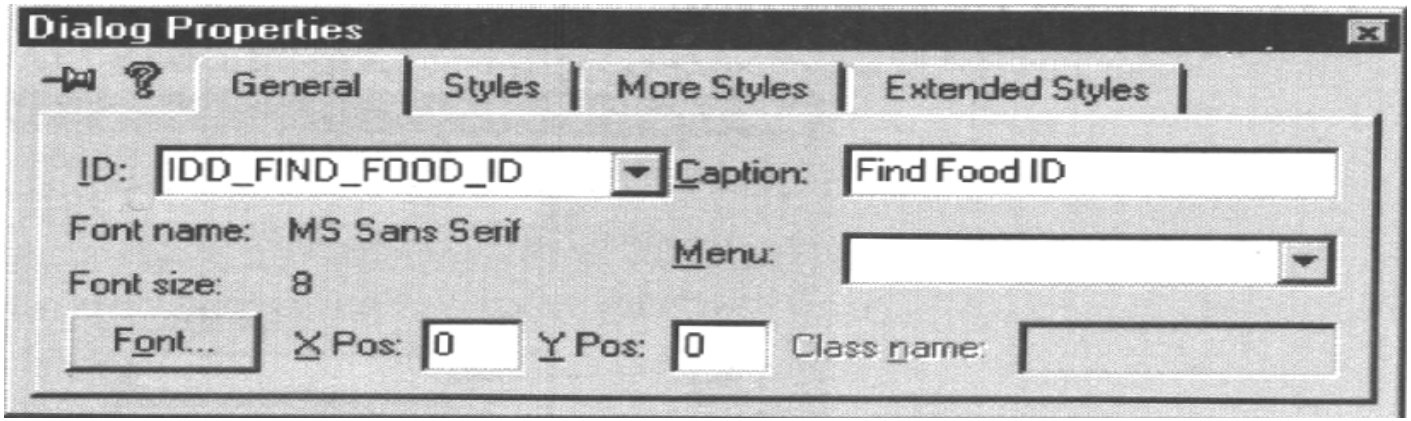
还遗漏了一件事。你打算如何询问用户要查找什么呢？完成这个任务需要一个比较小的对话框。单击 OK 关闭 MFC ClassWizard 对话框。回到 ResourceView（资源视图）。右击 Dialog（对话框）文件夹，从上下文菜单中选择 Insert Dialog（插入对话框）。Visual C++会生成一个新的对话框。在我们继续操作之前，可以更改该对话框的一些属性。原因只有一个，为对话框赋予更容易辩认的名字。右击 ResourceView（资源视图）中的 IDD_DIALOG1 条目，从上下文菜单中选择 Properties（属性），会看到如下图所示的 Dialog Properties（对话框属性）对话框。



在 ID 字段中输入 **IDD_FIND_FOOD_ID**。单击 Close（关闭）框完成对话框属性的更改。

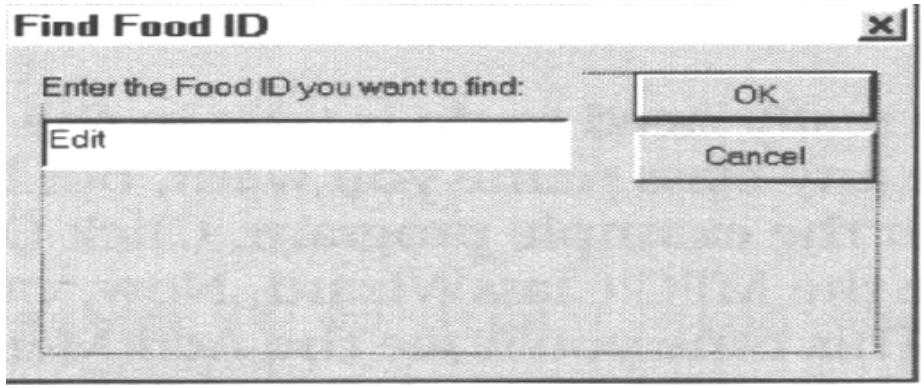
对对话框本身我们只需要做两件事。要做的第一件事是，给对话框说明一个意义明确的标题。右击对话框本身，从上下文菜单中选择 Properties（属性）。这一次会看到如下图所示的 Dialog Properties（对话框属性）对话框（尽管这个

对话框的名称和前面看到的相同，但是它的目的不同）。



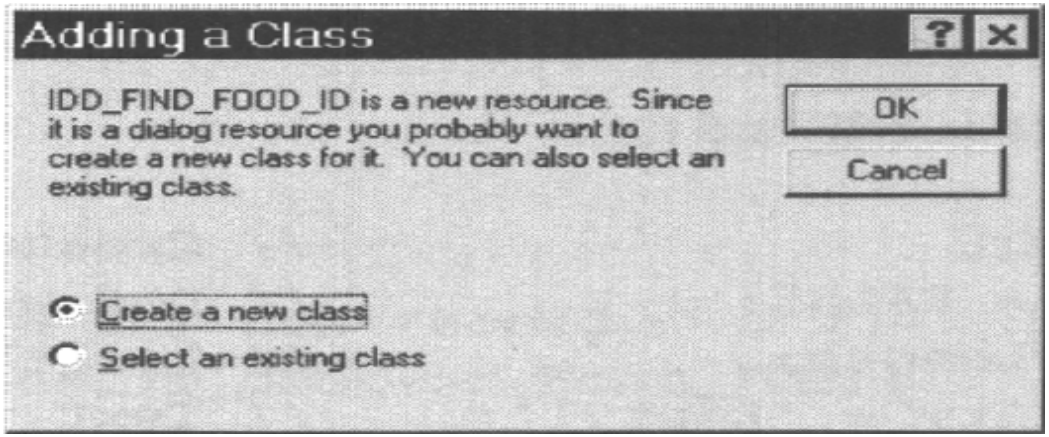
在 Caption（标题）字段中输入 Find Food ID（查找食品 ID），然后单击 Close（关闭）框完成此次更改。

现在我们要添加单行 Static Text（静态文本）控件和相关的 Edit Box（编辑框）控件来完成这个对话框。下图是最后的对话框外观。

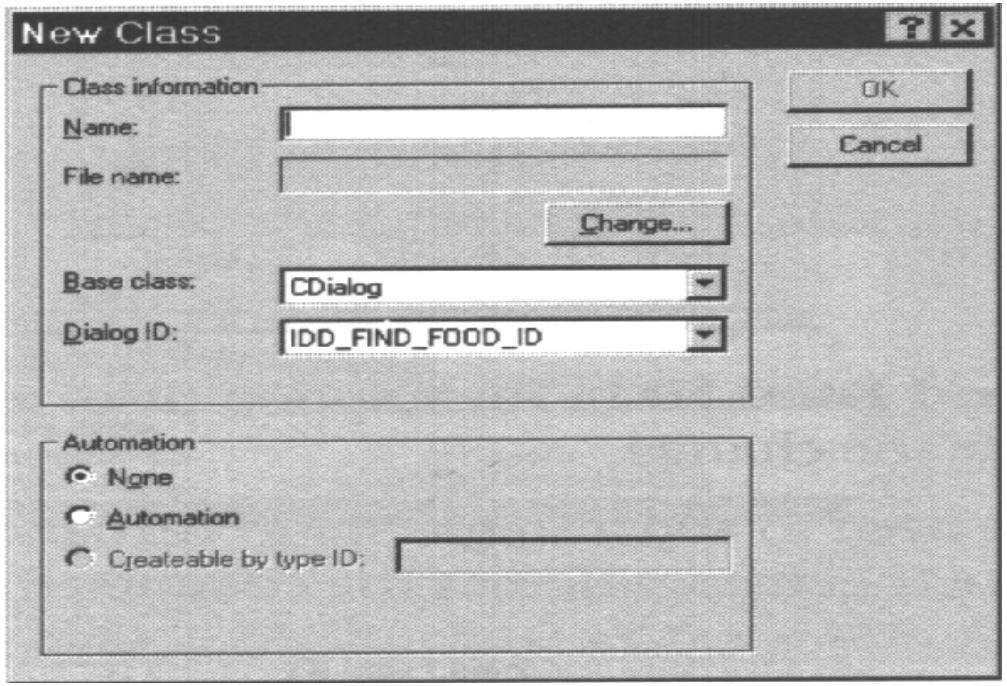


对这个对话框所做的最后一件工作是，设置 Edit Box（编辑框）控件。右击 Edit Box（编辑框）控件，从上下文菜单中选择 Properties（属性）。把 Edit Properties

（编辑属性）对话框中的 ID 属性改成 IDC_FIND_FOOD_ID。单击 Close（关闭）框完成这个动作。现在按住 CTRL 并双击 Edit Box（编辑框）控件。你首先看到的是如下图所示的 Adding a Class（添加类）对话框，它使人感到有些迷惑。



出现 Adding a Class（添加类）对话框的原因是，程序中的每一个对话框都属于某一个类。在很少情况下，你会想把对话框赋给一个现有类。就像应用程序中的 About（关于）框，每一个对话框都应该有其自己的类。单击 OK 为 Find Food ID（查找食品 ID）对话框创建新类。Visual C++显示如下图所示的 New Class（新类）对话框。



在这个对话框中需要完成的工作只是在 Name（名字）域中输入类名。可以使用任何想要的类名，但是我已经为示例程序中的类名选择了 FindFoodID。单击 OK 完成新类的创建。单击 OK 选择 MFC ClassWizard。现在，再次按住 CTRL 并双击 Edit Box（编辑框）控件，这次会看到一开始就希望看到的 Add Member Variable（添加成员变量）对话框。在 Member Variable Name（成员变量名）域中输入 m_FindFoodID。确保 Category（分类）域包含 Value（值），Variable Type（变量类型）域包含 CString。单击 OK 添加成员变量。

还记得我们以前在 CODBC1View 类中创建的 OnRecordFind()函数框架吗？现在要添加一些代码了。程序列表 6.5 显示了用来查找数据库中记录的代码。

程序列表 6.5

```
void CODBC1View::OnRecordFind()
{
 FindFoodID          oFindIt;                // Create an instance of our dialog box.
 BOOL                lFound = FALSE;         // Did we find a match record?
 int                 iStrLength;             // Length of the string.
 char*               cQuantity = "Empty1";   // Text value of Quantity field.
 char*               cStorage = "Empty2";    // Text value of Storage Life field.
 CDBVariant          varBookmark;            // Positioning bookmark.

 // Display the dialog and determine which button the user pressed to exit.
 if (oFindIt.DoModal() == IDOK)
  {
      // Save the current position.
      if (m_pSet->GetBookmark())
            m_pSet->CanBookmark(varBookmark);

      // Go to the beginning of the query and search
      // for the Food ID entered by the user.
        m_pSet->MoveFirst();
        while (!m_pSet->IsEOF() ^ lFound)
```

```
{
        // Check if the value is equal.
        if (m_pSet->m_Food_ID ==oFindIt.m_FindFoodID)
            lFound = TRUE;

        // Go to the next record.
        else
             m_pSet->MoveNext();
}

if (!lFound)
{
      // Display an error message if we didn't find the record.
      MessageBox("Record not found!",
          "Database Error",
          MB_OK | MB_ICONERROR);

      if    (m_pSet->CanBookmark())
            // Restore the current position if
            // the database supports bookmarks.
             m_pSet->SetBookmark(varBookmark);
```

```
    else
            // Otherwise, move to the first record.
              m_pSet->MoveFirst();
  }
  else
{
        // Display the data in the new record.
        m_oFood_ID.SetWindowText(m_pSet->m_Food_ID2);
        m_oName.SetWindowText(m_pSet->m_Nane);
        m_oPerishable.SetCheck(m_pSet->m_Perishable);

        // Process the Quantity and Storage Life fields.
          itoa(m_pSet->m_Quantity,cQuantity,10);
          itoa(m_pSet->m_Storage_Life, cStorage, 10);
          m_oQuantity.SetWindowText(cQuantity);
          m_oStorage_Life.SetWindowText(cStorage);

        // Format the price for display by adding a $--the trailing
        // zeroes were removed during the data exchange.
        iStrLength = m_pSet->m_Price.GetLength();
        m_oPrice.SetWindowText("$" + m_pSet->m_Price);
```

```
            // Get the date value and convert it to a string.
            m_oPurchase.SetWindowText(m_pSet->m_Purchase,Format("%d %B, %Y"));
        }
    }
}
```

这段程序代码中的一部分看上去与本示例程序中其它部分的代码很相似。该代码引进了一两个新的函数。应该注意的第一件事是，要检查当前的数据库位置。某些类型的 DBMS（Access 不属于这一类）支持书签。它们的作用就像是看书时用来记住阅读位置的书签一样。不过，必须做的第一件事是，用 CanBookmark()函数询问 DBMS 实际上是否支持书签。如果它支持书签，就可以用 GetBookmark()函数来保存当前的位置，用 SetBookmark()函数来恢复它。

请注意，我们的搜索例程非常简单。如果找到了用户在对话框中请求的记录，就把 Boolean（布尔）变量设成 TRUE。否则，翻到下一条记录。继续这一查找过程，直至找到与用户查找标准相匹配的第一个记录，或者是已经查询到了最后一条记录。如果你的程序采用了相似的搜索例程，那么这两种情况一定都要测试到。

注 完成一次搜索后总要复位记录指针，这样查询才能在搜索例程结束时停留在某个已知的记录位置。

此时，程序要处理两种情况。如果没有找到所要检索的记录，程序显示一个错误消息对话框。如果 DBMS 支持书签的话，程序还试图返回到前一条记录。

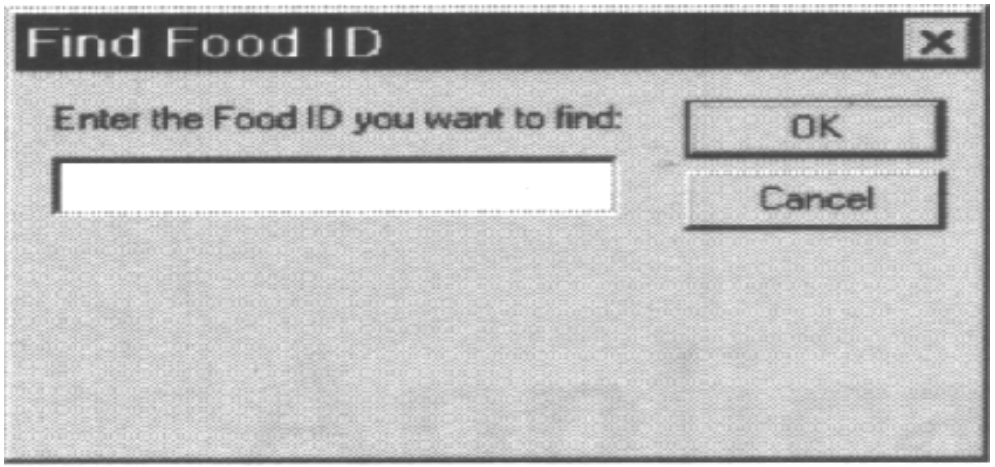
否则，程序将返回查询中的第一个记录。设置记录指针的位置是极其重要的；否则，DBMS 可能会出现异常。如果找到了记录，那么我们就处理另一种情况，在数据输入窗体中显示信息。这由本节剩余的代码来完成。你应该注意到，这些代码看上去与我们在 OnChoice2()函数末尾使用的代码十分相似。

在实际编译并运行这个应用程序之前，还要再编写一段编码。在 ODBCView1.CPP 文件的顶部，你会看到很多#include 指令。如果想使用 Find Food ID（查找食品 ID）对话框，就要在模块中包括它的头文件。添加一条如下所示的新#include 指令：

```
// Added for Find Food ID dialog support.
#include "FindFoodID.h"
```

注： 在模块中用#include 指令包括你想要使用的对话框的头文件。

现在把所有的代码段都集中在一起，编译并运行程序。你会发现，在我们的小型表上，搜索例程运行得非常快，而当表变大时，它的速度就慢了下来。在大多数情况下，如果你计划处理巨大的数据库，就要购买一本大部头的包含了几千个搜索例程的书。当表包含了多达 10,000 条记录时，这个搜索例程也应该运行良好。当然，所有这些都依赖于表之间关系的复杂性以及它们所包含的字段数量。下面是搜索过程中 Find FOOD ID 对话框的外观。


Find Food ID
Enter the Food ID you want to find:
OK
Cancel

第 7 章　使用 OLE-DB 或 ADO 建立数据库应用程序

OLE-DB（对象链接与嵌入数据库）和 ADO（ActiveX 数据对象）是对 Microsoft 数据库访问技术的最新补充。这两种技术为访问所有类型的数据库提供了统一的接口，不论它出现在驱动器上的什么地方。另外，它们提供了 Microsoft 数据访问技术的最大灵活性。

阅读本章时要记住的一点是，OLE-DB 是一种基本的数据访问技术，而 ADO 是在 OLE-DB 上添加了一层便于使用的包装。另外，OLE-DB 这种产品与 ODBC 是完全脱离的。正如我们在第 6 章中所做的那样，用 DAO 创建应用程序和用 ODBC 创建应用程序有很多类似之处，用 OLE-DB 创建应用程序则要求完全不同的技术。事实上，可以把 DAO 看作是 ODBC 的高级（功能更强大）形式，只不过灵活性稍差一些（可访问的数据库引擎要少一些）。另一方面，本章会说明，OLE-DB 代表了一种全新的数据库访问方式。

在开始本章之前，假定你已经花时间阅读了第 4 章，了解了 OLE-DB 和 ODBC 之间的差别。你还要知道 ADO 是干什么的。如果你还没有阅读第 4 章，最好请补读。你还要创建我在第 5 章中已经说明如何创建的示例数据库，我们要在本章中用到该数据库。

Web 链接 在研究 Visual C++和各种支持 DAO，ADO 以及 OLE-DB 的数据库访问技术时，你一定想知道到哪里可以得到其他人的帮助。例如，microsoft.public.vc.database 新闻组中有很多关于这些技术的讨论。你还会在 microsoft.public.officedev，microsoft.public.vc.mfc 和 microsoft.public.vc.mfcdatabase 新闻组中发现一些与数据库技术相关的讨论。还有一些和非 Microsoft 的数据库技术相关的新闻组：comp.os.ms-windows.programmer.win32，comp.os.ms-windows.programmer.tools.mfc 和 comp.databases.ms-access。有趣的是，尽管在 Internet 上关于 ADO，DAO 和 OLE-DB 的讨论好像占据了很多的新闻组讨论，但是很少有专门谈论这些技术的新闻组，特别是关于 ODBC 的新闻组就更少了。显然，这就是说，为了在更为一般的新闻组（像我刚刚提到的新闻组）中寻找特定的信息，你还要花费一些额外的时间。

我们准备在本章中花很多的时间来讨论 OLE-DB 和 ADO。事实上，本章开始的方式和第 6 章很相似——建立一个简单的应用程序。但是，在这里我们会创建一个网格视图，而不是前一章中所用的窗体视图。知道如何创建多种形式的简单应用程序相当重要。否则的话，也许你就不能给客户提供正常管理其数据时所需的全部视图。

注释 因为 ADO 和 OLE-DB 在很大程度上代替了 DAO，所以本章不研究 DAO，不过，你还是可以从 Osborne Web 站点下载本章前两个示例的 DAO 版本。只要寻找 Web 站点上的 DAO1 和 DAO2 示例就可以了。我已经检查了代码，确保它可以在 Visual C++ 6.0 版上运行。你只要看一看代码本身，就会知道如何实现网格数据库实用程序的 DAO 版了。代码中的注释将有助于你对 MFC AppWizard 生成的应用程序进行必要的修改。为了使这些示例程序能够运行，必须在机器上安装 DAO。这些示例使用了 DAO 3.5 版。有几种方式可以用来检查 DAO 版本。首先，查看 Program Files\Common Files\Microsoft Shared\DAO 文件夹。如果看到了 DAO350.DLL 文件，就表明已经安装了 3.5 版 DAO。还可以查看 Control Panel（控制面板）中的 Add/Remove Programs（添加/删除程序）小程序。如果安装了 DAO 3.5 版，那么 Install/Uninstall（安装/卸载）页应该有一个 Data Access Objects(DAO) 3.5 列表项。

和上一章一样，我们要建立示例应用程序并增强其功能。在这里，我们要向示例应用程序添加报告能力。我准备说明一些 OLE-DB 专用的方式。你会发现，上一章必须使用的大量代码，比如用来搜索数据库的代码，至少有一部分现在被 OLE-DB 或 ADO 类支持的简单函数调用所代替了。

技巧 与 ODBC 不同，使用 OLE-DB 或 ADO 时不需要使用 DSN。但是，这两种技术都必须知道本地驱动器或网络上数据的特定位置。你可以用 UNC（统一命名协议）路径来减少用户找不到数据库的机会，但这种机会依然是存在的。一定要在每台机器上检查你的进行数据访问的 OLE-DB 或 ADO 应用程序。不要简单地因为你可以用 Explorer 看到相关的数据库，就假定用户可以访问这些数据了。

7.1 创建简单的网格视图应用程序

我们在第 5 章看到了两种不同的应用程序。第 6 章说明了如何创建其中的第一种，即窗体视图应用程序。在本节中，我们来看看其中的第二种，即网格视图应用程序。与第 6 章中一次只浏览查询中的一条记录不同，这回我们要一次看到所有的记录（至少要像屏幕所显示的那样多）。当然，这就要使用网格而不是用编辑框来存储各种记录值了。

你多半会发现，在谈到数据库时，网格给程序员带来了一些特殊的挑战。例如，如果你现在不能马上阅读整个查询的话，就必须跟踪查询已显示了多少记录（有一些查询可能非常巨大，内存不允许创建可以保存所有需要向用户显示的值的网格控件）。你可能还会发现，要管理所有的字段，尤其是像备忘录这样的长文本字段，还需要再做一些工作。冗长或复杂的字段可能要求一些条件或要求使用第二个对话框（与备忘录字段共同出现）。为了使这个示例更简单一些，我们要走在这些问题的前面。但是，如果数据库表包含了带这些类型

值的字段，那么记着专门考虑这些字段的处理方式是非常重要的（和第 6 章一样，我们要研究一些特殊的和日期及货币有关的格式化问题）。

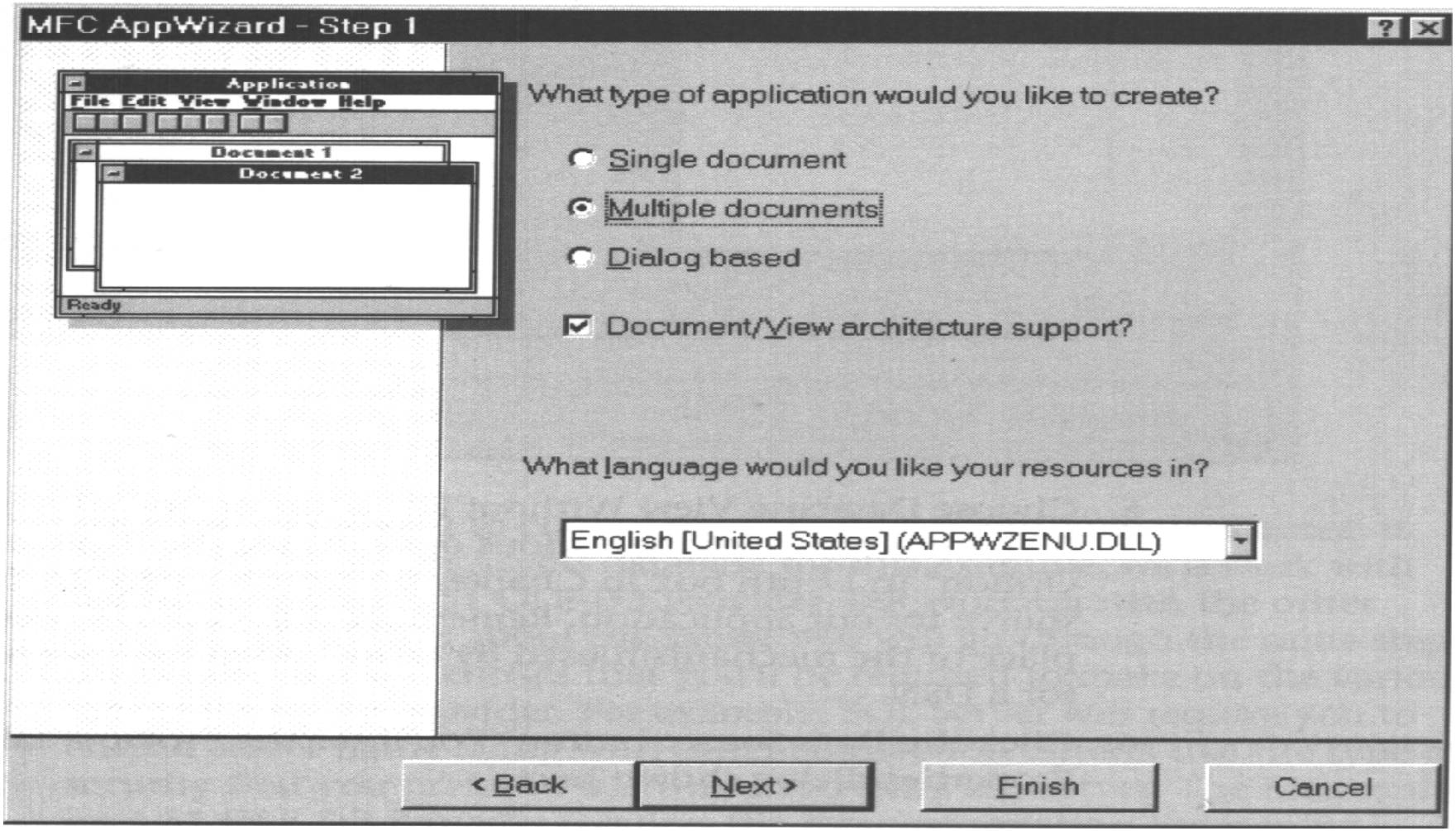
Web 链接 你可能会发现，一开始阅读本章就想进行一些额外的训练。Internet 上有很多地方提供关于用 Visual C++处理 ADO，OLE-DB，DAO 和 ODBC 的课程。例如，你会发现，http://www.unisoftinc.com/courses/ 处的 Universal Software Solutions（通用软件解决方案）提供了关于使用 Visual C++、关系数据库和更新的 Microsoft 对象技术如 DCOM 和 COM+的高级课程。另外还有 http://www.develop.com/处的 DevelopMentor（开发顾问）。它大体提供了各种有关对象技术和 Visual C++的课程（包括各种形式的数据库访问）。该公司过去常常提供一些 DAO 专用的类，但后来停止了这类业务。

创建 ADO1 工程

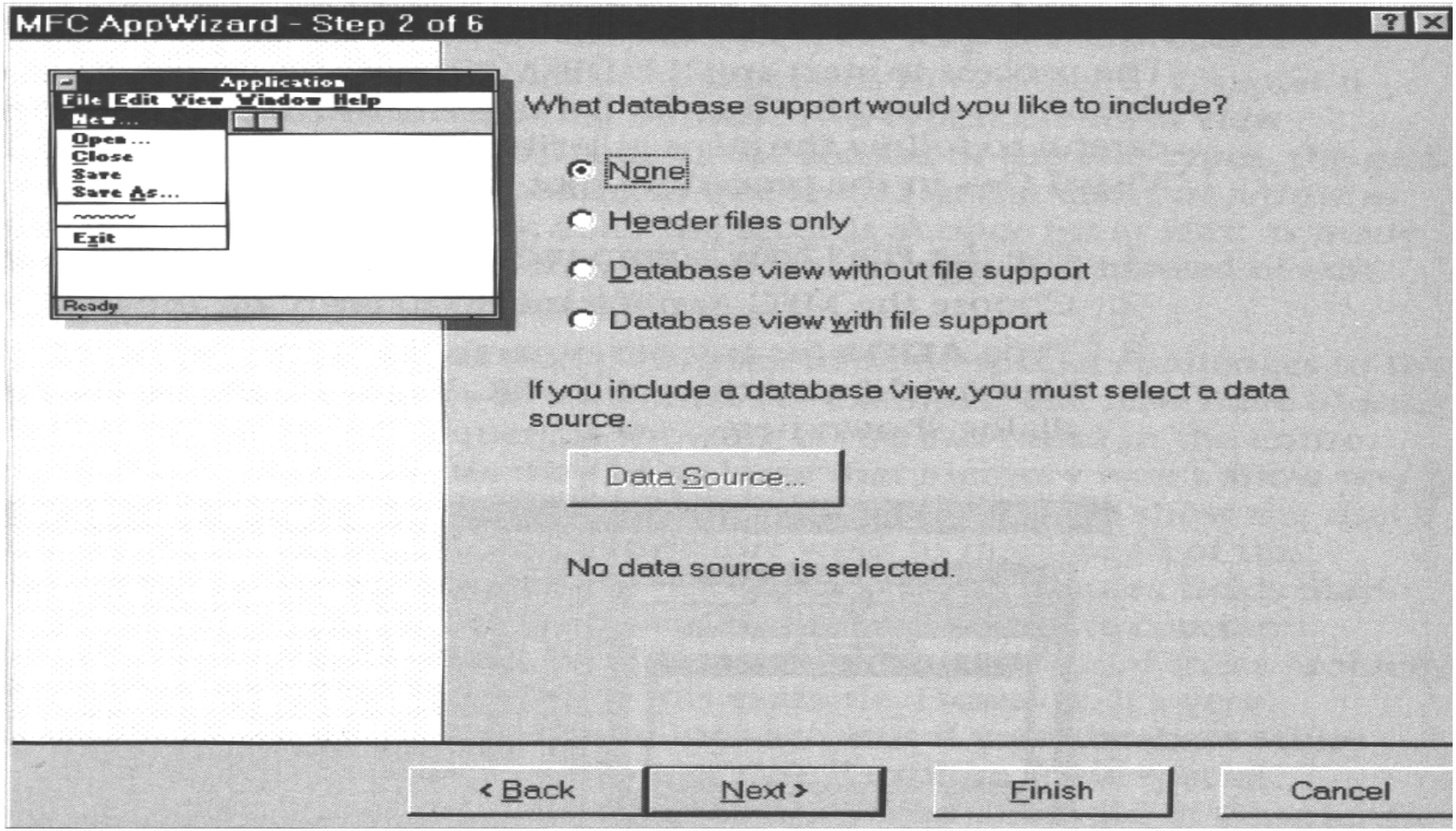
启动 OLE-DB/ADO 网格视图应用程序的过程，有点像在第 6 章中用到的过程，但是有一些重要的差别，所以要仔细遵循下述的步骤。在下面的过程中假设你已经启动了 Visual C++。

1. 使用 File|New（文件|视图）命令显示 New（新建）对话框。
2. 选择 Projects（工程）页上的 MFC AppWizard(EXE)条目。
3. 在 Project Name（工程名）域中输入 ADO1（或任何其它适合需要的名

字），然后单击 OK。你会看到如下图所示的 MFC AppWizard-Step1 对话框。

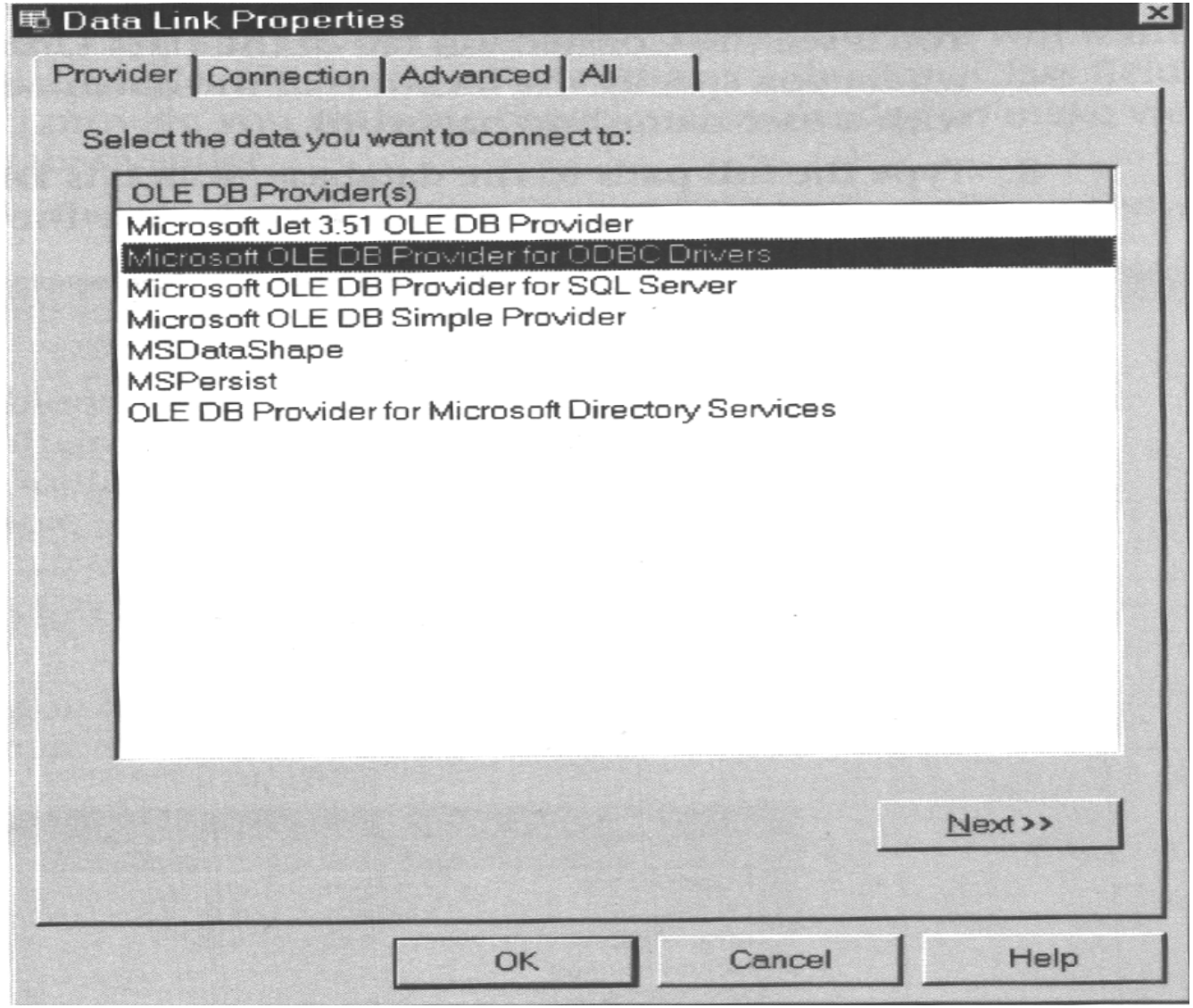


4. 选择 Single Document（单文档）选项，然后单击 Next（下一个）。你会看到如下图所示的 MFC AppWizard-Step 2 of 6 对话框。我们要做的第一件事是为应用程序选择数据库支持层次。



5. 在一系列选项中选择 Database View Without File Support（无文件支持的数据库视图）（各种选项的概述参见第 6 章技巧框中的“选择数据库选项”段）。现在要为我们的应用程序选择数据源。请记住，OLE-DB 用提供者代替了 ODBC 使用的机制；因此，我们不需要 DSN。

6. 单击 Data Source（数据源）按钮。会看到如下图所示的 Data Link Properties（数据链接属性）对话框的 Provider（提供者）选项卡。



我们用这个对话框来选择数据源及其访问方式。在这个示例中我们使用了Access数据库，所以要使用Microsoft Jet 3.51 OLE-DB Provider，不过使用其它OLE-DB提供者也差不多。你会经过相同的步骤，但是在不同选项卡上要求选择的项目会因提供者提供者不同而不同。例如，SQL Server会要求你输入服务器名和其它连接条件，如访问服务器时要使用的安全类型。在获得数据库访问方面，Microsoft Jet 3.51 OLE-DB Provider所需的信息最少。

7. 选择Microsoft Jet 3.51 OLE-DB Provider选项，然后单击Next（下一个）。你会看到Data Link Properties（数据链接属性）对话框的Connection（连接）选项卡。在这里指定想要访问的数据库的位置以及用户名和口令。

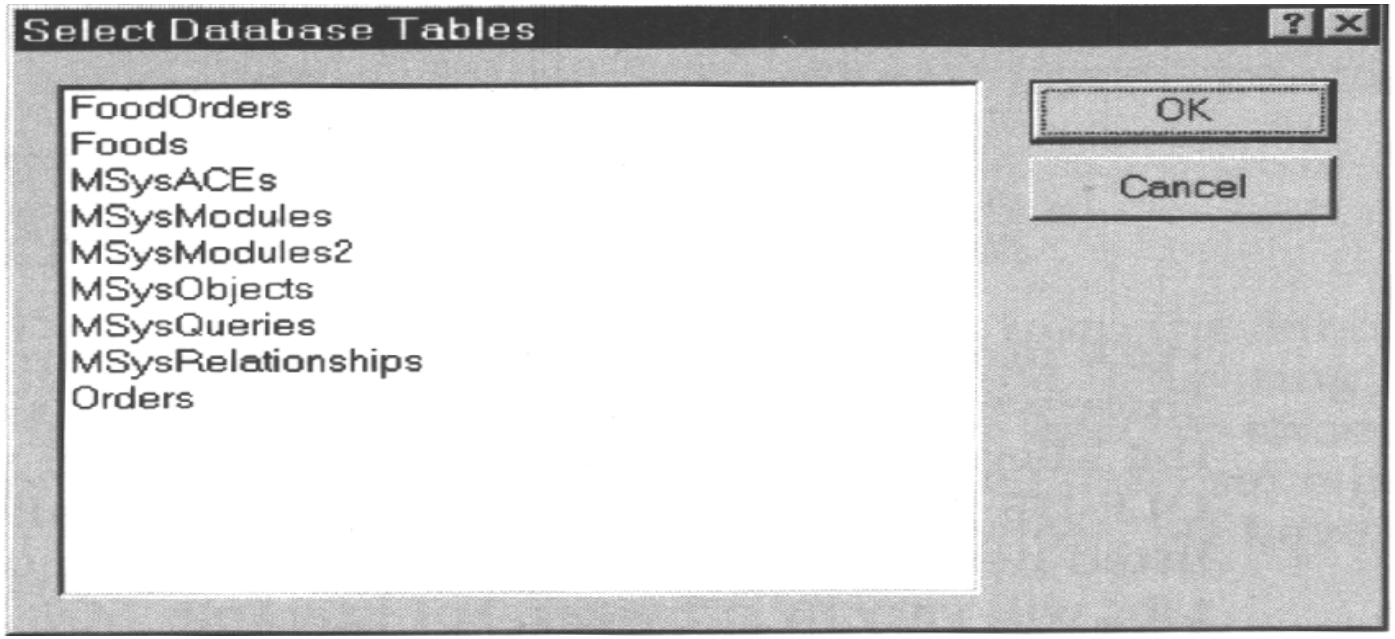
8. 输入访问数据库的完整路径，本示例中使用的路径是DBSamp\Sample.MDB（也可以用Browse（浏览）按钮来搜索硬盘驱动器上的数据库）。

9. 在适当域中输入用户名和口令。示例数据库对这两个值都不作要求。

10. 单击Test Connection（测试连接）按钮。如果在Connection（连接）选项卡中输入了正确的信息，那么你就应该看到Test Connection Succeeded（连接文本成功）对话框。否则，要检查所有提供的值是否都正确。最常见的问题是口令输入错误（这也好办，因为Visual C++可以用星号来代替口令）。

11. 单击OK。现在已经连接数据库了，它并不依赖DSN。

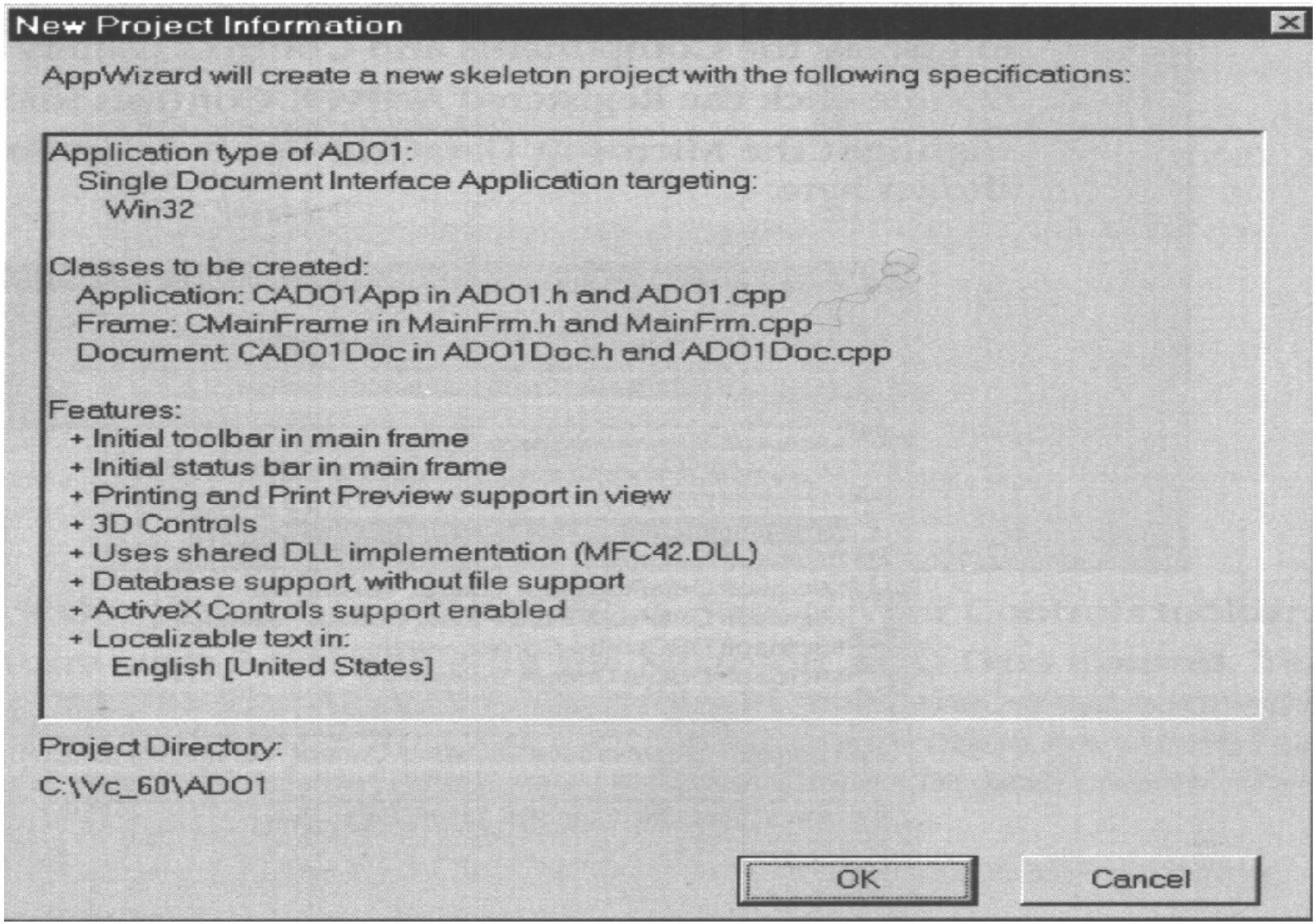
12. 单击OK关闭Database Options（数据库选项）。你应该看到如下图所示的Select Database Tables（选择数据库表）对话框。



13. 加亮 FoodOrders 选项。像第 6 章中的示例那样，我们也可以在需要时选择某个表。然而，通常我们使用查询来访问数据库的内容，这样才能以希望的顺序提取想要的数据。单击 OK。MFC AppWizard-Step 2 of 6 对话框现在应该显示已经选择了数据源。

14. 单击 Next（下一个）四次。你会看到 MFC AppWizard-Step 6 of 6 对话框。在这个对话框上不用改变任何设置，但是你要验证一下在 Base Class（基类）域中是否选择了 COleDBRecordView。这个类提供的功能让你能够更容易地浏览数据库。

15. 单击 Finish（完成）。你会看到如下图所示的 New Project Information（新工程信息）对话框。



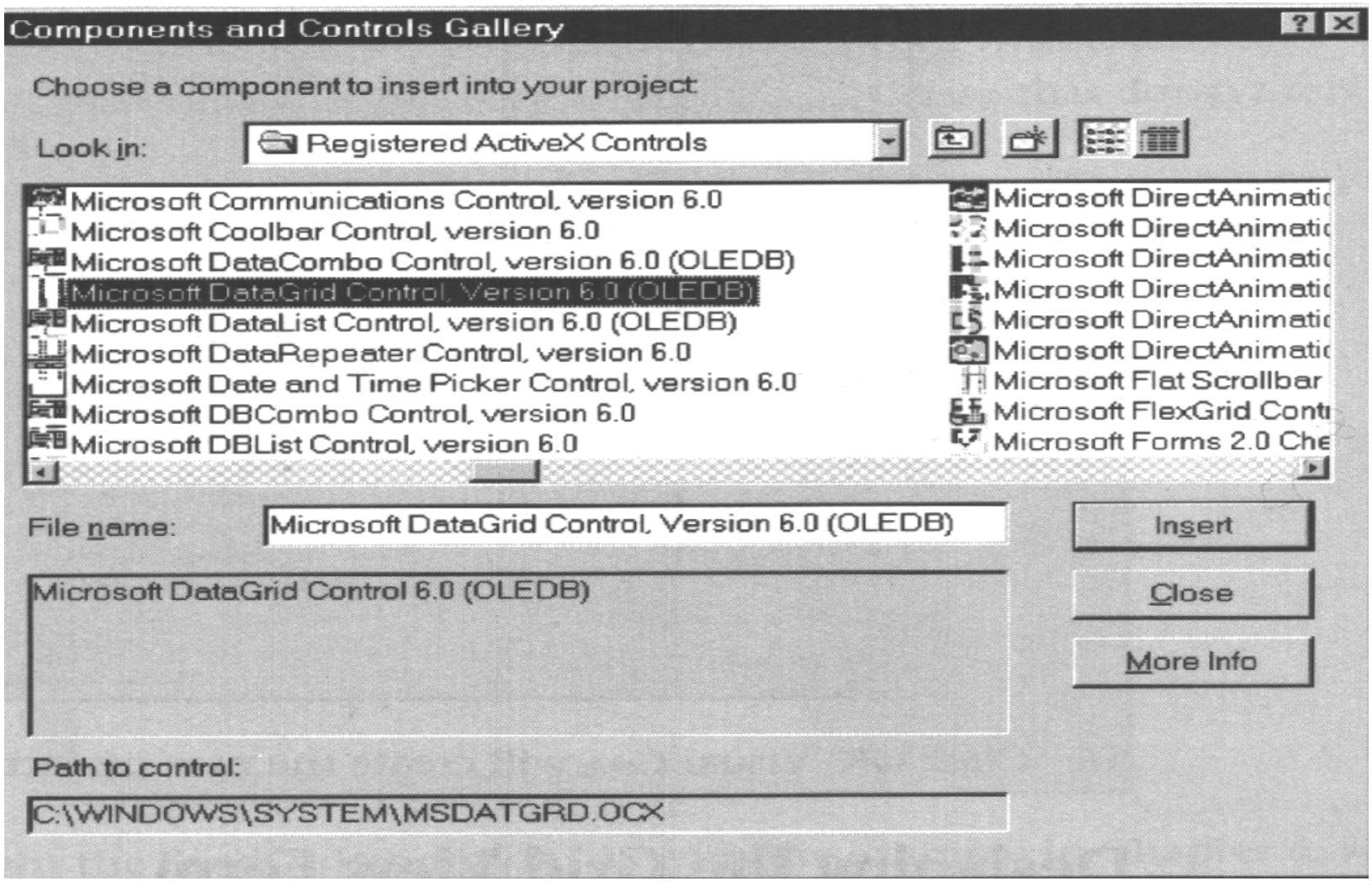
16. 单击 OK。Visual C++创建新的工程。

设计网格视图窗体

此时，应该看到一个名为 IDD_ADO1_FORM 的空白对话框。可以在这个窗体上放置任何东西，包括以报表形式查看所有记录的网格，或者查看每个数据字段的分离控件。现在你到了要选择如何查看信息的时候了。在本例中，我们要创建网格，但是，这里并没有任何东西要把你限制在某种视图类型上，这一点很重要。

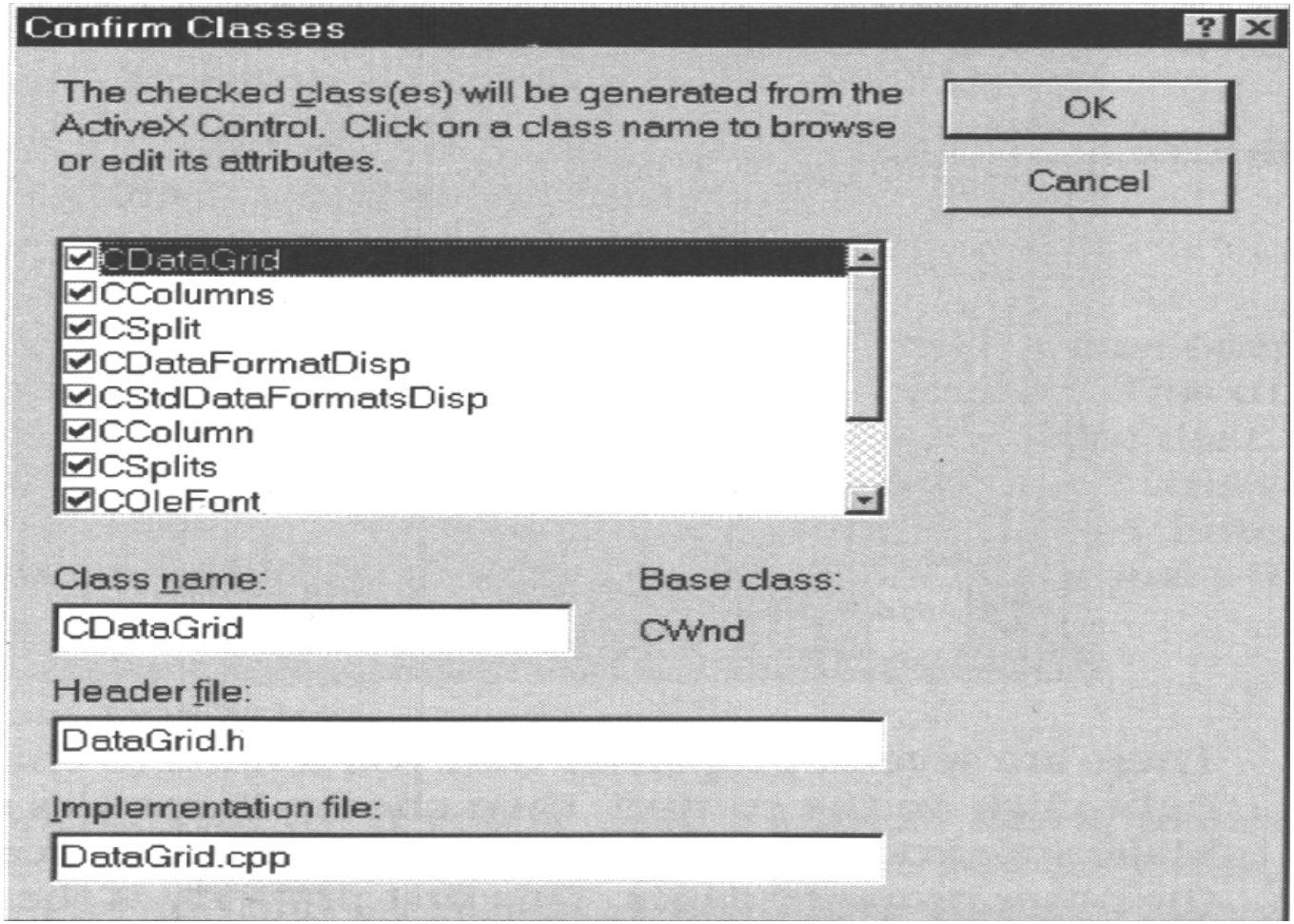
创建网格视图要做的第一件事是，向控件列表中添加一个网格。幸运的是，已经有了一个特殊的网格被设计成可以应用 OLE-DB。很明显，我们需要在窗体上做另外一些工作，但是添加了这个特殊的控件将会为我们节省许多的时间。下面的过程帮助你把该控件添加到 Controls（控件）工具条上，然后再把它添加到 IDD_ADO1_FORM 对话框中。

1. 用 Project|Add To Project|Components and Controls（工程|添加到工程|部件和控件）命令显示 Components and Controls Gallery（部件和控件库）。
2. 双击 Registered ActiveX Controls（已注册的 ActiveX 控件）文件夹。
3. 加亮如下图所示的 Microsoft DataGrid Control,Version 6.0(OLEDB)条目。



4. 单击 Insert（插入）。Visual C++显示一个对话框询问是否想插入这个部件。

5. 单击 OK。你会看到一个如下图所示的 Confirm Classes（确认类）对话框。



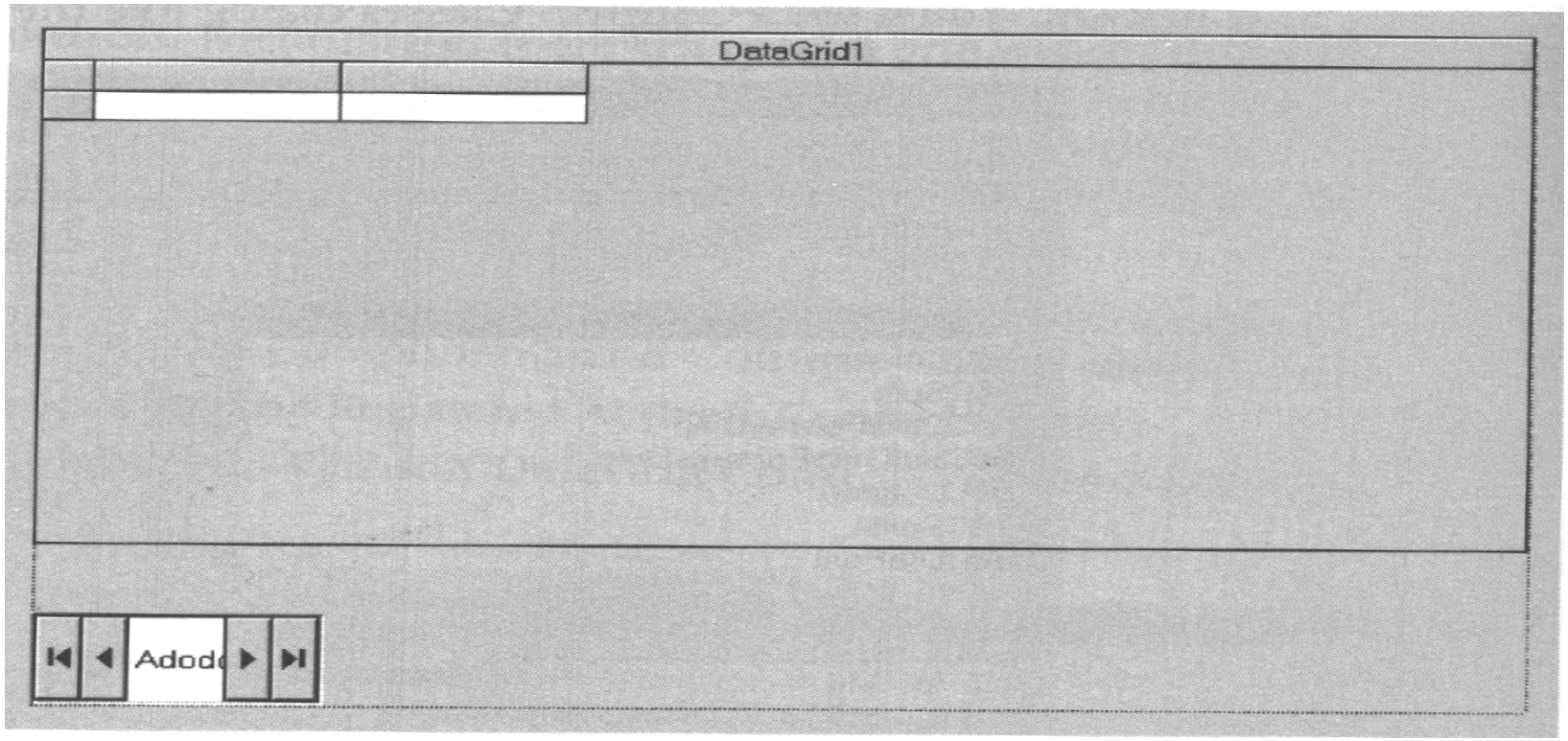
6. 单击 OK。Visual C++把控件添加到 Controls（控件）工具条中。

7. 执行 Microsoft ADO Data Control,Version 6.0(OLEDB)的步骤 3 到 6。ADO Data Control 将通过我们先前创建的 OLE-DB 连接建立与示例数据库的实际连接。

8. 单击 Close（关闭）关闭 Components and Controls Gallery（部件和控件库）

对话框。

现在我们已经有了两个控件可以使用，把它们放在窗体上并加以配置。不要担心精确调整控件大小的问题——我们用代码来完成这项工作。你需要把这些控件放置到窗体的角上。下图是示例中使用的窗体。



在 DataGrid1 上有几个属性需要更改。只要右击控件，然后从上下文菜单中选择 Properties（属性）即可。因为有一些属性不在其它的属性页上出现，所以一定要选择 All（全部）选项卡。第一个属性最重要，因为它设置控件的数据源。把 DataSource（数据源）属性设成 IDC_ADODC1。由于我们想完整地控制数据

库以便于彻底地测试应用程序，所以你也要把 AllowAddNew 和 AllowDelete 属性设置为 True。通常情况下，Data Grid（数据网格）控件被设计成允许查看和修改记录，但不允许添加和删除记录。这也是一种对数据库保护的中等措施。此时，人们可以少许地修改数据库的内容，但不会达到破坏数据库的程度。

现在配置数据源本身。右击 ADODC1 控件，然后从上下文菜单中选择 Properties（属性）。选择 Control（控件）选项卡。注意这个选项卡上的 Use Connection String（使用连接字符串）域——如果它还未被选中，那么一定要选中它，这是我们选择数据源的地方（我们在这里选择数据源的过程与本章前面所用的过程相似）。下面是建立连接字符串的过程。

1. 单击 Build（建立）。你会看到 Data Link Properties（数据链接属性）的 Provider（提供者）选项卡。

2. 加亮 Microsoft Jet 3.51 OLE-DB Provider 选项，然后单击 Next（下一个）。你会看到 Data Link Properties（数据链接属性）的 Connection（连接）选项卡。

3. 在 Select or Enter a Database Name（选择或输入数据库名）域中输入数据库的位置。也可以用 Browse（浏览）按钮在本地或网络驱动器上定位数据库。示例程序在这个域中填入了 DB_Samp\Sample.MDB。

4. 如果需要的话，输入用户名和口令（示例数据库对这两个条目都不做要求）。

5. 单击 Test Connection（测试连接）。如果连接信息正确，你就会看到一个 Test Connection Succeeded（测试连接成功）对话框。

6. 单击 OK 两次。完成连接工作。

在应用程序设置完成之前，我们还必须再做一件事。再次打开 ADODC1 控件的 Properties（属性）对话框。这一次选择 RecordSource（记录源）选项卡。在这个选项卡的底部会看到 Command Text(SQL)字段。在这里输入从刚刚取得连接的数据库中提取数据的命令字符串。输入下面的字符串访问在第 5 章中设置的 FoodOrders（食品订购）查询所提供的所有信息。

```
Select * from FoodOrders
```

关闭 Properties（属性）对话框。如果现在编译并运行这个应用程序，就能看到网格形式表达的数据。不但如此，还可以在数据库中编辑、删除和添加记录。然而，有些地方我们需要编写一些代码，比如，实现当主窗口调整大小时 Data Grid（数据网格）控件也随之调整自身大小的功能。我们会在本章的下一节处理这些任务。

高级技巧

在应用程序中使用两个数据库连接

你可能已经注意到了，为 ADO1 应用程序使用了两个连接。我们在应用程序的设计过程中创建了第一个连接，第二个连接是在向应用程序添加 AdoDC1 之后创建的。AdoDC1（第二个连接）是查看窗体时的活动连接。如果你试着使用第一个连接（直到对菜单和工具条做了更改时才可用），不会看到 Data Grid（数据网格）的内容有任何变化。

下面是 C++应用程序开发者要考虑的问题。没有那么多的控件像 ADODC 控件那样生来就支持 OLE-DB。如果计划向应用程序或报告中添加单记录视图窗体，就可能需要第二个连接来访问数据库中的数据。换言之，即使你没有打算立即使用第二个连接，也要在应用程序中包括它，至少在 Microsoft 使所有的 Visual C++数据绑定控件都与 OLE-DB 兼容之前是这样。

使用两个或者更多的数据库连接还有另外一个原因。使用两个连接允许你在创建高速应用程序时有更大的灵活性。可以为打印窗口使用一个连接，为数据编辑窗口使用另一个连接。有两个连接意味着，在处理一个窗口时不必担心为另一个窗口所使用的连接状态。连接是分离的，所以每个窗口的连接状态都被保留。

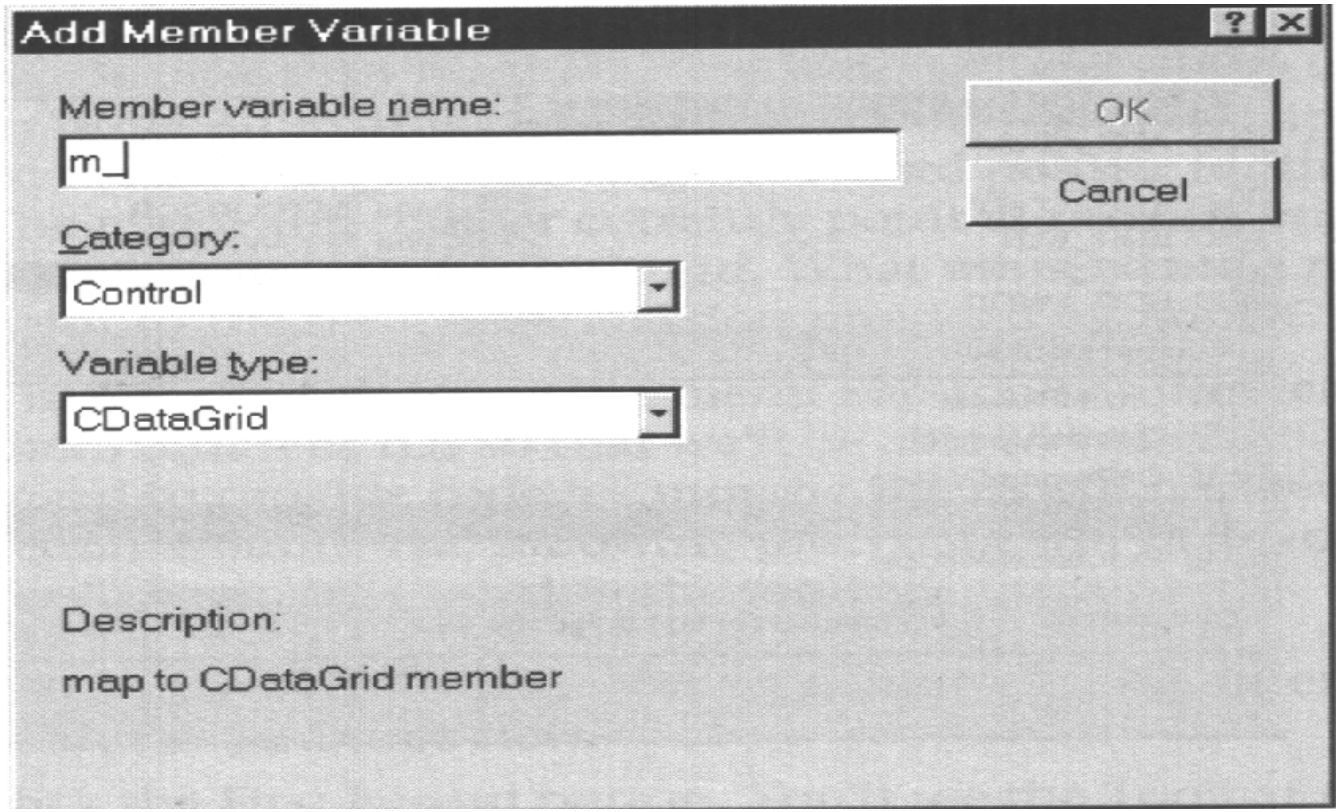
很明显，使用多个连接时存在着平衡问题。毫无疑问，使用多个连接后，应用程序的可靠性会提高一些，或许速度也有所提高。但是，获取这些特性的代价　　是使用了更多的内存。如果你在为内存有限的系统编写应用程序，那么就不必

奢侈地使用两个或多个数据库连接。内存不足的系统已经比大多数机器都慢了，用户不太可能注意到速度有所降低。然而，你需要添加另外一些代码，以确保两个窗口的连接状态信息被适当地保存，确保用户在进行一些诸如同时打印和编辑数据库操作时，不会粗心地损坏数据库。

向 AD01 中添加一些窗体代码

示例程序在很多方面不完整，其中大部分属于可用而不完美这一类。例如，如果你现在调整应用程序的大小，窗体会变得更大，而 Data Grid（数据网格）控件不会。出现的另一个问题是，Data Grid（数据网格）控件中的所有列都被设成通用格式，这种格式可以很好地适应大多数想要显示的数据。但是，我们需要为 Boolean（布尔）值将格式设成其它的风格，而在使用通用格式时 Boolean 值显示为数字。

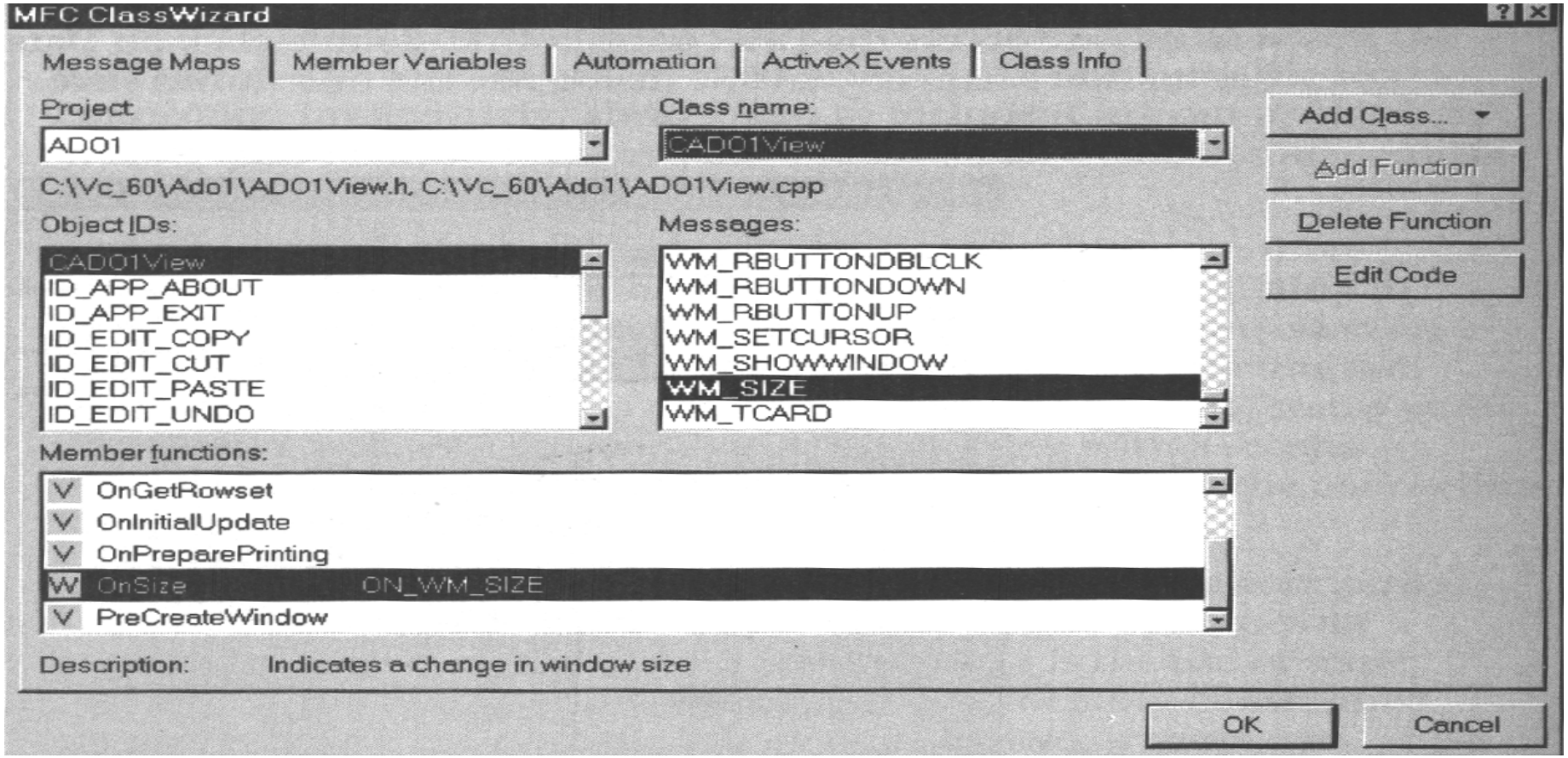
在解决这些问题之前，我们需要创建几个访问这两个控件的变量。按住 CTRL 键并双击 DataGrid1。你会看到如下图所示的 Add Member Variable（添加成员变量）对话框。



请注意，Category（分类）域选择了 Control（控件），Variable Type（变量类型）域包含了 CDataGrid。在 Member Variable Name（成员变量名）域中输入 dataGrid1（完成时应该是 m_dataGrid1）。单击 OK 完成这个过程。现在，为 ADODC1 进行相同的操作。这一次使用 m_adoDC1 作为变量名。

现在添加第一个函数。使用 View|ClassWizard（视图|类向导）命令打开 MFC ClassWizard。第一段代码让我们在必要时改变两个控件的位置并调整其大小。在 Class Name（类名）域中选择 CADO1View，在 Object IDs（对象 ID）域中

选择 CADO1View。在 Messages（消息）域中找到 WM_SIZE 消息。单击 Add Function（添加函数），MFC AppWizard 对话框看起来应该如下图所示。



单击 Edit Code（编辑代码），进入刚刚创建的新函数。程序列表 7.1 显示了当我们改变应用程序窗体的大小时也同时改变控件大小的代码。

程序列表 7.1

```
void CADO1View::OnSize(UINT nType, int cx, int cy)
{
      int iHeight, iWidth;                    // Control height and width.
      CRect rect;                  // Control size.

    // Perform the default action.
   COleDBRecordView::OnSize(nType, cx, cy);

   // Reposition AdoDC1.
   m_adoDC1.GetClientRect(rect);
   iHeight = rect.Height();
   iWidth = rect.Width();
   m_adoDC1.MoveWindow(cx, cy, iWidth, iHeight, TRUE);

   // Resize and reposition DataGrid1
   m_dataGrid1.MoveWindow(0, 0, cx, cy - iHeight, TRUE);
}
```

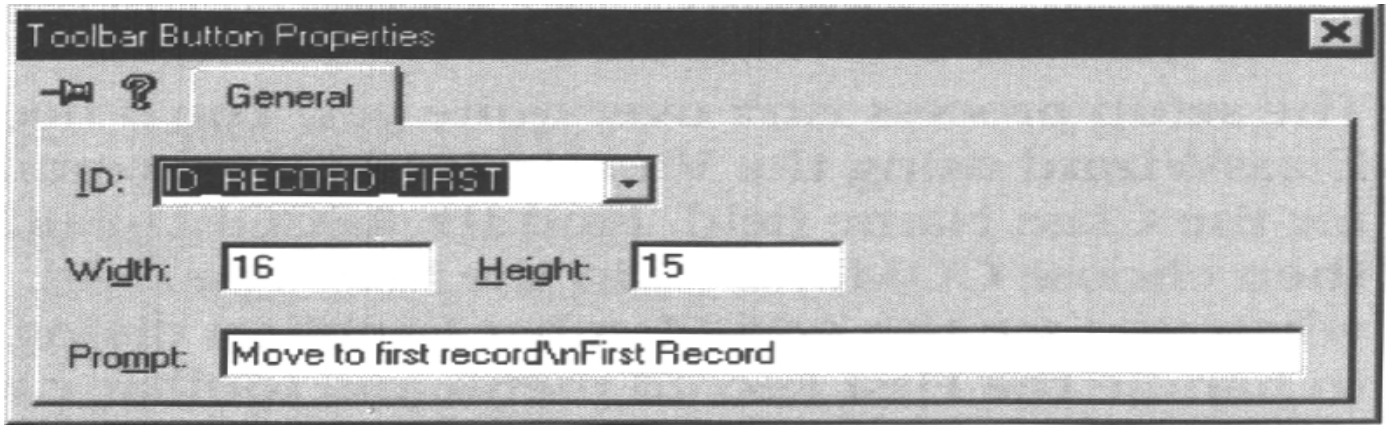
修复工具条和菜单

工具条和菜单是现在大多数应用程序的一部分，所以 MFC AppWizard 会自动地创建它们。可惜的是，自动创建过程并非总是很完美，并且在某些情况下也没有人会抱怨出现了故障。如果用 AdoDC1 控件在记录间移动，我们的应用程序会运行良好，但是如果用户想使用工具条或菜单的话，会怎么样呢？菜单和工具条好像不会做任何事情，尽管它们正在为第一个连接移动记录指针。菜单和工具条当前指向我们用 MFC AppWizard 创建的第一个连接。我们需要的是指向由 AdoDC1 提供的第二个连接的菜单和工具条。

处理菜单和工具条问题主要分两步。第一，要给它们不同的 ID，这样我们才能为它编写代码。第二，要为移动第二个连接的记录指针提供代码。下面的过程帮助你完成第一步的工作（我们从更改工具条开始）。

1. 在 ResourceView（资源视图）中双击 Toolbar（工具条）文件夹内相应项目，打开 IDR_MAINFRAME 工具条。

2. 双击 First Record（第一个记录）按钮。你会看到如下所示的 Toolbar Button Properties（工具条按钮属性）对话框。在这个对话框中无需修改很多信息，只要修改 ID 字段即可，这样我们才能添加访问第二个数据库连接所需的特殊代码。



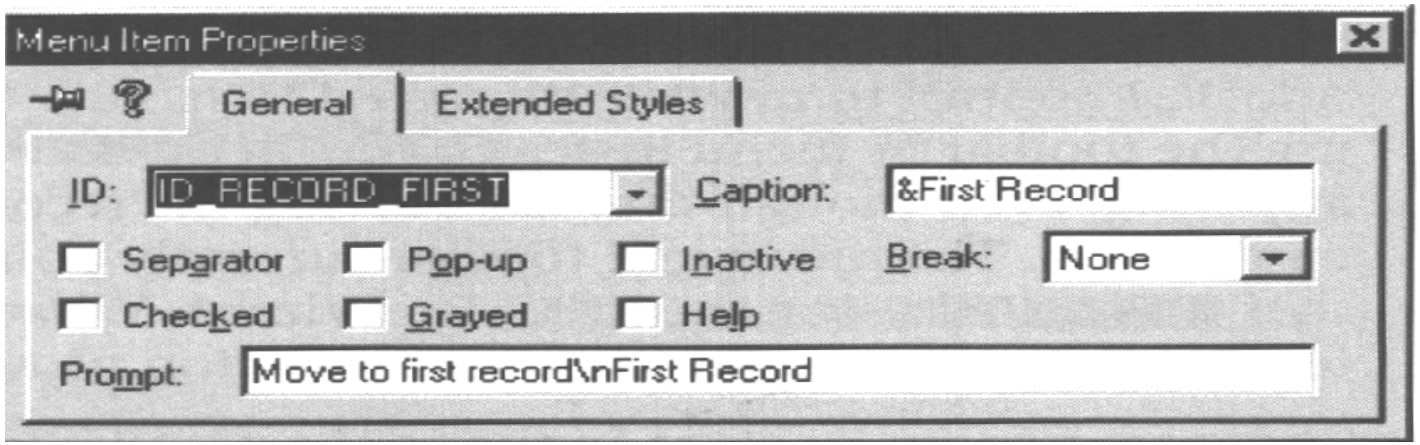
注　First Record（第一个记录）按钮的图案如下图所示：

3. 在 ID 字段中输入 ID_RECORD_FIRST2。我们向访问第二个连接的任何对象名称中都附加上数字 2，使代码更易于阅读。

4. 对 Previous Record（前一个记录），Next Record（下一个记录）和 Last Record（最后一个记录）按钮执行步骤 2 和 3。

5. 在 ResourceView（资源视图）中双击 Menu（菜单）文件夹内的相应条目，打开 IDR_MAINFRAME 菜单。

6. 右击 Record|First Record（记录|第一个记录）命令，从上下文菜单中选择 Properties（属性）。你会看到如下图所示的 Menu Item Properties（菜单项属性）对话框。同样，除了 ID 域的值，我们不必修改其它内容。



7. 在 ID 域中输入 ID_RECORD_FIRST2。这个域段中的项目正好匹配工具条的项目，这一点绝对是必要的。否则，工具条和菜单不会互相“跟踪”。

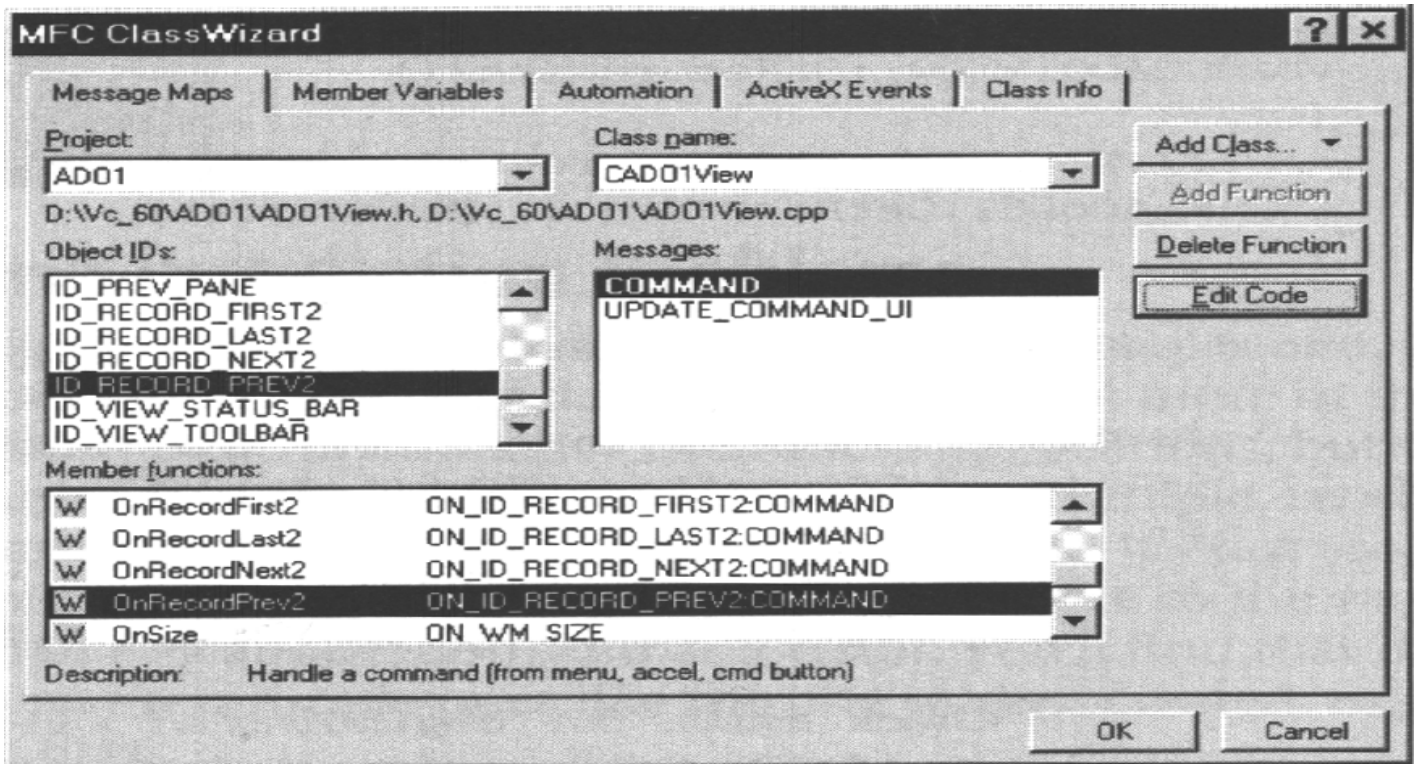
技巧 可以这样确保工具条和菜单名互相匹配，即从下拉列表框中选择正确的 ID 值，而不是把值输入到 ID 域中去。然而，在大多数情况下，如果是添加一个数字，就和输入值一样容易。

---

8. 对 Record|Previous Record（记录|前一个记录）、Record|Next Record（记录|下一个记录）和 Record|Last Record（记录|最后一个记录）命令重复步骤 6 到 7。

设置过程还没有完成。要用 View|ClassWizard（视图|类向导）命令打开 MFC ClassWizard。为 Class Name（类名）选择 CADO1View。在 Object IDs 字段中找到 IN_RECORD_FIRST2，然后选择 Messages（消息）字段中的 COMMAND（命令）。单击 Add Function（添加函数），看到 Add Member Function（添加

成员函数）对话框时单击 OK。你刚才已经添加了一个处理 First Record（第一个记录）菜单和工具条项目的函数。对 ID_RECORD_LAST2，ID_RECORD_NEXT2 和 ID_RECORD_PREV2 Object Ids 重复这个过程。完成时，MFC AppWizard 对话框应该如下图所示。



此时，你已经为访问为这个程序而创建的第二个连接设置了菜单和工具条。现在要添加一些使菜单和工具条项目发挥作用的代码。在 MFC AppWizard 对话框中单击 Edit Code（编辑代码），Visual C++把你带到我们刚刚创建的新函数。程序列表 7.2 显示了需要添加的代码。请注意，#include 指令要和其它的#include

指令一起放置在代码文件的开始位置。

程序列表 7.2

```
// Include the C_Recordset class so that we
// can manipulate the recordset directly.
#include "_Recordset.h"

void CADO1View::OnRecordFirst2()
{
        // Move to the first record.
        m_adoDC1.GetRecordset().MoveFirst();
}

void CADO1View::OnRecordLast2()
{
        // Move to the last record.
       m_adoDC1.GetRecordset().MoveLast();
}

void CADO1View::OnRecordNext2()
{
        // Move to the next record.
        m_adoDC1.GetRecordset() .MoveLast();
```

```
}

void CADO1View::OnRecordPrev2()
{
        // Check to see if we can move the record pointer.
         if (!m_adoDC1.GetRecordset().GetBof())
{
         // Move to the previous record.
         m_adoDC1.GetRecordset() .MovePrevious();

         // Check again for beginning of file.
         if (m_adoDC1.GetRecordset().GetBof())
  {
                // If we are at the beginning of the file,
                // advance the record pointer to the first record.
               m_adoDC1.GetRecordset() .MoveNext();
         }
   }
}
```

程序列表 7.2 中的大多数代码都很简单。我们用 m_adoDC1 内存变量得到对记录集（用户当前正在观看的记录集）的访问。我们一开始访问记录集，就可以把记录指针移到适当的记录位置上。

现在来看看 OnRecordPrev2()函数。由于我们在查阅文件起始记录时没有执行任何缺省动作，所以这个函数要求的代码比其它函数要稍微多一些（如果查阅到文件的末尾，应用程序自动向表中添加一个新记录）。所有这些额外的代码帮助你避免可视化问题，它可能会导致调用许多支持。如果记录指针在文件的开始位置遇到了文件结束，那么网格控件上的小型记录指针指示器就会消失。即使网格和 ADODC 都在正常工作，对用户来说，这可能表示发生了某种错误。要避免这个问题很简单：只要检查文件指示器的开始，并适当改变记录指针的位置。

高级技巧

避免安装问题

数据库应用程序是你可以创建的最复杂的应用程序之一。人们不但希望这些应用程序在稳定性和坚固性方面有更大的提高，而且应用程序本身的整体规模也越来越庞大了。例如，大多数的数据库应用程序要求某种网络的支持。判断出数据库应用程序在要求支持方面比任何程序都多并不难，结果你会发现，即使像我们刚刚创建的这样简单的数据库应用程序，都要求众多的支持文件。

创建一个安装程序时会碰到的第一个问题是，找出所需的全部文件。通常，在获得完整的文件列表之前，要检查多个信息源。必须寻找的两个文件集是，厂商专用的数据库访问驱动程序和操作系统支持文件。可能还需要配置实用程序和依赖于所使用的 DBMS 的其它程序文件。厂商通常会提供完整的厂商专用文件，它可能隐藏在文档的某些角落中，一定要花些时间把它们都找出来。在某些情况下，厂商还会慷慨地提供一系列通用操作系统文件，但是在假定厂商提供了正确或最新的信息之前，一定要检查一下 Microsoft 的 Web 站点，看看有没有最新的驱动程序。表 7.1 提供了一系列完整的文件，你在处理依赖 Microsoft Jet Engine 的数据库应用程序时需要它（这个表是特别为 DAO 设置的，但也可用于其它形式的数据库访问）。

表 7.1 Microsoft Jet Engine(DAO)所要求的文件

用途	3.0 版	3.5 版
Microsoft Jet Engine 库	MSJT3032.DLL	MSJET35.DLL
Microsoft Jet Database Engine 出错 DLL	MSJTER32.DLL	MSJTER35.DLL
Microsoft Jet Database Engine 国际化 DLL	MSJINT32.DLL	MSJINT35.DLL
Visual Basic for Applications 开发环境——表达式服务加载程序	VBAJET32.DLL	VBAJET32.DLL

续表

Visual Basic for Applications Runtime——表达式服务	VBAR2232.DLL	VBAR332.DLL
Visual Basic for Applications 对象库	VEN2232.OLB	N/A
Microsoft(r) Red ISAM	MSRD2X32.DLL	MSRD2X35.DLL
*Microsoft Jet xBase ISAM	MSXB3032.DLL	MSXBSE35.DLL
*Micorsoft Jet Paradox ISAM	MSPD3032.DLL	MSPDOX35.DLL
*Micorsoft Jet Text ISAM	MSTX3032.DLL	MSTEXT35.DLL
*Micorsoft Jet Excel ISAM	MSXL3032.DLL	MSEXCL35.DLL
*Microsoft Jet Lotus 1-2-3 ISAM	MSLT3032	MSLTUS35.DLL
*Microsoft Jet Exchange ISAM	N/A	MSEXCH35.DLL
Microsoft Wingman 库	MSWNG300.DLL	MSWNG300.DLL
Microsoft ODBC 桌面驱动程序包	ODBCJT32.DLL	ODBCJT32.DLL
Microsoft ODBC 桌面驱动程序包	ODBCJI32.DLL	ODBCJI32.DLL
ODBC Helper 函数 DLL	ODBCTL32.DLL	ODBCTL32.DLL
*Microsoft RDO（远程数据对象）Engine 控件	N/A	MSRDO20.DLL
*Microsoft 客户机光标 DLL	N/A	RDOCURS.DLL
*Microsoft 复制库	N/A	MSREPL35.DLL

* 根据计划要提供的数据库支持，你可能需要的可选文件（例如，如果计划访问像 FoxPro 和 dBASE 创建的 xBase 数据库，就只需要 xBase ISAM DLL）。

另一个安装问题是数据库的存放位置。某些程序员会假定用户将需要直接的网络访问，并且管理人员会把数据库放在服务器上的特定位置。不要做这样的假定，尤其是在今天这个 Internet 主宰的市场中。服务器访问可能要经历一次 Internet 连接，甚至可能来自存储于数据库中的记录的本地子集。换言之，除非你也可以保证亲自把数据放在预期的位置，否则不要做任何关于数据会存储在哪里的假设（即使这样，管理也可能决定它随时需要 Internet 的连通性）。

我们再回过头来谈一谈支持文件。你们大多认识到了，在应用程序依赖特定版本的公共支持文件（如 MFC 运行文件）的时候会有问题出现。创建数据库应用程序时，这个问题会大大增多。首先，你还要考虑 MFC 的运行。添加到这些支持文件中的是数据库访问所需要的公共操作系统文件和你所需要的厂商专用文件。现在你要及时了解三个产品版本，它们随时都可能会改变。

解决这个问题的办法是，一定要检查安装过程中用户安装在系统上的每一个文件的版本（现在你知道了为什么一些安装程序会工作上几个小时！）。如果一个文件的版本和为这个应用程序所使用的文件不匹配，那就要征得用户的同意，用所需要的文件改写这个文件。现在，有一个问题，用户都乐于假设新版本会更好一些。另一方面，如果应用程序依赖于较早版本的文件，那它有可能在新版本下运行不了。你要特别检查一下新版本的文件，并在消息中放入某种关于应用程序可能在新版本下运行不了的警告（除非你事先就知道它可以运行）。

技巧 每次应用程序启动时，很多的数据库程序员都会执行检查文件版本号的例程。在一些情况下，检查应用程序所使用的每一个文件是没有必要的。但是，检查公共或共享文件的版本号却是一个好主意。用户可以安装另一个应用程序，它改写了你的应用程序所要求的文件版本。确保应用程序可以满足需要的唯一办法是，在程序初始化过程中检查这些文件。在应用程序实际启动之前就中断，比应用程序一启动和数据库一打开程序就崩溃要好一些。

其它的安装问题和通常安装程序的设计方式有关。例如，大多数安装程序假定，如果在用户的硬盘驱动器上有足够的空间来存储应用程序的所有文件，应用程序就会正常运行。可惜的是，在数据库应用程序的情况下，你还必须考虑像记录集的本地拷贝这样的事情，因为记录集可能非常大，内存存储不了。换言之，要在安装程序中适当调大硬盘容量需求以满足应用程序的处理需要，而这一工作在安装大多数其它类型的应用程序时没有必要考虑。

7.2 向应用程序添加报告

人们都希望为数据库应用程序至少创建一个报告。即使是像我们在本章创建的这样一个小的应用程序，也可能因报告类型丰富多采而容光无限。例如，概述报告通过摘要相似的数据来从整体上显示数据库的内容（甚至摘要的方式也可以创建多个报告），细节报告则显示所有的记录。除了报告所包含的数据及其出现的顺序以外，还有一些事情要考虑，比如从审美观点来讲的报告的外

观。你可能不想花时间用最佳的文字和图形为工作组会议打印报告，但我可以肯定，经理会议通常需要这样的报告。

我无法在一个章节中告诉你创建报告的每一个要点。本章要讨论的是关于创建一批复杂报告的技巧，其中包括浏览数据的各种方法，并介绍一个如何完成上述部分目标的程序设计示例。

存在打印困难吗

创建一个你所需要的报告可能有点困难，在某些情况下也确实是如此。处理一项工程时，可能五个经理都想把他们自己的一组报告添加到应用程序中去，而你知道，在允许的时间范围内，你确实只有工夫创建一两组报告。不要失望，如果花点时间分析一下经理们在期待些什么，并且在打印过程中增加一点灵活性的话，就可能根本没有必要去编写所有的报告了。

当你试图减低应用程序的复杂性时，应该做的第一件事是，把所需的报告分解成几个简单的需求。以这样的列来创建一个表是个好主意，即按列说明简单的需求，然后在旁边列出报告。如果在报告中找到了某种对比方式，就有可能把几个报告合并成一个，然后按需要添加配置对话框。下面是关于如何把报告分解成一组部件的一些想法。

- 排序　每一个报告都要求某种排序。否则，就会形成一系列没有人能够理解的紊乱数据。请记住，数据库的一个主要目标是，把所包含的数据组织起来，使它更容易被理解。你的报告也应该是一样。排序是一种把两个或多个报告合并到单个程序设计任务中的最容易的方式。

- **分组** 有时候你想要细节报告在单个标题下组织数据。例如，你可能在合同数据库中用 ZIP 代码创建分组。ZIP 代码作为标题显示，该 ZIP 代码区中的所有合同都在该标题下按顺序排列。谈到组织数据时，分组和排序并没有太大的差别。但是，即使打印输出的主要部分都相同，把两个报告和不同的组结合起来也是非常困难的。原因很简单：把各种报表头组织在一起将使得打印例程十分复杂。然而，你可以把一个包含分组，另一个不包含分组的两个相似报告结合起来。此时，你只需要在程序中提供一个开关，用它打开标题或者关闭标题就可以了。
- **分组合计/摘要** 许多报告会提供这样的脚注，整个报告或一组有关的数字在那里被记录下来。脚注还有很多其它的统计用法，但数字记录是最普通的类型。在有组标题的情况下，如果有可能，要把两个有不同合计或者摘要的报告结合起来是很困难的。但是，可以把未使用任何类型脚注的报告和使用了脚注的报告结合起来，只要它们的列是相同的即可。
- **报告外观** 如果你有两个报告使用相似的数据但具备不同的外观风格，那么可以试着把它们结合起来。如果唯一的差别只是文字所用的字体或者是像添加到报告顶端的公司徽标这样的小窗口，那么创建两个不同的打印例程就没有什么实际意义了。
- **细节数据的层次** 某些报告只是其它报告的简单综述。换言之，有一些报告完全只使用父数据。通过以某种方式摘要子记录来提供综述信息。把综述报告和显示数据库详细内容的报告结合起来相当容易。当然，这

也存在着报告中包含什么内容的问题。例如，如果一个报告以售货员顺序显示本月的每一张发票，而另一个报告显示各位售货员的总销售额，那么你就可以把这两个报告合成为一个。使用一个简单的开关来判断用户是要摘要数据或者要按原样打印子表中的每一个记录。

- 筛选　在数据库中寻找部分而非全部的记录，对人们来说十分普通。如果两个报告唯一的差别是用来决定该筛选哪些记录，那么你就可以把这些报告结合起来。筛选是你对 DBMS 所做的查询的函数，而不是你编写的代码的函数。换言之，你要做的只是更改一两行代码。

此时，你可能会感到奇怪，为什么不创建一个带有众多配置对话框的单个报告，让用户来产生他们所需的报告呢？不这样做有几个充足的理由。第一，大多数用户会对创建报告所需的大量对话框不知所措。你会发现，和自己设计报告相比，你还要花大量的时间让用户学会提高创建报告的速度。第二，你要引进自定义报告，还要为用户创建某种把那些报告存盘的方法，这样他们以后才能在无需很多支持的情况下，创建相同的报告。

避免通用报告的另一个原因是与安全有关的风险。如果提供给用户太多不同的创建自定义报告的方法，你可能会发现，这也带来了很多的安全漏洞。管理上通常要保证公司的秘密数据不会误入他人之手。这意味着，除了要增强数据库本身的安全性外，还要增强已打印文件的安全性并把它牢牢地控制住。

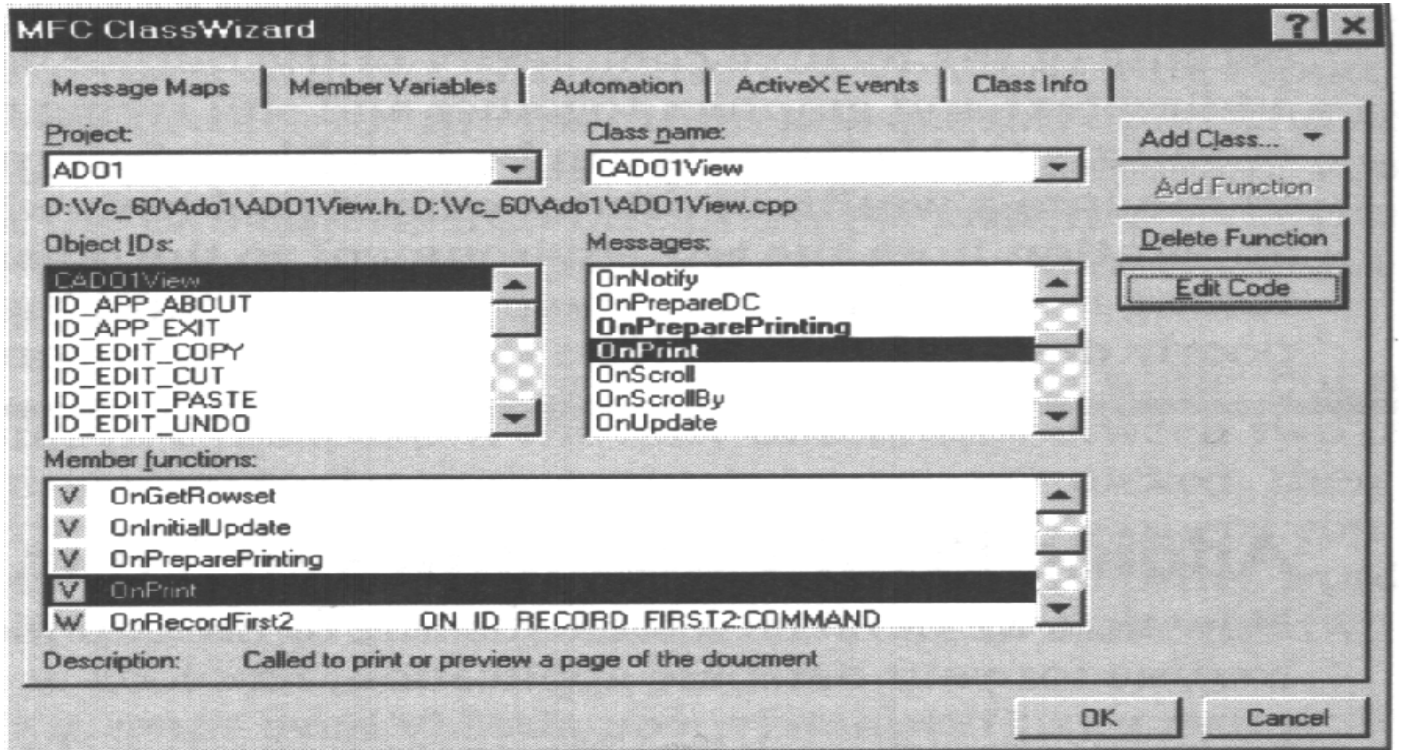
可移植性 当使用 Windows 碰到打印机不兼容的问题时，不要误以为是安全性问题。当然，Windows 为打印例程提供了一致的程序设计 API，这使编程工作变得更加容易。你还得到了打印驱动程序支持的明显优点——至少不必像 DOS 环境下的程序员那样，还要编写自己的驱动程序。但是，优点也就这么多。厂商仍然有可能提供单薄的驱动程序，这使得很难从看上去正确的应用程序中得到正确的输出。你还会发现，由于产生输出的相对复杂性，与其它开发人员相比，这些问题会更多地发生在数据库程序员身上。仅仅把文本和图形添加在一起并保持各种表格元素对齐就可能存在某些问题。可以这么说，把应用程序从测试环境转移到实用环境中时，会遇到大量与打印机有关的问题。即使是世界上最好的测试也不能帮助你避免设计粗糙的打印驱动程序所带来的问题以及其它各种各样的兼容性问题。

给应用程序添加打印功能

看过了为数据库应用程序编写打印例程的复杂性，现在我们来看一看解决这些问题的办法。幸运的是，在我们使用的 ADO1 应用程序中，现在已经有了所需的三个函数框架。我们只看一看 CADO1View 类。有三个函数：OnPreparePrinting()，OnBeginPrinting()和 OnEndPrinting()。第一个例程让你设置打印机并完成其它的预备任务，如在查询中保存当前的记录数，这样才能在打印作业完成后返回到那里。第二个例程 OnBeginPrinting()让你准备设备的上

下文信息。可以把设备上下文看作是画家的调色板——用它来绘制想要发送给打印机的信息。在本示例中，我们会经常碰到 Windows GDI（图形设备接口）这么一个特别的东西，所以你迟早会很好地理解设备上下文究竟意味着什么。最后，OnEndPrinting()例程让你把应用程序的总体设置恢复成打印前的条件。例如，你应该在这里把查询指针返回到其打印前的记录处。

哎，还遗漏了一个函数。我们还没有用来向打印机发送数据的打印例程。我们要做的第一件事是使用 MFC ClassWizard 向程序中添加一个 OnPrint()函数。OnPrint()函数包含实际把打印信息输出到打印机所需的代码。其它与打印有关的任务也调用这个例程，如大多数应用程序提供的 Print|Preview（打印|预览）命令。使用 View|ClassWizard（视图|类向导）命令打开 MFC ClassWizard 对话框。在 Class Name（类名）域中选择 CADO1View，加亮 Object IDs（对象 ID）列表中的 CADO1View 和 Messages（消息）列表中的 OnPrint。单击 Add Function（添加函数）。Member Functions（成员函数）列表中应该添加了一个新函数，如下图所示。



单击 Edit Code（编辑代码）按钮。现在向应用程序中添加一些与打印有关的代码。程序列表 7.3 包含了需要向 OnPrint()函数添加的代码。

程序列表 7.3

```
void CADO1View::OnPrint(CDC*DC, CPrintInfo* pInfo)
{
    int         iRowCount = 1;          // Current print row count.
    LPSTR       lpstrRow ="EMPTY1";     // Text form of row count.
```

```
CString        cLine;          // One line of printed output.
CPen           oPen;           // Pen for drawing.
CBrush         oBrush;         // Brush for shading.
CFont          oTextFont;      // Font used for displaying text.
CFont          oHeadFont;      // Font used to display the heading.
CFont          oColFont;       // Font used to display column headings.
LOGFONT        lfFont;         // Font characteristic structure.
CSize          oFontSize;      // Size of a font.
COLORREF       clrRef;         // Colorstructure.
int            iRowPos = 120;  // Row position on printed page.
int            iTextHeight =0; // Current text height.
CRect          oDrawRect;      // Drawing area for printer.
int            iRecNumPos;     // Record number position.
int            iFoodIDPos;     // Food ID position.
int            iNamePos;       // Name position.
int            iPerishablePos; // Perishable position.
int            iPricePos;      // Price position.
int            iPurchasePos;   // Purchase Date position.
int            iQuantityPos;   // Quantity position.
```

```
int                iStoragePos;              // Storage Life position.

// Get the drawing area for our print routine.
ODrawRect = pInfo->m_rectDraw;

// Create a pen and select it into our device context.
clrRef = 0x00000000;
oPen.CreatePen(PS_SOLID, 2, clrRef);
pDC->SelectObject(&oPen);

// Create a brush and select it into our device context.
clrRef = 0x00C0C0C0;
oBrush.CreateSolidBrush(clrRef);
pDC->SelectObject(&oBrush);

// Create a heading font and select it into our device context.
oHeadFont.CreatePointFont(240, "Arial", pDC);
pDC->SelectObject(&oHeadFont);

// Display our heading.
oFontSize = pDC->GetOutputTextExtent("The ABC Company");
```

```
pDC->Ellipse(500,
        iRowPos - (oFontSize.cy / 2) - 10,
        oDrawRect.Width() - 500,
        iRowPos + (oFontSize.cy / 2) + 10);
pDC->SetBkMode(TRANSPARENT);
pDC->TextOut((oDrawRect.Width() - oFontSize.cx) / 2,
        iRowPos - (oFontSize.cy / 2) - 10,
        "The ABC Company");
pDC->SetBkMode(OPAQUE);

// Creat the appropriate space.
oHeadFont. GetLogFont (& lfFont);
iRowPos = abs (lfFont.lfHeight) + 175

// Create a text font.
oTextFont.CreatePointFont(120, "Arial", pDC);

// Get the current text font height.
oTextFont.GetLogFont(&lfFont);
iTextHeight = abs(lfFont.lfHeight) + 10;
```

```
// Create a font for displaying column headings.
lfFont.lfWeight = 700;          // Make it bold, mormal is 400.
oColFont.CreateFontIndirect(&lfFont);
pDC->SelectObject(&oColFont);

// Compute the column spacings. Set the first column to 1/2 inch.
iRecNumPos = int(oDrawRect.Width() / 17);
iFoodIDPos = iRecNumPos + 50 + pDC->GetOutputTextExtent("##").cx;
iNamePos = iFoodIDPos + 50 + pDC->GetOutputTextExtent("XXX00000XXX").cx;
iPerishablePos = iNamePos + 50 + pDC->GetOutputTextExtent("Xxxxxxxxxxxxx").cx;
iPricePos = iPerishablePos + 50 + pDC->GetOutputTextExtent("Perishable").cx;
iPurchasePos = iPricePos + 50 + pDC->GetOutputTextExtent("$00.00").cx;
iQuantityPos =iPurchasePos + 50 + pDC->GetOutputTextExtent("Purchase Date").cx;
iStoragePos = iQuantityPos + 50 + pDC->GetOutputTextExtent("Quantity").cx;

// Display the column headings.
pDC->TextOut(iRecNumPos, iRowPos,"#");
pDC->TextOut(iFoodIDPos, iRowPos,"Food ID");
pDC->TextOut(iNamePos, iRowPos,"Name");
pDC->TextOut(iPerishablePos, iRowPos,"Perishable");
pDC->TextOut(iPricePos, iRowPos, "Price");
```

```
pDC->TextOut(iPurchasePos, iRowPos, "Purchase Date");
pDC->TextOut(iQuantityPos, iRowPos, "Quantity");
pDC->TextOut(iStoragePos, iRowPos, "Storage Life");

// Create a space between the column heading and the text.
iRowPos += iTextHeight;
pDC->MoveTo(iRecNumPos, iRowPos);
pDC->LineTo(oDrawRect.Width() - iRecNumPos, iRowPos);
iRowPos += 20;

// Select our text font into the device context.
pDC->SelectObject(&oTextFont);

// Determine the row height.
iTextHeight = 20 + pDC->GetOutputTextExtent("Xy").cy;

// Print the records in a loop.
while (iRowCount<m_dataGrid1.GetVisibleRows())
{

//Display the current record number.
```

```
itoa(iRowCount,lpstrRow, 10);
cLine + lpstrRow;
pDC->TextOut(iRecNumPos, iRowPos, cLine);

// Print the data.
m_dataGrid1.SetRow(iRowCount - 1);
m_dataGrid1.SetCol(0);
pDC->TextOut(iFoodIDPos, iRowPos, m_dataGrid1.GetText());
m_dataGrid1.SetCol(1);
pDC->TextOut(iNamePos, iRowPos,m_dataGrid1.GetText());
m_dataGrid1.SetCol(2);
if (m_dataGrid1 GetText() == "-1")
	pDC->TextOut(iPerishablePos, iRowPos, "Yes");
else
	pDC->TextOut(iPerishablePos, iRowPos, "No");
m_dataGrid1.SetCol(6);
pDC->TextOut(iPricePos, iRowPos, "$" + (m_dataGrid1 GetText());
m_dataGrid1.SetCol(4);
pDC->TextOut(iPurchasePos, iRowPos, m_dataGrid1 GetText());
m_dataGrid1.SetCol(5) ;
pDC->TextOut(iQuantityPos, iRowPos, m_dataGrid1 GetText());
```

```
        m_dataGrid1.SetCol(3);
        pDC->TextOut(iStoragePos, iRowPos, m_dataGrid1 GetText());

        // Advance the row.
        iRowPos += iTextHeight;
        iRowCount ++;
    }

    // Perform the default action
    COleDBRecordView::OnPrint(pDC, pInfo);
}
```

这看上去代码可真不少，也确实如此。处理打印输出是件相当困难的工作。但是，如果将任务分成几个小块，就不难看出该怎么做了。

需要了解的第一件事是 CDC 类的神秘的 pDC 对象。这就是我在前面提到的设备上下文。如果把 pDC 看作是调色板，也就是内存中绘制想要发送给打印机的信息的地方，会很有帮助。事实上，Microsoft 已尽其所能在函数的命名和处理 GDI 的结构中孕育了这个观点。

我们应该做的第一件事（除了产生似乎有成千上万个变量之外）是创建一些绘图工具。如果没有必需的绘图工具，就不能在调色板上绘图。在这里，我要说明如何创建三个基本的绘图工具，你会在大多数程序中用到它：刷子、画笔和字体。画笔可以是任何颜色，任何宽度，你甚至可以选择像点这样的绘图

图案。用画笔可以画线，它也包括了对象的轮廓。例如，矩形不需要轮廓，但大多数其它对象都需要。你可以画出一个只用填充色填充的矩形。我们来看一看刷子。刷子为实线对象提供填充色。刷子还可以有特别的颜色，你可以选择创建带图案的刷子，如交叉阴影线。最后，字体用来在屏幕上写出一些东西 。它有很多的特性，但大多数特性对典型的程序员来说是比较陌生的。程序列表 7.3 中的源代码显示了创建字体的几种方式，使用它们并不要求全面的桌面排版知识。

注 Windows 支持三种基本的绘图工具：字体、画笔和刷子。

仅仅有了绘图工具并不表示就可以使用它。画家可能在墙上挂了一支笔，但并不表示就可以用它来画画。同样地，当你在 Windows 中绘图时，首先必须选择工具。这就是 SelectObject()函数要完成的任务。它让你为绘图工作选择一个对象。

注 在使用工具之前必须首先选择工具。

对 Windows 中的每个设备上下文来说，都有一个刷子、画笔和字体对象。可以把它看成是有三只手，每只手都以不同的工具装备起来。在使用另一种工具之前，必须把当前的工具放下，然后选择新的工具。Windows 自动为你处理放下工具的任务。每次调用 SelectObject()函数时，它就放下旧工具并捡起新的工具。

每一个工具在很多重要的方面都很独特。如果想用 Arial 10 点字体，就需要一种特殊的字体工具。如果需要该字体的粗体格式，或者突然想以 Times New Roman 12 字体显示文字，就必须创建另一种字体工具。幸运的是，不必为想要

使用的每一种颜色都创建分离的字体——这个特征由设备上下文来控制。说明工具的独特性的另一个示例是画笔。你要为想要的每一种宽度和风格的线创建画笔。刷子在有些方面显得更为独特。可以用相同的函数创建想要的任何字体或画笔，但对每种刷子来说，就要调用不同的函数。例如，如果你想创建交叉阴影线刷子，就要在代码中调用 CreateHatchBrush()函数而不是 CreateSolidBrush()函数。认识到现有工具的局限性并按需要创建新的工具是很重要的。

注释 本示例的绘图命令做了打印机优化，分辨率范围是每英寸 300 到 360 点。如果你的打印机分辨率更高的话，可能需要修改位置值以获得清晰的画面。但是，不论打印机的分辨率有多高，你都可以看到这些命令是如何相互配合、从而为数据库应用程序提供某种形式的输出的。

在绘图方面我们为本例做的第一件事是，用 Ellipse()函数创建一个椭圆。和大多数实线绘图命令一样，要指定图形左上角和右下角的坐标。Windows 假定你想使用当前选定的刷子和画笔，所以无需指定这些值。

绘制椭圆之后，代码会在其中放入一些文本。我们要做的第一件事是，把背景绘图模式设置成透明方式，这样我们就不用擦去椭圆的任何部分了。另一种模式是不透明的，意思是在前景中添加某种绘图的同时，要替换背景的颜色。现在看看代码是如何用 GetOutputTextExtent()函数来确定文本的长度的，其单位是像素。这大概是方便地确定一行文本行长度的唯一方式。确定了文本长度后，你就可以把这些文本在打印页面上居中放置了。下一行用 TextOut()函数向打印机发送文本。本节代码的最后一行用 SetBkMode()函数把背景绘图模式复位成不透明方式（缺省设置）。

现在讨论一下字体。创建字体有两种便利的方法，下一段中的代码对此做了说明。第一种方法是用 CreatePointFont()函数。你需要提供字体名称和想要创建的字体的大小，字体的单位是十分之一点。所以，如果你想创建 10 点的字体，就要指定字体大小值为 100。

字体使用特殊的 LOGFONT 结构传送它们所需的全部参数。如果你使用了 CreatePointFont()函数，Windows 就会为你创建这个结构，并在创建过程中设置一些缺省值。如果不想要这些缺省值时该怎么办呢？我们可以使用 GreatePointFont()创建简单的字体（就像我们刚做的那样），然后用 GetLogFont()函数填充 LOGFONT 结构。现在，你已经有了功能完整的结构，可以稍微做一些修改。我之所以强调“稍微”是因为，如果你试图在 Windows 刚刚返回的结构中做大的更改，那是搬起石头砸自己的脚。我把字体的磅数从正常改成粗体，只要字体磅数没有太大的差别，这个修改就是可以接受的。现在可以用 CreateFontIndirect()函数创建字体的粗体风格。下面的工作只需要把 Windows 提供的、修改了 IfWeight 成员的 LOGFONT 结构传送回去也就可以了。

请注意，一旦代码把字体选进了设备上下文，就用粗体字体创建了一组报表头。请始终谨记，首先要创建工具，然后根据在设备上下文上绘图的需要选择工具。我们还创建了一组变量，用来跟踪列的定位信息。两个报表头和细节信息使用了这些定位变量。请注意，用来导出定位信息的方法和设备无关。换言之，不论你是用 300 dpi 的打印机还是 600 dpi 的打印机，它都应该看上去差不多。当然，这里的随机成分是，我们用当前字体提供的文本所占用的空间量，Windows 到底安排得怎么样。在大多数情况下，你会发现这个算法是精确的，

但是要在公司里所有打印机上都打印出精美的报表，还是需要下一些功夫的。

下一步在标题下面绘制一条线段。请注意，绘制线段的过程分为两个步骤。第一，用 MoveTo()函数把画笔移到打印页上的特定点，然后用 LineTo()函数实际绘制这条线。可以把 MoveTo()看成是抬起画笔的移动，把 LineTo()看成是按下画笔的移动。

打印例程的最后一部分使用在 DataGrid1 中已有的数据。该代码完成的工作是在数据网格中的单元间移动，并在打印机或屏幕上打印其内容。那么，我们的应用程序的打印输出看起来是什么样的呢？下图就是个打印页面示例。

Untitled - ADO1

Print... | Next Page | Prev Page | Two Page | Zoom In | Zoom Out | Close

The ABC Company

#	Food ID	Name	Perishable	Price	Purchase Date	Quantity	Storage Life
1	VEG00001SPL	Carrots	Yes	$0.95	3/13/98	25	7
2	VEG00002SPL	Tomatoes	Yes	$0.81	3/13/98	45	7
3	VEG00002SPL	Tomatoes	Yes	$0.95	3/10/98	20	7
4	VEG00003SPL	Onions	No	$0.92	3/13/98	19	7
5	VEG00003SPL	Onions	No	$0.77	3/11/98	33	7
6	VEG00003SPL	Onions	No	$0.95	3/10/98	25	7

Page 1