Aplicações em Visual C++ com Acesso a Bases de Dados

Ambientes de Desenvolvimento Avançados

Engenharia Informática

Instituto Superior de Engenharia do Porto

José António Tavares

2002

Sem qualquer dúvida, as bases de dados são uma das mais populares aplicações computacionais. Virtualmente cada negócio utiliza bases de dados para se manter a par de tudo do que se passa na organização e no mercado em que actua, desde a sua lista de clientes à folha de pagamentos da companhia. Infelizmente, há muitos tipos diferentes de aplicações de base de dados, cada uma definindo as suas próprias disposições de ficheiros e regras. No passado, a programação de aplicações de base de dados era um pesadelo dado que era da responsabilidade do programador a resolução de todos os problemas de acesso aos diferentes tipos de ficheiros de bases de dados. Com a utilização do Visual de C++ esta tarefa está um tanto ou quanto simplificada uma vez que as classes MFC incluem já classes construídas em cima dos sistemas ODBC (*Open Database Connectivity*) e DAO (*Data Access Objects*). Outras tecnologias de base de dados da Microsoft estão a ganhar também suporte por parte das classes MFC.

O Visual C++ possui uma ferramenta, o AppWizard, que permite em função da aplicação desejada, através de parametrizações e selecção de opções pelo programador, gerar uma primeira e simples aplicação perfeitamente funcional sem que seja necessário gerar qualquer linha de código. O programador só tem de acrescentar o código específico para a aplicação que está a desenvolver. Mesmo nesta situação, o Visual C++ pode dar uma grande ajuda com a ferramenta ClassWizard. Acredite-se ou não, pela utilização do

AppWizard e da ClassWizard, é possível criar uma aplicação simples de base de dados sem escrever qualquer linha de código em C++. As tarefas mais complexas requerem alguma programação, mas não tanto quanto se possa pensar.

Este tutorial pretende dar uma introdução à programação com as classes relacionadas com o ODBC em Visual C++. Ao longo deste tutorial, será criada uma aplicação de base de dados que, além de mostrar alguns registos de uma base de dados, permite também actualiza-los, adicionar novos, remove-los, ordena-los e filtrar registos.

As Classes ODBC do Visual C++

Criar uma aplicação com acesso a uma base de dados pela utilização do AppWizard do Visual C++ resulta num programa que utiliza extensivamente várias classes MFC que incorporam e escondem toda a complexidade do ODBC. As classes mais importantes que importa conhecer são as classes **CDatabase**, **CRecordset**, e **CRecordView**. Outras classes úteis são as classes **CDBException**, **CFieldExchange**, **CLongBinary** e **CDBVariant**. Aconselha-lhe a consulta da ajuda (*Help*) com o MSDN Library Visual Studio, para a obtenção de mais informação acerca destas classes.

Com o AppWizard é possível gerar automaticamente todo o código necessário para criar um objecto da classe CDatabase. Este objecto representa a ligação entre a aplicação e a fonte de dados que se pretende aceder. Na maior parte dos casos, o uso da classe CDatabase num programa gerado com o AppWizard é transparente para o programador.

O AppWizard gera também o código necessário para criar um objecto da classe CRecordset para a aplicação. O objecto da classe CRecordset representa os dados reais que são seleccionados da fonte de dados, e os seus métodos manipulam os dados da base de dados.

Finalmente, o objecto da classe CRecordView representa a janela principal da aplicação tal como uma janela comum em Windows gerada com o AppWizart. No entanto, uma janela do tipo CRecordView é como uma caixa de diálogo utilizada para interacção com a informação da aplicação. Esta janela do tipo caixa de diálogo contem a ligação com o objecto da classe CRecordset da aplicação e permite navegar pela informação da base de dados, inserir, actualizar e apagar informação. Quando se cria uma aplicação pela primeira vez com o AppWizard, é da responsabilidade do programador adicionar controlos de edição à janela representada pelo objecto do tipo CRecordView. Estes controlos devem ser relacionados (*bind*) com os campos dos registos da base de dados de modo que a aplicação saiba onde deve apresentar a informação que se quer mostrar.

Nas secções seguintes, será mostrado como é que as diferentes classes de bases de dados se encaixam para construir uma aplicação.

Classe CDatabase

Um objecto da classe **CDatabase** representa uma conexão a uma fonte de dados, através da qual é possível operar sobre essa fonte de dados. Uma fonte de dados é uma instância específica dos dados hospedados por alguns exemplos de sistemas de gestão de base de dados (DBMS). Que incluem o Microsoft SQL Server, Microsoft Access, Borland® dBASE®, e xBASE. É possível ter um ou mais objectos da classe **CDatabase** activos em qualquer momento na aplicação.

Para usar a classe **CDatabase**, deve-se construir um objecto da **CDatabase** e invocar o seu método Open ou OpenEx de modo a efectuar a abertura de uma conexão. Quando então se criam objectos da classe **CRecordset** para se operar com a fonte de dados, deve-se passar ao construtor do *recordset* um ponteiro para o objecto da classe **CDatabase** anteriormente construída que representa a ligação à fonte de dados. Quando o uso da conexão termina, o método **Close** deve ser invocado antes de se destruir o respectivo objecto da classe **CDatabase**. Este método fecha todos os *recordsets* que não foram previamente fechados.

Classe CRecordset

Um objecto da classe **CRecordset** representa um conjunto de registos seleccionados de uma fonte de dados. Conhecidos como "*recordsets*" os objectos do tipo **CRecordset** são usados tipicamente sob duas formas: *dynasets* e *snapshots*. Um *dynaset* permanece sincronizado com as actualizações (*updates*) dos dados feitas por outros utilizadores. Um *snapshot* é uma imagem estática dos dados. Cada forma representa um conjunto de registos tal como existem na fonte de dados no momento em que o *recordset* é aberto. Quando se efectua um *scroll* para um dado registo considerando a forma dynaset, este reflecte as mudanças feitas subsequentemente ao registo, tanto por outros utilizadores como por outros *recordsets* de aplicação, o que não acontece sob a forma de *snapshot*.

Para trabalhar com qualquer uma das formas de *recordset*, é comum derivar uma classe *recordset* específica para a aplicação a partir da classe **CRecordset**. Os *recordsets* seleccionam registos de uma fonte de dados, e então é possível:

- Realizar o *scroll* através dos registos;
- Actualizar os registos e especificar o modo de *lock*;

- Filtrar o *recordset* para restringir quais os registos são seleccionados a partir dos disponíveis na fonte de dados;
- Ordenar o *recordset*.
- Parametrizar o *recordset* para adaptar a sua selecção com informação desconhecida à partida.

Para usar a classe **CRecordset**, basta abrir uma base de dados e construir um objecto do tipo *recordset*, passando ao construtor um ponteiro para o objecto da classe **CDatabase**. Será então invocada o método **Open** do objecto *recordset*, onde é possível especificar se o objecto é da forma *dynaset* ou da forma *snapshot*. Invocar **Open** selecciona dados da fonte de dados. Depois de o objecto *recordset* estar aberto, será possível usar os seus métodos para efectuar o *scroll* através dos registos e operar sobre eles. As operações disponíveis dependem se o objecto é da forma *dynaset* ou da forma *snapshot* e se é actualizável ou apenas de leitura. Para refrescar os registos que podem ter sido alterados ou adicionados desde a invocação de **Open**, deve ser invocado o método **Requery**. O método **Close** deve ser também invocado antes de destruir o correspondente objecto *recordset* quando este não é mais necessário.

Na classe derivada de **CRecordset**, *record field exchange* (RFX) ou *bulk record field exchange* (Bulk RFX) são usados para suportar a leitura e actualização dos campos dos registos.

Todos estes conceitos se tornarão mais claros à medida que o leitor for seguindo este toturial.

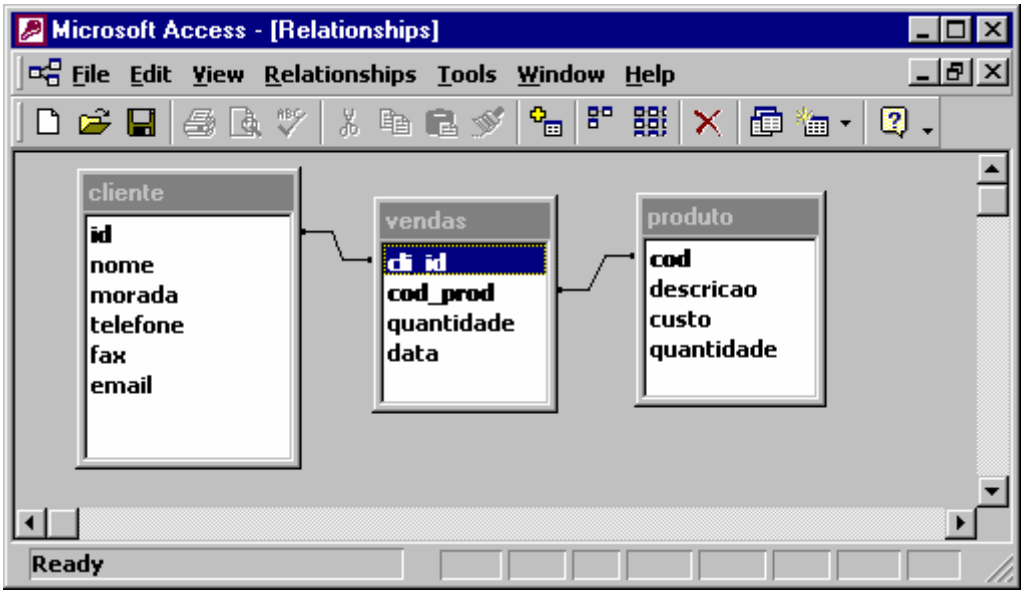
A Base de Dados Utilizada neste Tutorial

Este tutorial mostra como pode ser feito o desenvolvimento de aplicações usando MFC com acesso a bases de dados. As tabelas da base de dados usado como exemplo são as que se indicam na figura seguinte. Como se pode observar esta base de dados armazena informação acerca das vendas realizadas de uma hipotética empresa.

Embora o processo de criação de um simples programa de base de dados ODBC com o Visual C++ seja relativamente fácil, devem ser completados alguns passos que se indicam a seguir:

1. Registar a base de dados com o sistema;
2. Usar a AppWizard para criar uma aplicação de base de dados básica;
3. Adicionar código à aplicação básica para implementar funcionalidades não suportadas automaticamente pela AppWizard;

Nas seguintes secções, será mostrado como se podem executar estas etapas através da escrita de uma aplicação exemplo que permite adicionar, remover, actualizar, ordenar, visualizar registos de uma tabela de empregados de uma base de dados de uma loja.



Registar a Base de Dados

Antes que se possa criar uma aplicação de base de dados, deve-se registar a base de dados que se pretende aceder como fonte de dados de modo a ser acessível através do *driver* ODBC. Para realizar esta tarefa devem ser executadas as etapas seguintes:

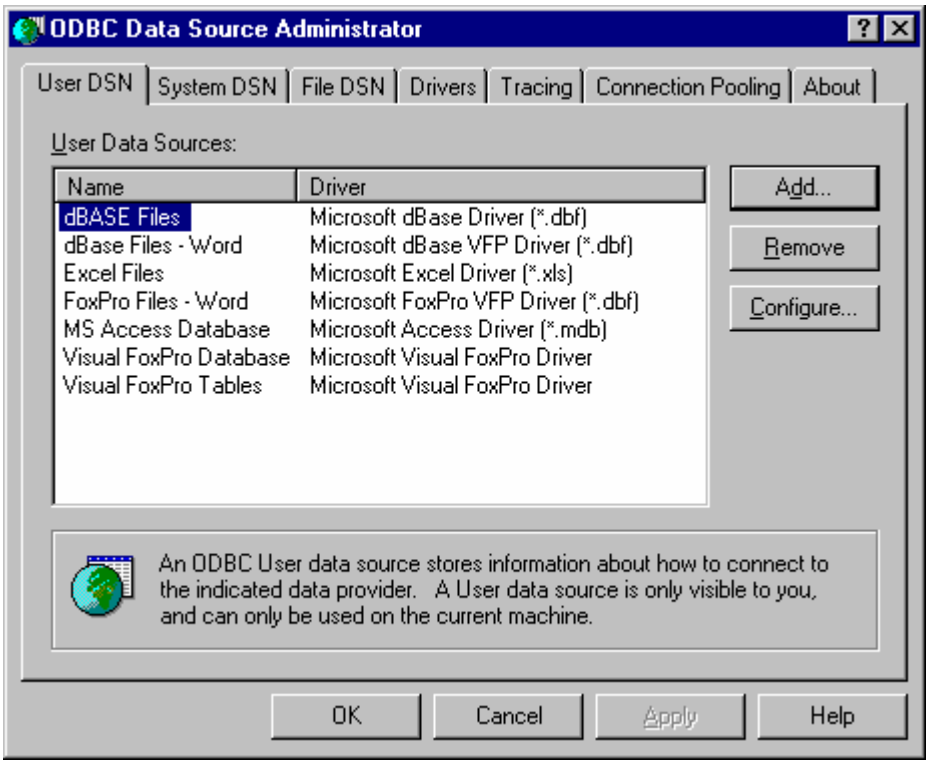
1. Criar uma pasta no disco com um nome da sua escolha (por exemplo *Database*) onde deverá colocar o(s) ficheiro(s) de base da sua base de dados. Se ainda não criou a base de dados deve cria-la com uma aplicação de desenvolvimento de base de dados como por exemplo o MS Access;

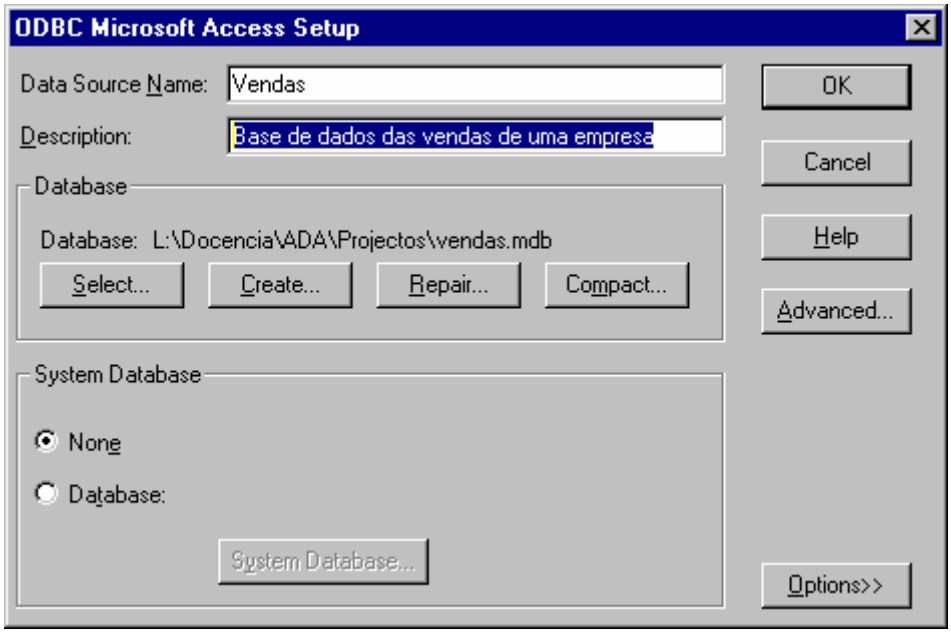
2. Considere o ficheiro vendas.mdb, gerado com o MS Access como sendo aquele que contem a base de dados. Esta base de dados será usada como fonte de dados para as aplicações a desenvolver;

3. Do menu Start do Windows, seleccionar Settings e depois Control Panel. Quando a caixa de diálogo do Control Panel aparece, faça um duplo clique no icon 32-Bit ODBC. A caixa de diálogo ODBC Data Source Administrator aparece, como mostra a figura seguinte:

4. Seleccione o botão Add para fazer aparecer a caixa de diálogo Create New Data Source. Seleccione Microsoft Access Driver da lista de *drivers* e clique em Finish. Este driver ODBC associado com a fonte de dados criada (vendas.mdb);

5. Quando a caixa de diálogo ODBC Microsoft Access 97 Setup aparecer introduza **Vendas** na caixa de texto Data Source Name e **Base de dados das vendas de uma empresa** na caixa de texto Description como mostra a figura abaixo. Data Source Name é um meio de identificar a fonte de dados criada enquanto que o campo Description permite incluir mais informação acerca da fonte de dados;





6. Seleccione ainda o botão Select para seleccionar o ficheiro de base de dados. Use a caixa de diálogo que surge para localizar e seleccionar o ficheiro vendas.mdb. Para terminar deve seleccionar sucessivamente os botões OK das caixas de diálogo abertas e finalmente fechar o Control Panel;

O sistema deve então estar agora configurado de modo a ser possível aceder ao ficheiro de base de dados vendas.mdb com o *driver* ODBC do MS Access.

Implementação de uma Pequena Aplicação de Base de Dados para a Consola

Nesta secção mostra-se como se pode criar uma pequena aplicação com acesso a uma base de dados para a consola (janela DOS) utilizando classes MFC.

O primeiro passo para criar uma destas aplicações passa por se criar um novo projecto com o Visual C++ Studio. O projecto a criar deve estar inicialmente vazio, sendo que são posteriormente adicionados os respectivos ficheiros de código. Para a aplicação que se irá construir são necessários três ficheiros:

1. ClientesRs.h
2. ClientesRs.cpp
3. CDB_App.cpp

Nota: Para que se possa compilar o código usando as classes MFC ODBC de uma dada aplicação para a consola é necessário especificar nas configurações do projecto que se pretende uma aplicação multi-thread. Para isso é necessário escolher o comando “Settings” do menu “Project”. De seguida deve-se escolher o separador “C/C++”. Na categoria “Code Generation” escolher a “Run-Time Library” adequada.

O código desta simples aplicação é apresentado abaixo. Começando por se analisar a função **main** verifica-se que são declarados dois objectos. O primeiro é da classe **CDatabase** e o segundo deriva da classe **CRecordset**. O motivo pela qual se deriva a classe CRecordset prende-se com a necessidade de adaptar esta classe para a tabela da base de dados que se pretende aceder. É, no entanto, possível criar e usar objectos da classe CRecordset directamente, contudo a abordagem seguida torna-se mais simples e natural de implementar. Neste caso, tal como na generalidade das aplicações que acedem a base de dados, a sequência de passos seguida consiste em:

1. Criar um objecto da classe CDatabase que representa a nossa base de dados;

2. É invocado o método “Open” do objecto criado da classe CDatabase de modo a estabelecer a ligação com a base de dados (os significado dos argumentos é apresentado em anexo);
3. É criado um objecto *recordset* considerando o objecto da classe CDatabase anteriormente criado;
4. O objecto *recordset* é aberto através da invocação do seu método “Open” de modo a ser possível aceder á tabela da base de dados que representa;
5. Os dados representados pelo *recordset* são processados;
6. O processamento terminou e, portanto, o objecto *recordset* é fechado pela invocação do método “Close”;
7. A ligação não é mais necessária e, deste modo, é invocado o método “Close” do objecto da classe CDatabase para fechar a ligação com a base de dados.

Neste ponto é necessário compreender como é constituída a classe CClienteRs derivada da classe CRecordset e que adapta o *recordset* para o caso particular da tabela de clientes. Esta classe derivada foi implementada da seguinte forma:

1. Foram adicionados membros de dados de acordo com os campos da tabela. Estes membros de dados são de acesso publico, ou seja, directamente acessíveis fora da classe;
2. Foi implementado um construtor que realiza todas as inicializações dos objectos da classe;
3. É redefinido o método “GetDefaultSQL”. Neste caso particular o método apenas retorna um objecto da classe CString que contem o nome da tabela cliente que a efectuar o acesso. Com esta *string* o *recordset* constrói todos os comandos SQL que necessita para realizar as operações pretendidas. Em geral este método deve devolver uma *string* com o nome de uma tabela ou um comando SQL SELECT.
4. Por fim, é redefinido o método “DoFieldExchange”. A redefinição deste método destina-se a efectuar o mapeamento (*Binding*) entre os membros de dados do objecto *recordset* com os campos da tabela.

As operações efectivamente realizadas à base de dados são implementadas com a função “*process*”. Essencialmente, esta rotina permite navegar pelos registos da tabela, inserir, modificar e remover registos. Como exercício sugere-se a interpretação e compreensão do código desta função que implementa as operações referidas.

ClienteRs.h

```
#include <iostream.h>

class CClienteRs : public CRecordset
{
	friend ostream& operator << ( ostream& os, const CClienteRs& cli );
	friend istream& operator >> ( istream& is, CClienteRs& cli );

public:
	enum { MAXSTRLEN = 50 };

	long int m_iID;
	CString  m_sNome;
	CString  m_sMorada;
	CString  m_sTelefone;
	CString  m_sFax;
	CString  m_sEMail;

	CClienteRs( CDatabase* pDatabase = NULL );
	~CClienteRs();

	virtual CString GetDefaultSQL();
	virtual void DoFieldExchange( CFieldExchange* pFX );  // RFX support
};

ostream& operator << ( ostream& os, const CClienteRs& cli );
istream& operator >> ( istream& is, CClienteRs& cli );
```

ClienteRs.cpp

```
#include <afxdb.h>
#include <iostream.h>
#include "ClienteRs.h"

CClienteRs :: CClienteRs( CDatabase* pDatabase )
	: CRecordset( pDatabase )
{
	m_iID = 0;
	m_sNome = _T("");
```

```
	m_sMorada = _T("");
	m_sTelefone = _T("");
	m_sFax = _T("");
	m_sEMail = _T("");
	m_nFields = 6;
}

CClienteRs :: ~CClienteRs()
{
}

CString CClienteRs :: GetDefaultSQL()
{
	return _T("[cliente]");
}

void CClienteRs :: DoFieldExchange( CFieldExchange* pFX )
{
	pFX->SetFieldType( CFieldExchange :: outputColumn );

	RFX_Long( pFX, _T("[id]"),       m_iID);
	RFX_Text( pFX, _T("[nome]"),     m_sNome);
	RFX_Text( pFX, _T("[morada]"),   m_sMorada);
	RFX_Text( pFX, _T("[telefone]"), m_sTelefone);
	RFX_Text( pFX, _T("[fax]"),      m_sFax);
	RFX_Text( pFX, _T("[email]"),    m_sEMail);
}

ostream& operator << ( ostream& os, const CClienteRs& cli )
{
	cout << "-----------------------------------" << endl;
	cout << "ID\t\t"     << cli.m_iID       << endl;
	cout << "Nome\t\t"   << cli.m_sNome     << endl;
	cout << "Morada\t\t" << cli.m_sMorada   << endl;
	cout << "Telefone\t" << cli.m_sTelefone << endl;
	cout << "Fax\t\t"    << cli.m_sFax      << endl;
	cout << "E-Mail\t\t" << cli.m_sEMail    << endl << endl;

	return(os);
}
```

```
istream& operator >> ( istream& is, CClienteRs& cli )
{
	char str[ CClienteRs::MAXSTRLEN + 1 ];

	cout << "-----------------------------------" << endl;
	//  o ID e automatico
	//	cout << "ID\t\t"	<< cli.m_iID	<< endl;

	cout << "Nome\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	cli.m_sNome	= str;

	cout << "Morada\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	cli.m_sMorada	= str;

	cout << "Telefone\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	cli.m_sTelefone = str;

	cout << "Fax\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	cli.m_sFax	= str;

	cout << "E-Mail\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	cli.m_sEMail	= str;

	return(is);
}
```

CDB_App.cpp

```
#include <afxdb.h>
#include "ClienteRs.h"
#include <iostream.h>
```

```
char getOption( char *options )
{
	char dummy[10];
	cout << options << " -> ";
	cin.getline( dummy, 10 );
	return dummy[0];
}

void modify( CClienteRs& cli )
{
	cli.Edit();

	char str[ CClienteRs::MAXSTRLEN + 1 ];

	cout << "-----------------------------------" << endl;

	cout << "Nome\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	if ( *str != '\0' ) cli.m_sNome = str;

	cout << "Morada\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	if ( *str != '\0' ) cli.m_sMorada = str;

	cout << "Telefone\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	if ( *str != '\0' ) cli.m_sTelefone = str;

	cout << "Fax\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	if ( *str != '\0' ) cli.m_sFax = str;

	cout << "E-Mail\t\t";
	cin.getline( str, CClienteRs::MAXSTRLEN+1 );
	if ( *str != '\0' ) cli.m_sEMail = str;

	cli.Update();
}

void process( CClienteRs& rs )
{
```

```
char op = '\0';

while ( op != 'q' )
{
	if ( !rs.IsEOF() )
		cout << rs;

	op = getOption(
		"(Q)uit (N)ext (P)rev (F)irst (L)ast (D)elete (A)dd (C)hange" );

	switch( op ) {
	case 'n' : // Move para o proximo registo
		rs.MoveNext();
		if ( rs.IsEOF() ) rs.MovePrev();
		break;
	case 'p' : // Move para o registo anterior
		rs.MovePrev();
		if ( rs.IsBOF() ) rs.MoveNext();
		break;
	case 'f' : // Move para o primeiro registo
		rs.MoveFirst();
		break;
	case 'l' : // Move para o ultimo registo
		rs.MoveLast();
		break;
	case 'd' : // Remove o registo actual
		try
		{
			rs.Delete();
			rs.MoveNext();
			if ( rs.IsEOF() )
				rs.MoveLast();
			if ( rs.IsBOF() )
				rs.SetFieldNull(NULL);
		}
		catch ( CDBException e )
		{
			cerr << e.m_strError << endl;
		}
		break;
	case 'a' : // Adiciona um novo registo
```

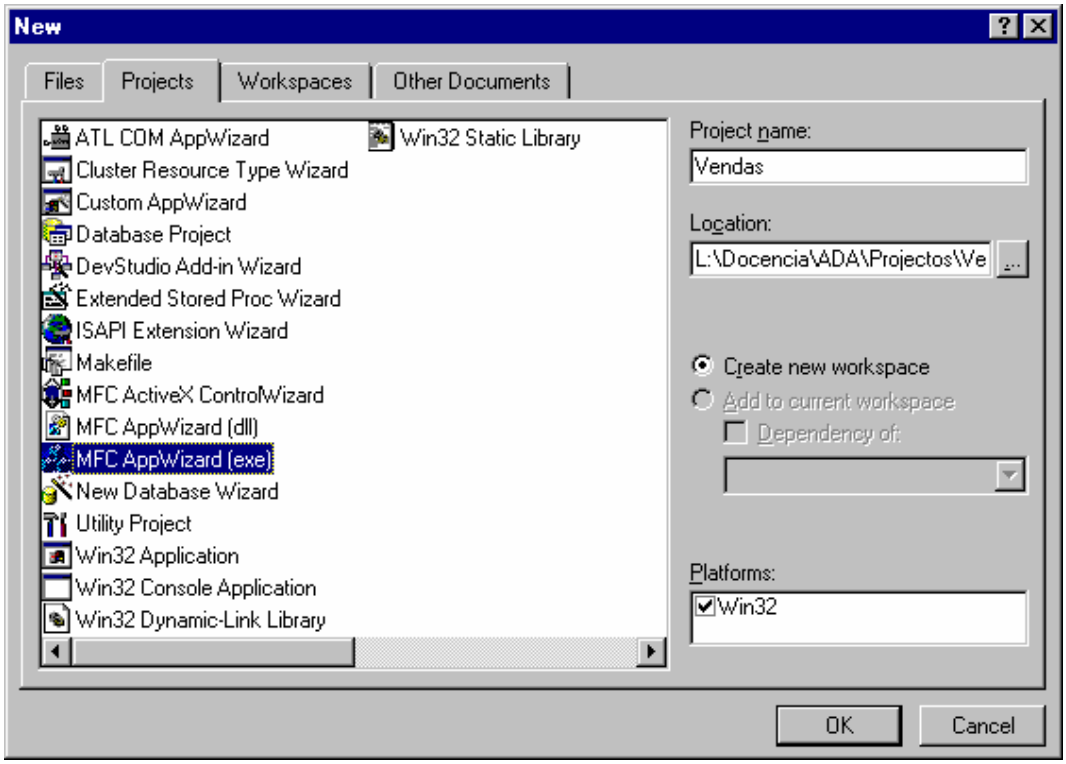
```
                        try
                        {
                                rs.AddNew();
                                cin >> rs;
                                rs.Update();
                                rs.MoveLast();
                        }
                        catch ( CDBException e )
                        {
                                cerr << e.m_strError << endl;
                        }
                        break;
                case 'c' : // Modifica o registo actual
                        try
                        {
                                modify( rs );
                        }
                        catch ( CDBException e )
                        {
                                cerr << e.m_strError << endl;
                        }
                        break;
                }
        }
}

void main()
{
        CDatabase db;
        db.Open( _T("Vendas"), FALSE, FALSE, "ODBC;", FALSE );
        // Alternativa ->
        //      db.OpenEx( _T("DSN=mydb"), CDatabase::noOdbcDialog );

        CClienteRs rsCli( &db );
        rsCli.Open( CRecordset::dynaset );

        process( rsCli );

        rsCli.Close();
        db.Close();
}
```

Implementação de uma Pequena Aplicação de Base de Dados para o Windows

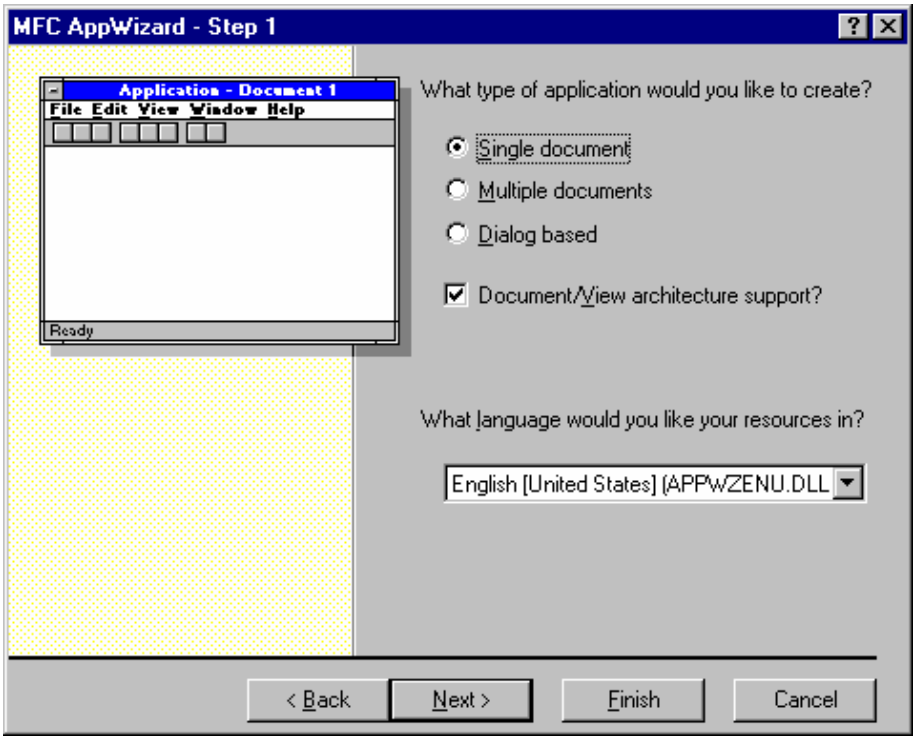
Criação da Aplicação Básica Inicial

A criação de uma aplicação básica inicial para gerir as vendas de uma empresa é um processo que apenas requer a escrita de algumas linhas de código, bastando apenas recorrer ao AppWizard e ou ClassWizard para a geração da maior parte do código. O processo baseia-se nos passos que a seguir se indicam. Como resultado final obtém-se uma aplicação capaz de aceder e visualizar as tabelas da base.

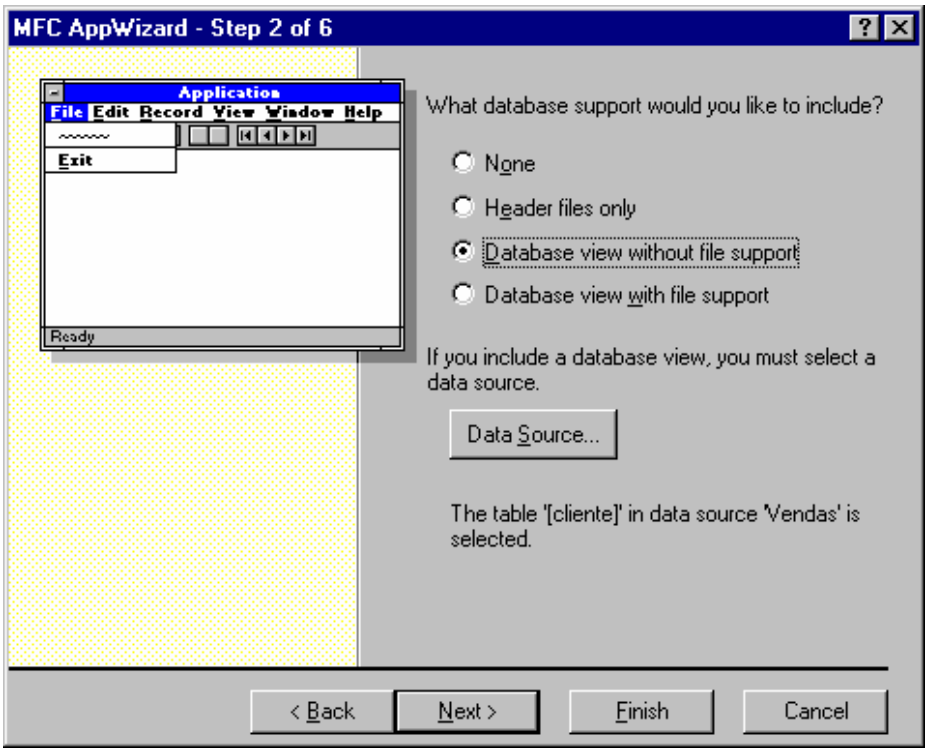
1. Seleccionar File, New da barra de menus do Developer Studio. Seleccionar o separador Projects;
2. Seleccionar uma aplicação MFC AppWizard (exe) e escrever **Vendas** na caixa Project Name, como mostra a figura seguinte e clicar em OK após ter seleccionado a pasta para os ficheiros do projecto;



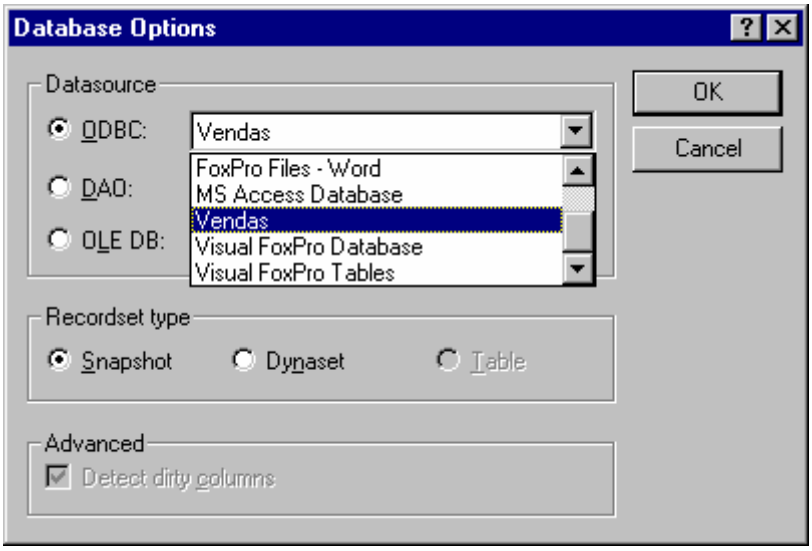
3. Na caixa de diálogo Step 1 seleccionar Single Document, como mostra a figura seguinte, para assegurar que a aplicação Vendas não abre mais que uma janela de cada vez. Clique em Next;



4. Seleccionar a opção Database View Without File Support, como mostra a figura seguinte, de forma a que o AppWizard gere as classes necessárias para aceder ao conteúdo da base de dados. Seleccionar também o botão Data Source para efectuar a ligação da aplicação à fonte de dados;



5. Na caixa de diálogo Database Options seleccionar da lista ODBC a fonte de dados Vendas como mostra a figura seguinte;



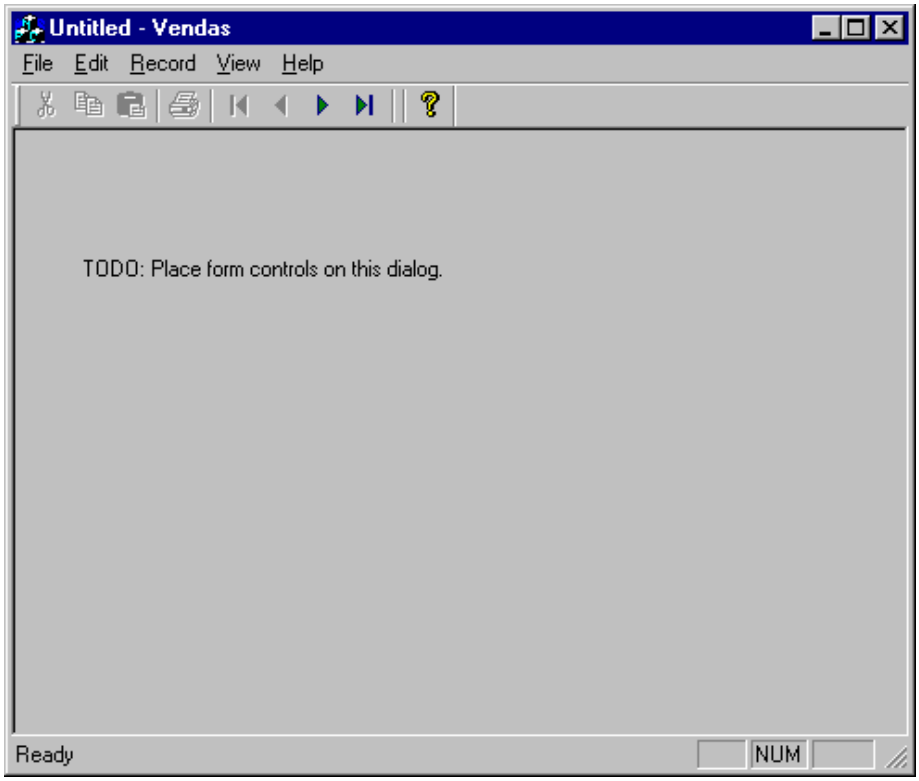
6. Na caixa de diálogo Select Database Tables seleccionar uma das tabelas indicadas na lista (opte, por exemplo, pela tabela vendas), e clique OK. Com isto, as tabelas da base de dados **Vendas** foram associadas com aplicação. Após esta operação deve seguir para passo seguinte;

7. No passo “Step 3” aceite as opções por defeito e avance para o próximo passo;

8. Na caixa de diálogo do passo “Step 4” desligue a opção “Printing and Print Preview” e avance para o próximo passo;

9. Aceite também as opções por defeito do passo “Step 5” ao clicar no botão Next. No passo “Step 6”, clique no botão Finish para finalizar as suas selecções para a sua aplicação vendas. Com isto surge a caixa de diálogo “New Project Information” onde deve clicar no botão OK para o AppWizard criar a aplicação básica de vendas.

Após ter concluído todos estes passos, pode compilar a aplicação. Para compilar deve seleccionar o botão Build da barra de ferramentas do Developer Studio, o comando Build do menu Build, ou pressionando na tecla F7 do teclado. Depois da compilação da aplicação pode correr o programa através das teclas Ctrl+F5. Como resultado surge a janela da aplicação gerada como a da figura seguinte. É possível usar o botões da barra de ferramentas para navegar pelos registos. No entanto, nada aparece na janela relacionado com o conteúdo da base de dados uma vez que é necessário associar os campos dos registos da base de dados com controlo visuais (caixas de texto, por exemplo).



Antes de prosseguir convém verificar quais são as classes geradas pelo o AppWizard. Estas foram as seguintes:

- CVendasApp – representa a classe relativa à aplicação. A criação de um objecto desta classe é equivalente a lançar esta aplicação;
- CMainFrame – constitui a classe a partir da qual é criado um objecto que é a face visível da aplicação (janela);
- CVendasDoc – é a classe que representa o documento manipulado na aplicação. Não esquecer que foi gerada uma aplicação com suporte "Documet/View";
- CVendasView – é a classe de objectos que permite a visualização de documentos dentro da janela da aplicação (CmainFrame);
- CVendasSet – é uma classe derivada de CRecordset adaptada para a tabela Vendas da base de dados acedida pela a aplicação;
- CAboutDlg – é a classe que representa a caixa de diálogo *about* típica das aplicações Windows.

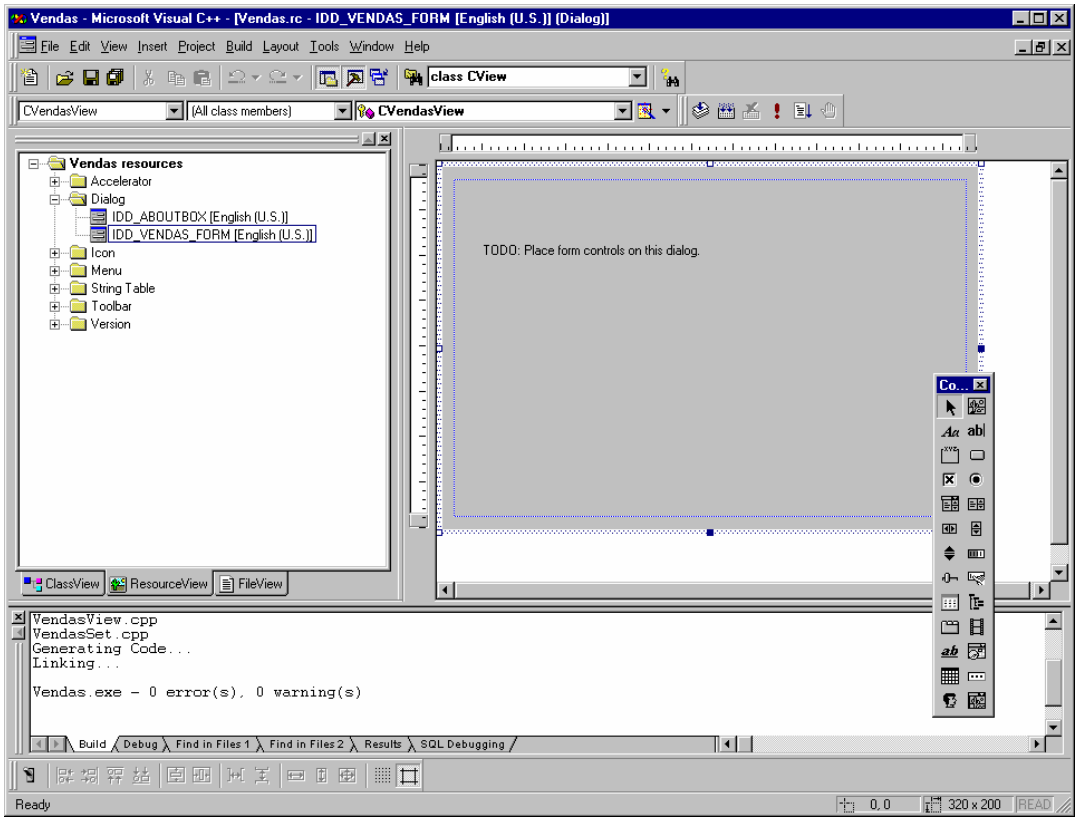
Convém referir que a classe CVendasSet contém um método novo ainda não apresentado neste documento, e que constitui uma redefinição do seu equivalente da classe base CRecordset. O seu código é o seguinte:

```
CString CVendasSet::GetDefaultConnect()
{
        return _T("ODBC;DSN=Vendas");
}
```

Dado que o *recordset* é criado sem que se lhe indique um objecto da classe CDatabase específico, este invoca este método para obter informação de modo a criar um objecto do tipo CDatabase que representará a ligação à fonte de dados associada ao *recordset*.

Criar o "*Display*" da Base de Dados

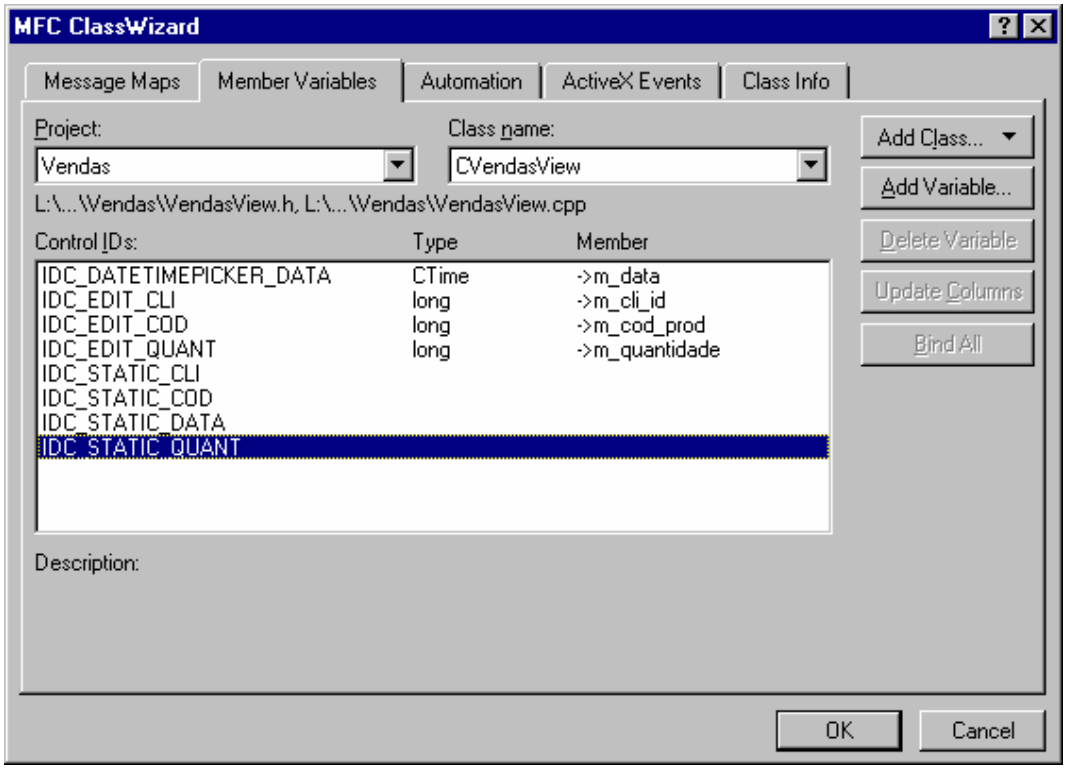
O próximo passo na criação da aplicação de base de dados Vendas consiste em modificar o formulário que mostra os dados na janela da aplicação. Dado que este formulário é um tipo especial de caixa de diálogo, é relativamente simples modifica-lo com editor de recursos do Developer Studio. Este pode ser acedido ao efectuar a selecção do separador Resource View, na janela workspace, de modo a mostrar os recursos da aplicação. O passo seguinte passa por se seleccionar a pasta "Dialog" e de seguida seleccionar com um duplo clique a caixa de diálogo IDD_VENDAS_FORM.



De seguida deve adicionar à caixa de diálogo os controlos indicados na tabela seguinte:

Tipo	ID	*Caption*	Propriedades
Texto Estático	IDC_STATIC_CLI	Cliente ID	
Texto Estático	IDC_STATIC_COD	Código Produto	
Texto Estático	IDC_STATIC_QUANT	Quantidade	
Texto Estático	IDC_STATIC_DATA	Data	
Edição de Texto	IDC_EDIT_CLI	-	*ReadOnly*
Edição de Texto	IDC_EDIT_COD	-	*ReadOnly*
Edição de Texto	IDC_EDIT_QUANT	-	*ReadOnly*
Date Time Picker	IDC_DATETIMEPICKER_DATA	-	*Disable*

Os controlos adicionados devem então ser associados com os respectivos membros de dados do *recordset*. De referir que o objecto *recordset* é um membro de dados do objecto do tipo CRecordView que tem como função mostrar a informação do recordset na janela da aplicação. Para efectuar esta associação devemos abrir a ClassWizard no menu View e seleccionar de seguida o separador "Member Variables". Depois para cada um dos controlos adiciona-se uma variável clicando no botão "Add Variable" que abre uma nova caixa de diálogo, sendo aqui possível escolher variável do *recordset* a associar ao respectivo controlo (ver figura seguinte).



Fechando o ClassWizard confirmado as escolhas com o botão OK é gerado todo o código que implementa a associação pretendida. Este código é colocado no método "DoDataExchange" da classe CVendasView que é uma classe derivada da classe CRecordView e que se indica a seguir.

```
void CVendasView::DoDataExchange(CDataExchange* pDX)
{
	CRecordView::DoDataExchange(pDX);

	//{{AFX_DATA_MAP(CVendasView)
	DDX_FieldText(pDX, IDC_EDIT_CLI, m_pSet->m_cli_id, m_pSet);
	DDX_FieldText(pDX, IDC_EDIT_COD, m_pSet->m_cod_prod, m_pSet);
	DDX_FieldText(pDX, IDC_EDIT_QUANT, m_pSet->m_quantidade, m_pSet);
	DDX_FieldDateTimeCtrl(pDX, IDC_DATETIMEPICKER_DATA, m_pSet->m_data, m_pSet);
	//}}AFX_DATA_MAP
}
```

De notar que a função DDX_FieldDateTimeCtrl apresentada não existe quando se consideram objectos *recordsets* e, portanto, é necessário implementar uma específica para este caso em particular. Uma forma possível é a que a seguir se indica.

```
void DDX_FieldDateTimeCtrl( CDataExchange* pDX, int nIDC, CTime& value, CRecordset* pRecordset )
{
	ASSERT_VALID(pRecordset);
	DDX_DateTimeCtrl(pDX, nIDC, value);
}
```

Adicionar, Modificar e Remover Registos

É chegado o momento de implementar as funcionalidades que permitem adicionar, modificar e remover registos. Não é objectivo aqui indicar qual o código completo necessário para estas operações. Parte desta funcionalidade já se encontra ilustrada no código da aplicação de acesso a uma base de dados para a consola anteriormente apresentado. Pretende-se aqui apenas fornecer pistas para que estas funcionalidades sejam implementadas.

O primeiro passo passa por se adicionar os comandos *Add*, *Modify* e *Delete* no menu *Record* com o editor de recursos. O ID para cada um destes comando do menu deve ser ID_RECORD_ADD, ID_RECORD_MODIFY, ID_RECORD_DELETE respectivamente.

Estes comandos podem ser ligados com botões na barra de ferramentas (*toolbar*) da aplicação, sendo este processo realizado também com o editor de recursos. O próximo passo consiste em criar os métodos que tratam a escolha de um destes três comandos. Para isso é necessário abrir a ClassWizard e seleccionar o separador “Message Maps”. Seleccionado a classe CVendasView e para cada ID destes três comandos deve-se efectuar o duplo clique em COMMAND na caixa “Messages”. Esta operação resulta na adição à classe CVendasView dos métodos OnRecordAdd, OnRecordModify e OnRecordDelete que serão invocados sempre o respectivo comando é seleccionado. Este métodos não contêm à partida qualquer código que deverá ser inserido manualmente. O código de cada um destes métodos, para esta aplicação em particular vai necessitar de uma variável booleana m_bEdit como membro de dados da classe protegido CvendasView que deverá ser inicializada no respectivo construtor com o valor *false*. Esta variável deve ser adicionada manualmente.

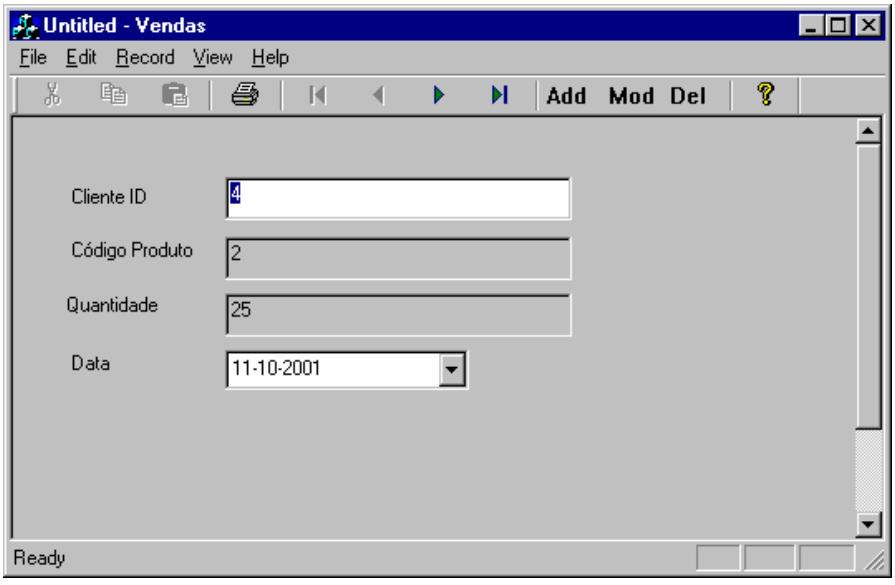
De modo a ilustrar a possibilidade de permitir ou inibir os controlos para a edição apresenta-se aqui apenas o código que permite modificar o campo do código de cliente e o campo da data de venda relativa ao registo seleccionado. Fica como exercício a implementação dos outros dois métodos e a implementação deste método de uma forma mais completa. O código que permite implementar esta funcionalidade é o seguinte:

```
void CVendasView::OnRecordModify()
{
   m_bEdit = true;
   m_pSet->Edit();

   CEdit* pCtrlCli = (CEdit*)GetDlgItem(IDC_EDIT_CLI);
   pCtrlCli->SetReadOnly(false);
   CDateTimeCtrl * pCtrlDTP = (CDateTimeCtrl *)GetDlgItem(IDC_DATETIMEPICKER_DATA);
   pCtrlDTP->EnableWindow(true);

   UpdateData(false);
}
```

O que este método faz é a activação do modo de edição permitindo que se possa alterar os dados dos controlos pretendidos. A invocação do método UpdateData com o valor *false* como parâmetro obriga a que os controlos sejam inicializados de forma conveniente. No momento em faz a deslocação para outro registo é necessário desactivar o modo de edição. Isto é feito com o método OnMove que deverá ser criado usando a ClassWizard. Para o “Object IDs” CVendasView, na caixa “Messages”fazer, duplo clique em OnMove.

```
bool CVendasView::OnMove(UINT nIDMoveCommand)
{
   if (m_bEdit)   {
      m_bEdit = false;
      UpdateData(true); // force to retreive data from controls
      if (m_pSet->CanUpdate())
            m_pSet->Update(); // Update the recordset object
      m_pSet->Requery();
      UpdateData(false); // force the initialization of controls

      // disable controls for editing
      CEdit* pCtrlCli = (CEdit*)GetDlgItem(IDC_EDIT_CLI);
      pCtrlCli->SetReadOnly(true);
      CDateTimeCtrl * pCtrlDTP = (CDateTimeCtrl *)GetDlgItem(IDC_DATETIMEPICKER_DATA);
      pCtrlDTP->EnableWindow(false);
   }
   return CRecordView::OnMove(nIDMoveCommand);
}
```

Este método OnMove é invocado sempre que o utilizador dá um comando de navegação entre registo do *recordset*. Por defeito todo trabalho é feito na classe CRecordView, no entanto, neste caso particular pretende-se realizar algo mais para além do que feito na classe CRecordView. Deste modo foi redefinido este método na classe derivada CVendasView de modo a permitir o trabalho adicional necessário, embora se invoque o método OnMove na sua classe base para efectuar o restante trabalho por defeito. Na figura seguinte mostra a janela da aplicação com os controlos que foram adicionados, estando esta em modo de edição.



Ordenação e Filtragem

Em muitos casos quando se esta a aceder a uma base de dados, pretende-se alterar a ordem com que os registos são apresentados, ou mesmo pretende-se procurar pelos registos que obedecem a determinados critérios. As classes MFC que encapsulam a API do ODBC para acesso a bases de dados possuem métodos que permitem ordenar os registos em função de um dado campo. Existem ainda métodos que permitem limitar os registos visualizados a um subconjunto de acordo uma dada informação, como um nome específico ou um dado código. Esta última operação é conhecida por filtragem. Esta secção mostra como adicionar a ordenação e a filtragem na aplicação exemplo.

Ordenação

Começando em primeiro lugar com a ordenação dos registos, vai-se adicionar a possibilidade de utilizador por visualizar os registos por ordem de código de cliente ou por ordem de código de produto. Para o efeito deve-se adicionar os seguintes controlos à caixa de diálogo IDC_VENDAS_FORM com o editor de recursos:

Tipo	ID	*Caption*
Caixa de grupo	IDC_STATIC_SORT	Ordenação de registos por:
Botão de rádio	IDC_RADIO_CLI	Código de clientes
Botão de rádio	IDC_RADIO_COD	Código de produtos

Para os controlos com IDC_RADIO_CLI e IDC_RADIO_COD adicionar os métodos OnRadioCli e OnRadioCod à classe CVendasView respectivamente de modo tratar os eventos BN_CLICKED que resulta de o utilizador efectuar um clique nos botões. É ainda necessário adicionar a variável booleana m_RsSort como protegida à classe CVendasView. Esta deverá ser inicializada com o valor *false* no construtor que indica que a ordenação por defeito é em relação ao código de clientes. O código para estes dois métodos é então o seguinte:

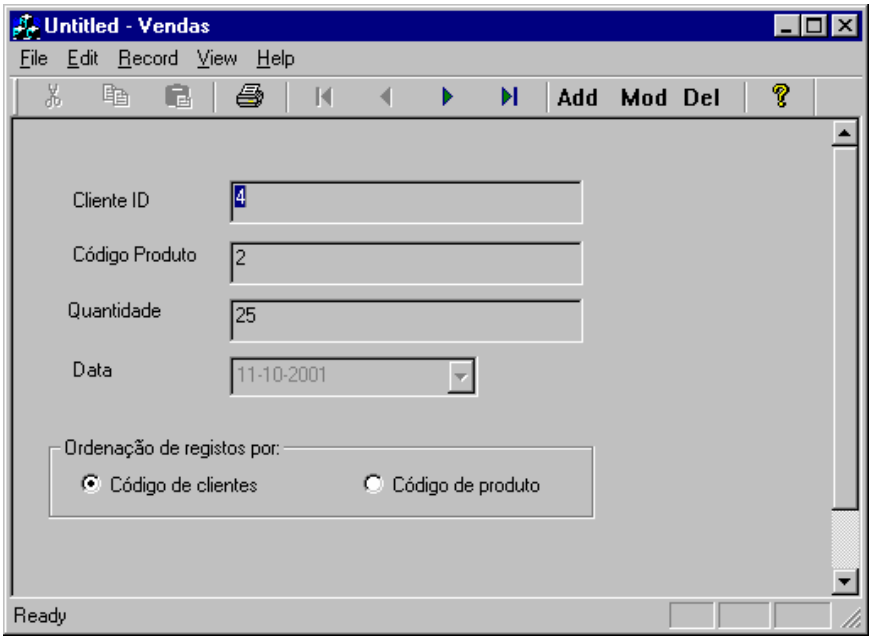
```
void CVendasView::OnRadioCod()
{
   m_bRsSort = true;
   m_pSet->m_strSort = "cod_prod";
   m_pSet->Requery();
   UpdateData(FALSE);
}


void CVendasView::OnRadioCli()
{
```

```
    m_bRsSort = false;
    m_pSet->m_strSort = "cli_id";
    m_pSet->Requery();
    UpdateData(FALSE);
}
```

Repare-se na variável m_strSort da classe CString que é um membro de dados da classe CRecordset à qual é atribuída o nome do campo que determina a ordenação dos registos. Esta variável é usada para internamente ser construído o comando SQL adequado. De notar que a invocação do método Requery é necessária para reconstruir o *recordset* de acordo com o novo filtro. Finalmente, relativamente à ordenação de dados, é necessário adicionar o código que a seguir se indica ao método OnInitialUpdate da classe CVendasView para que controlos de rádio e a ordenação seja correctamente inicializada.

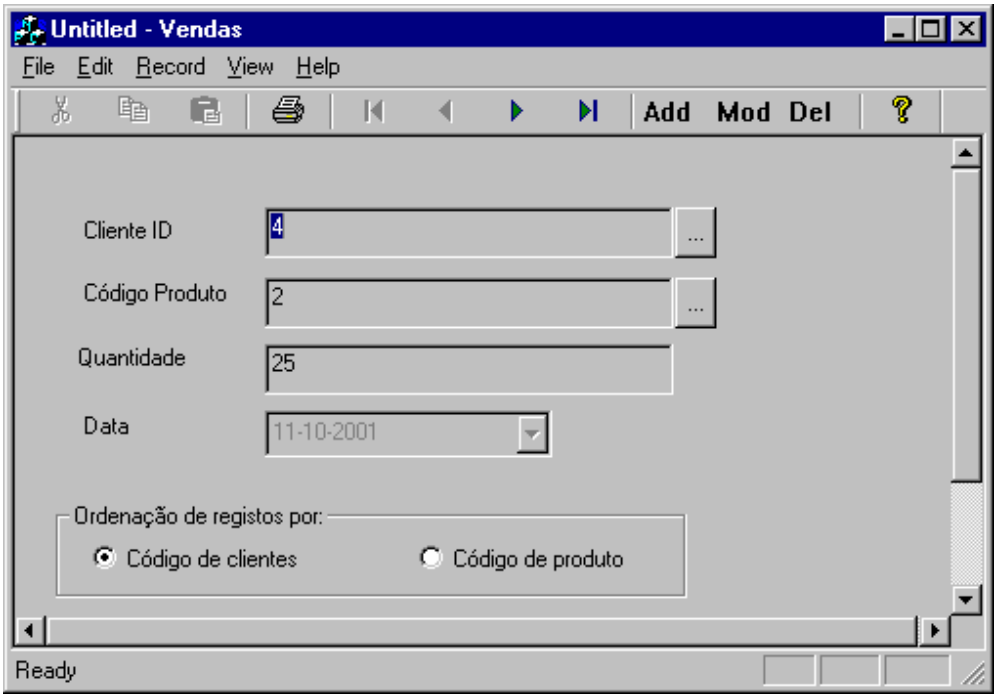
```
        CButton* pRB;
        if (!m_bRsSort)    {
                pRB = (CButton*)GetDlgItem(IDC_RADIO_CLI);
                OnRadioCli();
        } else    {
                pRB = (CButton*)GetDlgItem(IDC_RADIO_COD);
                OnRadioCod();
        }
        pRB->SetCheck( 1 );
```

A figura seguinte mostra a aplicação com estes novos controlos para a ordenação de registos.



Filtragem

Relativamente à filtragem de dados pretende-se adicionar a possibilidade de o utilizador poder ver os dados completos do cliente ou do produto do registo corrente de vendas. Para isso na caixa de diálogo com a identificação IDC_VENDAS_FORM serão adicionados dois botões, um junto ao código de cliente e outro junto ao código de produto. Estes botões servirão para abrir uma nova caixa de diálogo que mostrará a respectiva informação detalhada. Um dos botões terá a identificação de IDC_BUTTON_CLI e o outro IDC_BUTTON_COD consoante se trate do código de cliente ou o código de produto. Ambos deverão ter a propriedade *caption* com o valor "...". O passo seguinte consiste em criar um método para a classe CVendasView destinado a cada um dos botões de modo a tratar a sua selecção por parte do utilizador. Para isso deve-se recorrer à ClassWizard e seleccionar o separador "Message Maps". Considerando a classe CVendasView e o ID de cada botão, deve-se efectuar um duplo clique em na "Message" BN_CLICKED. Isto gera os métodos OnButtonCli e OnButtonCod pretendidos. A janela da aplicação fica então semelhante à figura seguinte.



O passo seguinte consiste em criar as caixas de diálogo que mostrarão a informação detalhada do cliente ou a informação detalhada do produto. Opta-se aqui por só apresentar o caso dos clientes, ficando como exercício o caso dos produtos dado que os procedimentos a seguir são semelhantes.

Mas antes de começar por criar a caixa de diálogo que mostrará a informação detalhada do cliente, é necessário criar um *recordset* que representará a tabela clientes. Para o

efeito deve-se abrir a ClassWizard e seleccionar o botão “Add Class” para adicionar uma nova classe. Na caixa de diálogo “New Class” indicar o nome da nova classe como CClientesSet e seleccionar a classe base CRecordSet. Seleccionando o botão “OK” abre a caixa de diálogo “Database Options” onde se deve seleccionar a base de dados ODBC usada pela aplicação. Para terminar seleccionar a tabela “cliente” e depois deve-se fechar a ClassWizard confirmando as selecções. Neste momento existe uma nova classe *recordset* adaptada à tabela clientes.

É então chegado o momento de criar a caixa de diálogo para apresentar a informação relativa aos clientes. Para começar é necessário criar os recursos relativos à caixa de diálogo com o editor de recursos. Começar por seleccionar a pasta Dialog do editor de recursos e com o botão direito do rato deve-se escolher “Insert Dialog”. Os recursos para uma caixa de diálogo é criada contendo dois botões (“OK” e “Cancel”). Em primeiro lugar é necessário alterar o ID por defeito IDC_DIALOG1 para IDC_CLIENTES_DLG. É ainda necessário alterar a propriedade *caption* para “Detalhes do Cliente” e remover os botões incluídos por defeito. De seguida devem ser inseridos os controlos indicados na tabela seguinte tendo-se o cuidado de fazer o redimensionamento do recurso relativa à caixa de diálogo para fazer caber todos controlos.

Tendo editado os recursos da caixa de diálogo é agora possível criar um objecto da classe CDialog para abrir a nova caixa de diálogo. Opta-se, no entanto, por criar uma nova classe CClientesDlg que é derivada da classe CDialog para esta caixa de diálogo em particular. O processo para criar esta nova classe passa pelo uso da ClassWizard tal como o foi feito no caso da criação da CClientesSet. Neste caso, como a nova classe deriva da classe CDialog, é necessário indicar apenas o ID para os recursos da caixa de diálogo (IDC_CLIENTES_DLG).

Tendo criado a classe CClientesDlg, o passo seguinte consiste em adicionar um objecto da classe CClinetesSet como membro de dados publico. Na janela “Workspace” e na vista “Classe View” deve-se seleccionar a classe CClientesDlg. De seguida com o botão direito do rato deve-se escolher o comando “Add Member Variable” para especificar o nome (m_CliSet), o tipo e o tipo de acesso do novo membro de dados da classe (não esquecer colocar ‘#include “ClientesSet.h”’ no ficheiro de definição da classe CClienteDlg).

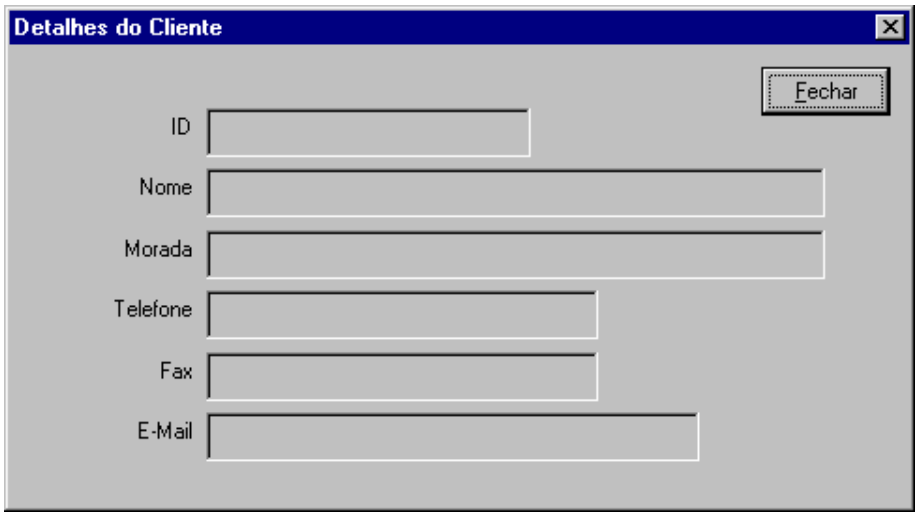
Tipo	**ID**	***Caption***	**Propriedades**
Texto Estático	IDC_STATIC_CLIID	Cliente ID	
Texto Estático	IDC_STATIC_CLINOME	Nome	
Texto Estático	IDC_STATIC_ CLIMORADA	Morada	
Texto Estático	IDC_STATIC_CLITEL	Telefone	
Texto Estático	IDC_STATIC_CLIFAX	Fax	

Texto Estático	IDC_STATIC_CLIEMAIL	E-Mail	
Edição de Texto	IDC_EDIT_CLIID	-	*ReadOnly*
Edição de Texto	IDC_EDIT_CLINOME	-	*ReadOnly*
Edição de Texto	IDC_EDIT_CLIMORADA	-	*ReadOnly*
Edição de Texto	IDC_ EDIT _CLITEL	-	*ReadOnly*
Edição de Texto	IDC_ EDIT _CLIFAX	-	*ReadOnly*
Edição de Texto	IDC_ EDIT _CLIEMAIL	-	*ReadOnly*
Botão	IDC_ BN_FECHAR	-	

Para que esta caixa de diálogo seja aberta é necessário adicionar o seguinte código ao método OnButtonCli da classe CVendasView:

```
CClientesDlg dlg;
dlg.m_CliSet.Open();
dlg.DoModal();
dlg.m_CliSet.Close();
```

Este código ainda não é suficiente para mostrar a informação desejada. No entanto, já abre o *recordeset* de clientes e fecha-o quando a caixa de diálogo é fechada. É ainda necessário completar algumas tarefas. Entretanto a caixa de diálogo já surge com um aspecto parecido com a seguinte.



O passo seguinte passa por se associar os controlos de edição da caixa de diálogo com as respectivos membros de dados do objecto m_CliSet da classe CClientesSet utilizando a ClassWizard. Dado que a ClassWizard, para caixas de diálogo, não mostra um *recordset* para fazer binding de campos é necessário especificar o *recordset* desejado seleccionando o separador “Class Info” da ClassWizard. Aqui, no grupo “Advanced options”, selecciona-se em “Foreign Class” o noma da classe derivada de CRecordset (CClienteSet) e em “Foreign variable” indica-se o nome de uma variável (m_pCliSet) que será um apontador para um objecto da classe CClienteSet. Neste momento já é possível

efectuar o *binding* dos campos do *recordset* com os controlos da caixa de diálogo seleccionando o separador “Member Variables”. Como resultado final, é adicionada a nova variável m_pCliSet como membro dados à classe CCientesDlg e é adicionado o código que efectua o *binding* pretendido com o seu método DoDataExchange tal como se indica a seguir

```
Void CClientesDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CClientesDlg)
    DDX_FieldText(pDX, IDC_EDIT_CLIID, m_pCliSet->m_id, m_pCliSet);
    DDX_FieldText(pDX, IDC_EDIT_CLINOME, m_pCliSet->m_nome, m_pCliSet);
    DDX_FieldText(pDX, IDC_EDIT_CLIMORADA, m_pCliSet->m_morada, m_pCliSet);
    DDX_FieldText(pDX, IDC_EDIT_CLITEL, m_pCliSet->m_telefone, m_pCliSet);
    DDX_FieldText(pDX, IDC_EDIT_CLIFAX, m_pCliSet->m_fax, m_pCliSet);
    DDX_FieldText(pDX, IDC_EDIT_CLIEMAIL, m_pCliSet->m_email, m_pCliSet);
    //}}AFX_DATA_MAP
}
```

Para completar o trabalho é necessário atribuir a m_pCliSet o endereço de m_CliSet. Isto pode ser efectuado no construtor da classe CClientesDlg. A partir deste instante já é possível observar o primeiro registo do *recordset* de clientes. No entanto, este não será em princípio o que se deseja visualizar. Aquele que se pretende visualizar tem de estar relacionado com o do *recordset* das vendas. É aqui que entra a filtragem, o assunto principal desta secção. Tal como para a ordenação, os objectos da classe CRecordset e derivados possuem um variável membro m_strFilter que contem o filtro a usar na recolha de informação da fonte de dados. Por defeito, esta variável contem uma *string* vazia. Para efectuar a filtragem de informação deve ser atribuída a esta variável um *string* que deverá conter a condição de filtragem do *recordset* ser aberto. O formato para a condição de filtragem e semelhante ao usado nas condições colocadas a seguir à palavra reservada WHERE do SQL. Sempre que seja necessário alterar o filtro deve-se primeiro fechar o *recordset*, caso este esteja aberto, atribuir o novo filtro a esta variável e, finalmente, voltar a abrir o *recordset*.

No caso particular desta aplicação, o filtro é definido após a criação de um objecto da classe CClientesDlg e antes de abrir o respectivo *recordset* m_CliSet. Deste modo o código do método OnBottunCli da classe CVendasView será o seguinte.

```
void CVendasView::OnButtonCli()
{
        CClientesDlg dlg;
        dlg.m_CliSet.m_strFilter.Format("id = %d", m_pSet->m_cli_id);
```

```
        dlg.m_CliSet.Open();
        if (dlg.m_CliSet.IsEOF())
                MessageBox("Error : No matching records.");
        else
                dlg.DoModal();
        dlg.m_CliSet.Close();
}
```

Para finalizar falta tratar o botão “Fechar” que deve fechar a caixa de diálogo quando escolhido. Para o efeito deve adicionar o método OnBnFechar à classe CClientesDlg para processar o evento BN_CLICKED usando a ClassWizard. Este método conterá o código que fecha a caixa de diálogo sendo semelhante ao seguinte.

```
void CClientesDlg::OnBnFechar()
{
        EndDialog(1);
}
```

Como já se referiu o processo para implementar o equivalente para os produtos é em tudo equivalente ao que aqui foi exposto para o caso dos clientes. Fica aqui, portanto, proposto o exercício para implementar esta situação.

Conclusão

Foi descrito neste documento um processo para o desenvolvimento de aplicações para o Windows baseados no paradigma “Document/View” com acesso a base de dados utilizando as classes MFC e o ODBC. No entanto, este documento inclui informação suficiente para a escrita de aplicações com acesso a base de dados que não seguem este paradigma “Document/View”. As aplicações baseadas em caixas de diálogo são alguns exemplos.

Este documento deve ser considerado como um ponto de partida para o desenvolvimento de aplicações mais complexas com acesso a base de dados utilizando MFC ODBC. Tudo depende da aplicação que tem de ser escrita, dos objectivos a atingir, do tempo a dispensar na exploração de novas possibilidades e da criatividade.

Anexos

CDatabase

Data Members

m_hdbc	Open Database Connectivity (ODBC) connection handle to a data source. Type **HDBC**.

Construction

CDatabase	Constructs a **CDatabase** object. You must initialize the object by calling **OpenEx** or **Open**.
Open	Establishes a connection to a data source (through an ODBC driver).
OpenEx	Establishes a connection to a data source (through an ODBC driver).
Close	Closes the data source connection.

Database Attributes

GetConnect	Returns the ODBC connect string used to connect the **CDatabase** object to a data source.
IsOpen	Returns nonzero if the **CDatabase** object is currently connected to a data source.
GetDatabaseName	Returns the name of the database currently in use.
CanUpdate	Returns nonzero if the **CDatabase** object is updatable (not read-only).

CanTransact	Returns nonzero if the data source supports transactions.
SetLoginTimeout	Sets the number of seconds after which a data source connection attempt will time out.
SetQueryTimeout	Sets the number of seconds after which database query operations will time out. Affects all subsequent recordset **Open**, **AddNew**, **Edit**, and **Delete** calls.
GetBookmarkPersistence	Identifies the operations through which bookmarks persist on recordset objects.
GetCursorCommitBehavior	Identifies the effect of committing a transaction on an open recordset object.
GetCursorRollbackBehavior	Identifies the effect of rolling back a transaction on an open recordset object.

Database Operations

BeginTrans	Starts a “transaction” — a series of reversible calls to the **AddNew**, **Edit**, **Delete**, and **Update** member functions of class **CRecordset** — on the connected data source. The data source must support transactions for **BeginTrans** to have any effect.
BindParameters	Allows you to bind parameters before calling **CDatabase::ExecuteSQL**.
CommitTrans	Completes a transaction begun by **BeginTrans**. Commands in the transaction that alter the data source are carried out.
Rollback	Reverses changes made during the current

	transaction. The data source returns to its previous state, as defined at the **BeginTrans** call, unaltered.
Cancel	Cancels an asynchronous operation or a process from a second thread.
ExecuteSQL	Executes an SQL statement. No data records are returned.

Database Overridables

OnSetOptions	Called by the framework to set standard connection options. The default implementation sets the query timeout value. You can establish these options ahead of time by calling **SetQueryTimeout**

CDatabase::Open

virtual BOOL Open(LPCTSTR *lpszDSN*, BOOL *bExclusive* = FALSE, BOOL *bReadOnly* = FALSE, LPCTSTR *lpszConnect* = “ODBC;”, BOOL *bUseCursorLib* = TRUE);
throw(CDBException, CMemoryException);

Return Value

Nonzero if the connection is successfully made; otherwise 0 if the user chooses Cancel when presented a dialog box asking for more connection information. In all other cases, the framework throws an exception.

Parameters

lpszDSN

Specifies a data source name — a name registered with ODBC through the ODBC Administrator program. If a DSN value is specified in *lpszConnect* (in the form “DSN=<data-source>”), it must not be specified again in *lpszDSN*. In this case, *lpszDSN* should be NULL. Otherwise, you can pass NULL if you want to present the user with a Data Source dialog box in which the user can select a data source. For further information, see Remarks.

bExclusive

Not supported in this version of the class library. Currently, an assertion fails if this parameter is TRUE. The data source is always opened as shared (not exclusive).

bReadOnly

TRUE if you intend the connection to be read-only and to prohibit updates to the data source. All dependent recordsets inherit this attribute. The default value is FALSE.

lpszConnect

Specifies a connect string. The connect string concatenates information, possibly including a data source name, a user ID valid on the data source, a user authentication string (password, if the data source requires one), and other information. The whole connect string must be prefixed by the string “ODBC;” (uppercase or lowercase). The “ODBC;” string is used to indicate that the connection is to an ODBC data source; this is for upward compatibility when future versions of the class library might support non-ODBC data sources.

bUseCursorLib

TRUE if you want the ODBC Cursor Library DLL to be loaded. The cursor library masks some functionality of the underlying ODBC driver, effectively preventing the use of dynasets (if the driver supports them). The only cursors supported if the cursor library is loaded are static snapshots and forward-only cursors. The default value is TRUE. If you plan to create a recordset object directly from CRecordset without deriving from it, you should not load the cursor library.

Remarks

Call this member function to initialize a newly constructed CDatabase object. Your database object must be initialized before you can use it to construct a recordset object.

Note Calling the OpenEx member function is the preferred way to connect to a data source and initialize your database object.

If the parameters in your Open call do not contain enough information to make the connection, the ODBC driver opens a dialog box to obtain the necessary information from the user. When you call Open, your connect string, *lpszConnect*, is stored privately in the CDatabase object and is available by calling the GetConnect member function.

If you wish, you can open your own dialog box before you call Open to get information from the user, such as a password, then add that information to the connect string you

pass to Open. Or you might want to save the connect string you pass so you can reuse it the next time your application calls Open on a CDatabase object.

You can also use the connect string for multiple levels of login authorization (each for a different CDatabase object) or to convey other data source-specific information. For more information about connect strings, see Chapter 5 in the *ODBC SDK Programmer's*

It is possible for a connection attempt to time out if, for example, the DBMS host is unavailable. If the connection attempt fails, Open throws a CDBException.

Example

```
// Embed a CDatabase object
// in your document class
CDatabase m_dbCust;

// Connect the object to a
// data source (no password)
// the ODBC connection dialog box
// will always remain hidden
m_dbCust.Open( _T( "MYDATASOURCE" ), FALSE,
        FALSE, _T( "ODBC;UID=JOES" ),

// ...Or, query the user for all
// connection information
m_dbCust.Open( NULL );
```

CRecordset

Data Members

m_hstmt	Contains the ODBC statement handle for the recordset. Type **HSTMT**.
m_nFields	Contains the number of field data members in the recordset. Type **UINT**.
m_nParams	Contains the number of parameter data members in the recordset. Type **UINT**.

m_pDatabase	Contains a pointer to the **CDatabase** object through which the recordset is connected to a data source.
m_strFilter	Contains a **CString** that specifies a Structured Query Language (SQL) **WHERE** clause. Used as a filter to select only those records that meet certain criteria.
m_strSort	Contains a **CString** that specifies an SQL **ORDER BY** clause. Used to control how the records are sorted.

Construction

CRecordset	Constructs a **CRecordset** object. Your derived class must provide a constructor that calls this one.
Open	Opens the recordset by retrieving the table or performing the query that the recordset represents.
Close	Closes the recordset and the ODBC **HSTMT** associated with it.

Recordset Attributes

CanAppend	Returns nonzero if new records can be added to the recordset via the **AddNew** member function.
CanBookmark	Returns nonzero if the recordset supports bookmarks.
CanRestart	Returns nonzero if **Requery** can be called to run the recordset's query again.
CanScroll	Returns nonzero if you can scroll through the records.
CanTransact	Returns nonzero if the data source supports

	transactions.
CanUpdate	Returns nonzero if the recordset can be updated (you can add, update, or delete records).
GetODBCFieldCount	Returns the number of fields in the recordset.
GetRecordCount	Returns the number of records in the recordset.
GetStatus	Gets the status of the recordset: the index of the current record and whether a final count of the records has been obtained.
GetTableName	Gets the name of the table on which the recordset is based.
GetSQL	Gets the SQL string used to select records for the recordset.
IsOpen	Returns nonzero if **Open** has been called previously.
IsBOF	Returns nonzero if the recordset has been positioned before the first record. There is no current record.
IsEOF	Returns nonzero if the recordset has been positioned after the last record. There is no current record.
IsDeleted	Returns nonzero if the recordset is positioned on a deleted record.

Recordset Update Operations

AddNew	Prepares for adding a new record. Call **Update** to

	complete the addition.
CancelUpdate	Cancels any pending updates due to an **AddNew** or **Edit** operation.
Delete	Deletes the current record from the recordset. You must explicitly scroll to another record after the deletion.
Edit	Prepares for changes to the current record. Call **Update** to complete the edit.
Update	Completes an **AddNew** or **Edit** operation by saving the new or edited data on the data source.

Recordset Navigation Operations

GetBookmark	Assigns the bookmark value of a record to the parameter object.
Move	Positions the recordset to a specified number of records from the current record in either direction.
MoveFirst	Positions the current record on the first record in the recordset. Test for **IsBOF** first.
MoveLast	Positions the current record on the last record or on the last rowset. Test for **IsEOF** first.
MoveNext	Positions the current record on the next record or on the next rowset. Test for **IsEOF** first.
MovePrev	Positions the current record on the previous record or on the previous rowset. Test for **IsBOF** first.
SetAbsolutePosition	Positions the recordset on the record corresponding to the specified record number.

SetBookmark	Positions the recordset on the record specified by the bookmark.

Other Recordset Operations

Cancel	Cancels an asynchronous operation or a process from a second thread.
FlushResultSet	Returns nonzero if there is another result set to be retrieved, when using a predefined query.
GetFieldValue	Returns the value of a field in a recordset.
GetODBCFieldInfo	Returns specific kinds of information about the fields in a recordset.
GetRowsetSize	Returns the number of records you wish to retrieve during a single fetch.
GetRowsFetched	Returns the actual number of rows retrieved during a fetch.
GetRowStatus	Returns the status of the row after a fetch.
IsFieldDirty	Returns nonzero if the specified field in the current record has been changed.
IsFieldNull	Returns nonzero if the specified field in the current record is Null (has no value).
IsFieldNullable	Returns nonzero if the specified field in the current record can be set to Null (having no value).
RefreshRowset	Refreshes the data and status of the specified row(s).
Requery	Runs the recordset's query again to refresh the

	selected records.
SetFieldDirty	Marks the specified field in the current record as changed.
SetFieldNull	Sets the value of the specified field in the current record to Null (having no value).
SetLockingMode	Sets the locking mode to “optimistic” locking (the default) or “pessimistic” locking. Determines how records are locked for updates.
SetParamNull	Sets the specified parameter to Null (having no value).
SetRowsetCursorPosition	Positions the cursor on the specified row within the rowset.

Recordset Overridables

Check	Called to examine the return code from an ODBC API function.
CheckRowsetError	Called to handle errors generated during record fetching.
DoBulkFieldExchange	Called to exchange bulk rows of data from the data source to the recordset. Implements bulk record field exchange (Bulk RFX).
DoFieldExchange	Called to exchange data (in both directions) between the field data members of the recordset and the corresponding record on the data source. Implements record field exchange (RFX).
GetDefaultConnect	Called to get the default connect string.

GetDefaultSQL	Called to get the default SQL string to execute.
OnSetOptions	Called to set options for the specified ODBC statement.
SetRowsetSize	Specifies the number of records you wish to retrieve during a fetch.

CRecordset::Open

virtual BOOL Open(UINT *nOpenType* = AFX_DB_USE_DEFAULT_TYPE, LPCTSTR *lpszSQL* = NULL, DWORD *dwOptions* = none); throw(CDBException, CMemoryException);

Return Value

Nonzero if the CRecordset object was successfully opened; otherwise 0 if CDatabase::Open (if called) returns 0.

Parameters

nOpenType

Accept the default value, **AFX_DB_USE_DEFAULT_TYPE**, or use one of the following values from the **enum OpenType**:

- **CRecordset::dynaset**.
- **CRecordset::snapshot**.
- **CRecordset::dynamic**.
- **CRecordset::forwardOnly** .

For **CRecordset**, the default value is **CRecordset::snapshot**.

lpszSQL

A string pointer containing one of the following:

- A **NULL** pointer.
- The name of a table.
- An SQL **SELECT** statement (optionally with an SQL **WHERE** or **ORDER BY** clause).

- A **CALL** statement specifying the name of a predefined query (stored procedure). Be careful that you do not insert whitespace between the curly brace and the **CALL** keyword.

  For more information about this string, see the table and the discussion of ClassWizard's role under Remarks.

  Note The order of the columns in your result set must match the order of the RFX or Bulk RFX function calls in your DoFieldExchange or DoBulkFieldExchange function override.

dwOptions

A bitmask which can specify a combination of the values listed below. Some of these are mutually exclusive. The default value is **none**.

- **CRecordset::none** No options set. This parameter value is mutually exclusive with all other values. By default, the recordset can be updated with Edit or Delete and allows appending new records with AddNew. Updatability depends on the data source as well as on the *nOpenType* option you specify. Optimization for bulk additions is not available. Bulk row fetching will not be implemented. Deleted records will not be skipped during recordset navigation. Bookmarks are not available. Automatic dirty field checking is implemented.

- **CRecordset::appendOnly** Do not allow **Edit** or **Delete** on the recordset. Allow **AddNew** only. This option is mutually exclusive with **CRecordset::readOnly**.

- **CRecordset::readOnly** Open the recordset as read-only. This option is mutually exclusive with **CRecordset::appendOnly**.

- **CRecordset::optimizeBulkAdd** Use a prepared SQL statement to optimize adding many records at one time. Applies only if you are not using the ODBC API function **SQLSetPos** to update the recordset. The first update determines which fields are marked dirty. This option is mutually exclusive with **CRecordset::useMultiRowFetch**.

- **CRecordset::useMultiRowFetch** Implement bulk row fetching to allow multiple rows to be retrieved in a single fetch operation. This is an advanced feature designed to improve performance; however, bulk record field exchange is not supported by ClassWizard. This option is mutually exclusive with **CRecordset::optimizeBulkAdd**. Note that if you specify **CRecordset::useMultiRowFetch**, then the option **CRecordset::noDirtyFieldCheck** will be turned on automatically (double buffering will not be available); on forward-only recordsets, the option **CRecordset::useExtendedFetch** will be turned on

automatically. For more information about bulk row fetching, see the article Recordset: Fetching Records in Bulk (ODBC) in *Visual C++ Programmer's Guide*.

- **CRecordset::skipDeletedRecords** Skip all deleted records when navigating through the recordset. This will slow performance in certain relative fetches. This option is not valid on forward-only recordsets. Note that **CRecordset::skipDeletedRecords** is similar to *driver packing*, which means that deleted rows are removed from the recordset. However, if your driver packs records, then it will skip only those records that you delete; it will not skip records deleted by other users while the recordset is open. **CRecordset::skipDeletedRecords** will skip rows deleted by other users.

- **CRecordset::useBookmarks** May use bookmarks on the recordset, if supported. Bookmarks slow data retrieval but improve performance for data navigation. Not valid on forward-only recordsets. For more information, see the article Recordset: Bookmarks and Absolute Positions (ODBC) in *Visual C++ Programmer's Guide*.

- **CRecordset::noDirtyFieldCheck** Turn off automatic dirty field checking (double buffering). This will improve performance; however, you must manually mark fields as dirty by calling the **SetFieldDirty** and **SetFieldNull** member functions.Note that double buffering in class **CRecordset** is similar to double buffering in class **CDaoRecordset**. However, in **CRecordset**, you cannot enable double buffering on individual fields; you either enable it for all fields or disable it for all fields. For more information about double buffering, see the DAO article DAO Record Field Exchange: Double Buffering Records in *Visual C++ Programmer's Guide*. Note that if you specify the option **CRecordset::useMultiRowFetch**, then **CRecordset::noDirtyFieldCheck** will be turned on automatically; however, **SetFieldDirty** and **SetFieldNull** cannot be used on recordsets that implement bulk row fetching.

- **CRecordset::executeDirect** Do not use a prepared SQL statement. For improved performance, specify this option if the **Requery** member function will never be called.

- **CRecordset::useExtendedFetch** Implement **SQLExtendedFetch** instead of **SQLFetch**. This is designed for implementing bulk row fetching on forward-only recordsets. If you specify the option **CRecordset::useMultiRowFetch** on a forward-only recordset, then **CRecordset::useExtendedFetch** will be turned on automatically.

- **CRecordset::userAllocMultiRowBuffers** The user will allocate storage buffers for the data. Use this option in conjunction with **CRecordset::useMultiRowFetch** if you want to allocate your own storage; otherwise, the framework will automatically allocate the necessary storage. For more information, see the article Recordset: Fetching Records in Bulk (ODBC) in *Visual C++ Programmer's Guide*. Note that specifying **CRecordset::userAllocMultiRowBuffers** without specifying **CRecordset::useMultiRowFetch** will result in a failed assertion.

Remarks

You must call this member function to run the query defined by the recordset. Before calling **Open**, you must construct the recordset object.

This recordset's connection to the data source depends on how you construct the recordset before calling **Open**. If you pass a CDatabase object to the recordset constructor that has not been connected to the data source, this member function uses GetDefaultConnect to attempt to open the database object. If you pass **NULL** to the recordset constructor, the constructor constructs a **CDatabase** object for you, and **Open** attempts to connect the database object. For details on closing the recordset and the connection under these varying circumstances, see Close.

When you call **Open**, a query, usually an SQL **SELECT** statement, selects records based on criteria shown in the following table.

Value of the lpszSQL parameter	**Records selected are determined by**	**Example**
NULL	The string returned by **GetDefaultSQL**.	
SQL table name	All columns of the table-list in **DoFieldExchange** or **DoBulkFieldExchange**.	"Customer"
Predefined query (stored procedure) name	The columns the query is defined to return.	"{call OverDueAccts}"
SELECT column-list **FROM** table-list	The specified columns from the specified table(s).	"SELECT CustId, CustName FROM

		Customer"

! Warning Be careful that you do not insert extra whitespace in your SQL string. For example, if you insert whitespace between the curly brace and the **CALL** keyword, MFC will misinterpret the SQL string as a table name and incorporate it into a **SELECT** statement, which will result in an exception being thrown. Similarly, if your predefined query uses an output parameter, do not insert whitespace between the curly brace and the '?' symbol. Finally, you must not insert whitespace before the curly brace in a **CALL** statement or before the **SELECT** keyword in a **SELECT** statment.

The usual procedure is to pass **NULL** to **Open**; in this case, **Open** calls GetDefaultSQL. If you are using a derived **CRecordset** class, **GetDefualtSQL** gives the table name(s) you specified in ClassWizard. You can instead specify other information in the *lpszSQL* parameter.

Whatever you pass, **Open** constructs a final SQL string for the query (the string may have SQL **WHERE** and **ORDER BY** clauses appended to the *lpszSQL* string you passed) and then executes the query. You can examine the constructed string by calling GetSQL after calling **Open**. For additional details about how the recordset constructs an SQL statement and selects records, see the article Recordset: How Recordsets Select Records (ODBC) in *Visual C++ Programmer's Guide*.

The field data members of your recordset class are bound to the columns of the data selected. If any records are returned, the first record becomes the current record.

If you want to set options for the recordset, such as a filter or sort, specify these after you construct the recordset object but before you call **Open**. If you want to refresh the records in the recordset after the recordset is already open, call Requery.

Example

The following code examples show different forms of the **Open** call.

```
// rs is a CRecordset or
// CRecordset-derived object

// Open rs using the default SQL statement,
// implement bookmarks, and turn off
// automatic dirty field checking
rs.Open( CRecordset::snapshot, NULL,
```

```
        CRecordset::useBookmarks |
        CRecordset::noDirtyFieldCheck );

// Pass a complete SELECT statement
// and open as a dynaset
rs.Open( CRecordset::dynaset,
        _T( "Select L_Name from Customer" ) );

// Accept all defaults
rs.Open( );
```

Tipos de cursor ODBC

O ODBC especifica que quatro tipos básicos de cursores são suportados. Um cursor numa base de dados é simplesmente um apontador “móvel” para o registo de interesse. Os cursores movem-se sobre o que se designa por conjunto resultante (*result set*) que representa uma colecção de registos retornados por uma *query*, geralmente uma expressão SELECT do SQL. Na sua actual implementação, um objecto da classe CRecordset pode ser configurado para usar um destes tipos de cursores desde que o driver ODBC utilizado o suporte.

Tipo de CRecordset	**Tipo de cursor ODBC**	**Características**
forwardOnly	Forward Only	O *scroll* é efectuado apenas para a frente e não actualizável
snapshot	Static	O *scroll* é bidireccional, actualizável e usualmente utiliza o ODBC cursor library. A ordem dos registos é determinada quando o recordset é aberto. As alterações efectuadas por outros utilizadores não são visíveis até o recordset ser fechado e novamente aberto.
dynaset	Keyset Driven	O *scroll* é bidireccional e actualizável. Usa SQLSetPos para actualizações, inserções e remoções. A ordem dos registos é determinada quando o recordset é aberto, mas alterações

		efectuadas por outros utilizadores são visiveis após uma operação de *fetch*.
dynamic	Dynamic	O *scroll* é bidireccional e actualizável. Usa SQLSetPos para actualizações, inserções e remoções. Muitos *drivers* não o suportam. A ordem dos registos é determinada quando o recordset é aberto, mas alterações efectuadas por outros utilizadores são visiveis após uma operação de *fetch*.