André Kishimoto

```
//------------------------------------------------------------------------
// Bibliotecas
//------------------------------------------------------------------------
#include <windows.h>

//------------------------------------------------------------------------
// Definições
//------------------------------------------------------------------------
// Nome da classe da janela
#define WINDOW_CLASS  "prog05-4"
// Titulo da janela
#define WINDOW_TITLE  "Prog 05-4"
// Largura da janela
#define WINDOW_WIDTH  320
// Altura da janela
#define WINDOW_HEIGHT  240

//------------------------------------------------------------------------
// Protótipo das funções
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

//------------------------------------------------------------------------
// W
//------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   nCmdShow)
{
  // Cria a classe da janela e especifica seus atributos
  WNDCLASSEX wcl;
  wcl.cbSize = sizeof(WNDCLASSEX);
  wcl.style = CS_HREDRAW | CS_VREDRAW;
  wcl.lpfnWndProc = (WNDPROC)WindowProc;
  wcl.cbClsExtra = 0;
  wcl.cbWndExtra = 0;
  wcl.hInstance = hInstance;
  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
  wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
  wcl.lpszMenuName = NULL;
  wcl.lpszClassName = WINDOW_CLASS;
  wcl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

  // Registra a classe da janela
  if(RegisterClassEx(&wcl))
  {
    // Cria a janela principal do programa
    HWND hWnd = NULL;
    hWnd = CreateWindowEx(
           NULL,
     WINDOW_CLASS,
     WINDOW_TITLE,
     WS_OVERLAPPEDWINDOW | WS_VISIBLE,
     (GetSystemMetrics(SM_CXSCREEN) - WINDOW_WIDTH) / 2,
     (GetSystemMetrics(SM_CYSCREEN) - WINDOW_HEIGHT) / 2,
     WINDOW_WIDTH,
     WINDOW_HEIGHT,
     HWND_DESKTOP,
```

Programação Windows: C e Win32 API com ênfase em Multimídia

ISBN 85-906129-1-0

9 788590 612919

Página em branco

Programação Windows: C e Win32 API com ênfase em Multimídia

André Kishimoto

E-Book gratuito disponível em http://www.tupinihon.com

Página em branco

André Kishimoto

Programação Windows: C e Win32 API com ênfase em Multimídia

São Paulo, 2006
1ª edição

ISBN 85-906129-1-0

André Kishimoto
kishimoto@tupinihon.com
http://www.tupinihon.com

À todos aqueles que têm força de vontade
em aprender e superar novos desafios.

Página em branco

Agradecimentos

Agradeço meus pais e minha família, por sempre terem me apoiado e pelas oportunidades que sempre me deram durante minha vida. Sou muito grato por ter um grande amor na minha vida, Natália, que sempre foi amiga e companheira, nos nossos momentos mais felizes e mesmo durante as nossas brigas. Agradeço também meus ex-professores Édson Rego Barros e Sérgio Vicente Pamboukian, por proporem tarefas onde pude aplicar meus conhecimentos que são demonstrados nesse trabalho e por darem a idéia e incentivo a escrever um livro sobre programação gráfica; amigos e colegas que apoiaram meu trabalho e que sempre ajudaram na medida do possível, e os profissionais e hobbyistas da área que compartilham seus conhecimentos, através de livros, artigos e Internet. À todos vocês, sou muito grato pela ajuda e força que sempre me deram.

Página em branco

Sobre o Autor

André Kishimoto começou a programar quando tinha 12 anos, desenvolvendo pequenos programas em *Basic* e *PCBoard Programming Language* (sistema para gerenciamento de BBS's). Desde o início da sua jornada auto-didata pela programação, sempre teve interesse em multimídia, criando animações em ASCII e ANSI tanto em *Basic* quanto em *Clipper*. Após alguns anos, aprendeu a trabalhar no modo 13h (VGA) via *Pascal* – ponto de partida para usar imagens mais sofisticadas e a ter interesse em aprender a programar em linguagem C e C++.

Especialista em Computação Gráfica (SENAC-SP) e bacharel em Ciência da Computação (Universidade Presbiteriana Mackenzie), desenvolveu *advergames* e jogos para dispositivos móveis Java, Brew, iOS e Android.

Atualmente é programador sênior da Electronic Arts e professor de Jogos Digitais da Universidade Cruzeiro do Sul.

Página em branco

Sobre a versão gratuita do e-book

O material desse e-book foi elaborado há quase uma década e desde a data de seu lançamento (2006) diversas pessoas se interessaram e adquiriram o material.

Por causa do apoio e interesse dessas pessoas é que hoje esse material está disponível gratuitamente para você (caso você tenha pago pelo material, por favor me avise pois alguém está se aproveitando da gente). Talvez eu tenha demorado um pouco mais que o ideal para deixar o conteúdo acessível para todos, mas, como diz o velho ditado, antes tarde do que nunca.

De lá pra cá, muita coisa mudou – o sistema operacional Windows teve várias edições, os computadores hoje são 64-bit, temos a plataforma .Net e o Microsoft Visual C++ que antes era somente pago, hoje possui versões gratuitas.

Atualmente também temos a diferença que o MS-Windows não está mais sozinho: com a popularidade do iPhone e iPad e preços mais acessíveis de computadores Apple, mais pessoas (e desenvolvedores) possuem o Mac OS rodando em suas máquinas.

Embora existam todas essas mudanças, o conteúdo desse e-book continua válido – afinal, a API do Windows sofreu alterações mas o seu *core* continua existindo, tanto que os exemplos criados para o e-book rodam nas máquinas atuais com Windows 7.

Outro ponto importante é que o material aborda uma API usada para desenvolver aplicações nativas e que faz a ponte entre software, sistema operacional e hardware, conceito aplicado em outros sistemas como Linux, Mac OS, iOS e Android. Ou seja, conceitos como função *callback*, mensagens do sistema, acesso ao dispositivo gráfico, bitmaps, áudio e timers são comuns em todos os sistemas e API's atuais.

Como disse à todos que adquiriram o e-book antes da sua versão gratuita: espero que esse material lhe seja útil no seu dia-a-dia, seja para algo profissional ou como hobby. E agradeço antecipadamente o interesse no meu trabalho!

- O Autor, março de 2012

Página em branco

Sumário

Página em branco

Introdução

A idéia de escrever um livro sobre programação Windows teve início em 2002, durante uma aula de Estrutura de Dados na faculdade. Nessa aula, todos os alunos tinham que entregar um trabalho sobre implementação de filas, e o projeto consistia em implementar uma simulação de fábrica: quando o usuário quisesse, o programa deveria criar diferentes tipos de matéria-prima, que seriam processados por uma suposta máquina, na qual, dependendo da matéria-prima que fosse processada, um produto diferente seria criado.

Nesse trabalho, o professor nos tinha dado total liberdade sobre a implementação visual, desde que o programa executasse o que fora pedido. Durante a entrega, todos estavam curiosos em saber como cada um tinha implementado sua "fábrica de produtos". Estávamos no segundo semestre, e o que tínhamos aprendido na faculdade até então era o básico da programação em C. Como sempre, havia alguns alunos com mais conhecimento que outros, e muitos trabalhos entregues eram interessantes – porém, todos feitos em modo texto. O único trabalho em modo gráfico tinha sido o meu (um ano antes de entrar para a faculdade, eu já havia iniciado meus estudos em programação C e Windows), e todos ficaram surpresos quando viram o trabalho, inclusive o professor.

Na ocasião, conversei com o professor sobre escrever uma biblioteca gráfica e junto anexar um documento sobre como utilizar as funções da biblioteca. Ele, me dando o conselho que seria interessante liberar o código-fonte, e não apenas criar uma biblioteca com a descrição das funções, sugeriu que eu pensasse na idéia de escrever um livro sobre programação gráfica para Windows. Era algo que eu nunca havia pensado, e essa idéia me fez imaginar como seria o trabalho de escrever um livro, as dificuldades, o que traria de benefício para mim e como o livro ajudaria as pessoas.

Embora ele não tinha me desafiado, eu aceitei a idéia como um desafio. Escrever um livro sobre programação Windows. Comecei a esboçar as primeiras páginas do futuro livro nas semanas seguintes, pedindo sugestões e dicas para o professor (que é co-autor dos livros "*Delphi para Universitários*" e "*C++ Builder para Universitário*", ambos lançados pela editora Páginas e Letras). Na época, por diversos motivos, acabei desistindo da idéia de escrever o livro. Foi após um ano, então, que voltei com a idéia; dessa vez, para concretizá-la.

Em 2003, estava cursando uma matéria onde o assunto principal era a criação de interfaces gráficas em Windows e Linux. Durante esse curso, pude notar que quase 98% da classe não tinha a mínima noção de como criar um programa gráfico sem o uso do Visual Basic, Delphi (e Kylix) ou C++ Builder. Na época, achei isso um absurdo, mas hoje vejo que não posso pensar dessa maneira, pois ainda são raros os livros sobre programação Windows (utilizando a linguagem C/C++ e Win32 API) disponíveis na língua portuguesa – um dos únicos lançados até o momento é o do professor dessa matéria ("*Desenvolvendo interfaces gráficas utilizando Win32 API e Motif*", Sérgio Vicente D. Pamboukian, da editora Scortecci). Além disso, outro fator que impede as pessoas de aprenderem a programar para Windows em C é o alto custo dos livros importados (além da própria língua inglesa ser um problema para muitos), inviável para muitos.

Depois de constatar tal fato, resolvi que realmente deveria escrever um livro sobre o assunto, em português, para que todas as pessoas interessadas pudessem ter por onde começar. Foi assim que surgiu esse livro, que tem a intenção de ensinar o leitor a criar programas com ênfase em multimídia para ambiente Windows, através da linguagem C e da Win32 API. Espero que meu objetivo seja atingido!

- O Autor

Convenções utilizadas no livro

Para melhor compreensão durante a leitura do livro, são utilizados diferentes estilos de escrita e alguns avisos:

Estilos de escrita:

- Textos em *itálico* indicam palavras que merecem destaque por serem termos técnicos ou palavras em inglês (com exceção de nomes de produtos, empresas e/ou marcas);

- Comandos de programação, como nomes de funções e variáveis, no meio dos textos, são escritas com fonte diferente, como no exemplo: `char teclaPressionada`;

- Partes de código de programação ou mesmo listagem de código-fonte de programas-exemplos aparecem como abaixo:

```
#include <stdio.h>

void main(void)
{
  printf("Texto exemplo\n");
}
```

Avisos:

- **Nota:** faz um breve comentário sobre a teoria explicada ou alerta o leitor sobre algo importante sobre o assunto;

- **Dica:** fornece dicas para que o leitor possa melhorar o seu aprendizado;

- **C++ explicado:** contém explicações quando conceitos da linguagem C++ são utilizados no livro.

O que você precisa saber

Antes de começar a leitura desse livro, você precisa estar familiarizado com o ambiente Windows (95 ou superior) e saber programar em linguagem C. Assim como em qualquer criação de algoritmos e programas, é essencial também que você tenha conhecimento e noção de lógica de programação.

Nota: apesar da linguagem C ser a única requisitada, os códigos do livro estão escritos em C++ (arquivo com extensão *.cpp*) para aproveitar alguns recursos da linguagem. Entretanto, isso não irá atrapalhar no entendimento dos programas, pois os códigos C e C++ do livro são muito semelhantes e as sintaxes C++ serão explicadas no decorrer do livro.

O que você irá aprender

Durante a leitura do livro, você aprenderá como criar programas Windows a partir do zero, utilizando linguagem C e Win32 API (*Application Programming Interface*, um conjunto de funções as quais são utilizadas para diversas ações de um programa Windows), através de explicações e

demonstrações com exemplos práticos (programas que encontram-se no CD-ROM). Com os exemplos, você irá construir pequenos programas a cada capítulo, exercitando diferentes funções de um programa multimídia Windows.

Recursos necessários

Para criar os programas-exemplo desse livro será necessário ter um compilador C/C++ 32-bit que crie arquivos executáveis para Windows instalado no computador. Existem diversos compiladores no mercado, tais como Dev-C++, lcc-Win32, Inprise/Borland C++ Builder e Microsoft Visual C++. Escolha um que você se adapte melhor.

Dica: lembre-se de consultar o manual ou ajuda do compilador/ambiente de programação que você estiver trabalhando, pois aprender a usar as ferramentas irá ajudar bastante e lhe salvará de futuras dores de cabeça.

Os programas-exemplo necessitam do sistema operacional Microsoft Windows 95 ou superior para serem executados, requerendo as mesmas configurações que os compiladores exigem do computador.

Nota: os programas-exemplo do livro foram criados com o Microsoft Visual C++.Net 2002 e testados sob o sistema operacional Microsoft Windows XP Professional, porém eles podem ser editados e compilados em qualquer outro compilador para Windows 32-bit.

Capítulo 1 – Iniciando

A criação de programas para Windows é um pouco diferente da programação de aplicativos console para MS-DOS, sendo à primeira vista algo mais complexo, pois o próprio Windows em si é um sistema mais complexo e avançado que o MS-DOS.

O Windows executa os programas através do processamento de mensagens, que ocorre da seguinte maneira: o programa que está sendo executado aguarda o recebimento de uma mensagem do Windows e quando o mesmo recebe a mensagem (por uma função especial do Windows), espera-se que o programa execute alguma ação para processar essa mensagem.

Existem diversas mensagens que o Windows pode enviar ao programa; por exemplo, quando o usuário modifica o tamanho da janela do programa, um clique no botão do mouse, a finalização do programa, enfim, para cada ação realizada pelo usuário (e conseqüentemente pelo Windows), uma mensagem é enviada para o programa processar. Obviamente, não é toda mensagem que deverá ser processada; se o seu programa utilizar somente o teclado, podemos descartar todas as mensagens enviadas sobre as ações do mouse.

Nota: na programação Windows, é o sistema quem inicia a interação com seu programa (ao contrário do que ocorre em programas MS-DOS) e apesar de ser possível fazer chamadas de funções da Win32 API em resposta a uma mensagem, ainda é o Windows quem inicia a atividade.

Win32 API, Platform SDK, MFC???

Caso você faça uma pesquisa na Internet sobre programação para Windows, provavelmente obterá milhares de resultados com essas siglas. Nesse livro, somente a Win32 API (*Application Programming Interface*, ou Interface de Programação de Aplicativos) e o *Platform SDK* (*Software Development Kit*, ou Kit de Desenvolvimento de Softwares) são utilizados para criar os programas. Na verdade, a Win32 API faz parte do Platform SDK (que é composto de bibliotecas, cabeçalhos e funções para programação Windows), e é através da utilização do Platform SDK que podemos criar os programas para Windows.

A *MFC*, sigla de *Microsoft Foundation Class*, é um conjunto de classes C++ criado pela Microsoft para simplificar a vida do programador. Porém, toda *MFC* foi baseada no *Platform SDK* e em classes, tornando-se mais complexa para programadores C. Ainda, não é suportada por todos os compiladores e linguagens (restringindo sua programação para determinados compiladores, sendo esses, na grande maioria, comerciais).

A razão da escolha do uso da Win32 API no livro é o fato que ela é baseado na linguagem C (ao contrário da MFC, baseada na linguagem C++), sendo acessível tanto para o programador C quanto ao programador C++. Ao aprender a utilizar a Win32 API, você poderá fazer tudo o que faria com a MFC e também utilizar suas funções em outros ambientes de programação, como o Inprise/Borland Delphi ou Microsoft Visual Basic.

C++ explicado: a MFC é composta de diversas *classes*, que podem ser vistas como uma espécie de *struct* com suporte a funções e uma grande quantidade de recursos para programação orientada a objetos (assunto o qual não será discutido no livro).

Notação húngara e nomenclatura de variáveis

Antes de partirmos para a explicação do primeiro programa-exemplo do livro, precisamos aprender um conceito muito importante quanto à programação Windows: a *notação húngara*, um conjunto de convenções para nomear variáveis criado por um programador da Microsoft, o húngaro Charles Simonyi (por isso o uso do termo *notação húngara*).

De uma maneira bem resumida, a notação húngara diz que as variáveis devem ser formadas por duas partes: a primeira, com todas as letras em minúsculo representando o tipo da variável; e a segunda, com a primeira letra em maiúsculo e as outras em minúsculo, distinguindo as variáveis do mesmo tipo.

Por exemplo, se quisermos declarar uma variável do tipo `int` chamada *contanumero*, devemos declará-la da seguinte forma: `int iContaNumero`, ao invés de `int contanumero`. O prefiro `i` no nome da variável indica que ela é do tipo `int`.

Todas as funções e variáveis utilizadas pela Win32 API seguem a notação húngara, por isso a necessidade de conhecer esse conceito. A Tabela 1.1 mostra as especificações da notação húngara. (Note que diversos tipos de dados não são comuns da linguagem C; eles foram definidos no cabeçalho *windows.h*).

Prefixo	Tipo de dado
b,f	BOOL (int) ou FLAG
by	BYTE (unsigned char)
c	char, WCHAR ou TCHAR
cx,cy	tamanho de x e y (c de "count")
dw	DWORD (unsigned long)
fn	função
h	handle
i	int
l	LONG (long)
lp	ponteiro long de 32-bit
msg	message (mensagem)
n	short ou int (referência à número)
s	string
sz,str	string terminado com byte 0
w	UINT (unsigned int) ou WORD (unsigned word)
x,y	coordenadas int x e y

Tabela 1.1: Os prefixos da notação húngara.

Existe também uma convenção para nomenclatura de funções (que também é utilizada para variáveis, quando a notação húngara é descartada), onde a primeira palavra da função é escrita toda em minúscula e a primeira letra de cada palavra extra no nome da função é escrita em maiúscula, por exemplo:

```
void testar();
void obterResultado();
```

Porém, essa é uma convenção adotada para programas em C++ e Java, sendo que a Win32 API adota uma convenção (que também será adotada nesse livro) onde a primeira letra de cada palavra da função deve ser escrita em maiúscula e o restante em minúscula:

```
void CapturarTela();
void WinMain();
```

No caso de constantes ou definições, os nomes devem ser escritos com todas as letras em maiúscula e as palavras devem ser separadas por *underscore* ("_"). Exemplo:

```
const int PI = 3.14159265358979323846264338327950;
```

```
#define WINDOW_WIDTH 640
```

Seu primeiro programa

Feito uma breve introdução à visão de programação para Windows, vamos iniciar, como todo primeiro exemplo, com um simples programa no estilo "*Hello World!*", mostrando apenas uma mensagem na tela. A Listagem 1.1 a seguir é um dos menores programas Windows possíveis de se escrever:

```
//------------------------------------------------------------------------
// prog01.cpp - Programa Windows estilo "Hello World!"
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// Bibliotecas
//------------------------------------------------------------------------
#include <windows.h>

//------------------------------------------------------------------------
// WinMain() -> Função principal
//------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
  MessageBox(NULL, "Esse é o Primeiro Programa-Exemplo!", "Prog 01", MB_OK);
  return(0);
}
```

Listagem 1.1: Um programa mínimo em Windows.

O resultado da execução desse programa será o seguinte (Figura 1.1):

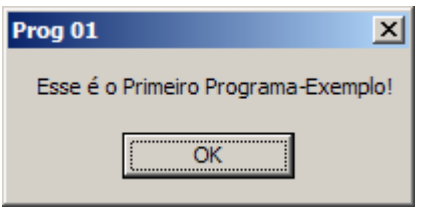

Figura 1.1: Seu primeiro programa.

C++ explicado: nesse primeiro programa-exemplo, existe uma sintaxe da linguagem C++, que são os comentários. Em C, comentários podem ser escritos somente entre `/*` e `*/`. Em C++, existe uma outra sintaxe de comentário conhecida por "comentários de linha", que são escritos após o uso de `//` (duas barras seguidas). O conteúdo após essas duas barras é

considerado comentário pelos compiladores e seu término é o final da linha.

A #include <windows.h>

Para que você possa começar a criar um programa para Windows, o primeiro passo a ser feito é incluir o arquivo de cabeçalho *windows.h* no seu programa. Esse arquivo de cabeçalho contém definições, macros e estruturas para escrever códigos portáveis entre as versões do Microsoft Windows.

Ao visualizar o código-fonte dos cabeçalhos que são inclusos junto com o *windows.h*, podemos verificar centenas de definições e macros, como por exemplo:

```
#define TRUE  1
#define FALSE 0
typedef int BOOL
```

Portanto, não estranhe caso você encontre uma variável do tipo `BOOL` e ela receba um valor `TRUE` ou `FALSE`, pois como podemos ver no trecho de código acima, a variável nada mais é do que um `int`, e `TRUE` e `FALSE` são 1 e 0, da mesma maneira que utilizamos esses valores e variáveis como uma espécie de *flag*/tipo booleano em linguagem C.

Nota: na Win32 API, foram criados novos tipos de variáveis, como `LONG`, `LPSTR` e `HANDLE`. Todos esses tipos são modificações dos tipos padrões da linguagem C (`LONG`, por exemplo, é um inteiro de 32 bits), e devem ser utilizados para compatibilidade, pois no futuro um `LONG` poderá ser de 64 bits, suportando os novos processadores e sistemas operacionais. Assim, facilitará a portabilidade de um sistema para outro.

Entendendo o programa

Como você pode notar, não existe mais uma função `main()` no código do programa, sendo essa substituído pela função `WinMain()`. A função `WinMain()` é a função principal de qualquer programa Windows e deve ser inclusa obrigatoriamente, pois é chamado pelo sistema como ponto inicial do programa. Diferente da função `main(int argv, char *argc[])`, ela recebe quatro parâmetros e possui o seguinte protótipo:

```
int WINAPI WinMain(
  HINSTANCE hInstance, // identificador para a atual instância do programa.
  HINSTANCE hPrevInstance, // identificador para a instância anterior do
programa.
  LPSTR lpCmdLine, // é um ponteiro para a linha de comando do programa.
  int nCmdShow // especifica como a janela do programa deve ser mostrada.
);
```

Nota: enquanto a função `main(int argv, char *argc[])` pode ser declarada apenas como `main()` – sem parâmetros, a função `WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)` deve sempre ser declarada com todos seus quatro parâmetros, mesmo que esses não sejam utilizados.

No Windows, é possível que diversas instâncias de um mesmo programa estejam sendo executadas ao mesmo tempo; assim, ao ser executado, o Windows cria um identificador (*handle*) para a instância atual do programa em `HINSTANCE hInstance` (o `hInstance` pode ser considerado como uma identidade única para cada instância do programa) para que o Windows e o programa possam sincronizar corretamente as informações e mensagens.

O parâmetro `HINSTANCE hPrevInstance` sempre será `NULL` para programas com tecnologia Win32 (ou seja, Windows 95 e versões superiores). Esse parâmetro era utilizado nas versões mais antigas do Windows (Win16), porém agora tornou-se obsoleto e consta como parâmetro para fins de compatibilidade de código.

O parâmetro `LPSTR lpCmdLine` tem a mesma função do `char *argc[]` da função `main()` de um programa DOS, armazenando a linha de comando do programa (excluindo o nome do programa) na variável `lpCmdLine`. A diferença entre eles é que a variável `*argc[]` armazena vetores de strings, enquanto `lpCmdLine` armazena a linha de comando como uma única e extensa string.

O último parâmetro, `int nCmdShow`, é utilizado para identificar o estado em que a janela do programa será iniciada (maximizada, minimizada, normal, etc). A Tabela 1.2 a seguir mostra os valores que esse parâmetro pode receber:

Valor	Descrição
`SW_HIDE`	Esconde a janela e ativa outra janela.
`SW_MINIMIZE`	Minimiza a janela especificada e ativa a janela de nível mais alto na listagem do sistema.

SW_RESTORE	Ativa e mostra a janela. Se a janela está minimizada ou maximizada, o sistema restaura o tamanho e posição original (idem ao SW_SHOWNORMAL).
SW_SHOW	Ativa a janela e a mostra no seu tamanho e posição atual.
SW_SHOWMAXIMIZED	Ativa a janela e a mostra maximizada.
SW_SHOWMINIMIZED	Ativa a janela e a mostra como um ícone.
SW_SHOWMINNOACTIVE	Mostra a janela como um ícone.
SW_SHOWNA	Mostra a janela no seu estado atual.
SW_SHOWNOACTIVATE	Mostra a janela em sua mais recente posição e tamanho.
SW_SHOWNORMAL	Ativa e mostra a janela. Se a janela está minimizada ou maximizada, o sistema restaura o tamanho e posição original (idem ao SW_RESTORE).

Tabela 1.2: Valores para o parâmetro `int nShowCmd` da `WinMain()`.

A função `WinMain()` sempre será do tipo `WINAPI` (algumas pessoas utilizam a definição `APIENTRY` ao invés de `WINAPI`, porém ambas têm o mesmo significado), fazendo com que ela tenha uma propriedade diferente na convenção de chamada da função. Como padrão, todas as funções do seu programa sempre utilizam a convenção de chamada de C. A convenção de chamada de funções difere em como os parâmetros são passados e manipulados para a pilha de memória e para os registradores do computador.

O retorno da função `WinMain()` é do tipo `int`, o qual deve ser retornado o valor de saída que está contido no parâmetro `wParam` da fila de mensagens se a `WinMain()` for finalizada normalmente com a mensagem `WM_QUIT` (será discutida no capítulo 2), ou deve retornar zero se a `WinMain()` terminou antes de entrar no *loop* de mensagens.

Nota: não se preocupe se o último parágrafo ficou meio obscuro para você nesse momento, pois irei falar sobre mensagens e laço de mensagens no próximo capítulo. Apenas expliquei o retorno da `WinMain()` pois sempre é necessário que ela retorne algo, e como no programa não existe um laço de mensagens, o retorno é zero.

A caixa de mensagem

Vamos passar agora para o código dentro da função `WinMain()`. A função `MessageBox()` é uma função da Win32 API que mostra uma caixa de mensagem para o usuário com alguns botões e ícones pré-definidos.

```
int MessageBox(
  HWND hWnd, // identificador da janela-pai
  LPCTSTR lpText, // ponteiro para texto da caixa de mensagem
  LPCTSTR lpCaption, // ponteiro para título da caixa de mensagem
  UINT uType // estilo da caixa de mensagem
);
```

O primeiro parâmetro `HWND hWnd` identifica a janela-pai da caixa de mensagem, ou seja, para qual janela do sistema a caixa de mensagem pertence. Se o parâmetro for `NULL`, a caixa de mensagem não pertence a nenhuma janela.

> **Nota:** o tipo de variável `HWND` armazena um identificador de janela, que pode ser considerado como uma variável `int` contendo um número exclusivo para cada janela criada no Windows. Assim, para manipularmos ou obter as informações de uma janela, utilizamos esse identificador. Você observará que na programação Windows eles sempre serão utilizados e que existem diferentes tipos de identificadores.

Os próximos dois parâmetros, do tipo `LPCTSTR`, são ponteiros de string; seus conteúdos serão utilizados para mostrar o texto da caixa de mensagem e o título da caixa de mensagem, respectivamente. Olhando para o programa-exemplo em execução e seu código-fonte, é possível verificar como esses parâmetros funcionam no `MessageBox()`.

O último parâmetro define que tipo de botão e ícone aparecerá na caixa de mensagem. Existem diversos *flags* para esse parâmetro que podem ser combinados através do operador `|` (ou bit-a-bit). Os principais *flags* estão listados na Tabela 1.3 abaixo.

Valor	Descrição
`MB_ABORTRETRYIGNORE`	A caixa de mensagem contém três botões: *Anular*, *Repetir* e *Ignorar*.
`MB_OK`	A caixa de mensagem contém um botão *OK* (esse é o padrão)
`MB_OKCANCEL`	A caixa de mensagem contém os botões *OK* e

	Cancelar.
`MB_RETRYCANCEL`	A caixa de mensagem contém os botões *Repetir* e *Cancelar*.
`MB_YESNO`	A caixa de mensagem contém os botões *Sim* e *Não*.
`MB_YESNOCANCEL`	A caixa de mensagem contém os botões *Sim*, *Não* e *Cancelar*.
`MB_ICONEXCLAMATION`	Um ícone de ponto de exclamação aparecerá na caixa de mensagem.
`MB_ICONINFORMATION`	Um ícone com a letra "i" aparecerá na caixa de mensagem.
`MB_ICONQUESTION`	Um ícone de ponto de interrogação aparecerá na caixa de mensagem.
`MB_ICONSTOP`	Um ícone de sinal de parada aparecerá na caixa de mensagem.
`MB_DEFBUTTON1`	O primeiro botão da caixa de mensagem é o botão padrão (*default*), a menos que `MF_DEFBUTTON2`, `MF_DEFBUTTON3` ou `MF_DEFBUTTON4` seja especificado.
`MB_DEFBUTTON2`	O segundo botão da caixa de mensagem é o botão padrão.
`MB_DEFBUTTON3`	O terceiro botão da caixa de mensagem é o botão padrão.
`MB_DEFBUTTON4`	O quarto botão da caixa de mensagem é o botão padrão.

Tabela 1.3: Os flags que podem ser utilizados no `MessageBox()`.

Caso você queira apresentar uma caixa de mensagem com o título "*Erro!*", na qual a janela-pai tenha como identificador o nome da variável `hWnd`, com a mensagem "*Arquivo não foi salvo. Salvar antes de sair?*", os botões *Sim* e *Não* e também um ícone com ponto de interrogação, utilize o seguinte código:

```
MessageBox(hWnd, "Arquivo não foi salvo. Salvar antes de sair?", "Erro!",
MB_YESNO | MB_ICONQUESTION);
```

Dica: tenha sempre em mãos o arquivo de ajuda do Platform SDK, pois ele é o melhor guia de consulta para as funções da Win32 API. Na internet é possível encontrar esses arquivos, que também está incluso em alguns compiladores. Outra ótima referência é o *MSDN* (*Microsoft Developer Network*), que pode ser consultado online pelo site http://www.msdn.com.

A função `MessageBox()` retorna um valor inteiro, que pode ser utilizado para verificar qual botão foi clicado pelo usuário; assim podemos fazer um algoritmo apropriado para cada escolha. Os valores retornados pela função estão listados na Tabela 1.4.

Valor	Descrição
`IDABORT`	Botão *Anular* foi pressionado.
`IDCANCEL`	Botão *Cancelar* foi pressionado.
`IDIGNORE`	Botão *Ignorar* foi pressionado.
`IDNO`	Botão *Não* foi pressionado.
`IDOK`	Botão *OK* foi pressionado.
`IDRETRY`	Botão *Repetir* foi pressionado.
`IDYES`	Botão *Sim* foi pressionado.

Tabela 1.4: Valores retornado pela função `MessageBox()`.

Nota: caso a caixa de mensagem contenha o botão *Cancelar*, a função retorna `IDCANCEL` se a tecla ESC for pressionada ou se o botão *Cancelar* for clicado. Se a caixa de mensagem não tiver o botão *Cancelar*, pressionar ESC não terá efeito algum – a caixa de mensagem será fechada, porém, nenhum valor será retornado pela função.

A linha após a chamada da função `MessageBox()` indica o fim da execução do programa, retornando o valor zero. Sempre que um programa terminar antes de entrar num laço de mensagens, ele deve retornar zero.

Como primeiro programa-exemplo, podemos perceber que há diversos detalhes e opções para a criação de programas Windows. A seguir, veremos como codificar um verdadeiro programa Windows, criando sua janela e obtendo e respondendo algumas mensagens do sistema.

Capítulo 2 – As Peças do Programa

Agora que você já aprendeu sobre a função `WinMain()`, está na hora de aprofundar o código dentro dela, criando uma programa com as principais etapas que necessitam ser realizadas para que você monte sua primeira janela Windows com a Win32 API, como mostra a Figura 2.1. A listagem do código-fonte do segundo programa-exemplo encontra-se no final do capítulo (arquivo *prog02.cpp* no CD-ROM), após a explicação passo-a-passo das etapas da criação da janela/programa.

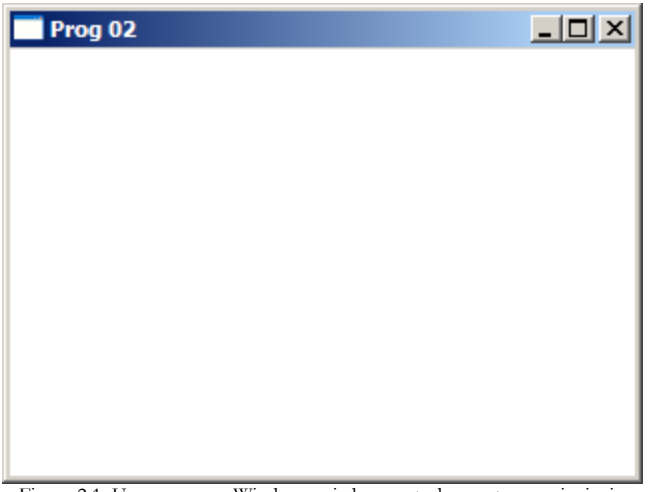

Figura 2.1: Um programa Windows criado com todas as etapas principais.

Definindo a classe

A primeira etapa para a criação da janela do seu programa é declarar uma classe da janela. Apesar de ser chamada de "classe", esse novo tipo de dado é uma *struct* que armazena as informações sobre a janela do programa. Sua estrutura é definida como:

```
typedef struct _WNDCLASSEX {
  UINT cbSize; // tamanho da estrutura, em bytes
  UINT style; // estilo da classe
  WNDPROC lpfnWndProc; // ponteiro para a função que processa mensagens
  int cbClsExtra; // bytes extras a serem alocados depois da estrutura da
janela
  int cbWndExtra; // bytes extras a serem alocados depois da instância da
janela
  HANDLE hInstance; // identificador da instância da janela
  HICON hIcon; // identificador do ícone
```

```
  HCURSOR hCursor; // identificador do cursor
  HBRUSH hbrBackground; // identificador da cor de fundo da janela
  LPCTSTR lpszMenuName; // nome do menu do programa
  LPCTSTR lpszClassName; // nome da classe da janela
  HICON hIconSm; // identificador do ícone na barra de título
} WNDCLASSEX;
```

Após declarar uma variável do tipo `WNDCLASSEX`, inicializamos todos os membros da estrutura. Cada membro tem um método para sua inicialização, com exemplos demonstrados logo abaixo e também inclusos no código-fonte do segundo programa-exemplo.

O membro `cbSize` deve ser inicializado como `sizeof(WNDCLASSEX)`, pois especifica qual o tamanho da estrutura.

```
WNDCLASSEX.cbSize = sizeof(WNDCLASSEX);
```

O membro `style` é usado para definir as propriedades de estilo da janela, como desabilitar o item "*Fechar ALT+F4*" do menu de sistema, aceitar duplo-clique do mouse ou configurar as opções de atualização de imagem da janela. Existem diversos *flags* que podem ser combinados com o operador `|` (ou bit-a-bit), conforme a Tabela 2.1.

```
WNDCLASSEX.style = CS_HREDRAW | CS_VREDRAW;
```

Para que o programa possa processar as mensagens enviadas pelo Windows, o membro `lpfnWndProc` é apontada para a função `WindowProc()`, uma função onde todos as mensagens do Windows são processadas para o seu programa (será comentada mais tarde nesse capítulo). Note que `WindowProc()` quando apontada por `lpfnWndProc` é escrita sem os parênteses, como no exemplo:

```
WNDCLASSEX.lpfnWndProc = (WNDPROC)WindowProc;
```

Os membros `cbClsExtra` e `cbWndExtra` geralmente não são utilizados; servem para reservar alguns bytes extras na estrutura e/ou na instância da janela para armazenar dados, mas é muito mais prático reservar memória através de outras variáveis. Portanto, ambos os membros recebem zero.

```
WNDCLASSEX.cbClsExtra = 0;
WNDCLASSEX.cbWndExtra = 0;
```

Valor	Descrição
CS_DBLCLKS	Envia mensagens de duplo-clique do mouse para o programa.
CS_HREDRAW	Redesenha toda a janela se um ajuste de movimento ou tamanho foi feito na horizontal da janela.
CS_NOCLOSE	Desabilita o menu "Fechar ALT+F4" do menu da janela.
CS_OWNDC	Aloca um contexto de dispositivo para cada janela da classe.
CS_VREDRAW	Redesenha toda a janela se um ajuste de movimento ou tamanho foi feito na vertical da janela.

Tabela 2.1: Alguns estilos de janela.

Dica: novamente, sempre consulte a documentação do *Platform SDK* / Win32 API, onde você encontrará todos os valores que podem ser passados para as variáveis e funções.

O membro `hInstance` armazena a instância do programa e recebe o parâmetro `hInstance` da função `WinMain()`.

```
WNDCLASSEX.hInstance = hInstance;
```

Os membros `hIcon` e `hIconSm` são quase idênticos: o primeiro serve para definir o ícone que será utilizado para o arquivo executável, atalho e também para quando o usuário pressionar ALT+TAB para alternar entre os programas em execução. O segundo serve para definir o ícone que é utilizado na barra de título e também na barra de tarefas do Windows. Se `hIconSm` for `NULL`, o ícone de `hIcon` será utilizado para ambos os membros.

Para carregar ícones, utilizamos a função `LoadIcon()` da Win32 API:

```
HICON LoadIcon(
  HINSTANCE hInstance, // identificador da instância do programa
  LPCTSTR lpIconName // nome do ícone ou identificador do recurso de ícone
);
```

Nessa função, o parâmetro `HINSTANCE hInstance` é identificado somente se o ícone que for usado está num arquivo de recursos (ou seja, é um ícone

personalizado). No caso, devemos enviar o parâmetro `hInstance` da `WinMain()` e o nome do ícone em `LPCTSTR lpIconName`.

Caso o ícone a ser utilizado seja um dos pré-definidos pelo sistema, o parâmetro `hInstance` deverá ser `NULL` e `lpIconName` receberá um dos valores da Tabela 2.2. Exemplo:

```
WNDCLASSEX.hIcon = LoadIcon(NULL, IDI_APPLICATION);
WNDCLASSEX.hIconSm = LoadIcon(NULL, IDI_ASTERISK);
```

Valor	Descrição
`IDI_APPLICATION`	Ícone padrão (parecido com um ícone de programas MS-DOS).
`IDI_ASTERISK`	Idem a `IDI_INFORMATION`.
`IDI_ERROR`	Ícone círculo vermelho com um X.
`IDI_EXCLAMATION`	Idem a `IDI_WARNING`.
`IDI_HAND`	Idem a `IDI_ERROR`.
`IDI_INFORMATION`	Ícone com ponto de exclamação num balão branco.
`IDI_QUESTION`	Ícone com ponto de interrogação num balão branco.
`IDI_WARNING`	Ícone com ponto de exclamação numa placa amarela.
`IDI_WINLOGO`	Ícone com o logotipo do Windows.

Tabela 2.2: Ícones pré-definidos pelo sistema.

O membro `hCursor` é utilizado para carregar o cursor de mouse padrão no programa. Para isso, utilizamos a função `LoadCursor()` da Win32 API:

```
HCURSOR LoadCursor(
  HINSTANCE hInstance, // identificador da instância do programa
  LPCTSTR lpCursorName // nome do cursor ou identificador do recurso de cursor
);
```

A função `LoadCursor()` trabalha semelhante à `LoadIcon()`. O primeiro parâmetro deve ser identificado se o cursor utilizado for extraído de um arquivo de recursos, caso contrário, será `NULL`. Quando `hInstance` é `NULL`, significa que o cursor a ser utilizado é um dos cursores pré-definidos pelo sistema e `lpCursorName` receberá um dos valores da Tabela 2.3. Exemplo:

```
WNDCLASSEX.hCursor = LoadCursor(NULL, IDC_UPARROW);
```

Valor	Descrição
IDC_APPSTARTING	Cursor padrão com pequena ampulheta.
IDC_ARROW	Cursor padrão.
IDC_CROSS	Cursor em forma de cruz.
IDC_HELP	Cursor seta e ponto de interrogação.
IDC_IBEAM	Cursor de texto (*I-beam*).
IDC_NO	Cursor com símbolo de proibido.
IDC_SIZEALL	Cursor com setas na vertical e horizontal.
IDC_SIZENESW	Cursor com setas na diagonal para direita.
IDC_SIZENS	Cursor com setas na vertical.
IDC_SIZENWSE	Cursor com setas na diagonal para esquerda.
IDC_SIZEWE	Cursor com setas na horizontal.
IDC_UPARROW	Cursor com seta para cima.
IDC_WAIT	Cursor de ampulheta.

Tabela 2.3: Cursores pré-definidos pelo sistema.

Nota: as funções `LoadIcon()` e `LoadCursor()` retornam valores do tipo `HICON` e `HCURSOR`, respectivamente (derivados da estrutura `HANDLE`). Todas as funções que têm retorno do tipo `HANDLE` (ou derivados) retornam o identificador (*handle*) do tipo especificado se a função foi executada corretamente ou retornam `NULL` no caso da execução do algoritmo da função falhar.

O próximo membro, `HBRUSH hbrBackground`, define a cor do fundo da janela através de um identificador de pincel (*brush*). Pincéis e identificadores gráficos serão discutidos em capítulos mais avançados, portanto não entrarei em detalhes nesse momento. Por enquanto, basta saber que o Windows utiliza pincéis e canetas (variável do tipo `HPEN`) para desenhar gráficos no programa e que por isso um pincel precisa estar associado à janela (no caso, para pintar o fundo dela com alguma cor). É possível criar pincéis ou utilizar os padrões do Windows; como esse assunto será tratado mais pra frente, no exemplo desse capítulo usaremos apenas os padrões já existentes.

```
WNDCLASSEX.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
```

Para utilizar os pincéis padrões, foi utilizado a função `GetStockObject()`, que retorna um identificador para objetos gráficos pré-definidos (ver protótipo da função abaixo) e recebe como parâmetro o tipo do objeto pré-definido, conforme a Tabela 2.4.

```
HGBIOBJ GetStockObject(
  int fnObject // tipo de objeto pré-definido
);
```

Nota: no comando `WNDCLASSEX.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH)`, perceba que foi feito um *type-casting* antes do nome da função. Esse type-casting é necessário, pois a função `GetStockObject()` retorna um identificador de objeto gráfico genérico (`HGDIOBJ`), enquanto o membro `hbrBackground` é do tipo `HBRUSH`. Sem essa conversão, ocorreria um erro de compilação.

Valor	Descrição
`BLACK_BRUSH`	Pincel preto.
`DKGRAY_BRUSH`	Pincel cinza escuro.
`GRAY_BRUSH`	Pincel cinza.
`HOLLOW_BRUSH`	Pincel transparente (idem a `NULL_BRUSH`).
`LTGRAY_BRUSH`	Pincel cinza claro.
`NULL_BRUSH`	Pincel transparente (idem a `HOLLOW_BRUSH`).
`WHITE_BRUSH`	Pincel branco.

Tabela 2.4: Pincéis pré-definidos do Windows.

O membro `LPCTSTR lpszMenuName` indica qual será o menu atribuído à classe da janela. A criação e utilização de menus serão estudadas mais adiante, no capítulo sobre arquivos de recursos. Por ora, não iremos utilizar nenhum menu e usamos a seguinte atribuição:

```
WNDCLASSEX.lpszMenuName = NULL;
```

Resta agora apenas um membro, `LPCTSTR lpszClassName`. Este é utilizado para especificar o nome da classe da janela, que será utilizado como referência para o Windows e funções Win32 API que necessitam informações dessa classe. Para esse membro, atribuímos uma string:

```
WNDCLASSEX.lpszClassName = "Nome da Classe";
```

Registrando a classe

Com a classe da janela já inicializada, a próxima etapa é registrá-la para que o Windows permita que você crie janelas a partir das informações fornecidas dentro da classe. É preciso registrar a classe antes de tentar criar

qualquer janela! Para isso, utilizamos a função RegisterClassEx(), que tem o seguinte protótipo:

```
ATOM RegisterClassEx(
  CONST WNDCLASSEX *lpwcx // endereço da estrutura com os dados da classe
);
```

A função recebe apenas um parâmetro, `CONST WNDCLASSEX *lpwcx`, que é um ponteiro para a classe da janela que foi criada anteriormente. Ela retorna zero se o registro da classe falhou ou retorna o valor de um `ATOM` que identifica exclusivamente a classe que está sendo registrada.

Nota: o tipo `ATOM` é um inteiro que faz referência à um banco de dados de strings definida e mantida pelo Windows; é como uma identidade única de cada programa que está sendo executado, assim como o nome da classe de uma janela.

Criando a janela

Depois de definir a classe da janela e registrá-la, podemos então criar a janela principal do programa. Para isso, utilizamos a seguinte função:

```
HWND CreateWindowEx(
  DWORD dwExStyle, // estilos extras da janela
  LPCTSTR lpClassName, // ponteiro string para o nome da classe registrada
  LPCTSTR lpWindowName, // ponteiro string para o título da janela
  DWORD dwStyle, // estilo da janela
  int x, // posição horizontal da janela
  int y, // posição vertical da janela
  int nWidth, // largura da janela
  int nHeight, // altura da janela
  HWND hWndParent, // identificador da janela-pai
  HMENU hMenu, // identificador do menu
  HINSTANCE hInstance, // identificador da instância do programa
  LPVOID lpParam // ponteiro para dados de criação da janela
);
```

Muitos desses parâmetros são auto-explicáveis, como `int x`, `int y`, `int nWidth`, `int nHeight`. Para eles, basta informar em valores inteiros a posição (x, y) onde a janela será inicialmente criada e a largura e altura da mesma.

O parâmetro `LPCTSTR lpClassName` deve ser o mesmo especificado no membro `lpszClassName` da classe da janela. Já `LPCTSTR lpWindowName` indica o texto que irá como título da janela (aparece na barra de título e na barra de tarefas do Windows).

Como a janela é a principal do programa, não existe nenhum identificador da janela-pai (a não ser que levemos em conta o próprio Windows) e, portanto, `HWND hWndParent` deve ser `NULL` (ou `HWND_DESKTOP` se levarmos em conta o Windows como janela-pai).

O parâmetro `HMENU hMenu` recebe o identificador do menu principal do programa, mas sempre iremos deixá-lo como `NULL` pois não vamos utilizar menus por ora, e, ao utilizarmos, carregaremos os menus através de funções da Win32 API.

`HINSTANCE hInstance` armazena o identificador da instância do programa e deve receber o parâmetro `hInstance` da `WinMain()`.

O último parâmetro da função, `LPVOID lpParam`, não é utilizado em grande parte dos programas; serve para recursos avançados que para nossas aplicações não terão importância, portanto esse parâmetro deve receber `NULL`.

Deixei para o final a explicação de dois parâmetros, `DWORD dwExStyle` e `DWORD dwStyle`. Ambos servem para modificar as propriedades do estilo da janela, como barra de título, menu do sistema, alinhamento, entre outros. O parâmetro `dwStyle` especifica o estilo da janela e `dwExStyle` especifica propriedades extras quanto ao estilo. Os principais valores que `dwStyle` podem receber são fornecidos na Tabela 2.5; a Tabela 2.6 contém os principais valores que podem ser recebidos por `dwExStyle`. Lembre-se que em ambos os casos, são possíveis combinar os valores através do operador `|` (ou bit-a-bit).

Estilo	Descrição
`WS_BORDER`	Cria uma janela com borda fina.
`WS_CAPTION`	Cria uma janela com barra de título (inclui o estilo `WS_BORDER`).
`WS_CHILD`	Cria uma janela-filho. Esse estilo não pode ser utilizado junto com o estilo `WS_POPUP`.
`WS_CLIPCHILDREN`	Exclui a área ocupada pelas janelas-filho quando é necessário fazer atualização da janela-pai. Esse estilo é usado quando se cria uma janela-pai.
`WS_DISABLED`	Cria uma janela inicialmente desabilitada. Uma janela desse estilo não pode receber

	entradas do usuário.
WS_DLGFRAME	Cria uma janela com borda típica de caixas de diálogo. Não possui barra de título.
WS_HSCROLL	Cria uma janela com barra de rolagem horizontal.
WS_MAXIMIZE	Cria uma janela inicialmente maximizada.
WS_MAXIMIZEBOX	Cria uma janela que contém o botão de Maximizar. Não pode ser combinado com o estilo WS_EX_CONTEXTHELP e o estilo WS_SYSMENU deve ser especificado.
WS_MINIMIZE	Cria uma janela inicialmente minimizada.
WS_MINIMIZEBOX	Cria uma janela que contém o botão de Minimizar. Não pode ser combinado com o estilo WS_EX_CONTEXTHELP e o estilo WS_SYSMENU deve ser especificado.
WS_OVERLAPPED	Cria uma janela com borda e barra de título.
WS_OVERLAPPEDWINDOW	Cria uma janela com os estilos WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX e WS_MAXIMIZEBOX.
WS_POPUP	Cria uma janela *pop-up*. Não pode ser utilizado com o estilo WS_CHILD.
WS_POPUPWINDOW	Cria uma janela *pop-up* com os estilos WS_BORDER, WS_POPUP e WS_SYSMENU. Os estilos WS_POPUPWINDOW e WS_CAPTION devem ser combinados para deixar a janela visível.
WS_SIZEBOX	Cria uma janela que contém uma borda de ajuste de tamanho.
WS_SYSMENU	Cria uma janela que tem o menu de sistema na barra de título. O estilo WS_CAPTION também deve ser especificado.
WS_TABSTOP	Especifica um controle que pode receber o cursor do teclado quando o usuário pressiona a tecla TAB. Ao pressionar a tecla TAB, o cursor do teclado muda para o próximo controle com o estilo WS_TABSTOP.
WS_VISIBLE	Cria uma janela inicialmente visível.
WS_VSCROLL	Cria uma janela com barra de rolagem vertical.

Tabela 2.5: Principais estilos da janela.

Estilo	Descrição
`WS_EX_ACCEPTFILES`	A janela com esse estilo aceita arquivos arrastados para ela.
`WS_EX_CLIENTEDGE`	Cria uma borda afundada na janela.
`WS_EX_CONTROLPARENT`	Permite o usuário navegar entre as janelas-filho usando a tecla TAB.
`WS_EX_MDICHILD`	Cria uma janela-filho MDI.
`WS_EX_TOOLWINDOW`	Cria uma janela de ferramentas flutuante; contém uma barra de título menor que a normal e utiliza uma fonte menor para o título. Esse estilo de janela não aparece na barra de tarefas ou quando o usuário aperta as teclas ALT+TAB.
`WS_EX_TOPMOST`	Especifica que a janela criada deve permanecer acima de todas as outras janelas do sistema, mesmo que essa seja desativada.

Tabela 2.6: Principais estilos extras da janela.

A função `CreateWindowEx()` retorna um identificador para a janela criada, que é do tipo `HWND`. Para armazenar o identificador da janela, declaramos a seguinte variável:

```
HWND hWnd = NULL;
```

E logo em seguida, chamamos e armazenamos o retorno da função `CreateWindowsEx()` na nova variável `hWnd` com o seguinte comando:

```
hWnd = CreateWindowEx(parâmetros);
```

Nota: a função `CreateWindowEx()` é utilizada não somente para criar a janela principal do programa, mas também para criar botões, caixas de textos, textos estáticos (*label*), caixas de verificação, etc, pois todos esses controles são considerados janelas pela Win32 API. Acostume-se com o uso do termo *window* (janela) para todos esses controles, e não apenas para a "janela principal" do programa (também conhecida como formulário ou *form* em ambientes de programação como Inprise/Borland Delphi ou Microsoft Visual Basic).

Depois de criada a janela, essa deve ser mostrada para o usuário, através da chamada à função `ShowWindow()`, que tem o protótipo:

```
BOOL ShowWindow(
  HWND hWnd, // identificador da janela
  int nCmdShow // estado da janela
);
```

O primeiro parâmetro da função recebe o identificador da janela que será mostrado, no caso, devemos enviar a variável `hWnd` anteriormente criada. O segundo parâmetro, `int nCmdShow`, indica o estado inicial da janela. Lembra-se do último parâmetro da função `WinMain()`? Ele é utilizado aqui, onde repassamos seu valor para a função `ShowWindow()`. Assim, a chamada à função deve ser feita como no exemplo:

```
ShowWindow(hWnd, nCmdShow);
```

O retorno dessa função é zero se a janela estava anteriormente escondida ou um valor diferente de zero se a janela estava previamente visível.

Além da função `ShowWindow()`, quando o programa é executado, chamamos outra função, `UpdateWindow()`, para que ela envie uma mensagem `WM_PAINT` diretamente ao programa, indicando que é preciso mostrar o conteúdo da janela. O protótipo da função é:

```
BOOL UpdateWindow(
  HWND hWnd // identificador da janela
);
```

O único parâmetro da função recebe o identificador da janela que será atualizado. Se a função falhar, ela retorna zero; caso contrário, o retorno será um valor diferente de zero.

O loop de mensagens

A próxima etapa é criar um *loop* onde serão verificadas todas as mensagens que o Windows enviará para a fila de mensagens do programa. Essa etapa consiste em três passos: em primeiro lugar, o programa deve receber a mensagem que está na fila; em seguida, traduz códigos de teclas virtuais ou aceleradoras (de atalho) em mensagens de caracteres/teclado, para finalmente despachar as mensagens para a função que processam elas. Porém, antes do loop, declaramos uma variável do tipo `MSG` que armazenará a mensagem atual, com as seguintes informações:

```
typedef struct tagMSG {
  HWND hWnd; // identificador da janela que recebe as mensagens
```

```
  UINT message; // número da mensagem
  WPARAM wParam; // informação adicional da mensagem
  LPARAM lParam; // informação adicional da mensagem
  DWORD time; // horário que a mensagem foi postada
  POING pt; // posição do cursor quando a mensagem foi postada
} MSG;
```

Nota: teclas virtuais ou aceleradoras são códigos independentes do sistema para várias teclas, que incluem as teclas de função F1-F12, setas e combinações de teclas, como CTRL+C ou ALT+A, por exemplo. Sem a tradução dessas teclas, o programa não irá processar diversos eventos envolvendo o teclado, como o acesso de menus (caso seja feito via teclado) ou atalhos como acessar o arquivo de ajuda pela tecla F1.

Durante toda a codificação, iremos manipular a variável `MSG` somente através das funções `GetMessage()`, `TranslateMessage()` e `DispatchMessage()`. A verificação de qual mensagem foi enviada e o processamento dela será feito pela função `WindowProc()`, que será explicada logo mais.

Vamos começar pela obtenção da mensagem a ser processada, com a função `GetMessage()`:

```
BOOL GetMessage(
  LPMSG lpMsg, // ponteiro para estrutura de mensagem
  HWND hWnd, // identificador da janela
  UINT wMsgFilterMin, // primeira mensagem
  UINT wMsgFilterMax // última mensagem
);
```

O primeiro parâmetro recebe um ponteiro para a variável do tipo `MSG` para que a sua estrutura seja preenchida automaticamente pela função.

O parâmetro `HWND hWnd` receberá o identificador da janela de onde as mensagens estão sendo obtidas. Normalmente, deixamos esse parâmetro como `NULL`, pois assim podemos processar qualquer mensagem que pertença ao programa, e não somente à uma janela.

Os dois últimos parâmetros são utilizados para especificar o intervalo inicial (`wMsgFilterMin`) e o final (`wMsgFilterMax`) do valor das mensagens que serão processadas. Indicando zero para esses parâmetros, a função processará todas as mensagens disponíveis.

A função pode retornar três valores diferentes: zero se a mensagem obtida for WM_QUIT; -1 caso tenha ocorrido um erro; ou um número diferente de zero caso não tenha ocorrido nenhuma das alternativas anteriores.

Depois de obtida a mensagem, as mensagens de teclas virtuais serão traduzidas para mensagens de caracteres com a função TranslateMessage():

```
BOOL TranslateMessage(
  CONST MSG *lpMsg // ponteiro para estrutura de mensagem
);
```

O único parâmetro da função recebe um ponteiro para a variável do tipo MSG para que sua mensagem seja traduzida (caso a mensagem seja de teclas virtuais). O retorno é diferente de zero se a mensagem foi traduzida (uma mensagem de ação de teclado foi enviada à fila de mensagens) ou zero se não foi traduzida.

Para finalizar a codificação do loop, devemos enviar a mensagem obtida para a função que processa as mensagens do programa, através da função DispatchMessage():

```
LONG DispatchMessage(
  CONST MSG *lpmsg // ponteiro para estrutura de mensagem
);
```

O parâmetro CONST MSG *lpmsg recebe um ponteiro para a estrutura de mensagem MSG indicando que essa estrutura será enviada ao processador de mensagens. O retorno da função especifica o valor retornado pelo processador de mensagens; esse valor depende da mensagem que foi enviada, porém geralmente o valor retornado pode ser ignorado.

O loop de mensagens é codificado conforme a Listagem 2.1:

```
MSG msg;
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

Listagem 2.1: O loop de mensagens.

Note que, no comando while(), é verificado se a função GetMessage() retorna um valor maior que zero, pois ela também pode retornar –1 no caso de erro.

Dica: o loop de mensagens (e, conseqüentemente, o programa) pára de ser executado quando `GetMessage()` retornar o valor `WM_QUIT`.

Para uma melhor visualização e entendimento de como o loop de mensagens e a função `WindowProc()` se comunicam, veja a Figura 2.2.

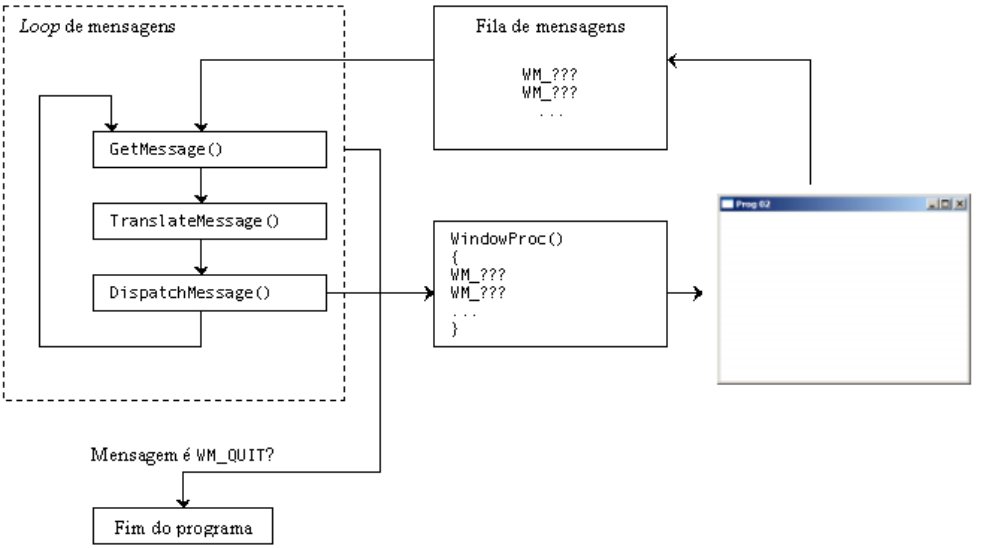

Figura 2.2: Comunicação entre o loop de mensagens e a `WindowProc()`.

Processando mensagens

Para o funcionamento do nosso programa, é preciso que as mensagens enviadas pelo Windows sejam processadas, e isso é feito através da função `WindowProc()`, que é uma função já definida para todos os programas (essa é a tal função especial que foi mencionada no segundo parágrafo do capítulo 1). Seu protótipo tem o seguinte aspecto:

```
LRESULT CALLBACK WindowProc(
  HWND hWnd, // identificador da janela
  UINT uMsg, // identificador da mensagem
  WPARAM wParam, // primeiro parâmetro da mensagem
  LPARAM lParam // segundo parâmetro da mensagem
);
```

Essa é uma função diferente de todas as outras do seu programa, pois você não faz nenhuma chamada à ela; o Windows faz por você. Por esse

motivo, a função tem o retorno do tipo `LRESULT CALLBACK`, significando que ela "é chamada pelo sistema e retorna ao Windows". O valor de retorno é o resultado do processamento da mensagem, o qual depende da mensagem enviada.

Quanto aos seus parâmetros, a função `WindowProc()` recebe um identificador da janela principal (`HWND hWnd`), a mensagem a ser processada, `UINT uMsg` (note que essa mensagem não é a mesma do loop de mensagens – a do loop é uma variável do tipo `MSG`), e dois parâmetros (`WPARAM wParam` e `LPARAM lParam`) que podem conter informações adicionais sobre a mensagem – por exemplo, na mensagem de movimento de mouse (`WM_MOUSEMOVE`), `wParam` indica se algum botão do mouse está pressionado e `lParam` armazena a posição (x, y) do cursor do mouse.

Existem centenas de mensagens que o seu programa poderá receber pelo Windows, porém nem sempre todas são verificadas (a Tabela 2.7 lista as principais mensagens que são utilizadas/processadas). Para essas, devemos deixar o próprio Windows processá-las, através do uso de uma função chamada `DefWindowProc()`, que é uma função de processamento de mensagens padrão do Windows. Ela recebe os mesmos parâmetros da `WindowProc()` e processa qualquer mensagem que não foi feita pelo seu programa, assegurando que todas as mensagens sejam verificadas e processadas.

Mensagem	Descrição
`WM_ACTIVATE`	Enviada quando a janela é ativada.
`WM_CHAR`	Enviada quando uma mensagem `WM_KEYDOWN` é traduzida pela função `TranslateMessage()`.
`WM_CLOSE`	Enviada quando a janela é fechada.
`WM_COMMAND`	Enviada quando o usuário seleciona um item de um menu, quando um controle (botão, *scrollbar*, etc) envia uma mensagem para a janela-pai ou quando uma tecla de atalho é ativada.
`WM_CREATE`	Enviada quando a janela é criada pela função `CreateWindowEx()`. Essa mensagem é enviada à função de processamento de mensagens após a janela ser criada, mas antes dela se tornar visível.
`WM_DESTROY`	Enviada quando a janela é destruída. Essa mensagem é enviada à função de

	processamento de mensagens após a janela ser removida da tela.
`WM_ENABLE`	Enviada quando o programa modifica o estado disponível/não disponível de uma janela. Essa mensagem é enviada para a janela que está tendo seu estado modificado.
`WM_KEYDOWN`	Enviada quando uma tecla que não é do sistema é pressionada, ou seja, quando a tecla ALT não está pressionada.
`WM_KEYUP`	Enviada quando uma tecla que não é do sistema é solta.
`WM_LBUTTONDOWN`	Enviada quando o usuário pressiona o botão esquerdo do mouse quando o cursor está dentro da janela do programa.
`WM_LBUTTONUP`	Enviada quando o usuário solta o botão esquerdo do mouse quando o cursor está dentro da janela do programa.
`WM_MBUTTONDOWN`	Enviada quando o usuário pressiona o botão central do mouse quando o cursor está dentro da janela do programa.
`WM_MBUTTONUP`	Enviada quando o usuário solta o botão central do mouse quando o cursor está dentro da janela do programa.
`WM_RBUTTONDOWN`	Enviada quando o usuário pressiona o botão direito do mouse quando o cursor está dentro da janela do programa.
`WM_RBUTTONUP`	Enviada quando o usuário solta o botão direito do mouse quando o cursor está dentro da janela do programa.
`WM_MOUSEMOVE`	Enviada quando o cursor do mouse é movido.
`WM_MOVE`	Enviada após a janela ser movida.
`WM_PAINT`	Enviada quando a janela precisa ser atualizada.
`WM_QUIT`	Enviada quando o programa chama a função `PostQuitMessage()`.
`WM_SHOWWINDOW`	Enviada quando a janela for escondida ou restaurada.
`WM_SIZE`	Enviada depois que a janela mudou de tamanho.
`WM_TIMER`	Enviada quando um *Timer* definido no

programa expira.

Tabela 2.7: Principais mensagens enviadas ao programa.

Segue a codificação (Listagem 2.2) demonstrando o corpo básico da função `WindowProc()` e como processar as mensagens:

```
//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  // Variáveis para manipulação da parte gráfica do programa
  HDC hDC = NULL;
  PAINTSTRUCT psPaint;

  // Verifica qual foi a mensagem enviada
  switch(uMsg)
  {

    case WM_CREATE: // Janela foi criada
    {
      // Retorna 0, significando que a mensagem foi processada corretamente
      return(0);
    } break;

    case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
    {
      /* Devemos avisar manualmente ao Windows que a janela já foi
atualizada, pois não é um processo automático. Se isso não for feito, o
Windows não irá parar de enviar a mensagem WM_PAINT ao programa. */

      // O código abaixo avisa o Windows que a janela já foi atualizada.
      hDC = BeginPaint(hWnd, &psPaint);
      EndPaint(hWnd, &psPaint);

      return(0);
    } break;

    case WM_CLOSE: // Janela foi fechada
    {
      // Destrói a janela
      DestroyWindow(hWnd);

      return(0);
    } break;

    case WM_DESTROY: // Janela foi destruída
    {
      // Envia mensagem WM_QUIT para o loop de mensagens
      PostQuitMessage(WM_QUIT);

      return(0);
    } break;

    default: // Outra mensagem
    {
```

```
      /* Deixa o Windows processar as mensagens que não foram verificadas na
função */
      return(DefWindowProc(hWnd, uMsg, wParam, lParam));
    }

  }
}
```

Listagem 2.2: Exemplo da função `WindowsProc()`.

No exemplo acima, foram verificadas apenas quatro mensagens, `WM_CREATE`, `WM_PAINT`, `WM_CLOSE` e `WM_DESTROY`. Note que sempre quando uma mensagem for processada, devemos retornar zero, indicando que a mensagem já foi processada e que o Windows não precisa tentar processá-la novamente. Já as mensagens que não foram processadas são passadas para a função `DefWindowProc()`, onde o Windows faz o trabalho do processo das mensagens.

Quando é enviada uma mensagem `WM_CLOSE` para a janela, o programa destrói a janela através da função `DestroyWindow()`, o que acaba gerando outra mensagem, a `WM_DESTROY`.

```
BOOL DestroyWindow(
  HWND hWnd // identificador da janela a ser destruída
);
```

`DestroyWindow()` recebe como parâmetro o identificador da janela que será destruída. A função também destrói o menu da janela, *timers* e limpa a fila de mensagens, além de gerar a mensagem `WM_DESTROY`. No processamento da mensagem `WM_DESTROY`, chamamos a função `PostQuitMessage()`.

```
VOID PostQuitMessage(
  int nExitCode // código de saída
);
```

Essa função indica ao sistema que uma foi feito um pedido para que o programa termine. O parâmetro `int nExitCode` especifica o código de saída do programa e é utilizado como o parâmetro `wParam` da mensagem `WM_QUIT` (lembre-se que após o loop de mensagens receber `WM_QUIT`, o programa é finalizado com a `WinMain()` retornando o valor de `msg.wParam`).

O processo da mensagem `WM_PAINT` será discutido no capítulo 4, que introduz a utilização da parte gráfica do Windows (*GDI – Graphics Device Interface*).

Agora vamos juntar todo o conhecimento obtido nesse capítulo e criar o programa demonstrado no início, conforme a Listagem 2.3:

```
//------------------------------------------------------------------------------
// prog02.cpp - Esqueleto completo de um programa Windows
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
// Bibliotecas
//------------------------------------------------------------------------------
#include <windows.h>

//------------------------------------------------------------------------------
// Definições
//------------------------------------------------------------------------------
// Nome da classe da janela
#define WINDOW_CLASS         "prog02"
// Título da janela
#define WINDOW_TITLE         "Prog 02"
// Largura da janela
#define WINDOW_WIDTH         320
// Altura da janela
#define WINDOW_HEIGHT        240

//------------------------------------------------------------------------------
// Protótipo das funções
//------------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

//------------------------------------------------------------------------------
// WinMain() -> Função principal
//------------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
  // Cria a classe da janela e especifica seus atributos
  WNDCLASSEX wcl;
  wcl.cbSize = sizeof(WNDCLASSEX);
  wcl.style = CS_HREDRAW | CS_VREDRAW;
  wcl.lpfnWndProc = (WNDPROC)WindowProc;
  wcl.cbClsExtra = 0;
  wcl.cbWndExtra = 0;
  wcl.hInstance = hInstance;
  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
  wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
  wcl.lpszMenuName = NULL;
  wcl.lpszClassName = WINDOW_CLASS;
  wcl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

  // Registra a classe da janela
  if(RegisterClassEx(&wcl))
  {
    // Cria a janela principal do programa
    HWND hWnd = NULL;
    hWnd = CreateWindowEx(
```

```
                  NULL,
                  WINDOW_CLASS,
                  WINDOW_TITLE,
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  (GetSystemMetrics(SM_CXSCREEN) - WINDOW_WIDTH) / 2,
                  (GetSystemMetrics(SM_CYSCREEN) - WINDOW_HEIGHT) / 2,
                  WINDOW_WIDTH,
                  WINDOW_HEIGHT,
                  HWND_DESKTOP,
                  NULL,
                  hInstance,
                  NULL);

    // Verifica se a janela foi criada
    if(hWnd)
    {
      // Mostra a janela
      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);

      // Armazena dados da mensagem que será obtida
      MSG msg;

      // Loop de mensagens, enquanto mensagem não for WM_QUIT,
      // obtém mensagem da fila de mensagens
      while(GetMessage(&msg, NULL, 0, 0) > 0)
      {
        // Traduz teclas virtuais ou aceleradoras (de atalho)
        TranslateMessage(&msg);

        // Envia mensagem para a função que processa mensagens (WindowProc)
        DispatchMessage(&msg);
      }

      // Retorna ao Windows com valor de msg.wParam
      return(msg.wParam);
     }
     // Se a janela não foi criada
     else
     {
       // Exibe mensagem de erro e sai do programa
       MessageBox(NULL, "Não foi possível criar janela.", "Erro!", MB_OK |
MB_ICONERROR);

       return(0);
     }
  }
  // Se a classe da janela não foi registrada
  else
  {
    // Exibe mensagem de erro e sai do programa
    MessageBox(NULL, "Não foi possível registrar a classe da janela.",
"Erro!", MB_OK | MB_ICONERROR);
    return(0);
  }

  // Retorna ao Windows sem passar pelo loop de mensagens
  return(0);
}
```

```
//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  // Variáveis para manipulação da parte gráfica do programa
  HDC hDC = NULL;
  PAINTSTRUCT psPaint;

  // Verifica qual foi a mensagem enviada
  switch(uMsg)
  {

    case WM_CREATE: // Janela foi criada
    {
      // Retorna 0, significando que a mensagem foi processada corretamente
      return(0);
    } break;

    case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
    {
      /* Devemos avisar manualmente ao Windows que a janela já foi
atualizada, pois não é um processo automático. Se isso não for feito, o
Windows não irá parar de enviar a mensagem WM_PAINT ao programa. */

      // O código abaixo avisa o Windows que a janela já foi atualizada.
      hDC = BeginPaint(hWnd, &psPaint);
      EndPaint(hWnd, &psPaint);

      return(0);
    } break;

    case WM_CLOSE: // Janela foi fechada
    {
      // Destrói a janela
      DestroyWindow(hWnd);

      return(0);
    } break;

    case WM_DESTROY: // Janela foi destruída
    {
      // Envia mensagem WM_QUIT para o loop de mensagens
      PostQuitMessage(WM_QUIT);

      return(0);
    } break;

    default: // Outra mensagem
    {
      /* Deixa o Windows processar as mensagens que não foram verificadas na
função */
      return(DefWindowProc(hWnd, uMsg, wParam, lParam));
    }

  }
}
```

Listagem 2.3: O programa completo para criar a janela.

C++ explicado: observe o código da função `WinMain()`, onde ocorre a declaração das variáveis `HWND hWnd` e `MSG msg` no meio da função. Em C++, é permitido declarar variáveis em qualquer parte do código, e não necessariamente no começo da função, como na linguagem C.

Nesse programa, foram declaradas quatro `#define` para facilitar um pouco a construção do programa, pois caso haja necessidade de modificar o nome da classe da janela, o título e/ou o tamanho da janela, podemos modificar essas `#define`, que são utilizadas em diferentes lugares do código.

Nota: a classe da janela poderia ser declarada e diretamente inicializada, eliminando a necessidade de digitar diversas linhas de atribuição aos membros da classe:

```
WNDCLASSEX wcl = {
  sizeof(WNDCLASSEX),
  CS_OWNDC | CS_HREDRAW | CS_VREDRAW,
  (WNDPROC)WindowProc,
  0,
  0,
  hInstance,
  LoadIcon(NULL,IDI_APPLICATION),
  LoadCursor(NULL,IDC_ARROW),
  (HBRUSH)GetStockObject(WHITE_BRUSH),
  NULL,
  WINDOW_CLASS,
  LoadIcon(NULL,IDI_APPLICATION)
};
```

O programa verifica se a classe da janela foi registrada e também se não ocorreram erros para criar a janela principal do programa. Caso alguma das duas etapas tenha falhado, uma mensagem de erro é emitida ao usuário e o programa finaliza sem passar pelo loop de mensagens.

Ao invés de pré-determinarmos uma posição (x, y) inicial da janela, para efeitos de estética, podemos iniciar o programa com sua janela bem no meio da tela, bastando utilizar uma simples fórmula que calcula a posição centralizada.

Para isso, devemos obter as configurações de largura e altura utilizadas pelo computador, usando a função `GetSystemMetrics()`, que foi chamada duas vezes dentro da função `CreateWindowEx()`:

```
HWnd = CreateWindowEx(
                  ...
                  (GetSystemMetrics(SM_CXSCREEN) - WINDOW_WIDTH) / 2,
                  (GetSystemMetrics(SM_CYSCREEN) - WINDOW_HEIGHT) / 2,
                  ...);
```

A função `GetSystemMetrics()` retorna diversos valores sobre a configuração do sistema. O seu protótipo é mostrado a seguir:

```
int GetSystemMetrics(
  int nIndex // valor da configuração a ser retornado
);
```

A função retorna zero se ocorreu algum erro ou então retorna o valor da configuração que foi requisitado no parâmetro `int nIndex`. O parâmetro da função pode receber diversos valores, como mostra a Tabela 2.8. Todas as dimensões retornadas pela função são em *pixels* (pontos); os valores que iniciam com `SM_CX*` são largura e os que começam com `SM_CY*` são altura.

Valor	Descrição
`SM_CLEANBOOT`	Valor que especifica como o sistema foi iniciado: 0 – Boot normal 1 – Boot em modo de segurança 2 – Boot em modo de segurança com rede
`SM_CMONITORS`	Número de monitores no desktop (apenas para Windows NT 5.0 ou superior e Windows 98 ou superior).
`SM_CMOUSEBUTTONS`	Número de botões do mouse, ou zero se não há mouse instalado.
`SM_CXBORDER`, `SM_CYBORDER`	Largura e altura da borda da janela. Equivalente ao valor de `SM_CXEDGE` e `SM_CYEDGE` para janelas com visual 3D.
`SM_CXCURSOR`, `SM_CYCURSOR`	Largura e altura do cursor.
`SM_CXEDGE`, `SM_CYEDGE`	Dimensões de uma borda 3D.
`SM_CXFIXEDFRAME`, `SM_CYFIXEDFRAME`	Espessura do quadro em volta do perímetro de uma janela que tem título, mas não é redimensionável. `SM_CXFIXEDFRAME` é a largura da borda horizontal e `SM_CYFIXEDFRAME` é a altura da borda vertical.

`SM_CXFULLSCREEN`, `SM_CYFULLSCREEN`	Largura e altura da área de uma janela em tela cheia.
`SM_CXMAXIMIZED`, `SM_CYMAXIMIZED`	Dimensões padrões de uma janela maximizada.
`SM_CXMIN`, `SM_CYMIN`	Largura e altura mínima de uma janela.
`SM_CXMINIMIZED`, `SM_CYMINIMIZED`	Dimensões de uma janela normal minimizada.
`SM_CXSCREEN`, `SM_CYSCREEN`	Largura e altura da tela do monitor.
`SM_CXSIZE`, `SM_CYSIZE`	Largura e altura de um botão na barra de título.
`SM_CYCAPTION`	Altura de uma área de título.
`SM_MOUSEPRESENT`	`TRUE` para mouse instalado ou `FALSE` caso contrário.
`SM_NETWORK`	O bit menos significante é acionado se há rede detectada.

Tabela 2.8: Alguns valores de configuração do sistema que podem ser obtidos com `GetSystemMetrics()`.

Enviando mensagens

Quando os usuários utilizam nossos programas, eles estão indiretamente gerando mensagens a serem processadas (cliques de mouse, uso do teclado, mudança de posição ou tamanho da janela, etc). Nós, como programadores, podemos enviar mensagens explicitamente para serem processadas pela `WindowProc()`.

Um exemplo do envio de mensagens pode ser a seguinte: quando o usuário pressiona a tecla F3 (gerando a mensagem `WM_KEYDOWN`, como veremos no capítulo 4), o texto da barra de título do nosso programa muda para "*F3 pressionado*". Nesse exemplo, o usuário está gerando apenas a mensagem `WM_KEYDOWN` quando aperta a tecla; a mudança do texto da barra de título deve ser feita via programação. Existe uma mensagem, `WM_SETTEXT`, que pode ser enviada ao programa para mudar o texto da janela. Mas como enviar mensagens à fila de mensagens do programa?

Podemos utilizar duas funções: `SendMessage()` e `PostMessage()`. A primeira função envia a mensagem especificada diretamente para a `WindowProc()` e não é retornada até que a mensagem seja processada. A segunda coloca a mensagem na fila de mensagens e retorna imediatamente

(independente da mensagem já ter sido processada ou não). Mensagens enviadas com `PostMessage()` são enviadas à `WindowProc()` pelo loop de mensagens.

```
LRESULT SendMessage(
  HWND hWnd, // identificador da janela-destino
  UINT Msg, // mensagem a ser enviada
  WPARAM wParam, // informação adicional da mensagem
  LPARAM lParam // informação adicional da mensagem
);
```

Essa função recebe o identificador da janela-destino da mensagem (pode ser a janela principal do programa, uma caixa de texto, um botão, etc) no primeiro parâmetro. A mensagem que será enviada é informada no segundo parâmetro; o terceiro (`wParam`) e o quarto (`lParam`) parâmetro podem receber informações adicionais da mensagem (caso a mesma necessite). O retorno da função indica o resultado do processamento da mensagem e depende da mensagem enviada.

```
BOOL PostMessage(
  HWND hWnd, // identificador da janela-destino
  UINT Msg, // mensagem a ser enviada
  WPARAM wParam, // informação adicional da mensagem
  LPARAM lParam // informação adicional da mensagem
);
```

A função `PostMessage()` recebe os mesmos parâmetros da função `SendMessage()`, porém, retorna um valor diferente de zero se não houver erros, ou zero caso contrário.

No exemplo citado nesse tópico, faríamos a codificação da Listagem 2.4 para modificar o texto da barra de título do programa. Não se preocupe sobre a mensagem `WM_KEYDOWN`, pois ela será estudada no capítulo 4.

```
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  switch(uMsg)
  {
    // mensagens...

    case WM_KEYDOWN: // Obtém tecla virtual
    {
      // Verifica se tecla pressionada foi F3
      if(wParam == VK_F3)
        // Envia mensagem WM_SETTEXT para a janela principal (hWnd)
        SendMessage(hWnd, WM_SETTEXT, (WPARAM)0, (LPARAM)"F3 pressionado");

      return(0);
```

```
    } break;

    // mensagens...
  }
}
```

Listagem 2.4: Enviando mensagens.

A mensagem `WM_SETTEXT` tem uma informação adicional em `lParam`, que recebe o texto que será utilizado na barra de título. Em `wParam` não há nenhuma informação e devemos passar zero para ela.

Chegamos ao fim desse capítulo. No próximo, veremos como trabalhar com os arquivos de recursos, adicionando ao programa ícones/cursores personalizados, imagens, sons, menus e caixas de diálogos.

Capítulo 3 – Arquivos de Recursos

A maioria dos programas para Windows possui menus, ícones personalizados (alguns incluem até cursores modificados) e caixas de diálogo (janelas como a da formatação de fonte em um editor de textos). Todos esses componentes (e mais alguns) do programa podem ser utilizados através do uso de arquivos de recursos (também conhecidos como *scripts* de recursos).

Arquivos de recursos podem ser criados em qualquer editor de textos no formato ASCII (como o Bloco de Notas), dentro da IDE do seu compilador, quando o mesmo possuir um editor de recursos (caso do Microsoft Visual C++ e Dev-C++) ou através de um programa editor de recursos. No caso de editores de recursos com interface visual (clique-e-arraste), o conteúdo do arquivo é gerado automaticamente pelo editor.

Esses arquivos têm extensão *.rc* e o conteúdo é definido com palavras-chave em inglês, indicando os recursos que serão utilizados pelo programa. Os arquivo são então compilados, criando-se arquivos binários com extensão *.res*, que são anexados ao final do programa executável, podendo ser carregados em tempo de execução (veja o esquema da Figura 3.1).

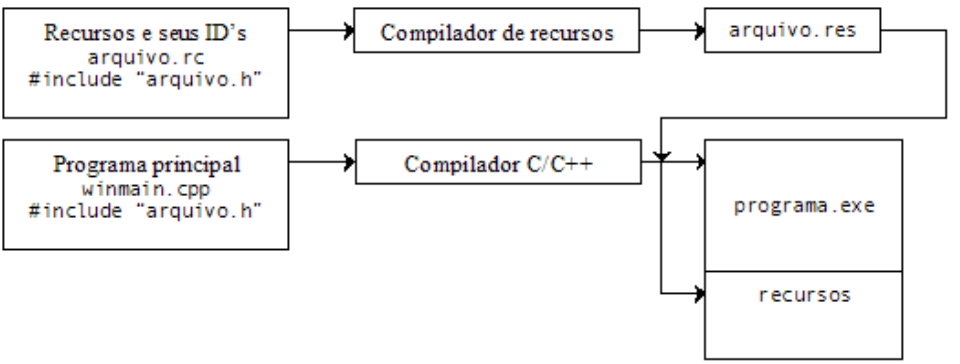

Figura 3.1: Inclusão dos recursos no programa executável.

Vamos aprender como criar um arquivo de recursos sem usar nenhum editor visual; assim, você poderá entender a estrutura do arquivo e como funcionam os editores.

ID's

Os recursos são reconhecidos pelo programa através de números inteiros e positivos, definidos num arquivo de cabeçalho *.h*. Esses números são conhecidos como sendo os ID's dos recursos. A Win32 API possui algumas funções que não aceitam variáveis-identificadores (*handles*) como parâmetro, e sim ID's de controles (botões, caixas de texto, etc) e recursos (ícones, menus, etc), como a função `LoadIcon()`.

ID não é um tipo de variável, e sim apenas uma definição, como podemos verificar num arquivo *.h* de exemplo:

```
// cabeçalho dos recursos (arquivo .h)
#define IDI_MEUICONE          101
#define IDC_MEUCURSOR         102
```

Os números referentes ao ID não precisam estar necessariamente na ordem, mas é importante que eles sejam diferentes e que sejam maiores que 100 (números menores que 100 são utilizados pelos ID's dos recursos definidos pelo sistema).

Sempre usaremos um arquivo de cabeçalho (como no exemplo) contendo os ID's dos recursos, quando esses forem utilizados. No arquivo de recursos *.rc*, temos que incluir a diretiva:

```
#include "arquivo_de_cabeçalho.h"
```

Onde `arquivo_de_cabeçalho.h` é o nome do arquivo de cabeçalho.

Ícones personalizados

O primeiro recurso que aprenderemos a incluir no programa é o ícone. Com o editor de textos ASCII aberto, digitamos a seguinte linha no arquivo *.rc*:

```
IDI_MEUICONE   ICON    "meuicone.ico"
```

Observe a palavra `ICON` em negrito. Ela é a palavra-chave que indica que `IDI_MEUICONE` é um ID para o arquivo de ícone de nome *meuicone.ico*. O ID `IDI_MEUICONE` pode ter qualquer outro nome, desde que seja definido no arquivo de cabeçalho dos recursos. A string `"meuicone.ico"` (incluindo as aspas duplas) informa o caminho e o nome do arquivo a ser inserido como ícone de recurso.

Depois de criado os arquivos *.h* e *.rc*, devemos incluí-los no projeto do nosso programa (geralmente dentro da IDE do compilador) e modificar alguns trechos da função `WinMain()` e incluir um `#include` no código-fonte.

Nota: sugiro a leitura da documentação do seu compilador, tanto para aprender como incluir arquivos no projeto quanto para verificar como criar executáveis Win32 utilizando recursos.

Antes de seguir para a modificação do código-fonte do programa, vamos verificar o conteúdo dos arquivo *.h* e *.rc*, nas Listagens 3.1 e 3.2, respectivamente:

```
//---------------------------------------------------------------------------
// prog03-1-res.h - Cabeçalho dos recursos
//---------------------------------------------------------------------------

#define IDI_MEUICONE  101
```

Listagem 3.1: O arquivo de cabeçalho.

```
//---------------------------------------------------------------------------
// prog03-1-res.rc - Arquivo de recursos
//---------------------------------------------------------------------------

#include "prog03-1-res.h"

IDI_MEUICONE   ICON    "meuicone.ico"
```

Listagem 3.2: O arquivo de recursos.

Para usar o ícone do arquivo de recursos, as seguintes modificações no arquivo *prog03-1.cpp* devem ser feitas, em relação ao arquivo *prog02.cpp* (Listagem 3.1):

```
//---------------------------------------------------------------------------
// prog03-1.cpp - Ícones personalizados
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Bibliotecas
//---------------------------------------------------------------------------
#include <windows.h>

// --- linha adicionada ---
// Cabeçalho dos recursos
#include "prog03-1-res.h"

// código omitido (veja o código completo no CD-ROM)
int WINAPI WinMain(...)
{
  // código omitido (veja o código completo no CD-ROM)
```

```
  // --- linha modificada ---
  // Carrega ícone do recurso
  wcl.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MEUICONE));
  // --- linha modificada ---
  // Carrega ícone do recurso
  wcl.hIconSm = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MEUICONE));

  // código omitido (veja o código completo no CD-ROM)
}
```

Listagem 3.1: Usando ícones personalizados.

No trecho do código da Listagem 3.1, incluímos a diretiva `#include "prog03-1-res.h"`, para que o programa possa reconhecer o ID do ícone personalizado. Também modificamos duas linhas referentes aos membros de ícone (`hIcon` e `hIconSm`) da estrutura `WNDCLASSEX`, utilizando a função `LoadIcon()` para carregar o ícone dos recursos.

Conforme explicado no capítulo anterior, devemos passar a variável `hInstance` no primeiro parâmetro de `LoadIcon()` e o nome do ícone no segundo, quando estamos utilizando ícones personalizados. Porém, ao invés de passar apenas o ID do ícone, foi utilizada a macro `MAKEINTRESOURCE()`.

A macro `MAKEINTRESOURCE()` deve ser utilizada quando precisamos converter um ID (que é um número) para uma string, somente nos parâmetros de funções que podem receber esse tipo de conversão (funções que podem obter dados dos recursos). Caso passássemos somente o ID do ícone, ocorreria um erro de tipos diferentes (`int` e `string`) durante a compilação.

Podemos ver o programa com o ícone personalizado (que está em destaque no canto superior direito) na Figura 3.2.

Dica: podemos mudar o ícone de uma janela em tempo de execução, bastando enviar a mensagem `WM_SETICON` para o programa. O parâmetro `wParam` pode receber `ICON_BIG` ou `ICON_SMALL` e `lParam` recebe o identificador do ícone. Exemplo: `SendMessage(hWnd, WM_SETICON, (WPARAM)ICON_BIG, (LPARAM)LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MEUICONE)));`

Figura 3.2: Programa com ícone personalizado.

Novos cursores

O próximo recurso que veremos é o cursor. Sua definição no arquivo de recursos é muito semelhante ao ícone:

```
IDC_MEUCURSOR  CURSOR  "meucursor.cur"
```

A palavra-chave CURSOR indica o ID (`IDC_MEUCURSOR`, no exemplo) para o arquivo do cursor (*meucursor.cur*), semelhante à palavra-chave `ICON` como já vimos.

Para utilizar o novo cursor no programa, basta modificar a seguinte linha da função `WinMain()`:

```
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Para a linha abaixo:

```
wcl.hCursor = LoadCursor(hInstance, MAKEINTRESOURCE(IDC_MEUCURSOR));
```

Lembrando que sempre devemos adicionar o arquivo de cabeçalho dos recursos (*.h*) e o arquivo de recursos (*.rc*) no projeto. Veja os arquivos *prog03-2-*

res.h, *prog03-2-res.rc* e *prog03-2.cpp* no CD-ROM, com o exemplo do uso de cursores diferentes.

Bitmaps e sons

Imagens (bitmaps) e sons são dois recursos multimídia que podemos incluir em nossos programas. A definição de ambos no arquivo de recursos também é muito parecida com a de ícones e cursores:

```
IDB_MEUBITMAP  BITMAP  "meubitmap.bmp"
IDW_MEUSOM     WAVE    "meusom.wav"
```

As palavras-chaves `WAVE` e `BITMAP` indicam os ID's de arquivos de som e imagem, respectivamente.

Não vamos aprender como carregar sons nem bitmaps por enquanto, pois há capítulos específicos para cada um desses assuntos. Apenas citei esses dois recursos para que você saiba que eles também podem ser inclusos no programa e como são definidos no arquivo de recursos.

Dica: mesmo que não vamos aprender sobre imagens e sons no momento, há um programa (*prog03-3.cpp*) no CD-ROM que carrega bitmaps e sons dos recursos. Veja os arquivos *prog03-3.cpp*, *prog03-3-res.h* e *prog03-3-res.rc* caso queira saber como o programa foi feito.

Informações sobre a versão do programa

Embora nem todas as pessoas tenham conhecimento sobre esse recurso, é possível adicionar algumas informações sobre direitos autorais e versão do programa. No Windows, essas informações podem ser encontradas quando visualizamos as propriedades de um arquivo *.exe*, *.dll*, *.drv*, *.fon*, *.fnt*, *.vxd* ou *.lib*, como na Figura 3.3.

Esse recurso pode ser útil para manter o controle de versões do programa, para o usuário saber o nome original do arquivo (caso seja renomeado) ou mesmo para manter informações sobre os direitos autorais. No Windows XP, por exemplo, algumas informações desse recurso são utilizadas pelo Windows Explorer (Figura 3.4), quando os arquivos são exibidos com a opção do menu *Exibir... Lado a Lado*.

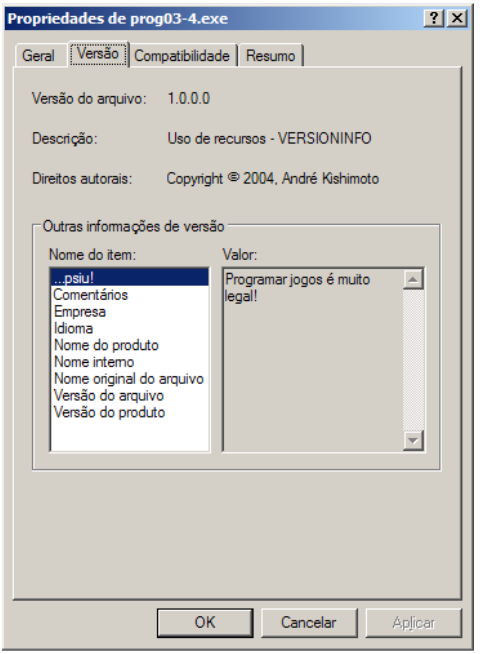

Figura 3.3: Propriedades do arquivo *prog03-4.exe*.

Figura 3.4: Informações utilizadas pelo Windows Explorer (no Windows XP).

A inclusão das informações sobre a versão do programa (ou biblioteca DLL, driver de dispositivo, etc) utiliza mais palavras-chaves que os recursos vistos até agora e não necessita de ID's.

A estrutura para definir o recurso de informação de versão é a seguinte (Listagem 3.2):

```
1 VERSIONINFO
FILEVERSION n1,n2,n3,n4
PRODUCTVERSION n1,n2,n3,n4
FILEFLAGSMASK fileflagsmask
FILEFLAGS fileflags
FILEOS fileos
FILETYPE filetype
FILESUBTYPE subtype
{
  BLOCK "StringFileInfo"
  {
    BLOCK "lang-charset"
    {
```

```
      VALUE "CompanyName", "texto"
      VALUE "FileDescription", "texto"
      VALUE "FileVersion", "texto"
      VALUE "InternalName", "texto"
      VALUE "LegalCopyright", "texto"
      VALUE "LegalTrademarks", "texto"
      VALUE "OriginalFilename", "texto"
      VALUE "PrivateBuild", "texto"
      VALUE "ProductName", "texto"
      VALUE "ProductVersion", "texto"
      VALUE "SpecialBuild", "texto"
      VALUE "Comentários", "texto"
    }
  }
  BLOCK "VarFileInfo"
  {
    VALUE "Translation", langID, charsetID
  }
}
```

Listagem 3.2: Estrutura do recurso informação de versão.

A primeira linha deve ser definida como `1 VERSIONINFO`. A segunda linha, `FILEVERSION n1,n2,n3,n4`, especifica o número da versão do arquivo, onde `n1,n2,n3,n4` recebem quatro números inteiros separados por vírgulas. A terceira linha especifica o número da versão do produto o qual o arquivo está sendo distribuído, onde `n1,n2,n3,n4` recebem quatro números inteiros separados por vírgulas.

Nota: apenas as palavras que não estão em negrito na Listagem 3.2 é que podem ser modificadas. Palavras negritadas são palavras-chave.

A linha `FILEFLAGSMASK fileflagsmask` especifica quais bits informados pela linha `FILEFLAGS fileflags` são válidos. O parâmetro `fileflagsmask` recebe como padrão 0x17L. Caso `fileflags` receba `VS_FF_PRIVATEBUILD`, devemos somar 0x8L em `fileflagsmask` (note que a soma é em hexadecimal). Quando `fileflags` recebe `VS_FF_SPECIALBUILD`, somamos 0x20L em `fileflagsmask`. Se tanto `VS_FF_PRIVATEBUILD` quanto `VS_FF_SPECIALBUILD` for especificado em `fileflags`, devemos somar 0x28L em `fileflagsmask`.

O parâmetro `fileflags` pode receber um ou mais valores da Tabela 3.1.

Valor	Descrição
`VS_FF_DEBUG`	Arquivo tem informações de depuração ou foi compilado no modo *debug*.
`VS_FF_PATCHED`	Arquivo foi modificado e não é igual ao original de mesmo número de versão.

`VS_FF_PRERELEASE`	Arquivo está em fase de desenvolvimento e não é um produto final.
`VS_FF_PRIVATEBUILD`	Arquivo não foi criado utilizando padrões do modo *release* (final). Se esse valor for utilizado, o bloco `"StringFileInfo"` precisa conter a string `"PrivateBuild"`.
`VS_FF_SPECIALBUILD`	Arquivo foi criado pela empresa original utilizando padrões do modo release, mas é uma variação do arquivo de mesma versão. Se esse valor for utilizado, o bloco `"StringFileInfo"` precisa conter a string `"SpecialBuild"`.
`0x1L`	Arquivo compilado no modo *release*.

Tabela 3.1: Valores para `fileflags`.

A linha `FILEOS fileos` especifica para qual sistema operacional o arquivo foi desenvolvido. O parâmetro `fileos` pode receber um dos valores da Tabela 3.2.

Valor	Descrição
`VOS_UNKNOWN`	Sistema operacional para o qual o arquivo foi desenvolvido é desconhecido.
`VOS_DOS`	Arquivo foi desenvolvido para MS-DOS.
`VOS_NT`	Arquivo foi desenvolvido para Windows NT/2000/XP.
`VOS__WINDOWS16`	Arquivo foi desenvolvido para Windows plataforma 16-bit.
`VOS__WINDOWS32`	Arquivo foi desenvolvido para Windows plataforma 32-bit.
`VOS_DOS_WINDOWS16`	Arquivo foi desenvolvido para Windows plataforma 16-bit rodando MS-DOS.
`VOS_DOS_WINDOWS32`	Arquivo foi desenvolvido para Windows plataforma 32-bit rodando MS-DOS.
`VOS_NT_WNDOWS32`	Arquivo foi desenvolvido para Windows NT/2000/XP.

Tabela 3.2: Valores para `fileos`.

A linha `FILETYPE filetype` especifica de que tipo é o arquivo, com `filetype` podendo receber um dos valores da Tabela 3.3.

Valor	Descrição
VFT_UNKNOWN	Arquivo desconhecido.
VFT_APP	Arquivo é um aplicatico.
VFT_DLL	Arquivo é uma biblioteca DLL.
VFT_DRV	Arquivo é um driver de dispositivo. No caso, subtype especifica mais detalhes do driver.
VFT_FONT	Arquivo é uma fonte. No caso, subtype especifica mais detalhes da fonte.
VFT_VXD	Arquivo é um dispositivo virtual.
VFT_STATIC_LIB	Arquivo é uma biblioteca estática.

Tabela 3.3: Valores para filetype.

A linha FILESUBTYPE subtype especifica mais detalhes do tipo de arquivo, no caso de drivers (filetype recebe VFT_DRV) e fontes (filetype recebe VFT_FONT). O parâmetro subtype pode receber um dos valores da Tabela 3.4 no caso de drivers, um dos valores da Tabela 3.5 no caso de fontes ou VFT2_UNKOWN em outros casos.

Valor	Descrição
VFT2_UNKNOWN	Tipo do driver é desconhecido.
VFT2_DRV_COMM	Driver é de comunicação.
VFT2_DRV_PRINTER	Driver é de impressora.
VFT2_DRV_KEYBOARD	Driver é de teclado.
VFT2_DRV_LANGUAGE	Driver é de língua.
VFT2_DRV_DISPLAY	Driver é de vídeo.
VFT2_DRV_MOUSE	Driver é de mouse.
VFT2_DRV_NETWORK	Driver é de rede.
VFT2_DRV_SYSTEM	Driver é de sistema.
VFT2_DRV_INSTALLABLE	Driver é de instalação.
VFT2_DRV_SOUND	Driver é de som.
VFT2_DRV_VERSIONED_PRINTER	Driver é de impressora (com versão).

Tabela 3.4: Valores de drivers para subtype.

Valor	Descrição
VFT2_UNKNOWN	Tipo de fonte desconhecida.
VFT2_FONT_RASTER	Fonte do tipo bitmap.
VFT2_FONT_VECTOR	Fonte do tipo vetorial.
VFT2_FONT_TRUETYPE	Fonte do tipo True Type.

Tabela 3.5: Valores de fontes para subtype.

Depois de definirmos esses parâmetros, devemos informar dois blocos de tipos de informações: de string e de variável. Esses blocos devem estar entre `{ }`, como se fossem códigos de uma função, conforme a estrutura da Listagem 3.2.

O primeiro bloco que vamos definir é de string, `BLOCK "StringFileInfo"`. Dentro desse bloco existe um outro definido como `BLOCK "lang-charset"`, onde `lang-charset` é a combinação da língua (a Tabela 3.6 lista alguns valores) e o tipo de caractere do programa em hexadecimal (a Tabela 3.7 lista alguns valores).

Valor	Língua
0407	Alemão
0408	Grego
0409	Inglês americano
0909	Inglês britânico
040A	Espanhol
040B	Finlandês
040C	Francês
0410	Italiano
0411	Japonês
0416	Português (Brasil)
0816	Português (Portugal)

Tabela 3.6: Alguns valores para `lang-charset`.

Valor (hexa)	Valor	Tipo de caractere
0000	0	ASCII 7-bit
03A4	932	Japão
03B5	949	Coréia
04B0	1200	Unicode
04E2	1250	Europa oriental
04E4	1252	Ocidente
04E7	1255	Hebreu

Tabela 3.7: Alguns valores para `lang-charset`.

Poderíamos definir a linha `BLOCK "lang-charset"` como `BLOCK "041604E4"`, ou seja, o programa está em português do Brasil e usa caracteres do ocidente.

Dentro do bloco `"lang-charset"` precisamos especificar informações sobre o arquivo, o produto e direitos autorais. Cada informação tem um nome e propósito específico e deve ser indicado no parâmetro `"texto"` das linhas `VALUE "Valor", "Descrição"`. Veja cada um deles e para que servem na Tabela 3.8.

Valor	Descrição
`CompanyName`	Nome da empresa que produziu o arquivo.
`FileDescription`	Descrição do arquivo.
`FileVersion`	Versão do arquivo.
`InternalName`	Nome interno do arquivo. Geralmente é o nome original do arquivo sem extensão.
`LegalCopyright`	Informações de direitos autorais que se aplicam ao arquivo. Essa informação é opcional.
`LegalTrademarks`	Informações de marcas registradas que se aplicam ao arquivo. Essa informação é opcional.
`OriginalFilename`	Nome original do arquivo, sem o caminho. Essa informação é útil para o programa determinar se o arquivo foi renomeado pelo usuário.
`PrivateBuild`	Informações sobre uma versão particular do arquivo. Essa informação só deve ser inclusa se `VS_FF_PRIVATEBUILD` for especificado na linha `FILEFLAGS fileflags`.
`ProductName`	Nome do produto o qual o arquivo está sendo distribuído junto.
`ProductVersion`	Versão do produto o qual o arquivo está sendo distribuído junto.
`SpecialBuild`	Especifica a diferença entre a versão desse arquivo com sua versão padrão. Essa informação só deve ser inclusa se `VS_FF_SPECIALBUILD` for especificado na linha `FILEFLAGS fileflags`.
`Comentários / Qualquer outro texto`	Informação adicional.

Tabela 3.8: Informações do bloco `"lang-charset"`.

Definido essas informações, fechamos os blocos "lang-charset" e "StringFileInfo" com duas } }. Precisamos agora definir o segundo bloco, BLOCK "VarFileInfo". Nesse bloco, informamos apenas um valor, através da linha VALUE "Translation", langID, charsetID. O parâmetro langID recebe 0x mais um dos valores da Tabela 3.6 e charsetID recebe um dos valores da Tabela 3.7 (valores decimais, segunda coluna). Para informar que o idioma do programa é português do Brasil e usa caracteres ocidentais, a linha ficaria com os seguintes valores: VALUE "Translation", 0x0416, 1252.

Não precisamos modificar nem adicionar nenhuma linha de código no nosso programa (arquivo *.cpp*) para que o recurso de informações de versão seja incluso. Basta que a estrutura da Listagem 4.2 esteja dentro do arquivo *.rc* e automaticamente o compilador incluirá as informações no executável. Porém, devemos incluir a diretiva #include <windows.h> no arquivo *.rc*, pois utilizamos definições dessa biblioteca (como VOS_UKNOWN).

A Listagem 3.3 mostra o conteúdo do arquivo *prog03-4-res.rc* com a definição do recurso de informações sobre a versão do programa. Parte das informações podem ser vistas na Figura 3.3.

```
//-----------------------------------------------------------------------
// prog03-4-res.rc - Arquivo de recursos
//-----------------------------------------------------------------------

#include <windows.h>
#include "prog03-4-res.h"

IDI_MEUICONE            ICON                    "meuicone.ico"
IDC_MEUCURSOR           CURSOR                  "meucursor.cur"

1 VERSIONINFO
FILEVERSION 1,0,0,0
PRODUCTVERSION 1,0,0,0
FILEFLAGSMASK 0x17L
FILEFLAGS 0x0L
FILEOS VOS__WINDOWS32
FILETYPE VFT_APP
FILESUBTYPE VFT2_UNKNOWN
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "041604e4"
    BEGIN
      VALUE "CompanyName", "André Kishimoto"
      VALUE "FileDescription", "Uso de recursos - VERSIONINFO"
      VALUE "FileVersion", "1.0"
      VALUE "InternalName", "prog03-4"
      VALUE "LegalCopyright", "Copyright © 2004, André Kishimoto"
      VALUE "OriginalFilename", "prog03-4.exe"
```

```
      VALUE "ProductName", "Uso de recursos - VERSIONINFO"
      VALUE "ProductVersion", "1.0"
      VALUE "Comentários", "Pode escrever qualquer coisa"
      VALUE "...psiu!", "Programar jogos é muito legal!"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x416, 1252
  END
END
```

Listagem 3.3: Definindo informações sobre o programa.

Dica: no arquivo de recursos, também podemos utilizar as palavras-chave `BEGIN...END` ao invés de `{...}`, como na Listagem 3.3, dando um "visual" de linguagem Pascal para a definição dos recursos.

Definindo menus e teclas de atalho

Vamos aprender agora a criar um recurso básico para 99% dos programas Windows: menus e teclas de atalho. Um menu é composto por uma lista de opções, como na Figura 3.5, que, quando selecionadas, executam certa ação. Um menu também pode conter sub-menus, com mais opções e também mais sub-menus, criando uma hierarquia de opções.

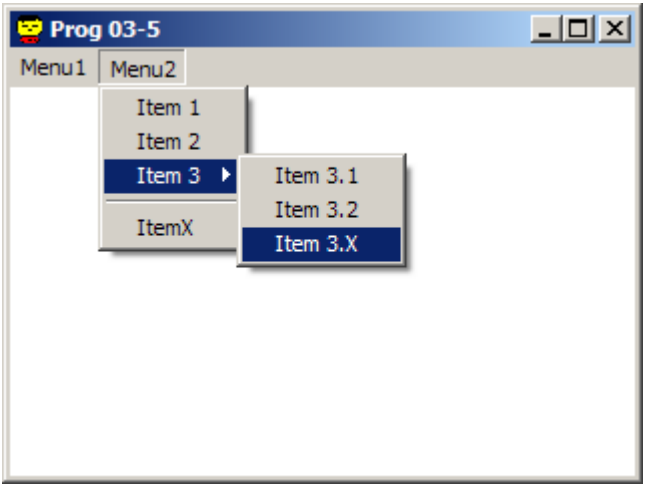

Figura 3.5: Hierarquia de opções de um menu.

Para criarmos um menu, utilizamos o seguinte esqueleto no arquivo de recursos (Listagem 3.4):

```
IDM_MEUMENU    MENU
{
  POPUP "menu1", tipos (opcional)
  {
    MENUITEM "item1", ID_DO_MENU1_ITEM1, tipos (opcional)
    MENUITEM "item2", ID_DO_MENU1_ITEM2, tipos (opcional)
    MENUITEM "itemX", ID_DO_MENU1_ITEMX, tipos (opcional)
  }
  POPUP "menu2", tipos (opcional)
  {
    MENUITEM "item1", ID_DO_MENU2_ITEM1, tipos (opcional)
    MENUITEM "item2", ID_DO_MENU2_ITEM2, tipos (opcional)
    MENUITEM "itemX", ID_DO_MENU2_ITEMX, tipos (opcional)
  }
}
```

Listagem 3.4: Esqueleto de um menu.

A palavra-chave `MENU` indica o ID do menu, que no esqueleto é `IDM_MEUMENU`. Após essa definição do ID, devemos abrir um bloco de código com `{` (ou `BEGIN`).

A palavra-chave `POPUP` cria um "menu-pai", que servirá para listar os itens (opções) do menu. O parâmetro `tipos` é opcional e pode receber um dos valores da Tabela 3.9 para modificar a aparência do menu. Para definir um menu com mais de um valor da Tabela 3.9, devemos separar os valores por vírgulas.

Não é preciso definir nenhum ID para o menu-pai. Depois de criar um menu-pai, abrimos um bloco de código para adicionar os itens do menu.

Valor	Descrição
`CHECKED`	Menu marcado/selecionado.
`GRAYED`	Menu não pode ser clicado (desabilitado). Não pode ser usado junto com `INACTIVE`.
`HELP`	Identifica um item de ajuda
`INACTIVE`	Menu inativo (não faz nada, mas pode ser clicado). Não pode ser usado junto com `GRAYED`.
`MENUBARBREAK`	Idem a `MENUBREAK`, exceto que adiciona uma linha separando as colunas (para itens do menu-pai).
`MENUBREAK`	Posiciona o menu-pai numa nova linha. Para itens do menu-pai, posiciona-os numa nova coluna, sem linhas separatórias.

Tabela 3.9: Aparência do item do menu.

Para adicionarmos um item no menu, colocamos a linha `MENUITEM "itemX", ID_DO_MENU_ITEMX, tipo` dentro do bloco de código do `POPUP "menu-pai"`, onde `MENUITEM` indica que `ID_DO_MENU_ITEMX` é o ID do item `"itemX"` do menu. O parâmetro `tipos` é opcional e pode receber um ou mais valores (separados por vírgulas) da Tabela 3.9 para modificar a aparência do item do menu.

Existe um item de menu especial, que é o separador de itens (veja a Figura 3.5 – o item separador é a linha traçada entre os itens "*Item 3*" e "*Item X*". Ele é definido no arquivo de recursos como `MENUITEM SEPARATOR`.

Nos textos dos menus, são muito utilizadas as teclas "*hot keys*", que são teclas de atalhos combinadas com o uso do ALT. Para definir uma letra do texto como *hot key,* usamos o símbolo & antes da letra. Por exemplo, ao definir `MENUITEM "Sai&r", ID_MENU_SAIR`, estamos informando que a *hot key* do item do menu *"Sair"* é a combinação ALT+R.

Dica: o texto informado em `"itemX"` pode ter o caractere `\t` para tabular o texto do item.

Além das *hot keys*, podemos criar e utilizar teclas de atalho (aceleradoras), como a famosa tecla F1 para abrir o arquivo de ajuda. Para utilizar teclas de atalho, devemos definí-las no arquivo de recursos, como na Listagem 3.5.

```
IDA_MEUATALHO          ACCELERATORS
{
  tecla, ID_DO_ATALHO, tipo (opcional), opções (opcional)
  tecla, ID_DO_ATALHO, tipo (opcional), opções (opcional)
}
```

Listagem 3.5: Definindo teclas de atalho como recursos.

A palavra-chave `ACCELERATORS` indica que `IDA_MEUATALHO` é o ID das teclas de atalho. Abrimos um bloco de código em seguida, para acrescentar as teclas de atalho através da linha `tecla, ID_DO_ATALHO, tipo, opções`.

O parâmetro `tecla` pode receber: um caractere entre aspas duplas (caractere com prefixo ^ indica que a tecla CONTROL também deve ser usada no atalho), ou um valor inteiro representando o caractere (o parâmetro `tipo` deve receber o valor `ASCII`), ou uma tecla virtual (Tabela 4.12; o parâmetro tipo deve receber o valor `VIRTKEY`).

O parâmetro `ID_DO_ATALHO` indica o ID da tecla de atalho. Quando usamos teclas de atalho para executar algum item do menu, podemos enviar o mesmo ID do item para esse parâmetro.

O parâmetro `tipo` só precisa ser definido quando o parâmetro `tecla` recebe um valor inteiro (`tipo` recebe `ASCII`) ou uma tecla virtual (`tipo` recebe `VIRTKEY`).

O parâmetro `opções` pode receber de um a três valores (separados por vírgulas), sendo esses: `ALT`, `SHIFT` e `CONTROL`. O envio desses valores para o parâmetro indica que a tecla de atalho é ativada apenas se as teclas desses valores enviados estiverem pressionadas.

Podemos verificar um exemplo de criação de teclas de atalho na Listagem 3.6.

```
IDA_MEUATALHO  ACCELERATORS
{
  "1", ID_DO_MENU1_ITEM1, ALT        ; ALT+1
  VK_F2, ID_DO_MENU1_ITEM2, VIRTKEY  ; F2
  "^X", ID_DO_MENU1_ITEMX            ; CONTROL+X
  VK_ESCAPE, IDA_ATALHO_ESC, VIRTKEY ; ESC
}
```

Listagem 3.6: Criando teclas de atalho.

Dica: teclas de atalho não precisam necessariamente estar associadas à um item do menu.

Usando menus e teclas de atalho

Temos três opções para usarmos os menus no programa: enviar o ID do menu para o membro `lpszMenuName` da estrutura `WNDCLASSEX`, passar um identificador de menu para o parâmetro `HMENU hMenu` da função `CreateWindowEx()` ou utilizar a função `SetMenu()`.

No primeiro caso, modificamos apenas uma linha dentro da função `WinMain()` no código-fonte do nosso programa, onde especificamos os atributos da classe da janela (`WNDCLASSEX`):

```
wcl.lpszMenuName = MAKEINTRESOURCE(IDM_MEUMENU);
```

Com a modificação acima, o menu já será inserido no programa. A segunda opção é enviar um identificador de menu para o décimo parâmetro da função `CreateWindowEx()`:

```
// Cria identificador do menu do recurso
HMENU hMenu = LoadMenu(hInstance, MAKEINTRESOURCE(IDM_MEUMENU));

hWnd = CreateWindowEx(..., ..., ..., ..., ..., ..., ..., ..., ...,
       hMenu, ..., ...);
```

Para criar e enviar o identificador de menu para a função `CreateWindowEx()`, obtemos o mesmo pelo retorno da função `LoadMenu()`.

```
HMENU LoadMenu(
  HINSTANCE hInstance, // identificador do módulo
  LPCTSTR lpMenuName // ID do recurso do menu
);
```

A função recebe o identificador do módulo que contém o recurso do menu (no nosso caso, é a instância atual do programa – o primeiro parâmetro da função `WinMain()` – porém, também podemos carregar recursos de bibliotecas DLL). O segundo parâmetro recebe o ID do recurso do menu, o qual devemos passar utilizando a macro `MAKEINTRESOURCE()`. O retorno de `LoadMenu()` é o identificador do menu do recurso ou `NULL` quando ocorre algum erro.

Quando utilizamos `LoadMenu()` para carregar um menu do recurso, devemos destruir o menu e liberar sua memória antes do programa ser fechado. Para isso, usamos `DestroyMenu()`.

```
BOOL DestroyMenu(
  HMENU hMenu // identificador do menu
);
```

A função recebe apenas o identificador de menu que queremos destruir. Ela também destrói todos os sub-menus contidos no menu. O retorno é um valor diferente de zero quando não há erros, ou zero, caso contrário.

Dica: podemos adicionar a chamanda à função `DestroyMenu()` antes da linha `return(msg.wParam)` da função `WinMain()`.

O último método para carregar um menu no programa é com o uso da função `SetMenu()`.

```
BOOL SetMenu(
  HWND hWnd, // identificador da janela
  HMENU hMenu // identificador do menu
);
```

A função recebe como primeiro parâmetro o identificador da janela (`HWND hWnd`) a qual o menu será adicionado. O segundo parâmetro é o identificador do menu, que obtemos com a função `LoadMenu()`. A função retorna um valor diferente de zero quando não há erros, ou zero, caso contrário.

Para adicionar um menu à janela principal pelo último método, devemos criar a janela antes de chamar `SetMenu()`, como no exemplo da Listagem 3.7.

```
// hWnd = CreateWindowEx() já foi executada

if(hWnd)
{
  // Cria identificador do menu do recurso
  HMENU hMenu = LoadMenu(hInstance, MAKEINTRESOURCE(IDM_MEUMENU));

  // Definimos o menu para a janela
  SetMenu(hWnd, hMenu);

  // etc…
}
```

Listagem 3.7: Definindo o menu com `SetMenu()`.

Depois de termos adicionado o menu, vamos habilitar o uso das teclas de atalho no programa, se existirem. Carregamos as teclas de atalho com a função `LoadAccelerators()`.

```
HACCEL LoadAccelerators(
  HINSTANCE hInstance, // identificador do módulo
  LPCTSTR lpTableName // recurso com teclas de atalho
);
```

O primeiro parâmetro da função recebe o identificador do módulo que contém o recurso das teclas de atalho (geralmente passamos a variável `hInstance` do parâmetro da `WinMain()`) e o segundo parâmetro recebe o ID do recurso, com o uso da macro `MAKEINTRESOURCE()`. A função retorna o identificador do recurso das teclas de atalho ou `NULL` em caso de erro.

Essa função apenas carrega as teclas de atalho e cria seu identificador. Para utilizá-las, devemos traduzi-las em comandos para que o programa possa processá-las. Usamos a função TranslateAccelerator(); porém, é necessário modificar o loop de mensagens da `WinMain()`.

```
int TranslateAccelerator(
  HWND hWnd, // identificador da janela
  HACCEL hAccTable, // identificador das teclas de atalho
  LPMSG lpMsg // ponteiro para estrutura de mensagem
);
```

A função recebe o identificador da janela (`hWnd`) no primeiro parâmetro; o segundo parâmetro recebe o identificador das teclas de atalho (que acabamos de obter com a função `LoadAccelerators()`) e o terceiro recebe um ponteiro para a estrutura `MSG`, que contém informações das mensagens da fila de mensagens. A função retorna um valor diferente de zero quando bem sucedida ou zero no caso de erro (ou seja, a mensagem não foi traduzida).

Nota: quando a função retorna um valor diferente de zero (mensagem processada), nosso programa não deve chamar a função `TranslateMessage()` para processar novamente a mensagem. Por isso devemos fazer uma modificação no loop de mensagens.

Quando usamos teclas de atalho, o loop de mensagens é codificado da seguinte maneira (Listagem 3.8):

```
// Armazena dados da mensagem que será obtida
MSG msg;

// Loop de mensagens, enquanto mensagem não for WM_QUIT,
// obtém mensagem da fila de mensagens
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
  // Verifica se tecla de atalho foi ativada, se for, não executa
  // TranslateMessage() nem DispatchMessage(), pois já foram processadas
  if(!TranslateAccelerator(hWnd, hAccel, &msg))
  {
    // Traduz teclas virtuais ou aceleradoras (de atalho)
    TranslateMessage(&msg);

    // Envia mensagem para a função que processa mensagens (WindowProc)
    DispatchMessage(&msg);
  }
}
```

Listagem 3.8: Loop de mensagens com teclas de atalho.

Já adicionamos o menu e carregamos as teclas de atalho no nosso programa, mas como sabemos se o usuário escolheu alguma opção do menu ou ativou alguma tecla de atalho? Quando ocorrem esses eventos, o Windows envia uma mensagem, `WM_COMMAND`, para a fila de mensagens do nosso programa.

No processo da mensagem `WM_COMMAND`, verificamos o valor do bit menos significativo do parâmetro `wParam` para sabermos qual item do menu/tecla de atalho foi selecionado/ativada. O bit mais significativo de `wParam` é 1 se a mensagem foi enviada por uma tecla de atalho ou zero se foi enviada por um item do menu.

Dica: o uso de controles (como botões, por exemplo) também gera a mensagem `WM_COMMAND`. No caso, podemos verificar qual controle enviou a mensagem comparando `wParam` com seu ID ou `lParam` com o identificador do controle. Veremos como criar e utilizar controles mais para frente.

A Listagem 3.9 mostra o processamento da mensagem `WM_COMMAND`, utilizando algumas ID's de menu e teclas de atalho, definidas como exemplos no decorrer desse capítulo. Para um exemplo integral da definição, criação e uso de menus e teclas de atalho, veja os arquivos *prog03-5.cpp*, *prog03-5-res.h* e *prog03-5-res.rc* no CD-ROM.

```
case WM_COMMAND: // Item do menu, tecla de atalho ou controle ativado
{
  // Verifica bit menos significativo de wParam (ID's)
  switch(LOWORD(wParam))
  {
    case ID_DO_MENU1_ITEMX:
    {
      // Executa alguma ação
    } break;
    case ID_DO_MENU2_ITEMX:
    {
      // Executa alguma ação
    } break;
    case ID_DO_MENU2_ITEM3_X:
    {
      // Executa alguma ação
    } break;
    case IDA_ATALHO_ESC:
    {
      // Destrói a janela
      DestroyWindow(hWnd);
    } break;
  }
  return(0);
} break;
```

Listagem 3.9: Processando a mensagem `WM_COMMAND`.

Modificando itens do menu

Durante a execução dos nossos programas, podemos modificar os itens do menu, deixando-os habilitado/desabilitado/indisponível e marcado/desmarcado. Existem três funções que podem modificar o estado dos itens de um menu: `EnableMenuItem()`, `CheckMenuItem()` e `CheckMenuRadioItem()`.

```
BOOL EnableMenuItem(
  HMENU hMenu, // identificador do menu
  UINT uIDEnableItem // item de menu a ser modificado
  UINT uEnable // opções
);
```

A função `EnableMenuItem()` faz com que um item de menu seja habilitado, desabilitado ou fique indisponível. O primeiro parâmetro dessa função recebe o identificador do menu. O segundo parâmetro depende do valor de `UINT uEnable`: quando `uEnable` receber `MF_BYCOMMAND`, devemos passar o ID do item que será modificado no segundo parâmetro; quando receber `MF_BYPOSITION`, o valor de `uIDEnableItem` indica a posição do item no menu (o primeiro item tem posição zero). O último parâmetro, além de controlar o modo de referência ao item de menu (do segundo parâmetro), indica qual será o estado desse item: `MF_ENABLED` (habilitado), `MF_DISABLED` (desabilitado) ou `MF_GRAYED` (indisponível). O retorno da função indica o estado anterior do item do menu (`MF_ENABLED`, `MF_DISABLED` ou `MF_GRAYED`) ou –1 caso o item não exista.

```
DWORD CheckMenuItem(
  HMENU hmenu, // identificador do menu
  UINT uIDCheckItem, // item do menu a ser modificado
  UINT uCheck // opções
);
```

Essa função faz com que um item do menu fique marcado ou não. Os parâmetros recebem os mesmos valores da função `EnableMenuItem()`, com uma única diferença no último: ao invés de receber `MF_ENABLED`, `MF_DISABLED` ou `MF_GRAYED`, a função deve receber `MF_CHECKED` (item será marcado) ou `MF_UNCHECKED` (item será desmarcado). A função retorna o estado anterior do item (`MF_CHECKED` ou `MF_UNCHECKED`) ou –1 se o item do menu não existir.

```
BOOL CheckMenuRadioItem(
  HMENU hmenu, // identificador do menu
  UINT idFirst, // ID ou posição do primeiro item do menu
  UINT idLast, // ID ou posição do último item do menu
  UINT idCheck, // ID ou posição do item do menu a ser modificado
  UINT uFlags // opções
```

```
);
```

A função `CheckMenuRadioItem()` checa um item de menu e o transforma num item de rádio. Um grupo de itens de rádio pode ter apenas um item selecionado por vez, ao contrário de itens do tipo *checkbox* (como na função `CheckMenuItem()`). O item selecionado com essa função é marcado com um círculo, e não com a figura de um visto (como no *checkbox*). Todos os outros itens no intervalo `idFirst - idLast` são desmarcados.

O primeiro parâmetro passado para a função é o identificador do menu. Os parâmetros `idFirst`, `idLast` e `idCheck` podem receber tanto o ID do primeiro item, último item e item a ser modificado (respectivamente) quanto a posição de cada um deles, dependendo do valor informado em `uFlags`. Se `uFlags` receber `MF_BYCOMMAND`, os três parâmetros anteriores devem receber o ID dos itens do menu. Se `uFlags` receber `MF_BYPOSITION`, os três parâmetros anteriores devem receber a posição inicial, final e do item a ser modificado, respectivamente. A função retorna um valor diferente de zero quando não há erros ou zero, caso contrário.

Dica: no CD-ROM existe um programa com exemplos do uso dessas três funções, assim como o código para criar menus *pop-up* (menus "flutuantes" que aparecem quando o usuário clica com o botão direito em algum lugar do programa). Estude o código-fonte *prog03-6.cpp*, pois você aprenderá mais funções de manipulação de menu.

Caixas de diálogo

O último recurso que veremos é a caixa de diálogo – janela muito parecida com as dos programas que estivemos criando até agora, mas que é definida no arquivo de recursos através de palavras-chave, facilitando a criação e layout da mesma.

Nota: existem diversas opções de recursos e estilos para a caixa de diálogo, mas veremos apenas os mais utilizados nos programas.

Para criarmos uma caixa de diálogo, utilizamos a estrutura conforme a Listagem 3.10.

```
IDD_MEUDIALOGO        DIALOGEX       x, y, largura, altura
CAPTION "título" (opcional)
```

```
EXSTYLE estilos_extendidos (opcional)
FONT tamanho, "fonte", espessura, itálico, charset (opcional)
MENU id_do_menu (opcional)
STYLE estilos (opcional)
{
  controles
}
```

Listagem 3.10: Estrutura de uma caixa de diálogo.

Na primeira linha dessa estrutura, utilizamos a palavra-chave `DIALOGEX` para indicar que `IDD_MEUDIALOGO` é o ID da caixa de dialogo. Na mesma linha, devemos informar, após `DIALOGEX`, a coordenada (`x`, `y`) inicial da caixa de diálogo e seu tamanho, passando os valores para os parâmetros `largura` e `altura`.

As próximas cinco linhas são opcionais, mas é bom utilizá-las para que nossas caixas de diálogo fiquem mais personalizadas. A palavra-chave `CAPTION` define o título (informado no parâmetro `"título"`) da caixa de diálogo. A palavra-chave `EXSTYLE` indica que a caixa de diálogo terá estilos extras, assim como o primeiro parâmetro da função `CreateWindowEx()` vista no capítulo anterior. O parâmetro `estilos_extendidos` pode receber os valores da Tabela 2.6, separados por vírgulas.

Na próxima linha, `FONT` define qual fonte será utilizado pelos controles (textos estáticos, botões, listas, etc) da caixa de diálogo. O parâmetro `tamanho` recebe o tamanho da fonte; `fonte` recebe uma string contendo o nome da fonte (por exemplo, `"MS Sans Serif"`); `espessura` pode receber um dos valores da Tabela 4.5 (capítulo 4, onde aprenderemos a utilizar fontes no programa); o parâmetro `itálico` recebe 1 (fonte itálica) ou 0 (fonte não-itálica) e `charset` recebe um dos valores da Tabela 4.6.

Se quisermos colocar um menu na caixa de diálogo, usamos a palavra-chave `MENU` e, em seguida, indicamos o ID do menu que vamos utilizar. No caso de caixas de diálogo, o menu é carregado e associado automaticamente, sem a necessidade de codificação.

A palavra-chave `STYLE` indica os estilos da caixa de diálogo. Esse parâmetro pode receber uma combinação dos valores da Tabela 2.5 e da Tabela 3.10 (a qual lista apenas alguns valores de estilos de diálogo).

Valor	Descrição
`DS_ABSALIGN`	Indica que a coordenada da caixa de diálogo está em coordenadas da tela. Se não utilizado,

	a coordenada da caixa de diálogo é em coordenadas da janela.
`DS_CENTER`	Centraliza a caixa de diálogo na área de trabalho.
`DS_CENTERMOUSE`	Centraliza a caixa de diálogo no cursor do mouse.
`DS_MODALFRAME`	Cria uma caixa de diálogo modal que pode ser combinada com uma barra de título e menu de sistema, especificando os estilos `WS_CAPTION` e `WS_SYSMENU`.

Tabela 3.10: Alguns estilos das caixas de diálogo.

Em seguida, devemos abrir um bloco de código, para que sejam adicionados controles à caixa de diálogo. Podemos utilizar quatro tipos de controles: textos estáticos, caixas de edição, botões e controles genéricos.

Para criarmos textos estáticos, como os *labels* dos ambientes de programação visual, utilizamos a seguinte sintaxe:

```
alinhamento "texto", id_do_texto_estático, x, y, largura, altura, estilos,
estilos_extendidos
```

O primeiro parâmetro, alinhamento, pode receber `LTEXT` (alinhamento à esquerda), `CTEXT` (centralizado) ou `RTEXT` (alinhamento à direita). O segundo parâmetro recebe uma string contendo o texto estático. O terceiro parâmetro, `id_do_texto_estático`, recebe o ID do texto, definido no arquivo *.h* dos recursos. Os outros parâmetros informam a coordenada (`x`, `y`) do texto (dentro da caixa do diálogo) e o tamanho ocupado por ele (`largura` e `altura`).

O parâmetro `estilos` (opcional) indica o estilo do controle, podendo receber `SS_CENTER` (mostra texto centralizado num retângulo invisível, padrão), `WS_TABSTOP` (move cursor do teclado para outro controle com a tecla TAB, padrão) e `WS_GROUP`. O parâmetro `estilos_extendidos` pode receber um ou mais valores da Tabela 2.6 e seu uso é opcional.

Nota: o uso do estilo `WS_GROUP` indica que o controle com esse estilo é o primeiro de um grupo de controles, onde o usuário pode movimentar o cursor do teclado entre eles através das setas direcionais. O próximo controle com esse estilo começa um outro grupo, finalizando o grupo anterior.

Para criarmos caixas de edição (*textbox*), onde o usuário pode digitar qualquer texto, utilizamos a sintaxe:

```
EDITTEXT id_da_caixa, x, y, largura, altura, estilos, estilos_extendidos
```

Onde a palavra-chave `EDITTEXT` indica que `id_da_caixa` é o ID da nova caixa de edição, posicionada em (`x`, `y`) e de tamanho (`largura`, `altura`).

O parâmetro `estilos` pode receber um ou mais valores da Tabela 3.11 e também: `WS_TABSTOP`, `WS_GROUP`, `WS_VSCROLL` (janela com barra de rolagem vertical), `WS_HSCROLL` (janela com barra de rolagem horizontal) e `WS_DISABLED` (janela inativa). O parâmetro `estilos_extendidos` pode receber um ou mais valores da Tabela 2.6. Ambos os parâmetros `estilos` e `estilos_extendidos` são opcionais.

Valor	Descrição
`ES_AUTOHSCROLL`	Rolagem automática do texto na horizontal.
`ES_AUTOVSCROLL`	Rolagem automática do texto na vertical.
`ES_CENTER`	Texto centralizado.
`ES_LEFT`	Texto alinhado à esquerda.
`ES_LOWERCASE`	Converte todos os caracteres para minúsculo.
`ES_MULTILINE`	A caixa possui múltiplas linhas, não apenas uma (que é o padrão). Note que para a tecla ENTER ser utilizada para pular de linha, o estilo `ES_WANTRETURN` deve ser especificado.
`ES_NUMBER`	A caixa aceita apenas números.
`ES_PASSWORD`	Mostra um asterisco para cada caractere inserido na caixa.
`ES_READONLY`	Impede que o usuário possa editar o conteúdo da caixa.
`ES_RIGHT`	Texto alinhado à direita.
`ES_UPPERCASE`	Converte todos os caracteres para maiúsculo.
`ES_WANTRETURN`	Indica que uma nova linha será inserida quando o usuário apertar a tecla ENTER numa caixa de múltiplas linhas.

Tabela 3.11: Alguns valores de estilo de caixas de edição.

Para a criação de botões, temos disponíveis oito palavras-chave, cada uma indicando um tipo de botão; porém, a sintaxe, independente da palavra-chave utilizada, é a seguinte:

```
palavra-chave "texto", id_do_botao, x, y, largura, altura, estilos,
estilos_extendidos
```

O parâmetro `palavra-chave` pode receber um dos valores da Tabela 3.12, indicando o tipo de botão que será criado. O segundo parâmetro recebe uma string contendo o texto que será mostrado no botão. O terceiro parâmetro, `id_do_botao`, recebe o ID do botão, definido no arquivo *.h* dos recursos. Os outros parâmetros informam a coordenada (`x`, `y`) do botão (dentro da caixa do diálogo) e o tamanho ocupado por ele (`largura` e `altura`).

Os valores do parâmetro `estilos` mudam confrome o tipo de botão que é criado (veja Tabela 2.6). O parâmetro `estilos_extendidos` pode receber um ou mais valores da Tabela 2.6 e seu uso é opcional.

Valor	Descrição
`AUTO3STATE`	Cria um c*heckbox* automático de três estados (marcado, desmarcado ou indeterminado). O parâmetro `estilos` pode receber `BS_AUTO3STATE` (padrão), `WS_TABSTOP` (padrão), `WS_DISABLED` e `WS_GROUP`.
`AUTOCHECKBOX`	Cria um *checkbox* automático de dois estados (marcado ou desmarcado). O parâmetro `estilos` pode receber `BS_AUTOCHECKBOX` (padrão), `WS_TABSTOP` (padrão) e `WS_GROUP`.
`AUTORADIOBUTTON`	Cria um botão de rádio automático. O parâmetro `estilos` pode receber `BS_AUTORADIOBUTTON` (padrão), `WS_TABSTOP` (padrão), `WS_DISABLED` e `WS_GROUP`.
`CHECKBOX`	Cria um *checkbox* de dois estados (marcado ou desmarcado). O parâmetro `estilos` pode receber `BS_CHECKBOX` (padrão), `WS_TABSTOP` (padrão) e `WS_GROUP`.
`PUSHBOX`	Cria um botão contendo apenas o texto, sem sua janela em volta. O parâmetro `estilos` pode receber `BS_PUSHBOX` (padrão), `WS_TABSTOP` (padrão), `WS_DISABLED` e `WS_GROUP`.
`PUSHBUTTON`	Cria um botão comum, com o texto centralizado. O parâmetro `estilos` pode receber `BS_PUSHBUTTON` (padrão), `WS_TABSTOP`

	(padrão), WS_DISABLED e WS_GROUP.
RADIOBUTTON	Cria um botão de rádio. O parâmetro estilos pode receber BS_RADIOBUTTON (padrão), WS_TABSTOP (padrão), WS_DISABLED e WS_GROUP.
STATE3	Cria um *checkbox* de três estados (marcado, desmarcado ou indeterminado). O parâmetro estilos pode receber BS_3STATE (padrão), WS_TABSTOP (padrão) e WS_GROUP.

Tabela 3.12: Tipos de botões.

Veja que existe um tipo automático para os *checkboxes* de dois/três estados e para os botões de rádio. A diferença entre um tipo automático ou não é que, quando automático, não precisamos escrever nenhuma linha de código para que o estado do botão seja alterado, em resposta ao clique de botão do usuário. Caso contrário, devemos utilizar a função CheckDlgButton() para modificar o estado de um *checkbox* e CheckRadioButton() para selecionar/modificar o estado de um botão de rádio. Ambas as funções trabalham semelhantemente às funções CheckMenuItem() e CheckMenuRadioItem(), respectivamente.

```
BOOL CheckDlgButton(
  HWND hDlg, // identificador da caixa de diálogo
  int nIDButton, // ID do checkbox
  UINT uCheck // estado do checkbox
);
```

A função recebe o identificador da caixa de diálogo que contém o botão no primeiro parâmetro. O segundo parâmetro recebe o ID do *checkbox* que terá seu estado modificado, e o terceiro parâmetro indica o novo estado do *checkbox*, podendo receber BST_CHECKED (marcado), BST_INDETERMINATE (indeterminado; válido apenas para *checkbox* de três estados) ou BST_UNCHECKED (desmarcado). A função retorna um valor diferente de zero quando bem sucedida ou zero, caso contrário.

```
BOOL CheckRadioButton(
  HWND hDlg, // identificador da caixa de diálogo
  int nIDFirstButton, // ID do primeiro botão de rádio do grupo
  int nIDLastButton, // ID do último botão de rádio do grupo
  int nIDCheckButton // ID do botão do rádio a ser selecionado
);
```

O primeiro parâmetro da função recebe o identificador da caixa de diálogo que contém o botão de rádio. Os outros parâmetros recebem os ID's

do primeiro botão de rádio do grupo (nIDFirstButton), do último (nIDLastButton) e do que será selecionado (nIDCheckButton). A função retorna um valor diferente de zero quando bem sucedida ou zero, caso contrário.

Nota: um grupo de botões de rádio é definido pelo primeiro botão de rádio com o estilo WS_GROUP até o próximo botão de rádio com esse estilo.

Para finalizar a parte de controles nas caixas de diálogo, vamos ver como criar controles genéricos definidos pela Win32 API, utilizando a seguinte sintaxe:

```
CONTROL "texto", id_do_controle, classe, estilos, x, y, largura, altura,
estilos_extendidos
```

Utilizamos a palavra-chave CONTROL para indicar que estamos criando um controle genérico. Nesse caso, o parâmetro estilos tem um significado diferente, dependendo de qual controle for criado.

Todos os outros parâmetros já foram explicados na criação de outros controles, com exceção do parâmetro classe. Esse parâmetro indica que tipo de controle iremos criar, podendo receber: BUTTON (botão), COMBOBOX (caixa de seleção), EDIT (caixa de texto), LISTBOX (lista de seleção), SCROLLBAR (barra de rolagem) ou STATIC (estático).

Quando classe recebe BUTTON, estamos criando um controle do tipo botão, uma janela retangular onde o usuário pode clicar para executar determinada ação/seleção. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.13 para o parâmetro estilo.

Valor	Descrição
BS_3STATE	Cria um *checkbox* de três estados.
BS_AUTO3STATE	Cria um *checkbox* automático de três estados.
BS_AUTOCHECKBOX	Cria um *checkbox* automático de dois estados.
BS_AUTORADIOBUTTON	Cria um botão de rádio automático.
BS_CHECKBOX	Cria um *checkbox* de dois estados.
BS_DEFPUSHBUTTON	Cria um botão comum com uma borda preta em volta, indicando que ele é o padrão; quando o usuário pressiona ENTER numa caixa de diálogo, mesmo que o cursor do teclado não esteja nele, o botão é executado.

BS_GROUPBOX	Cria um retângulo onde outros controles podem ser agrupados. O texto associado ao controle desse estilo aparecerá no canto superior esquerdo do retângulo.
BS_LEFTTEXT	Alinha o texto do *checkbox* ou do botão de rádio no lado esquerdo. Idem a BS_RIGHTBUTTON.
BS_PUSHBUTTON	Cria um botão comum.
BS_RADIOBUTTON	Cria um botão de rádio.
BS_BITMAP	Indica que o botão mostra um bitmap.
BS_BOTTOM	Alinha o texto na parte inferior do botão.
BS_CENTER	Texto centralizado horizontalmente.
BS_ICON	Indica que o botão mostra um ícone.
BS_FLAT	Indica que o botão é bidimensional (não utiliza o sombreamento padrão que dá o aspecto 3D nos botões).
BS_LEFT	Alinha o texto na lateral esquerda do botão.
BS_MULTILINE	Botão com múltiplas linhas, quando o texto do mesmo não cabe em uma única.
BS_PUSHLIKE	Faz com que *checkboxes* e botões de rádio atuem como botões comuns.
BS_RIGHT	Alinha o texto na lateral direita do botão.
BS_RIGHTBUTTON	Posiciona o círculo do botão de rádio ou o quadrado de um *checkbox* do lado direito. Idem a BS_LEFTTEXT.
BS_TEXT	Indica que o botão mostra um texto.
BS_TOP	Alinha o texto na parte superior do botão.
BS_VCENTER	Texto centralizado verticalmente.

Tabela 3.13: Alguns estilos de botão.

Quando classe recebe COMBOX, estamos criando um controle do tipo caixa de seleção, que consiste de um campo de seleção, similar à combinação de uma caixa de texto e uma lista de seleção. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.14 para o parâmetro estilo.

Valor	Descrição
CBS_AUTOHSCROLL	Faz a rolagem de texto automática quando o usuário digita um caracter no fim da linha. Se esse estilo não for definido, apenas textos que se encaixam dentro da caixa de seleção são permitidos.

CBS_DISABLENOSCROLL	Mostra uma barra de rolagem vertical desativada quando a caixa não contém itens suficientes para fazer a rolagem. Sem esse estilo, a barra de rolagem fica escondida enquanto seu uso não for necessário.
CBS_DROPDOWN	Idem a CBS_SIMPLE, exceto que a lista de seleção não é visível, a menos que o usuário clique na caixa de seleção.
CBS_DROPDOWNLIST	Idem a CBS_DROPDOWNLIST, exceto que a caixa mostra o texto do item selecionado na lista.
CBS_LOWERCASE	Converte todos os caracteres dos itens da lista para minúsculo.
CBS_NOINTERNALHEIGHT	Indica que o tamanho da caixa de seleção é do tamanho especificado pelo programa quando o mesmo foi criado. Geralmente, o sistema modifica o tamanho da caixa de seleção para que não mostre itens pela metade.
CBS_SIMPLE	Lista de seleção sempre visível. A seleção atual da lista é mostrada na caixa.
CBS_SORT	Ordena automaticamente todos os itens adicionados na caixa.
CBS_UPPERCASE	Converte todos os caracteres dos itens da lista para maiúsculo.

Tabela 3.14: Alguns estilos de caixa de seleção.

Quando `classe` recebe `EDIT`, estamos criando um controle do tipo caixa de texto, uma janela retangular onde o usuário pode entrar dados através do teclado. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.11 para o parâmetro `estilo`.

Quando `classe` recebe `LISTBOX`, estamos criando um controle do tipo lista de seleção, uma janela contendo uma lista de itens (strings) que o usuário poderá selecionar. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.16 para o parâmetro `estilo`.

Valor	Descrição
LBS_DISABLENOSCROLL	Mostra uma barra de rolagem vertical desativada quando a lista não contém itens suficientes para fazer a rolagem. Sem esse estilo, a barra de rolagem fica escondida

	enquanto seu uso não for necessário.
LBS_EXTENDEDSEL	Permite múltipla seleção de itens utilizando a tecla SHIFT e clique do mouse.
LBS_MULTICOLUMN	Cria uma lista (com rolagem horizontal) com uma ou mais colunas.
LBS_MULTIPLESEL	Seleciona/deseleciona item cada vez que o usuário clicar num item da lista.
LBS_NOINTEGRALHEIGHT	Indica que o tamanho da lista é do tamanho especificado pelo programa quando a mesma foi criada. Geralmente, o sistema modifica o tamanho da lista para que não mostre itens pela metade.
LBS_NOSEL	Indica que a lista contém itens que não podem ser selecionados, apenas visualizados.
LBS_SORT	Ordena os itens da lista alfabeticamente.
LBS_STANDARD	Ordena os itens da lista alfabeticamente. A lista contém bordas nos quatro lados.

Tabela 3.16: Alguns estilos de lista de seleção.

Quando `classe` recebe `SCROLLBAR`, estamos criando um controle do tipo barra de rolagem, controle que mostra setas em ambos os sentidos (esquerda/direita e cima/baixo) para fazer uma rolagem (incremento/decremento) em determinado sentido. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.17 para o parâmetro `estilo`.

Valor	Descrição
SBS_BOTTOMALIGN	Alinha a parte inferior da barra de rolagem com a parte inferior do retângulo definido pelos parâmetros `x`, `y`, `largura` e `altura`. Use esse estilo com `SBS_HORZ`.
SBS_HORZ	Cria uma barra de rolagem horizontal.
SBS_LEFTALIGN	Alinha o lado esquerdo da barra de rolagem com o lado esquerdo do retângulo definido pelos parâmetros `x`, `y`, `largura` e `altura`. Use esse estilo com `SBS_VERT`.
SBS_RIGHTALIGN	Alinha o lado direito da barra de rolagem com o lado direito do retângulo definido pelos parâmetros `x`, `y`, `largura` e `altura`. Use esse estilo com `SBS_VERT`.
SBS_TOPALIGN	Alinha a parte superior da barra de rolagem

	com a parte superior do retângulo definido pelos parâmetros `x`, `y`, `largura` e `altura`. Use esse estilo com `SBS_HORZ`.
`SBS_VERT`	Cria uma barra de rolagem vertical.

Tabela 3.17: Alguns estilos de barra de rolagem.

Quando `classe` recebe `STATIC`, estamos criando um controle do tipo estático, que pode ser um texto, caixa, retângulo ou imagem. Configuramos o seu estilo fornecendo um dos valores da Tabela 3.18 para o parâmetro `estilo`.

Valor	Descrição
`SS_BITMAP`	Indica que um bitmap será mostrado no controle estático. O texto que o controle recebe é o ID do bitmap, definido no arquivo de recursos. A largura e altura especificadas são ignoradas, pois o controle se ajusta ao tamanho do bitmap.
`SS_BLACKFRAME`	Indica que o controle será um retângulo com borda preta (sem preenchimento).
`SS_BLACKRECT`	Indica que o controle será um retângulo preto (com preenchimento).
`SS_CENTER`	Indica que o controle exibirá um texto centralizado.
`SS_CENTERIMAGE`	Indica que o bitmap será centralizado no controle.
`SS_GRAYFRAME`	Indica que o controle será um retângulo com borda cinza (sem preenchimento).
`SS_GRAYRECT`	Indica que o controle será um retângulo cinza (com preenchimento).
`SS_ICON`	Indica que um ícone será mostrado no controle estático. O texto que o controle recebe é o ID do ícone, definido no arquivo de recursos. A largura e altura especificadas são ignoradas, pois o controle se ajusta ao tamanho do ícone.
`SS_LEFT`	Indica que o controle exibirá um texto alinhado à esquerda.
`SS_RIGHT`	Indica que o controle exibirá um texto alinhado à direita.
`SS_SUNKEN`	Indica que o controle terá uma borda em

	baixo relevo.
SS_WHITEFRAME	Indica que o controle será um retângulo com borda branca (sem preenchimento).
SS_WHITERECT	Indica que o controle será um retângulo branco (com preenchimento).

Tabela 3.18: Alguns estilos de controle estático.

Criando e destruindo caixas de diálogo

É possível criar dois tipos de caixa de diálogo: *modal* ou *modeless*. O primeiro tipo faz com que a caixa de diálogo fique ativa e, ao mesmo tempo, desativa a janela-pai (e suas filhas) que a criou. A janela-pai é reativada somente após a caixa de diálogo modal ser destruída. O segundo tipo cria uma caixa de diálogo inicialmente invisível e não desativa sua janela-pai.

Para criarmos uma caixa de diálogo modal, devemos utilizar a função `DialogBox()`.

```
INT_PTR DialogBox(
  HINSTANCE hInstance, // identificador do módulo
  LPCTSTR lpTemplate, // recurso da caixa de diálogo
  HWND hWndParent, // identificador da janela-pai
  DLGPROC lpDialogFunc // função que processa mensagens da caixa de diálogo
);
```

Essa função recebe o identificador do módulo no primeiro parâmetro. O segundo parâmetro recebe o ID da caixa de diálogo (passada com o uso da macro `MAKEINTRESOURCE()`). O terceiro parâmetro recebe o identificador da janela-pai, ou seja, a janela que está criando o diálogo. O último parâmetro recebe um ponteiro para a função que processa mensagens da caixa de diálogo (semelhante ao membro `lpfnWndProc` da estrutura `WNDCLASSEX`, com exceção que é uma função que processa mensagens da caixa de diálogo).

A função pode retornar três valores: se não ocorrer nenhum erro, ela retorna o valor do parâmetro `nResult` passado na função `EndDialog()` (veremos logo adiante). No caso de erro, ela pode retornar zero, se `hWndParent` for inválido, ou –1 para qualquer outro tipo de erro.

Quando não precisarmos mais da caixa de diálogo modal, devemos destruí-la utilizando a função `EndDialog()`, chamando-a de dentro da função de processamento de mensagens da caixa de diálogo.

```
BOOL EndDialog(
  HWND hDlg, // identificador da caixa de diálogo
  INT_PTR nResult // valor de retorno
);
```

O primeiro parâmetro da função recebe o identificador da caixa de diálogo que será destruída, e o segundo recebe o valor que será retornado para o programa através da função `DialogBox()`. A função retorna um valor diferente de zero quando não ocorrer erros, ou zero caso contrário.

No caso da criação de uma caixa de diálogo modeless, devemos utilizar a função `CreateDialog()`.

```
HWND CreateDialog(
  HINSTANCE hInstance, // identificador do módulo
  LPCTSTR lpTemplate, // recurso da caixa de diálogo
  HWND hWndParent, // identificador da janela-pai
  DLGPROC lpDialogFunc // função que processa mensagens da caixa de diálogo
);
```

Veja que a função `CreateDialog()` recebe os mesmos parâmetros que a função `DialogBox()`. A diferença entre essas funções é o tipo de retorno; neste caso, `CreateDialog()` retorna um identificador da caixa de diálogo criada ou `NULL` se houver erros.

Conforme dito anteriormente, uma caixa de diálogo modeless é inicialmente invisível. Para mostrar o diálogo na tela, precisamos chamar a função `ShowWindow()`, passando como parâmetro o identificador da caixa de diálogo.

Nota: diferente da função `DialogBox()`, devemos utilizar a função `DestroyWindow()` para destruir uma caixa de diálogo modeless.

Processando mensagens das caixas de diálogo

Quando estamos utilizando caixas de diálogo, devemos processar suas mensagens, além das da janela principal do programa. Para isso, criamos uma função do tipo *callback* para cada caixa de diálogo. O protótipo dessa função é a seguinte:

```
INT_PTR CALLBACK DialogProc(
  HWND hwndDlg, // identificador da caixa de diálogo
  UINT uMsg, // identificador da mensagem
  WPARAM wParam, // primeiro parâmetro da mensagem
```

```
  LPARAM lParam // segundo parâmetro da mensagem
);
```

Essa função, por ser do tipo callback, é chamada somente pelo Windows, quando há alguma mensagem da caixa de diálogo a ser processada. Ela deve retornar `TRUE` quando uma mensagem é processada ou `FALSE` caso contrário. Quando a função retorna `FALSE`, o processamento da mensagem é feita automaticamente pela administração interna da caixa de diálogo.

Quando uma caixa de diálogo (modal ou modeless) é criada, o sistema envia uma mensagem de inicialização, `WM_INITDIALOG`, que deve retornar `TRUE`. Para fechar a caixa de diálogo, o usuário pode usar a combinação de teclas ALT+F4, utilizar o menu de sistema, ou ainda, clicar no "x" que fecha a janela. O uso de qualquer uma dessas opções gera a mensagem `WM_COMMAND`, com `LOWORD(wParam)` contendo `IDCANCEL`. Devemos então processar a mensagem `WM_INITDIALOG` e `WM_COMMAND`. A Listagem 3.11 demonstra a criação de uma caixa de diálogo modal, a função `DialogProc()` e o processamento das mensagens básicas de diálogo.

```
// Protótipo da função de processamento de mensagens da caixa de diálogo.
// Veja que podemos escolher qualquer nome para a função, desde que o tipo
// de retorno e parâmetros sejam iguais ao padrão e que o nome da função
// seja passada corretamente na função DialogBox() ou CreateDialog().

INT_PTR CALLBACK DlgProcExemp(HWND, UINT, WPARAM, LPARAM);

//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  // Processando alguma mensagem da WindowProc()
  ...

  // Armazena retorno da DialogBox()
  int teste = 0;

  // Obs: IDD_MEUDIALOGO foi declarado e criado num arquivo de recursos.
  // Cria caixa de diálogo modal e salva retorno da DialogBox() em teste.

  teste   =   DialogBox(hInstance,   MAKEINTRESOURCE(IDD_MEUDIALOGO),   hWnd,
(DLGPROC)DlgProcExemp);

  // Não foi possível criar a caixa de diálogo
  if(teste <= 0)
    MessageBox(hWnd, "Erro", "Erro", MB_OK);

  ...
  // continuação da WindowProc()
}
```

```
//------------------------------------------------------------------------
// DlgProcExemp() -> Processa as mensagens enviadas para a caixa de diálogo
//------------------------------------------------------------------------
INT_PTR CALLBACK DlgProcExemp(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  // Verifica qual foi a mensagem enviada
  switch(uMsg)
  {

    case WM_INITDIALOG: // Caixa de diálogo foi criada
    {
      // Retorna TRUE, significando que a mensagem
      // foi processada corretamente
      return(TRUE);
    } break;

    case WM_COMMAND: // Item do menu, tecla de atalho ou controle ativado
    {
      switch(LOWORD(wParam))
      {
        case IDCANCEL: // ALT+F4, menu do sistema, botão "x"
        {
          // Destrói a caixa de diálogo
          EndDialog(hDlg, 0);

          return(TRUE);
         } break;
      }
    } break;

    default: // Outra mensagem
    {
      // Retorna FALSE, deixando a caixa de diálogo processar a mensagem
      return(FALSE);
    } break;

  }
}
```

Listagem 3.11: Criação e processamento de mensagens de uma caixa de diálogo modal.

A criação e uso de uma caixa de diálogo modeless é um pouco diferente da modal: além de usarmos as funções `CreateDialog()` / `DestroyWindow()`, as mensagens que devem ser processadas pela função da caixa de diálogo são passadas pela fila de mensagens do programa. Portanto, o loop de mensagens da `WinMain()` deve ser modificado, conforme a Listagem 3.12:

```
// g_hDlg é um identificador global da caixa de diálogo (HWND g_hDlg)

// Loop de mensagens, enquanto mensagem não for WM_QUIT,
// obtém mensagem da fila de mensagens
while(GetMessage(&msg, NULL, 0, 0) > 0)
{
  // Verifica se a caixa de diálogo modeless foi criada ou
  // se a mensagem é uma mensagem para a caixa de diálogo
```

```
  if((!IsWindow(g_hDlg)) || (!IsDialogMessage(g_hDlg, &msg)))
  {
    // Traduz teclas virtuais ou aceleradoras (de atalho)
    TranslateMessage(&msg);

    // Envia mensagem para a função que processa mensagens (WindowProc)
    DispatchMessage(&msg);
  }
}
```

Listagem 3.12: Loop de mensagens modificado para caixa de diálogo modeless.

A modificação que fizemos no loop de mensagens verifica duas condições: se a caixa de diálogo modeless foi criada e se a mensagem que será processada é uma mensagem da caixa de diálogo. Quando a mensagem processada é uma mensagem de diálogo, não devemos executar as funções `TranslateMessage()` e `DispatchMessage()`, pois a função `IsDialogMessage()` faz toda a tradução e envio de mensagem automaticamente.

Inicialmente, declaramos uma variável global `HWND g_hDlg = NULL`, que armazenará o retorno da função `CreateDialog()`, ou seja, ela será o identificador da caixa de diálogo modeless.

Para verificarmos se a caixa de diálogo (ou qualquer outro tipo de janela) foi criada, chamamos a função `IsWindow()`.

```
BOOL IsWindow(
  HWND hWnd // identificador da janela
);
```

Essa função recebe o identificador da janela que queremos verificar se foi criada ou não e retorna um valor diferente de zero caso a janela foi criada, ou zero, quando o identificador passado no parâmetro `hWnd` contém zero ou `NULL` (significando que ele não identifica nenhuma janela).

A função `IsDialogMessage()` verifica se uma mensagem é ou não destinada a uma caixa de diálogo.

```
BOOL IsDialogMessage(
  HWND hDlg, // identificador da caixa de diálogo
  LPMSG lpMsg // ponteiro para estrutura de mensagem
);
```

A função recebe o identificador da caixa de diálogo como primeiro parâmetro e um ponteiro para uma variável do tipo `MSG`, que contém a

mensagem que será processada, no segundo. Se a mensagem for processada por essa função, ela retorna um valor diferente de zero; se não, retorna zero.

A Listagem 3.13 contém as partes principais de um código-fonte que cria, mostra e destrói uma caixa de diálogo modeless.

```
// Protótipo da função de processamento de mensagens da caixa de diálogo.
// Veja que podemos escolher qualquer nome para a função, desde que o tipo
// de retorno e parâmetros sejam iguais ao padrão e que o nome da função
// seja passada corretamente na função DialogBox() ou CreateDialog().

INT_PTR CALLBACK DlgProcModeless(HWND, UINT, WPARAM, LPARAM);

// Variável global que armazena o identificador da caixa de diálog modeless
HWND g_hDlg = NULL;

//---------------------------------------------------------------------------
// WinMain() -> Função principal
//---------------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
  ...

  while(GetMessage(&msg, NULL, 0, 0) > 0)
  {
    // Verifica se a caixa de diálogo modeless foi criada ou
    // se a mensagem é uma mensagem para a caixa de diálogo
    if((!IsWindow(g_hDlg)) || (!IsDialogMessage(g_hDlg, &msg)))
    {
      // Traduz teclas virtuais ou aceleradoras (de atalho)
      TranslateMessage(&msg);

      // Envia mensagem para a função que processa mensagens (WindowProc)
      DispatchMessage(&msg);
    }
  }

  ...
}

//---------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//---------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  // Processando alguma mensagem da WindowProc()
  ...

  // Obs: IDD_MEUDIALOGO2 foi declarado e criado num arquivo de recursos.
  // Cria caixa de diálogo modeless

  // Verifica se a caixa de diálogo já foi criada...
  if(!IsWindow(g_hDlg))
```

```
  {
    // Se não foi, cria e salva identificador em g_hDlg
    g_hDlg = CreateDialog(hInstance, MAKEINTRESOURCE(IDD_MEUDIALOGO2), hWnd,
(DLGPROC)DlgProcModeless);

    // Mostra a caixa de diálogo
    ShowWindow(g_hDlg, SW_SHOW);
  }
  else // Caixa de diálogo já foi criada...

    // Mostra a caixa de diálogo
    SetFocus(g_hDlg);

  ...
  // continuação da WindowProc()
}


//---------------------------------------------------------------------------
// DlgProcModeless() -> Processa as mensagens enviadas para a caixa de
// diálogo
//---------------------------------------------------------------------------
INT_PTR CALLBACK DlgProcModeless(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  // Verifica qual foi a mensagem enviada
  switch(uMsg)
  {

    case WM_INITDIALOG: // Caixa de diálogo foi criada
    {
      // Retorna TRUE, significando que a mensagem
      // foi processada corretamente
      return(TRUE);
    } break;

    case WM_COMMAND: // Item do menu, tecla de atalho ou controle ativado
    {
      switch(LOWORD(wParam))
      {
        case IDCANCEL: // ALT+F4, menu do sistema, botão "x"
        {
          // Destrói a caixa de diálogo
          DestroyWindow(hDlg);

          // Indica que g_hDlg não identifica mais a caixa de diálogo
          g_hDlg = NULL;

          return(TRUE);
         } break;
      }
    } break;

    default: // Outra mensagem
    {
      // Retorna FALSE, deixando a caixa de diálogo processar a mensagem
      return(FALSE);
    } break;
```

```
    }
}
```

Listagem 3.13: Criação e processamento de mensagens de uma caixa de diálogo modeless.

Observe o trecho de código da linha `case IDCANCEL`. O identificador da caixa de diálogo (`hDlg`) que passamos para a função `DestroyWindow()` não é o mesmo que utilizamos na chamada da função `CreateDialog()`. Esse identificador é o parâmetro passado na função de processamento de mensagens da caixa de diálogo (`DlgProcModeless()`), enquanto `g_hDlg` é uma global que usamos para saber se a caixa de diálogo foi criada ou não. Depois de destruir a caixa de diálogo, devemos definir o valor `NULL` para `g_hDlg`, indicando que essa variável não identifica mais a caixa de diálogo.

Dica: no CD-ROM, você encontra um programa (*prog03-7.cpp*) com o exemplo da criação e uso das caixas de diálogo modal e modeless.

Há outros tipos de recursos que não foram discutidos nesse capítulo, mas os que você aprendeu já são suficientes para a criação da maioria de seus programas. A seguir, veremos como processar mensagens do teclado e do mouse, além de uma introdução à GDI (parte gráfica do Windows).

Capítulo 4 – GDI, Textos e Eventos de Entrada

Para melhorar a interação do usuário com os nossos programas, devemos verificar os controles de entrada (teclado e mouse), e em seguida processar suas mensagens.

No programa-exemplo *prog04-1.cpp*, além da interação do usuário via teclado, acrescentaremos algo mais: como escrever textos na janela do programa através da GDI do Windows. O programa-exemplo irá detectar a tecla pressionada pelo usuário e colocará a informação na janela do programa, conforme a Figura 4.1:

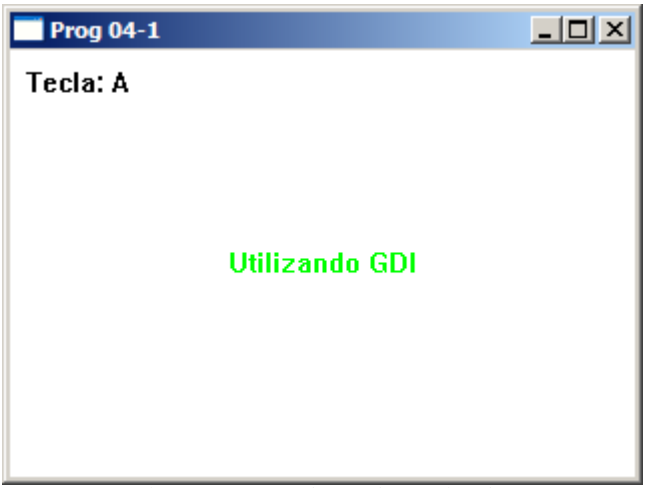

Figura 4.1: Detectando as teclas pressionadas

Antes de verificarmos as mensagens para processar os eventos do teclado e mouse, vamos ver como funciona a interface gráfica do Windows e também a mensagem `WM_PAINT`, que já foi mencionada no capítulo 2.

GDI e Device Context

No Windows, tudo relacionado à gráficos (janelas, textos, linhas, desenhos, ...) está relacionado à GDI. A GDI é uma interface que engloba um conjunto de funções e estruturas necessárias para facilitar a ligação entre o programador e os *drivers* dos dispositivos, como monitores e impressoras.

Dessa maneira, o programador não precisa se preocupar com os diversos tipos de dispositivos e fabricantes, pois esse trabalho fica por conta da GDI (a menos que seja necessário escrever uma rotina para um dispositivo específico, o que não é o nosso caso).

Para desenharmos qualquer gráfico na área cliente de um programa (ou seja, toda a área do programa, com exceção da barra de título, menus, bordas e controles), utilizamos um contexto de dispositivo (*device context*) para comunicarmos com a GDI. Um contexto de dispositivo é uma estrutura que define um conjunto de objetos gráficos e atributos os quais serão utilizados para a saída gráfica do seu programa.

Dica: assim como um pintor utiliza uma tela para fazer seus desenhos, podemos dizer que o contexto de dispositivo de vídeo é a nossa tela. Em ambientes de programação como o C++ Builder, o termo *canvas* (tela) é utilizado em referência à área onde iremos desenhar.

Existem diferentes tipos de contexto de dispositivo, tais como de impressoras, arquivos *metafile*, memória e vídeo. Iremos estudar apenas os dois últimos tipos, que se referem à arquivos bitmap e tela de vídeo, respectivamente. Vamos estudar primeiro o DC (abreviação de *device context* ou contexto de dispositivo) de vídeo, pois já estamos trabalhando com ele.

Em geral, os programas utilizam um DC "público", que é mantido pelo Windows. Esse DC é compartilhado entre todos os programas em execução e é utilizado para saída de gráficos na área cliente das janelas. Ele também é o DC padrão de todas as janelas, não necessitando nenhuma especificação no membro `WNDCLASSEX.style` da classe da janela.

Nota: o termo *gráfico* utilizado no livro refere-se à textos, linhas, elipses, retângulos, imagens bitmap, enfim, qualquer elemento que seja possível ser desenhado na área cliente da janela.

Para trabalharmos com o DC público, criamos um identificador para ele declarando uma variável do tipo `HDC`:

```
HDC hDC = NULL;
```

A partir daí, devemos obter o identificador do DC sempre que quisermos desenhar algum gráfico na área cliente da janela. Algo importante que você deve ter em mente é que os programas não sabem o que deve ser desenhado na área cliente; por isso, devemos informar a eles o que desenhar sempre que houver uma atualização na área cliente da janela (através da mensagem `WM_PAINT`).

O que os programas fazem é detectar o retângulo da área cliente que deve ser redesenhada. Isso ocorre quando a posição/tamanho da janela é modificada ou quando qualquer ação afeta a área cliente, resultando na *invalidação* de parte da área cliente (ou a área inteira, dependendo do caso). A Figura 4.2 demonstra um exemplo de invalidação da área cliente.

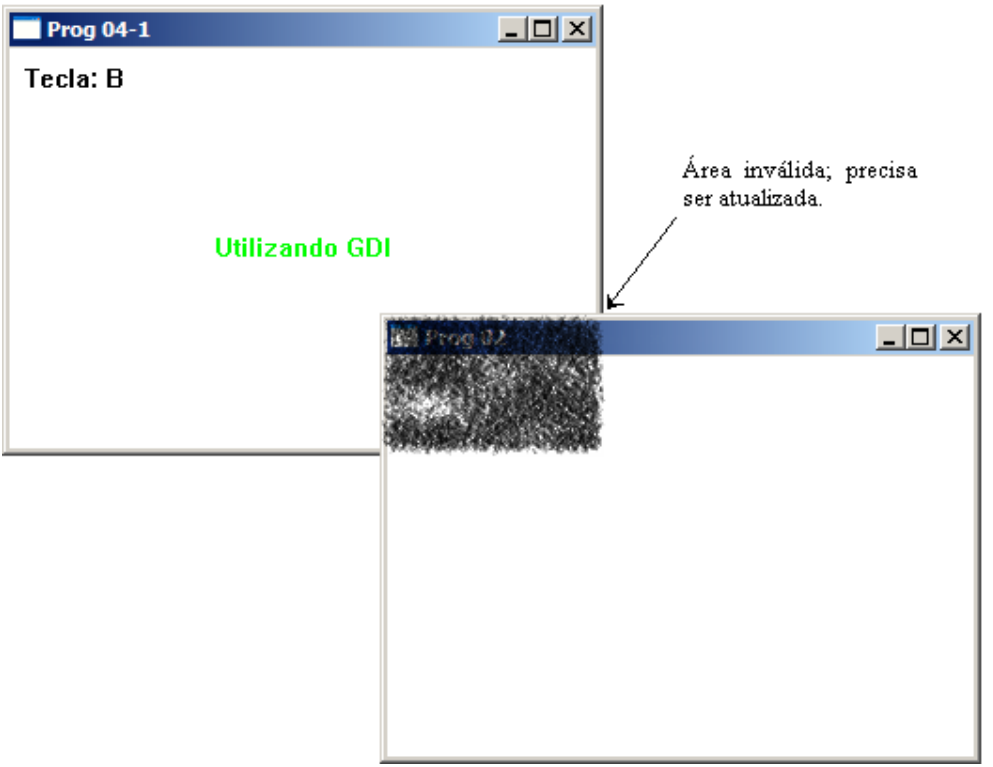

Figura 4.2: Parte da área cliente invalidada quando outra janela é sobreposta.

Quando qualquer parte da área cliente é inválida, o Windows envia ao programa uma mensagem `WM_PAINT`. Nessa mensagem, devemos *validar* a área que foi marcada como inválida, ou seja, redesenhamos essa área para que o seu conteúdo não seja perdido e informamos ao Windows que a atualização já foi

feita (caso contrário, o Windows continua enviando mensagens `WM_PAINT` até ser informado sobre a atualização).

Processando a mensagem WM_PAINT

Conforme dito anteriormente, para desenhar qualquer gráfico na área cliente, utilizamos um DC de vídeo, que no processamento da mensagem `WM_PAINT` é obtido através das funções `BeginPaint()` e `EndPaint()`. Esse par de funções tem a função de obter o identificador do DC e validar a área que foi atualizada.

> **Nota:** `BeginPaint()` e `EndPaint()` devem ser usados somente no processo da mensagem `WM_PAINT` e, para cada chamada à `BeginPaint()`, deve existir um `EndPaint()` correspondente.

```
HDC BeginPaint(
  HWND hWnd, // identificador da janela
  LPPAINTSTRUCT lpPaint // ponteiro para estrutura que contém informações de
desenho
);
```

A função `BeginPaint()` recebe dois parâmetros: o primeiro, `HWND hWnd`, indica qual janela deve ser preparada para receber as ações de desenho; o segundo, `LPPAINTSTRUCT lpPaint`, é um ponteiro para uma estrutura contendo informações de desenho, que é preenchida automaticamente pela função. A estrutura é a seguinte:

```
typedef struct tagPAINTSTRUCT {
  HDC hdc; // identificador do DC de vídeo usado para desenho
  BOOL fErase; // flag indicando se o fundo será apagado
  RECT rcPaint; // retângulo que deve ser redesenhado
  BOOL fRestore; // reservado, utilizado internamente pelo sistema
  BOOL fIncUpdate; // reservado, utilizado internamente pelo sistema
  BYTE rgbReserved[32]; // reservado, utilizado internamente pelo sistema
} PAINTSTRUCT;
```

O primeiro membro, `HDC hDC`, recebe o identificador do DC de vídeo que será utilizado para desenhar gráficos (o mesmo retornado pela função `BeginPaint()`).

`BOOL fErase` é um *flag* que indica se o fundo da janela deve ou não ser apagado, utilizando a cor declarada no membro `WNDCLASSEX.hbrBackground` da classe da janela.

O membro `RECT rcPaint` armazena o retângulo da área que deve ser redesenhada e também indica a área de corte (*clipping region*), ou seja, qualquer área fora desse retângulo será descartada para fins de atualização. A estrutura `RECT` tem seus membros auto-explicativos e é definida como:

```
typedef struct _RECT {
  LONG left; // coordenada x inicial (ângulo esquerdo superior)
  LONG top; // coordenada y inicial (ângulo esquerdo superior)
  LONG right; // coordenada x final (ângulo direito inferior)
  LONG bottom; // coordenada y final (ângulo direito inferior)
} RECT;
```

Os outros membros são reservados e são utilizados internamente pelo Windows. O retorno da função `BeginPaint()` é o identificador do DC para a janela especificada, se não houver erros. Se ocorrer algum erro, a função retorna `NULL`, indicando que nenhum DC de vídeo está disponível.

Após a chamada da função `BeginPaint()`, fazemos toda a codificação para o desenho dos gráficos. Feito isso, devemos indicar o fim do desenho e também a validação da área, através da função `EndPaint()`.

```
BOOL EndPaint(
  HWND hWnd, // identificador da janela
  CONST PAINTSTRUCT *lpPaint // ponteiro para estrutura que contém
informações de desenho
);
```

Os parâmetros de `EndPaint()` são idênticos ao da função `BeginPaint()` e seu retorno sempre é diferente de zero.

A codificação da mensagem `WM_PAINT` tem a seguinte estrutura genérica (sendo que `HDC hDC` e `PAINTSTRUCT psPaint` já foram declaradas anteriormente):

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  //
  // Código para desenhar gráficos
  //

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 4.1: mensagem `WM_PAINT`.

Gráficos fora da WM_PAINT

Há casos em que queremos desenhar gráficos na área cliente sem ser durante o processamento da mensagem WM_PAINT; por exemplo, a cada clique do botão esquerdo do mouse (mensagem WM_LBUTTONDOWN), podemos desenhar um círculo na tela. Nesse caso, não utilizamos as funções BeginPaint() e EndPaint() para obtermos o identificador do DC de vídeo (eles são utilizados somente na mensagem WM_PAINT), mas sim as funções GetDC() e ReleaseDC().

Uma das diferenças entre essas funções é que GetDC() e ReleaseDC() não utilizam apenas a parte inválida da área cliente, mas sim ela inteira. Assim, não ficamos limitados a desenhar gráficos apenas nas áreas que houve invalidação. Outro detalhe é que utilizando essas funções nós não validamos a área cliente; a validação é feita sempre durante a mensagem WM_PAINT. Vamos verificar como funcionam as funções GetDC() e ReleaseDC():

```
HDC GetDC(
  HWND hWnd // identificador de uma janela
);
```

A função GetDC() recebe como único parâmetro um identificador de uma janela (geralmente o da janela principal) e retorna o identificador do DC da área cliente da janela especificada ou NULL se ocorreu algum erro. O parâmetro HWND hWnd também pode receber NULL, indicando que o DC será da tela inteira.

Com o retorno da GetDC() sendo diferente de NULL, fazemos todas as ações necessárias para desenhar nossos gráficos. Depois, devemos liberar o DC para que ele seja utilizado por outros programas, através da função ReleaseDC():

```
int ReleaseDC(
  HWND hWnd, // identificador da janela
  HDC hDC // identificador do DC
);
```

O primeiro parâmetro da função recebe o identificador da janela da qual o DC será liberado. O segundo parâmetro recebe o identificador do DC a ser liberado. A função retorna 1 se o DC é liberado ou zero caso contrário.

Nota: assim como ocorre com BeginPaint() e EndPaint(), para cada GetDC() chamado, deve existir um ReleaseDC() correspondente.

Como foi citado o exemplo de desenhar um círculo a cada clique do botão esquerdo do mouse, vamos ver como podemos utilizar `GetDC()` e `ReleaseDC()`, processando a mensagem `WM_LBUTTONDOWN` (dentro da função `WindowProc()`), com o seguinte trecho de código (Listagem 4.2):

```
case WM_LBUTTONDOWN: // Botão esquerdo do mouse pressionado
{
  // Obtém posição (x, y) atual do cursor do mouse
  int x = LOWORD(lParam);
  int y = HIWORD(lParam);

  // Obtém identificador do DC
  hDC = GetDC(hWnd);

  // Desenha círculo
  Ellipse(hDC, x - 10, y - 10, x + 10, y + 10);

  // Libera DC
  ReleaseDC(hWnd, hDC);

  return(0);
} break;
```

Listagem 4.2: Uso das funções `GetDC()` e `ReleaseDC()`.

Na mensagem `WM_LBUTTONDOWN`, podemos obter a coordenada (x, y) na área cliente onde o cursor do mouse está localizado no momento do clique do botão. Essa informação é obtida através do parâmetro adicional `lParam`. A coordenada x está no bit menos significativo de `lParam` e a coordenada y está no bit mais significativo de `lParam` (obtém-se os valores do bit mais/menos significativo utilizando as macros `HIWORD` e `LOWORD` conforme o exemplo). As mensagens e seus parâmetros adicionais podem ser encontrados na documentação da Win32 API (há dezenas deles, tornando-se inviável citá-los no livro).

Para desenhar um círculo na tela, utilizamos a função da GDI `Ellipse()`, que será explicada no próximo capítulo. O exemplo nos serviu apenas para ilustrar como são utilizadas as funções `GetDC()` e `ReleaseDC()`.

Nota: um detalhe importante deve ser observado! Como não estamos processando a mensagem `WM_PAINT`, quando ocorrer invalidação da área onde desenhamos o gráfico, seu conteúdo será perdido, pois o programa não sabe que modificamos o conteúdo da área cliente e que essa modificação deve ser permanecida.

Gerando a mensagem WM_PAINT

Quando vimos o processamento da mensagem `WM_PAINT`, observamos que o membro `rcPaint` da estrutura `PAINTSTRUCT` armazena a área que está inválida, atualizando e permitindo o desenho de gráficos apenas dentro dessa área, cortando fora (*clipping*) todo o resto que não está no limite dos valores do retângulo.

Caso haja a necessidade de desenhar gráficos fora da área que está inválida, devemos informar ao programa que estamos invalidando outra área, com a função `InvalidateRect()`. O uso dessa função gera uma mensagem `WM_PAINT`, portanto, podemos utilizar essa função para atualizar a área cliente logo após ocorrer uma ação do teclado, por exemplo. Ambos exemplos citados aqui são demonstrados após a explicação do protótipo da `InvalidateRect()`.

```
BOOL InvalidateRect(
  HWND hWnd, // identificador da janela
  CONST RECT *lpRect, // ponteiro para coordenadas do retângulo
  BOOL bErase // flag para limpar fundo
);
```

O parâmetro `HWND hWnd` recebe o identificador da janela onde receberá a nova área inválida. Se receber `NULL`, o Windows invalida e redesenha todas as janelas.

O segundo parâmetro recebe um ponteiro para uma variável do tipo `RECT`, que contém as coordenadas que serão adicionadas à área de atualização/invalidação. Esse parâmetro também pode receber `NULL`, fazendo com que toda área cliente seja invalidada.

O último parâmetro, `BOOL bErase`, é um *flag* que determina se o fundo da área inválida será apagado no próximo processo da mensagem `WM_PAINT`. Caso seja `TRUE`, todo o conteúdo da área cliente é apagado quando a função `BeginPaint()` for chamada, permanecendo apenas o que for informado para ser desenhado dentro da `WM_PAINT`. Em caso de `FALSE`, o fundo não é modificado.

Para invalidar toda a área cliente na mensagem `WM_PAINT` e ser possível desenhar gráficos em qualquer região dela, utilize o seguinte código (Listagem 4.3):

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
```

```
  // Invalida toda a área cliente sem modificar o fundo
  InvalidateRect(hWnd, NULL, FALSE);

  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  //
  // Código para desenhar gráficos
  //

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 4.3: Invalidando a área cliente.

Conforme o segundo exemplo mencionado, vamos enviar uma mensagem `WM_PAINT` ao programa logo após uma ação do teclado, invalidando uma área onde escreveremos um texto armazenado numa variável `szMensagem` (Listagem 4.4):

```
char szMensagem[1]; // Armazena tecla

case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Mostra mensagem na área cliente
  TextOut(hDC, 8, 8, szMensagem, 1);

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;

case WM_CHAR: // Tecla foi pressionada
{
  // Coordenadas da área a ser invalidada
  RECT rcArea = { 8, 8, 28, 28 };

  // Indica qual tecla foi pressionada
  sprintf(szMensagem, "%c",  (char)wParam);

  // Invalida área e gera mensagem WM_PAINT
  InvalidateRect(hWnd, &rcArea, FALSE);

  return(0);
} break;
```

Listagem 4.4: Enviando mensagem `WM_PAINT`.

Validando áreas

Assim como podemos invalidar partes da área cliente, também podemos validá-las, através da função `ValidateRect()`, que valida a parte da área cliente especificada num retângulo (passada como parâmetro para a função).

```
BOOL ValidateRect(
  HWND hWnd, // identificador da janela
  CONST RECT *lpRect // ponteiro para coordenadas do retângulo
);
```

O parâmetro `HWND hWnd` recebe o identificador da janela onde receberá a área que será validada. Se receber `NULL`, o Windows valida e redesenha todas as janelas.

O segundo parâmetro recebe um ponteiro para uma variável do tipo `RECT`, que contém as coordenadas que serão removidas da área de atualização/invalidação. Esse parâmetro também pode receber `NULL`, fazendo com que toda área cliente seja validada.

A função retornará zero caso ocorra erros ou um valor diferente de erro caso contrário.

Não faz muito sentido validarmos parte ou toda a área cliente, pois isso é feito ao processarmos a mensagem `WM_PAINT` com as funções `BeginPaint()` e `EndPaint()`. Então, quando podemos validar manualmente a área cliente?

A validação manual da área cliente pode ser feita quando estamos utilizando um DC privado (assunto de um capítulo posterior), onde não precisamos obrigatoriamente obter o identificador do DC na mensagem `WM_PAINT` ou utilizar as funções `BeginPaint()` e `EndPaint()`, porém, ainda devemos fazer a validação da área cliente.

Objetos GDI

Além do DC, precisamos trabalhar com objetos GDI. Os objetos GDI são tipos de variáveis definidas pela Win32 API os quais devem ser criados e selecionados no DC antes de desenharmos gráficos na área cliente.

Os principais objetos GDI são: caneta (*pen*), pincel (*brush*), bitmap, fonte (*font*) e região (*region*). Cada um desses objetos tem um propósito, que serão discutidos futuramente em maiores detalhes. Nesse momento, devemos saber que para utilizar cada objeto, o mesmo deve estar selecionado no DC, e isso é feito com a função `SelectObject()`:

```
HGDIOBJ SelectObject(
  HDC hdc, // identificador do DC
  HGDIOBJ hgdiobj // identificador do objeto GDI
);
```

O primeiro parâmetro da função é o identificador do DC para o qual estamos selecionando o objeto. O segundo parâmetro recebe o objeto que vai ser associado ao DC. Note que o segundo parâmetro é do tipo `HGDIOBJ`. Podemos considerar esse tipo de variável como sendo um objeto genérico para os outros citados, podendo receber qualquer valor listado na Tabela 4.1.

Caso o objeto selecionado no DC não seja uma região, o retorno da função será o identificador do objeto que estava selecionado anteriormente no DC, caso a função seja bem sucedida; a função retorna `NULL` se não foi bem sucedida.

Se o objeto selecionado for uma região e não ocorrer erros, o retorno da função poderá ser `SIMPLEREGION` (a região contém apenas um retângulo), `COMPLEXREGION` (a região contém mais de um retângulo) ou `NULLREGION` (a região é vazia). Quando ocorrer erro, o retorno será `HGDI_ERROR`.

Em todos os casos, o objeto que é retornado pela função é do mesmo tipo do objeto que foi selecionado no DC.

Valor (tipo de variável)	Objeto GDI
`HPEN`	Caneta (*pen*)
`HBRUSH`	Pincel (*brush*)
`HBITMAP`	Bitmap (*bitmap*)
`HFONT`	Fonte (*font*)
`HRGN`	Região (*region*)
`HGDIOBJ`	Objeto GDI genérico

Tabela 4.1: Objetos GDI.

Sempre que selecionarmos um objeto no DC, logo após utilizá-lo, devemos restaurar o objeto que estava previamente selecionado no DC, pois

isso evita problemas inesperados que possam ocorrer. Veja o trecho de código a seguir (Listagem 4.5), que seleciona uma caneta no DC e depois restaura a caneta antiga (não se preocupe em entender como criar uma caneta, mas sim como funciona a função `SelectObject()`):

```
HPEN hPen; // Nova caneta;
HPEN hPenOld; // Caneta antiga

// Cria nova caneta
hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 0));

// Seleciona nova caneta no DC e salva antiga
hPenOld = (HPEN)SelectObject(hDC, hPen);

//
// Código para desenhar gráficos
//

// Nova caneta não vai ser mais utilizada, restaura antiga
SelectObject(hDC, hPenOld);

// Exclui a nova caneta e libera memória associada à ela
DeleteObject(hPen);
```

Listagem 4.5: Usando a função `SelectObject()`.

Você deve ter percebido que a última linha desse código contém uma função que ainda não foi discutida aqui: `DeleteObject()` exclui um objeto e libera a memória associada ao mesmo. Uma vez excluído, o identificador do objeto não é mais válido.

```
BOOL DeleteObject(
  HGDIOBJ hObject // identificador do objeto gráfico
);
```

O único parâmetro da função, `HGDIOBJ hObject`, recebe o objeto que será excluído, podendo receber qualquer um dos objetos da Tabela 4.1.

A função retorna um valor diferente de zero quando a exclusão for bem sucedida. Se o identificador especificado não for válido ou está selecionado no DC, a função retorna zero.

Nota: não devemos deletar objetos que ainda estejam selecionados no DC; primeiro devemos retirá-lo do DC, restaurando o objeto que estava selecionado no DC anteriormente. Também não podemos selecionar mais de um objeto no DC ao mesmo tempo, pois isso é uma limitação da GDI. Basta ver como o MS-Paint funciona: podemos utilizar apenas um objeto

de desenho por vez. Por exemplo, quando escolhemos o lápis, não podemos trabalhar ao mesmo tempo com o balde de tinta, e vice-versa.

Obtendo informações de um objeto GDI

Além de criarmos objetos GDI, podemos obter as informações dos mesmos. Isso é útil, por exemplo, quando queremos modificar as informações de um objeto GDI em tempo de execução, como a cor de um pincel, espessura da caneta, etc... Para obter as informações, utilizamos a função `GetObject()`:

```
int GetObject(
  HGDIOBJ hgdiobj, // identificador do objeto GDI
  int cbBuffer, // tamanho do buffer das informações do objeto
  LPVOID lpvObject // buffer para armazenar as informações do objeto
);
```

O primeiro parâmetro da função (`HGDIOBJ hgdiobj`) recebe o identificador do objeto GDI de onde iremos obter as informações.

O segundo parâmetro, `int cbBuffer`, indica o tamanho em bytes que serão gravados no buffer (o terceiro parâmetro). O valor desse parâmetro deve indicar o tamanho do objeto que estamos obtendo as informações, portanto, basta enviar o comando `sizeof(`*tipo*`)` como parâmetro, onde *tipo* é o tipo do objeto do primeiro parâmetro (consulte a Tabela 4.1).

O último parâmetro, `LPVOID lpvObject`, recebe um ponteiro `void` para o buffer onde as informações serão armazenadas. O valor de `lpvObject` recebe uma estrutura conforme o objeto GDI que foi enviado no primeiro parâmetro. A Tabela 4.2 lista quais estruturas podem ser enviadas para `lpvObject`.

Valor	Objeto GDI (primeiro parâmetro)
`BITMAP`	`HBITMAP` ou `HBITMAP` retornado da função `CreateDIBSection()` e se `cbBuffer` é `sizeof(BITMAP)`.
`DIBSECTION`	`HBITMAP` retornado da função `CreateDIBSection()` e se `cbBuffer` é `sizeof(DIBSECTION)`.
`EXTLOGPEN`	`HPEN` retornado da função `ExtCreatePen()`.
`LOGPEN`	`HPEN`.
`LOGBRUSH`	`HBRUSH`.
`LOGFONT`	`HFONT`.

Tabela 4.2: Valores para `lpvObject`.

Quando a função é executada corretamente, o retorno pode ser: a) `lpvObject` é um ponteiro válido: o retorno é o número de bytes que foi armazenado no buffer ou b) `lpvbObject` é `NULL`: a função retorna o número de bytes necessário para armazenar as informações, que seriam armazenadas no buffer. Se a função falhar, o retorno é zero.

Nota: veja o último exemplo (Listagem 4.12) do tópico "Trabalhando com fontes" desse capítulo, que exemplifica a utilização da função `GetObject()`.

Escrevendo textos na área cliente

Existem duas principais funções GDI para escrever textos na área cliente: `TextOut()` e `DrawText()`. A primeira é uma função mais simplista, mostrando textos sem formatações, enquanto a segunda suporta formatações de texto, como quebra de linha e tabulações. No programa *prog04-1.cpp*, ambas as funções são utilizadas; `TextOut()` para escrever o texto no canto esquerdo superior e `DrawText()` para escrever "Utilizando GDI" centralizado na área cliente do programa. Vamos ver cada uma dessas funções:

```
BOOL TextOut(
  HDC hdc, // identificador do DC
  int nXStart, // posição x do texto
  int nYStart, // posição y do texto
  LPCTSTR lpString, // ponteiro para string
  int cbString // quantidade de caracteres na string
);
```

A função `TextOut()` recebe o identificador do DC (que obtivemos anteriormente), a posição (x, y) onde o texto será mostrado, um ponteiro para a string contendo o texto, e a quantidade de caracteres da string.

A função retorna um valor diferente de zero se não houve falhas, caso contrário, zero é retornado indicando erro.

O quarto parâmetro, `LPCTSTR lpString`, pode ser uma constante e não precisa ser necessariamente terminado com zero, pois o último parâmetro indica quantos caracteres devem ser impressos, como mostra o exemplo a seguir:

```
TextOut(hDC, 0, 0, "Mostrar o texto", 15);
```

Porém, nem sempre queremos mostrar um texto constante; muitas vezes é preciso mostrar valores e conteúdos de variáveis. `TextOut()` não aceita strings formatadas como o `printf()` da linguagem C. A solução para isso é utilizar a função `sprintf()` para pré-formatar a string antes de passá-la para `TextOut()`, conforme mostra a Listagem 4.6:

```
char texto[255];
char nome[] = "Adauto";
int idade = 20;

sprintf(texto, "%s tem %d anos.", nome, idade);
TextOut(hDC, 0, 0, texto, lstrlen(texto));
```

Listagem 4.6: texto pré-formatado para `TextOut()`.

Usar uma variável do tipo `char[]` (ou `LPSTR`) também facilita quando precisamos passar o último parâmetro da função, pois podemos usar `lstrlen()` para indicarmos a quantidade de caracteres da string automaticamente.

Mesmo utilizando a função `sprintf()` para pré-formatar a string, `TextOut()` não aceita certas seqüências de escape como quebra de linha ou tabulação. Para isso, utilizamos `DrawText()`:

```
int DrawText(
  HDC hDC, // identificador do DC
  LPCTSTR lpString, // ponteiro para string
  int nCount, // quantidade de caracteres da string
  LPRECT lpRect, // ponteiro para retângulo onde o texto será formatado
  UINT uFormat // flags para formatação do texto
);
```

`DrawText()` recebe diferentes parâmetros da `TextOut()`, pois trabalha baseado em áreas retangulares ao invés de coordenadas (x, y). O identificador do DC é passado para o primeiro parâmetro da função, `HDC hDC`.

O segundo parâmetro, `LPCTSTR lpString`, recebe um ponteiro para a string que contém o texto, enquanto `int nCount` recebe a quantidade de caracteres da string. Nesse terceiro parâmetro, podemos passar o valor –1, assim, a função irá calcular automaticamente a quantidade de caracteres da string passada para a função.

O parâmetro `LPRECT lpRect` recebe um ponteiro de uma variável do tipo `RECT`, que define o retângulo onde será impresso o texto informado em `LPCTSTR lpString`.

O último parâmetro, `UINT uFormat`, pode receber diversos valores combinados com o operador `|`, conforme mostra alguns valores da Tabela 4.3, os quais fazem a formatação do texto.

Quando a função `DrawText()` é executada sem problemas, ela retorna a altura do texto ou zero se houve algum problema.

Valor	Descrição
`DT_BOTTOM`	Alinha o texto para a parte inferior do retângulo. Esse valor deve ser combinado com `DT_SINGLELINE`.
`DT_CENTER`	Centraliza o texto horizontalmente no retângulo.
`DT_EXPANDTABS`	Aumenta os caracteres de tabulação. O valor padrão de caracteres por tabulação é oito.
`DT_LEFT`	Alinha o texto à esquerda.
`DT_RIGHT`	Alinha o texto à direita.
`DT_SINGLELINE`	Mostra o texto em uma única linha. Quebra de linhas (`\n`) não têm efeito nesse caso.
`DT_TOP`	Alinha o texto na parte superior do retângulo. Esse valor deve ser combinado com `DT_SINGLELINE`.
`DT_VCENTER`	Centraliza o texto verticalmente. Esse valor deve ser combinado com `DT_SINGLELINE`.
`DT_WORDBREAK`	Quebra de linhas são automaticamente inseridas caso uma palavra ultrapasse a área do retângulo especificado no parâmetro `lpRect`.

Tabela 4.3: Alguns valores de formatação de texto para `DrawText()`.

Podemos utilizar a função `GetClientRect()` para obter o retângulo da área cliente do programa e imprimir um texto centralizado no meio da área cliente (como no programa-exemplo *prog04-1.cpp*):

```
BOOL GetClientRect(
  HWND hWnd, // identificador da janela
  LPRECT lpRect // ponteiro para retângulo onde serão armazenadas as
coordenadas da área cliente
);
```

A função `GetClientRect()` recebe o identificador da janela de onde será obtida a área cliente e um ponteiro para uma variável do tipo `RECT` onde serão

armazenadas as coordenadas da área cliente. Os valores dos membros de `RECT` serão: `left = 0`, `top = 0`, `right = largura da área cliente` e `bottom = altura da área cliente`. A função retorna zero se houve algum erro ou um valor diferente de zero se a função foi executada corretamente.

Para imprimir um texto centralizado no meio da área cliente, utilizamos o seguinte trecho de código (Listagem 4.7):

```
// Obtém área cliente
RECT rcArea;
GetClientRect(hWnd, &rcArea);

// Imprime texto na área cliente
DrawText(hDC, "Utilizando GDI", -1, &rcArea, DT_SINGLELINE | DT_CENTER |
DT_VCENTER);
```

Listagem 4.7: Texto centralizado na área cliente.

Com uma pequena modificação (Listagem 4.8), podemos imprimir o mesmo texto centralizado horizontalmente e centralizado verticalmente utilizando apenas metade da altura da área cliente:

```
// Obtém área cliente
RECT rcArea;
GetClientRect(hWnd, &rcArea);

// Utiliza apenas metade da altura da área cliente
rcArea.bottom = rc.Area.bottom / 2;

// Imprime texto na área determinada
DrawText(hDC, "Utilizando GDI", -1, &rcArea, DT_SINGLELINE | DT_CENTER |
DT_VCENTER);
```

Listagem 4.8: Texto na metade da altura da área cliente.

E caso você queira imprimir um texto de múltiplas linhas, iniciando da coordenada (8, 20) da área cliente, utilize o seguinte código (Listagem 4.9):

```
// Obtém área cliente
RECT rcArea;
GetClientRect(hWnd, &rcArea);

rcArea.left = 8; // Retângulo inicia em x = 8
rcArea.top = 20; // Retângulo inicia em y = 20

// Imprime texto na área determinada
DrawText(hDC, "Essa é a linha 1\nEssa é a linha 2\nEssa é linha 3.", -1,
&rcArea, DT_LEFT);
```

Listagem 4.9: Texto com múltiplas linhas.

Cores RGB – COLORREF

Antes de verificarmos como modificar os atributos de texto, vamos ver um novo tipo de variável: `COLORREF`, variável de 32-bits utilizada para especificar uma cor no formato RGB (de *Red*, *Green*, *Blue*, ou seja, Vermelho, Verde, Azul). Podemos criar cerca de 16,7 milhões de cores através da combinação de diferentes valores para os bits que armazenam informações sobre as cores, pois é possível definir 256 intensidades/tons diferentes para cada uma dessas três cores. A Figura 4.3 demonstra como são utilizados os bits da `COLORREF` e sua estrutura.

Nota: obviamente, a quantidade máxima de cores que será visualizada dependerá do monitor e da placa de vídeo utilizada.

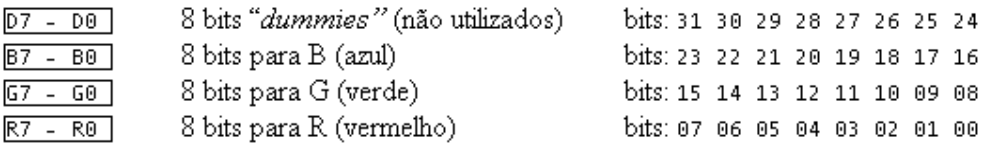

Figura 4.3: A estrutura `COLORREF` e como seus bits são armazenados.

Na GDI, as funções que recebem parâmetros de cor utilizam a variável `COLORREF`. Podemos definir uma cor RGB de duas maneiras: explicitamente, utilizando valores em hexadecimal, ou através da macro `RGB()`.

Quando uma cor RGB é definida através de valores hexadecimais, o valor da variável do tipo `COLORREF` deve ser informado no formato 0x00BBGGRR, ou seja, o byte menos significativo (RR) armazena a intensidade da cor vermelha, o segundo byte (GG) armazena a intensidade da cor verde, o terceiro byte (BB) armazena a intensidade da cor azul e o byte mais significativo (00) deve ser zero. Os valores de cada byte podem estar entre os intervalos 00 e FF.

O mais comum e prático é utilizar a macro `RGB()` para definir uma cor. A definição da macro RGB é a seguinte:

```
#define RGB(r, g, b)  ((DWORD) (((BYTE)(r) | ((WORD)(g) << 8)) | (((DWORD)
(BYTE)(b)) << 16)))
```

Podemos pensar nessa macro como uma função com o seguinte protótipo:

```
COLORREF RGB(
  BYTE bRed, // cor vermelha
  BYTE bGreen, // cor verde
  BYTE bBlue // cor azul
);
```

Onde a função recebe três parâmetros que são os valores das intensidades de cada cor e retorna uma variável do tipo `COLORREF` contendo a cor informada nos parâmetros da função.

A Tabela 4.4 mostra algumas cores criadas com a macro `RGB()` e também criadas explicitamente através de valores hexadecimais:

RGB()	Valor hexadecimal	Cor
`RGB(0, 0, 0)`	`0x00000000`	Preto
`RGB(255, 255, 255)`	`0x00FFFFFF`	Branco
`RGB(255, 0, 0)`	`0x000000FF`	Vermelho claro
`RGB(128, 0, 0)`	`0x0000007F`	Vermelho escuro
`RGB(0, 255, 0)`	`0x0000FF00`	Verde claro
`RGB(0, 128, 0)`	`0x00007F00`	Verde escuro
`RGB(0, 0, 255)`	`0x00FF0000`	Azul claro
`RGB(0, 0, 128)`	`0x007F0000`	Azul claro
`RGB(255, 255, 0)`	`0x0000FFFF`	Amarelo claro
`RGB(128, 128, 0)`	`0x00007F7F`	Amarelo escuro
`RGB(255, 0, 255)`	`0x00FF00FF`	Magenta claro
`RGB(128, 0, 128)`	`0x007F007F`	Magenta escuro
`RGB(0, 255, 255)`	`0x00FFFF00`	Ciano claro
`RGB(0, 128, 128)`	`0x007F7F00`	Ciano escuro

Tabela 4.4: Algumas cores RGB.

Nota: os programas Windows sempre utilizam o padrão de cores 24-bit (8 bits por canal/cor). Caso a placa de vídeo esteja configurada para uma quantidade diferente de cores, o Windows faz uma aproximação com as cores disponíveis no sistema.

Modificando atributos de texto

Quando `TextOut()` ou `DrawText()` são utilizados, o resultado do texto impresso na área cliente não é muito agradável, pois vemos o texto com cor preta e fundo branco, mesmo se o fundo da área cliente for, por exemplo, verde.

Isso ocorre porque não definimos alguns atributos da fonte-padrão utilizada pelos programas – a cor do texto, a cor de fundo e o modo como o fundo é misturado com a área cliente. Para isso, utilizamos as funções `SetTextColor()`, `SetBkColor()` e `SetBkMode()`, respectivamente.

`SetTextColor()` é utilizada para definir a cor do texto impressa com `TextOut()` ou `DrawText()` em um DC específico.

```
COLORREF SetTextColor(
  HDC hdc, // identificador do DC
  COLORREF crColor // cor do texto
);
```

A função recebe o identificador do DC, no primeiro parâmetro, onde o atributo de cor de texto será modificado com a cor especificada no parâmetro `COLORREF crColor`. O retorno da função é a cor de texto que estava definida anteriormente ou `CLR_INVALID` caso a função falhe.

O uso da cor de retorno da função é útil quando queremos modificar a cor de um texto e depois voltar para a cor original antes da modificação. O trecho de código (Listagem 4.10) a seguir exemplifica esse trabalho:

```
COLORREF crOld;

// Modifica cor e salva cor antiga em crOld
crOld = SetTextColor(hDC, RGB(255, 0, 0));

// Mostra texto vermelho
TextOut(hDC, 0, 0, "Texto em Vermelho", lstrlen("Texto em Vermelho"));

// Restaura cor antiga
SetTextColor(hDC, crOld);

// Mostra texto na cor original
TextOut(hDC, 50, 50, "Cor Original", lstrlen("Cor Original"));
```

Listagem 4.10: Modificando cor do texto.

Para modificar a cor de fundo atual do texto, utilizamos a função `SetBkColor()`.

```
COLORREF SetBkColor(
  HDC hdc, // identificador do DC
  COLORREF crColor // cor do fundo
);
```

O primeiro parâmetro da função recebe o identificador do DC e o segundo recebe a cor que será atribuída ao fundo do texto. O retorno da

função é a cor de fundo que estava definida anteriormente ou `CLR_INVALID` caso a função falhe.

Assim como `SetTextColor()`, o retorno `COLORREF` da função `SetBkColor()` pode ser utilizado para armazenar a cor de fundo antiga e restaurá-la quando necessário.

Nota: `SetBkColor()` também é utilizada para preencher os espaços entre estilos de linhas desenhadas usando uma caneta (*pen*) criado com a função `CreatePen()`. Estudaremos *pens* e suas funções no próximo capítulo.

Além de modificar a cor da face do texto e do seu fundo, podemos definir como a cor de fundo é misturada com a área cliente, através da função `SetBkMode()`.

```
int SetBkMode(
  HDC hdc, // identificador da janela
  int iBkMode // modo de fundo
);
```

A função `SetBkMode()` recebe o identificador do DC como primeiro parâmetro; no segundo, `int iBkMode`, definimos o modo que a cor de fundo será misturada com a área cliente. `iBkMode` recebe o valor `OPAQUE`, caso desejemos que o fundo do texto seja preenchido com a cor de fundo (definida em `SetBkColor()`) ou `TRANSPARENT`, para que o fundo do texto fique transparente.

O retorno da função, quando bem sucedida, é o valor do modo de fundo (`iBkMode`) que estava definido anteriormente. Caso contrário, o retorno é zero.

Nota: `SetBkMode()` também afeta os estilos de linhas desenhadas pela função `CreatePen()`.

Para exemplificar o uso dessas três funções (Listagem 4.11), vamos imprimir dois textos na área cliente, um de fundo amarelo e opaco (o segundo parâmetro de `SetBkColor()` recebe `OPAQUE`) com texto azul e outro de fundo transparente e texto verde escuro:

```
COLORREF crOldText; // Armazena cor de texto antiga
COLORREF crOldBk; // Armazena cor de fundo antiga
int iOldMode; // Armazena modo de fundo antigo
```

```
// Muda cor de texto para azul e salva antiga
crOldText = SetTextColor(hDC, RGB(0, 0, 255));

// Muda cor de fundo para amarelo e salva antiga
crOldBk = SetBkColor(hDC, RGB(255, 255, 0));

// Muda modo de fundo para opaco e salva antigo
iOldMode = SetBkMode(hDC, OPAQUE);

// Mostra texto em azul e fundo amarelo
TextOut(hDC, 0, 0, "Brasil", 6);

// Muda cor de texto para verde escuro
SetTextColor(hDC, RGB(0, 128, 0));

// Muda modo de fundo para transparente
SetBkMode(hDC, TRANSPARENT);

/* Note que não salvamos as cores antigas, pois o que queremos restaurar
depois são as cores que estavam definidas anteriormente (as quais já foram
salvas na primeira chamada das funções SetTextColor() e SetBkMode()). */

// Mostra texto em verde escuro e fundo transparente
TextOut(hDC, 0, 20, "Amazônia", 8);

// Restaura configurações antigas
SetTextColor(hDC, crOldText);
SetBkColor(hDC, crOldBk);
SetBkMode(hDC, iOldMode);
```

Listagem 4.11: Mudando atributos de cor de texto e fundo.

Trabalhando com fontes

Uma outra opção para melhorar a visualização dos textos em nossos programas é modificar a fonte utilizada para impressão na área cliente. Conforme mencionado em "Objetos GDI", fontes são objetos GDI e devem ser criados e associados ao DC que estamos trabalhando. Podemos criar um objeto fonte (`HFONT`) de duas maneiras: através da chamada à função `CreateFont()` ou à função `CreateFontIndirect()`. A diferença entre elas é que a primeira recebe 14 parâmetros informando o tipo de fonte, enquanto a segunda recebe uma estrutura `LOGFONT` que contém esses 14 parâmetros. Vamos ver como utilizar ambas funções:

```
HFONT CreateFont(
  int nHeight, // altura da fonte
  int nWidth, // largura da fonte
  int nEscapement, // ângulo do texto
  int nOrientation, // ângulo do texto
  int fnWeight, // espessura da fonte
  DWORD fdwItalic, // estilo de fonte itálica
  DWORD fdwUnderline, // estilo de fonte sublinhada
  DWORD fdwStrikeOut, // estilo de fonte tachada
  DWORD fdwCharSet, // identificador do estilo dos caracteres
```

```
  DWORD fdwOutputPrecision, // precisão de saída
  DWORD fdwClipPrecision, // precisão de corte (clipping)
  DWORD fdwQuality, // qualidade de saída
  DWORD fdwPitchAndFamily, // pitch e família da fonte
  LPCTSTR lpszFace // nome da fonte
);
```

O primeiro parâmetro, `int nHeight`, especifica a altura da fonte. Esse parâmetro pode receber três tipos de valores: se `nHeight` for zero, o programa utiliza o tamanho padrão da fonte; se maior que zero, o programa obtém a altura da cela dos caracteres da fonte, e se for menor que zero, o programa obtém o valor absoluto da altura dos caracteres da fonte.

Nota: a cela de caractere é um retângulo imaginário que abrange cada caractere da fonte. Sua altura é calculada obtendo a distância entre a parte debaixo da letra "g" minúscula até o topo da letra "m" maiúscula, mais um valor conhecido por *internal leading*. A altura do *internal leading* é ocupada por caracteres que possuem acentos, tais como as letras é, ô, ü. A altura de um caractere é obtida subtraindo a altura do *internal leading* pela altura da cela. Veja esse esquema demonstrado através da Figura 4.4.

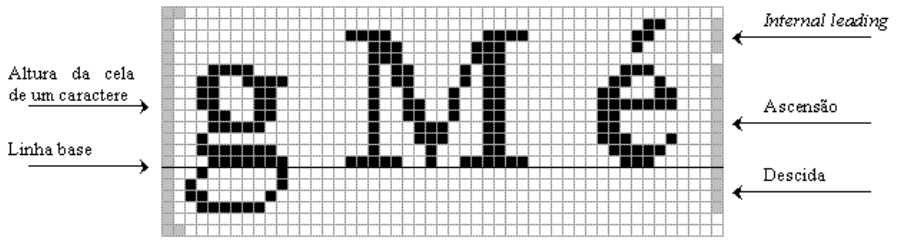

Figura 4.4: Cela, caractere e *internal leading*.

O segundo parâmetro, `int nWidth`, especifica a largura dos caracteres da fonte. Para que o mapeador de fontes calcule automaticamente a largura padrão para nós, enviamos zero para esse parâmetro.

O terceiro parâmetro, `int nEscapement`, especifica o ângulo de rotação da fonte, porém, cada ângulo é passado para esse parâmetro sendo multiplicado por 10: o ângulo de 90 graus deve ser passado como 900, o de 30 graus deve ser passado como 300, etc... O parâmetro `int nOrientation` deve receber o mesmo valor que `nEscapement`.

O parâmetro `int fnWeight` identifica a espessura da fonte (entre 0 a 1000), indicando se o texto será escrito com a fonte normal ou em negrito. Os

valores da Tabela 4.5 podem ser enviados a esse parâmetro. O valor zero faz com que uma espessura padrão da fonte seja utilizada.

Valor	Espessura
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400 (normal)
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700 (negrito)
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

Tabela 4.5: Espessuras da fonte.

Os três próximos parâmetros, `DWORD fdwItalic`, `DWORD fdwUnderline` e `DWORD fdwStrikeOut` indicam se a fonte será itálica, sublinhada e/ou tachada, respectivamente. Cada parâmetro recebe `TRUE` (aplica estilo na fonte) ou `FALSE` (não aplica estilo).

O parâmetro `DWORD fdwCharSet` especifica o estilo dos caracteres da fonte e pode receber um dos valores da Tabela 465. O valor associado a esse parâmetro deve ser do mesmo estilo de caracteres da fonte informada no parâmetro `LPCTSTR lpszFace`. Em alguns casos, é possível utilizar o valor `DEFAULT_CHARSET`, mas devemos ter certeza que a fonte especificada realmente existe na máquina onde o programa está sendo executado.

Valor
ANSI_CHARSET
BALTIC_CHARSET
CHINESEBIG5_CHARSET
DEFAULT_CHARSET
EASTEUROPE_CHARSET

GB2312_CHARSET	
GREEK_CHARSET	
HANGUL_CHARSET	
MAC_CHARSET	
OEM_CHARSET	
RUSSIAN_CHARSET	
SHIFTJIS_CHARSET	
SYMBOL_CHARSET	
TURKISH_CHARSET	
VIETNAMESE_CHARSET	
JOHAB_CHARSET	(versão Windows na linguagem coreana)
ARABIC_CHARSET	(versão Windows na linguagem do Oriente Médio)
HEBREW_CHARSET	(versão Windows na linguagem do Oriente Médio)
THAI_CHARSET	(versão Windows na linguagem tailandesa)

Tabela 4.6: Valores para `fdwCharSet`.

O parâmetro `DWORD fdwOutputPrecision` especifica a precisão de saída, definindo o quão próximo a saída deve ser em relação aos valores fornecidos para a fonte (altura, largura, ângulo, pitch e tipo de fonte). O parâmetro pode receber um dos valores da Tabela 4.7.

Valor	Descrição
OUT_CHARACTER_PRECIS	Não é utilizado.
OUT_DEFAULT_PRECIS	Especifica o comportamento padrão do mapeador de fontes.
OUT_DEVICE_PRECIS	Instrui o mapeador de fontes a escolher uma fonte de dispositivo quando o sistema possui contém diversas fontes com o mesmo nome (por exemplo, uma fonte fornecida pelo fabricante de impressora).
OUT_OUTLINE_PRECIS	Instrui o mapeador de fontes a escolher fontes *True Type* ou variações de fontes *outline* quando existe mais de uma fonte com o mesmo nome.
OUT_PS_ONLY_PRECIS	Instrui o mapeador de fontes a escolher apenas fontes *PostScript*. Caso não exista nenhuma fonte *PostScript* instalada no sistema,

	o mapeador retorna o comportamento padrão.
`OUT_RASTER_PRECIS`	Instrui o mapeador de fontes a escolher uma variação de fontes *raster* quando existe mais de uma fonte com o mesmo nome.
`OUT_STRING_PRECIS`	Retornado quando fontes *raster* são enumeradas.
`OUT_STROKE_PRECIS`	Retornado quando fontes *True Type* ou vetoriais são enumeradas. No Windows 95/98/Me, também é utilizado para o mapeador de fontes escolher variações de fontes vetoriais ou *True Type*.
`OUT_TT_ONLY_PRECIS`	Instrui o mapeador de fontes a escolher apenas fontes *True Type*. Caso não exista nenhuma fonte *True Type* instalada no sistema, o mapeador retorna o comportamento padrão.
`OUT_TT_PRECIS`	Instrui o mapeador de fontes a escolher fontes *True Type* quando existe mais de uma fonte com o mesmo nome.

Tabela 4.7: Valores para `fdwOutputPrecision`.

Podemos utilizar os valores `OUT_DEVICE_PRECIS`, `OUT_RASTER_PRECIS`, `OUT_TT_PRECIS` ou `OUT_PS_ONLY_PRECIS` para controlar como o mapeador de fontes escolhe uma fonte quando o sistema possui diversas com o mesmo nome. Por exemplo, caso exista uma fonte chamada "Symbol" tanto como *True Type* quanto *raster*, ao especificarmos o valor `OUT_TT_PRECIS`, o mapeador de fonte escolherá a versão *True Type*. Utilizando `OUT_TT_ONLY_PRECIS` força o mapeador de fontes a escolher uma fonte *True Type*, mesmo que ela seja substituída por uma fonte de outro nome.

O parâmetro DWORD `fdwClipPrecision` indica a precisão de corte, definindo como cortar os caracteres que estão parcialmente do lado de fora região de corte, através de um dos valores da Tabela 4.8.

Valor	Descrição
`CLIP_DEFAULT_PRECIS`	Precisão de corte padrão.
`CLIP_EMBEDDED`	Este valor deve ser especificado quando utilizamos uma fonte *embedded read-only*.
`CLIP_LH_ANGLES`	Quando utilizado, a rotação das fontes

	depende do sistema de coordenadas, caso contrário, a rotação sempre ocorre no sentido anti-horário.
CLIP_STROKE_PRECIS	Valor retornado quando fontes são enumeradas.

Tabela 4.8: Valores para fdwClipPrecision.

Definimos a qualidade de saída do texto informando um dos valores da Tabela 4.9 para o parâmetro DWORD fdwQuality. A qualidade de saída define como a GDI deve igualar os atributos da LOGFONT com a fonte em si.

Valor	Descrição
ANTIALIASED_QUALITY	A impressão da fonte é feita suavemente.
DEFAULT_QUALITY	Qualidade padrão da fonte.
DRAFT_QUALITY	A aparência da fonte não é importante.
NONANTIALIASED_QUALITY	A impressão da fonte não é feita suavemente.
PROOF_QUALITY	A qualidade da fonte é alta e não ocorrem distorções na aparência.

Tabela 4.9: Valores para fdwQuality.

DWORD fdwPitchAndFamily define o *pitch* e a família da fonte. *Pitch* está ligado ao espaçamento entre os caracteres, podendo ser fixo ou variável. Um *pitch* fixo é como a fonte Courier New, onde todas as celas dos caracteres têm o mesmo espaçamento – ou seja, mesmo que a letra i tenha uma largura menor que a letra o, ambas ocupam a mesma largura na impressão. Fontes com *pitch* variáveis têm celas de larguras diferentes, dependendo do caractere. A Figura 4.5 demonstra exemplos de fontes com *pitch* fixo e variável.

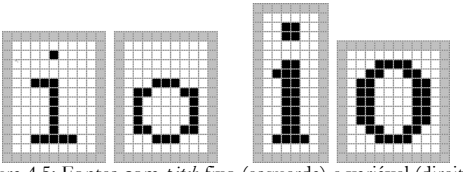

Figura 4.5: Fontes com *pitch* fixo (esquerda) e variável (direita).

O parâmetro fdwPitchAndFamily pode receber DEFAULT_PITCH (*pitch* definido pela fonte), FIXED_PITCH (força o *pitch* da fonte a ser fixo – basicamente, o mapeador de fonte escolhe uma outra fonte que tem o *pitch* fixo) ou VARIABLE_PITCH (força o *pitch* da fonte a ser variável). Podemos ainda fazer uma combinação (usando o operador ou bit-a-bit) com os valores da

Tabela 4.10, que descrevem o estilo da família da fonte para o mapeador de fontes (que usa a informação quando a fonte requisitada não existe).

Valor	Descrição
FF_DECORATIVE	Fontes diferentes, como *Old English*.
FF_DONTCARE	Usar a fonte padrão.
FF_MODERN	Fontes com traçado constante, como *Courier New*.
FF_ROMAN	Fontes com traçado variável e com *serif*, como *MS Serif*.
FF_SCRIPT	Fontes com estilo de escrita à mão, como *Script*.
FF_SWISS	Fontes com traçado variável e sem *serif*, como *MS Sans Serif*.

Tabela 4.10: Famílias das fontes.

O último parâmetro, `LPCTSTR lpszFace`, recebe uma string especificando o nome da fonte que será utilizada. O tamanho da string não deve ultrapassar 32 caracteres, contando com o último byte `\0` (*null-string*). Se esse parâmetro receber `NULL` ou uma string vazia, a GDI usa a primeira fonte que coincide com os atributos informados nos outros parâmetros.

Quando não houver problemas, a função retorna um identificador `HFONT` com os atributos passados para os parâmetros. Caso contrário, o retorno é `NULL`.

Outro método para criarmos uma fonte é através da função `CreateFontIndirect()`:

```
HFONT CreateFontIndirect(
  CONST LOGFONT* lplf // características da fonte
);
```

A função `CreateFontIndirect()` recebe um ponteiro para uma estrutura `LOGFONT`, que contém os atributos da fonte que será criada. O retorno dessa função é idêntica à da função `CreateFont()`.

Antes de chamar a função `CreateFontIndirect()`, devemos inicializar os membros da estrutura `LOGFONT`:

```
typedef struct tagLOGFONT {
  LONG lfHeight; // altura da fonte
  LONG lfWidth; // largura da fonte
  LONG lfEscapement; // ângulo do texto
  LONG lfOrientation; // ângulo do texto
  LONG lfWeight; // espessura da fonte
  BYTE lfItalic; // estilo de fonte itálica
  BYTE lfUnderline; // estilo de fonte sublinhada
  BYTE lfStrikeOut; // estilo de fonte tachada
  BYTE lfCharSet; // identificador do estilo dos caracteres
  BYTE lfOutPrecision; // precisão de saída
  BYTE lfClipPrecision; // precisão de corte (clipping)
  BYTE lfQuality; // qualidade de saída
  BYTE lfPitchAndFamily; // pitch e família da fonte
  TCHAR lfFaceName[LF_FACESIZE]; // nome da fonte
} LOGFONT, *PLOGFONT;
```

Note que os membros dessa estrutura são os mesmos dos parâmetros da função `CreateFont()`; assim, a explicação dos parâmetros de `CreateFont()` são aplicados da mesma maneira para a estrutura `LOGFONT`.

Para exemplificar (Listagem 4.12), vamos criar uma fonte *Times New Roman* com tamanho padrão (12), itálica e com ângulo de 45 graus, utilizando ambas funções:

```
// Cria fonte utilizando função CreateFont()
HFONT hFonte1 = NULL;
HFonte = CreateFont(0, 0, 450, 450, FW_NORMAL, TRUE, FALSE, FALSE,
ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
DEFAULT_PITCH, “Times New Roman”);

// Cria fonte utilizando função CreateFontIndirect():
// Cria estrutura LOGFONT e inicializa valores
HFONT hFonte2 = NULL;
LOGFONT lf = { 0, 0, 450, 450, FW_NORMAL, TRUE, FALSE, FALSE, ANSI_CHARSET,
OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH,
“Times New Roman” };

// Cria fonte com as características informadas em LOGFONT
hFonte2 = CreateFontIndirect(&lf);
```

Listagem 4.12: Criando fontes.

Quando queremos utilizar todas as características padrões de uma fonte, basta enviar zero para todos os parâmetros, exceto o nome da fonte:

```
HFONT hFonte3 = CreateFont(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, “Times New
Roman”);
```

Como as funções `CreateFont()` e `CreateFontIndirect()` fazem o mesmo serviço, mudando somente seus parâmetros, qual a diferença entre usar uma e outra? Na realidade, não há nenhuma diferença, porém, a função

`CreateFontIndirect()` é mais flexível quando queremos deixar o usuário modificar os atributos de uma fonte durante a execução do programa, pois obtemos os atributos atuais da fonte, armazenamos numa variável do tipo `LOGFONT` e em seguida fazemos as alterações requisitadas pelo usuário nos membros da variável `LOGFONT`.

Se quisermos modificar as características da fonte `HFONT hFonte2` que criamos nesse tópico, deixando a fonte com ângulo 0, em negrito, retirar o atributo itálico e mudá-la para Arial, podemos usar o seguinte código (Listagem 4.13):

```
// HFONT hFonte2 e LOGFONT lf já foram criados

// Obtém informações da fonte e armazena em lf
GetObject(hFonte2, sizeof(LOGFONT), &lf);

// Modifica ângulo para 0, negrita fonte, retira itálico e muda para Arial
lf.lfEscapement = 0;
lf.lfOrientation = 0;
lf.lfWeight = FW_BOLD;
lf.lfItalic = FALSE;
lstrcpy(lf.lfFaceName, "Arial");

// Define novos atributos para fonte
hFonte2 = CreateFontIndirect(&lf);
```

Listagem 4.13: Modificando atributos da fonte com GetObject().

Verificando o teclado

Na programação Windows, verificamos o estado do teclado de duas maneiras: através do código de tecla virtual ou através do caractere referente à tecla pressionada. Para isso, podemos processar três mensagens do teclado: `WM_KEYDOWN`, `WM_KEYUP` e `WM_CHAR`. As duas primeiras estão relacionadas ao código de tecla virtual, sendo que `WM_KEYDOWN` é enviada ao programa quando o usuário pressiona alguma tecla e `WM_KEYUP` é enviada quando o usuário solta a tecla. `WM_CHAR` é enviada quando uma mensagem `WM_KEYDOWN` é traduzida pela função `TranslateMessage()`.

Nota: geralmente utilizamos `WM_KEYDOWN` para verificar se teclas como F1, ESC, ENTER ou INSERT foram pressionadas e `WM_CHAR` quando estamos manipulando strings com o teclado.

Nas três mensagens, o parâmetro `lParam` armazena os dados da tecla, indicando a contagem de repetição da tecla, *scan code* e *flags*. O `lParam` dessas

mensagens é conhecido como *Bit Encoded Key-Data* e seus bits são compostos de acordo com a Tabela 4.11.

Bits	Descrição
0-15	Contagem de repetição da tecla.
16-23	*Scan code*.
24	Indica se a tecla pressionada é de extensão, como a tecla ALT+CTRL (ALT direito) em teclados de 101 ou 102 teclas. O valor é 1 se a tecla é de extensão ou 0 caso contrário.
25-28	Reservado.
29	Indica código de contexto. O valor é sempre zero na mensagem WM_KEYDOWN.
30	Indica o estado anterior da tecla. O valor é 1 se a tecla estava pressionada antes da mensagem ser enviada ou 0 se tecla não estava pressionada.
31	Indica o estado de transição. O valor é sempre zero na mensagem WM_KEYDOWN.

Tabela 4.11: lParam das mensagens de teclado.

A Tabela 4.12 lista alguns valores das teclas virtuais, que são verificadas durante o processo da mensagem WM_KEYDOWN e WM_KEYUP. Note que todos os códigos das teclas virtuais começam com VK_*.

Valor	Tecla correspondente
VK_CANCEL	Control+break
VK_BACK	Backspace
VK_TAB	Tab
VK_RETURN	Enter
VK_ESCAPE	Esc
VK_SHIFT	Shift
VK_CONTROL	Control
VK_MENU	Alt
VK_PAUSE	Pause
VK_SPACE	Barra de espaço
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_END	End
VK_HOME	Home
VK_INSERT	Insert

VK_DELETE	Delete
VK_UP	Seta acima
VK_DOWN	Seta abaixo
VK_LEFT	Seta esquerda
VK_RIGHT	Seta direita
VK_SNAPSHOT	Print Screen
VK_CAPITAL	Caps Lock
VK_SCROLL	Scroll Lock
VK_NUMLOCK	Num Lock
VK_LWIN	"Windows" da esquerda
VK_RWIN	"Windows" da direita
VK_APPS	Tecla de Menu Pop-up
VK_NUMPAD0 ... VK_NUMPAD9	Teclado numérico 0 ... 9
VK_ADD	Sinal +
VK_SUBTRACT	Sinal -
VK_MULTIPLY	Sinal *
VK_DIVIDE	Sinal /
VK_F1 ... VK_F12	Teclas F1 … F12

Tabela 4.12: Alguns valores das teclas virtuais.

Nota: não há códigos de teclas virtuais para os números 0 - 9 e letras A - Z. Para verificá-las nas mensagens WM_KEYDOWN e WM_KEYUP, utilize os valores hexadecimais 30...39 para os números e 41...5A para as letras, ou 'A' - 'Z', '0' – '9' (letras maiúsculas).

Para exemplificar o processamento de cada uma dessas mensagens (Listagem 4.14), podemos mostrar na área cliente quais teclas o usuário pressionou, no seu formato ASCII (mensagem WM_CHAR) e mostrar uma mensagem quando o usuário pressionar alguma seta do teclado (mensagem WM_KEYDOWN):

```
case WM_CHAR: // Obtém o código ASCII da tecla pressionada
{
  char szTecla[1] = { 0 }; // Armazena tecla pressionada
  static int itx = 8; // Posição x na área cliente

  // Obtém identificador do DC
  hDC = GetDC(hWnd);

  // Armazena tecla pressionada em szTecla
  sprintf(szTecla, "%c", wParam);

  // Mostra tecla na área cliente
  TextOut(hDC, itx, 8, szTecla, strlen(szTecla));
```

```
  // Incrementa posição x
  itx += 16;

  // Libera DC
  ReleaseDC(hWnd, hDC);

  return(0);
} break;

case WM_KEYDOWN: // Obtém tecla virtual
{
  switch(wParam) // Verifica qual tecla virtual foi pressionada
  {
    case VK_LEFT: // Seta Esquerda
    {
      MessageBox(hWnd, "SETA PRA ESQUERDA PRESSIONADA!", "TECLADO", MB_OK);
    } break;
    case VK_RIGHT: // Seta Direita
    {
      MessageBox(hWnd, "SETA PRA DIREITA PRESSIONADA!", "TECLADO", MB_OK);
    } break;
    case VK_UP: // Seta Acima
    {
      MessageBox(hWnd, "SETA PRA CIMA PRESSIONADA!", "TECLADO", MB_OK);
    } break;
    case VK_DOWN: // Seta Abaixo
    {
      MessageBox(hWnd, "SETA PRA BAIXO PRESSIONADA!", "TECLADO", MB_OK);
    } break;
  }
  return(0);
} break;
```

Listagem 4.14: Processando as mensagens `WM_CHAR` e `WM_KEYDOWN`.

O código-fonte do programa-exemplo *prog04-1.cpp* é mostrado na Listagem 4.15:

```
//------------------------------------------------------------------------
// prog04-1.cpp - Processando mensagens do teclado e GDI
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// Bibliotecas
//------------------------------------------------------------------------
#include <windows.h>
#include <stdio.h>

//------------------------------------------------------------------------
// Definições
//------------------------------------------------------------------------
// Nome da classe da janela
#define WINDOW_CLASS         "prog04-1"
// Título da janela
#define WINDOW_TITLE         "Prog 04-1"
// Largura da janela
#define WINDOW_WIDTH         320
```

```
// Altura da janela
#define WINDOW_HEIGHT        240

//------------------------------------------------------------------------
// Protótipo das funções
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

//------------------------------------------------------------------------
// WinMain() -> Idêntica ao código do prog02.cpp
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
  // Variáveis para manipulação da parte gráfica do programa
  HDC hDC = NULL;
  PAINTSTRUCT psPaint;

  // Armazena qual tecla foi pressionada
  static char szMensagem[10] = { 0 };

  // Verifica qual foi a mensagem enviada
  switch(uMsg)
  {

    case WM_CREATE: // Janela foi criada
    {
      // Retorna 0, significando que a mensagem foi processada corretamente
      return(0);
    } break;

    case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
    {
      // Obtém identificador do DC e preenche PAINTSTRUCT
      hDC = BeginPaint(hWnd, &psPaint);

      // Imprime texto na parte superior da área cliente
      SetTextColor(hDC, RGB(128, 0, 128));
      TextOut(hDC, 8, 8, szMensagem, strlen(szMensagem));

      // Obtém área cliente
      RECT rcArea;
      GetClientRect(hWnd, &rcArea);

      // Imprime texto na área cliente
      SetTextColor(hDC, 0x0000FF00);
      DrawText(hDC, "Utilizando GDI", -1, &rcArea, DT_SINGLELINE | DT_CENTER
| DT_VCENTER);

      // Libera DC e valida área
      EndPaint(hWnd, &psPaint);

      return(0);
    } break;

    case WM_CLOSE: // Janela foi fechada
```

```
    {
      // Destrói a janela
      DestroyWindow(hWnd);

      return(0);
    } break;

    case WM_DESTROY: // Janela foi destruída
    {
      // Envia mensagem WM_QUIT para o loop de mensagens
      PostQuitMessage(0);

      return(0);
    } break;

    case WM_CHAR: // Obtém o código ASCII da tecla pressionada
    {
      // Obtém identificador do DC
      hDC = GetDC(hWnd);

      // Verifica a tecla pressionada e mostra no programa
      sprintf(szMensagem, "Tecla: %c  ", (char)wParam, lParam);
      TextOut(hDC, 8, 8, szMensagem, strlen(szMensagem));

      // Libera DC
      ReleaseDC(hWnd, hDC);

      return(0);
    } break;

    case WM_KEYDOWN: // Obtém tecla virtual
    {
      switch(wParam) // Verifica qual tecla virtual foi pressionada
      {
        case VK_LEFT: // Seta Esquerda
        {
          MessageBox(hWnd, "SETA PRA ESQUERDA PRESSIONADA!", "TECLADO",
MB_OK);
        } break;
        case VK_RIGHT: // Seta Direita
        {
          MessageBox(hWnd, "SETA PRA DIREITA PRESSIONADA!", "TECLADO",
MB_OK);
        } break;
        case VK_UP: // Seta Acima
        {
          MessageBox(hWnd, "SETA PRA CIMA PRESSIONADA!", "TECLADO", MB_OK);
        } break;
        case VK_DOWN: // Seta Abaixo
        {
          MessageBox(hWnd, "SETA PRA BAIXO PRESSIONADA!", "TECLADO", MB_OK);
        } break;
      }

      return(0);
    } break;

    default: // Outra mensagem
    {
```

```
      /* Deixa o Windows processar as mensagens que não foram verificadas na
função */
      return(DefWindowProc(hWnd, uMsg, wParam, lParam));
    }

  }
}
```

Listagem 4.15: Código-fonte do programa-exemplo *prog04-1.cpp*.

Outra forma de verificar o teclado

Uma maneira diferente de verificar o estado das teclas é através da função `GetAsyncKeyState()`. Essa função pode ser chamada em qualquer parte do programa, independente de estarmos processando uma mensagem de teclado ou não (embora não seja recomendado chamar essa função no processamento de mensagens de teclado); ela simplesmente retorna o estado de uma tecla (pressionada/não pressionada) no momento que for chamada.

```
SHORT GetAsyncKeyState(
  int vKey // código de tecla virtual
);
```

A função recebe o parâmetro `int vKey`, que é o código de tecla virtual, listados previamente na Tabela 4.12. Ela retorna um tipo `SHORT` indicando se a tecla (informada em `vKey`) está pressionada (bit mais significativo está ativado) ou não. Para verificarmos se uma tecla está pressionada, fazemos uma operação AND bit-a-bit do retorno da função com o valor hexadecimal 0x8000 (que verifica se o bit mais significativo está ativado).

Por exemplo, para verificar se a tecla ENTER foi pressionada, utilizamos o código da Listagem 4.16:

```
if(GetAsyncKeyState(VK_RETURN) & 0x8000)
  // Tecla pressionada
else
  // Tecla não pressionada
```

Listagem 4.16: Usando `GetAsyncKeyState()`.

Além dos códigos da Tabela 4.12, a função `GetAsyncKeyState()` aceita os códigos da Tabela 4.13, que diferenciam as teclas SHIFT, CONTROL e ALT da esquerda e direita e os botões do mouse.

Valor	Tecla correspondente
`VK_LSHIFT`	Shift da esquerda.

VK_RSHIFT	Shift da direita.
VK_LCONTROL	Control da esquerda.
VK_RCONTROL	Control da direita.
VK_LMENU	Alt da esquerda.
VK_RMENU	Alt da direita.
VK_LBUTTON	Botão esquerdo do mouse.
VK_MBUTTON	Botão do meio do mouse.
VK_RBUTTON	Botão direito do mouse.

Tabela 4.13: Códigos de teclas-virtuais extras para `GetAsyncKeyState()`.

Verificando o mouse

Além do teclado, outro dispositivo de entrada/controle é o mouse. O funcionamento de um mouse é mais simples que o de um teclado, pois há apenas a movimentação do mesmo e o uso de dois ou três botões, em geral. A Win32 API possui mensagens para processamento de movimentação do mouse e clique dos botões, conforme a Tabela 4.14:

Mensagem	Descrição
WM_MOUSEMOVE	Enviada quando o mouse sofre movimentação.
WM_LBUTTONDBLCLK	Enviada quando o usuário dá um duplo-clique no botão esquerdo do mouse. É necessário definir o membro `WNDCLASSEX.style` com o flag `CS_DBLCLKS`.
WM_LBUTTONDOWN	Enviada quando o usuário clica no botão esquerdo do mouse.
WM_LBUTTONUP	Enviada quando o usuário solta o botão esquerdo do mouse.
WM_MBUTTONDBLCLK	Enviada quando o usuário dá um duplo-clique no botão do meio do mouse. É necessário definir o membro `WNDCLASSEX.style` com o flag `CS_DBLCLKS`.
WM_MBUTTONDOWN	Enviada quando o usuário clica no botão do meio do mouse.
WM_MBUTTONUP	Enviada quando o usuário solta o botão do meio do mouse.
WM_RBUTTONDBLCLK	Enviada quando o usuário dá um duplo-clique no botão direito do mouse. É necessário definir o membro `WNDCLASSEX.style` com o flag

	CS_DBLCLKS.
WM_RBUTTONDOWN	Enviada quando o usuário clica no botão direito do mouse.
WM_RBUTTONUP	Enviada quando o usuário solta o botão direito do mouse.

Tabela 4.14: Mensagens do mouse.

Para trabalharmos com o mouse, devemos processar as mensagens (Tabela 4.14) de acordo com as ações do mouse que o programa responderá. Já exemplificamos anteriormente como desenhar um círculo no local onde o ponteiro do mouse está localizado (dentro da área cliente), quando o usuário clica o botão esquerdo do mesmo (veja a Listagem 4.2), mas vamos verificar outro exemplo.

O código da Listagem 4.17 processa a mensagem e `WM_MOUSEMOVE`, ou seja, sempre será processada quando o mouse mudar de posição. A mensagem `WM_RBUTTONDOWN` também é processada, armazenando o ponto inicial do cursor durante o traçado. Nesse exemplo, verificamos se o botão direito do mouse está pressionado e, se estiver, traçamos uma linha seguindo o ponteiro do mouse. Não se preocupe caso não entenda o uso de certas funções como `MoveToEx()` e `LineTo()`, pois iremos estudá-las no próximo capítulo.

```
// Armazena posição (x, y) do cursor do mouse
static int x = 0;
static int y = 0;

case WM_RBUTTONDOWN: // Botão direito do mouse pressionado
{
  // Obtém posição (x, y) atual do cursor do mouse
  x = LOWORD(lParam);
  y = HIWORD(lParam);

  return(0);
} break;

case WM_MOUSEMOVE: // Cursor do mouse modificado
{
  // Verifica se o botão direito está pressionado
  if(wParam == MK_RBUTTON)
  {
    // Obtém identificador do DC
    hDC = GetDC(hWnd);

    // Move cursor para posição (x, y) anterior do mouse
    MoveToEx(hDC, x, y, NULL);

    // Obtém posição (x, y) atual do cursor do mouse
    x = LOWORD(lParam);
    y = HIWORD(lParam);
```

```
    // Traça reta
    LineTo(hDC, x, y);

    // Libera DC
    ReleaseDC(hWnd, hDC);
  }

  return(0);
} break;
```

Listagem 4.17: Processando a mensagem `WM_MOUSEMOVE`.

No processo das mensagens do mouse, existem alguns valores pré-definidos (Tabela 4.15) que indicam o estado de seus botões (pressionado ou não), fazendo a verificação em `wParam`. No exemplo da Listagem 4.17, comparamos `wParam` com `MK_RBUTTON`, para verificarmos se o botão direito do mouse está pressionado.

Valor	Descrição
`MK_LBUTTON`	Botão esquerdo do mouse está pressionado.
`MK_MBUTTON`	Botão do meio do mouse está pressionado.
`MK_RBUTTON`	Botão direito do mouse está pressionado.

Tabela 4.15: Estado dos botões do mouse.

Veja o arquivo *prog04-2.cpp* no CD-ROM, o qual contém o código-fonte com o programa-exemplo de processamento de mensagens do mouse, ilustrado na Figura 4.6.

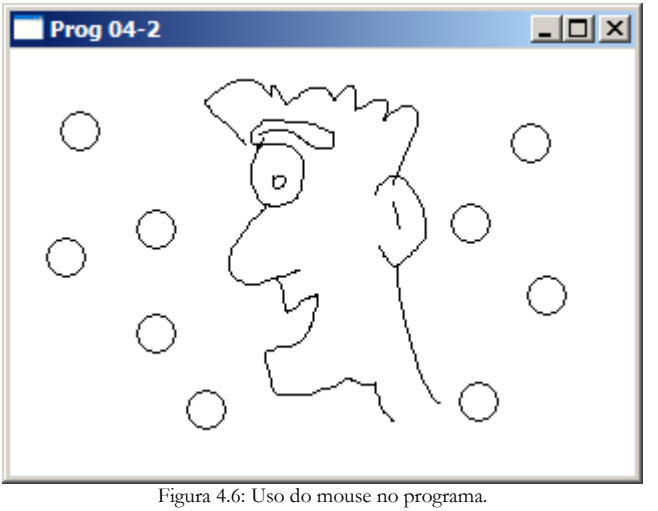

Figura 4.6: Uso do mouse no programa.

Verificando o mouse, II

Também podemos utilizar a função `GetAsyncKeyState()` para obtermos o estado dos botões do mouse quando não estamos processando suas mensagens (veja valores dos botões do mouse na Tabela 4.12).

Para sabermos a posição do cursor do mouse a qualquer momento, utilizamos a função `GetCursorPos()`:

```
BOOL GetCursorPos(
  LPPOINT lpPoint // posição do cursor
);
```

Essa função recebe um parâmetro `LPPOINT`, que é um ponteiro para um tipo de dado definido pelo Windows, `POINT`. O retorno da função é zero se falhar ou diferente de zero caso contrário.

O tipo de dado `POINT` tem a seguinte estrutura:

```
typedef struct tagPOINT {
  LONG x; // coordenada x do ponto
  LONG y; // coordenada y do ponto
} POINT, *PPOINT;
```

Podemos, ainda, modificar a posição do cursor do mouse, através da função `SetCursorPos()`.

```
BOOL SetCursorPos(
  int X, // nova posição x do cursor
  int Y // nova posição y do cursor
);
```

A função `SetCursorPos()` recebe como parâmetros a nova coordenada (`X`, `Y`) da posição do cursor do mouse. O retorno da função é zero se falhar ou diferente de zero caso contrário.

A Listagem 4.18 mostra como obter a posição do cursor do mouse em qualquer momento, armazenando-a numa variável do tipo `POINT`, e em seguida, modificar a posição atual do cursor.

```
// Armazena a posição do cursor do mouse
POINT ptMousePos;

// Obtém posição do cursor
GetCursorPos(&ptMousePos);
```

```
// Modifica a posição do cursor do mouse
ptMousePos.x += 10;
ptMousePos.y -= 25;
SetCursorPos(ptMousePos.x, ptMousePos.y);
```

Listagem 4.18: Utilizando `GetCursorPos()` e `SetCursorPos()`.

Com o término desse capítulo, você já está apto a fazer a interação do usuário com o seu programa, através do teclado e do mouse. Além disso, você aprendeu a mostrar textos na área cliente do programa. O próximo passo é verificar como desenhar pontos, retas e outros tipos de gráficos no seu programa.

Capítulo 5 – Gráficos com GDI

No capítulo anterior, fizemos uma introdução à *Graphics Device Interface* e seus objetos gráficos. Agora, vamos aprender como trabalhar com canetas e pincéis e como desenhar gráficos vetoriais.

Nota: o Windows trabalha com dois tipos de gráficos: vetoriais e mapas de bits. Gráficos vetoriais são formados por retas e curvas (que nada mais são do que aproximações de retas) que podem ser facilmente modificadas através de equações matemáticas. Gráficos do tipo mapa de bits (ou bitmap, também conhecido por *raster bitmap*) são representados, como o nome diz, por um mapa de bits, onde cada bit representa uma cor no mapa. A manipulação desse tipo de gráfico é um pouco mais complicada, pois é preciso manipulá-lo ponto-a-ponto. Programas que trabalham com gráficos vetoriais tendem a ser mais rápidos na manipulação dos mesmos do que programas que trabalham com gráficos bitmap, porém, bitmaps possuem mais detalhes e maior nível de realismo (como fotos, por exemplo).

Um simples ponto

Em computação gráfica, o *pixel* (ponto) é a base da criação de todos os gráficos e o menor gráfico que você pode manipular. Retas e curvas são formadas por diversos pixels, assim como uma foto no computador é composta por milhares de pixels, cada um com uma certa tonalidade de cor.

Podemos desenhar pontos na área cliente de um programa com duas funções: `SetPixel()` e `SetPixelV()`.

```
COLORREF SetPixel(
  HDC hdc, // identificador do DC
  int X, // coordenada x do pixel
  int Y, // coordenada y do pixel
  COLORREF crColor // cor do pixel
);
```

```
BOOL SetPixelV(
  HDC hdc, // identificador do DC
  int X, // coordenada x do pixel
  int Y, // coordenada y do pixel
  COLORREF crColor // cor do pixel
);
```

Ambas funções desenham um ponto de cor `crColor` na coordenada (`X`, `Y`) da área cliente, especificado pelos parâmetros. A diferença entre essas funções é o tipo de retorno de cada uma.

A função `SetPixel()` retorna o valor da cor RGB que a função desenhou o ponto. O valor retornado pode ser diferente do especificado no parâmetro `crColor`, pois nem sempre o sistema está configurado para exibir todas as combinações de cores (por exemplo, um sistema configurado com exibição de apenas 256 cores). A função retorna –1 em caso de falha (ocorre quando as coordenadas estão fora de alcance da área cliente).

Já a função `SetPixelV()` retorna zero se falhar ou um valor diferente de zero caso contrário. Assim como `SetPixel()`, a função faz uma aproximação da cor especificada em `crColor`. Por não retornar o valor da cor RGB que o ponto foi desenhado, essa função é mais rápida que `SetPixel()`.

A Figura 5.1 demonstra a execução do programa-exemplo *prog05-1.cpp*, que encontra-se no CD-ROM.

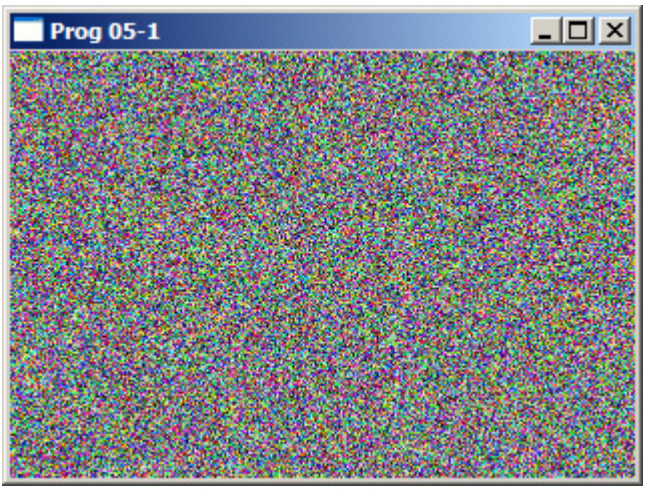

Figura 5.1: Desenhando pontos.

Dica: assim como existem funções do tipo `SetAlgumaCoisa()` (para definir "AlgumaCoisa"), podem existir equivalentes `GetAlgumaCoisa()`, que obtêm informações de "AlgumaCoisa". No caso da função `SetPixel()`, há o seu equivalente `GetPixel()`, que retorna a cor RGB de determinada coordenada (x, y) da área cliente.

Canetas e pincéis

Alguns termos utilizados na programação gráfica do Windows podem ser facilmente relacionados com a vida real. Assim, veremos como funciona o uso de canetas e pincéis GDI com exemplos do dia-a-dia.

Um dos principais usos da caneta é a caligrafia, mas também podemos usar canetas (e lápis) para desenhar, ou seja, fazer o contorno de uma figura. Existem diversos tipos e modelos de canetas – algumas têm cor da tinta azul, preta, vermelha, entre outras cores; Há canetas com ponta fina e outras com ponta grossa. Quando desenhamos algo, podemos tracejar linhas contínuas, espaçadas, pontilhadas e de diversas outras maneiras. É exatamente para isso que serve uma caneta na GDI: definir propriedades (cor, espessura, tipo de linha) para ser aplicado em retas, curvas e contornos de figuras.

Numa pintura à óleo, depois que o pintor faz o esboço (traçado) da sua obra na tela, ele começa a trabalhar com pincéis e tintas (e outras ferramentas que não vem ao caso) para preencher os espaços vazios entre os traçados. No caso da GDI, os pincéis também são utilizados para preenchimento (podemos pensar nos pincéis como a ferramenta "balde de tinta" encontrada em *softwares* de edição gráfica), podendo ter cores sólidas, preenchimento de linhas ou de texturas.

Criando canetas

Para criarmos uma caneta, usamos a função `CreatePen()`, definindo suas propriedades.

```
HPEN CreatePen(
  int fnPenStyle, // estilo da caneta
  int nWidth, // espessura da caneta
  COLORREF crColor // cor da caneta
);
```

A função recebe como parâmetros um estilo de caneta, `int fnPenStyle`, que pode ser um dos valores da Tabela 5.1, a espessura (`int nWidth`) e sua cor (`COLORREF crColor`). O retorno da função é o identificador da nova caneta ou `NULL` no caso de erro.

Valor	Descrição
`PS_SOLID`	Caneta sólida.

PS_DASH	Caneta tracejada.
PS_DOT	Caneta pontilhada.
PS_DASHDOT	Caneta alternando traço e ponto.
PS_DASHDOTDOT	Caneta alternando traço e pontos duplos.
PS_NULL	Caneta invisível.

Tabela 5.1: Estilos de caneta.

Se nWidth receber zero, a espessura da caneta será de um pixel. Apenas canetas com estilo PS_SOLID podem ter a espessura maior que um pixel. Se a caneta for criada com os estilos PS_DASH, PS_DOT, PS_DASHDOT ou PS_DASHDOTDOT e nWidth for maior que um, a função irá criar uma caneta do tipo PS_SOLID com a espessura indicada.

A Listagem 5.1 exemplifica o procedimento para criar e usar uma caneta dentro da mensagem WM_PAINT.

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Cria e seleciona nova caneta no DC e salva caneta antiga
  HPEN hPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
  HPEN hPenOld = (HPEN)SelectObject(hDC, hPen);

  //
  // Código para desenhar gráficos
  //

  // Restaura caneta antiga e deleta nova caneta
  SelectObject(hDC, hPenOld);
  DeleteObject(hPen);

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.1: Criando e utilizando uma caneta.

Nota: para utilizarmos canetas e qualquer outro objeto GDI, devemos selecioná-los no DC através da função SelectObject(), conforme explicado no capítulo 4.

Criando pincéis

Podemos criar e utilizar quatro tipos de pincéis: preenchimento com cores sólidas, com linhas, com texturas e pincéis definidos pelo sistema. Para cada tipo, existem funções específicas para criar pincéis, respectivamente: `CreateSolidBrush()`, `CreateHatchBrush()`, `CreatePatternBrush()` e `GetStockObject()`.

```
HBRUSH CreateSolidBrush(
  COLORREF crColor // cor do pincel
);
```

A função `CreateSolidBrush()` recebe o parâmetro `COLORREF crColor`, um valor enviado com o uso da macro `RGB()`, determinando a cor do pincel que será criado. Retorna o identificador do novo pincel ou `NULL` no caso de erro.

```
HBRUSH CreateHatchBrush(
  int fnStyle, // estilo de linha
  COLORREF clrref // cor de fundo
);
```

O primeiro parâmetro que `CreateHatchBrush()` recebe indica o estilo de preenchimento de linhas, conforme a Tabela 5.2. O segundo parâmetro indica a cor das linhas de preenchimento. A função retorna o identificador do novo pincel ou `NULL` no caso de erro.

Valor	Descrição
`HS_BDIAGONAL`	Linhas diagonais (da esquerda para direita e para cima).
`HS_CROSS`	Linhas horizontais e verticais.
`HS_DIAGCROSS`	Linhas diagonais cruzadas.
`HS_FDIAGONAL`	Linhas diagonais (da esquerda para direita e para baixo).
`HS_HORIZONTAL`	Linhas horizontais.
`HS_VERTICAL`	Linhas verticais.

Tabela 5.2: Estilos de preenchimento de linhas.

```
HBRUSH CreatePatternBrush(
  HBITMAP hbmp // identificador do bitmap
);
```

O parâmetro `hbmp` da função `CreatePatternBrush()` recebe o identificador de uma imagem bitmap. Estudaremos o uso de bitmaps no

próximo capítulo. A função retorna o identificador do novo pincel ou `NULL` no caso de erro.

Já discutimos o uso de pincéis definidos pelo sistema no capítulo 2, durante a definição da classe da janela de um programa. Usamos a função `GetStockObject()`, que recebe valores da Tabela 2.4.

Para criação e uso de pincéis, seguimos o mesmo processo das canetas: criamos um pincel, com sua respectiva função, salvando-o numa variável do tipo `HBRUSH`. Em seguida, selecionamos o novo pincel no DC e, ao mesmo tempo, salvamos o pincel antigo que estava selecionado no DC, com a função `SelectObject()`. Depois de utilizado, restauramos o pincel antigo e deletamos o novo. A Listagem 5.2 exemplifica esse processo, criando um pincel com linhas horizontais.

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Cria e seleciona novo pincel no DC e salva pincel antigo
  HBRUSH hBrush = CreateHatchBrush(HS_HORIZONTAL, RGB(255, 255, 128));
  HBRUSH hBrushOld = (HBRUSH)SelectObject(hDC, hBrush);

  //
  // Código para desenhar gráficos
  //

  // Restaura pincel antigo e deleta novo pincel
  SelectObject(hDC, hBrushOld);
  DeleteObject(hBrush);

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.2: Criando e utilizando um pincel.

Combinação de cores (*mix mode*)

A GDI utiliza o *mix mode* (modo de combinação de cores) para determinar como a cor de uma caneta ou pincel será aplicada/misturada em cima das cores já existentes na área cliente, conforme a Tabela 5.3.

Valor	Descrição
`R2_BLACK`	Cor sempre preta.

R2_COPYPEN	Cor da caneta / pincel (padrão).
R2_MASKNOTPEN	Combinação das cores comuns da área cliente e da cor inversa da caneta / pincel.
R2_MASKPEN	Combinação das cores comuns da área cliente e da caneta / pincel.
R2_MASKPENNOT	Combinação das cores comuns da cor inversa da área cliente e da caneta / pincel.
R2_MERGENOTPEN	Combinação da cor da área cliente e da cor inversa da caneta / pincel.
R2_MERGEPEN	Combinação da cor da área cliente e da cor da caneta / pincel.
R2_MERGEPENNOT	Combinação da cor inversa da área cliente e da cor da caneta / pincel.
R2_NOP	Cor igual da área cliente (nada é desenhado).
R2_NOT	Cor inversa da área cliente.
R2_NOTCOPYPEN	Cor inversa da caneta / pincel.
R2_NOTMASKPEN	Cor inversa de R2_MASKPEN.
R2_NOTMERGEPEN	Cor inversa de R2_MERGEPEN.
R2_NOTXORPEN	Cor inversa de R2_XORPEN.
R2_WHITE	Cor sempre branca.
R2_XORPEN	Combinação das cores na área cliente e na caneta / pincel, mas não em ambas.

Tabela 5.3: *Mix mode*.

Para configurar o *mix mode*, utilizamos a função `SetROP2()`.

```
int SetROP2(
  HDC hdc, // identificador do DC
  int fnDrawMode // mix mode
);
```

O primeiro parâmetro da função é o identificador do DC utilizado. O segundo parâmetro pode receber um dos valores da Tabela 5.3, especificando o *mix mode*. A função retorna o valor do *mix mode* anterior (Tabela 5.3) ou zero no caso de erro.

O *mix mode* padrão da GDI é `R2_COPYPEN`, ou seja, não importa a cor da área cliente, os pontos das linhas sempre serão desenhadas com as propriedades da caneta atual e o preenchimento será feito com as propriedades do pincel atual. Se definirmos o *mix mode* como `R2_XORPEN`, tudo que for desenhado na área cliente terá sua cor (da caneta e pincel) invertida, dando

aquele efeito de "negativo da imagem". Com o *mix mode* definido como `R2_NOT`, uma linha será desenhada com a cor inversa da área cliente. Caso a mesma linha seja desenhada novamente, a mesma será apagada, pois a cor da área cliente voltará ao seu original.

Nota: o valor `R2_NOT` funciona como o operador `!` (*not*) da linguagem C, como no exemplo abaixo:

```
bool teste = true; // teste é verdadeiro
teste = !teste; // teste fica falso
teste = !teste; // teste volta a ser verdadeiro
```

Traçando linhas retas

Antes de desenharmos uma linha reta, devemos indicar a coordenada (x, y) inicial da mesma. A GDI armazena um "cursor invisível" indicando essa coordenada. Para modificá-la, usamos a função `MoveToEx()`:

```
BOOL MoveToEx(
  HDC hdc, // identificador do DC
  int X, // nova coordenada x
  int Y, // nova coordenada y
  LPPOINT lpPoint // coordenada anterior
);
```

Os três primeiros parâmetros são auto-explicativos. O último, `LPPOINT lpPoint`, pode receber um ponteiro para uma estrutura `POINT`, na qual será armazenada a coordenada anterior à chamada da função. Se esse parâmetro receber `NULL`, então a coordenada anterior não será armazenada. A função retorna um valor diferente de zero quando bem sucedida ou zero se falhar.

Nota: todas as funções de desenho sempre recebem como primeiro parâmetro o identificador do DC (`HDC hdc`) que estamos utilizando.

Feito isso, o próximo passo é indicar a coordenada (x, y) final da reta que iremos traçar, através da função `LineTo()`. Quanto à cor e espessura da reta, são utilizadas as propriedades da caneta selecionada no DC.

```
BOOL LineTo(
  HDC hdc, // identificador do DC
  int nXEnd, // coordenada x do fim da reta
  int nYEnd // coordenada y do fim da reta
);
```

Novamente, os parâmetros da função são auto-explicativos. A função retorna um valor diferente de zero quando bem sucedida ou zero se falhar.

Nota: `LineTo()` traça uma reta da posição atual do "cursor invisível" até a coordenada (x, y) informada na função, porém, esse ponto (x, y) não é incluso no traçado da reta.

Depois de executada, a função `LineTo()` atualiza a posição do "cursor invisível" para a coordenada (x, y) final da reta. Assim, se quisermos traçar uma reta ligada à que acabamos de traçar, podemos chamar novamente a função `LineTo()`, sem a necessidade de definir a coordenada inicial com `MoveToEx()`. A Listagem 5.3 mostra como traçar quatro linhas ligadas entre si, formando um losango:

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Caneta já foi criada e selecionada no DC

  // Move "cursor invisível" para (50, 50)
  MoveToEx(hDC, 50, 50, NULL);
  // Desenha quatro retas, formando um losango
  LineTo(hDC, 20, 70);
  LineTo(hDC, 50, 90);
  LineTo(hDC, 80, 70);
  LineTo(hDC, 50, 50);

  // Restaura caneta antiga e deleta nova caneta

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.3: Uso de `MoveToEx()` e `LineTo()`.

Ao invés de fazermos quatro chamadas à função `LineTo()` para desenhar o losango, podemos criar um vetor do tipo `POINT`, onde serão armazenados os pontos do losango, e traçar as retas com o uso da função `PolylineTo()`.

```
BOOL PolylineTo(
  HDC hdc, // identificador do DC
  CONST POINT *lppt, // vetor de pontos
  DWORD cCount // número de pontos no vetor
);
```

A função recebe o identificador do DC como primeiro parâmetro. Em `CONST POINT *lppt`, passamos o vetor onde estão armazenados os pontos das retas e informamos a quantidade de pontos no parâmetro `DWORD cCount`. A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

`PolylineTo()`, assim como `LineTo()`, atualiza a posição do "cursor invisível" para a coordenada (x, y) do último ponto do vetor passado em `*lppt`.

A Listagem 5.4 é uma modificação da Listagem 5.3; dessa vez, utilizamos `PolylineTo()` ao invés de `LineTo()`.

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Caneta já foi criada e selecionada no DC

  // Define pontos do losango
  POINT ptLosango[4];
  ptLosango[0].x = 20;
  ptLosango[0].y = 70;
  ptLosango[1].x = 50;
  ptLosango[1].y = 90;
  ptLosango[2].x = 80;
  ptLosango[2].y = 70;
  ptLosango[3].x = 50;
  ptLosango[3].y = 50;

  // Move "cursor invisível" para (50, 50)
  MoveToEx(hDC, 50, 50, NULL);

  // Desenha quatro retas formadas pelos pontos do vetor ptLosango[4],
  // formando o losango
  PolylineTo(hDC, ptLosango, 4);

  // Restaura caneta antiga e deleta nova caneta

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.4: Uso de `PolylineTo()`.

Nota: há também a função `Polyline()`, que têm a mesma função que `PolylineTo()`, com a única diferença que ela não utiliza nem atualiza o "cursor invisível". `Polyline()` ignora a coordenada (x, y) informada em `MoveToEx()` e usa o primeiro ponto do vetor do tipo `POINT` como primeira

coordenada. Verifique a diferença entre essas funções, chamando `Polyline()` no lugar de `PolylineTo()` no exemplo da Listagem 5.4.

A Figura 5.2 demonstra a execução do programa-exemplo *prog05-2.cpp*, que encontra-se no CD-ROM.

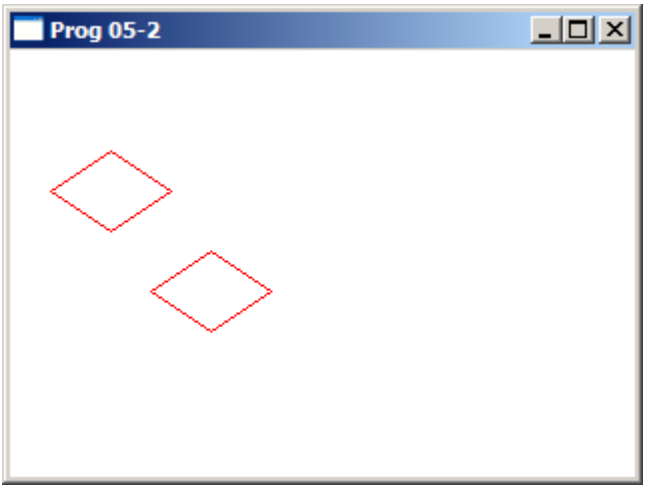

Figura 5.2: Losangos com linhas retas.

Traçando linhas curvas

Além de linhas retas, podemos traçar linhas curvas, que podem ser curvas parciais de uma elipse ou de forma livre (curvas de *Bézier*). Vamos trabalhar inicialmente com arcos.

Quando vamos traçar um arco, a GDI, por padrão, desenha o arco em sentido anti-horário, como no sistema cartesiano. Podemos modificar o sentido com a função `SetArcDirection()`:

```
int SetArcDirection(
  HDC hdc, // identificador do DC
  int ArcDirection // direção do arco
);
```

A função recebe o identificador do DC no primeiro parâmetro. O segundo parâmetro define o sentido do arco e pode receber os valores `AD_COUNTERCLOCKWISE` (sentido anti-horário) ou `AD_CLOCKWISE` (sentido horário). A

função retorna o sentido que estava definido anteriormente ou zero se ocorrer erro.

Para traçarmos um arco, chamamos a função `Arc()`:

```
BOOL Arc(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo do retângulo
  int nTopRect, // coordenada y do canto superior esquerdo do retângulo
  int nRightRect, // coordenada x do canto inferior direito do retângulo
  int nBottomRect, // coordenada y do canto inferior direito do retângulo
  int nXStartArc, // coordenada x do ponto inicial do arco
  int nYStartArc, // coordenada y do ponto inicial do arco
  int nXEndArc, // coordenada x do ponto final do arco
  int nYEndArc // coordenada y do ponto final do arco
);
```

O primeiro parâmetro da função recebe o identificador do DC. Os próximos quatro parâmetros indicam o retângulo (invisível) formado pelas extremidades da elipse. Os parâmetros `nXStartArc` e `nYStartArc` indicam a coordenada do ponto inicial do arco; `nXEndArc` e `nYEndArc` indicam a coordenada do ponto final do arco. Essas duas coordenadas formam retas (também invisíveis) com o centro da elipse, determinando o arco que será traçado. A função retorna um valor diferente de zero quando o arco é desenhado, ou zero quando o arco não é desenhado.

Dica: caso as coordenadas do ponto inicial e final sejam iguais, uma elipse completa será desenhada.

A Figura 5.3 mostra como funciona a função `Arc()`, com a direção no sentido anti-horário.

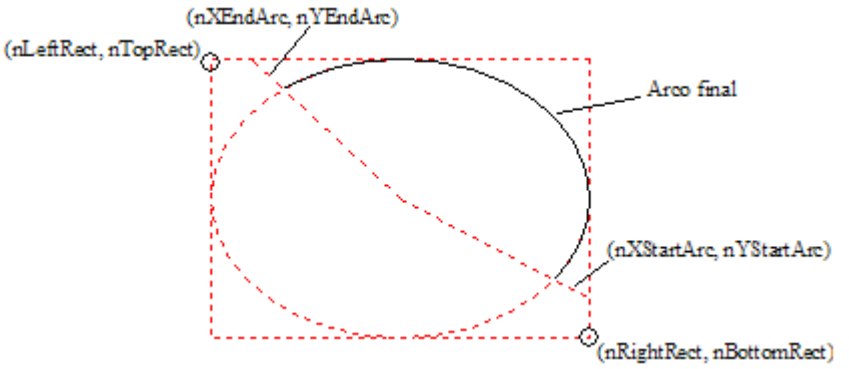

Figura 5.3: Como funciona a função `Arc()`.

Nota: há também a função `ArcTo()` para desenhar arcos, porém, ela utiliza e atualiza o "cursor invisível". Uma linha é traçada da posição atual do "cursor invisível" até a coordenada do ponto inicial do arco quando utilizamos `ArcTo()`.

Há casos em que não queremos desenhar curvas como arcos, mas sim curvas com formato livre. Para isso, utilizamos curvas de *Bézier*, com a função `PolyBezier()`. As curvas de *Bézier* são criadas com quatro pontos: os pontos das extremidades da curva e dois pontos de controle, que determinam a forma da curva. A matemática por trás do cálculo das curvas de *Bézier* está fora do alcance desse livro, portanto, veremos apenas como funciona a função `PolyBezier()`.

Dica: esse tipo de curva foi criado por um engenheiro francês, *Pierre Bézier*, que as utilizou para criar o design de um carro da Renault nos anos 70.

```
BOOL PolyBezier(
  HDC hdc, // identificador do DC
  CONST POINT* lppt, // pontos das extremidades e de controle
  DWORD cPoints // quantidade de pontos
);
```

A função recebe o identificador do DC (`HDC hdc`), um ponteiro para um vetor de pontos (`CONST POINT* lppt`) da curva e a quantidade de pontos do vetor (`DWORD cPoints`). A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

A quantidade de pontos passada no parâmetro `cPoints` deve ser conforme a equação `QP = (3 * QC) + 1`, onde `QP` = quantidade de pontos e `QC` = quantidade de curvas. Isso se deve porque, para cada curva, são necessários dois pontos de controle e um da extremidade final (a extremidade inicial de uma curva é o mesmo ponto da extremidade final da curva anterior). O "`+ 1`" da equação é para a extremidade inicial da primeira curva.

Sabendo-se disso, para cada curva extra, precisaremos de três pontos, pois a extremidade inicial é obtida da curva anterior. Se traçarmos apenas uma curva, serão necessários quatro pontos.

A Listagem 5.5 exemplifica a criação de uma única curva de *Bézier* e de três curvas de *Bézier* consecutivas. As curvas criadas com esse trecho de código

são mostradas na Figura 5.4, com seus respectivos pontos de controle e extremidades, ligados com retas pontilhadas.

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Caneta já foi criada e selecionada no DC

  // Define extremidades e pontos de controle da curva
  POINT ptUmaCurva[4];
  ptUmaCurva[0].x = 290; ptUmaCurva[0].y = 5;   // extremidade
  ptUmaCurva[1].x = 70 ; ptUmaCurva[1].y = 10;  // pt controle
  ptUmaCurva[2].x = 190; ptUmaCurva[2].y = 100; // pt controle
  ptUmaCurva[3].x = 200; ptUmaCurva[3].y = 100; // extremidade

  // Desenha a curva
  PolyBezier(hDC, ptUmaCurva, 4);

  // Define extremidades e pontos de controle das 3 curvas
  POINT ptTresCurvas[10];
  ptTresCurvas[0].x = 10 ; ptTresCurvas[0].y = 100; // extremidade
  ptTresCurvas[1].x = 40 ; ptTresCurvas[1].y = 30;  // pt controle
  ptTresCurvas[2].x = 50 ; ptTresCurvas[2].y = 130; // pt controle
  ptTresCurvas[3].x = 100; ptTresCurvas[3].y = 100; // extremidade
  ptTresCurvas[4].x = 140; ptTresCurvas[4].y = 75;  // pt controle
  ptTresCurvas[5].x = 170; ptTresCurvas[5].y = 90;  // pt controle
  ptTresCurvas[6].x = 200; ptTresCurvas[6].y = 200; // extremidade
  ptTresCurvas[7].x = 220; ptTresCurvas[7].y = 180; // pt controle
  ptTresCurvas[8].x = 200; ptTresCurvas[8].y = 140; // pt controle
  ptTresCurvas[9].x = 240; ptTresCurvas[9].y = 100; // extremidade

  // Desenha as 3 curvas
  PolyBezier(hDC, ptTresCurvas, 10);

  // Restaura caneta antiga e deleta nova caneta

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.5: Criando curvas de *Bézier* com `PolyBezier()`.

Dica: para que duas curvas de *Bézier* consecutivas sejam interligadas sem uma mudança brusca de angulação, certifique-se que o segundo ponto de controle da curva anterior, sua extremidade final e o primeiro ponto de controle da curva atual estejam em uma mesma linha (como entre a primeira e a segunda curva da Figura 4.4).

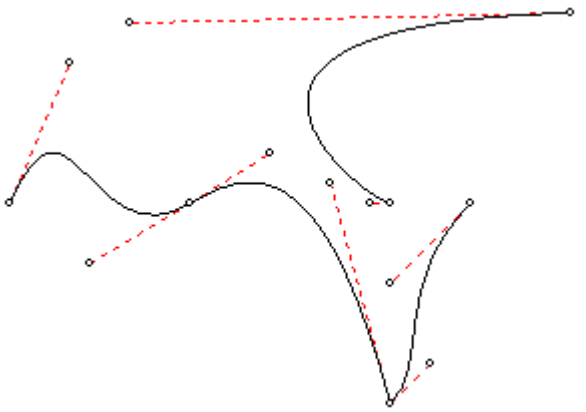

Figura 5.4: Curvas de *Bézier* da Listagem 4.4.

Veja o arquivo *prog05-3.cpp* no CD-ROM, o qual contém o código-fonte com o programa-exemplo para traçar arcos e curvas de *Bézier*.

Acabamos de aprender como traçar linhas retas e curvas, as quais utilizavam apenas as propriedades da caneta selecionada no DC. A partir de agora, veremos como desenhar figuras fechadas (retângulos, elipses, combinação de ambos e polígonos), que utilizam as propriedades do pincel selecionado no DC para preenchimento e da caneta para contorno da figura.

Desenhando retângulos

O primeiro tipo de figura fechada que veremos é o retângulo, que é criado utilizando a função `Rectangle()`.

```
BOOL Rectangle(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo
  int nTopRect, // coordenada y do canto superior esquerdo
  int nRightRect, // coordenada x do canto inferior direito
  int nBottomRect // coordenada y do canto inferior direito
);
```

O primeiro parâmetro da função recebe o identificador do DC. Os parâmetros `nLeftRect` e `nTopRect` indicam a coordenada (x, y) do canto superior esquerdo do retângulo. Os parâmetros `nRightRect` e `nBottomRect` indicam a coordenada (x, y) do cano inferior direito do retângulo. A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

A função `Rectangle()` desenha um retângulo com as seguintes dimensões: `altura = (nBottomRect - 1) - nTopRect` e `largura = (nRightRect - 1) - nLeftRect`. Se a caneta utilizada for do tipo `PS_NULL`, as dimensões do retângulo diminuirão em 1 pixel na altura e 1 pixel na largura.

Dica: todas as funções que desenham figuras fechadas preenchem o seu interior com as propriedades do pincel selecionado no DC. Caso queira o interior das figuras sem preenchimento (transparente), crie um pincel com `(HBRUSH)GetStockObject(NULL_BRUSH)`. Para descartar o contorno das figuras, crie uma caneta com o estilo `PS_NULL`.

Podemos também desenhar retângulos com seus cantos arredondados. A função `RoundRect()` realiza esse trabalho.

```
BOOL RoundRect(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo
  int nTopRect, // coordenada y do canto superior esquerdo
  int nRightRect, // coordenada x do canto inferior direito
  int nBottomRect, // coordenada y do canto inferior direito
  int nWidth, // largura da elipse
  int nHeight // altura da elipse
);
```

Os cinco primeiros parâmetros de `RoundRect()` são idênticos aos da função `Rectangle()`. Os dois últimos indicam a largura (`int nWidth`) e altura (`int nHeight`) das elipses que são utilizadas para arredondar os cantos do retângulo. A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

Além dessas duas funções, existe uma terceira, `FillRect()`, que é utilizada para preencher um retângulo com as propriedades de um pincel (não necessariamente o que está selecionado no DC). As dimensões do retângulo preenchido por `FillRect()` são definidas com as mesmas equações da função `Rectangle()`; porém, essa função não desenhar o contorno do retângulo, dispensando o uso de canetas.

```
int FillRect(
  HDC hDC, // identificador do DC
  CONST RECT *lprc // retângulo
  HBRUSH hbr // pincel
);
```

A função recebe o identificador do DC no primeiro parâmetro. O segundo parâmetro recebe um ponteiro para uma variável do tipo `RECT`, que contém as coordenadas do retângulo. O terceiro parâmetro indica qual pincel deve ser utilizado para o preenchimento do retângulo. A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

O uso das três funções de retângulo é demonstrado na Listagem 5.6.

```
case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
{
  // Obtém identificador do DC e preenche PAINTSTRUCT
  hDC = BeginPaint(hWnd, &psPaint);

  // Caneta já foi criada e selecionada no DC

  // Cria e seleciona novo pincel no DC e salva pincel antigo
  HBRUSH hBrush = CreateSolidBrush(RGB(255, 255, 0));
  HBRUSH hBrushOld = (HBRUSH)SelectObject(hDC, hBrush);

  // Cria pincel hBrush2
  HBRUSH hBrush2 = CreateHatchBrush(HS_DIAGCROSS, RGB(0, 0, 255));

  // Cria retângulo
  RECT rcRect = { WINDOW_WIDTH / 2 - 50 , WINDOW_HEIGHT / 2 - 50,
WINDOW_WIDTH / 2 + 50, WINDOW_HEIGHT / 2 + 50 };

  // Preenche retângulo rcRect com pincel hBrush2
  FillRect(hDC, &rcRect, hBrush2);

  // Desenha retângulo
  Rectangle(hDC, 20, 20, 120, 100);

  // Desenha retângulo com cantos arredondados
  RoundRect(hDC, 200, 150, 250, 200, 20, 20);

  // Deleta pincel hBrush2
  DeleteObject(hBrush2);
  // Restaura pincel antigo e deleta novo pincel
  SelectObject(hDC, hBrushOld);
  DeleteObject(hBrush);

  // Restaura caneta antiga e deleta nova caneta

  // Libera DC e valida área
  EndPaint(hWnd, &psPaint);

  return(0);
} break;
```

Listagem 5.6: Desenhando retângulos.

A saída da Listagem 5.6 (que é parte do código-fonte do arquivo *prog05-4.cpp*) é mostrada na Figura 5.5 abaixo.

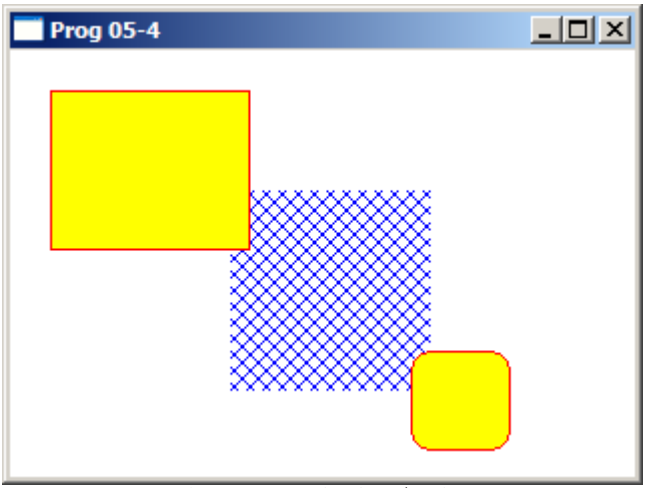

Figura 5.5: Desenhando retângulos.

Desenhando elipses

Outro tipo de figura fechada que a GDI pode desenhar são as elipses, através do uso da função `Ellipse()`. Quando aprendemos a desenhar arcos, vimos que para desenhá-los era necessário especificar um retângulo "invisível", formado pelas extremidades da elipse (Figura 5.3). Essa regra continua valendo para desenhar elipses, sendo que o centro da elipse é o centro do retângulo especificado na função.

```
BOOL Ellipse(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo do retângulo
  int nTopRect, // coordenada y do canto superior esquerdo do retângulo
  int nRightRect, // coordenada x do canto inferior direito do retângulo
  int nBottomRect // coordenada y do canto inferior direito do retângulo
);
```

O primeiro parâmetro da função recebe o identificador do DC. Os outros quatro parâmetros indicam o retângulo (invisível) formado pelas extremidades da elipse. A função retorna um valor diferente de zero quando bem sucedida ou zero se ocorrer algum erro.

É possível fazer combinações de arcos e cordas (linhas retas com extremos pertencentes à elipse), criando áreas delimitadas pelos mesmos, com a função `Chord()`. Nesse caso, a corda estaria cortando parte da elipse e descartando-a (veja a Figura 5.6).

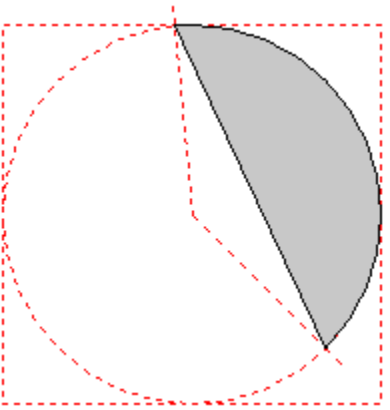

Figura 5.6: Combinação de arco e corda (área cinza).

```
BOOL Chord(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo do retângulo
  int nTopRect, // coordenada y do canto superior esquerdo do retângulo
  int nRightRect, // coordenada x do canto inferior direito do retângulo
  int nBottomRect, // coordenada y do canto inferior direito do retângulo
  int nXRadial1, // coordenada x do ponto inicial do arco
  int nYRadial1, // coordenada y do ponto inicial do arco
  int nXRadial2, // coordenada x do ponto final do arco
  int nYRadial2 // coordenada y do ponto final do arco
);
```

A função `Chord()` recebe os mesmo parâmetros que a função `Arc()` e possui os mesmos valores de retorno. A diferença entre elas é que `Chord()` adiciona a corda ao arco e preenche o interior da área delimitada pelos dois.

Uma variação dessa combinação de arcos e cordas é a "pizza", muito utilizada em gráficos. Nesse caso, são combinados o arco e duas retas que vão do centro da elipse até sua borda. A área delimitada é preenchida com o pincel selecionado no DC. A Figura 5.7 têm as mesmas propriedades da Figura 5.6, exceto que foi criada utilizando a função `Pie()` ao invés de `Chord()`.

```
BOOL Pie(
  HDC hdc, // identificador do DC
  int nLeftRect, // coordenada x do canto superior esquerdo do retângulo
  int nTopRect, // coordenada y do canto superior esquerdo do retângulo
  int nRightRect, // coordenada x do canto inferior direito do retângulo
  int nBottomRect, // coordenada y do canto inferior direito do retângulo
  int nXRadial1, // coordenada x do ponto inicial do arco
  int nYRadial1, // coordenada y do ponto inicial do arco
  int nXRadial2, // coordenada x do ponto final do arco
  int nYRadial2 // coordenada y do ponto final do arco
);
```

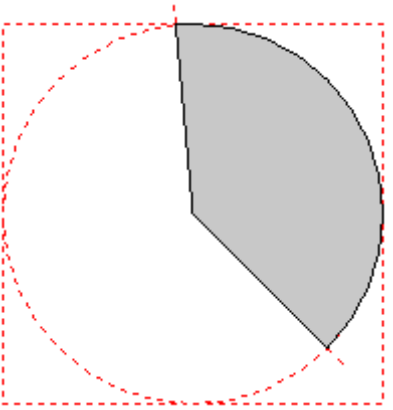

Figura 5.7: Uma "pizza".

Note que apenas o nome da função muda, em relação à `Chord()`. A diferença do resultado do uso de `Pie()` e `Chord()` pode ser comparado entre as Figuras 5.6 e 5.7.

Veja o arquivo *prog05-5.cpp* no CD-ROM, o qual contém o código-fonte com o programa-exemplo para traçar figuras discutidas nessa seção, inclusive uma "pizza" no formato do Pac-Man :-).

Desenhando polígonos

Para concluir a seção de figuras fechadas, vamos aprender como desenhar polígonos com o uso da função `Polygon()`. Polígonos são figuras formadas por um conjunto de pontos interligados com retas consecutivas (vértices).

```
BOOL Polygon(
  HDC hdc, // identificador do DC
  CONST POINT *lpPoints, // vértices do polígono
  int nCount // quantidade de vértices do polígono
);
```

A função recebe o identificador do DC no primeiro parâmetro. O segundo parâmetro, `CONST POINT *lpPoints`, recebe um ponteiro para um vetor do tipo `POINT`, que contém os pontos dos vértices do polígono. O último parâmetro recebe a quantidade de pontos que formam o polígono, que deve ser no mínimo dois pontos para criar-se um vértice (porém, para formar uma figura fechada, deve haver pelo menos três pontos). A função retorna um valor diferente de zero se não ocorrer erros, ou zero, caso contrário.

Nota: `Polygon()` traça uma reta do último vértice até o primeiro, fechando automaticamente o polígono.

Existem dois modos de preenchimento para os polígonos, quando esses possuem vértices que cruzam a área interna do polígono: modo alternado e completo. No modo alternado, o polígono começa a ser preenchido da borda até encontrar um vértice; desse vértice até o próximo, o polígono não é preenchido, no próximo vértice, ele é preenchido, e assim por diante. No modo completo, todas as áreas são preenchidas.

Os modos de preenchimento são definidos pela função `SetPolyFillMode()`:

```
int SetPolyFillMode(
  HDC hdc, // identificador do DC
  int iPolyFillMode // modo de preenchimento
);
```

A função recebe o identificador do DC no primeiro parâmetro. O segundo parâmetro pode receber os valores `ALTERNATE` (modo alternado – padrão da GDI) ou `WINDING` (modo completo). O retorno da função é o valor do modo de preenchimento anterior ou zero em caso de erro.

A Figura 5.8 mostra a execução do programa-exemplo *prog05-6.cpp* que encontra-se no CD-ROM. O programa cria um vetor `POINT` com cinco elementos, que definem os pontos de uma estrela, desenha o polígono com o preenchimento padrão (alternado) e em seguida modifica a posição do polígono e o modo de preenchimento para modo completo, para desenhá-lo novamente.

Inversão de cores e preenchimento de áreas

Existe uma função da GDI, `InvertRect()`, que faz a inversão das cores de determinado retângulo (o efeito "negativo" de fotos), realizando uma operação *not* para cada ponto do retângulo.

```
BOOL InvertRect(
  HDC hDC, // identificador do DC
  CONST RECT *lprc // retângulo
);
```

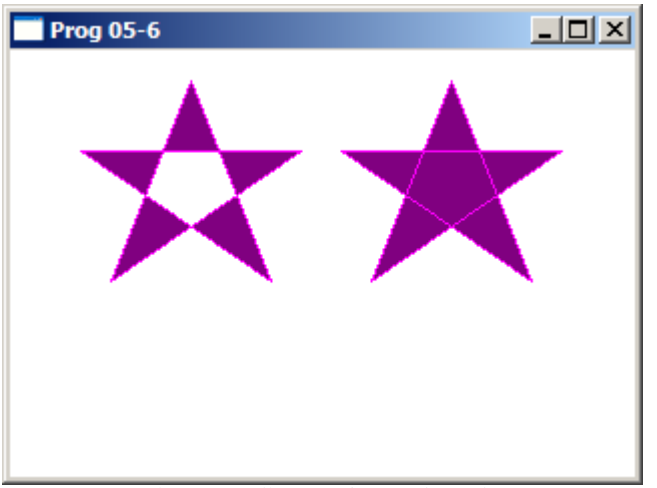

Figura 5.8: Polígono no formato de estrela.

O primeiro parâmetro da função é o identificador do DC. O segundo parâmetro recebe um ponteiro para uma variável `RECT`, que contém as coordenadas da área do retângulo que terá as cores invertidas. A função retorna um valor diferente de zero se não ocorrer erros, ou zero, caso contrário.

No tópico *"Desenhando retângulos"*, vimos que a função `FillRect()` é utilizada para o preenchimento de áreas retangulares. Porém, como podemos preencher áreas não-retangulares, como, por exemplo, áreas delimitadas por linhas retas e curvas de *Bézier*?

Para tal ação, utilizamos a função `ExtFloodFill()`, parecida com o "balde de tinta" de diversos *softwares* gráficos. A função pode fazer o preenchimento de duas maneiras: preenchendo áreas que são definidas por uma cor-limite (ou seja, a função continua preenchendo áreas até que seja encontrado um ponto, uma reta ou curva com a cor-limite) ou preenchendo áreas de uma mesma cor (deixando de preencher áreas de cores diferentes). Em ambos os casos, a função utiliza o pincel selecionado no DC.

```
BOOL ExtFloodFill(
  HDC hdc, // identificador do DC
  int nXStart, // coordenada x onde começa o preenchimento
  int nYStart, // coordenada y onde começa o preenchimento
  COLORREF crColor, // cor de preenchimento
  UINT fuFillType // tipo de preenchimento
);
```

A função recebe o identificador do DC no primeiro parâmetro. Os segundo e terceiro parâmetros (`nXStart` e `nYStart`, respectivamente) recebem a coordenada (x, y) de onde a função deve começar o preenchimento. O parâmetro `COLORREF crColor` depende do valor passado em `UINT fuFillType`: caso `fuFillType` receba `FLOODFILLBORDER`, a área a ser preenchida será limitada pela cor indicada em `crColor`. `fuFillType` também pode receber o valor `FLOODFILLSURFACE`; nesse caso, o preenchimento será feito apenas nas áreas que estiverem com a cor indica em `crColor`.

A função retorna um valor diferente de zero se não ocorrer erros, ou zero, caso contrário. A função também retorna zero nos seguintes casos:

- o ponto da coordenada (x, y) especificada é da mesma cor que a indicada em `crColor`, quando `fuFillType` recebe `FLOODFILLBORDER`;
- o ponto da coordenada (x, y) especificada não possui a mesma cor que a indicada em `crColor`, quando `fuFillType` recebe `FLOODFILLSURFACE`;
- o ponto da coordenada (x, y) especificada estiver fora da área cliente.

Veja o arquivo *prog05-7.cpp* no CD-ROM, o qual contém o código-fonte com o programa-exemplo para inverter cores de áreas retangulares e o uso da função `ExtFloodFill()` para preenchimento de áreas.

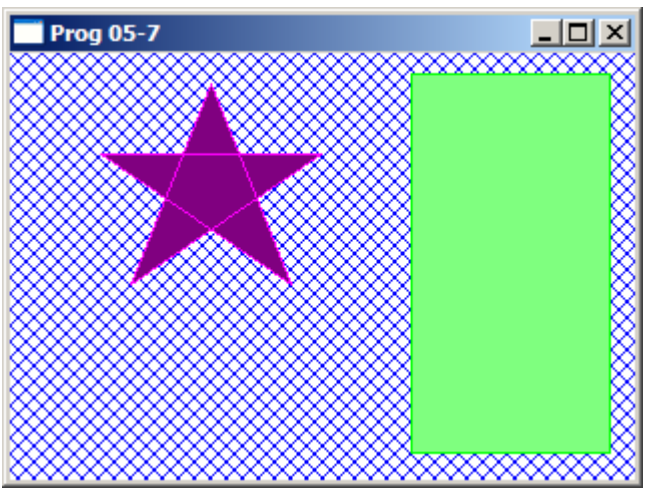

Figura 5.9: Invertendo cores e preenchendo áreas.

A Figura 5.9 mostra a execução do programa. O retângulo do programa teve sua cor original (roxa) invertida, tornando-se verde. Foi utilizado a função `ExtFloodFill()` para preencher áreas de cor branca a partir do ponto (100, 100). As áreas brancas foram preenchidas com um pincel azul e com linhas diagonais cruzadas.

Um simples programa de desenho

O arquivo *prog05-8.cpp* incluso no CD-ROM contém o código-fonte de um pequeno programa de desenho (se é que pode ser considerado assim), demonstrando como desenhar figuras através da interação do usuário (Figura 5.10).

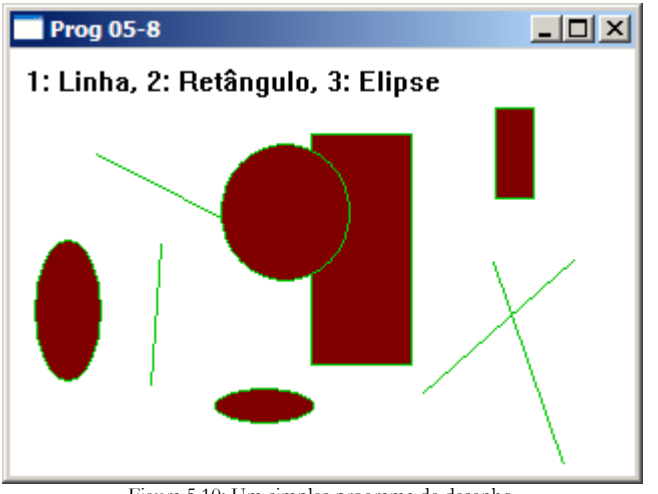

Figura 5.10: Um simples programa de desenho.

O programa é bem simples e desenha apenas retas, retângulos e elipses. Para torná-lo um verdadeiro programa de desenho, deveríamos dar suporte a mais figuras, opções para configurar as canetas e pincéis, criar uma boa interface para o usuário (o uso de menus seria uma boa idéia), traçar a sombra da figura que está sendo criada (para auxiliar o usuário na criação da mesma), entre outras implementações.

O "truque" do programa é processar as mensagens `WM_LBUTTONDOWN` e `WM_LBUTTONUP`. Quando o botão esquerdo do mouse é pressionado, armazenamos a coordenada (x, y) do cursor na variável `POINT ptMouseInicio`. O usuário arrasta o mouse até certo lugar, e para desenhar a figura, solta o botão

esquerdo do mouse. Processamos então a mensagem `WM_LBUTTONUP`, onde obtemos a coordenada (x, y) atual do cursor, sendo esta a coordenada final, armazenada na variável `POINT ptMouseFim`. Ainda na mensagem `WM_LBUTTONUP`, verificamos qual figura está selecionada para desenhar e chamamos sua respectiva função. Essa é a lógica utilizada para a criação de *softwares* gráficos.

Esse programa-exemplo tem um problema grave, pois ele não salva o conteúdo da área cliente, ou seja, se alguma janela sobrepor o programa, a área sobreposta será perdida. Isso acontece porque todos os desenhos são feitos na mensagem `WM_LBUTTONUP`, e não na `WM_PAINT` (que seria a maneira correta). Esse problema é eliminado utilizando um DC de memória, assunto do próximo capítulo, onde estudaremos o uso de bitmaps.

Capítulo 6 – Bitmaps

Entre os objetos GDI citados no capítulo 4, um deles foi `HBITMAP`, que é um objeto do tipo bitmap. Nesse capítulo, estaremos estudando como utilizar esse objeto, carregando e mostrando imagens armazenadas em arquivos **.bmp* nos nossos programas.

O que são bitmaps?

Podemos definir bitmaps como um vetor de (largura * altura) bits, onde uma certa quantidade de bits representa um ponto/cor que compõe a figura. Atualmente, existem dezenas de métodos para armazenar uma figura no computador, resultando em diferentes tipos de arquivos, como, por exemplo, imagens do tipo JPEG ou GIF, famosos pela utilização em páginas HTML.

Cada tipo de arquivo de imagem possui um cabeçalho, um método de armazenamento dos bits e compressão de dados, tornando-se inviável o estudo de todos os tipos de imagem nesse livro. Veremos como utilizar apenas o tipo nativo do Windows (bitmap, arquivos com extensão *.bmp* ou *.dib*).

Dica: você pode verificar a estrutura de outros tipos de arquivos de imagens consultando o livro *Encyclopedia of Graphics File Formats* (Murray, James D. e Vanryper, William – editora O'Reilly) ou pela Internet – existe um ótimo site especificamente sobre estrutura de arquivos em http://www.myfileformats.com.

Os bitmaps, quanto à quantidade de cores, podem ser classificados conforme a Tabela 6.1

Bitmap de n-bit	Quantidade máxima de cores
1	2 (2^1)
4	16 (2^4)
8	256 (2^8)
16	65.536 (2^{16})
24	16.777.216 (2^{24})
32	4.294.967.296 (2^{32})

Tabela 6.1: Classificação de bitmaps quanto à quantidade de cor.

O que significam todos esses números? O lado esquerdo da tabela indica a quantidade de bits que um bitmap utiliza para representar a cor de cada pixel, através da combinação de valores dos seus bits. O lado direito informa a quantidade máxima de cores que é possível obter através dessa combinação.

Bitmaps de até 8-bit necessitam de uma tabela de cores, também conhecida por "paleta de cores". Para um bitmap de 8-bit, visto que o mesmo pode ter até 256 cores diferentes, podemos imaginar um vetor do tipo `char` de 256 posições, onde cada posição no vetor equivale à uma cor. O vetor pode ser do tipo `char` pois um `char` ocupa 1 byte de memória (ou 8 bits), ou seja, cada pixel de um bitmap de 8-bit é representado por um byte. A Listagem 6.1 contém um misto de pseudo-código com linguagem C demonstrando como uma imagem de 8-bit poderia utilizar uma paleta de cores:

```
/*
 Cria Paleta de Cores com as seguintes configurações / índices:
 - cPaletaDeCores[0] = preto
 - cPaletaDeCores[1 – 84] = tons de vermelho
 - cPaletadeCores[85 – 170] = tons de verde
 - cPaletaDeCores[171 – 254] = tons de azul
 - cPaletaDeCores[255] = branco
*/

char cPaletaDeCores[256];
for(int i = 0; i < 256; i++)
  cPaletaDeCores[i] = i;

// Cria um emoticon branco com fundo preto
char cRosto[] = {
  0,  0 , 255, 255, 255,  0 , 0,
  0, 255, 255, 255, 255, 255, 0,
  0, 255,  0 , 255,  0 , 255, 0,
  0, 255, 255, 255, 255, 255, 0,
  0, 255, 255,  0 , 255, 255, 0,
  0, 255, 255, 255, 255, 255, 0,
  0, 255, 255, 255, 255, 255, 0,
  0, 255,  0 ,  0  , 0 , 255, 0,
  0, 255, 255, 255, 255, 255, 0,
  0,  0 , 255, 255, 255,  0,  0
};

// Plota os pixels do emoticon na tela
desenha_na_tela(cRosto);
```

Listagem 6.1: Pseudocódigo de paleta de cores.

Bitmaps de 16-bit (65 mil cores) ou mais não necessitam de uma paleta de cores. No caso de bitmaps de 16-bit, existem duas maneiras diferentes que os bits da imagem podem ser armazenados: 5-5-5 ou 5-6-5. O bitmap 16-bit 5-

5-5 é um bitmap cujos 5 primeiros bits (0 – 4) armazenam a intensidade da cor azul, os bits 5 - 9 armazenam a intensidade da cor verde e os bits 10 – 14 armazenam a intensidade da cor vermelha, "gastando" assim um bit que não é utilizado (bit 15). O bitmap 16-bit 5-6-5 é um bitmap que utiliza 6 bits para a cor verde (a escolha de um bit a mais para a cor verde é devido aos olhos humanos serem mais sensíveis à essa cor) e 5 para as cores azul e vermelha. Bitmaps de 24-bit utilizam 8 bits para armazenar a intensidade de cada cor (azul, verde, vermelha), enquanto bitmaps de 32-bit podem utilizar 8 bits para armazenar a intensidade de cada cor e os últimos 8 bits para o canal alfa (transparência).

Para representar a cor vermelha num bitmap de 16-bit (5-5-5), os 16 bits seriam: 0-11111-00000-00000 (ou 0x7C00 em hexadecimal). Num bitmap 16-bit (5-6-5), a cor vermelha seria representada da seguinte maneira: 11111-000000-00000 (ou 0xF800 em hexadecimal). A cor azul num bitmap 24-bit ficaria: 00000000-00000000-11111111 (ou 0x0000FF em hexadecimal). Veja que a notação das cores (24-bit) em hexadecimal é semelhante à utilizada na codificação de cores em HTML.

Bitmaps no Windows: DDB e DIB

Quando estamos utilizando bitmaps em nossos programas, é necessário o conhecimento de dois tipos de bitmap definidos pela Microsoft: o *device-dependent bitmap* (DDB) e o *device-independent bitmap* (DIB). Há certas diferenças entre essas definições e seus métodos de trabalho com imagens.

Um DDB é um objeto GDI (objeto `HBITMAP`), sendo possível selecionar o mesmo no DC que estamos utilizando. Isso significa que um DDB é compatível com o DC onde está selecionado, isto é, ele fica dependente das configurações do DC. Também podemos utilizar funções GDI para mostrar os bitmaps na tela, assim como fazemos com qualquer outro objeto GDI.

Porém, os DDB's têm suas limitações: não podemos ter acesso direto aos bits dos bitmaps (útil para manipulação e processamento de imagens) e há um limite de tamanho que um DDB pode suportar (16MB para Windows 95/98 e 48MB para Windows NT/2000). Para isso, a Microsoft criou os DIB's.

Conforme o nome diz, um DIB é um bitmap que independe de um dispositivo, não havendo restrições quanto à cores ou tamanho. Com o uso de um DIB, é possível também acessar diretamente os bits da imagem e manipulá-las, ao contrário de um DDB.

Diferente dos bitmaps DDB, um DIB não é um objeto GDI, mas sim, uma "descrição" de um bitmap; assim, não é possível selecionar um DIB no DC e não há funções que mostrem esse tipo de bitmap na tela. O que deve ser feito, nesse caso, é uma conversão DIB para DDB, usando funções GDI especiais. Devido à essa conversão, mostrar DIB's na tela é um processo mais lento que utilizar diretamente um bitmap compatível com o DC.

Nota: estudaremos o uso de bitmaps DDB, enquanto o DIB não será discutido. Ao invés, iremos ver um outro tipo de bitmap, *DIB Section*, um híbrido do DDB e DIB, o qual une vantagens de ambos (velocidade e objeto GDI de um DDB com o acesso direto aos bits e a não-restrição quanto ao tamanho de imagem de um DIB).

Carregando bitmaps

Em nossos programas, é possível utilizar imagens armazenadas em disco (arquivos **.bmp*) ou imagens de recursos (conforme visto no capítulo 3), armazenadas no final do arquivo executável. Carregamos bitmaps através da função `LoadImage()`.

```
HANDLE LoadImage(
  HINSTANCE hinst, // identificador da instância
  LPCTSTR lpszName, // imagem a ser carregada
  UINT uType, // tipo de imagem
  int cxDesired, // largura desejada
  int cyDesired, // altura desejada
  UINT fuLoad // opções de carregamento
);
```

Dica: essa função também pode ser utilizada para carregar ícones e cursores, pois ela tem o retorno do tipo `HANDLE` genérico (sendo necessário fazer type-casting para o tipo de imagem que estamos carregando).

O primeiro parâmetro da função `LoadImage()` recebe o identificador da instância do programa, se a imagem for carregada de um recurso, ou NULL quando for carregada de um arquivo em disco. Quando estamos carregando a

imagem de um recurso, o segundo parâmetro recebe o ID do mesmo, com o uso da macro `MAKEINTRESOURCE()`. No caso de uma imagem em disco, o segundo parâmetro recebe o local e nome de arquivo da imagem. O parâmetro `UINT uType` indica o tipo de imagem que estamos carregando: `IMAGE_BITMAP` (bitmap), `IMAGE_CURSOR` (cursor) ou `IMAGE_ICON` (ícone).

Os parâmetros `cxDesired` e `cyDesired` indicam o tamanho (largura e altura) da imagem que está sendo carregada. No caso de imagem de recurso, podemos passar zero para ambos os parâmetros; assim, a função automaticamente obtém a largura e altura da imagem a ser carregada.

O último parâmetro recebe opções para carregar a imagem. O valor padrão desse parâmetro é `LR_DEFAULTCOLOR`. Quando estamos carregando uma imagem do disco, passamos o valor `LR_LOADFROMFILE` para esse parâmetro. O valor `LR_CREATEDIBSECTION` é passado no último parâmetro quando queremos que a função retorne um DIB Section ao invés de um DDB. A função retorna o identificador da nova imagem carregada ou `NULL` em caso de erro.

Para carregar uma imagem de um recurso, podemos utilizar o seguinte trecho de código (Listagem 6.2):

```
// Cria um identificador para o bitmap
HBITMAP hBmp = NULL;

// Carrega o bitmap do recurso, retornando um identificador HBITMAP.
// Note o type-casting feito no retorno da função LoadImage()

// Os parâmetros cxDesired e cyDesired recebem zero, ou seja, a função
// obtém esses valores automaticamente da imagem no recurso.
hBmp = (HBITMAP) LoadImage(hInstance, MAKEINTRESOURCE(IDB_MEUBITMAP),
IMAGE_BITMAP, 0, 0, LR_DEFAULTCOLOR);
```

Listagem 6.2: Carregar bitmap de recurso.

Nesse trecho de código, passamos a variável `hInstance` para `LoadImage()`, a mesma do parâmetro da função `WinMain()`, e informamos o ID do recurso da imagem (`IDB_MEUBITMAP`). Como é necessário informar todos os parâmetros, passamos `LR_DEFAULTCOLOR` para o último, que é o valor padrão de `UINT fuLoad`.

A Listagem 6.3 demonstra como carregar um arquivo de imagem (*imagem.bmp*) de tamanho 100x100 pixels armazenada em disco, retornando um DIB Section ao invés de um DDB (retorno padrão).

```
// Cria um identificador para o bitmap
HBITMAP hBmp = NULL;
```

```
// Carrega o bitmap do disco (LR_LOADFROMFILE),
// retornando um DIB Section (LR_CREATEDIBSECTION).
// Note o type-casting feito no retorno da função LoadImage()

hBmp = (HBITMAP) LoadImage(NULL, "imagem.bmp", IMAGE_BITMAP, 100, 100,
LR_CREATEDIBSECTION | LR_LOADFROMFILE);
```

Listagem 6.3: Carregar bitmap do disco.

Repare que, no caso de imagem em disco, passamos `NULL` para o primeiro parâmetro de `LoadImage()`, informamos o caminho e nome do arquivo *.bmp* no segundo parâmetro e o valor `LR_LOADFROMFILE` no parâmetro de opções de carregamento (`fuLoad`).

Obtendo informações de um bitmap

Conforme já visto, a função `GetObject()` obtém informações de objetos GDI (informado no primeiro parâmetro da função) e as salva na variável enviada no terceiro parâmetro.

No caso de objetos `HBITMAP`, obtemos suas informações com o uso da estrutura `BITMAP`, chamando `GetObject()` da seguinte maneira (Listagem 6.4):

```
HBITMAP hBmp;
BITMAP bmp;

// ...

// Preenche estrutura BITMAP com os dados do objeto GDI HBITMAP
GetObject((HBITMAP)hBmp, sizeof(BITMAP), &bmp);
```

Listagem 6.4: Obtendo informações de um bitmap.

```
typedef struct tagBITMAP {
  LONG bmType;
  LONG bmWidth;
  LONG bmHeight;
  LONG bmWidthBytes;
  WORD bmPlanes;
  WORD bmBitsPixel;
  LPVOID bmBits;
} BITMAP, *PBITMAP;
```

O membro `bmType` indica o tipo do bitmap, e deve ser zero. `bmWidth` e `bmHeight` armazenam a largura e altura do bitmap, respectivamente. `bmWidthBytes` indica a quantidade de bytes em cada linha da imagem. Esse valor pode ser calculado pela equação `bmWidthBytes = (bmBitsPixel / 8) * bmWidth`. `bmPlanes` indica a quantidade de camadas para as cores do bitmap, e normalmente recebe valor 1. O membro `bmBitsPixel` especifica quantos bits

devem ser utilizados para representar a cor de um ponto. `bmBits` é um ponteiro para os bits do bitmap, porém, como não temos acesso direto aos bits de um bitmap DDB, esse valor é zero. No caso de bitmaps DIB Section, usamos esse ponteiro para acessar e manipular os bits do bitmap.

DC de memória

Antes de aprendermos como mostrar um bitmap na área cliente de um programa, devemos saber como criar um DC de memória, que auxilia essa operação.

Um DC de memória pode ser utilizado em três ocasiões principais: quando não queremos que as modificações do DC de vídeo sejam passadas diretamente para a tela; quando precisamos salvar o conteúdo do DC de vídeo (problema que surgiu no programa *prog05-8.cpp*); ou quando vamos mostrar um bitmap na tela (o objetivo desse capítulo). Nessa última ocasião, é obrigatório o uso de um DC de memória, pois não há funções da Win32 API que mostrem diretamente um bitmap na tela.

O DC de memória trabalha como o DC de vídeo – podemos definir os objetos GDI que estão selecionados nele, suas propriedades, etc. Mas, conforme o nome diz, um DC de memória só existe na memória; para que possamos mostrar o seu conteúdo, devemos copiar o mesmo para o DC de vídeo (explicado mais adiante).

> **Dica:** um DC de memória costuma ser chamado também como um DC *off-screen*, devido ao fato que suas operações e seu conteúdo não são mostrados diretamente no vídeo.

A função `CreateCompatibleDC()` é utilizada para criar um DC de memória compatível com o dispositivo especificado no seu parâmetro, conforme o protótipo abaixo:

```
HDC CreateCompatibleDC(
  HDC hdc // identificador do DC
);
```

Se o parâmetro `hdc` receber `NULL`, a função cria um DC de memória compatível com as configurações atuais de vídeo onde o programa está sendo

exibido. A função retorna o identificador do novo DC de memória ou `NULL` no caso de erro.

Quando esse DC é criado, sua altura e largura, por padrão, é de um pixel monocromático cada. Antes de usá-lo para desenhar figuras, nosso programa deve selecionar nele um bitmap compatível com o DC de vídeo, que indica o tamanho correto do DC de memória e também sua configuração de cor.

A criação de um bitmap compatível com o DC de vídeo é feita através da função `CreateCompatibleBitmap()`.

Nota: um bitmap compatível criado com `CreateCompatibleBitmap()` é utilizado quando queremos executar operações de desenho no modo *off-screen* ou para salvar o conteúdo do DC de vídeo. Quando estamos criando um DC de memória para mostrar um bitmap, podemos selecionar o identificador do bitmap retornado pela função `LoadImage()` no DC de memória, configurando-o automaticamente.

```
HBITMAP CreateCompatibleBitmap(
  HDC hdc, // identificador do DC
  int nWidth, // largura do bitmap, em pixels
  int nHeight // altura do bitmap, em pixels
);
```

A função recebe em `HDC hdc` o identificador do DC de vídeo, cujo bitmap será compatível. Os parâmetros `int nWidth` e `int nHeight` indicam a largura e a altura do bitmap, respectivamente. Se não houver erros, a função retorna o identificador de um bitmap compatível (DDB); caso contrário, o retorno é `NULL`.

Nota: se passarmos o identificador do DC de memória para o primeiro parâmetro de `CreateCompatibleBitmap()`, o retorno da mesma será um bitmap monocromático, pois o DC de memória é monocromático (ou seja, o bitmap será compatível com o DC de memória).

Quando o bitmap criado por essa função não é mais utilizado, devemos deletá-lo com a função `DeleteObject()`. Também devemos deletar o DC de memória quando o mesmo não tiver mais utilidade, usando a função `DeleteDC()`.

```
BOOL DeleteDC(
  HDC hdc // identificador do DC
);
```

A função recebe o identificador do DC que será deletado e retorna um valor diferente de zero se a operação ocorreu normalmente ou zero em caso de erro.

Nota: não devemos deletar um DC criado com a função `GetDC()`; ao invés disso, devemos liberar o mesmo com `ReleaseDC()`.

A Listagem 6.5 mostra os passos para criar e configurar um DC de memória e um bitmap compatível.

```
HDC hDC = NULL; // DC de vídeo
HDC hMemDC = NULL; // DC de memória

HBITMAP hBmp = NULL; // Bitmap compatível
HBITMAP hBmpOld = NULL; // Salva bitmap anterior

// ...

// Obtém identificador do DC de vídeo
hDC = GetDC(hWnd);

// Cria DC de memória
hMemDC = CreateCompatibleDC(hDC);

// Cria bitmap compatível com DC de vídeo
hBmp = CreateCompatibleBitmap(hDC, WINDOW_WIDTH, WINDOW_HEIGHT);

// Seleciona bitmap no DC de vídeo e salva bitmap anterior
// Com isso, o DC de memória é configurado para ter a largura, a altura e
// a cor do bitmap selecionado nele, pois ele é inicialmente criado como
// um DC de tamanho 1x1 monocromático
hBmpOld = (HBITMAP)SelectObject(hMemDC, hBmp);

//
// Chama funções para desenhar no DC de memória
//

// Restaura bitmap anterior e deleta objeto GDI HBITMAP
SelectObject(hMemDC, hBmpOld);
DeleteObject(hBmp);

// Deleta DC de memória
DeleteDC(hMemDC);

// Libera DC de vídeo
ReleaseDC(hDC);

// ...
```

Listagem 6.5: Criando um DC de memória.

DC particular de um programa

Até agora, estávamos trabalhando com um DC de vídeo que tinha sua memória compartilhada com o sistema e os programas em execução. Para programas que usam operações de desenho intensivamente (como programas gráficos e jogos), o compartilhamento da memória do DC de vídeo não é viável, sendo recomendado o uso de um DC com sua memória restrita unicamente para o programa.

O uso de um DC particular é extremamente simples, sendo necessário informar apenas o valor `CS_OWNDC` no membro `style` da estrutura `WNDCLASSEX` (localizado dentro da `WinMain()`). Com isso, o programa já terá uma memória reservada para o seu DC. Ainda, não é necessário liberar ou deletar o DC do programa (como fazemos com um DC compartilhado), pois o mesmo é automaticamente liberado quando o programa é finalizado.

Mostrando bitmaps

Agora que já aprendemos que para mostrar um bitmap na tela é necessário o uso de um DC de memória, vamos ver como podemos visualizar um bitmap.

Para isso, o que fazemos, na verdade, não é mostrar o bitmap, mas sim copiar o conteúdo do DC de memória para o DC de vídeo. Dessa maneira, se um bitmap estiver selecionado no DC de memória, o mesmo será mostrado na tela quando houver a cópia de conteúdo entre os DC's (por isso criamos um identificador do tipo `HBITMAP` e o selecionamos no DC de memória).

A cópia do conteúdo de um DC para outro pode ser feito com duas funções: `BitBlt()` e `StretchBlt()`. O nome da função `BitBlt()` (pronunciado "*bit blit*") vem de "*BIT BLock Transfer*" (transferência de um bloco de bits), que é exatamente o que a função faz. A função `StretchBlt()` faz a transferência de um bloco de bits ajustando o tamanho da imagem de acordo com o retângulo de destino especificado na função, podendo alargar (se o retângulo de destino for maior que o de origem) ou comprimir (retângulo de destino menor que de origem) a imagem. Essa função ainda pode inverter a visualização da imagem (como um espelho) tanto na horizontal quanto na vertical (veja próximo tópico).

Dica: para salvar o conteúdo do DC de vídeo, copiamos o mesmo para um DC de memória. Quando a restauração do DC de vídeo for necessária, basta copiar o conteúdo do DC de memória de volta para o de vídeo.

```
BOOL BitBlt(
  HDC hdcDest, // identificador do DC de destino
  int nXDest, // x do canto superior esquerdo do retângulo de destino
  int nYDest, // y do canto superior esquerdo do retângulo de destino
  int nWidth, // largura do retângulo de destino (canto inferior direito)
  int nHeight, // altura do retângulo de destino (canto inferior direito)
  HDC hdcSrc, // identificador do DC de origem
  int nXSrc, // x do canto superior esquerdo do retângulo de origem
  int nYSrc, // y do canto superior esquerdo do retângulo de origem
  DWORD dwRop // modo de transferência
);
```

O primeiro parâmetro de `BitBlt()` indica o DC de destino (em qual DC o conteúdo de outro será copiado). Os quatro próximos parâmetros indicam três informações: a posição (`nXDest` e `nYDest`) e tamanho (`nWidth` e `nHeight`) do retângulo onde o conteúdo do DC de origem será copiado no DC de destino e também o tamanho do retângulo (`nWidth` e `nHeight`) que será transferido do DC de origem (esse é informado no sexto parâmetro, `hdcSrc`). Os parâmetros `nXSrc` e `nYSrc` indicam o ponto (x, y) de onde o conteúdo do DC de origem começará a ser transferido. O último parâmetro deve receber um dos valores da Tabela 6.2, que indicam o modo como as cores do DC de origem serão misturados ao DC de destino. A função retorna um valor diferente de zero quando não ocorrer erros, ou zero, caso contrário.

Valor	Descrição
`BLACKNESS`	Preenche o retângulo de destino com a cor preta.
`MERGECOPY`	Combina as cores do retângulo de origem com o pincél selecionado no DC de destino usando a operação AND.
`MERGEPAINT`	Combina as cores inversas do retângulo de origem com as cores do retângulo de destino usando a operação OR.
`NOTSRCCOPY`	Copia as cores inversas do retângulo de origem para o destino.
`NOTSRCERASE`	Combina as cores dos retângulos de origem e destino usando a operação OR e inverte a cor resultante.
`PATCOPY`	Copia o pincel selecionado no DC de destino

	no bitmap de destino.
PATINVERT	Combina as cores do pincel selecionado no DC de destino com as cores do retângulo de destino usando a operação XOR.
PATPAINT	Combina as cores do pincel selecionado no DC de destino com as cores inversas do retângulo de origem usando a operação OR. O resultado dessa operação é combinada com as cores do retângulo de destino usando a operação OR.
SRCAND	Combina as cores dos retângulos de origem e destino usando a operação AND.
SRCCOPY	Copia o retângulo de origem diretamente no retângulo de destino.
SRCERASE	Combina as cores inversas do retângulo de destino com as cores do retângulo de origem usando a operação AND.
SRCINVERT	Combina as cores dos retângulos de origem e destino usando a operação XOR.
SRCPAINT	Combina as cores dos retângulos de origem e destino usando a operação OR.
WHITENESS	Preenche o retângulo de destino com a cor branca.

Tabela 6.2: Modos de transferência (combinação de cores entre os DC's).

```
BOOL StretchBlt(
  HDC hdcDest, // identificador do DC de destino
  int nXOriginDest, // x do canto superior esquerdo do retângulo de destino
  int nYOriginDest, // y do canto superior esquerdo do retângulo de destino
  int nWidthDest, // largura do retângulo de destino (canto inferior
direito)
  int nHeightDest, // altura do retângulo de destino (canto inferior
direito)
  HDC hdcSrc, // identificador do DC de origem
  int nXOriginSrc, // x do canto superior esquerdo do retângulo de origem
  int nYOriginSrc, // y do canto superior esquerdo do retângulo de origem
  int nWidthSrc, // largura do retângulo de origem (canto inferior direito)
  int nHeightSrc, // altura do retângulo de origem (canto inferior direito)
  DWORD dwRop // modo de transferência
);
```

Veja que a função `StretchBlt()` recebe parâmetros parecidos com os da função `BitBlt()` (e tem valores de retorno igual ao dessa função). A diferença é o acréscimo dos parâmetros `int nWidthSrc` e `int nHeightSrc`, que indicam, respectivamente, a largura e a altura do retângulo do DC de origem que será

copiado para o DC de destino. Dessa maneira, os parâmetros `int nWidthDest` e `int nHeightDest` passam a indicar apenas o tamanho do retângulo do DC de destino, diferentemente da função `BitBlt()`.

A Listagem 6.6 demonstra um exemplo de como utilizar ambas as funções, mostrando um bitmap normalmente com `BitBlt()` e `StretchBlt()` e o mesmo bitmap redimensionado para o tamanho do retângulo de origem (com `StretchBlt()`). A execução desse trecho de código é mostrado na Figura 6.1.

```
// Verifica qual foi a mensagem enviada
switch(uMsg)
{

  case WM_CREATE: // Janela foi criada
  {
    // Cria DC de memória (hMemDC é global)
    hDC = GetDC(hWnd);
    hMemDC = CreateCompatibleDC(hDC);

    // Cria / carrega bitmap do arquivo “prog06-1.bmp” (hBmp é global)
    hBmp = (HBITMAP)LoadImage(NULL, " prog06-1.bmp", IMAGE_BITMAP, 160, 120,
LR_LOADFROMFILE);

    // Seleciona bitmap no DC de memória (configura DC de memória)
    SelectObject(hMemDC, hBmp);

    // Retorna 0, significando que a mensagem foi processada corretamente
    return(0);
  } break;

  case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
  {
    // Obtém DC de vídeo
    hDC = BeginPaint(hWnd, &psPaint);

    // Faz transferência de bits entre os DC’s de memória e vídeo
    BitBlt(hDC, 0, 0, 160, 120, hMemDC, 0, 0, SRCCOPY);

    // Faz transferência de bits entre os DC’s de memória e vídeo
    StretchBlt(hDC, 160, 0, 160, 120, hMemDC, 0, 0, 160, 120, SRCCOPY);

    // Faz transferência de bits entre os DC’s de memória e vídeo
    // (alargando a figura)
    StretchBlt(hDC, 0, 120, 240, 90, hMemDC, 0, 0, 160, 120, SRCCOPY);

    // Faz transferência de bits entre os DC’s de memória e vídeo
    // (comprimindo a figura)
    StretchBlt(hDC, 240, 120, 80, 60, hMemDC, 0, 0, 160, 120, SRCCOPY);

    // Libera DC de vídeo
    EndPaint(hWnd, &psPaint);

    return(0);
  } break;
```

```
  case WM_CLOSE: // Janela foi fechada
  {
    // Deleta bitmap
    DeleteObject(SelectObject(hMemDC, hBmp));

    // Deleta DC de memória
    DeleteDC(hMemDC);

    // Destrói a janela
    DestroyWindow(hWnd);

    return(0);
  } break;
  // ...
}
```

Listagem 6.6: Usando `BitBlt()` e `StretchBlt()`.

Dica: a listagem completa do código-fonte do exemplo de como mostrar bitmaps na tela encontra-se no arquivo *prog06-1.cpp* do CD-ROM. Nesse arquivo também foi inserido o código para mostrar bitmaps invertidos (que será discutido a seguir). Nesse programa, o clique do botão esquerdo do mouse modifica um flag que indica se os bitmaps devem ser invertidos ou não.

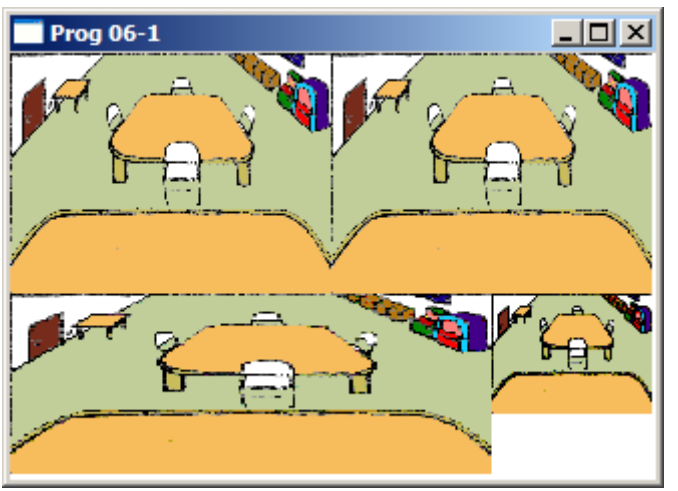

Figura 6.1: Mostrando imagens com `BitBlt()` e `StretchBlt()`.

Mostrando bitmaps invertidos

A função `StretchBlt()` também pode inverter bitmaps, fazendo um espelhamento da imagem. Para que a função inverta uma imagem, devemos

informar um valor negativo para o parâmetro `nWidthDest` (espelhamento na horizontal) e/ou `nHeightDest` (espelhamento na vertical).

Como já visto, os parâmetros `nXOriginDest`, `nYOriginDest`, `nWidthDest` e `nHeightDest` da função `StretchBlt()` servem para indicar o retângulo de destino da imagem; porém, quando `nWidthDest` e/ou `nHeightDest` recebem valores negativos, a imagem é mostrada invertida (na horizontal, vertical ou ambos os sentidos, dependendo de quais parâmetros receberem valores negativos).

A Listagem 6.7 contém um pequeno trecho de código que mostra um bitmap invertido com `StretchBlt()` – o código espelha horizontal e verticalmente a segunda imagem mostrada com `StrecthBlt()` da Listagem 6.4.

```
// Faz transferência de bits entre os DC's de memória e vídeo
// (alargando a figura, bitmap invertido na horizontal e vertical)

// Parâmetros de StretchBlt() para inverter bitmap:
// nXOriginDest = Ponto X de destino + largura do bitmap;
// nYOriginDest = Ponto Y de destino + altura do bitmap;
// nWidthDest = nWidthDest negativo (inverte imagem na horizontal);
// nHeightDest = nHeightDest negativo (inverte imagem na vertical);

StretchBlt(hDC, 240, 210, -240, -90, hMemDC, 0, 0, 160, 120, SRCCOPY);
```

Listagem 6.7: Bitmap invertido com `StretchBlt()`.

Você deve ter observado que os valores passados para os parâmetros `nXOriginDest` e `nYOriginDest` estão acrescentados com a largura e altura do bitmap, respectivamente. Somamos esses valores pois quando um bitmap está sendo invertido por `StretchBlt()`, a imagem é desenhada na tela da direita para esquerda (espelhamento horizontal) e de baixo para cima (espelhamento vertical). Por exemplo, se enviássemos o valor zero para `nXOriginDest`, a imagem não seria mostrada na tela, pois ela não estaria dentro da área cliente (a imagem começaria a ser desenhada a partir do ponto `X = 0` até `X = -240`, o que estaria incorreto). Caso haja dúvidas em relação ao modo que `StretchBlt()` desenha os bitmaps (normais e invertidos) na tela, veja o esquema apresentado na Figura 6.2.

A Figura 6.3 demonstra o programa *prog06-1.cpp* em execução, com o *flag* do programa configurado para mostrar os bitmaps invertidos, através do uso da função `StretchBlt()`.

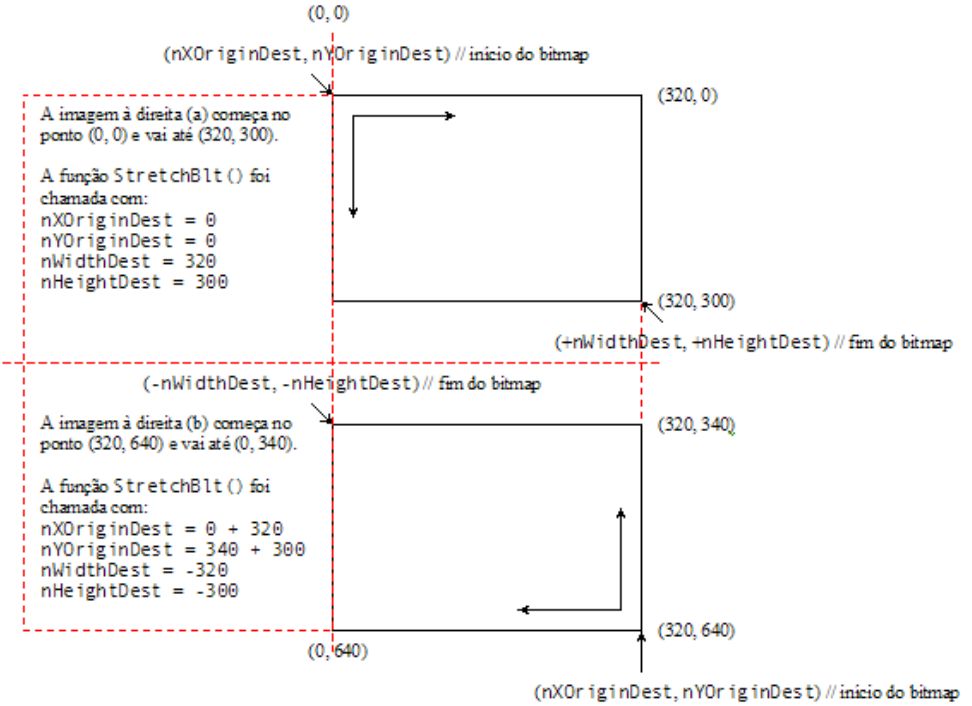

Figura 6.2: Direção que `StretchBlt()` mostra bitmaps: (a) normais e (b) invertidos.

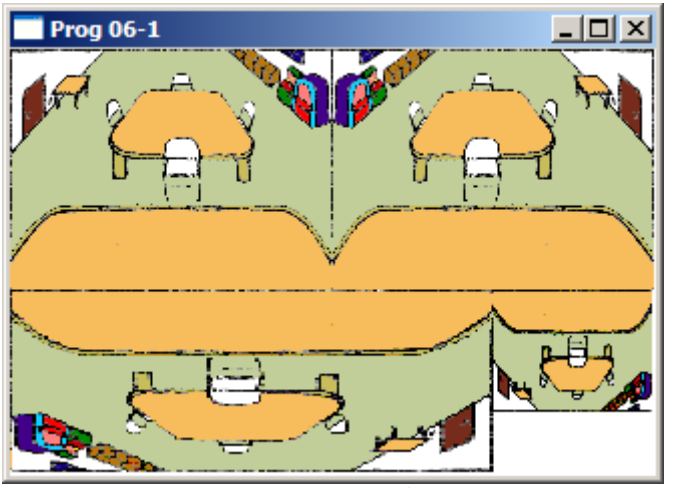

Figura 6.3: Bitmaps invertidos com `StretchBlt()`.

DIB Section

Os bitmaps do tipo DDB possuem a desvantagem de não podermos acessar e manipular diretamente os seus bits, como já dito anteriormente. Por exemplo, não podemos carregar uma imagem colorida e mostrá-la em tons de cinza, pois para isso devemos manipular os bits da imagem.

A solução para esse problema é utilizarmos bitmaps do tipo DIB, mas como é necessário uma conversão DIB para DDB para mostrar a imagem na tela (ocorrendo perda de performance), o melhor caminho é criar um objeto GDI `HBITMAP` configurado como DIB Section.

Embora exista mais de uma maneira para criar e carregar um bitmap do tipo DIB Section, apresentarei aqui a solução mais rápida e fácil, utilizando a função `LoadImage()`, que já estudamos no começo do capítulo.

O último parâmetro da função `LoadImage()` pode receber o valor `LR_CREATEDIBSECTION`, indicando que o retorno da função será um objeto `HBITMAP` do tipo DIB Section. Simples, não? Inclusive, já vimos um exemplo da criação de um DIB Section com `LoadImage()` na Listagem 6.3.

Criar um DIB Section com `LoadImage()` é rápido e fácil. Mas não queremos somente isso; o nosso objetivo de usar um DIB Section é manipular os bits de uma imagem para podermos criar certos efeitos na mesma. Para acessarmos os bits de uma imagem, devemos preencher uma estrutura `DIBSECTION`:

```
typedef struct tagDIBSECTION {
  BITMAP dsBm;
  BITMAPINFOHEADER dsBmih;
  DWORD dsBitfields[3];
  HANDLE dshSection;
  DWORD dsOffset;
} DIBSECTION, *PDIBSECTION;
```

Os dois primeiros membros (`BITMAP` e `BITMAPINFOHEADER`) da estrutura `DIBSECTION` são duas outras estruturas definidas pela Win32 API, que contêm informações sobre o bitmap. O membro `dsBitfields[3]` é válido apenas quando o membro `biBitCount` da estrutura `BITMAPINFOHEADER` é 16- ou 32-bit. No caso afirmativo, cada posição do vetor armazena uma máscara indicando quais bits de cada pixel correspondem ao canal de cor RGB. Embora não estaremos utilizando os dois últimos membros (`dshSection` e `dsOffset`), eles são utilizados

quando um DIB Section é criado com a função `CreateDIBSection()` utilizando um objeto *file mapping*.

> **Dica:** *File mapping* é utilizado para associar o conteúdo de um arquivo com um espaço de memória virtual de um processo, assunto que está fora do contexto desse livro.

Para preenchermos essa estrutura, podemos usar a função `GetObject()`, que obtém informações do objeto GDI, da seguinte maneira (Listagem 6.8):

```
// Assumimos que hBmp é um HBITMAP DIB Section, no qual carregamos um bitmap
// com a função LoadImage()

// Cria e preenche uma estrutura DIBSECTION
DIBSECTION dibSect;
GetObject((HBITMAP)hBmp, sizeof(DIBSECTION), (LPVOID)&dibSect);
```

Listagem 6.8: Preenchendo uma estrutura `DIBSECTION`.

Uma estrutura que ainda não discutimos é a `BITMAPINFOHEADER`, que faz parte da `DIBSECTION`. Essa estrutura contém informações referentes às dimensões e cores de um DIB.

```
typedef struct tagBITMAPINFOHEADER {
  DWORD biSize;
  LONG biWidth;
  LONG biHeight;
  WORD biPlanes;
  WORD biBitCount;
  DWORD biCompression;
  DWORD biSizeImage;
  LONG biXPelsPerMeter;
  LONG biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

O primeiro membro da estrutura (`biSize`) indica o número de bytes utilizados pela estrutura (`sizeof(BITMAPINFOHEADER)`). `biWidth` e `biHeight` armazenam a largura e altura do bitmap, respectivamente. `biPlanes` indica a quantidade de camadas do dispositivo, e deve ser 1. `biBitCount` indica a quantidade de bits que o bitmap usa para representar um pixel. O membro `biCompression` informa o tipo de compressão do bitmap. Geralmente, os valores são `BI_RGB` ou `BI_BITFIELDS`. Em `biSizeImage` é armazenado o tamanho da imagem em bytes, que pode ser calculado pela equação `biSizeImage = biWidth * biHeight * biBitCount / 8`. `biXPelsPerMeter` e `biYPelsPerMeter` indicam a resolução horizontal e vertical (em pixels por metro) do dispositivo de destino

do bitmap. O membro `biClrUsed` especifica a quantidade de cores na paleta de cores que realmente são utilizados pelo bitmap. Quando recebe zero, significa que o bitmap usa o número máximo de cores indicado por `biBitCount`. Por fim, `biClrImportant` especifica a quantidade de cores necessárias para mostrar o bitmap. No caso de zero, todas as cores são necessárias. Os quatro últimos membros dessa estrutura normalmente recebem o valor zero.

Nota: caso `biHeight` seja positivo, isso significa que a origem do bitmap é o canto inferior-esquerdo. Caso seja negativo, o bitmap tem sua origem no canto superior-esquerdo.

Manipulando os bits de um bitmap: tons de cinza e contraste

Os bits que compõem uma imagem são armazenados no membro `LPVOID bmBits` da estrutura `BITMAP dsBm` (a qual é um membro da estrutura `DIBSECTION`). Através desse ponteiro, podemos acessar e manipular os bits de um bitmap, tendo a possibilidade de modificar a imagem original.

Vamos estudar um caso onde manipulamos os bits de uma imagem bitmap de 24-bit (um byte para cada cor RGB). O arquivo *prog06-2.cpp* do CD-ROM carrega a imagem mostrada na Figura 6.4 (*frutas_24bit.bmp*, uma imagem colorida de 24-bit), criando um `HBITMAP` configurado como DIB Section. No processo da mensagem `WM_PAINT`, o programa chama a função `ManipulaPixels()` (Listagem 6.9), onde uma estrutura `DIBSECTION` é criada e preenchida conforme mencionado anteriormente.

Figura 6.4: Um bitmap 24-bit prestes a ser modificado.

Na função `ManipulaPixels()`, declaramos uma variável (`pImgBits`) do tipo `unsigned char*`, que é um ponteiro para os bits da imagem. Em seguida, preenchemos a estrutura `DIBSECTION`, com a chamada à função `GetObject()` e apontamos `pImgBits` para `DIBSECTION::BITMAP::bmBits`. Note que esse membro é do tipo `LPVOID`, sendo necessário um type-casting para o tipo `unsigned char*`. Em seguida, entramos em um `for` (que vai de 0 até o tamanho da imagem em bytes, com passo 3), onde os bits da imagem serão modificados.

```
void ManipulaPixels(void)
{
  DIBSECTION dibSect; // Informações do bitmap DIB Section
  unsigned char *pImgBits; // Ponteiro para bits do bitmap

  // Preenche estrutura DIBSECTION com informações do bitmap
  // e faz pImgBits apontar para os bits do bitmap
  GetObject((HBITMAP)hBmp, sizeof(DIBSECTION), (LPVOID)&dibSect);
  pImgBits = (unsigned char *)dibSect.dsBm.bmBits;

  // Converte RGB para tons de cinza
  for(int i = 0; i < dibSect.dsBmih.biSizeImage; i += 3)
  {
    // Obtém os valores RGB dos pontos da imagem
    int R = pImgBits[i + 2]; // Red (Vermelho)
    int G = pImgBits[i + 1]; // Green (Verde)
    int B = pImgBits[i];     // Blue (Azul)

    // Calcula luminance / brilho / intensidade de luz
    R = (R << 6) + (R << 3) + (R << 2) + 1;         // R * 77
    G = (G << 7) + (G << 4) + (G << 2) + (G << 1);  // G * 150
    B = (B << 4) + (B << 3) + (B << 2) + 1;         // B * 29
    int L = (R + G + B) >> 8; // L = (R + G + B) / 256

    // Converte para tons de cinza
    pImgBits[i + 2] = L; // Red (Vermelho)
    pImgBits[i + 1] = L; // Green (Verde)
    pImgBits[i]     = L; // Blue (Azul)
  }
}
```

Listagem 6.9: Manipulando os bits de uma imagem.

Dentro do loop `for` da função, os bits da imagem são modificados, fazendo uma conversão RGB (imagem colorida) para tons de cinza. A modificação dos bits é feita associando o valor de `iBrilho` para `pImgBits[i]`, `pImgBits[i + 1]` e `pImgBits[i + 2]`. Esses índices (`i`, `i + 1` e `i + 2`) são utilizados pois bitmaps 24-bit utilizam um byte para representar cada componente RGB.

Para converter uma imagem colorida para uma imagem com tonalidades de cinza, aplicamos o seguinte algoritmo (Listagem 6.10, em pseudo-código):

```
para cada ponto P na imagem, (P contém Red, Green e Blue)
  calcular L = (0.299 * P.Red) + (0.587 * P.Green) + (0.114 * P.Blue)
  atribuir P.Red   = L
  atribuir P.Green = L
  atribuir P.Blue  = L
```

Listagem 6.10: Conversão RGB para tons de cinza (pseudo-código).

Nota: L vem do inglês *luminance* (brilho ou intensidade de luz em uma cor), que atribuindo seu valor para os componentes RGB, obtém-se a tonalidade de cinza.

Veja que a equação do pseudo-código é diferente do código da Listagem 6.9, porém o resultado é o mesmo, pois a equação `L = ((77 * P.Red) + (150 * P.Green) + (29 * P.Blue)) / 256` foi utilizada para que não fosse necessário o uso de variáveis `float`. Outra otimização realizada foi o uso dos operadores binários *shift-left* e *shift-right*, substituindo operações de multiplicação e divisão, respectivamente.

Dica: embora nesse caso a otimização não seja um ponto tão crítico, é bom saber que em certas ocasiões é possível otimizar cálculos e ganhar performance através dos operadores binários shift-left e shift-right. No cálculo da variável `R` da função `ManipulaPixels()`, a operação `R = (R << 6) + (R << 3) + (R << 2) + 1` é equivalente a $R * 2^6 + R * 2^3 + R * 2^2 + 1$, que resulta em R * 77. Shift-left executa a operação $num1 << num2 = num1 * 2^{num2}$ e shift-right executa a operação $num1 >> num2 = num1 / 2^{num2}$.

Com uma pequena modificação no loop `for` da função `ManipulaPixels()`, podemos criar outra função, `Contraste()` (Listagem 6.11, contendo o pseudo-código da função), que configura o *gamma* (contraste) da imagem, podendo clarear ou escurecer a mesma.

```
/*
pseudo-código para configurar contraste de uma imagem:

definir o valor de gamma:
  valores entre 0.0 e 1.0 clareiam a imagem
  valores acima de 1.0 escurecem a imagem

para cada ponto P na imagem, (P contém Red, Green e Blue)
  atribuir P.Red   = 255 * (P.Red / 255.0)^gamma
  atribuir P.Green = 255 * (P.Green / 255.0)^gamma
  atribuir P.Blue  = 255 * (P.Blue / 255.0)^gamma
*/

void Contraste(DIBSECTION dibSect, float fGamma)
{
```

```
  // Ponteiro para bits do bitmap
  unsigned char *pImgBits = (unsigned char *)dibSect.dsBm.bmBits;

  // Configura o contraste da imagem
  for(int i = 0; i < dibSect.dsBmih.biSizeImage; i += 3)
  {
    // Obtém os valores RGB dos pontos da imagem
    int R = pImgBits[i + 2]; // Red (Vermelho)
    int G = pImgBits[i + 1]; // Green (Verde)
    int B = pImgBits[i];     // Blue (Azul)

    // Aplica novo contraste
    R = 255 * pow(R / 255.0, fGamma);
    G = 255 * pow(G / 255.0, fGamma);
    B = 255 * pow(B / 255.0, fGamma);

    // Atualiza os pontos da imagem
    pImgBits[i + 2] = R; // Red (Vermelho)
    pImgBits[i + 1] = G; // Green (Verde)
    pImgBits[i]     = B; // Blue (Azul)
  }
}
```

Listagem 6.11: Configurar contraste de uma imagem (pseudo-código e código).

A função `Contraste()` recebe dois parâmetros: um `DIBSECTION`, que contém as informações de um bitmap DIB Section, e um `float`, que configura o contraste da imagem. Supomos que uma `DIBSECTION` já esteja preenchida e seja passada para a função `Contraste()`, mas poderíamos obter as informações de um DIB Section dentro dessa função, assim como foi feito em `ManipulaPixels()`.

Assim, chegamos ao final de mais um capítulo. Você aprendeu como utilizar imagens bitmap nos programas, a diferença entre DDB's e DIB's e como modificar os bits de uma imagem, podendo aplicar diversos efeitos nas imagens, como transformar uma imagem colorida em tons de cinza e modificar seu contraste. No próximo capítulo, estudaremos regiões e algumas de suas aplicações.

Capítulo 7 – Regiões

Há alguns anos, os programas para Windows eram "monótomos"; todos eles tinham o mesmo visual-padrão retangular que já estamos acostumados a trabalhar. Com o lançamento e o grande uso do tocador de músicas digitais Winamp, essa padronização de programas com janelas retangulares perdeu um pouco seu território, pois o Winamp deu ao usuário a possibilidade de modificar totalmente o visual do programa, através de imagens conhecidas por *"skins"*. Seguindo o conceito de utilização de skins, diversos programas foram surgindo, onde acabaram acrescentando algo mais: a personalização não somente do visual, como também do formato da janela do programa.

Hoje, os programas não precisam mais serem retangulares; podemos criar programas com janelas redondas, em forma de estrela, com os cantos de um retângulo arredondados e até mesmo com o formato de uma foto. A criação desses tipos de janelas é possível através do uso de regiões.

O que são regiões?

Regiões são objetos GDI (`HRGN`) que armazenam retângulos, polígonos, elipses ou combinações desses três tipos de geometria, sendo possível preencher, executar testes da posição de cursor e modificar o formato da janela (área cliente) de um programa.

Dica: uma região pode ser classificada de acordo com sua complexidade, ou seja, de que forma ela é formada. Uma região é do tipo `SIMPLEREGION` quando é composta por um único retângulo e é do tipo `COMPLEXREGION` quando é composta por mais de um retângulo. `NULLREGION` significa que uma região é vazia.

Criando regiões

Podemos criar quatro tipos de regiões: no formato de uma elipse (função `CreateEllipticRgn()`), retangulares (função `CreateRectRgn()`), retangulares com cantos arredondados (função `CreateRoundRectRgn()`) e poligonais (função `CreatePolygonRgn()`). Cada um desses formatos possuem suas próprias funções para criação de regiões, conforme mencionados entre

parênteses, sendo que elas trabalham de forma muito semelhantes às funções `Ellipse()`, `Rectangle()` e `RoundRect()` e `Polygon()`, respectivamente, vistas no capítulo 5. As diferenças entre as funções de figuras e de regiões são que as últimas não recebem um identificador de DC como parâmetro e retornam uma região (variável do tipo `HRGN`) quando criada ou `NULL` caso contrário. Os protótipos de cada uma das funções são listados a seguir.

```
HRGN CreateEllipticRgn(
  int nLeftRect, // coordenada x do canto superior esquerdo do retângulo
  int nTopRect, // coordenada y do canto superior esquerdo do retângulo
  int nRightRect, // coordenada x do canto inferior direito do retângulo
  int nBottomRect // coordenada y do canto inferior direito do retângulo
);
```

```
HRGN CreateRectRgn(
  int nLeftRect, // coordenada x do canto superior esquerdo
  int nTopRect, // coordenada y do canto superior esquerdo
  int nRightRect, // coordenada x do canto inferior direito
  int nBottomRect // coordenada y do canto inferior direito
);
```

```
HRGN CreateRoundRectRgn(
  int nLeftRect, // coordenada x do canto superior esquerdo
  int nTopRect, // coordenada y do canto superior esquerdo
  int nRightRect, // coordenada x do canto inferior direito
  int nBottomRect, // coordenada y do canto inferior direito
  int nWidthEllipse, // largura da elipse
  int nHeightEllipse // altura da elipse
);
```

```
HRGN CreatePolygonRgn(
  CONST POINT *lppt, // vértices do poligono
  int cPoints, // quantidade de vértices do polígono
  int fnPolyFillMode // modo de preenchimento
);
```

Veja que a função `CreatePolygonRgn()` possui um parâmetro `int fnPolyFillMode`, que recebe o mesmo valor (`ALTERNATE` ou `WINDING`) que a função `SetPolyFillMode()` (vista no capítulo 5) e possui o mesmo significado.

Desenhando regiões

Uma região fica invisível até que o programa seja instruído para preencher seu interior por determinado pincel, inverter as cores do seu interior ou traçar uma borda em volta da região.

Para preencher o interior de uma região, podemos utilizar duas funções: `FillRgn()` e `PaintRgn()`. A primeira função preenche a região com as

configurações de um pincel enviado como parâmetro, enquanto a segunda preenche a região usando o pincel que está selecionado no DC.

```
BOOL FillRgn(
  HDC hdc, // identificador do DC
  HRGN hrgn, // identificador da região a ser preenchida
  HBRUSH hbr // identificador do pincel a ser utilizado
);
```

A função `FillRgn()` recebe o identificador do DC no primeiro parâmetro (`HDC hdc`), o identificador da região que será preenchida no segundo (`HRGN hrgn`) e o pincel que será utilizado para o preenchimento, no terceiro parâmetro (`HBRUSH hbr`). O retorno da função será diferente de zero quando não houver erros ou zero caso contrário.

```
BOOL PaintRgn(
  HDC hdc, // identificador do DC
  HRGN hrgn // identificador da região a ser preenchida
);
```

A função `PaintRgn()` recebe apenas o identificador do DC (`HDC hdc`) e o identificador da região (`HRGN hrgn`) que será preenchida, visto que a mesma utiliza o pincel selecionado no DC para preenchimento. O retorno da função será diferente de zero quando não houver erros ou zero caso contrário.

Para inverter as cores de uma região, basta utilizar a função `InvertRgn()`.

```
BOOL InvertRgn(
  HDC hdc, // identificador do DC
  HRGN hrgn // identificador da região a ser invertida
);
```

A função recebe o identificador do DC no primeiro parâmetro (`HDC hdc`) e o identificador da região que terá sua cor invertida (`HRGN hrgn`). O retorno da função será diferente de zero quando não houver erros ou zero caso contrário.

Para traçar uma borda em volta da região, utilizamos a função `FrameRgn()`.

```
BOOL FrameRgn(
  HDC hdc, // identificador do DC
  HRGN hrgn, // identificador da região onde será traçada a borda
  HBRUSH hbr, // identificador do pincel para traçar a borda
  int nWidth, // largura da borda
  int nHeight // altura da borda
```

```
);
```

A função recebe o identificador do DC no primeiro parâmetro (`HDC hdc`), o identificador da região em `HRGN hrgn`, o identificador do pincel que será utilizado para traçar a borda em `HBRUSH hbr` e a largura e altura da borda (`int nWidth` e `int nHeight`, respectivamente). Ela retorna zero em caso de erros ou um valor diferente de zero caso contrário.

Operações com regiões

Além de criar e desenhar regiões, podemos realizar algumas operações, tais como verificar se duas regiões são iguais, verificar se um ponto ou um retângulo está dentro de uma região, mover uma região ou fazer uma combinação de duas regiões.

Para verificarmos se duas regiões são idênticas, usamos a função `EqualRgn()`. Duas regiões são consideradas idênticas se são iguais em tamanho e formato.

```
BOOL EqualRgn(
  HRGN hSrcRgn1, // identificador da primeira região
  HRGN hSrcRgn2 // identificador da segunda região
);
```

A função recebe como parâmetros (`hSrcRgn1` e `hSrcRgn2`) os dois identificadores das regiões que serão comparadas. O retorno pode ser um valor diferente de zero se ambas regiões são idênticas, zero quando são diferentes ou `ERROR`, indicando que pelo menos um dos identificadores é inválido.

A função `PtInRegion()` verifica se um determinado ponto encontra-se ou não dentro de uma região especificada.

```
BOOL PtInRegion(
  HRGN hrgn, // identificador da região
  int X, // coordenada x do ponto
  int Y // coordenada y do ponto
);
```

Essa função recebe o identificador da região no primeiro parâmetro e a coordenada do ponto (x, y) no segundo e terceiro parâmetro. Ela então verifica se o ponto está dentro da região e retorna um valor diferente de zero em caso afirmativo ou zero caso o ponto esteja fora da região.

Podemos verificar também se um retângulo ou parte dele está dentro de uma região, através da função `RectInRegion()`.

```
BOOL RectInRegion(
  HRGN hrgn, // identificador da região
  CONST RECT *lprc // ponteiro para o retângulo
);
```

A função recebe o identificador da região e um ponteiro para uma estrutura `RECT`, que contém as coordenadas do retângulo. Caso o retângulo, integral ou parcialmente, esteja dentro da região, a função retorna um valor diferente de zero. Quando nenhuma parte do retângulo está dentro da região, a função retorna o valor zero.

Quando quisermos mover uma região, chamamos a função `OffsetRgn()`.

```
int OffsetRgn(
  HRGN hrgn, // identificador da região
  int nXOffset, // desvio no eixo x
  int nYOffset // desvio no eixo y
);
```

`OffsetRgn()` recebe em `HRGN hrgn` o identificador da região que será movida para esquerda/direita (`int nXOffset`) e/ou para cima/baixo (`int nYOffset`). O retorno pode ser `NULLREGION` (região é vazia), `SIMPLEREGION` (região de um único retângulo), `COMPLEXREGION` (região de mais de um retângulo) ou `ERROR` (ocorreu erro e região não foi modificada).

Uma outra operação que podemos fazer com regiões é a combinação delas, resultando em uma nova região. Para isso, utilizamos a função `CombineRgn()`.

```
int CombineRgn(
  HRGN hrgnDest, // identificador da região de destino
  HRGN hrgnSrc1, // identificador da região de origem 1
  HRGN hrgnSrc2, // identificador da região de origem 2
  int fnCombineMode // modo de combinação das regiões
);
```

Para realizar a combinação de regiões, a função `CombineRgn()` recebe duas regiões de origem (as regiões que serão combinadas), nos parâmetros `HRGN hrgnSrc1` e `HRGN hrgnSrc2`. O resultado da combinação é armazenada numa outra região, indicada no parâmetro `HRGN hrgnDest`. O último parâmetro, `int fnCombineMode` indica como as duas regiões serão combinadas, podendo receber

RGN_AND (intersecção das duas regiões), RGN_COPY (copia apenas a região identificada em hsrcSrc1), RGN_DIFF (combina a parte de hrgnSrc1 que não faz parte de hrgnSrc2), RGN_OR (união das duas regiões) ou RGN_XOR (união das duas regiões exceto áreas que fazem parte de ambas). A função pode retornar NULLREGION (região é vazia), SIMPLEREGION (região de um único retângulo), COMPLEXREGION (região de mais de um retângulo) ou ERROR (região não foi criada).

Dica: a região de destino não precisa ser necessariamente uma outra região, pode ser uma das regiões de origem; por exemplo, a região hrgnSrc1 pode ser a mesma que hrgnDest. Dessa maneira, a função irá combinar as regiões hrgnSrc1 e hrgnSrc2 e armazenará a combinação em hrgnSrc1.

Regiões de corte

As regiões também atuam como regiões de corte (*clipping region*), especificando um espaço onde qualquer operação gráfica fora desse limite não será afetada. Isso pode ser útil quando precisamos modificar apenas um determinado pedaço da área cliente ou quando queremos que todas as operações sejam feitas em apenas uma região da área cliente, não limitando-se à regiões retangulares.

Para determinar uma região de corte, basta que criemos uma região (e, se necessário, realizar operações com regiões, conforme mencionado nesse capítulo) e em seguida selecionarmos a região no DC atual. Por exemplo, podemos criar uma região de corte circular em um programa que mostre um bitmap (retangular), conforme a Listagem 7.1. Como definimos uma região de corte, toda a área ocupada pela imagem que não esteja dentro da região delimitada não será mostrada na tela (Figura 7.1).

```
//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  // ...

  case WM_CREATE: // Janela foi criada
  {
    // Cria uma região elíptica
    hRgn = CreateEllipticRgn(80, 50, 230, 200);

    // Cria DC de memória (hMemDC é global)
    hDC = GetDC(hWnd);
```

```
    hMemDC = CreateCompatibleDC(hDC);

    // Cria DIB Section, carregando bitmap do arquivo "frutas_24bit.bmp"
    hBmp = (HBITMAP)LoadImage(NULL, "frutas_24bit.bmp", IMAGE_BITMAP, 320,
240, LR_LOADFROMFILE | LR_CREATEDIBSECTION);

    // Seleciona bitmap no DC de memória (configura DC de memória)
    SelectObject(hMemDC, hBmp);

    // Libera DC de vídeo
    ReleaseDC(hWnd, hDC);

    // Retorna 0, significando que a mensagem foi processada corretamente
    return(0);
  } break;

  case WM_PAINT: // Janela (ou parte dela) precisa ser atualizada
  {
    // Obtém DC de vídeo
    hDC = BeginPaint(hWnd, &psPaint);

    // Seleciona região no DC, definindo região de corte
    SelectObject(hDC, hRgn);

    // Faz transferência de bits entre os DC's de memória e vídeo
    BitBlt(hDC, 0, 0, 320, 240, hMemDC, 0, 0, SRCCOPY);

    // Libera DC de vídeo
    EndPaint(hWnd, &psPaint);

    return(0);
  } break;

  // ...
}
```

Listagem 7.1: Definindo uma região de corte.

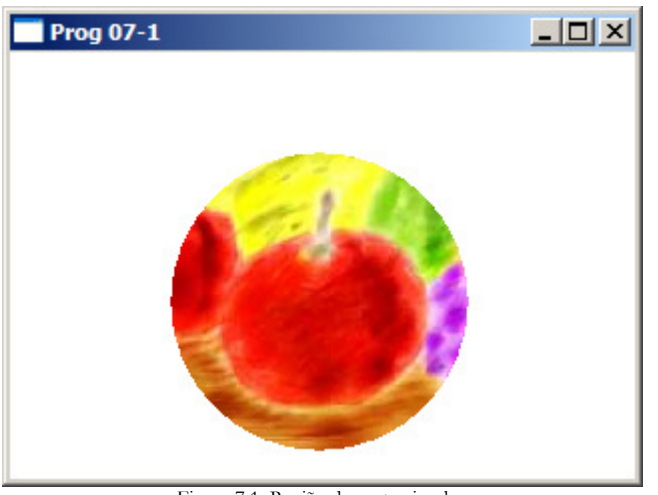

Figura 7.1: Região de corte circular.

Nota: o arquivo *prog07-1.cpp* do CD-ROM contém o código-fonte da Listagem 7.1 na íntegra, demonstrando a criação e uso de regiões.

Criando janelas não-retangulares

A última operação com regiões que veremos é a criação de janelas não-retangulares, mencionadas no início desse capítulo. Programas com janelas não-retangulares contêm algumas diferenças, comparadas aos com janelas retangulares. Uma das primeiras diferenças é a barra de título do programa: geralmente ela não é utilizada, por questões de estética. Logo, a janela deve ser criada com a propriedade `WS_POPUP | WS_VISIBLE` (parâmetro `dwStyle` da função `CreateWindowEx()`).

A segunda diferença é que, ao desabilitarmos a barra de título, o usuário não pode arrastar o programa para onde ele quiser. Para contornar esse problema, enviamos a mensagem `WM_NCLBUTTONDOWN` para o loop de mensagens com o valor `HTCAPTION` quando o usuário clicar com o botão esquerdo do mouse em determinada área do programa, simulando um clique na barra de título. Dessa maneira, o programa automaticamente irá considerar tal clique como um clique na barra de título, permitindo ao usuário arrastar o programa pela tela. O trecho de código da Listagem 7.2 simula um clique na barra de título quando o cursor do mouse estiver no topo da janela.

```
case WM_LBUTTONDOWN:
{
  // Se a posição y do cursor do mouse for menor que 100
  if(HIWORD(lParam) < 100)
    // Envia mensagem WM_NCLBUTTONDOWN, indicando que foi um
    // clique na barra de título
    SendMessage(hWnd, WM_NCLBUTTONDOWN, HTCAPTION, lParam);

  return(0);
} break;
```

Listagem 7.2: Simulando clique na barra de título.

Para criarmos uma janela não-retangular (definir o formato de uma região para a janela do programa) usamos a função `SetWindowRgn()`.

```
int SetWindowRgn(
  HWND hWnd, // identificador da janela
  HRGN hRgn, // identificador da região
  BOOL bRedraw // redesenha janela?
);
```

A função `SetWindowRgn()` recebe o identificador da janela que será modificada, o identificador da região que determinará o formato da janela e um flag que especifica se a janela será redesenhada após ter seu formato modificado. Geralmente, `bRedraw` recebe `TRUE` quando a janela está visível. A função retorna um valor diferente de zero quando nenhum problema ocorrer ou zero caso contrário.

Nota: quando `SetWindowRgn()` é executada sem erros, o sistema obtém controle da região indicada em `hRgn`; portanto, não devemos executar nenhuma operação nesse identificador nem deletar o objeto `HRGN`, pois o próprio sistema o fará quando o identificador da região não for mais utilizado.

Para exemplificar a criação de janelas não-retangulares, o código da função `WindowSkin()` (Listagem 7.3) executa um simples algoritmo que pode ser utilizado para o uso de skins de um programa (uma skin deve ser uma imagem carregada em um objeto `HBITMAP` e suas partes transparentes devem ter a cor RGB(255, 0, 255)).

A função primeiro seleciona o bitmap no DC (ambos enviados como parâmetros da função). Em seguida, executa dois loops onde varre os pontos do bitmap, obtendo o formato da figura. Dentro dos loops, a função verifica se a cor do ponto (x, y) atual da imagem é igual a cor de transparência (definida na variável `COLORREF rgbTransparente`). Se for, a função simplesmente ignora tal ponto e verifica o próximo. Quando o ponto não for transparente, a função então salva o primeiro ponto não-transparente e varre a imagem na horizontal até que encontre o último ponto não-transparente (ou seja, até chegar ao fim da linha atual (y) ou até encontrar um ponto transparente). Depois disso, a função cria uma região temporária, contendo os pontos não-transparentes da linha atual, e a combina com a região atual, que ao final dos loops conterá o formato da figura. Por fim, a função define o formato da janela de acordo com a região atual.

```
//--------------------------------------------------------------------------
// WindowSkin() -> Cria janela no formato de uma figura
//--------------------------------------------------------------------------
void WindowSkin(HWND hWnd, HDC hMemDC, HBITMAP hBmp)
{
  // Cria região no formato da figura
  HRGN hRgn = CreateRectRgn(0, 0, 0, 0);
  HRGN hRgnTemp = hRgn;

  // Cor transparente
```

```
  COLORREF rgbTransparente = RGB(255, 0, 255);

  // Salva o primeiro ponto não-transparente da posição x da imagem
  int ix = 0;

  // Seleciona bitmap no DC de memória (configura DC de memória)
  SelectObject(hMemDC, hBmp);

  // Varre o bitmap para obter cores e o formato da figura
  for(int y = 0; y <= WINDOW_HEIGHT; y++)
  {
    for(int x = 0; x <= WINDOW_WIDTH; x++)
    {
      // Verifica se a cor do ponto atual é transparente
      if(GetPixel(hMemDC, x, y) != rgbTransparente)
      {
        // Se não for, salva o primeiro ponto não-transparente da
        // posição x da imagem
        ix = x;

        // E incrementa a posição x até achar um ponto que é transparente
        // ou até chegar ao final da imagem (na horizontal)
        while((x <= WINDOW_WIDTH) &&
              (GetPixel(hMemDC, x, y) != rgbTransparente))
          x++;

        // Depois cria uma região temporária onde:
        // x = primeiro ponto não-transparente até último ponto
        // não-transparente
        // y = posição y atual até próximo y
        hRgnTemp = CreateRectRgn(ix, y, x, y + 1);

        // Combina a região atual (hRgn) com a região temporária (hRgnTemp)
        CombineRgn(hRgn, hRgn, hRgnTemp, RGN_OR);
      }
    }
  }

  // Depois que varreu todo o bitmap, verificando seu formato (de acordo com
  // as partes transparentes e não-transparentes da imagem), define o
  // formato da janela conforme a região atual (hRgn).
  SetWindowRgn(hWnd, hRgn, TRUE);
}
```

Listagem 7.3: Criação de skin para uma janela.

O arquivo *prog07-2.cpp* do CD-ROM contém a implementação completa de um programa que utiliza a função `WindowSkin()` e que aplica as duas modificações (para programas com janelas não-retangulares) discutidas nessa seção. A saída desse programa pode ser vista na Figura 7.2: o programa tem o formato de um boneco de massinha, que na figura está sobrepondo o Microsoft Visual C++ .Net (que no momento estava com o código do programa aberto).

Figura 7.2: Programa com janela não-retangular.

Nesse capítulo, vimos como criar e utilizar regiões, incluindo a combinação de duas ou mais regiões e a criação de janelas não-retangulares – muito utilizadas em programas onde o usuário pode definir o formato da janela através de bitmaps. Veremos a seguir como utilizar sons e timers em nossos programas.

Capítulo 8 – Sons e Timers

Para a criação de programas multimídia, músicas e efeitos sonoros são essenciais. Através da Win32 API, usamos funções para reproduzir sons do tipo *wave* (arquivos com extensão *.wav*) e músicas do tipo MIDI (arquivos com extensão *.mid*). Podemos utilizar outros recursos multimídia também, tais como reproduzir um vídeo, tocar as faixas de um CD de áudio ou mesmo gravar sons, porém esses assuntos não serão vistos nesse livro.

Reproduzindo sons

A maneira mais rápida e simples de reproduzir um som wave é utilizando a função `PlaySound()`. Essa função pode reproduzir arquivos wave do disco ou um som wave que está em um arquivo de recursos (que geralmente é anexado ao programa executável em tempo de compilação), porém, não suporta a reprodução de múltiplos sons ao mesmo tempo.

```
BOOL PlaySound(
  LPCSTR pszSound, // arquivo ou recurso wave
  HMODULE hmod, // instância do executável que contém o recurso (som)
  DWORD fdwSound // flags
);
```

Para reproduzir um arquivo wave, devemos enviar o valor `SND_FILENAME` para o último parâmetro (`DWORD fdwSound`) e informar o nome do arquivo no primeiro parâmetro (`LPCSTR pszSound`). Nesse caso, o parâmetro `HMODULE hmod` deve receber `NULL`, visto que o som será carregado de um arquivo.

Para reproduzir um recurso wave, enviamos o valor `SND_RESOURCE` para o último parâmetro e informamos o identificador do recurso no primeiro (utilizando o a macro `MAKEINTRESOURCE()`). O segundo parâmetro deve receber a instância do programa que contém o recurso (uma variável do tipo `HINSTANCE`).

Para parar a reprodução de um som, chamamos a função com o valor `NULL` no parâmetro `pszSound`.

A função retorna `TRUE` quando a mesma for executada corretamente ou `FALSE` caso contrário (o arquivo ou recurso wave enviado para a função não foi encontrado, por exemplo).

O parâmetro `fdwSound` da função ainda recebe informações (através de uma combinação `OR`) de como o som deve ser reproduzido. Consulte a Tabela 8.1 para a descrição dos principais valores desse parâmetro.

Valor	Descrição
`SND_ASYNC`	A função retorna o controle para o programa imediatamente após o som começar a ser reproduzido.
`SND_FILENAME`	O parâmetro `pszSound` recebe o nome de um arquivo.
`SND_LOOP`	A função reproduz o som repetidamente, até que a função seja chamada novamente com o valor `NULL` no parâmetro `pszSound`. O valor `SND_ASYNC` também deve ser informado nesse caso.
`SND_NOWAIT`	Se o driver de som estiver ocupado, retorna imediatamente sem reproduzir o som.
`SND_RESOURCE`	O parâmetro `pszSound` recebe um identificador de recurso e o parâmetro `hmod` indica a instância que contém o recurso.
`SND_SYNC`	O controle para o programa é retornado somente após o som ter sido reproduzido por completo.

Tabela 8.1: Como `PlaySound()` deve reproduzir um som.

A biblioteca Windows Multimedia

Para que a função `PlaySound()` (e qualquer outra função multimídia) possa ser utilizada, é necessário incluir o arquivo de cabeçalho *mmsystem.h* no código-fonte do programa e também a biblioteca *winmm.lib*.

Para incluir o arquivo de cabeçalho, basta inserir a diretiva `#include <mmsystem.h>` no começo do código-fonte do seu programa. Para incluir a biblioteca *winmm.lib*, é necessário modificar as configurações do projeto, dentro do seu compilador. Ou então, podemos "adicionar" a biblioteca no próprio código-fonte, através da diretiva `#pragma comment(lib, "winmmb.lib")`. O uso dessa diretiva faz com que o compilador procure a biblioteca *winmm.lib* para ser linkada ao arquivo executável final.

MCI

Para trabalharmos com multimídia através da Win32 API, podemos utilizar a Interface de Controle de Mídia (Media Control Interface, ou simplesmente MCI). Essa interface nos fornece comandos padrões para o acesso e uso de diversos dispositivos multimídia, tais como som, música e vídeo.

O uso da MCI pode ser feito através de strings de comando ou por mensagens de comando. Quando usamos strings de comando, enviamos uma string de determinada ação para a função `mciSendString()`. No caso de mensagens de comando, usamos constantes e estruturas para manipulação da MCI através da função `mciSendCommand()`.

Nota: o sistema operacional converte as strings de comando para mensagens de comando antes de enviá-las para o processamento do driver MCI. Estudaremos como trabalhar com a MCI através de mensagens de comando.

Vamos supor que precisamos reproduzir um arquivo wave em disco, chamado *teste.wav*, e que queremos usar strings de comando. O código para essa ação ficaria como na Listagem 8.1.

```
// Reproduz o arquivo wave “teste.wav” em disco
mciSendString("play teste.wav", NULL, 0, NULL);
```

Listagem 8.1: Uso de strings de comando MCI.

Para reproduzir o mesmo arquivo wave, mas agora utilizando mensagens de comando, usaríamos o código da Listagem 8.2.

```
// Abre arquivo wave “teste.wav” do disco
MCI_OPEN_PARMS mciOpenParms;
mciOpenParms.lpstrDeviceType = "waveaudio";
mciOpenParms.lpstrElementName = "teste.wav";
mciOpenParms.dwCallback = (DWORD)hWnd;
mciSendCommand(0,     MCI_OPEN,     MCI_OPEN_TYPE     |     MCI_OPEN_ELEMENT,
(DWORD)&mciOpenParms);
MCIDEVICEID mciDeviceID = mciOpenParms.wDeviceID;

// Reproduz o arquivo wave “teste.wav” em disco
MCI_PLAY_PARMS mciPlayParms;
mciPlayParms.dwCallback = (DWORD)hWnd;
mciSendCommand(mciDeviceID, MCI_PLAY, NULL, (DWORD)&mciPlayParms);
```

Listagem 8.2: Uso de mensagens de comando MCI.

Como é possível notar, há uma grande diferença entre o uso de strings e mensagens de comando, pelo fato que as mensagens de comando utilizam estruturas que devem ser preenchidas antes de chamarmos a função `mciSendCommand()`. Embora pareça complicado, o uso das mensagens de comando é simples e comum na programação Windows, além de não haver a conversão como no caso do uso de strings.

O protótipo das funções de strings e mensagens de comando são listadas e explicadas a seguir.

```
MCIERROR mciSendString(
  LPCTSTR lpszCommand, // string de comando
  LPTSTR lpszReturnString, // buffer com informação de retorno
  UINT cchReturn, // tamanho do buffer, em caracteres
  HANDLE hwndCallback // identificador de um callback
);
```

A função `mciSendString()` recebe no primeiro parâmetro uma string contendo o comando MCI que será executado. Caso seja necessário uma informação de retorno da função, enviamos um ponteiro para o segundo parâmetro (`lpszReturnString`) e indicamos o tamanho do buffer desse ponteiro no terceiro parâmetro (`cchReturn`). Quando não há a necessidade de informação de retorno, basta enviar `NULL` para `lpszReturnString` e zero para `cchReturn`. O último parâmetro deve receber um identificador de uma função callback se a flag `notify` for especificado na string de comando, caso contrário, passamos `NULL`. A função retorna zero quando executada corretamente ou um erro caso contrário.

Nota: sugiro uma consulta ao MSDN para obter as strings de comando e códigos de erros.

```
MCIERROR mciSendCommand(
  MCIDEVICEID IDDevice, // identificador do dispositivo
  UINT uMsg, // mensagem de comando
  DWORD fdwCommand, // flags da mensagem
  DWORD_PTR dwParam // estrutura que contém parâmetros da mensagem
);
```

A função `mciSendCommand()` recebe no primeiro parâmetro o identificador do dispositivo que receberá a mensagem. O segundo parâmetro recebe a mensagem que será executada (veja Tabela 8.2 para as principais mensagens). O terceiro parâmetro, `DWORD fdwCommand`, recebe flags das mensagens. Note que para cada mensagem/comando específico, existem

diferentes tipos de flags. O último parâmetro recebe um ponteiro para a estrutura que contém os parâmetros de configuração da mensagem de comando. A função retorna zero quando executada corretamente ou um erro caso contrário.

Nota: quando a mensagem `MCI_OPEN` é enviada para a função `mciSendCommand()`, o primeiro parâmetro dessa função deve receber `NULL`. Ainda, essa mensagem obtém o identificador do dispositivo que deve ser enviado para o primeiro parâmetro quando queremos executar outros comandos MCI.

Valor	Descrição
`MCI_CLOSE`	Libera acesso ao dispositivo ou arquivo.
`MCI_OPEN`	Inicia um dispositivo ou abre um arquivo.
`MCI_PAUSE`	Pausa o dispositivo.
`MCI_PLAY`	Começa reprodução dos dados do dispositivo.
`MCI_RESUME`	Despausa o dispositivo.
`MCI_SEEK`	Muda a posição atual dos dados do dispositivo.
`MCI_STOP`	Pára a reprodução dos dados do dispositivo.

Tabela 8.2: Principais mensagens de comando.

Reprodução de múltiplos sons

Conforme visto no começo desse capítulo, a função `PlaySound()` não permite a reprodução de dois ou mais sons simultaneamente. Temos que esperar o primeiro som terminar para então reproduzirmos o segundo ou paramos a reprodução do primeiro para iniciar o segundo. Com o uso da MCI, essa restrição é eliminada, pois criamos um identificador para cada dispositivo (som) que utilizamos no programa.

Para reproduzir um som, devemos proceder com as seguintes etapas:

1) Definir o tipo de dispositivo e arquivo a ser aberto, configurando a estrutura `MCI_OPEN_PARMS`;
2) Iniciar o dispositivo, obtendo seu identificador;
3) Reproduzir os dados do dispositivo.

A estrutura `MCI_OPEN_PARMS` define as configurações do dispositivo, tais como o tipo do dispositivo, o nome do arquivo que será aberto para reprodução e o identificador do dispositivo.

```
typedef struct {
  DWORD_PTR dwCallback;
  MCIDEVICEID wDeviceID;
  LPCSTR lpstrDeviceType;
  LPCSTR lpstrElementName;
  LPCSTR lpstrAlias;
} MCI_OPEN_PARMS;
```

O membro `lpstrDeviceType` define o tipo de dispositivo. No caso de arquivos wave, atribuímos o valor "`waveaudio`" para ele. Enviamos o nome do arquivo que será reproduzido para o membro `lpstrElementName`. O membro `dwCallback` só é utilizado quando queremos que a mensagem `MCI_NOTIFY` seja processada por algum identificador de janela (que é atribuído a esse membro). `lpstrAlias` define um pseudônimo para o dispositivo e é opcional. O identificador do dispositivo fica armazenado em `wDeviceID`, somente após a mensagem `MCI_OPEN` for enviada para `mciSendCommand()`.

Após ter configurado a estrutura `MCI_OPEN_PARMS`, iniciamos o dispositivo enviando a mensagem `MCI_OPEN` para `mciSendCommand()`. Essas etapas podem ser realizadas da seguinte forma (Listagem 8.3):

```
// Armazena identificador do dispositivo
MCIDEVICEID mciDeviceID = NULL;

// ...

// Verifica se o dispositivo já foi inicializado
if(mciDeviceID == NULL)
{
  // Configura o dispositivo
  MCI_OPEN_PARMS mciOpenParms;
  mciOpenParms.lpstrDeviceType = "waveaudio";
  mciOpenParms.lpstrElementName = "teste.wav";

  // Abre o dispositivo
  mciSendCommand(NULL, MCI_OPEN, MCI_OPEN_TYPE | MCI_OPEN_ELEMENT,
(DWORD)&mciOpenParms);

  // Obtém identificador do dispositivo
  mciDeviceID = mciOpenParms.wDeviceID;
}

// ...
```

Listagem 8.3: Configurando `MCI_OPEN_PARMS` e inicializando um dispositivo.

Note que ao chamar a função `mciSendCommand()`, seu primeiro parâmetro recebe `NULL`, pois ainda não possuímos o identificador do dispositivo. A função ainda recebe dois flags, `MCI_OPEN_TYPE` e `MCI_OPEN_ELEMENT`, indicando, respectivamente, que o dispositivo a ser aberto é do tipo definido em `lpstrDeviceType` e que um arquivo foi especificado em `lpstrElementName`, ambos definidos na estrutura enviada para o último parâmetro da função (`(DWORD)&mciOpenParms`).

A etapa seguinte é reproduzir o som que foi carregado. Podemos fazer isso de duas maneiras: com ou sem o envio da mensagem `MM_MCINOTIFY` para a função callback de um identificador de janela. Quando queremos que o dispositivo envie a mensagem `MM_MCINOTIFY` após completar uma determinada ação (no caso, quando terminar de reproduzir o som), devemos configurar a estrutura `MCI_PLAY_PARMS`. Para a reprodução de um som, sem necessidade de enviar tal mensagem, basta enviar `MCI_PLAY` para a função `mciSendCommand()`.

```
typedef struct {
  DWORD_PTR dwCallback; // função callback de um identificador de janela que
irá processar a mensagem MM_MCINOTIFY
  DWORD dwFrom; // posição inicial do som
  DWORD dwTo; // posição final do som
} MCI_PLAY_PARMS;
```

Na Listagem 8.4, temos o código para reproduzir um som das duas maneiras citadas acima.

```
// Dispositivo já inicializado com um som wave
// Seu identificador foi obtido e armazenado em mciDeviceID
// ...

// Reproduz som com envio da mensagem MM_MCINOTIFY para a função callback
// hWnd, que irá processar a mensagem
MCI_PLAY_PARMS mciPlayParms;
mciPlayParms.dwCallback = (DWORD)hWnd;
mciSendCommand(mciDeviceID, MCI_PLAY, MCI_NOTIFY, (DWORD)&mciPlayParms);

// Reproduz som sem envio da mensagem MM_MCINOTIFY
mciSendCommand(mciDeviceID, MCI_PLAY, NULL, NULL);
```

Listagem 8.4: Reproduzindo um som wave.

Quando não precisamos mais de algum dispositivo MCI, devemos liberar o seu acesso. Para isso, usamos a mensagem de comando `MCI_CLOSE`. A Listagem 8.5 demonstra como fechar um dispositivo MCI.

```
// Pára a reprodução do dispositivo e libera o mesmo
mciSendCommand(mciDeviceID, MCI_STOP, NULL, NULL);
mciSendCommand(mciDeviceID, MCI_CLOSE, NULL, NULL);
```

```
mciDeviceID = NULL;
```

Listagem 8.5: Fechando um dispositivo MCI.

Nota: quando o programa é fechado, o sistema não libera automaticamente os dispositivos MCI que estiverem abertos. Devemos, portanto, incluir o código da Listagem 8.5 no processamento da mensagem `WM_CLOSE`, para evitarmos problemas de memória.

Reproduzindo músicas MIDI

Para a reprodução de músicas no formato MIDI, procedemos com as mesmas etapas necessárias para reproduzir um som wave. A única diferença é na configuração do dispositivo; ao invés de declararmos um dispositivo do tipo `"waveaudio"`, declaramos o dispositivo como `"sequencer"`. A Listagem 8.6 configura um dispositivo MIDI (carregando o arquivo *teste.mid*) e logo em seguida reproduz a música.

```
// Armazena identificador do dispositivo
MCIDEVICEID mciDeviceIDmidi = NULL;

// ...

// Verifica se o dispositivo já foi inicializado
if(mciDeviceIDmidi == NULL)
{
  // Configura o dispositivo
  MCI_OPEN_PARMS mciOpenParms;
  mciOpenParms.lpstrDeviceType = "sequencer"; // MIDI
  mciOpenParms.lpstrElementName = "teste.mid";
  // Abre o dispositivo
  mciSendCommand(NULL, MCI_OPEN, MCI_OPEN_TYPE | MCI_OPEN_ELEMENT,
(DWORD)&mciOpenParms);
  // Obtém identificador do dispositivo
  mciDeviceIDmidi = mciOpenParms.wDeviceID;
}

// Reproduz o dispositivo
mciSendCommand(mciDeviceIDmidi, MCI_PLAY, NULL, NULL);
```

Listagem 8.6: Configurando e reproduzindo dispositivo MIDI.

Nota: não é possível reproduzir duas ou mais músicas MIDI ao mesmo tempo. Lembre-se de liberar o acesso ao dispositivo quando o mesmo não for mais utilizado ou quando o programa for finalizado.

O arquivo *prog08-1.cpp* no CD-ROM contém o código-fonte de um programa que demonstra a reprodução de sons wave e músicas MIDI conforme estudado nesse capítulo.

Timers

Há momentos em que precisamos utilizar timers (temporizadores) em nossos programas – recursos que enviam mensagens `WM_TIMER` em um determinado intervalo de tempo para serem processadas pela função `WindowProc()` (ou uma função callback específica para timers – `TimerProc()`, como veremos mais a frente). Um exemplo do uso de timers é quando queremos mostrar a hora atual no programa, sendo atualizada a cada segundo. Seguiremos com esse exemplo para o estudo de timers.

Basicamente, o uso de timers é feito da seguinte forma: cria-se um timer com intervalo de n-milisegundos, executa-se um algoritmo a cada intervalo do timer (ou seja, quando o timer envia a mensagem `WM_TIMER` ou chama a função `TimerProc()`) e o destrói quando não for mais útil.

Para criar um timer, usamos a função `SetTimer()`.

```
UINT_PTR SetTimer(
  HWND hWnd, // identificador da janela associada ao timer
  UINT_PTR nIDEvent, // identificador do timer
  UINT uElapse, // intervalo do timer em milisegundos
  TIMERPROC lpTimerFunc // função callback de timers
);
```

O primeiro parâmetro da função recebe o identificador da janela que será associada ao timer, ou seja, a janela que receberá a mensagem `WM_TIMER` quando a contagem do timer se esgotar. O segundo parâmetro recebe um identificador do timer, que deve ser único para cada timer diferente. O valor desse parâmetro é um número inteiro; geralmente usamos um `#define` para utilizar uma constante ao invés de enviar números ao parâmetro. No terceiro parâmetro fornecemos o intervalo (em milisegundos) que o timer envia a mensagem `WM_TIMER` (ou que é processado pela função `TimerProc()`). O último parâmetro da função recebe um ponteiro para a função callback que processa os timers ou `NULL` para que o timer envie a mensagem `WM_TIMER` à janela associada. A função retorna um valor diferente de zero quando sua execução é bem-sucedida ou zero em caso negativo.

Para o exemplo do programa que mostra a hora na tela, criamos o timer da seguinte maneira (Listagem 8.7):

```
#define IDT_TIMER1    1000 // Identificador do timer para WM_TIMER
#define IDT_TIMER2    1001 // Identificador do timer para a hora
```

```
// Cria um timer que envia a mensagem WM_TIMER à WindowProc() em um
// intervalo de 300 milisegundos – último parâmetro recebeu NULL
SetTimer(hWnd, IDT_TIMER1, 300, NULL);

// O timer de hora será processado em uma função callback de timers,
// TimerProc(), a cada 1000 milisegundos, ou seja, a cada 1 segundo
SetTimer(hWnd, IDT_TIMER, 1000, (TIMERPROC)TimerProc);
```

Listagem 8.7: Criando um timer.

Assim que `SetTimer()` é executada, um timer é criado e ativado. No exemplo acima, criamos dois timers: um que envia a mensagem `WM_TIMER` à `WindowProc()` em um intervalo de 300 milisegundos e outro que é processado pela função `TimerProc()` a cada 1 segundo. Para o intervalo do timer `IDT_TIMER1`, processamos a mensagem `WM_TIMER`; para o intervalo do timer `IDT_TIMER2`, codificamos a função callback `TimerProc()`. Veja a Listagem 8.8:

```
//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  switch(uMsg)
  {
    // ...

    case WM_TIMER: // Algum timer foi ativado
    {
      // Identificador do timer ativado é IDT_TIMER1
      if(wParam == IDT_TIMER1)
      {
        // Executa código a cada intervalo do IDT_TIMER1 (300 milisegundos)
      }

      return(0);
    } break;

    // ...
  }

  // ...
}

//------------------------------------------------------------------------
// TimerProc() -> Processa os eventos de timer
//------------------------------------------------------------------------
VOID CALLBACK TimerProc(HWND hWnd, UINT uMsg, UINT_PTR idEvent, DWORD
dwTime)
{
  // Armazena hora atual
  char cHora[9] = { 0 };

  // Identificador do timer ativado é IDT_TIMER2
  if(idEvent == IDT_TIMER2)
  {
```

```
    // Obtém hora atual (a cada segundo) e mostra na área cliente
    _strtime(cHora);
    HDC hDC = GetDC(hWnd);
    TextOut(hDC, 8, 28, cHora, strlen(cHora));
    ReleaseDC(hWnd, hDC);
  }
}
```

Listagem 8.8: Intervalo dos timers.

Na função `WindowProc()`, o parâmetro `WPARAM wParam` informa o identificador do timer que enviou a mensagem `WM_TIMER`. Verificamos se o identificador é `IDT_TIMER1` – em caso positivo, executamos um bloco de código.

Repare que a função callback `TimerProc()` é muito semelhante à `WindowProc()`. O parâmetro `HWND hWnd` contém o identifcador da janela associada ao timer. No parâmetro `UINT uMsg`, a função recebe a mensagem `WM_TIMER`. O parâmetro `UINT_PTR idEvent` contém o identificador do timer que foi ativado e `DWORD dwTime` indica os milisegundos que se passaram desde que o sistema foi iniciado. Note ainda que a função não retorna valor.

Dentro da função `TimerProc()`, verificamos se o timer ativado é `IDT_TIMER2` (timer para atualizar a hora) – em caso positivo, chamamos a função `_strtime()` para que ela converta a hora atual para uma string, que é armazenada em `cHora`. Em seguida, mostramos o conteúdo dessa string na área cliente através da `TextOut()`.

Nota: o intervalo dos timers nem sempre é regular (é uma aproximação do que foi especificado em `UINT uElapse`), pois a mensagem `WM_TIMER` (de prioridade baixa) é enviada para a fila de mensagens do programa somente quando não houver mensagens de prioridade alta na fila.

Depois que os timers são utilizados e não são mais necessários (ou quando o programa é fechado), devemos destruí-los, usando a função `KillTimer()`.

```
BOOL KillTimer(
  HWND hWnd, // identificador da janela
  UINT_PTR uIDEvent // identificador do timer
);
```

A função recebe o identificador da janela (`hWnd`) a qual o timer está associado e o identificador do timer que será destruído (`uIDEvent`). A função

retorna um valor diferente de zero quando bem sucedida ou zero caso contrário.

No exemplo do horário, podemos destruir os dois timers (`IDT_TIMER1` e `IDT_TIMER2`) através do código da Listagem 8.9.

```
//------------------------------------------------------------------------
// WindowProc() -> Processa as mensagens enviadas para o programa
//------------------------------------------------------------------------
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
  switch(uMsg)
  {
    // ...

    case WM_CLOSE: // Janela foi fechada
    {
      // Desativa/destrói os timers (para que não haja falta de recursos)
      KillTimer(hWnd, IDT_TIMER1);
      KillTimer(hWnd, IDT_TIMER2);

      // Destrói a janela
      DestroyWindow(hWnd);

      return(0);
    } break;

    // ...
  }

  // ...
}
```

Listagem 8.9: Destruindo timers.

Nota: poderíamos ter chamado a função `KillTimer()` em outro lugar do programa (por exemplo, na mensagem `WM_COMMAND`, quando um botão é pressionado), mas lembre-se que sempre devemos destruir os timers quando o programa é fechado; caso contrário, os recursos de timers continuarão na memória, mesmo após finalizado o programa.

Nesse capítulo, vimos como trabalhar com sons wave e músicas MIDI. Vimos também como utilizar timers, através da criação de um programa (arquivo *prog08-2.cpp* no CD-ROM) que mostra a hora atual na tela, atualizando a informação a cada segundo. No capítulo a seguir, estudaremos a manipulação de arquivos através da Win32 API e como trabalhar com o registro do Windows.

Capítulo 9 – Arquivos e Registro

A grande maioria dos programas Windows utilizam algum meio para armazenar e recuperar informações; esse meio, em geral, são arquivos gravados em disco (já trabalhamos com arquivos de imagem no formato *.bmp* em um capítulo anterior e nesse veremos como trabalhar com arquivos de texto). O modo básico de como a Win32 API trabalha com arquivos é bem semelhante com o ANSI-C, embora possua muito mais opções (tornando-se um sistema mais complexo).

Para armazenar configurações de um programa (como versão do programa instalado, nome do usuário que registrou o programa, entre outras informações), até o lançamento do Windows 95, o normal era utilizar arquivos com extensão *.ini* (de *initialization*, ou inicialização). Hoje em dia, esse tipo de arquivo praticamente foi extinto, e agora o comum é utilizar o Registro do Windows, que será o segundo assunto desse capítulo.

Nota: nesse capítulo, será adotada uma abordagem diferente em relação aos outros. As explicações serão mais resumidas, dando ênfase ao código utiizado para as ações básicas de arquivos, tais como abrir, fechar e excluir arquivos e escrita/leitura. Ainda, parte do código contém algumas funções que não foram vistas no decorrer da leitura – por exemplo, a criação de botões na área cliente. Adotei esse método nesse capítulo pois acredito que você, leitor, já esteja com um bom conhecimento sobre programação Win32 API e que esteja apto a estudar certas funções mais a fundo através de outras fontes de referência.

Criando e abrindo arquivos

Os arquivos podem ser criados e abertos utilizando uma função da Win32 API, chamada `CreateFile()`.

```
HANDLE CreateFile(
  LPCTSTR lpFileName, // nome do arquivo
  DWORD dwDesiredAccess, // modo de acesso
  DWORD dwShareMode, // modo de compartilhamento
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // atributos de segurança
  DWORD dwCreationDisposition, // como criar/sobrepor o arquivo
  DWORD dwFlagsAndAttributes, // atributos do arquivo
  HANDLE hTemplateFile // identificador de arquivo temporário
);
```

A Listagem 9.1 contém um exemplo de como abrir o arquivo *dados.txt* para escrita/leitura, que encontra-se no mesmo diretório que o programa, não compartilhando seu acesso (ou seja, enquanto o identificador do arquivo – retornado pela função – estiver sendo usado, nenhum outro programa terá acesso ao arquivo). Se o arquivo não existir, a função criará o arquivo (`OPEN_ALWAYS`) especificado em `lpFileName`. O identificador para trabalhar com o arquivo é retornado pela função (no exemplo, é salvo em `HANDLE hFile`).

```
HANDLE hFile = NULL;

// Se arquivo ainda não foi aberto
if(hFile == NULL)
{
  // Abre/cria arquivo
  hFile = CreateFile("dados.txt", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

  // Verifica se identificador do arquivo é inválido
  if(hFile == INVALID_HANDLE_VALUE)
    MessageBox(hWnd, "Erro ao abrir/criar arquivo.", "Erro!", MB_ICONERROR |
MB_OK);
  else // Identificador válido, arquivo aberto
  {
    MessageBox(hWnd, "Arquivo aberto/criado.", "Aviso!", MB_ICONINFORMATION
| MB_OK);
    SendMessage(hWnd,      WM_SETTEXT,      (WPARAM)0,      (LPARAM)"Arquivo
aberto/criado.");

    // Cria botão na área cliente
    HWND hWndBotao = CreateWindowEx(WS_EX_CLIENTEDGE, "BUTTON", "Clique
aqui!", WS_CHILD | WS_VISIBLE, 10, 10, 100, 30, hWnd, (HMENU)IDC_BOTAO,
NULL, NULL);
  }
}
```

Listagem 9.1: Criando e abrindo arquivos, criação de botão na área cliente.

Note que, se o arquivo é válido, o trecho acima cria um botão (`HWND hWndBotao`) na área cliente do programa com a ID `IDC_BOTAO`. Clicando nesse botão (Figura 9.1), o programa enviará uma mensagem `WM_COMMAND` para a fila de mensagens com a ID do botão no parâmetro `LOWORD(wParam)` e com o identificador do botão no parâmetro `lParam`.

Dica: a definição de objetos como botões e caixa de texto é parecida com as macros utilizadas em arquivos de recursos (Capítulo 3), porém utilizamos a função `CreateWindowEx()`, que retorna um identificador de janela (visto que tais objetos são considerados janelas pelo Windows).

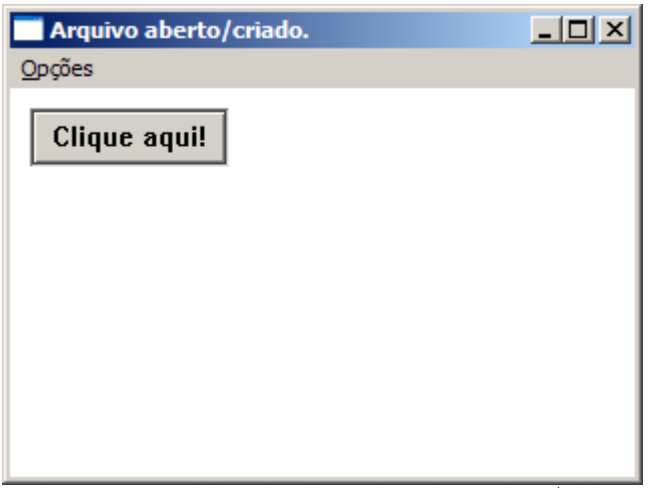

Figura 9.1: Botão criado na área cliente após arquivo ser aberto/criado.

Fechando arquivos

Para fechar um arquivo, liberando a memória do seu identificador (e o acesso ao arquivo para outros programas, dependendo do modo de compartilhamento em que o mesmo foi aberto), chamamos a função `CloseHandle()`, que recebe o identificador do arquivo como parâmetro.

```
BOOL CloseHandle(
  HANDLE hObject  // identificador do arquivo
);
```

A Listagem 9.2 mostra o uso da função para fechar o arquivo aberto anteriormente.

```
// Se o arquivo está aberto, fecha
if(hFile)
{
  CloseHandle(hFile);
  hFile = NULL;
  MessageBox(hWnd, "Arquivo fechado.", "Aviso", MB_ICONINFORMATION | MB_OK);
}
```

Listagem 9.2: Fechando arquivos.

Escrita em arquivos

Para escrever alguma informação no arquivo aberto, utilizamos a função `WriteFile()`.

```
BOOL WriteFile(
  HANDLE hFile, // identificador do arquivo
  LPCVOID lpBuffer, // buffer de dados
  DWORD nNumberOfBytesToWrite, // número de bytes a serem escritos
  LPDWORD lpNumberOfBytesWritten, // número de bytes escritos
  LPOVERLAPPED lpOverlapped // buffer de sobreescrita
);
```

O trecho de código da Listagem 9.3 escreve o conteúdo contido na caixa de texto do programa no arquivo *dados.txt*, que foi aberto na Listagem 9.1.

```
if(hFile != INVALID_HANDLE_VALUE) // Arquivo está aberto
{
  // Obtém quantidade de caracteres da caixa de texto
  int tam = GetWindowTextLength(hWndTexto);

  // Obtém conteúdo da caixa de texto, alocando memória para os dados
  // (alocação no modo Windows)
  LPSTR lpstrBuffer = (LPSTR)GlobalAlloc(GPTR, tam + 1);
  GetWindowText(hWndTexto, lpstrBuffer, tam + 1);

  // Armazena quantos bytes foram escritos no arquivo
  DWORD dwBytesEscritos;

  // Modifica posição do arquivo para o início
  SetFilePointer(hFile, NULL, NULL, FILE_BEGIN);

  // Grava conteúdo da caixa de texto no arquivo
  WriteFile(hFile, lpstrBuffer, tam, &dwBytesEscritos, NULL);

  // Define fim do arquivo
  SetEndOfFile(hFile);

  // Libera memória dos dados
  GlobalFree(lpstrBuffer);

  MessageBox(hWnd, "Conteúdo escrito no arquivo.", "Erro!",
MB_ICONINFORMATION | MB_OK);
}
else // Arquivo não foi aberto
  MessageBox(hWnd, "Erro ao escrever no arquivo.", "Erro!", MB_ICONERROR |
MB_OK);
```

Listagem 9.3: Escrita em arquivos.

Nesse trecho de código, modificamos a posição do arquivo para o início (`SetFilePointer()`), indicando que os dados serão gravados a partir do início do arquivo. Depois de gravarmos o conteúdo da caixa de texto no arquivo (`WriteFile()`), definimos o fim do arquivo (`SetEndOfFile()`), para que qualquer outro dado que estava no arquivo, antes da sua modificação, seja descartado, não havendo assim duplicação de dados.

Leitura em arquivos

Depois de aberto o arquivo, podemos acessar seu conteúdo através da função `ReadFile()`.

```
BOOL ReadFile(
  HANDLE hFile, // identificador do arquivo
  LPVOID lpBuffer, // buffer de dados
  DWORD nNumberOfBytesToRead, // número de bytes a serem lidos
  LPDWORD lpNumberOfBytesRead, // número de bytes lidos
  LPOVERLAPPED lpOverlapped // buffer de sobreescrita
);
```

O trecho de código da Listagem 9.4 processa a mensagem `WM_COMMAND`, que é enviada quando o botão criado na Listagem 9.1 é pressionado pelo usuário. O código destrói o botão da área cliente e em seguida cria uma caixa de texto, onde o conteúdo do arquivo lido é copiado.

```
// ...
case WM_COMMAND: // Item do menu, tecla de atalho ou controle ativado
{
  // Verifica bit menos significativo de wParam (ID's)
  switch(LOWORD(wParam))
  {
    // ...
    case IDC_BOTAO:
    {
      // Destrói botão
      if(hWndBotao)
      {
        DestroyWindow(hWndBotao);
        hWndBotao = NULL;
      }

      // Cria caixa de texto para o usuário entrar com os dados
      hWndTexto = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "", WS_CHILD |
WS_VISIBLE | ES_MULTILINE | WS_HSCROLL | WS_VSCROLL, 0, 0, 312, 194, hWnd,
NULL, NULL, NULL);

      if(hFile != INVALID_HANDLE_VALUE) // Arquivo está aberto
      {
        // Obtém tamanho do arquivo
        int tam = GetFileSize(hFile, NULL);

        // Obtém conteúdo do arquivo, alocando memória para os dados
        // (alocação no modo Windows)
        LPSTR lpstrBuffer = (LPSTR)GlobalAlloc(GPTR, tam + 1);

        // Armazena quantos bytes foram escritos no arquivo
        DWORD dwBytesEscritos;

        // Lê conteúdo do arquivo
        ReadFile(hFile, lpstrBuffer, tam, &dwBytesEscritos, NULL);
```

```
        // Coloca conteúdo do arquivo na caixa de texto
        SetWindowText(hWndTexto, lpstrBuffer);

        // Libera memória dos dados
        GlobalFree(lpstrBuffer);

        MessageBox(hWnd,    "Conteúdo    lido    do    arquivo.",    "Aviso!",
MB_ICONINFORMATION | MB_OK);

        // Define cursor na caixa de texto
        SetFocus(hWndTexto);
      }
      else // Arquivo não foi aberto
        MessageBox(hWnd, "Erro ao ler o arquivo.", "Erro!", MB_ICONERROR |
MB_OK);
    } break;
  }

  // ...
  return(0);
} break;
// ...
```

Listagem 9.4: Leitura em arquivos, processamento de botão e criação de caixa de texto.

Excluindo arquivos

Para apagar um arquivo em disco, usamos a função `DeleteFile()`. Essa função recebe o nome do arquivo que será excluído, retornando um valor diferente de zero caso o arquivo seja apagado ou zero caso contrário.

```
BOOL DeleteFile(
  LPCTSTR lpFileName // nome do arquivo a ser excluído
);
```

A função pode falhar quando o arquivo a ser excluído não existe ou quando o mesmo está sendo utilizado pelo sistema, inclusive quando abrimos o arquivo pelo programa não o fechamos através da função `CloseHandle()`.

Na Listagem 9.5, vemos um exemplo de como deletar o arquivo *dados.txt* – o qual espera-se estar localizado no mesmo diretório do programa executável.

```
// Tenta excluir arquivo e mostra resultado para usuário
if(DeleteFile("dados.txt"))
  MessageBox(hWnd,  "Arquivo  excluído.",  "Aviso!",  MB_ICONINFORMATION  |
MB_OK);
else
  MessageBox(hWnd, "Arquivo não pode ser excluído.", "Erro!", MB_ICONERROR |
MB_OK);
```

Listagem 9.5: Excluindo arquivos.

No CD-ROM, você encontra o arquivo *prog09-1.cpp*, que contém o código-fonte completo dos trechos listados nesse capítulo. O programa possui um menu para abrir/criar um arquivo chamado *dados.txt*, salvar alterações, fechar o arquivo, excluir o arquivo e sair do programa. A Figura 9.2 mostra o programa em execução, com o arquivo já aberto (repare na caixa de texto, que foi criada em tempo real no processo do clique do botão `IDC_BOTAO`).

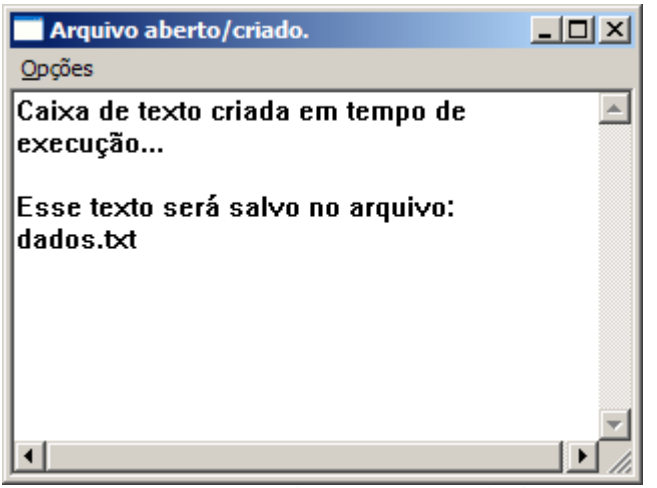

Figura 9.2: Programa-exemplo em execução com caixa de texto.

Registro do Windows

O registro do Windows é um banco de dados onde são armazenadas informações de configuração sobre o ambiente do sistema, dispensando o uso de arquivos de inicialização (*.ini*) que eram comuns em sistemas Windows 16-bit (até a versão 3.11). No registro, por exemplo, é possível armazenar as preferências de diferentes usuários, fazendo com que cada usuário trabalhe em um ambiente personalizado (escolha de temas, configuração de teclado, entre outros). Ainda, podemos associar determinado tipo de arquivo a ser aberto com um determinado programa, como também podemos incluir um programa para ser executado assim que o Windows é inicializado (ou seja, assim que um usuário faça login no sistema).

Podemos verificar o conteúdo e a estrutura do registro do Windows através do programa *regedit.exe*, localizado no diretório onde o Windows foi instalado. O registro trabalha com uma hierarquia de chaves (*keys*), subchaves (*subkeys*) e valores (*values*), conforme exemplo na Figura 9.3.

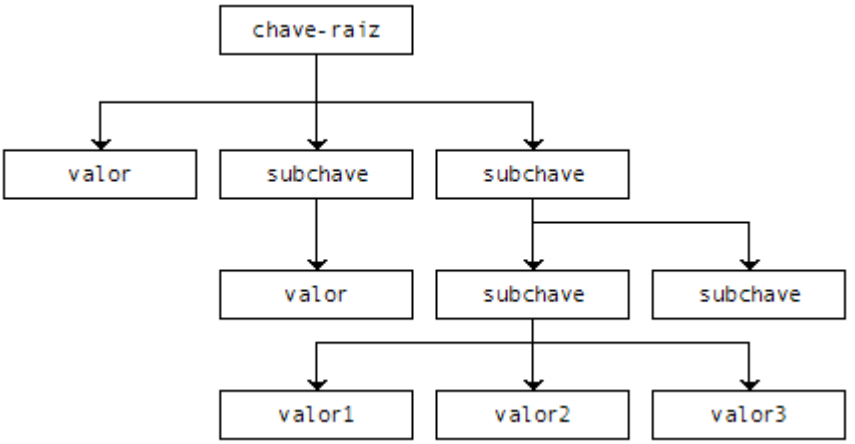

Figura 9.3: Hierarquia-exemplo do registro do Windows.

Dica: podemos fazer uma analogia com o Windows Explorer: as chaves são os diretórios/pastas, as subchaves são os subdiretórios/subpastas e os valores são os arquivos, inclusive pelo modo como o registro é apresentado ao usuário através do programa *regedit.exe*.

A estrutura do registro possui 5 chaves-raízes comuns nos sistemas Windows 32-bit (Figura 9.4), que são utilizadas conforme a Tabela 9.1.

Chave	Descrição
`HKEY_CLASSES_ROOT`	Parte da chave `HKEY_LOCAL_MACHINE`, contém definições de tipos de documentos, associação de arquivos e interface de comando (*shell*).
`HKEY_CURRENT_USER`	Ligação com `HKEY_USERS`, corresponde às configurações do usuário atual. Usuários padrões que não possuem configurações específicas utilizam as informações de `.DEFAULT` (da chave `HKEY_USERS`)
`HKEY_LOCAL_MACHINE`	Armazena configurações de hardware, protocolos de rede e classes de software.
`HKEY_USERS`	Utilizado para armazenar as preferências de cada usuário.
`HKEY_CURRENT_CONFIG`	Configuração selecionada na subchave de configuração de `HKEY_LOCAL_MACHINE`.

Tabela 9.1: Descrição das 5 principais chaves-raízes.

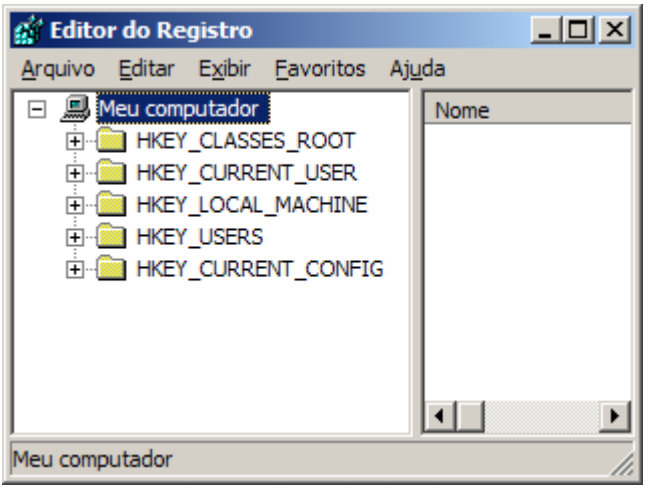

Figura 9.4: Chaves-raízes do registro.

Abrindo e fechando chaves do registro

Em primeiro lugar, para trabalharmos com o registro do Windows, devemos definir uma variável/identificador do tipo `HKEY`. É através desse identificador que iremos fazer a manipulação do registro.

Definido o identificador (por exemplo, `HKEY hKey = NULL;`), podemos começar a trabalhar com o registro. Vamos estudar primeiro como abrir o registro, ou melhor, como obter acesso ao registro utilizando a função `RegOpenKeyEx()`.

```
LONG RegOpenKeyEx(
  HKEY hKey, // chave-raíz ou identificador da chave de registro
  LPCTSTR lpSubKey, // subchave
  DWORD ulOptions, // reservado (deve receber zero)
  REGSAM samDesired, // modo de acesso
  PHKEY phkResult // ponteiro para identificador da chave de registro
);
```

Essa função abre uma chave e uma subchave do registro, identificada em `HKEY hKey` (que pode receber uma das chaves-raíz da Tabela 9.1 ou um identificador de chave) e `LPCTSTR lpSubKey`, respectivamente. O modo de acesso geralmente recebe `KEY_ALL_ACCESS` (permissão para ler e escrever), `KEY_READ` (permissão para ler) ou `KEY_WRITE` (permissão para escrever). No último parâmetro (`PHKEY phkResult`) enviamos um ponteiro para o identificador que criamos anteriormente – o qual será utilizado para manipular o registro.

Dica: todas as funções de registro retornam um valor `LONG`. Quando as funções forem executadas corretamente, o retorno é `ERROR_SUCCESS`.

Para fecharmos o acesso ao registro, usamos a função `RegCloseKey()`, que recebe apenas o identificador da chave que foi aberta.

```
LONG RegCloseKey(
  HKEY hKey // identificador da chave de registro
);
```

Nota: lembre-se que, dependendo das modificações no registro do Windows, o sistema pode não funcionar corretamente. A modificação das informações contidas no registro do Windows não é aconselhada; o risco e responsabilidade de modificá-lo fica totalmente ao leitor.

O arquivo *prog09-2.cpp* do CD-ROM contém o código-fonte de um programa que cria e edita chaves/valores no registro, além de fazer a leitura de valores e exclusão tanto das chaves quanto de valores. O código da Listagem 9.6 contém parte do código-fonte; no caso, utilizado para obter acesso ao registro, ler um valor de uma subchave e fechar o registro.

```
HKEY hKey = NULL; // Identificador do registro aberto (incluindo subchaves)
LONG lResultado = 0; // Verifica se ação com o registro foi bem sucedida

// Abre o registro para leitura, a partir da chave-raíz HKEY_LOCAL_MACHINE
// e na subchave SOFTWARE\___prog09-2
// Salvando o identificador desse local em hKey
lResultado = RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\___prog09-2", 0,
KEY_READ, &hKey);

// Registro aberto com sucesso
if(lResultado == ERROR_SUCCESS)
{
  // Obtém o valor "(Padrão)" da subchave
  // (PS: Padrão no caso é uma string NULL, ou seja, "")
  // Salva o valor em byteValor e
  // a quantidade de caracteres lidos em dwTamanho
  BYTE byteValor[255];
  DWORD dwTamanho;
  RegQueryValueEx(hKey, "", NULL, NULL, byteValor, &dwTamanho);

  // Mostra valor obtido na tela
  MessageBox(hWnd, (LPCSTR)byteValor, "Valor obtido no registro:",
MB_ICONINFORMATION | MB_OK);

  // Fecha o registro, usando o identificador hKey
  RegCloseKey(hKey);
}
else // Registro não pôde ser aberto
```

```
  MessageBox(hWnd, "Erro ao tentar abrir chave do registro.", "Erro!",
MB_ICONERROR | MB_OK);
```

Listagem 9.6: Acessando o registro, lendo valores e finalizando o acesso.

Note que, no trecho de código acima, caso a subchave `HKEY_LOCAL_MACHINE\SOFTWARE\___prog09-2` não exista, o programa mostrará o `MessageBox()` do `else`.

Criando e excluindo chaves do registro

Para criarmos uma chave (subchave) no registro, utilizamos a função `RegCreateKeyEx()`. Quando a chave indicada nessa função já existe, a mesma é aberta, como em `RegOpenKeyEx()`.

```
LONG RegCreateKeyEx(
  HKEY hKey, // chave-raiz ou identificador da chave de registro
  LPCTSTR lpSubKey, // subchave
  DWORD Reserved, // reservado (deve receber zero)
  LPTSTR lpClass, // classe da chave (recebe string null)
  DWORD dwOptions, // opções especiais
  REGSAM samDesired, // modo de acesso
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // herança de processos (null)
  PHKEY phkResult, // identificador da chave de registro
  LPDWORD lpdwDisposition // ação (cria nova chave ou abre chave existente)
);
```

Os parâmetros dessa função são parecidos com os da função `RegOpenKeyEx()`. Em `DWORD dwOptions`, especificamos o valor `REG_OPTION_NON_VOLATILE` para que as informações gravadas no registro sejam armazenadas mesmo que o computador seja desligado. A função modifica o valor da variável enviada no parâmetro `LPDWORD lpdwDisposition`, que indica se a chave indicada já existe (`REG_OPENED_EXISTING_KEY`) ou não (`REG_CREATED_NEW_KEY`).

A Listagem 9.7 cria uma nova subchave no registro e também modifica o valor padrão "*(Padrão)*"da subchave e adiciona um novo valor "*valor1*" à subchave.

```
HKEY hKey = NULL; // Identificador do registro aberto (incluindo subchaves)
LONG lResultado = 0; // Verifica se ação com o registro foi bem sucedida

// Cria uma subchave no registro para escrita, a partir da chave-raiz
// HKEY_LOCAL_MACHINE, na subchave SOFTWARE
// Salvando o identificador do novo local,
// (HKEY_LOCAL_MACHINE\SOFTWARE\___prog09-2) em hKey
// Em dwAcao, é armazenado se a subchave indicada já existia ou não
DWORD dwAcao;
lResultado = RegCreateKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\___prog09-2", 0,
NULL, REG_OPTION_NON_VOLATILE, KEY_WRITE, NULL, &hKey, &dwAcao);
```

```
// Ação no registro realizada com sucesso
if(lResultado == ERROR_SUCCESS)
{
  // Subchave não existia, criou uma nova
  if(dwAcao == REG_CREATED_NEW_KEY)
    MessageBox(...); // chave nova
  // Subchave já existia, apenas abriu a mesma
  if(dwAcao == REG_OPENED_EXISTING_KEY)
    MessageBox(...); // chave existente

  // Muda o valor "(Padrão)" da subchave, sendo do tipo REG_SZ
  // (string terminada com \0)
  lResultado = RegSetValueEx(hKey, "", 0, REG_SZ, (BYTE *)"{Adicionado por
prog09-2}\0", 26);
  if(lResultado == ERROR_SUCCESS)
    MessageBox(...); // modificado
  else
    MessageBox(...); // erro

  // Cria um novo valor, "valor1" dentro da subchave
  // HKEY_LOCAL_MACHINE\SOFTWARE\___prog09-2
  // O novo valor é do tipo número de 32-bit (REG_DWORD)
  // Note que se o valor "valor1" já existisse, o comando abaixo iria apenas
  // modificar seu valor, como feito na chamada à função acima
  // (que modificou o valor "(Padrão)")
  DWORD dwValor = 2005;
  lResultado = RegSetValueEx(hKey, "valor1", 0, REG_DWORD, (BYTE *)&dwValor,
sizeof(DWORD));
  if(lResultado == ERROR_SUCCESS)
    MessageBox(...); // modificado
  else
    MessageBox(...); // erro

  // Fecha o registro, usando o identificador hKey
  RegCloseKey(hKey);
}
else // Chave no registro não pôde ser criado
  MessageBox(hWnd, "Erro ao tentar criar chave no registro.", "Erro!",
MB_ICONERROR | MB_OK);
```

Listagem 9.7: Criando uma chave no registro e modificando valores.

Para excluirmos uma chave do registro, utilizamos `RegDeleteKey()`.

```
LONG RegDeleteKey(
  HKEY hKey, // chave-raíz ou identificador da chave de registro
  LPCTSTR lpSubKey // subchave que será excluída
);
```

O primeiro parâmetro da função recebe uma das chaves-raíz da Tabela 9.1 ou o identificador da chave de registro; o segundo parâmetro recebe o nome da subchave (que será excluída).

O trecho de código da Listagem 9.8 exclui a subchave criada pela Listagem 9.7.

```
HKEY hKey = NULL; // Identificador do registro aberto (incluindo subchaves)
LONG lResultado = 0; // Verifica se ação com o registro foi bem sucedida

// Abre o registro para escrita, a partir da chave-raiz HKEY_LOCAL_MACHINE
// e na subchave SOFTWARE\___prog09-2
// Salvando o identificador desse local em hKey
lResultado = RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\___prog09-2", 0,
KEY_WRITE, &hKey);

// Registro aberto com sucesso
if(lResultado == ERROR_SUCCESS)
{
  // Exclui a subchave HKEY_LOCAL_MACHINE\SOFTWARE\___prog09-2
  // e verifica se foi excluída ou não
  lResultado = RegDeleteKey(HKEY_LOCAL_MACHINE, "SOFTWARE\\___prog09-2");

  if(lResultado == ERROR_SUCCESS)
    MessageBox(...); // excluída
  else
    MessageBox(...); // erro

  // Fecha o registro, usando o identificador hKey
  RegCloseKey(hKey);
}
else // Registro não pôde ser aberto
  MessageBox(hWnd, "Erro ao tentar abrir chave do registro.", "Erro!",
MB_ICONERROR | MB_OK);
```

Listagem 9.8: Excluindo uma subchave do registro.

Gravando, obtendo e excluindo valores do registro

Para gravar informações nos valores do registro, podemos utilizar a função `RegSetValueEx()`; seu uso já foi visto na Listagem 9.7.

```
LONG RegSetValueEx(
  HKEY hKey, // identificador da chave de registro
  LPCTSTR lpValueName, // nome do valor ("" para valor padrão de uma chave)
  DWORD Reserved, // reservado (deve receber zero)
  DWORD dwType, // tipo do valor (REG_SZ, REG_DWORD, REG_BINARY, ...)
  CONST BYTE *lpData, // dados do valor
  DWORD cbData // tamanho dos dados do valor
);
```

Lembre-se que antes de gravar um valor, é necessário que o acesso ao registro tenha sido efetuado, inclusive onde o valor será gravado (chave ou subchave). Caso o nome do valor já exista, a função modifica o valor; caso ainda não exista, um novo valor é criado no registro.

Para obter informações dos valores do registro, utilizamos a função `RegQueryValueEx()`, também já vista, na Listagem 9.6.

```
LONG RegQueryValueEx(
  HKEY hKey, // chave-raíz ou identificador da chave de registro
  LPCTSTR lpValueName, // nome do valor ("" para valor padrão de uma chave)
  LPDWORD lpReserved, // reservado (deve receber zero)
  LPDWORD lpType, // tipo do valor (REG_SZ, REG_DWORD, REG_BINARY, ...)
  LPBYTE lpData, // ponteiro para os dados do valor
  LPDWORD lpcbData // ponteiro para o tamanho dos dados do valor
);
```

O conteúdo obtido do valor do registro é armazenado na variável que é enviada para o parâmetro `LPBYTE lpData` da função. O tamanho do conteúdo do valor é armazenado na variável enviada para o parâmetro `LPDWORD lpcbData`. Na Listagem 9.6, o conteúdo foi armazenado na variável `byteValor` e o tamanho, em `dwTamanho`.

A exclusão de valores do registro é feita através da função `RegDeleteValue()`, que simplesmente recebe o identificador da chave de registro que contém o local onde se encontra o valor a ser excluído e o nome do valor.

```
LONG RegDeleteValue(
  HKEY hKey, // identificador da chave de registro
  LPCTSTR lpValueName // nome do valor que será excluído
);
```

Um exemplo de como excluir um valor do registro pode ser visto na Listagem 9.9.

```
// ... Abre registro para escrita

// Registro aberto com sucesso
if(lResultado == ERROR_SUCCESS)
{
  // Exclui o valor "valor1" e verifica se foi excluído ou não
  lResultado = RegDeleteValue(hKey, "valor1");
  if(lResultado == ERROR_SUCCESS)
    MessageBox(...); // excluído
  else
    MessageBox(...); // erro

  // Fecha o registro, usando o identificador hKey
  RegCloseKey(hKey);
}
else // Registro não pôde ser aberto
  MessageBox(hWnd, "Erro ao tentar abrir chave do registro.", "Erro!",
MB_ICONERROR | MB_OK);
```

Listagem 9.9: Excluindo um valor do registro.

Conforme já mencionado, você pode encontar o arquivo *prog09-2.cpp* no CD-ROM que acompanha o livro, contendo a implementação de um programa que cria uma subchave no registro, modifica seu valor padrão, adiciona um novo valor, lê o valor, exclui o valor adicionado e a subchave criada. A Figura 9.5 mostra o registro do Windows após o programa criar a subchave e modificar os valores.

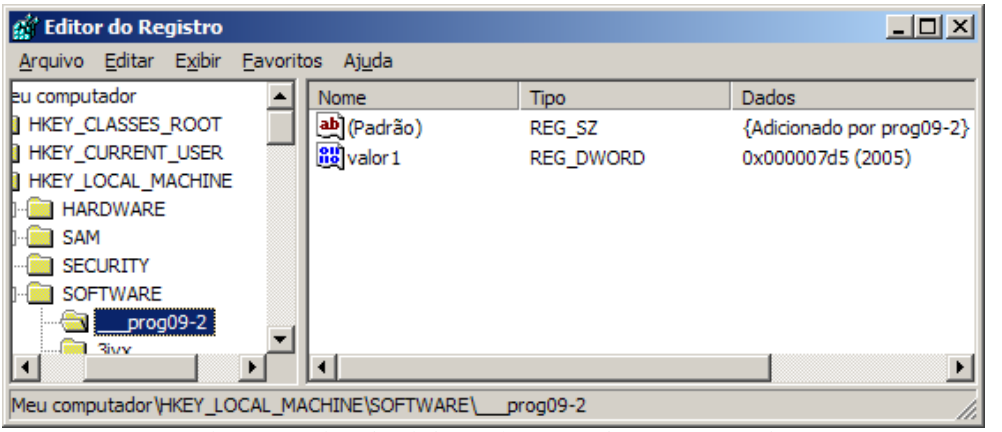

Figura 9.5: Registro do Windows após execução do programa-exemplo.

Nesse capítulo, aprendemos como utilizar arquivos e como manipular informações no registro do Windows, para que configurações e preferências do usuário sejam salvas e recuperadas futuramente. Com o término desse capítulo, chegamos ao final do livro, restando agora algumas palavras e considerações finais que me direciono a você, caro leitor.

Capítulo 10 – Considerações Finais

Espero que, após a leitura desse livro, você tenha obtido um ótimo conhecimento sobre como funciona a programação Windows, e que a experiência do aprendizado tenha sido agradável (embora muitas vezes acredito ter sido cansativa).

Meu objetivo nesse livro foi de introduzir a programação Windows com uma ênfase em multimídia, e, portanto, diversos assuntos não foram abordados pelo livro (mesmo os livros de mais de 1000 páginas não conseguem abordar a vasta quantidade de tópicos da programação Windows). Caso tenha interesse em outros assuntos (tais como DLL's, controles e caixas de diálogos comuns, *threads* e *atoms*), sugiro pesquisar as fontes informadas na Bibliografia. Após a leitura desse livro, acredito que você já esteja apto em pesquisar outras fontes de referência sem que as informações pareçam obscuras.

Conforme já mencionado na Introdução, a idéia de escrever esse livro surgiu em 2002, mas somente em 2003 é que comecei a trabalhar nele. Foram necessários cerca de 15 meses para que o objetivo inicial do meu trabalho fosse concluído; nesse período, fiquei sem escrever durante alguns meses, pois tive outros compromissos, como faculdade, família e projetos que acabaram atrasando a finalização do livro. Foi um trabalho duro, mas que valeu a pena, um desafio pessoal que consegui atingir. Mas... esse desafio não pára por aqui! Nesse tempo entre o início e o término do livro, tive a idéia de estender o livro para o desenvolvimento de jogos, que será uma continuação desse trabalho.

Agradeço a todos os leitores pelo interesse no meu trabalho e dou-lhes meus parabéns em ter aprendido a programação Windows e pelo auto-didatismo de cada um. Continuem assim!

Caso queira entrar em contato comigo, envie um e-mail para:
`kishimoto@tupinihon.com`

Ou visite o site:
`http://www.tupinihon.com`

- André Kishimoto

Bibliografia

CALVERT, Charles. *Teach Yourself Windows Programming in 21 Days*. Indianopolis: Sams. 1993.

CHANDLER, Damon; FÖTSCH, Michael. *Windows 2000 Graphics API Black Book*. Scottsdale: The Coriolis Group. 2001.

LAMOTHE, André. *Tricks Of The Windows Game Programming Gurus – Fundamentals Of 2D And 3D Game Programming*. Indianopolis: Sams. 1999.

LAMOTHE, André. *Windows Game Programming for Dummies*. New York: Hungry Minds. 1998.

MORRISON, Michael; WEEMS, Randy. *Windows 95 Game Developer's Guide Using the Game SDK*. Indianopolis: Sams, 1996.

MSDN. Microsoft Developers Network. Disponível em: *http://www.msdn.com*.

PAMBOUKIAN, Sérgio Vicente D. *Desenvolvendo Interfaces Gráficas Utilizando Win32 API e Motif – 2ª Edição*. São Paulo: Scortecci. 2003.

PETZOLD, Charles. *Programming Windows, Fifth Edition*. Redmond: Microsoft Press. 1998.

SIMON, Richard J. *Windows NT Win32 API SuperBible*. Corte Madera: The Waite Group. 1997.

STEVENS, Roger T. *Computer Graphics Dictionary*. Hingham: Charles River Media. 2002.

YOUNG, Michael J. *Introduction To Graphics Programming For Windows 95 – Vector Graphics Using C++*. Chestnut Hill: Academic Press. 1996.

YUAN, Feng. *Windows Graphics Programming – Win32 GDI and DirectDraw*. Upper Saddle River: Prentice Hall. 2001.

Índice Remissivo

Página em branco

Programação Windows:
C e Win32 API com ênfase em Multimídia

kishimoto@tupinihon.com
http://www.tupinihon.com

ISBN 85-906129-1-0
9 788590 612919