

## Windows Fundamentals

<http://www.functionx.com/win32/index.htm>

<http://www.functionx.com/visualc/index.htm>

### Introduction to windows      Overview

Microsoft Windows is an operating system that helps a person interact with a **personal computer** (PC). Programmers who want to create applications that people can use on MS Windows base them on a library called Win32.

Win32 is a library of data types, constants, functions, and classes (mainly structures) used to create applications for the Microsoft Windows operating system.

To create a basic application, you will first need a compiler that runs on a **Microsoft Windows** operating system. Although you can apply Win32 on various languages, including Pascal (namely Borland Delphi), we will use only one language. In reality the Win32 library is written in C, which is also the primary language of the Microsoft Windows operating systems. All of our programs will be written in C++. You will not see a difference between C and C++ in our programs. Although all of the structures of Win32 are mostly C objects, we will use Win32 as if it were a C++ library. This simply means that, whenever needed, we will apply C++ rules.

### Creating a Win32 Program

All Win32 programs primarily look the same and behave the same but, just like C++ programs, there are small differences in terms of creating a program, depending on the compiler you are using. For my part, I will be testing our programs on Borland C++ Builder, **Microsoft Visual C++ 6**, Dev-C++, and Microsoft Visual C++ .NET.

For a basic Win32 program, the contents of a Win32 program is the same. You will feel a difference only when you start adding some objects called resources.

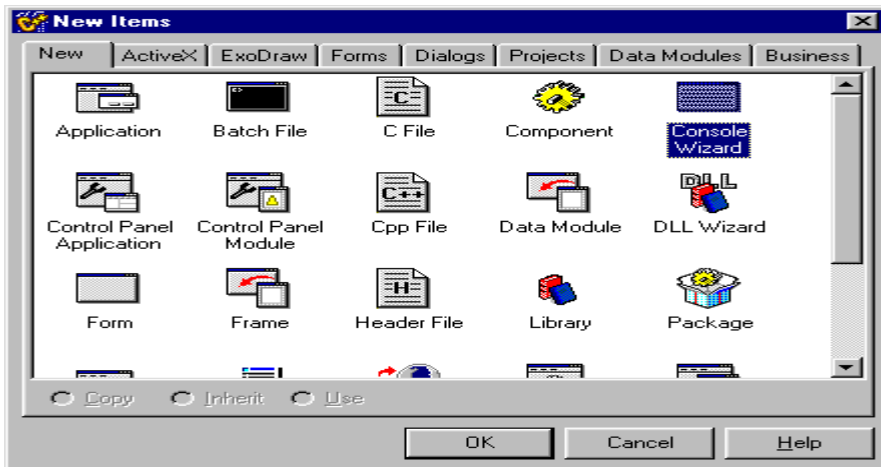
### Using Borland C++ Builder

To create a Win32 program using Borland C++ Builder, you must create a console application using the Console Wizard. You must make sure you don't select any option from the Console Wizard dialog box. After clicking OK, you are presented with a semi-empty file that contains only the inclusion of the **windows.h** library and the **WinMain()** function declaration. From there, you are ready.

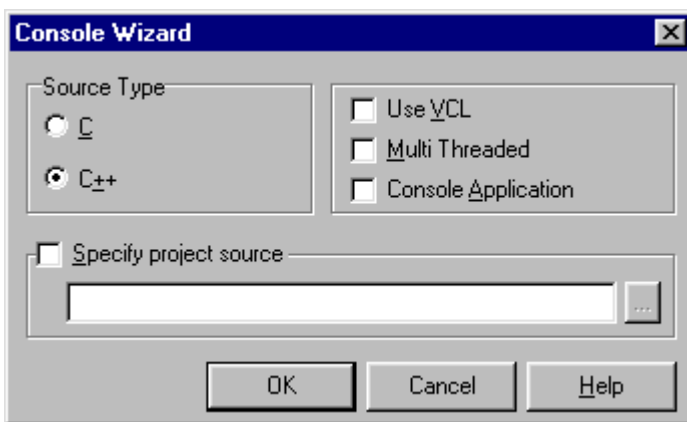
From most environments used, Borland C++ builder is the only one that provides the easiest, but also unfortunately the emptiest template to create a Win32 application. It doesn't provide any real template nor any help on what to do with the provided file. In defense of the Borland C++ Builder, as you will see with Microsoft Visual C++ and Dev-C++, the other environments may fill your file with statements you don't need, you don't like, or you don't want. Therefore, Borland C++ Builder provides this empty file so you can freely decide how you want to create you program and what you want to include in your program. This means that I agree with Borland C++ Builder providing you with an empty file because at least the syntax of the **WinMain()** function is provided to you.

### Practical Learning: Introducing Windows Programming

1. Start Borland C++ Builder (5 or 6 I don't care and it doesn't matter).
2. On the main menu, click File -> New... or File -> New -> Other...



3. In the New Items dialog box, click Console Wizard and click OK
4. In the Console Wizard, make sure that only the C++ radio button is selected:



5. Click OK.  
You are presented with a file as follows:

```
//-----
#include <windows.h>
#pragma hdrstop

//-----

#pragma argsused
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
return 0;
}

//-----
```

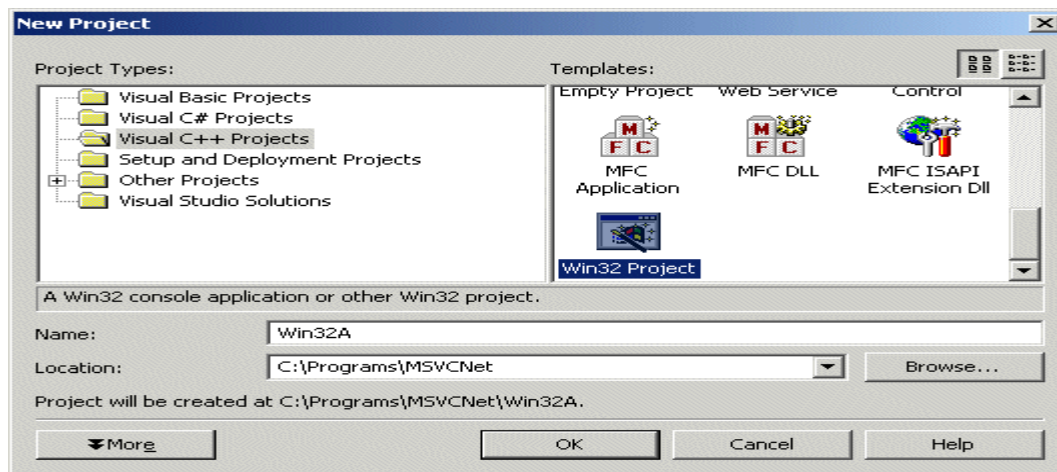
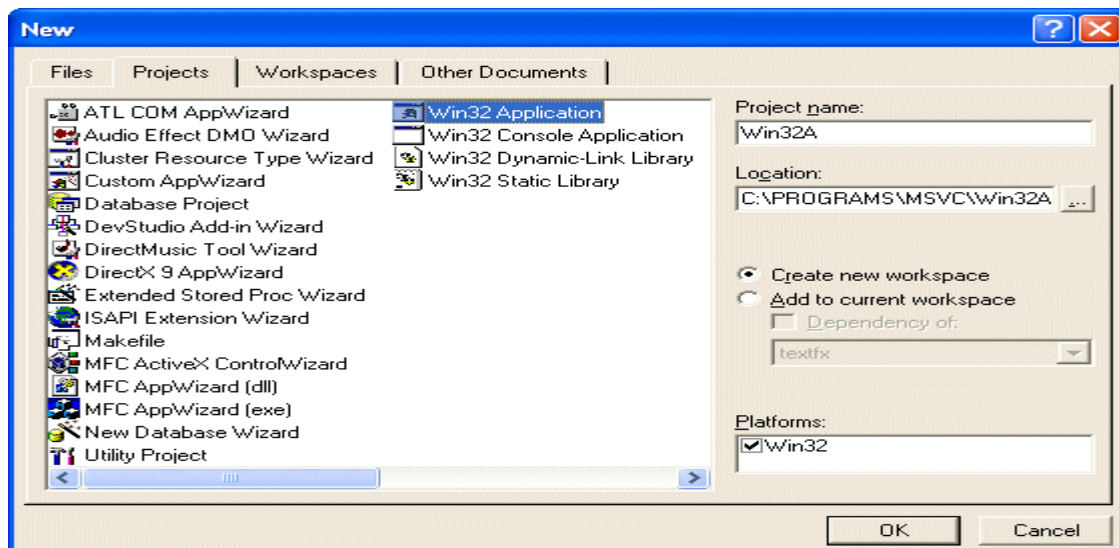
6. Save the application in a new folder named **Win32A**
7. Save the first file as **Main.cpp** and save the project as **SimpleWindow**

### Using Microsoft Visual C++

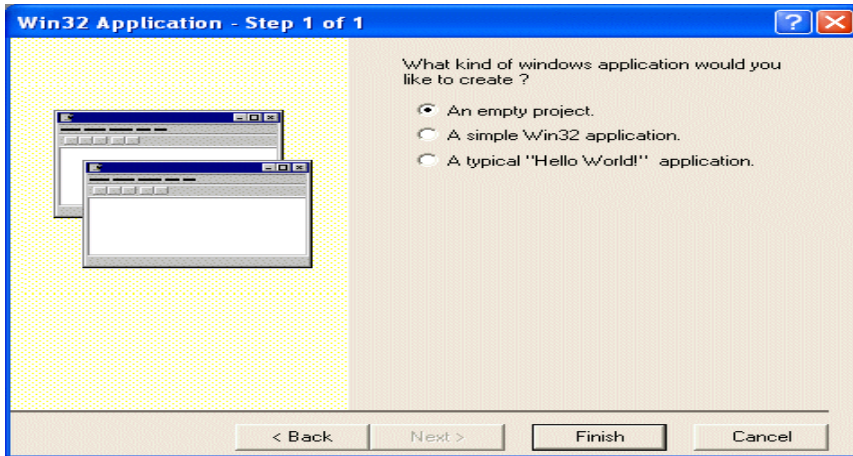
To create a Win32 application using Microsoft Visual C++, display the New (5 and 6 versions) or New Project (.Net version) dialog box and select Win32 application (5 and 6) or Win32 Project (.Net) item.

Microsoft Visual C++ provides the fastest and fairly most complete means of creating a Win32 application. For example it provides a skeleton application with all of the code a basic application would need. Because we are learning Win32, we will go the hard way, which consists of creating an application from scratch. In fact, this allows me to give almost (but not exactly) the same instructions as Borland C++ Builder.

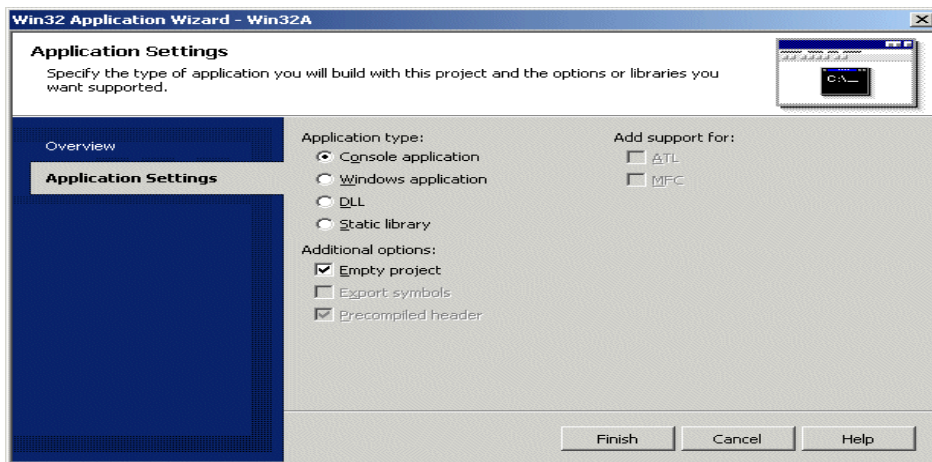
1. Start the Microsoft Development Environment.
2. On the main menu, click File -> New... or File -> New -> Project...
3. In the New or New Project dialog box, click either Win32 Application or click Visual C++ Projects and Win32 Project:



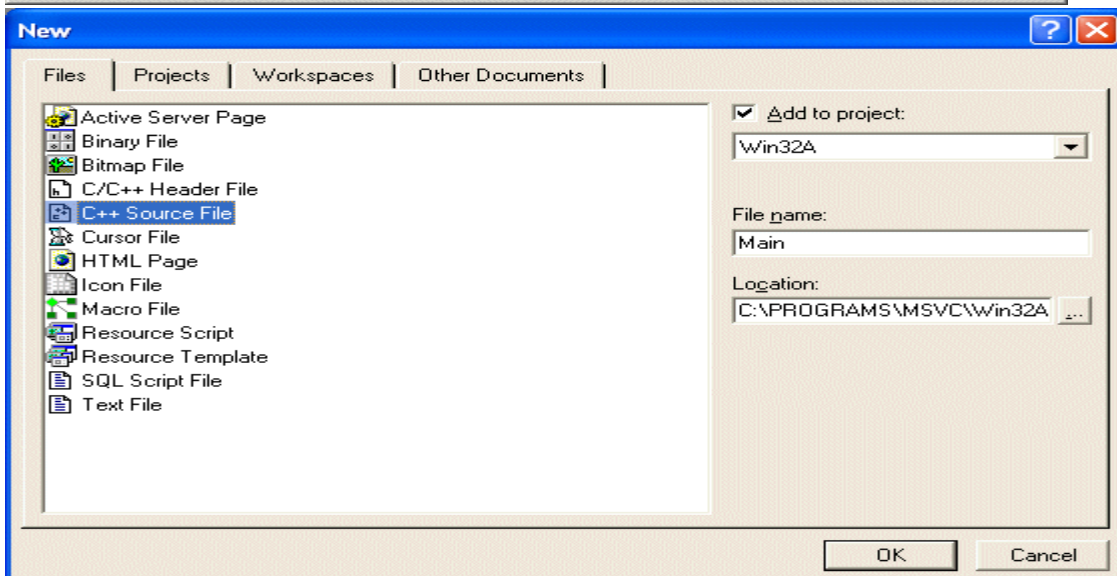
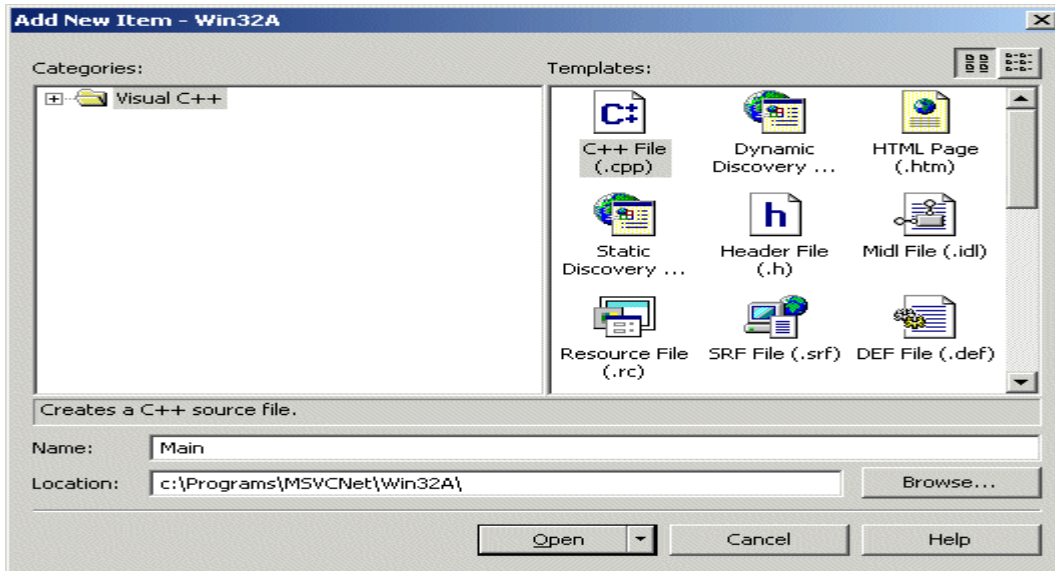
4. In the location, type the path where the application should be stored, such as C:\Programs\MSVC
5. In the Name edit box, type the name of the application as **Win32A** and click OK
6. In the next dialog box of the wizard, if you are using MSVC 5 or 6, click the An Empty Project radio button:



If you are using MSVC .Net, click Application Settings, then click the Console Application radio button, then click the Empty Project check box:



7. Click Finish. If you are using MSVC 6, you will be presented with another dialog box; in this case click OK
8. To create the first needed file of the program, if you are using MSVC 5 or 6, on the main menu, click File -> New. If you are using MSVC .Net, on the main menu, click Project -> Add New Item...
9. If you are using MSVC .Net, make sure that Visual C++ is selected in the Categories tree view. In both cases click either C++ Source File or C++ File (.cpp)



10. In the Name edit box, replace the contents with a name for a file. In the case, replace it with **Main** and press Enter

### About Microsoft Windows Messages

A computer application is equipped with Windows controls that allow the user to interact with the computer. Each control creates messages and sends them to the operating system. To manage these messages, they are handled by a function pointer called a Windows procedure. This function can appear as follows:

```
LRESULT CALLBACK MessageProcedure(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

This function uses a **switch** control to list all necessary messages and process each one in turn. This processes only the messages that you ask it to. If you have left-over messages, and you will always have un-processed messages, you can call the DefWindowProc() function at the end to take over.

The most basic message you can process is to make sure a user can close a window after using it. This can be done with a function called **PostQuitMessage()**. Its syntax is:

```
VOID PostQuitMessage(int nExitCode)
```

This function takes one argument which is the value of the **LPARAM** argument. To close a window, you can pass the argument as **WM\_QUIT**.

Based on this, a simple Windows procedure can be defined as follows:

```
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
        default:
            return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```

## Introduction to Resources

### Introduction

---

A resource is an object that cannot be defined in C++ terms but that is needed to complete a program. In the strict sense, it is text that contains a series of terms or words that the program can interpret through code. Examples of resources are menus, icons, cursors, dialog boxes, sounds, etc.

There are various means of creating a resource and the approach you use depends on the resource. For example, some resources are completely text-based, such is the case for the String Table or the Accelerator Table. Some other resources must be designed, such is the case for icons and cursors. Some other resources can be imported from another, more elaborate application, such is the case for high graphic pictures. Yet some resources can be a combination of different resources.

### Resource Creation

---

As mentioned already, resources are not a C++ concept but a **Microsoft Windows** theory of completing an application. Therefore, the programming environment you use may or may not provide you with the means of creating certain resources. Some environments like Borland C++ Builder or Visual C++ (6 and .NET) are complete with (almost) anything you need to create (almost) any type of resources. Some other environments may appear incomplete, allowing you to create only some resources, the other resources must be created using an external application not provided; such is the case for C++BuilderX.

Upon creating a resource, you must save it. Some resources are created as their own file, such is the case for pictures, icons, cursors, sound, etc. Each of these resources has a particular extension depending on the resource. After creating the resources, you must add them to a file that has the extension .rc. Some resources are listed in this file using a certain syntax. That's the case for icons, cursors, pictures, sounds, etc. Some other resources must be created directly in this file because these resources are text-based; that's the case for menus, strings, accelerators,

version numbers, etc.

After creating the resource file, you must compile it. Again, some environments, such as **Microsoft Visual C++**, do this automatically when you execute the application. Some other environments may require you to explicitly compile the resource. That's the case for Borland C++ Builder and C++BuilderX. (The fact that these environments require that you compile the resource is not an anomaly. For example, if you create a Windows application that is form-based in C++ Builder 6 or Delphi, you can easily add the resources and they are automatically compiled and added to the application. If you decide to create a Win32 application, C++ Builder believes that you want to completely control your application; so, it lets you decide when and how to compile a resource. This means that it simply gives you more control).

1.

### Practical Learning: Introducing Windows Resources

If you are using Borland C++ Builder, create a new Win32 application using Console Wizard as we did in Lesson 1

a. Save the project as **Resources1** in a new folder called **Resources1**

b. Save the unit as **Exercise.cpp**

2. If you are using Microsoft Visual C++,

a. Create a new Win32 Application as done the previous time. Save the project as **Resources1** and

b. Create a new C++ Source file named **Exercise.cpp**

3. Implement the source file as follows(for Borland C++ Builder, only add the parts that are not in the code):

```
//-----  
#include <windows.h>  
#pragma hdrstop  
  
//-----  
  
#pragma argsused  
//-----  
const char *ClsName = "FundApp";  
const char *WndName = "Resources Fundamentals";  
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,  
                               WPARAM wParam, LPARAM lParam);  
//-----  
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                  LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG        Msg;  
    HWND       hWnd;  
    WNDCLASSEX WndClsEx;  
  
    // Create the application window  
    WndClsEx.cbSize      = sizeof(WNDCLASSEX);  
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;  
    WndClsEx.lpfnWndProc = WndProcedure;  
    WndClsEx.cbClsExtra  = 0;  
    WndClsEx.cbWndExtra  = 0;
```

```

WndClsEx.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndClsEx.lpszMenuName  = NULL;
WndClsEx.lpszClassName = ClsName;
WndClsEx.hInstance     = hInstance;
WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

// Register the application
RegisterClassEx(&WndClsEx);

// Create the window object
hWnd = CreateWindowEx(0,
                    ClsName,
                    WndName,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);

// Find out if the window was created
if( !hWnd ) // If the window was not created,
    return FALSE; // stop the application

// Display the window to the user
ShowWindow(hWnd, nCmdShow); // SW_SHOWNORMAL);
UpdateWindow(hWnd);

// Decode and treat the messages
// as long as the application is running
while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}

return Msg.wParam;
}
//-----
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        // Process the left-over messages
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
}
// If something was not done, let it go

```



```

return 0;
}
4. //-----
Execute the application to test it

```

# List-Based Resources

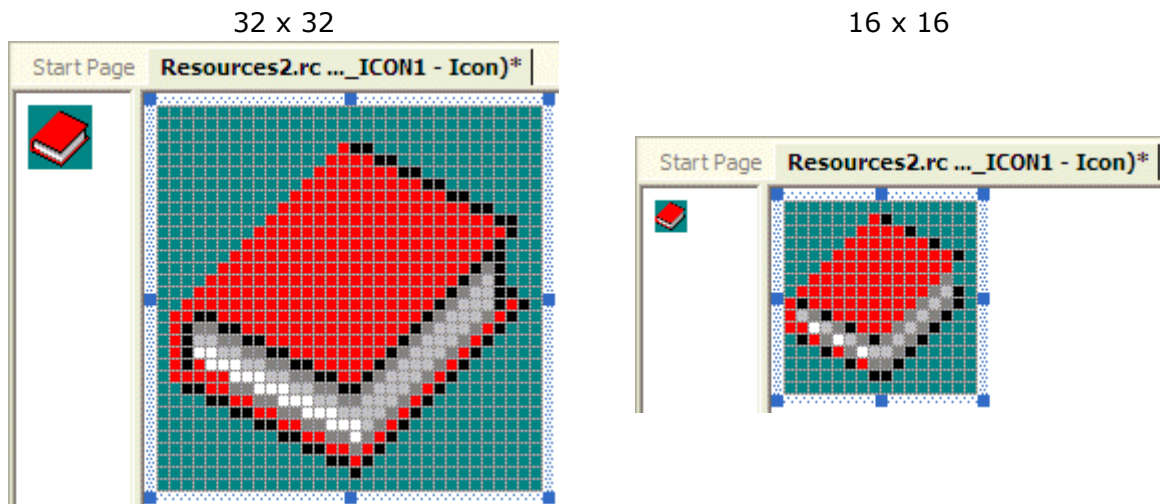
## Menus

### Introduction

A menu is a list of commands that allow the user to interact with an application. To use one of the commands, the user accesses the list and clicks the desired item. There are two main types of menus. On most applications, a menu is represented in the top section with a series of words such as File, Edit, Help. Each of these words represents a category of items. To use this type of menu, the user can display one of the categories (using the mouse or the keyboard). A list would display and the user can select one of the items. The second type of menu is called context sensitive. To use this type of menu, the user typically right-clicks a certain area of the application, a list comes up and the user can select one of the items from the list.

1. **Practical Learning: Introducing List-Based Resources**

1. Create a new Win32 Project named **Resources2** and create it as an empty project
2. On the main menu, click Project -> Add Resource...
3. Double-click Icon and design it as follows (make sure you add the 16x16 version)
- 4.



Change the ID of the icon to **IDI\_RESFUND2** and its File Name to **resfund2.ico**

5. Create a source file and name it **Exercise**
6. From what we have learned so far, type the following code in the file:
- 7.

```

//-----
-----
#include <windows.h>
#include "resource.h"

//-----
-----
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);

//-----
-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;
    const char *ClsName = "ResFund";
    const char *WndName = "Resources Fundamentals";

    // Create the application window
    WndClsEx.cbSize      = sizeof(WNDCLASSEX);
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc = WndProcedure;
    WndClsEx.cbClsExtra  = 0;
    WndClsEx.cbWndExtra  = 0;
    WndClsEx.hIcon       = LoadIcon(hInstance,
                                     MAKEINTRESOURCE(IDI_RESFUND2));
    WndClsEx.hCursor     = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName = NULL;
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance    = hInstance;
    WndClsEx.hIconSm      = LoadIcon(hInstance,
                                     MAKEINTRESOURCE(IDI_RESFUND2));

    // Register the application
    RegisterClassEx(&WndClsEx);

    // Create the window object
    hWnd = CreateWindowEx(0,
                          ClsName,
                          WndName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);

    // Find out if the window was created
    if( !hWnd ) // If the window was not created,
        return FALSE; // stop the application
}

```

## Menu Creation

---

A menu is one of the text-based resources. It is created directly in the rc file. As with other resources, the process of creating a menu depends on the environment you are using. If you are using Borland C++ Builder, you can open your rc file and manually create your menu.

If you are using Microsoft Visual C++, you can use the built-in menu editor. In this case, the actual text that defines and describes the menu would be automatically added to the rc file.

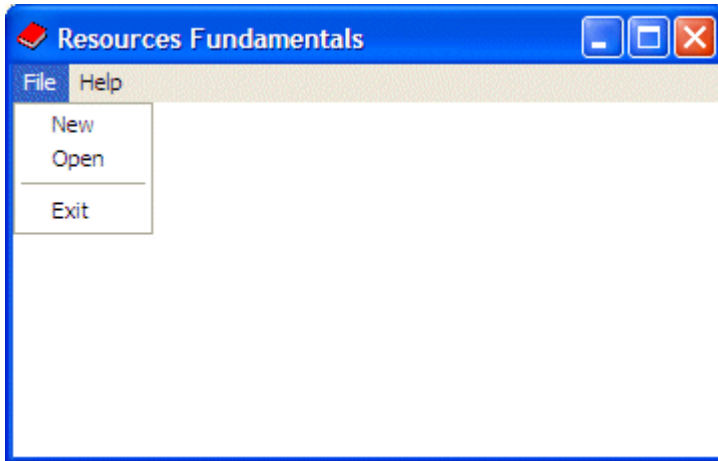
### ▼ Practical Learning: Creating a Menu

---

1. On the main menu, click Project -> Add Resource...
2. In the Insert Resource dialog box, click Menu and click New
3. While the first menu item is selected, type **&File**
4. Click the next empty item under File. Type **&New**
5. In the Properties window, click the ID edit box, type **IDM\_FILE\_NEW** and press Enter
6. Click the next empty item under New and type **&Open**
7. In the Properties window, click the ID edit box, type **IDM\_FILE\_OPEN** and press Tab
8. Click the next empty item under Open and type -
9. Click the next empty item under the new separator and type **E&xit**
10. In the Properties window, click the ID edit box, type **IDM\_FILE\_EXIT** and press Tab. In the Caption edit box, press Enter
11. Click the next empty item on the right side of File. Type **&Help**
12. Click the next empty item under Help and type **&About**
13. In the Properties window, click the ID edit box, type **IDM\_HELP\_ABOUT** and press Tab. In the Caption edit box, and press Enter
14. In the ResourceView tab of the Workspace, under the Menu node, click IDR\_MENU1. In the Menu Properties window, change the ID to **IDR\_MAINFRAME**
15. Open the Exercise.cpp source file and change the lpszMenuName member of the WndClsEx variable as follows:
16. 

```
WndClsEx.lpszMenuName = MAKEINTRESOURCE (IDR_MAINFRAME);
```

  
To test the application, press Ctrl + F5 and press Enter



17. Return to your programming environment

## String Tables

---

### Introduction

---

As its name indicates, a string table is a list of strings created in the resource file. Like the menu resource, the string table is created directly in the resource file. The advantage of using a string table is that the string defined in it are declared globally, meaning that they can be accessed by any object of the application without being declared

### String Table Creation

---

As mentioned already, a string table is created in the rc file. Therefore, in Borland C++ Builder and other environments, the list can be manually entered in the resource file. In Visual C++, to create the list, you can add the String Table from the Add Resource dialog box. This opens a special window in which you can edit each string.

Besides new strings you create in the list with their own new identifiers, you can also use the string table to assign strings to already created identifiers. For example, to can assign a string to some or all of the menu items we created earlier.

After creating a string table, you can access any of its strings from anywhere in the application. To access a string, call the **LoadString()** function.

### ▼ Practical Learning: Creating and Using a String Table

---

1. To create a string table, on the main menu, click Project -> Add Resource...
2. In the Add Resource dialog box, double-click String Table
3. Replace the first IDS\_ identifier with **IDS\_APP\_NAME** and press Tab twice
4. Type the string as **Fundamentals of Windows Resources** and press Enter
5. In the same way, create the following strings and their identifiers (let the string table add the Values):

- |                    | Value |                                       |
|--------------------|-------|---------------------------------------|
| IDM_ARROW          | 105   | No tool selected                      |
| IDM_DRAW_LINE      | 106   | Selects the Line tool\nLine           |
| IDM_DRAW_RECTANGLE | 107   | Selects the Rectangle tool\nRectangle |
| IDM_DRAW_ELLIPSE   | 108   | Selects the Ellipse tool\nEllipse     |
6. To assign to an existing identifier, click the arrow of the second identifier and select IDM\_FILE\_NEW
  7. Click its corresponding Caption section and type **Creates a new file\nNew**
  8. To use a string from the string table, change the source file as follows:

```

//-----
-----
#include <windows.h>
#include "resource.h"

//-----
-----
char AppCaption[40];
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);

//-----
-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;
    const char *ClsName = "ResFund";

    LoadString(hInstance, IDS_APP_NAME, AppCaption, 40);

    // Create the application window
    WndClsEx.cbSize      = sizeof(WNDCLASSEX);
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc = WndProcedure;
    WndClsEx.cbClsExtra  = 0;
    WndClsEx.cbWndExtra  = 0;
    WndClsEx.hIcon       = LoadIcon(hInstance,
                                     MAKEINTRESOURCE(IDI_RESFUND2));
    WndClsEx.hCursor     = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName = MAKEINTRESOURCE(IDR_MAINFRAME);
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance    = hInstance;
    WndClsEx.hIconSm      = LoadIcon(hInstance,

MAKEINTRESOURCE(IDI_RESFUND2));

    // Register the application
    RegisterClassEx(&WndClsEx);

    // Create the window object
    hWnd = CreateWindowEx(0,
                          ClsName,
                          AppCaption,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);

    // Find out if the window was created

```

# Toolbars

---

## Introduction

---

A toolbar is an object used to let the user perform actions on an application. Like the menu, the toolbar provides a list of commands. While a menu may require various actions to get to the desired command, a toolbar presents buttons that would lead to the same actions as the menu, only faster. As such, a toolbar is mainly made of buttons placed on it, but a toolbar can also contain many other types of controls.

## Toolbar Creation

---

A toolbar is primarily made of buttons that each displays a small icon, typically 16x16. If you intend to create this type of toolbar, you can start by creating one or more pictures. There are two main types of pictures you would use. The first type uses a kind of picture called a bitmap (we will learn about bitmaps in future lessons). You can create a long picture resource that has a height of 16 pixels (a toolbar can also be taller than that but this height is the most common). Then, add a factor of 16 pixels width for each desired button. This means that, if the toolbar will have one button, you can create a bitmap of 16x16 pixels. If the toolbar will have 4 buttons, you can create a bitmap of height = 16pixels and width = 16 \* 4 = 64pixels. After creating this type of bitmap, save it and give it an identifier. The other type of toolbar can use icons that each is created on its own.

Like most other objects you will use in your applications, a toolbar should have an identifier. This would help you and Windows identifier the toolbar.

Since a toolbar is made of small buttons, each button is an object of type **TBBUTTON**. The **TBBUTTON** structure is defined as follows:

```
typedef struct _TBBUTTON {
    int         iBitmap;
    int         idCommand;
    BYTE        fsState;
    BYTE        fsStyle;
#ifdef _WIN64
    BYTE        bReserved[6]    // padding for alignment
#elif defined(_WIN32)
    BYTE        bReserved[2]    // padding for alignment
#endif
    DWORD_PTR   dwData;
    INT_PTR     iString;
} TBBUTTON, NEAR *PTBBUTTON *LPTBBUTTON;
```

The buttons are stored in an array of **TBBUTTON** values.

Unlike the other resources we have used so far, to add a toolbar to your application, you must programmatically create it. To do this, call either the **CreateWindowEx** or the **CreateToolBarEx()** function. The syntax of the **CreateToolBarEx()** function is:

```
HWND CreateToolBarEx(HWND hwnd,
                    DWORD ws,
                    UINT wID,
                    int nBitmaps,
                    HINSTANCE hBMInst,
                    UINT_PTR wBMID,
```



```

        LPCTBBUTTON lpButtons,
        int iNumButtons,
        int dxButton,
        int dyButton,
        int dxBitmap,
        int dyBitmap,
        UINT uStructSize
);

```

The first argument is the window that serves as the parent to the toolbar. This is usually the first window you would have created.

The second argument specifies the style used to display the toolbar. This parameter is a combination of windows styles and toolbar styles.

The third argument is the identifier of the toolbar.

The *nBitmaps* is used to specify the number of pictures included in the bitmap you will use

The *hBMInst* is the application that will manage the toolbar

The *hBMID* is an identifier of the bitmap

The *lpButtons* argument is a pointer to the array of **TBBUTTON** values you have defined already.

The *iNumButtons* specifies the number of buttons that will be created on the toolbar

The *dxButton* and *dyButton* parameters represent the dimension of each button

The *dxBitmap* and *dyBitmap* parameters represent the dimension of the bitmap that will be used on the buttons

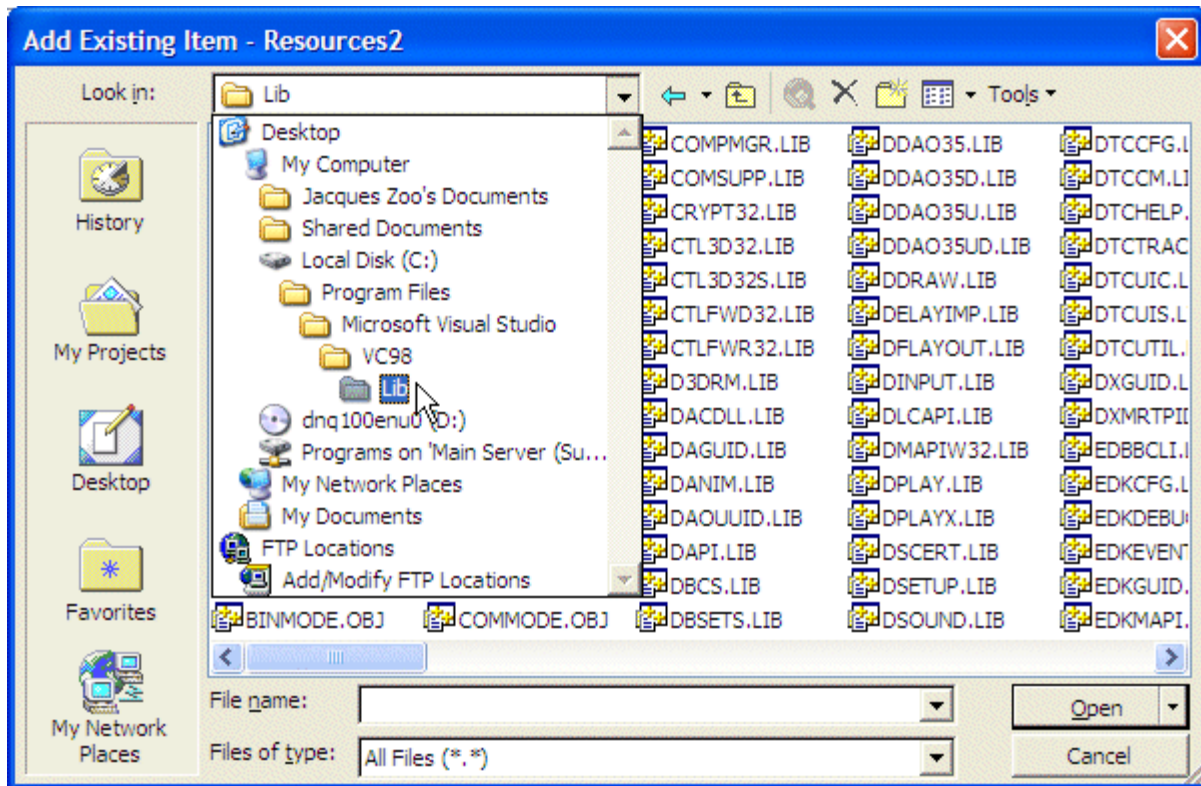
The last argument is the size of the **TBBUTTON** structure.

The functions and classes (actually, structures) used to create and manage a toolbar are defined in the `comctl.h` header file. The `comctl.h` header is part of the `comctl32.lib` library. This library is not carried by the `windows.h` header or its associated libraries. Therefore, since the toolbar belongs to the family of Common Controls, you must explicitly add the `comctl32.lib` library whenever you want to use one of them.

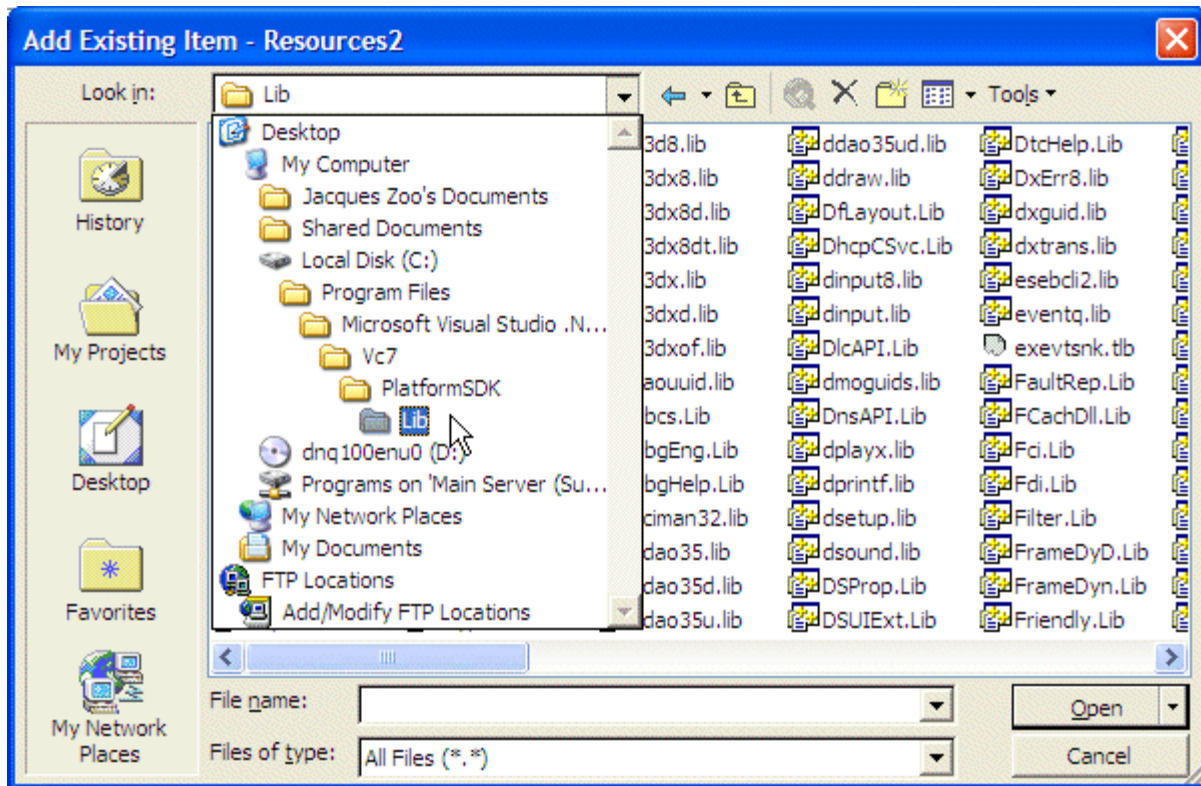
## ▼ Practical Learning: Creating a Toolbar

---

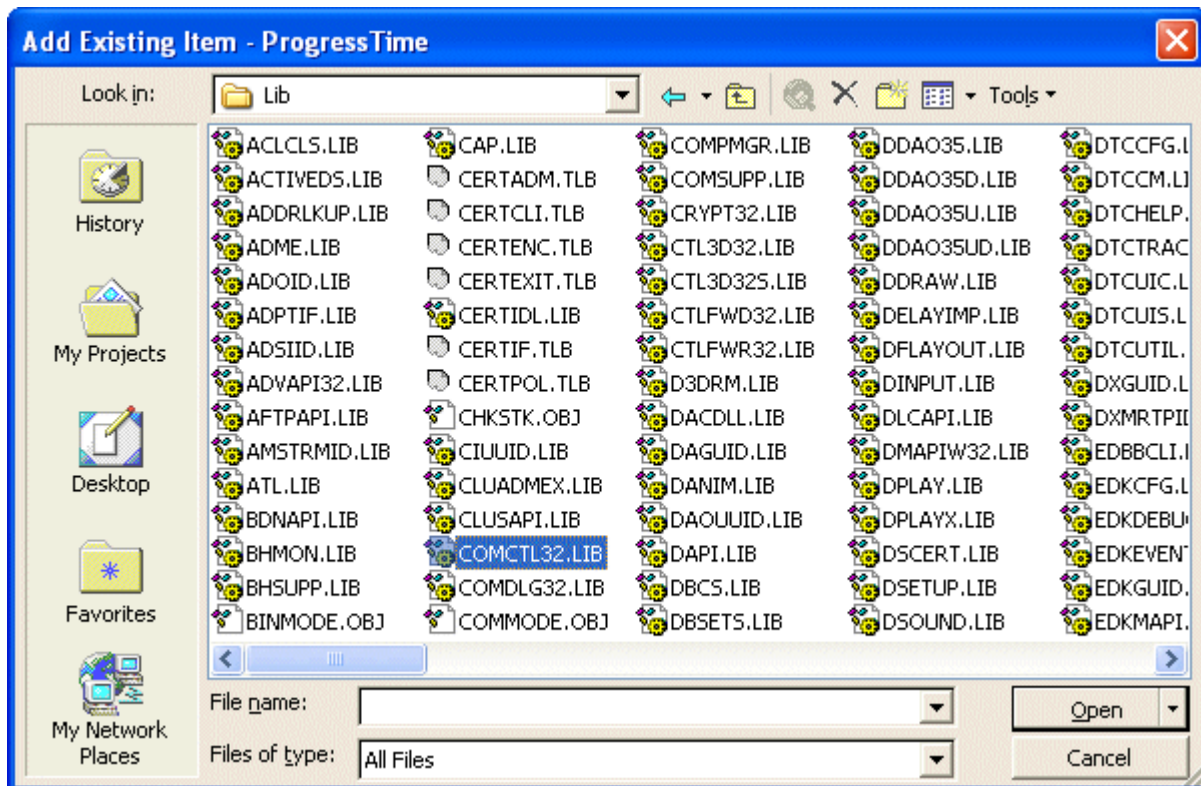
1. To include the `comctl32.lib` library in your application, on the main menu, click Project -> Add Existing Item...
2. Locate the folder that contains your libraries and display it in the Look In combo box. For Visual C++ 6.0, this would be in `Drive:\Program Files\Microsoft Visual Studio\VC98\Lib`



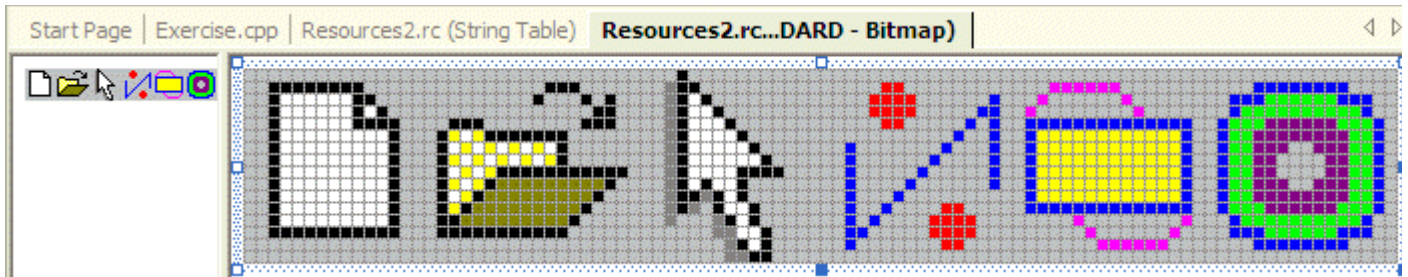
For Visual C++ .NET, this would be *Drive:\Program Files\Microsoft Visual Studio .NET 2003\VC7\PlatformSDK\Lib*



3. Click ComCtl32.Lib



4. Click Open
5. To start a toolbar, on the main menu, click Project -> Add Resources and double-click Bitmap
6. Design the bitmap as 96 x 16 (width x height) as follows:



7. Change its ID to **IDB\_STANDARD** and its file name to standard.bmp
8. To add the toolbar to the application, change the source file as follows:

```

//-----
#include <windows.h>
#include <commctrl.h>
#include "resource.h"

//-----
char AppCaption[40];
HINSTANCE hInst;
const int NUMBUTTONS = 7;
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);

//-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;
    const char *ClsName = "ResFund";

    LoadString(hInstance, IDS_APP_NAME, AppCaption, 40);

    hInst = hInstance;

    // Create the application window
    WndClsEx.cbSize        = sizeof(WNDCLASSEX);
    WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc   = WndProcedure;
    WndClsEx.cbClsExtra    = 0;
    WndClsEx.cbWndExtra    = 0;
    WndClsEx.hIcon         = LoadIcon(hInstance,

MAKEINTRESOURCE(IDI_RESFUND2));
    WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName  = MAKEINTRESOURCE(IDR_MAINFRAME);
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance     = hInst;
    WndClsEx.hIconSm       = LoadIcon(hInstance,

MAKEINTRESOURCE(IDI_RESFUND2));

    // Register the application
    RegisterClassEx(&WndClsEx);

    // Create the window object
    hWnd = CreateWindowEx(0,
                          ClsName,
                          AppCaption,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,

```

# Dialog Boxes

---

## Message Boxes

---

### Introduction

---

A message box is a rectangle object that displays short message to the user. The message can be made of one sentence, one paragraph, or a few paragraphs. To make the creation of a message box easy, the Win32 library provides a specific function that can be used to for this purpose.

### Message Box Creation

---

To create a message box, use the **MessageBox()** function. Its syntax is:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

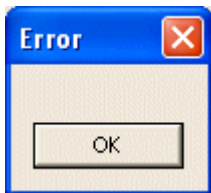
The first argument, hWnd, can be a handle to the window from where the message box will be called. Otherwise, it can NULL.

The second argument, lpText, is a null-terminated string, such as an array of characters. This is the actual message that will be presented to the user. As stated already, it can be one word, a whole sentence, a paragraph, even a hew paragraphs.

The third argument, lpCaption, is the title that will display on the title bar. It also can be a null-terminated string, if you know what title you would like to display. Otherwise, it can be NULL, in which case the title bar would display Error.

The simplest way you can create a message is by calling the MessageBox() function with all arguments set to NULL, in which case the message box would not make any sense:

```
MessageBox(NULL, NULL, NULL, NULL);
```



As stated already, the first argument is either a handle of the window that is calling it, or NULL.

The simplest way to specify the second argument is by including a word or a sentence in double-quotes. Here is an example:

```
MessageBox(NULL, "I am just trying my wedding dress", NULL, NULL);
```



If you want to display the message on various lines, you can separate sections with the new line character '\n'. Here is an example:

```
MessageBox(NULL, "It happened earlier\nDidn't it?", NULL, NULL);
```







You can also use string editing techniques to create a more elaborate message. This means that you can use functions of the C string library to create your message.

The caption of the message can be any word or sentence but convention wisdom would like this sentence to be in tune with the actual message. After all, unless the message is about bad news, Error as a title is not particularly cute.

The fourth argument actually does three things. First it displays one or a few buttons. The buttons depend on the value specified for the argument. If this argument is NULL, the message box displays (only) OK. The values and their buttons can be as follows:

Constant Integer	Buttons
MB_OK	OK
MB_OKCANCEL	OK Cancel
MB_ABORTRETRYIGNORE	Abort Retry Ignore
MB_YESNOCANCEL	Yes No Cancel
MB_YESNO	Yes No
MB_RETRYCANCEL	Retry Cancel
MB_CANCELTRYCONTINUE	Cancel Try Again Continue
MB_HELP	Help

Besides the buttons, the message box can also display a friendly icon that accompanies the message. Each icon is displayed by specifying a constant integer. The values and their buttons are as follows:

Value	Icon	Suited when
MB_ICONEXCLAMATION MB_ICONWARNING		Warning the user of an action performed on the application
MB_ICONINFORMATION MB_ICONASTERISK		Informing the user of a non-critical situation
MB_ICONQUESTION		Asking a question that expects a Yes or No, or a Yes, No, or Cancel answer
MB_ICONSTOP MB_ICONERROR MB_ICONHAND		A critical situation or error has occurred. This icon is appropriate when informing the user of a termination or deniability of an action

The icons are used in conjunction with the buttons constant. To combine these two flags, use the bitwise OR operator "|".

The second thing this fourth argument does is to let the user close the message box after selecting one of the buttons. Once the user clicks one of the buttons, the message box is closed.

The third role of this fourth argument is to control the result derived from the user dismissing the message box. For example, clicking OK usually means that the user acknowledges what the message. Clicking Cancel usually means the user is changing his or her mind about the action performed previously. Clicking Yes instead of No usually indicates that the user agrees to perform an action.

In reality, the message box only displays a message and one or a few buttons. It is your responsibility as the programmer to decide what to do when what button is clicked.

When a message box is configured to display more than one button, the operating system is set to decide which button is the default. The default button has a thick border that sets it apart from the other button(s). If the user presses Enter, the message box would behave as if the user had clicked the default button. Fortunately, if the message box has more than one button, you can decide what button would be the default. To specify the default button, use one of the following constants:

Value	If the message box has more than one button, the default button would be
MB_DEFBUTTON1	The first button
MB_DEFBUTTON2	The second button
MB_DEFBUTTON3	The third button
MB_DEFBUTTON4	The fourth button

To specify the default button, use the bitwise OR operator to combine the constant integer of the desired default button with the button's constant and the icon.

## ▼ Practical Learning: Introducing Additional Resources

1. Create a new Win32 application
2. If you are using Borland C++ Builder, save the application in a new folder called **Win32C** and save the project as **MessageBox**



If you are using Microsoft Visual C++, set the location to a folder called **Win32C**

3. If you are using Borland C++ Builder, save the unit as **Main.cpp**  
If you are using Microsoft Visual C++, create a C++ source file and save it as **Main.cpp**

4.

```
//-----  
#include <windows.h>  
  
//-----  
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                  LPSTR lpCmdLine, int nCmdShow)  
{  
    MessageBox(NULL, "Welcome to Win32 Application Development\n", NULL, NULL);  
  
    return 0;  
}  
//-----
```

Replace the file's content with the following:

5. Test the program and return to your programming environment
6. To create a more elaborate message, change the file as follows:

```
//-----  
#include <windows.h>  
  
//-----  
  
const char Caption[] = "Application Programming Interface";  
  
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                  LPSTR lpCmdLine, int nCmdShow)  
{  
    MessageBox( NULL,  
              "Welcome to Win32 Application Development\n"  
              "You will learn about functions, classes, "  
              "communication, and other cool stuff\n"  
              "Are you ready to rumble!!!!!!!!!!!!!!!!!!!!",  
              Caption,  
              MB_YESNOCANCEL | MB_ICONQUESTION);  
  
    return 0;  
}  
//-----
```

7. the application

Test

8.



Return to your programming environment

## Dialog Boxes

---

### Introduction

---

A dialog box is a rectangular object that displays a message, a control, or a few controls, allowing the user to interact with an application.

A dialog box is created from a resources file, which is a file with the rc extension. The resource file contains all pertinent information about the dialog box. This means that, in this file, you must specify the size and location of the dialog box, its caption, whether it contains buttons.

After creating the resource file, make sure you add it to your project so you can compile it.

There are two main reasons you would create or use a dialog box. You can create a dialog-based application, that is, a dialog that uses only one or a few dialog box. You can also use a dialog box as an addition to your application.

### Using Borland C++ Builder

---

1. Create a new Win32 application using the Console Wizard
2. Save the application in a new folder called **Win32D** and save the project as **DialogBox**
3. Save the unit as **Main.cpp**
4. To create the resource header file, on the main menu, click File -> New... or File -> New -> Other...
5. In the New Items dialog box, click Header File and click OK
6. Save the new file as **Resource.h** (make sure you specify the .h extension when saving the file).
7. In the empty header file, type:
8. 

```
#define IDD_DLGFIRST 101
```

To create the rc resource file, on the main menu of C++ Builder, click File -> New -> Other...
9. In the New Items dialog box, click the **Text** icon and click OK
10. To save the resource file, on the Standard toolbar, click the Save All button
11. Type the file name as "**Win32D.rc**" (the double-quotes allow you to make sure the file is saved as rc and not as txt)

12. In the empty file, type the following (the referenced header file will be created next):

```
#include "Resource.h"

IDD_DLGFIRST DIALOG 260, 200, 188, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Win32 Programming"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON    "OK", IDOK, 130, 10, 50, 14
END
```

13.

In the Code Editor, click the Win32D.rc tab to select it. To compile the resource, on the main menu of Bcb, click Project -> Compile Unit

14. After the unit has been compiled, click OK

15. To add the rc file to the project, on the main menu, click Project -> Add To Project...

16. Select Win32D.rc and click Open

## Using Microsoft Visual C++

---

1. Create a new Win32 Application. Specify the desired path in the Location edition box. In the Project Name, type Win32D
2. Make sure you create it as an Empty Project.
3. To create a dialog box along with the resource files, on the main menu, click Insert -> Resource...
4. In the Insert Resource dialog box, click Dialog and click New
5. Right-click in the main body of the new dialog box and click Properties. In the Dialog Properties window, change the ID to IDD\_DLGFIRST and press Tab.
6. In the Caption box, type Win32 Programming
7. In the X Pos box, type 260
8. In the Y Pos box, type 200
9. To close and save the resource file, click the system Close button of the Script1 - IDD\_DLGFIRST window (or the lower X in the upper-right section)
10. When asked whether you want to save the script, click Yes.
11. Locate the folder in which the application is being created (in this case Win32D) and display it in the Save In combo box.
12. In the File Name box, replace the Script1 name with Win32D.rc and click Save
13. To add the rc file to your project, on the main menu, click Project -> Add To Project -> Files...
14. Select Win32D.rc and click OK.
15. To create the main source file of the project, on the main menu, click File -> New...
16. In the Files property page of the New dialog box, click C++ Source File. In the File Name edit box, type Main and press Enter.

## Programmatically Creating Dialog Boxes

---

A dialog box is created using the DialogBox function. Its syntax is:

```
INT_PTR DialogBox(HINSTANCE hInstance, LPCTSTR lpTemplate, HWND hWndParent,  
DLGPROC lpDialogFunc);
```

The first argument of this function is a handle to the application that is using the dialog box.

The lpTemplate specifies the dialog box template.

The hWndParent is a handle to the parent window that owns this dialog box.

The lpDialogFunc must be a procedure that is in charge of creating this dialog box.

Therefore, you must define a CALLBACK procedure that whose syntax is:

```
INT_PTR CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam, LPARAM  
lParam);
```

1. In the Main.cpp file, create the program as follows:

```

#include <windows.h>
#include "Resource.h"

//-----
//-----
HWND hWnd;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
//-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    DialogBox(hInstance, MAKEINTRESOURCE(IDD_DLGFIRST),
hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return FALSE;
}
//-----
//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,
LPARAM lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDOK:
            EndDialog(hWndDlg, 0);
            return TRUE;

        }
        break;

    }

    return FALSE;
}
//-----
//-----

```

2. Test the program

## Windows Messages

---

### Introduction to Messages

---

#### Microsoft Window as a Messaging Center

---

The computer is a machine that only follows instructions. It almost doesn't know anything. Because of this, the computer cannot predict what a user wants to do with the computer. In fact, a great deal of the responsibility is left to the programmer who must decide what can and cannot or should not be done on an application. To help the users with computer interaction, the operating system provides a series of objects called Windows controls. The programmer decides what objects are necessary for a given application.

Each computer application is equipped with Windows controls that allow the user to interact with the computer. Because the computer cannot and would not predict what the user wants to do when using the computer, the operating system lets each object tell it when it needs something from Windows. To do this, a control sends a *message* to the operating system every time something is new. Because there can be so many messages a control can send and because many controls can send various messages, there is a formula each message or almost every one of them must follow, just like there are rules the post office wants you to follow in order to send a letter.

A message to Windows must provide four pieces of information:

- WHO sent the message? Every object you will need in your program, just like everything in the computer, must have a name. The operating system needs this name to identify every object, for any reason. An object in Microsoft Windows is identified as a Handle. For Windows controls, the handle is called **HWND**
- WHAT message? The object that sends a message must let the operating system know what message it is sending. As we will learn, there are various types of messages for different circumstances. Nevertheless, to make matters a little easier, each message is a constant positive natural number (unsigned int) identified with a particular name. Therefore, the message identifier is passed as **UINT**
- Accompanying items: Because there are so many types of messages, you must provide two additional pieces of information to help process the message. These two items depend on the type of message and could be anything. The first accompanying item is a 32-bit type (unsigned int) called **WPARAM** (stands for **WORD** Parameter; in other words, it is a **WORD** (unsigned int) argument). The second accompanying item is a 32-bit type of value (long) called **LPARAM** (stands for LONG Parameter; in other words, it is a LONG (long in C/C++) argument). Remember that these two can be different things for different messages.

To manage the messages sent to Windows, they are communicated through a function pointer called a Windows procedure. The name of the function is not important but it must return a 32-bit integer, in fact a C/C++ long or Win32 LONG. Therefore, it is declared as **LRESULT** (LONG Result). Because this is a function pointer, it must be declared and defined as **CALLBACK**. The messages can be carried in a function defined as follows:

```
LRESULT CALLBACK MessageProcedure(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

To process the messages, and because there can be so many of them, this function typically uses a **switch** control to list all necessary messages and process each one in turn. After processing a message, its case must return a value indicating that the message was successfully processed or not.

No matter how many messages you processed, there will still be messages that you did not deal with. It could be because they were not sent even though they are part of the Windows controls used on an application. If you didn't process some messages, you should/must let the operating system know so it can take over. What happens is that the operating system is aware of all messages and it has a default behavior or processing for each one of them. Therefore, you should/must return a value for this to happen. The value returned can be placed in the **default**

section of the **switch** condition and must simply be a **DefWindowProc()** function. Its syntax is:

```
LRESULT DefWindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

This function is returned to Windows, saying "There are messages I couldn't process. Do what you want with them". The operating system would simply apply a default processing to them. The values returned by the **DefWindowProc()** function should be the same passed to the procedure.

The most basic message you can process is to make sure a user can close a window after using it. This can be done with a function called **PostQuitMessage()**. Its syntax is:

```
VOID PostQuitMessage(int nExitCode)
```

This function takes one argument which is the value of the **LPARAM** argument. To close a window, you can pass the argument as **WM\_QUIT**.

Based on this, a simple Windows procedure can be defined as follows:

```
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
        default:
            return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```

A basic program with one message can be written as follows:

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

//-----
---
HWND hWnd;
const char ClsName[] = "WndMsg";
const char WindowCaption[] = "Windows and Controls Messages";
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
---
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    MSG          Msg;
    WNDCLASSEX   WndClsEx;

    WndClsEx.cbSize          = sizeof(WNDCLASSEX);
    WndClsEx.style           = CS_HREDRAW | CS_VREDRAW;
```

```

WndClsEx.lpfWndProc    = WndProc;
WndClsEx.cbClsExtra   = NULL;
WndClsEx.cbWndExtra   = NULL;
WndClsEx.hInstance   = hInstance;
WndClsEx.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
WndClsEx.hCursor      = LoadCursor(NULL, IDC_ARROW);
WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndClsEx.lpszMenuName = NULL;
WndClsEx.lpszClassName = ClsName;
WndClsEx.hIconSm      = LoadIcon(NULL, IDI_APPLICATION);

RegisterClassEx(&WndClsEx);

hWnd = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                    ClsName,
                    WindowCaption,
                    WS_OVERLAPPEDWINDOW,
                    100,
                    120,
                    640,
                    480,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}

return Msg.wParam;
}
//-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----

```

## Windows Messages

---



## Window Creation

---

**WM\_CREATE:** Once you have decided to create a message, you send a message to Windows. Actually, when you are creating a window, a message called **WM\_CREATE** is sent to Windows. This is the favorite message you can use to perform any early processing that you want to make sure happens before most other things show up. For example, you can use this message to initialize anything in your application. Therefore, this message is the first sent to the operating system. Here is an example:

```
//-----  
---  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)  
{  
    switch(Msg)  
    {  
        case WM_CREATE:  
            MessageBox(NULL, "The window is being created", WindowCaption,  
MB_OK);  
            break;  
        case WM_DESTROY:  
            PostQuitMessage(WM_QUIT);  
            break;  
        default:  
            return DefWindowProc(hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}  
//-----  
---
```

## Window Display

---

**WM\_SHOWWINDOW:** After a window has been created, it needs to be displayed, that is, the window needs to be shown. Also, if the window was previously hidden, you can decide to show it. On the other hand, if a window is displaying, you may want to hide it, for any reason you judge necessary. To take any of these actions, that is, to show or hide a window, you must send the **WM\_SHOWWINDOW** message. The syntax of this message is:

```
OnCreate(HWND hWnd, WM_SHOWWINDOW, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that sends the message.

*wParam* is a Boolean value. If you want to display or show the *hWnd* window, set the *wParam* value to TRUE. If you want to hide the *hWnd* window, set this value to FALSE.

*lParam* specifies the status of the window.

```
//-----  
---  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)  
{  
    switch(Msg)  
    {  
        case WM_CREATE:  

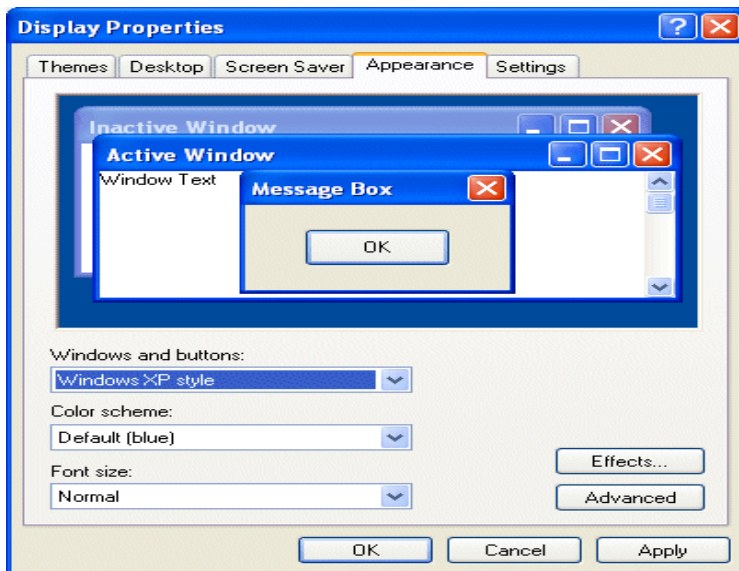
```

```

        MessageBox(NULL, "The window is being created", WindowCaption,
MB_OK);
        break;
    case WM_SHOWWINDOW:
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
---
```

## Window Activation

**WM\_ACTIVATE:** When two or more windows are running on the computer, only one can receive input from the user, that is, only one can actually be directly used at one particular time. Such a window has a title bar with the color identified in Control Panel as Active Window. The other window(s), if any, display(s) its/their title bar with a color called Inactive Window:



To manage this setting, the windows are organized in a 3-dimensional coordinate system and they are incrementally positioned on the Z coordinate, which defines the (0, 0, 0) origin on the screen (actually on the top-left corner of your monitor) with Z coordinate coming from the screen towards you.

In order to use a window other than the one that is active, you must activate it. To do this, you can send a message called **WM\_ACTIVATE**.

The syntax of this message is:

```
OnActivate(HWND hWnd, WM_ACTIVATE, WPARAM wParam, LPARAM lParam);
```

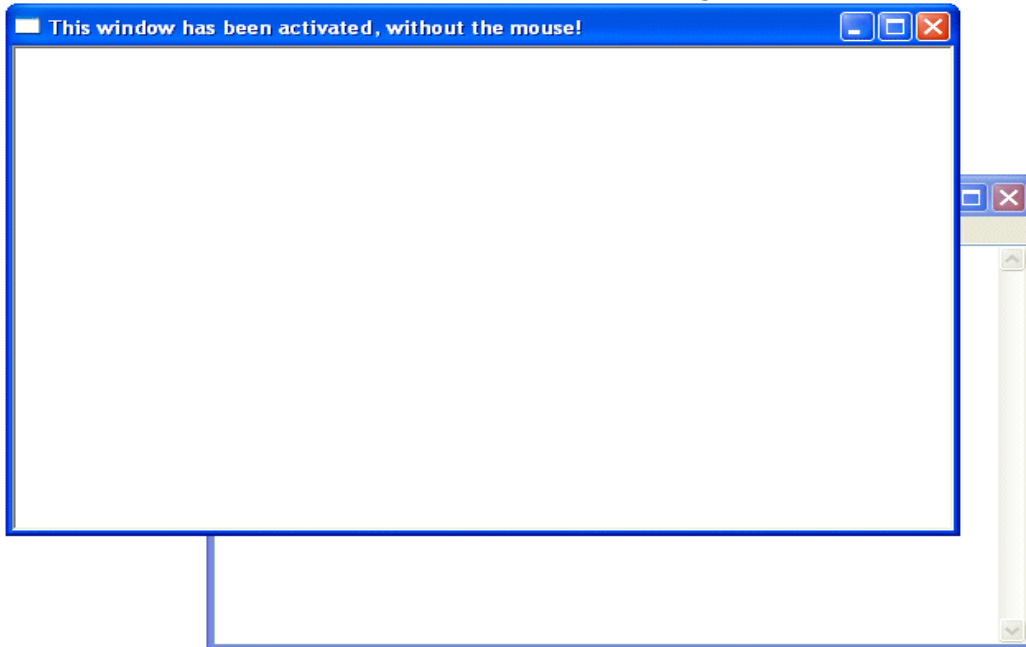
Actually, this message is sent to two objects: the window that is being activated and the one

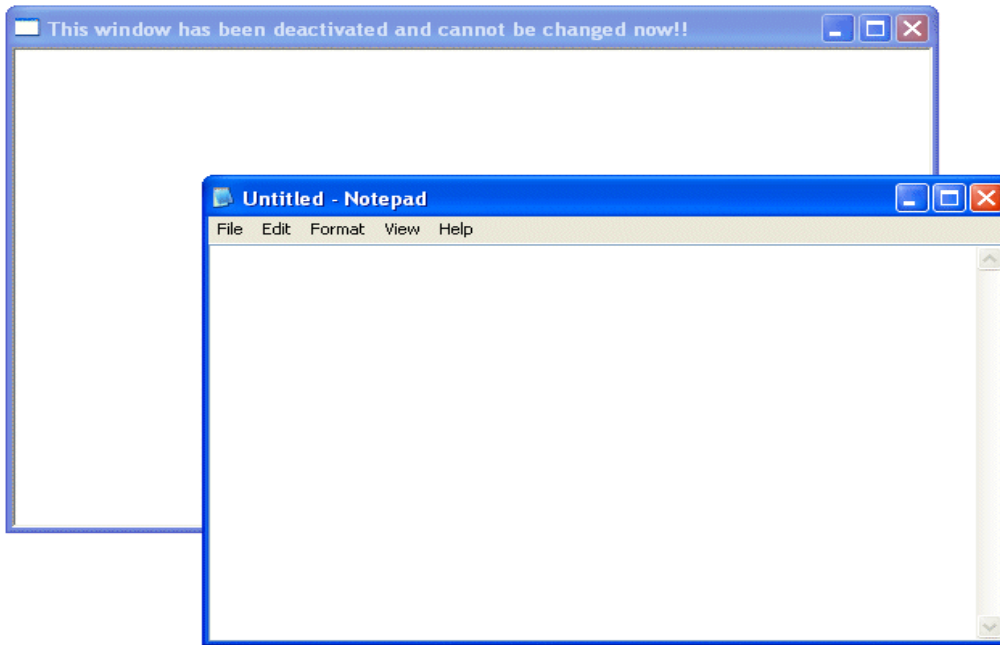
that is being deactivated.

*hWnd* identifies a window involved in this message and is related to the *wParam* parameter.

The *wParam* parameter specifies the action to take. It is a constant value that can be one of the following:

<b>Value</b>	<b>Description</b>
WA_ACTIVE	Used to activate the window
WA_INACTIVE	Used to deactivate the window without using the mouse, for example by pressing Alt+Tab
WA_CLICKACTIVE	Used to activate the window using the mouse





```

//-----
---
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_CREATE:
        MessageBox(NULL, "The window is being created", WindowCaption,
MB_OK);
        break;
    case WM_ACTIVATE:
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
---

```

## Window Painting

---

**WM\_PAINT:** Whenever Microsoft Windows is asked to display (whether it must unhide or activate) a window, it must use its location (measure from left and top) and size (width and height). It must give it the Active Window color and it must restore its other active colors. To do this, the operating system must paint the window. When doing this, a message called **WM\_PAINT** is sent. The syntax of this message is:

```
OnPaint(HWND hWnd, WM_PAINT, WPARAM wParam, LPARAM lParam);
```

The only thing Windows needs to know is the window that needs to be painted or repainted, which is specified as the *hWnd* value.

```
//-----  
---  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)  
{  
    switch(Msg)  
    {  
        case WM_CREATE:  
            MessageBox(NULL, "The window is being created", WindowCaption,  
MB_OK);  
            break;  
        case WM_SHOWWINDOW:  
            break;  
        case WM_ACTIVATE:  
            break;  
        case WM_PAINT:  
            break;  
        case WM_DESTROY:  
            PostQuitMessage(WM_QUIT);  
            break;  
        default:  
            return DefWindowProc(hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}  
//-----  
---
```

## Window Sizing

---

**WM\_SIZE:** When using an application, one of the actions a user may perform is to change its size. Whenever the size of a window has changed, the window receives the **WM\_SIZE** message. Its syntax is:

```
OnSize(HWND hWnd, WM_SIZE, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that was resized.

*wParam* determines how the sizing action should be performed. It can be one of the following values

Value	Description
SIZE_MAXHIDE	Sent to this window if it was maximized from being previously hidden
SIZE_MAXIMIZED	Sent to this window if it was maximized
SIZE_MAXSHOW	Sent if the window was restored
SIZE_MINIMIZED	Sent if the window was minimized
SIZE_RESTORED	Sent if the window was resized, but neither the SIZE_MINIMIZED nor SIZE_MAXIMIZED value applies

*lParam* specifies the dimensions of the window (the width and the height)

```

//-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_CREATE:
        break;
    case WM_SHOWWINDOW:
        break;
    case WM_ACTIVATE:
        break;
    case WM_PAINT:
        break;
    case WM_SIZE:
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----

```

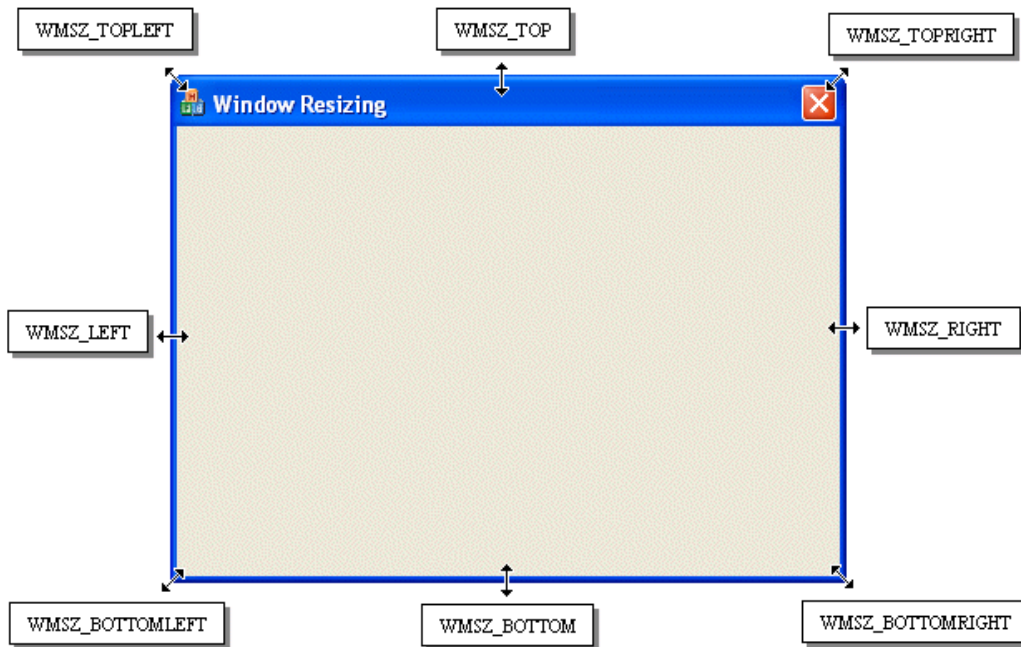
**WM\_SIZING:** This message is sent to a window that is being resized with the above **WM\_SIZE** message. Its syntax is:

```
OnSizing(HWND hWnd, WM_SIZING, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that is being resized.

To resize a window, the user usually grabs one of the four corners or one of the four borders. The *lParam* parameter specifies what border or corner the user is moving. It can be one of the following values:

<b>Value</b>	<b>Border or Corner</b>
WMSZ_BOTTOM	Bottom edge
WMSZ_BOTTOMLEFT	Bottom-left corner
WMSZ_BOTTOMRIGHT	Bottom-right corner
WMSZ_LEFT	Left edge
WMSZ_RIGHT	Right edge
WMSZ_TOP	Top edge
WMSZ_TOPLEFT	Top-left corner
WMSZ_TOPRIGHT	Top-right corner



*lParam* is actually the rectangular dimensions of the window that is being resized. It is a **RECT** object.

## Window Moving

---

**WM\_MOVE**: When a window has been moved, the operating system needs to update its location. Therefore, the window sends a message called **WM\_MOVE**. Its syntax is:

```
OnMove(HWND hWnd, WM_MOVE, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that needs to be moved. The *wParam* and *lParam* values are not used.

**WM\_DESTROY**: Once the window has been used and the user has closed it, the window must send a message to the operating system to destroy it and reclaim the memory space it was using. The message is called **WM\_DESTROY** and its syntax is:

```
OnDestroy(HWND hWnd, WM_DESTROY, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that needs to be destroyed. The *wParam* and *lParam* values are not used.

## Window Destruction

---

**WM\_DESTROY**: Once the window has been used and the user has closed it, the window must send a message to the operating system to destroy it. The message sent is called **WM\_DESTROY** and its syntax is:

```
OnDestroy(HWND hWnd, WM_DESTROY, WPARAM wParam, LPARAM lParam);
```

*hWnd* is the window that is being destroyed. The *wParam* and *lParam* values are not used.

## Anytime Messages

---

### Introduction

---

The messages we have used so far belong to specific events generated at a particular time by a window. Sometimes in the middle of doing something, you may want to send a message regardless of what is going on. This is made possible by a function called **SendMessage()**. The syntax of the **SendMessage()** function is as follows:

```
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

The *hWnd* argument is the object or control that is sending the message. The *Msg* argument is the message to be sent. The *wParam* and the *lParam* values depend on the message that is being sent.

### Sending Messages

---

The advantage of using the **SendMessage()** function is that, when sending this message, it would target the procedure that can perform the task and this function would return only after its message has been processed. Because this (member) function can sometimes universally be used, that is by any control or object, the application cannot predict the type of message that **SendMessage()** is carrying. Therefore, (the probable disadvantage is that) you must know the (name or identity of the) message you are sending and you must provide accurate accompanying items (like sending a letter with the right stamp; imagine you send a sexy letter to your grand-mother in Australia about her already dead grand grand-father who is celebrating his first job while he has just become 5 years old).

In order to send a message using the **SendMessage()** function, you must know what message you are sending and what that message needs in order to be complete. For example, to change the caption of a window at any time, you can use the **WM\_SETTEXT** message. The syntax to use would be:

```
SendMessage(hWnd, WM_SETTEXT, wParam, lParam);
```

Obviously you would need to provide the text for the caption you are trying to change. This string is carried by the *lParam* argument as a null-terminated string. For this message, the *wParam* is ignored.

```
const char *Msg = "This message was sent";  
SendMessage(hWnd, WM_SETTEXT, 0, (LPARAM)(LPCTSTR)Msg);
```

## Object-Oriented Win32

---



# Window Objects

---

## Introduction

---

Every object you see on your screen that can be located, can be clicked, or moved, is called a window. As you may imagine, these window objects can be as different as the human eye can distinguished them. Some of these objects are icons, buttons, pictures, menu items, etc. As different as they are, there are also various ways of creating them.

There are three main ways you create a window, as we will learn throughout this site. One of the techniques you can use consists of creating a script in a resource file, as we have introduced scripts in lessons 1 and 2. Another technique you can use consists of calling one of the Win32 functions such as **CreateWindow()** or **CreateWindowEx()** to create a window; we have introduced it when creating the main window in all previous applications. The last option you have is to use your own class and customize the creation of a window.

## OOP Win32 Fundamentals

---

In C++, you probably learned that it is a good idea to make sure that the **main()** function be not crowded, which can make your program easier to read and troubleshoot when problems happen. You can also apply this to Win32 programming by separating tasks in their own units. Based on this, you can define functions that perform specific tasks such as creating or registering the window

To apply object oriented programming (OOP) to a Win32 application, you can create a class for each window you use in your application. Since each object is primarily a window, you can start with a general window that lays a foundation for other windows. Such a class can be created as follows (this is a primary window class; we will add methods to it as needed:

```
#ifndef WINCTRLS_H
#define WINCTRLS_H

#include <windows.h>

//-----
class WControl
{
public:
    WControl();
    virtual ~WControl();

    HWND Create(HINSTANCE hinst, LPCTSTR clsname,
                LPCTSTR wndname, HWND parent = NULL,
                DWORD dStyle = WS_OVERLAPPEDWINDOW,
                int x = CW_USEDEFAULT, int y = CW_USEDEFAULT,
                int width = 450, int height = 380);
    BOOL Show(int dCmdShow = SW_SHOWNORMAL);
    operator HWND();
protected:
    HWND hwnd;
    HINSTANCE mhInst;
public:
    HINSTANCE GetInstance();
```

```
private:
};
//-----

#endif // WINCTRLS_H
```

This class can be implemented as follows:

```
#include "winctrl.h"

//-----
WControl::WControl()
    : hwnd(0)
{
}
//-----
WControl::~WControl()
{
}
//-----
WControl::operator HWND()
{
    return hwnd;
}
//-----
HWND WControl::Create(HINSTANCE hinst, LPCTSTR clsname,
                    LPCTSTR wndname, HWND parent,
                    DWORD dStyle,
                    int x, int y,
                    int width, int height)
{
    hwnd = CreateWindow(clsname, wndname, dStyle,
        x, y, width, height, parent, NULL, hinst, NULL);

    return hwnd;
}
//-----
BOOL WControl::Show(int dCmdShow)
{
    BOOL CanShow = ::ShowWindow(hwnd, dCmdShow);

    if( CanShow )
        return TRUE;
    return FALSE;
}
//-----
HINSTANCE WControl::GetInstance()
{
    return mhInst;
}
//-----
```

To use this class, simply declare it in your application and call the necessary methods. Here is an example:

```
#include <windows.h>
#include "winctrl.h"
```

```

//-----
HINSTANCE hInst;
const char *ClsName = "BasicApp";
const char *WndName = "A Simple Window";
//-----
ATOM RegWnd(HINSTANCE hInst);
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);
//-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    // The message class of the application
    MSG      Msg;

    // Initialize the instance of this application
    hInst = hInstance;
    // create and register the application
    RegWnd(hInstance);
    // Create the window object
    WControl Ctrl;

    Ctrl.Create(hInst, ClsName, WndName);
    Ctrl.Show();

    // Process the messages
    while( GetMessage(&Msg, NULL, 0, 0) )
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    return Msg.wParam;
}
//-----
ATOM RegWnd(HINSTANCE hInst)
{
    WNDCLASSEX WndClsEx;

    // Create the application window
    WndClsEx.cbSize      = sizeof(WNDCLASSEX);
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc = WndProcedure;
    WndClsEx.cbClsExtra  = 0;
    WndClsEx.cbWndExtra  = 0;
    WndClsEx.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor     = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    WndClsEx.lpszMenuName = NULL;
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance   = hInst;
    WndClsEx.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

    // Register the application
    return RegisterClassEx(&WndClsEx);
}

```

```

//-----
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_CREATE:
            break;
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;

        default:
            return DefWindowProc(hWnd, Msg, wParam, lParam);
    }

    return 0;
}
//-----

```

## Win32 Object Programming

---

### The Application

---

In order to create an application, you must first create an application (sorry for the repetition). As we have seen so far, an application is created using the **WNDCLASS** or the **WNDCLASSEX** structures. Therefore, you can start with a class that would hold a member variable of this type. As some members of these structures are usually the same for most basic applications, you can create a class that uses default values and can then be easily initialized when it is time to create an application.

### ▼ Practical Learning: Initializing an Application

---

1. Start your programming environment
2. Create a Win32 project (If you are using Visual C++, create the project as empty) named **Win32OOP1**  
 If you are using C++ Builder, save the unit as Exercise  
 If you are using Visual C++, create a source file and name it Exercise
3. Create a header file. Save it as **WinApp** and type the following in it:

```

#pragma once
#include <windows.h>

//-----
class WApplication
{
public:
    // This constructor will initialize the application
    WApplication(HINSTANCE hInst, char *ClsName,
                WNDPROC WndPrc, LPCTSTR MenuName = NULL);

    // Class Registration
    void Register();

protected:
    // Global variable that holds the application
    WNDCLASSEX _WndClsEx;
};
//-----

```

4. Create a  
source file. Save it as **WinApp** and type the following in it:

```

5. #include "WinApp.h"

//-----
WApplication::WApplication(HINSTANCE hInst, char *ClsName,
                          WNDPROC WndPrc, LPCTSTR MenuName)
{
    // Initializing the application using the application member variable
    _WndClsEx.cbSize = sizeof(WNDCLASSEX);
    _WndClsEx.style = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS;
    _WndClsEx.lpfnWndProc = WndPrc;
    _WndClsEx.cbClsExtra = 0;
    _WndClsEx.cbWndExtra = 0;
    _WndClsEx.hInstance = hInst;
    _WndClsEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    _WndClsEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    _WndClsEx.hbrBackground = static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    _WndClsEx.lpszMenuName = MenuName;
    _WndClsEx.lpszClassName = ClsName;
    _WndClsEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
}
//-----
void WApplication::Register()
{
    RegisterClassEx(&_WndClsEx);
}
//-----

```

Save all

## The Window

---

As we have done in previous lessons, after initializing the application, you can create a window,

or the main window. This is done by calling either the **CreateWindow()** or the **CreateWindowEx()** functions. After creating the main window, you can display it, then process its messages.

## ▼ Practical Learning: Creating a Window

---

1. Create a header file. Save it as **MainWnd** and type the following in it:

```
#pragma once
#include <windows.h>

//-----
class WWindow
{
public:
    // We will use a default constructor to declare a window
    WWindow();
    // The Create() method will be used to initialize a window
    HWND Create(HINSTANCE hinst,
               LPCTSTR clsname,
               LPCTSTR wndname,
               HWND parent = NULL,
               DWORD dStyle = WS_OVERLAPPEDWINDOW,
               DWORD dxStyle = 0L,
               int x = CW_USEDEFAULT,
               int y = CW_USEDEFAULT,
               int width = CW_USEDEFAULT,
               int height = CW_USEDEFAULT);

    // This method will be used to display the window
    BOOL Show(int dCmdShow = SW_SHOWNORMAL);

    // Because each window is of type HWND, we will need a way
    // to recognize the window handle when used in our application
    operator HWND();

protected:
    // This will be a global handle available to
    // this and other windows
    HWND _hwnd;
};
//-----
```

2. Create a source file. Save it as **MainWnd** and type the following in it:
- 3.

```

#include "MainWnd.h"

//-----
WWindow::WWindow()
{
    // If we declare a window class with a default constructor,
    // we need to reset the window to a nothing
    _hwnd = NULL;
}
//-----
HWND WWindow::Create(HINSTANCE hinst,
                    LPCTSTR clsname,
                    LPCTSTR wndname,
                    HWND parent,
                    DWORD dStyle,
                    DWORD dxStyle,
                    int x,
                    int y,
                    int width,
                    int height)
{
    // When call the Create() method, we can use it to create a new window
    _hwnd = CreateWindowEx(dxStyle, clsname, wndname, dStyle, x, y, width,
                          height, parent, NULL, hinst, NULL);

    // We hope everything went alright and the window was created
    if( _hwnd != NULL )
        return _hwnd;
    // If something went wrong, for example if the window could not
    // be created, return a "nothing" window
    return NULL;
}
//-----
BOOL WWindow::Show(int dCmdShow)
{
    // We will display the main window as a regular object and update it
    if( ShowWindow(_hwnd, dCmdShow) && UpdateWindow(_hwnd) )
        return TRUE;
    return FALSE;
}
//-----
WWindow::operator HWND()
{
    // This overloaded operator allows us to use HWND anyway we want
    return _hwnd;
}
//-----

```

Save all

## **Main Application Creation**

There are some steps that you should/must follow to create an application. You start by initializing an application object, then create a window, display it, and process its messages. We have mentioned that the messages of an application are treated in a function pointer referred to as callback. Since C++ doesn't like or doesn't encourage function pointers to be members of a class (some libraries such as Borland VCL implemented in C++ Builder solve this

problem another way; the Microsoft .NET Framework also solves this problem somehow), the function should be created globally and passed to **WNDCLASS** or **WNDCLASSEX**.

## ▼ Practical Learning: Creating an Application

---

1. Open the Exercise.cpp source file and type the following in it (if using C++ Builder, make only the changes):



```

#include <windows.h>
#include "WinApp.h"
#include "MainWnd.h"

//-----
-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam);
//-----
-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG    Msg;
    char *ClsName = "Win32OOP";
    char *WndName = "Object-Oriented Win32 Programming";

    // Initialize the application class
    WApplication WinApp(hInstance, ClsName, MainWndProc);
    WinApp.Register();

    // Create the main window
    WWindow Wnd;
    Wnd.Create(hInstance, ClsName, WndName);
    // Display the main winow
    Wnd.Show();

    // Process the main window's messages
    while( GetMessage(&Msg, NULL, 0, 0) )
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    return 0;
}
//-----
-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            return 0;
    }

    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
//-----
-----

```

2.

Execute the application



3. Return to your programming environment

## Anatomy of a Window

---

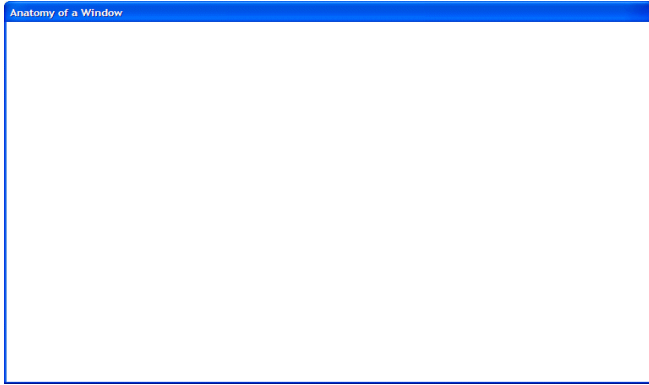
### The Presence of a Window

---

For the user to use an application, it must display the window that can be located on the screen. A window is primarily distinguishable from the rest of the screen by its being surrounded by borders. To create a window that has borders, add the **WS\_BORDER** flag to the *dwStyle* of the **CreateWindow()** or the **CreateWindowEx()** functions. Here is an example:

```
CreateWindow("AnatWnd",  
            "Anatomy of a Window",  
            WS_BORDER,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            NULL,  
            NULL,  
            hInstance,  
            NULL);
```

This would produce:



## The Title Bar: The Window Icon

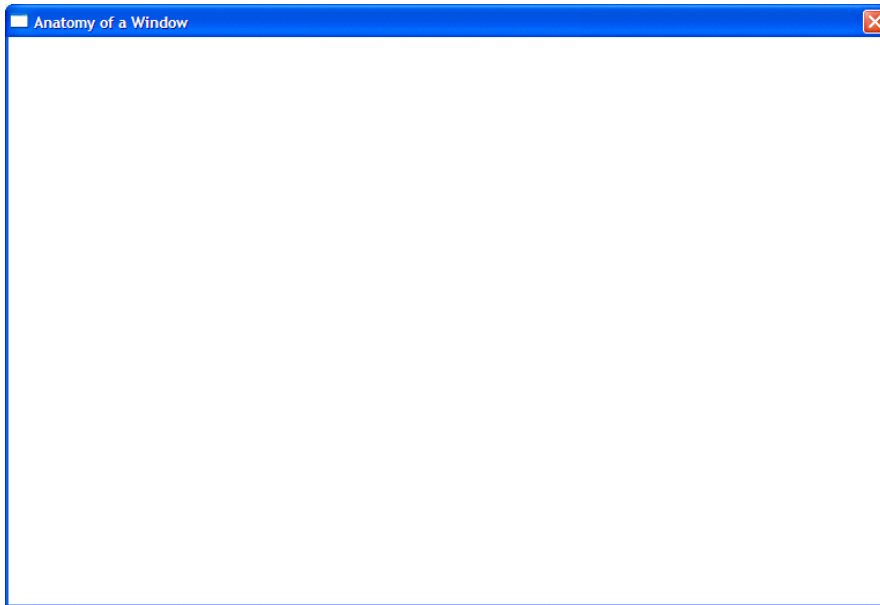
---

When a window comes up, it may start on top with a long bar called the title bar. If you want a window to be equipped with a title bar, add the **WS\_CAPTION** flag to the *dwStyle* of the **CreateWindow()** or the **CreateWindowEx()** functions.

The title bar itself is divided in three sections. On the left side, there may be a small picture we call an icon and it is primarily managed as a Windows resource, through a name. To display this icon, the window must be created with the **WS\_SYSMENU** flag to its list of styles. Here is an example:

```
CreateWindow("AnatWnd",  
            "Anatomy of a Window",  
            WS_BORDER | WS_CAPTION | WS_SYSMENU,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            CW_USEDEFAULT,  
            NULL,  
            NULL,  
            hInstance,  
            NULL);
```

This would produce:



By default, the operating system provides a simple icon named **IDI\_APPLICATION**. To use it, you pass it to the **LoadIcon()** function with the *hInstance* argument set to NULL. Instead of **LoadIcon()**, Microsoft suggests you use the **LoadImage()** function. Its syntax is:

```
HANDLE LoadImage(HINSTANCE hinst,
                 LPCTSTR lpszName,
                 UINT uType,
                 int cxDesired,
                 int cyDesired,
                 UINT fuLoad);
```

If you want to provide your own icon, first create it, preferably in two versions. The first version, with a dimension of 32x32 pixels is used to represent an application in Large View of certain windows such as My Documents or Windows Explorer. This version must be assigned to the **WNDCLASS::hIcon** or the **WNDCLASSEX::hIcon** member variable.

A second version of the same icon has a dimension of 16x16 pixels. It is used on the top-left corner of a window. It is also used all but the Large View or Windows Explorer or My Documents windows. This version is passed as the **WNDCLASSEX::hIconSm** member variable. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG         Msg;
    HWND        hWnd;
    WNDCLASSEX  WndClsEx;

    // Create the application window
    WndClsEx.cbSize       = sizeof(WNDCLASSEX);
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc  = WndProcedure;
    WndClsEx.cbClsExtra   = 0;
    WndClsEx.cbWndExtra   = 0;
    WndClsEx.hIcon       = static_cast<HICON>(LoadImage(hInstance,
```

```

        MAKEINTRESOURCE (IDI_ANATWND) ,
        IMAGE_ICON,
        32,
        32,
        LR_DEFAULTSIZE) );
WndClsEx.hCursor      = LoadCursor (NULL, IDC_ARROW);
WndClsEx.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
WndClsEx.lpszMenuName = NULL;
WndClsEx.lpszClassName = ClsName;
WndClsEx.hInstance    = hInstance;
WndClsEx.hIconSm     = static_cast<HICON> (LoadImage (hInstance,
        MAKEINTRESOURCE (IDI_ANATWND) ,
        IMAGE_ICON,
        16,
        16,
        LR_DEFAULTSIZE) );

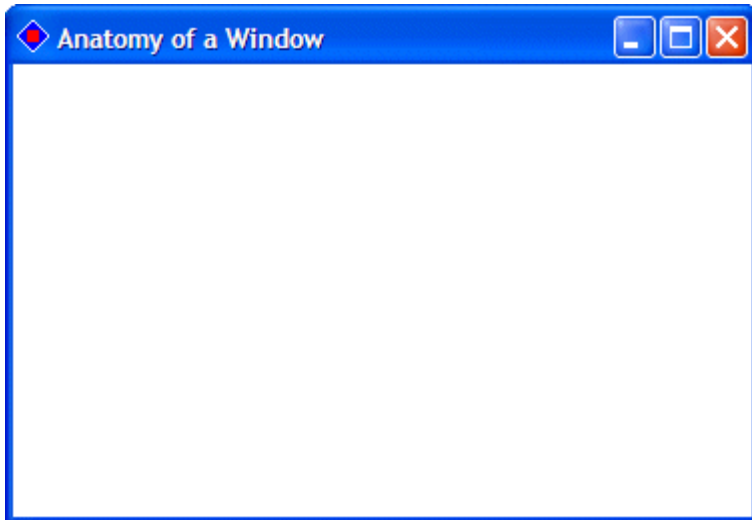
// Register the application
RegisterClassEx (&WndClsEx);

. . .

return 0;
}

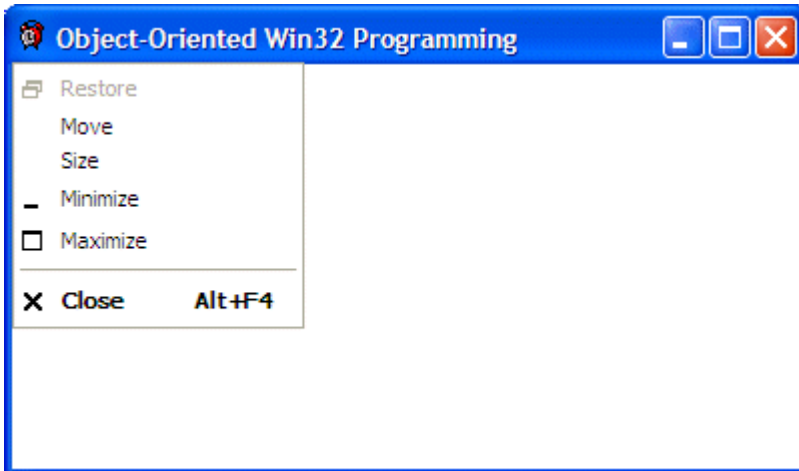
```

This would produce:



## The Title Bar: The Window Menu

Besides representing the application or the window, the Window icon also holds a group of actions called the Window Menu:



This menu allows the user to close, move, or perform other necessary operations. By default, the items in this menu are Restore, Move, Size, Minimize, Maximize, and Close. These actions are usually enough for a regular application. If for some reason you would like to modify this menu, you can.

The items on the menu are stored in a zero-based array with the top-most item having an index of 0, the second at 1, etc. Before taking any action on the menu, you should first open a handle to the system menu. This can be done by calling the **GetSystemMenu()** function. Its syntax is:

```
HMENU GetSystemMenu(HWND hWnd, BOOL bRevert);
```

To remove an item from the menu, you can call the **RemoveMenu()** function. Its syntax is:

```
BOOL RemoveMenu(HMENU hMenu, UINT uPosition, UINT uFlags);
```

Here is an example that removes the third menu item:

```
//-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam)
{
    HMENU hSysMenu;

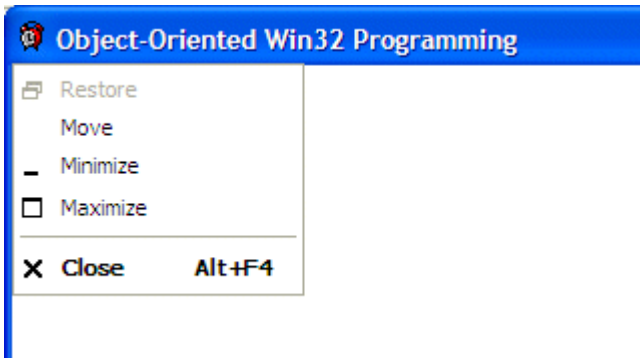
    switch(Msg)
    {
    case WM_CREATE:
        hSysMenu = GetSystemMenu(hWnd, FALSE);
        RemoveMenu(hSysMenu, 2, MF_BYPOSITION);

        return 0;

    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        return 0;
    }

    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
//-----
```

This would produce:



To add an item to the menu, you can call the **AppendMenu()** function. Its syntax is:

```
BOOL AppendMenu(HMENU hMenu,  
                UINT uFlags,  
                UINT_PTR uIDNewItem,  
                LPCTSTR lpNewItem);
```

To insert a new menu item inside of existing ones, you can call the **InsertMenu()** function.

## The Window Name

---

On the right side of the window icon, there is a long section that serves two main purposes. Like the system icon, the middle section of the title bar holds a menu. To access this menu, you can right-click the title bar. The menu that appears is the same as set in the window icon. If the menu of the window icon has been previously modified, it would appear as such when the title bar is right-clicked.

Another task of the main section of the title bar is to display the name of the window. As far as Windows and the user are concerned, the word or the group of words that appears in the middle of the title bar is the name of the window. In the programming world, this word or group of words is also referred to as the caption.

When you are a window using the **CreateWindow()** or the **CreateWindowEx()** functions, you can specify the caption by passing a string to the *lpWindowName* argument. To do this, you can provide a null-terminated string to the argument or declare a global string. Here is an example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
                  LPSTR lpCmdLine, int nCmdShow)  
{  
    WNDCLASSEX    WndClsEx;  
  
    . . .  
  
    RegisterClassEx(&WndClsEx);  
  
    CreateWindow(ClsName, WndName,  
}
```

After the main window of the application has been created, you can change its caption easily and there are at least two main ways you can do this. To change the caption of a window, you can call the **SetWindowText()** function. Its syntax is:

```
BOOL SetWindowText(HWND hWnd, LPCTSTR lpString);
```

Here is an example:

```
//-----  
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,  
                             WPARAM wParam, LPARAM lParam)  
{  
    switch(Msg)  
    {  
        case WM_CREATE:  
  
            SetWindowText(hWnd, "FunctionX Tutorials");  
  
            return 0;  
  
        case WM_DESTROY:  
            PostQuitMessage(WM_QUIT);  
            return 0;  
    }  
  
    return DefWindowProc(hWnd, Msg, wParam, lParam);  
}  
//-----
```

Alternatively, you can send the **WM\_SETTEXT** message using the **SendMessage()** function. With this message, the *wParam* parameter is not used. The *lParam* argument is a string that holds the caption.







To retrieve the current caption of the window, you can call the **GetWindowText()** function. Its syntax is:

```
int GetWindowText(HWND hWnd, LPTSTR lpString, int nMaxCount);
```

To perform the same operation, you can send the **WM\_GETTEXT** message.

## The System Menu



---

On the right side of the title bar, a window can display one to three buttons: Minimize  or   
, Maximize  or , Close  or . The presence or absence of these buttons can be managed from various alternatives. Although they appear as buttons, they are not resources. In some cases, only the Close button displays. In some other cases, the buttons would appear in a combination of three. In this case, all of the buttons might be usable or one of the buttons might be disabled. To start, if you want a window to have any of these buttons, add the **WS\_SYSMENU** flag to the *dwStyle* of the **CreateWindow()** or the **CreateWindowEx()** functions.



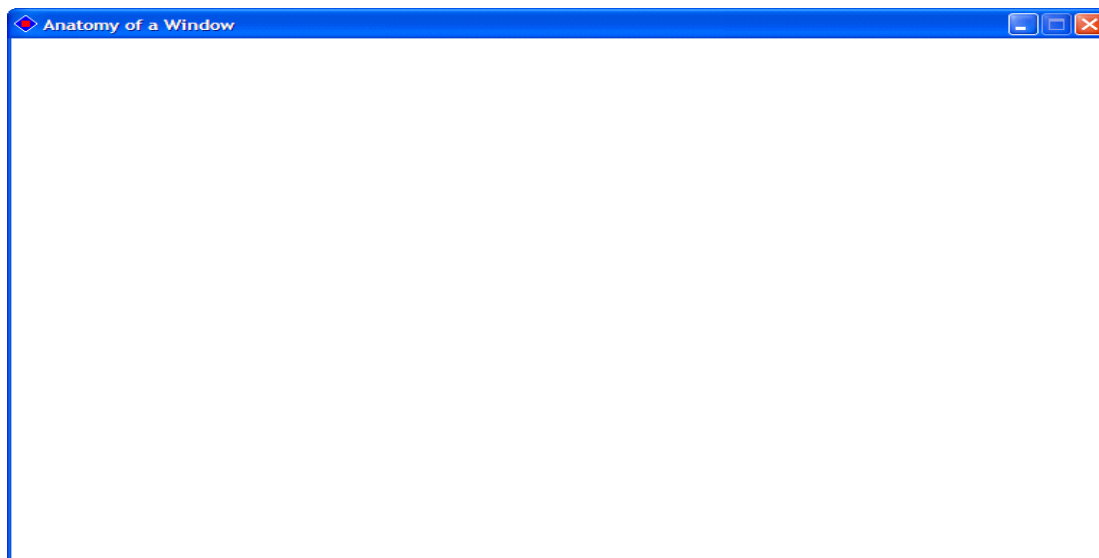
## The Minimize Button

---

The Minimize button appears as  or . The Minimize button cannot appear on its own, only with the Maximize and the Close buttons. To display the Minimize button, when creating the window, add the **WS\_MINIMIZEBOX** flag to the *dwStyle* of the **CreateWindow()** or the **CreateWindowEx()** functions. Here is an example:

```
CreateWindow("AnatWnd",
    "Anatomy of a Window",
    WS_BORDER | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL);
```

This would produce:



As seen on the above picture, when you add only the **WS\_MINIMIZEBOX** style, a Minimize button appears with a disabled Maximize button. If the user clicks a Minimize button, the window disappears from the screen and becomes represented by a button on the taskbar. The button that represents a window on the taskbar also displays its application icon and its window name.

To minimize a window, the user can click its Minimize button, to programmatically minimize a window, you can call the **ShowWindow()** function. Its syntax is:

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

The first argument specifies the window on which you are taking the action. If you are minimizing the window, the second argument can have a value of **SW\_MINIMIZE**. Here is an

example:

```
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_LBUTTONDOWN:
            ShowWindow(hWnd, SW_MINIMIZE);
            break;

        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
    }



    // Process the left-over messages
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
```

At anytime, to find out whether a window is minimized, you can call the **IsIconic()** function. Its syntax is:

```
BOOL IsIconic(HWND hWnd);
```

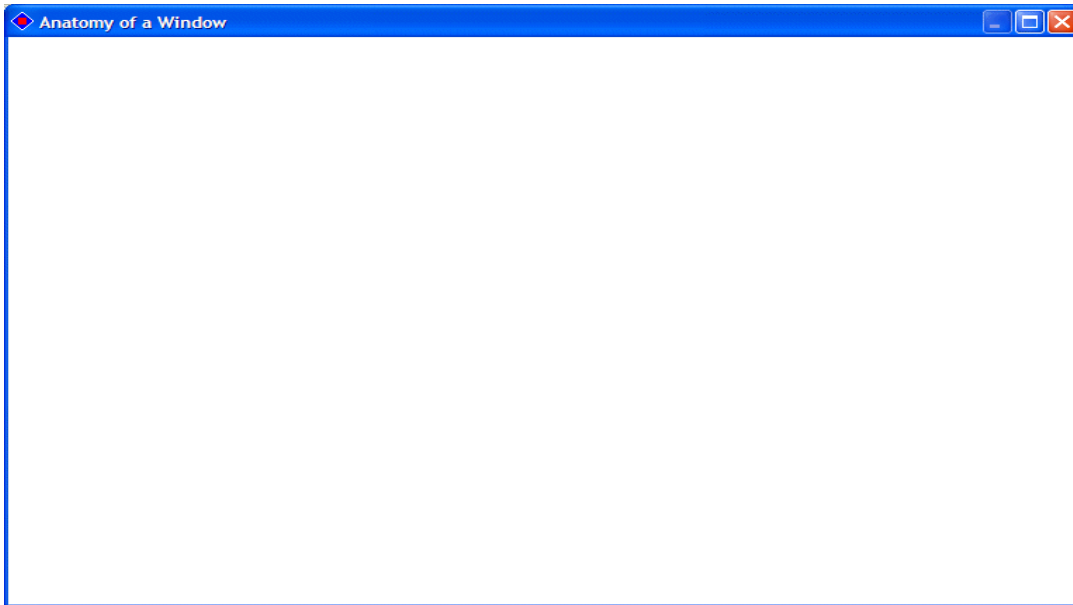
The argument of this function is the window whose minimized state you are checking.


## The Maximize Button

The Maximize button appears as  or . Like Minimize, the Maximize button cannot appear on its own, only with the Minimize and the Close buttons. To display the Maximize button, when creating the window, add the **WS\_MAXIMIZEBOX** flag to the *dwStyle* of the **CreateWindow()** or the **CreateWindowEx()** functions. Here is an example:

```
CreateWindow("AnatWnd",
             "Anatomy of a Window",
             WS_BORDER | WS_CAPTION | WS_SYSMENU | WS_MAXIMIZEBOX,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             NULL,
             NULL,
             hInstance,
             NULL);
```

This would produce:



If you add only the **WS\_MAXIMIZEBOX** flag without **WS\_MINIMIZEBOX**, a Maximize button appears with a disabled Minimize button. If the user clicks the Maximize button, the size of the window increases to occupy the whole screen. At this time, the window is described as being maximized. When a window is maximized, the Maximize button changes into a Restore button . If the user clicks the Restore button, the window comes back to the size it had the last time it was not maximized (most of this information may be stored in the Registry by the operating system; this means that the Registry can "remember" the location, dimensions, and maximized state even if it occurred days or months before).

While the user can maximize a window by clicking the Maximize button, to programmatically maximize a window, you can call the **ShowWindow()** function, passing the second argument as **SW\_MAXIMIZE**. Here is an example:

```
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_LBUTTONDOWN:
            ShowWindow(hWnd, SW_MAXIMIZE);
            break;

        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
    }

    // Process the left-over messages
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
```

At anytime, to find out whether a window is maximized, you can call the **IsZoomed()** function. Its syntax is:

```
BOOL IsZoomed (HWND hWnd) ;
```

The argument of this function is the window whose maximized state you are checking.

## Types of Windows

---

### The Window's Real Estate

---

#### Application's Instance

---

A window is referred to as parent when it can be used to host, hold, or carry other windows. For examples, when the computer starts, it draws its main screen, also called the desktop, which occupies the widest area that the monitor screen can offer. This primary window becomes the host of all other window that will display as long as the computer is own. This desktop is also a complete window in its own right. As mentioned already, to get its handle, you can call the **GetDesktopWindow()** function.

After the desktop has been created, a window of yours can display if the user starts your application. This means that an application must have been created for the user to use it. When the user opens an application, we also say that the application has been instantiated or an instance of the application has been created. Based on this, any time you create an application, you must provide an instance of it. This allows the operating system to manage your application with regards to its interaction with the user and also its relationship with other resources. Therefore, you must always create an instance for your application. This is taken care of by the first argument of the **WinMain()** function.

If an application has already been created, to get its instance, you can call the **GetWindowLong()** function. Its syntax is:

```
LONG GetWindowLong (HWND hWnd, int nIndex) ;
```

Although this function is used for many other reasons, it can also help you get the instance of an application. To do this, pass the first argument as the handle to a window of the application you are examining and pass the second argument as **GWL\_HINSTANCE**.

### Window Parenting

---

There are two types of windows or object you will deal with in your applications. The type referred to here is defined by the relationship a window has with regards to other windows that are part of an application:

- **Parent:** a window is referred to as a parent when there are, or there can be, other windows that depend on it. For example, the toolbar of your browser is equipped with some buttons. The toolbar is a parent to the buttons. When a parent is created, it "gives life" to other windows that can depend on it.
- **Child:** A window is referred to as child when its existence and especially its visibility depend on another window called its parent.

When a parent is created, made active, or made visible, it gives existence and visibility to its children. When a parent gets hidden, it also hides its children. If a parent moves, it moves with its children. The children keep their positions and dimensions inside the parent. When a parent is destroyed, it also destroys its children (sometimes it does not happen so smoothly; a parent may make a child unavailable but the memory space the child was occupying after the parent has been destroyed may still be in use, sometimes filled with garbage, but such memory may not be available to other applications until you explicitly recover it).

Child controls depend on a parent because the parent "carries", "holds", or hosts them. All of the Windows controls you will use in your applications are child controls. A child window can also be a parent of another control. For example, a toolbar of the browser is the parent of the buttons on it. If you close or hide the toolbar, its children disappear. At the same time, the toolbar is a child of the application's frame. If you close the application, the toolbar disappears, along with its own children. In this example, the toolbar is a child of the frame but is a parent to its buttons.

After initializing an application with either the **WNDCLASS**, or the **WNDCLASSEX** structure and registering it, as we have done so far, you must create the primary parent of all objects of your class. This is usually done with either the **CreateWindow()** or the **CreateWindowEx()** function. Here is an example:

```
HWND hWndParent;

// Create the parent window
hWndParent = CreateWindowEx(0, ClassName, StrWndName,
                           WS_OVERLAPPEDWINDOW,
                           0, 100, 140, 320,
                           NULL, NULL, hInstance, NULL);
```

## A Window's Childhood

---

After creating the main window, you can use it as a parent for other windows. To specify that a window is a child of another window, when creating it with either the **CreateWindow()** or the **CreateWindowEx()** function, pass the handle of the parent as the *hWndParent* argument. Here is an example:

```
// Create a window
CreateWindowEx(0, WndClassName, CaptionOrText,
               ChildStyle, Left, Top, Width, Height,
               hWndParent, NULL, hInstance, NULL);
```

If a window is a child of another window, to get a handle to its parent, you can call the **GetParent()** function. Its syntax is:

```
HWND GetParent(HWND hWnd);
```

The *hWnd* argument is a handle to the child window whose parent you want to find out. Alternatively, you can also use the **GetWindowLong()** function, passing the second argument as **GWL\_HWNDPARENT**, to get a handle to the parent of a window.

## The Borders of a Window

---

To distinguish a particular window from the other objects on a screen, a window can be defined by surrounding borders on the left, the top, the right, and the bottom. One of the effects the user may want to control on a window is its size. For example, the user may want to narrow, enlarge, shrink, or heighten a window. To do this, a user would position the mouse on one of the borders, click and drag in the desired direction. This action is referred to as resizing a window. For the user to be able to change the size of a window, the window must have a special type of border referred to as a thick frame. To provide this border, apply or add the **WS\_THICKFRAME** style:

```
CreateWindow(ClsName, WndName,  
            WS_VISIBLE | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_THICKFRAME);
```

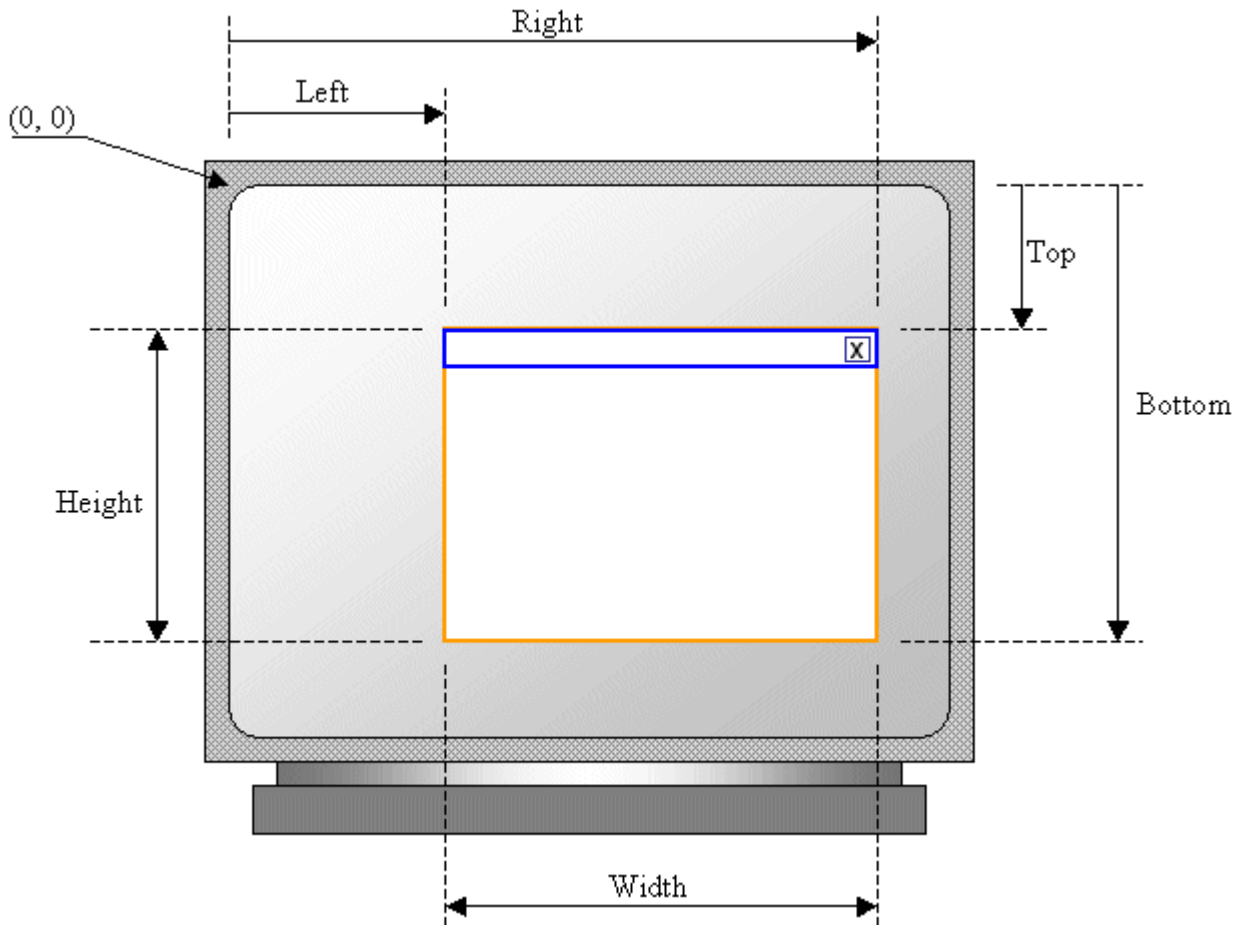
Because many windows will need this functionality, a special style can combine them and it is called **WS\_OVERLAPPEDWINDOW**. Therefore, you can create a resizable window as follows:

```
CreateWindow(ClsName, WndName, WS_OVERLAPPEDWINDOW,
```

## Window's Location and Size

---

The location of a window is defined by the distance from the left border of the monitor to the window's left border and its distance from the top border of the monitor to its own top border. The size of a window is its width and its height. These can be illustrated for a main window frame as follows:



For a Win32 application, the original distance from the left border of the monitor is passed as the `x` argument to the **CreateWindow()** or the **CreateWindowEx()** function. The distance from top is specified using the `y` argument. The `x` and `y` arguments define the location of the window. The distance from the left border of the monitor to the right border of the window is specified as the `nWidth` argument. The distance from the top border of the monitor to the lower border of the window is specified with the `nHeight` value.

If you cannot make up your mind for these four values, you can use the **CW\_USEDEFAULT** (*when-Creating-the-Window-USE-the-DEFAULT-value*) constant for either one or all four arguments. In such a case, the compiler would select a value for the argument. Here is an example:

```

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    HWND hWnd = CreateWindow(ClsName,
                            WndName,
                            WS_OVERLAPPEDWINDOW,

```

```

        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
    return 0;
}

```

## Displaying the Window

---

Once a window has been created and if this was done successfully, you can display it to the user. This is done by calling the **ShowWindow()** function. Its syntax is:

```

BOOL ShowWindow(HWND hWnd, int nCmdShow);

```

The *hWnd* argument is a handle to the window that you want to display. It could be the window returned by the **CreateWindow()** or the **CreateWindowEx()** function.

The *nCmdShow* specifies how the window must be displayed. Its possible values are:

	Value	Description
	SW_SHOW	Displays a window and makes it visible
	SW_SHOWNORMAL	Displays the window in its regular size. In most circumstances, the operating system keeps track of the last location and size a window such as Internet Explorer or My Computer had the last time it was displaying. This value allows the OS to restore it.
	SW_SHOWMINIMIZED	Opens the window in its minimized state, representing it as a button on the taskbar
	SW_SHOWMAXIMIZED	Opens the window in its maximized state
	SW_SHOWMINNOACTIVE	Opens the window but displays only its icon. It does not make it active
	SW_SHOWNA	As previous
	SW_SHOWNOACTIVATE	Retrieves the window's previous size and location and displays it accordingly
	SW_HIDE	Used to hide a window
	SW_MINIMIZE	Shrinks the window and reduces it to a button on the taskbar
	SW_MAXIMIZE	Maximizes the window to occupy the whole screen area
	SW_RESTORE	If the window was minimized or maximized, it would be restored to its previous location and size

To show its presence on the screen, the window must be painted. This can be done by calling the **UpdateWindow()** function. Its syntax is:

```

BOOL UpdateWindow(HWND hWnd);

```

This function simply wants to know what window needs to be painted. This window is specified by its handle. Here is an example:

```

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)

```



```

{
    WNDCLASSEX WndClsEx;

    . . .

    RegisterClassEx(&WndClsEx);

    HWND hWnd = CreateWindow(ClsName, WndName, WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL);

    if( !hWnd ) // If the window was not created,
        return 0; // stop the application

    ShowWindow(hWnd, SW_SHOWNORMAL);
    UpdateWindow(hWnd);

    return 0;
}

```

## The Multiple Document Interface (MDI)

---

### Introduction

---

A multiple document is the type of application that uses a main, external window that acts as a frame and hosts other, floating window that act as its children. This concept allows the user to open more than one document at a time in an application, making it possible to switch from one document to another without closing the application.

### MDI Application Creation

---

You start an MDI like the types of applications we have created so far. One of the primary differences is that the window procedure must return the **DefFrameProc()** function. Its syntax is:

```

LRESULT DefFrameProc(HWND hWnd,
                    HWND hWndMDIClient,
                    UINT uMsg,
                    WPARAM wParam,
                    LPARAM lParam);

```

## The Graphical Device Interface

---

### Introduction to the GDI

---

#### The Device Context

---

Imagine you want to draw an orange. You can pick up a piece of stone and start drawing somewhere. If you draw on the floor, the next rain is likely to wipe your master piece away. If you draw on somebody's wall, you could face a law suit. Nevertheless, you realize that, to draw, you need at least two things besides your hands and your imagination: a platform to draw on and a tool to draw with.

As it happens, drawing in a studio and drawing on the computer have differences. To draw in real life, the most common platform is probably a piece of paper. Then, you need a pen that would show the evolution of your work. Since a pen can have or use only one color, depending on your goal, one pen may not be sufficient, in which case you would end up with quite a few of them. Since the human hand sometimes is not very stable, if you want to draw straight line, you may need a ruler. Some other tools can also help you draw geometric figures faster.

A device context is everything under one name. It is an orchestra, an ensemble of what need in order to draw. It includes the platform you draw on, the dimensioning of the platform, the orientation and other variations of your drawing, the tools you need to draw on the platform, the colors, and various other accessories that can complete your imagination.

When using a computer, you certainly cannot position tools on the table or desktop for use as needed. To help with drawing on the Windows operating system, Microsoft created the Graphical Device Interface, abbreviated as GDI. It is a set of classes, functions, variables, and constants that group all or most of everything you need to draw on an application. The GDI is provided as a library called Gdi.dll and is already installed on your computer.

## Grabbing the Device Context

---

As mentioned already, in order to draw, you need at least two things: a platform and a tool. The platform allows you to know what type of object you are drawing on and how you can draw on it. On a Windows application, you get this platform by creating a device context.

A device context is actually a whole class that provides the necessary drawing tools to perform the job. For example, it provides functions for selecting the tool to use when drawing. It also provides functions to draw text, lines, shapes etc.

**HDC:** This is the most fundamental class to draw in your applications. It provides all of the primary functions used to perform the basic drawing steps. In order to use this class, first declare a variable from it. Then call the **BeginPaint()** function to initialize the variable using the **PAINSTRUCT** class. Once the variable has been initialized, you can use it to draw. After using the device context call the **EndPaint()** function to terminate the drawing.

## The Process of Drawing

---

### Getting a Device Context

---

In order to draw using a device context, you must first declare an **HDC** variable. This can be done as follows:

```
HDC hdc;
```

After declaring this variable, you must prepare the application to paint by initializing it with a call to the **BeginPaint()** function. The syntax of the **BeginPaint()** function is:

```
HDC BeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint);
```

The *hWnd* argument is a handle to the window on which you will be painting

The *lpPaint* argument is a pointer to the **PAINTSTRUCT** structure. This means that, the **BeginPaint()** function returns two values. It returns a device context as HDC and it returns information about the painting job that was performed. That painting job is stored in a **PAINTSTRUCT** value. The **PAINTSTRUCT** structure is defined as follows:

```
typedef struct tagPAINTSTRUCT {  
    HDC  hdc;  
    BOOL fErase;  
    RECT rcPaint;  
    BOOL fRestore;  
    BOOL fIncUpdate;  
    BYTE rgbReserved[32];  
} PAINTSTRUCT, *PPAINTSTRUCT;
```

After initializing the device context, you can call a drawing function or perform a series of calls to draw. After painting, you must let the operating system know by calling the **EndPaint()** function. Its syntax is:

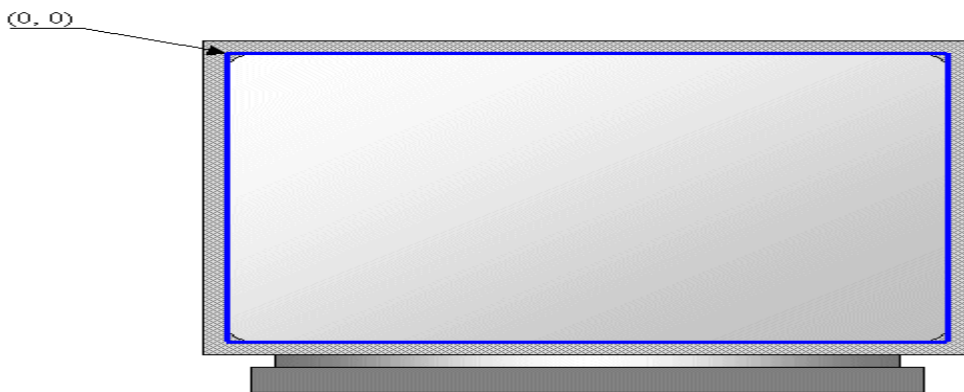
```
BOOL EndPaint(HWND hWnd, CONST PAINTSTRUCT *lpPaint);
```

Painting with the **BeginPaint()** and **EndPaint()** functions must be performed in the **WM\_PAINT** message.

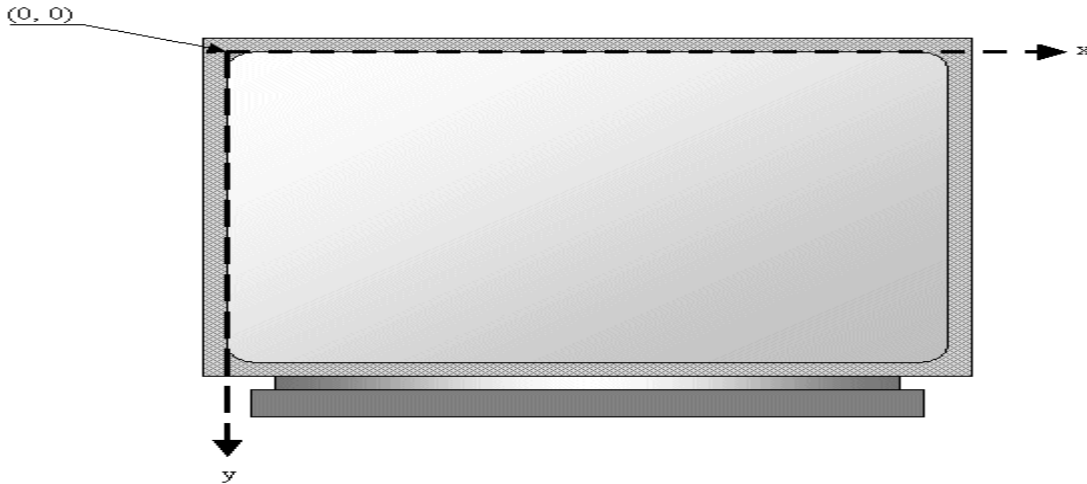
## Starting a Device Context's Shape

---

To keep track of the various drawings, the device context uses a coordinate system that has its origin (0, 0) on the top-left corner of the desktop:



Anything that is positioned on the screen is based on this origin. This coordinate system can get the location of an object using a horizontal and a vertical measurements. The horizontal measures are based on an x axis that moves from the origin to the right right direction. The vertical measures use a y axis that moves from the origin to the bottom direction:



This means that, if you start drawing something such as a line, it would start on the origin and continue where you want it to stop.

## GDI Fundamental Shapes

---

### Line-Based Shapes

---

#### Introduction

---

As mentioned in the previous lesson, in order to display something in a graphical application, you must draw that thing. The result of your drawing can be called a shape if it displays a recognizable figure. Sometimes it will simply be referred to as a graphic. The fundamental and easier shapes you can draw are geometric and they are the lines, rectangles, ellipse, etc. Of course, there are more complicated or advanced shapes than that.

```

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

//-----
HWND hWnd;
const char ClsName[] = "GDIFund";
const char WindowCaption[] = "GDI Fundamentals";
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
//-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG          Msg;
    WNDCLASSEX  WndClsEx;

    WndClsEx.cbSize          = sizeof(WNDCLASSEX);
    WndClsEx.style          = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc    = WndProc;
    WndClsEx.cbClsExtra    = NULL;
    WndClsEx.cbWndExtra    = NULL;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIcon         = LoadIcon(hInstance, IDI_APPLICATION);
    WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName  = NULL;
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hIconSm       = LoadIcon(hInstance, IDI_APPLICATION);

    RegisterClassEx(&WndClsEx);

    hWnd = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                          ClsName,
                          WindowCaption,
                          WS_OVERLAPPEDWINDOW,
                          100,
                          120,
                          640,
                          480,
                          NULL,
                          NULL,
                          hInstance,
                          NULL);

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

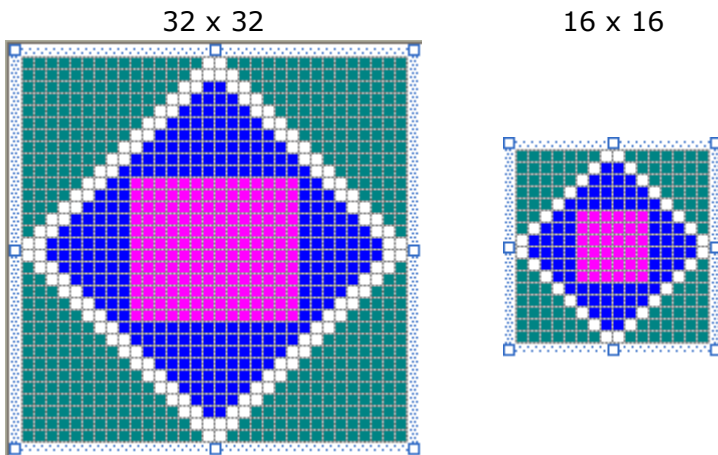
    while( GetMessage(&Msg, NULL, 0, 0) )
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    return 0;
}
//-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)

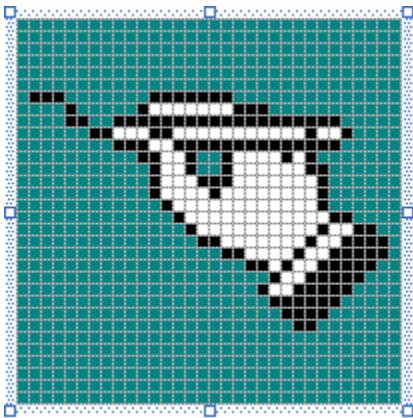
```

## ▼ Practical Learning: Introducing GDI Shapes

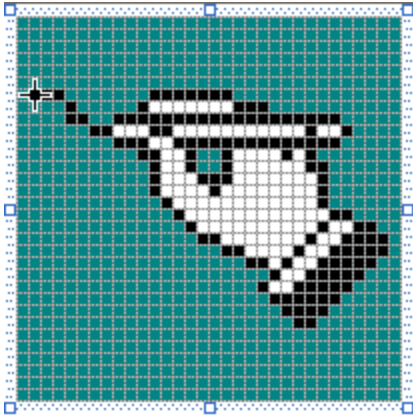
1. Start your programming environment and create a Win32 project or application. If you are using Visual C++, create it as an empty project
2. Save it in a new folder named **RainDrop1**  
If you are using C++ Builder 6 or Dev-C++, save the unit as **Exercise** and save the project as **RainDrop1**  
If you are using Visual C++ or C++BuilderX, create a new source file and name it **Exercise**
3. Create or add a new icon. Design it as follows:



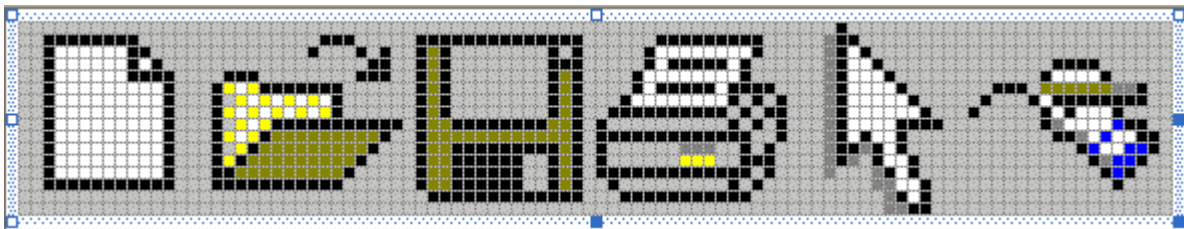
4. it as **raindrop.ico**  
Set its ID to **IDI\_RAINDROP** and save
5. Create or add a new cursor designed as follows:



6. Set its hot spot just at the tip of the left line



7. Save it as **IDC\_FREEHAND**
8. Create or add a new 16 x 96 bitmap and design it as follows:



9. Set its ID to **IDB\_STANDARD** and save it as **standard.bmp**
10. Create a menu as follows:

Caption	ID	Prompt
&File		
&New\tCtrl+N	IDM_FILE_NEW	Creates a new document\nNew
&Open...\tCtrl+O	IDM_FILE_OPEN	Opens an existing document\nOpen
&Save\tCtrl+S	IDM_FILE_SAVE	Saves the active document\nSave
Save &As...	IDM_FILE_SAVEAS	Custom saves the active document\nSave As
-		
&Print...\tCtrl+P	IDM_FILE_PRINT	Prints the current document\nPrint
-		
E&xit	IDM_FILE_EXIT	Closes the application\nExit
&Draw		
&Arrow	IDM_DRAW_ARROW	No tool selected\nNo Tool
&Free Hand	IDM_DRAW_FREEHAND	Draws with a free hand\nFree Hand

11. ID of the menu is **IDR\_MAIN\_MENU**
12. Save the resource file as RainDrop2.rc and add it to your project (it is automatically done in Visual C++ .NET)
13. Create a header file named WinApp.h and type the following in it:

```

#pragma once
#include <windows.h>

//-----
class WApplication
{
public:
    // This constructor will initialize the application
    WApplication();
    void Create(HINSTANCE hInst, char *ClasName,
               WNDPROC WndPrc, LPCTSTR MenuName = NULL);

    // Class Registration
    void Register();

protected:
    // Global variable that holds the application
    WNDCLASSEX _WndClsEx;
};
//-----

```

14.

source file named WinApp.cpp and type the following in it:

Create a

15.



```

#include "WinApp.h"
#include "resource.h"

//-----
WApplication::WApplication()
{
}

void WApplication::Create(HINSTANCE hInst, char *ClsName,
                          WNDPROC WndPrc, LPCTSTR MenuName)
{
    // Initializing the application using the application member variable
    _WndClsEx.cbSize = sizeof(WNDCLASSEX);
    _WndClsEx.style = CS_VREDRAW | CS_HREDRAW | CS_DBLCLKS;
    _WndClsEx.lpfWndProc = WndPrc;
    _WndClsEx.cbClsExtra = 0;
    _WndClsEx.cbWndExtra = 0;
    _WndClsEx.hInstance = hInst;
    _WndClsEx.hIcon = static_cast<HICON>(LoadImage(hInst,
                                                    MAKEINTRESOURCE(IDI_RAINDROP),
                                                    IMAGE_ICON,
                                                    32,
                                                    32,
                                                    LR_DEFAULTSIZE));
    _WndClsEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    _WndClsEx.hbrBackground = static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    _WndClsEx.lpszMenuName = MAKEINTRESOURCE(IDR_MAIN_MENU);
    _WndClsEx.lpszClassName = ClsName;
    _WndClsEx.hIconSm = static_cast<HICON>(LoadImage(hInst,
                                                    MAKEINTRESOURCE(IDI_RAINDROP),
                                                    IMAGE_ICON,
                                                    16,
                                                    16,
                                                    LR_DEFAULTSIZE));
}
//-----
void WApplication::Register()
{
    RegisterClassEx(&_WndClsEx);
}
//-----

```

Create a header file named MainWnd.h and type the following in it:

```

#pragma once
#include <windows.h>

//-----
class WWindow
{
public:
    // We will use a default constructor to declare a window
    WWindow();
    // The Create() method will be used to initialize a window
    HWND Create(HINSTANCE hinst,
               LPCTSTR clsname,
               LPCTSTR wndname,
               HWND parent = NULL,
               DWORD dStyle = WS_OVERLAPPEDWINDOW,
               DWORD dxStyle = 0L,
               int x = CW_USEDEFAULT,
               int y = CW_USEDEFAULT,
               int width = CW_USEDEFAULT,
               int height = CW_USEDEFAULT);

    // This method will be used to display the window
    BOOL Show(int dCmdShow = SW_SHOWNORMAL);

    // Because each window is of type HWND, we will need a way
    // to recognize the window handle when used in our application
    operator HWND();

    // Accessories
public:
    void SetText(LPCTSTR strCaption);

protected:
    // This will be a global handle available to
    // this and other windows
    HWND _hwnd;
};
//-----

```

16. source file named MainWnd.cpp and type the following in it:

Create a

```

#include "MainWnd.h"

//-----
WWindow::WWindow()
{
    // If we declare a window class with a default constructor,
    // we need to reset the window to a nothing
    _hwnd = NULL;
}
//-----

HWND WWindow::Create(HINSTANCE hinst,
                    LPCTSTR clsname,
                    LPCTSTR wndname,
                    HWND parent,
                    DWORD dStyle,
                    DWORD dxStyle,
                    int x,
                    int y,
                    int width,
                    int height)
{
    // When call the Create() method, we can use it to create a
    new window
    _hwnd = CreateWindowEx(dxStyle, clsname, wndname, dStyle, x,
y, width,
                                height, parent, NULL, hinst, NULL);

    // We hope everything went alright and the window was created
    if( _hwnd != NULL )
        return _hwnd;
    // If something went wrong, for example if the window could
not
    // be created, return a "nothing" window
    return NULL;
}
//-----

BOOL WWindow::Show(int dCmdShow)
{
    // We will display the main window as a regular object and
    update it
    if( ShowWindow(_hwnd, dCmdShow) && UpdateWindow(_hwnd) )
        return TRUE;
    return FALSE;
}
//-----

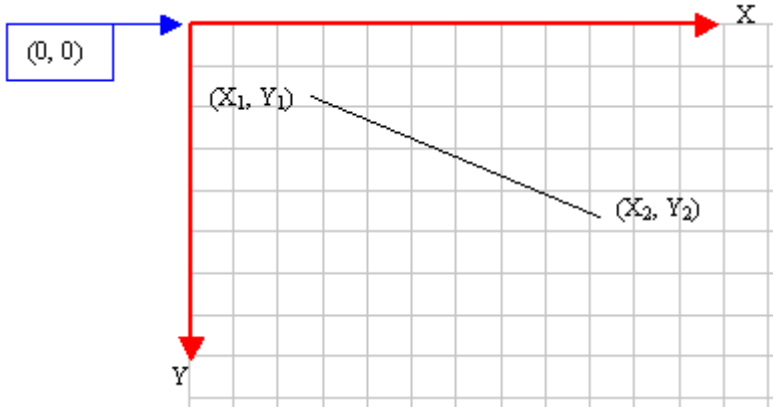
WWindow::operator HWND()
{
    // This overloaded operator allows us to use HWND anyway we
    want
    return _hwnd;
}
//-----

```

## Lines

---

A line is a junction of two points. This means that a line has a beginning and an end:



The beginning and the end are two distinct points. In real life, before drawing, you should define where you would start. To help with this, you can use **MoveToEx()** function. Its syntax is:

```
BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint);
```

The origin of a drawing is specified as the (X, Y) point.

To end the line, you use the **LineTo()** function. Its syntax is:

```
BOOL LineTo(HDC hdc, int nXEnd, int nYEnd);
```

The end of a line can be defined by its horizontal (nXEnd) and its vertical measures (nYEnd).

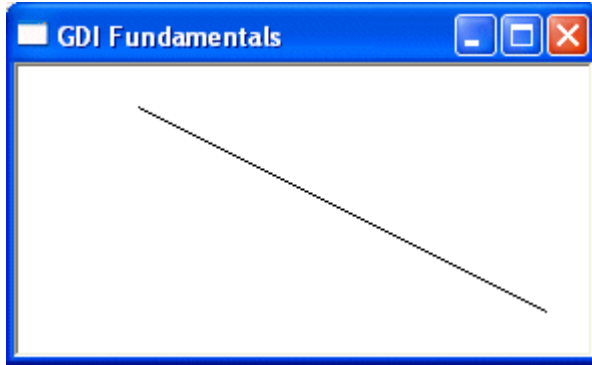
Here is an example that draws a line starting at a point defined as (10, 22) coordinates and ending at (155, 64):

```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hdc;  
    PAINTSTRUCT Ps;  
  
    switch(Msg)  
    {  
    case WM_PAINT:  
        hdc = BeginPaint(hWnd, &Ps);  
        MoveToEx(hdc, 60, 20, NULL);  
        LineTo(hdc, 264, 122);  
        EndPaint(hWnd, &Ps);  
        break;  
    case WM_DESTROY:  
        PostQuitMessage(WM_QUIT);  
        break;  
    default:
```

```

        return DefWindowProc (hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



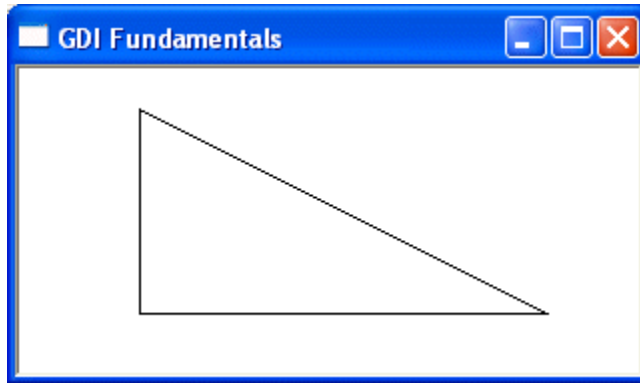
We have mentioned that the **MoveToEx()** function is used to set the starting position of a line. When using **LineTo()**, the line would start from the **MoveToEx()** point to the **LineTo()** end. As long as you do not call **MoveToEx()**, any subsequent call to **LineTo()** would draw a line from the previous **LineTo()** to the new **LineTo()** point. You can use this property of the **LineTo()** function to draw various lines. Here is an example:

```

//-----
-----
LRESULT CALLBACK WndProc (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

    switch (Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint (hWnd, &Ps);
        MoveToEx (hDC, 60, 20, NULL);
        LineTo (hDC, 60, 122);
        LineTo (hDC, 264, 122);
        LineTo (hDC, 60, 20);
        EndPaint (hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage (WM_QUIT);
        break;
    default:
        return DefWindowProc (hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```

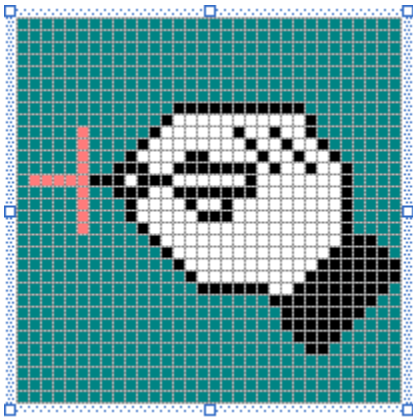


## ▼ Practical Learning: Drawing a Line

1. Under the Arrow menu item of the Draw category, add a new menu item Identified as **IDM\_DRAW\_LINE** of **&Line** with a Prompt of **Draws a straight line\nLine**
2. Add a new bitmap to the right of the IDB\_STANDARD resource as follows:



3. Save the bitmap
4. Design a new cursor as follows:



5. Set its hot spot to the intersection of the left cross
6. Set its ID to **IDC\_LINE** and save it as line.cur
7. To be able to draw a line, change the Exercise.cpp source file as follows:

```

//-----
-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    static BOOL IsDrawing = FALSE;
    static int StartX, StartY;
    static int EndX, EndY;
    UINT iButtonState;

    switch(Msg)
    {
    case WM_CREATE:
        Exo.CreateStandardToolbar(hWnd, Exo.hInst);
        SendMessage(Exo.hWndToolbar, TB_SETSTATE,
IDM_DRAW_ARROW, TBSTATE_CHECKED | TBSTATE_ENABLED);
        return 0;

    case WM_ACTIVATE:
        Exo.Wnd.SetText("RainDrop - Untitled");
        return 0;

    case WM_LBUTTONDOWN:
        // Find out if the Line button is clicked
        iButtonState = SendMessage(Exo.hWndToolbar,
TB_GETSTATE, IDM_DRAW_LINE, 0);

        // If the Line button is down, draw with it
        if( iButtonState & TBSTATE_CHECKED )
        {
            hDC = GetDC(hWnd);
            StartX = LOWORD(lParam);
            StartY = HIWORD(lParam);

            EndX = LOWORD(lParam);
            EndY = HIWORD(lParam);

            SetROP2(hDC, R2_XORPEN);

            MoveToEx(hDC, StartX, StartY, NULL);
            LineTo(hDC, EndX, EndY);
            IsDrawing = TRUE;
            ReleaseDC(hWnd, hDC);
        }

        return 0;

    case WM_MOUSEMOVE:

        hDC = GetDC(hWnd);
        if( IsDrawing == TRUE )
        {
            SetROP2(hDC, R2_NOTXORPEN);

            MoveToEx(hDC, StartX, StartY, NULL);
            LineTo(hDC, EndX, EndY);
        }
    }
}

```

## Polylines

---

A polyline is a series of connected lines. The lines are stored in an array of **POINT** values. To draw a polyline, you can use the **Polyline()** function. Its syntax is:

```
BOOL Polyline(HDC hdc, CONST POINT *lppt, int cPoints);
```

The *lppt* argument is an array of points that can be of **POINT** types. The *cPoints* argument specifies the number of members of the array. When executing, the compiler moves the starting point to *lppt[0]*. The first line is drawn from *lppt[0]* to *lppt[1]* as in:

```
//-----  
---  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)  
{  
    HDC hdc;  
    PAINTSTRUCT Ps;  
    POINT Pt[] = { 60, 20, 60, 122 };  
  
    switch(Msg)  
    {  
    case WM_PAINT:  
        hdc = BeginPaint(hWnd, &Ps);  
        MoveToEx(hdc, Pt[0].x, Pt[0].y, NULL);  
        LineTo(hdc, Pt[1].x, Pt[1].y);  
        EndPaint(hWnd, &Ps);  
        break;  
    case WM_DESTROY:  
        PostQuitMessage(WM_QUIT);  
        break;  
    default:  
        return DefWindowProc(hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}  
//-----  
---
```

To draw a polyline, you must have at least two points. If you define more than two points, each line after the first would be drawn from the previous point to the next point until all points have been included. Here is an example:

```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hdc;  
    PAINTSTRUCT Ps;  
    POINT Pt[7];  
    Pt[0].x = 20; Pt[0].y = 50;  
    Pt[1].x = 180; Pt[1].y = 50;  
    Pt[2].x = 180; Pt[2].y = 20;  
    Pt[3].x = 230; Pt[3].y = 70;
```

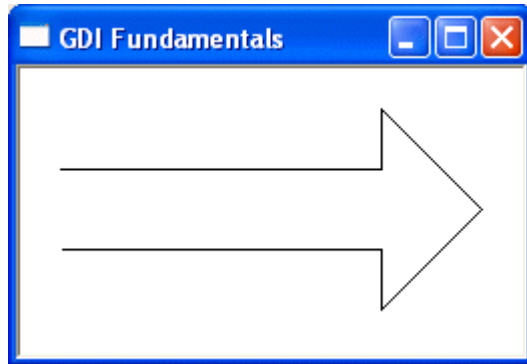


```

Pt[4].x = 180; Pt[4].y = 120;
Pt[5].x = 180; Pt[5].y = 90;
Pt[6].x = 20; Pt[6].y = 90;

switch(Msg)
{
case WM_PAINT:
    hDC = BeginPaint(hWnd, &Ps);
    Polyline(hDC, Pt, 7);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



Besides the **Polyline()** function, you can use the **PolylineTo()** function to draw a polyline. Its syntax is:

```

BOOL PolylineTo(HDC hdc, CONST POINT *lppt, DWORD cCount);

```

The *lppt* argument is the name of an array of **POINT** objects. The *cCount* argument specifies the number of points that would be included in the figure. Here is an example:

```

//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

    POINT Pt[7];
    Pt[0].x = 20; Pt[0].y = 50;
    Pt[1].x = 180; Pt[1].y = 50;
    Pt[2].x = 180; Pt[2].y = 20;
    Pt[3].x = 230; Pt[3].y = 70;

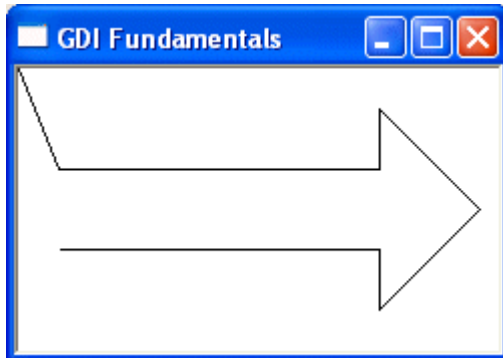
```

```

    Pt[4].x = 180; Pt[4].y = 120;
    Pt[5].x = 180; Pt[5].y = 90;
    Pt[6].x = 20; Pt[6].y = 90;

switch (Msg)
{
case WM_PAINT:
    hDC = BeginPaint (hWnd, &Ps);
    PolylineTo (hDC, Pt, 7);
    EndPaint (hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage (WM_QUIT);
    break;
default:
    return DefWindowProc (hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



While the **Polyline()** function starts the first line at *lppt[0]*, the **PolylineTo()** member function does not control the beginning of the first line. Like the **LineTo()** function, it simply starts drawing, which would mean it starts at the origin (0, 0). For this reason, if you want to control the starting point of the **PolylineTo()** drawing, you can use the **MoveToEx()** function:

```

//-----
-----
LRESULT CALLBACK WndProc (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

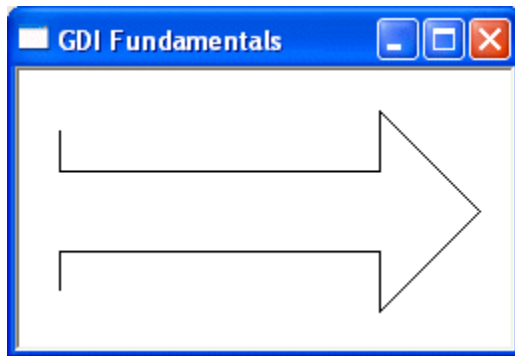
    POINT Pt[7];
    Pt[0].x = 20; Pt[0].y = 50;
    Pt[1].x = 180; Pt[1].y = 50;
    Pt[2].x = 180; Pt[2].y = 20;
    Pt[3].x = 230; Pt[3].y = 70;
    Pt[4].x = 180; Pt[4].y = 120;
    Pt[5].x = 180; Pt[5].y = 90;
    Pt[6].x = 20; Pt[6].y = 90;

```

```

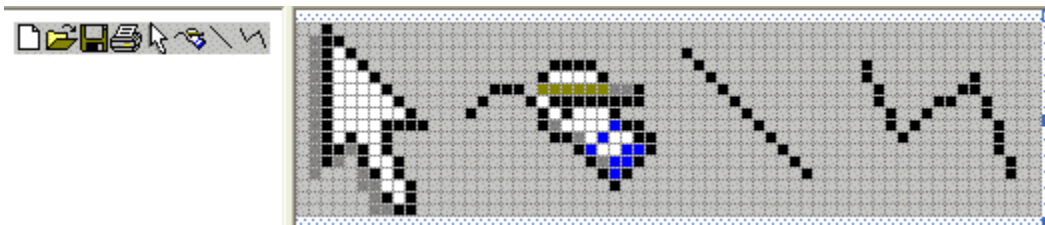
switch (Msg)
{
case WM_PAINT:
    hDC = BeginPaint (hWnd, &Ps);
    MoveToEx (hDC, 20, 30, NULL);
    PolylineTo (hDC, Pt, 7);
    LineTo (hDC, 20, 110);
    EndPaint (hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage (WM_QUIT);
    break;
default:
    return DefWindowProc (hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



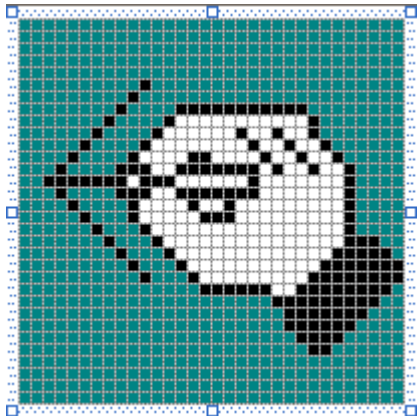
### ▼ Practical Learning: Drawing a Polyline

1. Under the Arrow menu item of the Draw category, add a new menu item Identified as **IDM\_DRAW\_PLOYLINE** with a caption of **&Polyline** with a Prompt of **Draws a series of lines\nPolyline**
2. Add a new bitmap to the right of the IDB\_STANDARD resource as follows:



3. Save the bitmap

4. Design a new cursor as follows:



5. Set its hot spot to the intersection of the left diagonal lines

6. Set its ID to **IDC\_POLYLINE** and save it as polyline.cur

7. To be able to draw a line, change the Exercise.cpp source file as follows:

```
. . . No Change

//-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam)
{
    HDC hDC;

    static BOOL IsDrawing = FALSE;
    static int StartX, StartY;
    static int EndX, EndY;

    switch(Msg)
    {
    case WM_CREATE:
        Exo.CreateStandardToolbar(hWnd, Exo.hInst);
        SendMessage(Exo.hWndToolbar, TB_SETSTATE, IDM_DRAW_ARROW,
TBSTATE_CHECKED | TBSTATE_ENABLED);
        return 0;

    case WM_ACTIVATE:
        Exo.Wnd.SetText("RainDrop - Untitled");
        return 0;

    case WM_LBUTTONDOWN:
        hDC = GetDC(hWnd);

        StartX = LOWORD(lParam);
        StartY = HIWORD(lParam);

        EndX = LOWORD(lParam);
        EndY = HIWORD(lParam);
```

```

        // If the Line button is down, draw with it
        if( SendMessage(Exo.hWndToolbar, TB_GETSTATE, IDM_DRAW_LINE, 0) &
TBSTATE_CHECKED )
        {
            SetROP2(hDC, R2_XORPEN);

            MoveToEx(hDC, StartX, StartY, NULL);
            LineTo(hDC, EndX, EndY);
        }
        else if( SendMessage(Exo.hWndToolbar, TB_GETSTATE,
IDM_DRAW_FREEHAND, 0) & TBSTATE_CHECKED )
        {
            StartX = LOWORD(lParam);
            StartY = HIWORD(lParam);
        }
        else if( SendMessage(Exo.hWndToolbar, TB_GETSTATE,
IDM_DRAW_POLYLINE, 0) & TBSTATE_CHECKED )
        {
            SetROP2(hDC, R2_XORPEN);

            MoveToEx(hDC, StartX, StartY, NULL);
            LineTo(hDC, EndX, EndY);
        }

        IsDrawing = TRUE;

        ReleaseDC(hWnd, hDC);

        return 0;

    case WM_MOUSEMOVE:
        hDC = GetDC(hWnd);

        if( IsDrawing == TRUE )
        {
            SetROP2(hDC, R2_NOTXORPEN);

            // Find out if the Line button is clicked
            if( SendMessage(Exo.hWndToolbar, TB_GETSTATE,
IDM_DRAW_LINE, 0) & TBSTATE_CHECKED )
            {
                MoveToEx(hDC, StartX, StartY, NULL);
                LineTo(hDC, EndX, EndY);

                EndX = LOWORD(lParam);
                EndY = HIWORD(lParam);

                MoveToEx(hDC, StartX, StartY, NULL);
                LineTo(hDC, EndX, EndY);
            }
            // Find out if the Polyline button is clicked
            else if( SendMessage(Exo.hWndToolbar, TB_GETSTATE,
IDM_DRAW_FREEHAND, 0) & TBSTATE_CHECKED )
            {
                MoveToEx(hDC, EndX, EndY, NULL);
                EndX = LOWORD(lParam);
            }
        }
    }
}

```

```

        EndY = HIWORD(lParam);
        LineTo(hDC, EndX, EndY);
    }
    else if( SendMessage(Exo.hWndToolbar, TB_GETSTATE,
IDM_DRAW_POLYLINE, 0) & TBSTATE_CHECKED )
    {
        MoveToEx(hDC, StartX, StartY, NULL);
        LineTo(hDC, EndX, EndY);

        EndX = LOWORD(lParam);
        EndY = HIWORD(lParam);

        MoveToEx(hDC, StartX, StartY, NULL);
        LineTo(hDC, EndX, EndY);
    }
}

ReleaseDC(hWnd, hDC);
break;

case WM_LBUTTONDOWN:

    hDC = GetDC(hWnd);

    EndX = LOWORD(lParam);
    EndY = HIWORD(lParam);

    SetROP2(hDC, R2_XORPEN);

    MoveToEx(hDC, StartX, StartY, NULL);
    LineTo(hDC, EndX, EndY);

    if( SendMessage(Exo.hWndToolbar, TB_GETSTATE, IDM_DRAW_LINE, 0) &
TBSTATE_CHECKED )
    {
        IsDrawing = FALSE;
    }

    if(SendMessage(Exo.hWndToolbar, TB_GETSTATE, IDM_DRAW_FREEHAND, 0)
& TBSTATE_CHECKED )
    {
        IsDrawing = FALSE;
    }

    ReleaseDC(hWnd, hDC);

    break;

case WM_KEYDOWN:
    switch(wParam)
    {
    case VK_ESCAPE:
        // If the user press Esc, may be he/she was drawing a
polyline
        // In that case, stop drawing
        IsDrawing = FALSE;
    }
}

```

```

        break;

    default:
        break;
    }
    break;

case WM_COMMAND:
    switch(LOWORD(wParam))
    {
    case IDM_FILE_NEW:
        break;

    case IDM_FILE_OPEN:
        break;

    case IDM_FILE_SAVE:
        break;

    case IDM_FILE_SAVEAS:
        break;

    case IDM_FILE_EXIT:
        PostQuitMessage(WM_QUIT);
        break;

    case IDM_DRAW_ARROW:
        SendMessage(Exo.hWndToolbar, TB_SETSTATE, IDM_DRAW_ARROW,
TBSTATE_CHECKED | TBSTATE_ENABLED);
        Exo.ChangeCurrentCursor(hWnd, IDC_ARROW);
        break;

    case IDM_DRAW_FREEHAND:
        SendMessage(Exo.hWndToolbar, TB_SETSTATE,
IDM_DRAW_FREEHAND, TBSTATE_CHECKED | TBSTATE_ENABLED);
        Exo.ChangeCurrentCursor(hWnd,
MAKEINTRESOURCE(IDC_FREEHAND));
        break;

    case IDM_DRAW_LINE:
        SendMessage(Exo.hWndToolbar, TB_SETSTATE, IDM_DRAW_LINE,
TBSTATE_CHECKED | TBSTATE_ENABLED);
        Exo.ChangeCurrentCursor(hWnd, MAKEINTRESOURCE(IDC_LINE));
        break;

    case IDM_DRAW_POLYLINE:
        SendMessage(Exo.hWndToolbar, TB_SETSTATE,
IDM_DRAW_POLYLINE, TBSTATE_CHECKED | TBSTATE_ENABLED);
        Exo.ChangeCurrentCursor(hWnd,
MAKEINTRESOURCE(IDC_POLYLINE));
        break;
    }
    return 0;

case WM_DESTROY:
    PostQuitMessage(WM_QUIT);

```

```

        return 0;

default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}

return TRUE;
}
//-----
HWND Exercise::CreateStandardToolbar(HWND hParent, HINSTANCE hInst)
{
    const int NUMBUTTONS = 9;
    TBBUTTON tbrButtons[NUMBUTTONS];

    tbrButtons[0].iBitmap    = 0;
    tbrButtons[0].idCommand  = IDM_FILE_NEW;
    tbrButtons[0].fsState    = TBSTATE_ENABLED;
    tbrButtons[0].fsStyle    = TBSTYLE_BUTTON;
    tbrButtons[0].dwData     = 0L;
    tbrButtons[0].iBitmap    = 0;
    tbrButtons[0].iString    = 0;

    tbrButtons[1].iBitmap    = 1;
    tbrButtons[1].idCommand  = IDM_FILE_OPEN;
    tbrButtons[1].fsState    = TBSTATE_ENABLED;
    tbrButtons[1].fsStyle    = TBSTYLE_BUTTON;
    tbrButtons[1].dwData     = 0L;
    tbrButtons[1].iString    = 0;

    tbrButtons[2].iBitmap    = 2;
    tbrButtons[2].idCommand  = IDM_FILE_SAVE;
    tbrButtons[2].fsState    = TBSTATE_ENABLED;
    tbrButtons[2].fsStyle    = TBSTYLE_BUTTON;
    tbrButtons[2].dwData     = 0L;
    tbrButtons[2].iString    = 0;

    tbrButtons[3].iBitmap    = 3;
    tbrButtons[3].idCommand  = IDM_FILE_PRINT;
    tbrButtons[3].fsState    = TBSTATE_ENABLED;
    tbrButtons[3].fsStyle    = TBSTYLE_BUTTON;
    tbrButtons[3].dwData     = 0L;
    tbrButtons[3].iString    = 0;

    tbrButtons[4].iBitmap    = 0;
    tbrButtons[4].idCommand  = 0;
    tbrButtons[4].fsState    = TBSTATE_ENABLED;
    tbrButtons[4].fsStyle    = TBSTYLE_SEP;
    tbrButtons[4].dwData     = 0L;
    tbrButtons[4].iString    = 0;

    tbrButtons[5].iBitmap    = 4;
    tbrButtons[5].idCommand  = IDM_DRAW_ARROW;
    tbrButtons[5].fsState    = TBSTATE_ENABLED;
    tbrButtons[5].fsStyle    = TBSTYLE_BUTTON | TBSTYLE_GROUP | TBSTYLE_CHECK;
    tbrButtons[5].dwData     = 0L;
    tbrButtons[5].iString    = 0;

```



```

tbrButtons[6].iBitmap      = 5;
tbrButtons[6].idCommand   = IDM_DRAW_FREEHAND;
tbrButtons[6].fsState     = TBSTATE_ENABLED;
tbrButtons[6].fsStyle     = TBSTYLE_BUTTON | TBSTYLE_GROUP | TBSTYLE_CHECK;
tbrButtons[6].dwData      = 0L;
tbrButtons[6].iString     = 0;

tbrButtons[7].iBitmap      = 6;
tbrButtons[7].idCommand   = IDM_DRAW_LINE;
tbrButtons[7].fsState     = TBSTATE_ENABLED;
tbrButtons[7].fsStyle     = TBSTYLE_BUTTON | TBSTYLE_GROUP | TBSTYLE_CHECK;
tbrButtons[7].dwData      = 0L;
tbrButtons[7].iString     = 0;

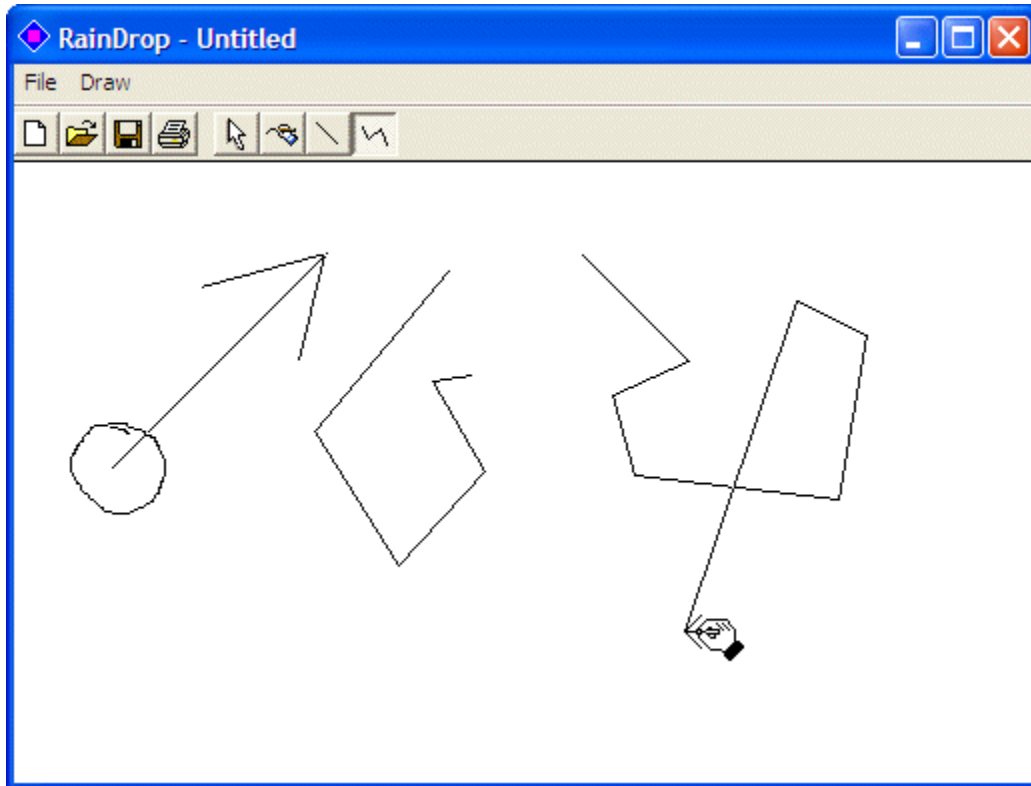
tbrButtons[8].iBitmap      = 7;
tbrButtons[8].idCommand   = IDM_DRAW_POLYLINE;
tbrButtons[8].fsState     = TBSTATE_ENABLED;
tbrButtons[8].fsStyle     = TBSTYLE_BUTTON | TBSTYLE_GROUP | TBSTYLE_CHECK;
tbrButtons[8].dwData      = 0L;
tbrButtons[8].iString     = 0;

hWndToolbar = CreateToolbarEx(hParent,
                              WS_VISIBLE | WS_CHILD | WS_BORDER,
                              IDB_STANDARD,
                              NUMBUTTONS,
                              hInst,
                              IDB_STANDARD,
                              tbrButtons,
                              NUMBUTTONS,
                              16, 16, 16, 16,
                              sizeof(TBBUTTON));

if( hWndToolbar )
    return hWndToolbar;
return NULL;
}
//-----

```

- Execute the application to test it



- Return to your programming environment

## Multiple Polylines

---

The above polylines were used each as a single entity. That is, a polyline is a combination of lines. If you want to draw various polylines in one step, you can use the **PolyPolyline()** function. By definition, the **PolyPolyline()** function is used to draw a series of polylines. Its syntax is:

```
BOOL PolyPolyline(HDC hdc, CONST POINT *lppt, CONST DWORD *lpdwPolyPoints, DWORD cCount);
```

Like the above **Polyline()** function, the *lppt* argument is an array of **POINT** values. The **PolyPolyline()** function needs to know how many polylines you would be drawing.

Each polyline will use the points of the *lpdwPolyPoints* value but when creating the array of points, the values must be incremental. This means that **PolyPolyline()** will not access their values at random. It will retrieve the first point, followed by the second, followed by the third, etc. Therefore, your first responsibility is to decide where one polyline starts and where it ends. The good news (of course depending on how you see it) is that a polyline does not start where the previous line ended. Each polyline has its own beginning and its own ending point.

Unlike **Polyline()**, here, the *cCount* argument is actually the number of shapes you want to draw and not the number of points (remember that each polyline "knows" or controls its beginning and end).

The *lpdwPolyPoints* argument is an array of positive integers. Each member of this array

specifies the number of vertices (lines) that its corresponding polyline will have. For example, imagine you want to draw M, followed by L, followed by Z. The letter M has 4 lines but you need 5 points to draw it. The letter L has 2 lines and you need 3 points to draw it. The letter Z has 3 lines so 4 points are necessary to draw it. You can store this combination of lines in an array defined as { 5, 3, 4 }.

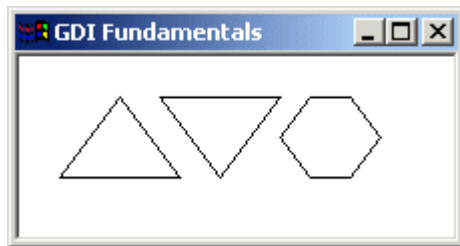
Here is an example:

```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hDC;  
    PAINTSTRUCT Ps;  
  
    POINT Pt[15];  
    DWORD lpPts[] = { 4, 4, 7 };  
  
    // Left Triangle  
    Pt[0].x = 50;  
    Pt[0].y = 20;  
    Pt[1].x = 20;  
    Pt[1].y = 60;  
    Pt[2].x = 80;  
    Pt[2].y = 60;  
    Pt[3].x = 50;  
    Pt[3].y = 20;  
  
    // Second Triangle  
    Pt[4].x = 70;  
    Pt[4].y = 20;  
    Pt[5].x = 100;  
    Pt[5].y = 60;  
    Pt[6].x = 130;  
    Pt[6].y = 20;  
    Pt[7].x = 70;  
    Pt[7].y = 20;  
  
    // Hexagon  
    Pt[8].x = 145;  
    Pt[8].y = 20;  
    Pt[9].x = 130;  
    Pt[9].y = 40;  
    Pt[10].x = 145;  
    Pt[10].y = 60;  
    Pt[11].x = 165;  
    Pt[11].y = 60;  
    Pt[12].x = 180;  
    Pt[12].y = 40;  
    Pt[13].x = 165;  
    Pt[13].y = 20;  
    Pt[14].x = 145;  
    Pt[14].y = 20;
```

```

switch(Msg)
{
case WM_PAINT:
    hDC = BeginPaint(hWnd, &Ps);
    PolyPolyline(hDC, Pt, lpPts, 3);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



## GDI Built-In Shapes

---

### Closed Shapes

---



#### Polygons

---

The polylines we have used so far were drawn by defining the starting point of the first line and the end point of the last line and there was no relationship or connection between these two extreme points. A polygon is a closed polyline. In other words, it is a polyline defined so that the end point of the last line is connected to the start point of the first line.

To draw a polygon, you can use the **Polygon()** function. Its syntax is:

```

BOOL Polygon(HDC hdc, CONST POINT *lpPoints, int nCount);

```

This function uses the same types of arguments as the **Polyline()** function. The only difference is on the drawing of the line combination. Here is an example:

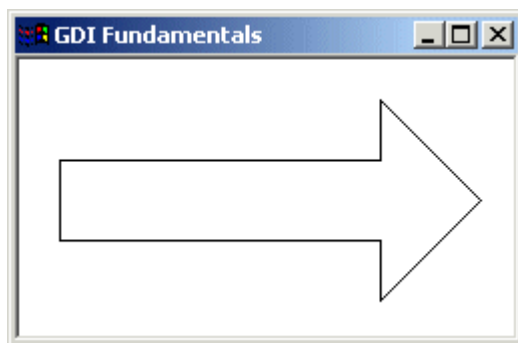
```

//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

    POINT Pt[7];
    Pt[0].x = 20;
    Pt[0].y = 50;
    Pt[1].x = 180;
    Pt[1].y = 50;
    Pt[2].x = 180;
    Pt[2].y = 20;
    Pt[3].x = 230;
    Pt[3].y = 70;
    Pt[4].x = 180;
    Pt[4].y = 120;
    Pt[5].x = 180;
    Pt[5].y = 90;
    Pt[6].x = 20;
    Pt[6].y = 90;

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        Polygon(hDC, Pt, 7);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



## Multiple Polygons

---

If you want to draw multiple polygons, you can use the PolyPolygon() function whose syntax is:

```
BOOL PolyPolygon(HDC hdc, CONST POINT *lpPoints,  
                CONST INT *lpPolyCounts, int nCount);
```

Like the **Polygon()** function, the *lpPoints* argument is an array of **POINT** values. The **PolyPolygon()** function needs to know the number of polygons you would be drawing. Each polygon uses the points of the *lpPoints* values but when creating the array of points, the values must be incremental. This means that **PolyPolygon()** will not randomly access the values of *lpPoints*. Each polygon has its own set of points.

Unlike **Polygon()**, the *nCount* argument of **PolyPolygon()** is the number of polygons you want to draw and not the number of points.

The *lpPolyCounts* argument is an array of integers. Each member of this array specifies the number of vertices (lines) that its polygon will have..

Here is an example:

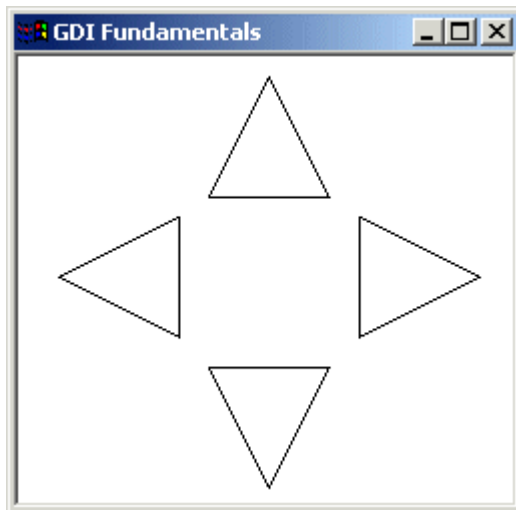
```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hdc;  
    PAINTSTRUCT Ps;  
  
    POINT Pt[12];  
    int lpPts[] = { 3, 3, 3, 3 };  
  
    // Top Triangle  
    Pt[0].x = 125;  
    Pt[0].y = 10;  
    Pt[1].x = 95;  
    Pt[1].y = 70;  
    Pt[2].x = 155;  
    Pt[2].y = 70;  
  
    // Left Triangle  
    Pt[3].x = 80;  
    Pt[3].y = 80;  
    Pt[4].x = 20;  
    Pt[4].y = 110;  
    Pt[5].x = 80;  
    Pt[5].y = 140;  
  
    // Bottom Triangle  
    Pt[6].x = 95;  
    Pt[6].y = 155;  
    Pt[7].x = 125;  
    Pt[7].y = 215;  
    Pt[8].x = 155;  
    Pt[8].y = 155;
```

```

    // Right Triangle
    Pt[9].x = 170;
    Pt[9].y = 80;
    Pt[10].x = 170;
    Pt[10].y = 140;
    Pt[11].x = 230;
    Pt[11].y = 110;

switch (Msg)
{
case WM_PAINT:
    hDC = BeginPaint(hWnd, &Ps);
    PolyPolygon(hDC, Pt, lpPts, 4);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

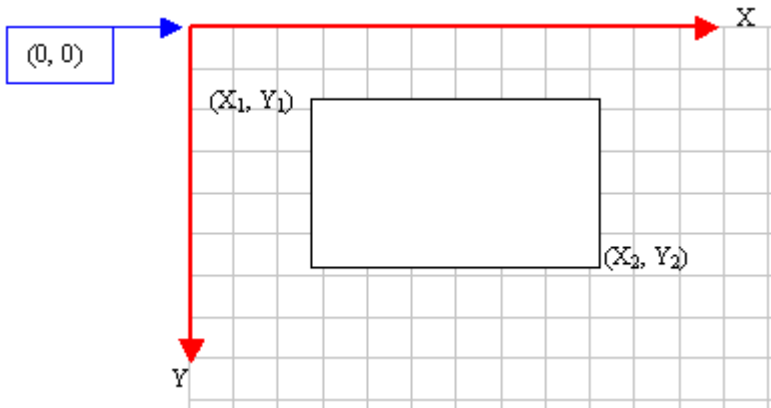
```



## Rectangles and Squares

---

A rectangle is a geometric figure made of four sides that compose four right angles. Like the line, to draw a rectangle, you must define where it starts and where it ends. This can be illustrated as follows:



The drawing of a rectangle typically starts from a point defined as  $(X_1, Y_1)$  and ends at another point  $(X_2, Y_2)$ .

To draw a rectangle, you can use the `Rectangle()` function. Its syntax is:

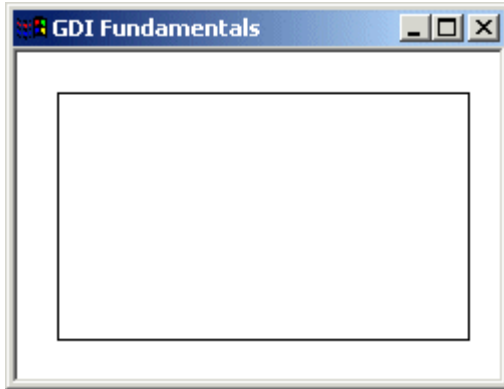
```
BOOL Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int
nBottomRect);
```

As seen on the figure and the formula, a rectangle spans from coordinates  $(nLeftRect, nTopRect)$  to  $(nRightRect, nBottomRect)$ . Here is an example:

```
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

    switch(Msg)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &Ps);
        Rectangle(hdc, 20, 20, 226, 144);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----
```





When drawing a rectangle, if the value of *nRightRect* is less than that of *nLeftRect*, then the *nRightRect* coordinate would mark the left beginning of the figure. This scenario would also apply if the *nBottomRect* coordinate were lower than *nTopRect*.

A square is a rectangle whose sides are all equal. Therefore, to draw a square, when specifying the arguments of the **Rectangle()** function, make sure that  $|x_1 - x_2| = |y_1 - y_2|$ .

## A Rectangle With Edges

---

The GDI library provides another function you can use to draw a rectangle. This time you can control how the edges of the rectangle would be drawn. The function used is called **DrawEdge** and its syntax is:

```
BOOL DrawEdge(HDC hdc, LPRECT qrc, UINT edge, UINT grfFlags);
```

The *qrc* argument is passed as a pointer to a **RECT** value, which is the rectangle that would be drawn.

The *edge* value specifies how the interior and the exterior of the edges of the rectangle would be drawn. It can be a combination of the following constants:

Value	Description
BDR_RAISEDINNER	The interior edge will be raised
BDR_SUNKENINNER	The interior edge will be sunken
BDR_RAISEDOUTER	The exterior edge will be raised
BDR_SUNKENOUTER	The exterior edge will be sunken

These values can be combined using the bitwise OR operator. On the other hand, you can use the following constants instead:

Value	Used For
EDGE_DUMP	BDR_RAISEDOUTER   BDR_SUNKENINNER
EDGE_ETCHED	BDR_SUNKENOUTER   BDR_RAISEDINNER
EDGE_RAISED	BDR_RAISEDOUTER   BDR_RAISEDINNER
EDGE_SUNKEN	BDR_SUNKENOUTER   BDR_SUNKENINNER

The *grfFlags* value specifies what edge(s) would be drawn. It can have one of the following values:

Value	Description
BF_RECT	The entire rectangle will be drawn
BF_TOP	Only the top side will be drawn
BF_LEFT	Only the left side will be drawn
BF_BOTTOM	Only the bottom side will be drawn
BF_RIGHT	Only the right side will be drawn
BF_TOPLEFT	Both the top and the left sides will be drawn
BF_BOTTOMLEFT	Both the bottom and the left sides will be drawn
BF_TOPRIGHT	Both the top and the right sides will be drawn
BF_BOTTOMRIGHT	Both the bottom and the right sides will be drawn
BF_DIAGONAL_ENDBOTTOMLEFT	A diagonal line will be drawn from the top-right to the bottom-left corners
BF_DIAGONAL_ENDBOTTOMRIGHT	A diagonal line will be drawn from the top-left to the bottom-right corners
BF_DIAGONAL_ENDTOPLEFT	A diagonal line will be drawn from the bottom-right to the top-left corners
BF_DIAGONAL_ENDTOPRIGHT	A diagonal line will be drawn from the bottom-left to the top-right corners

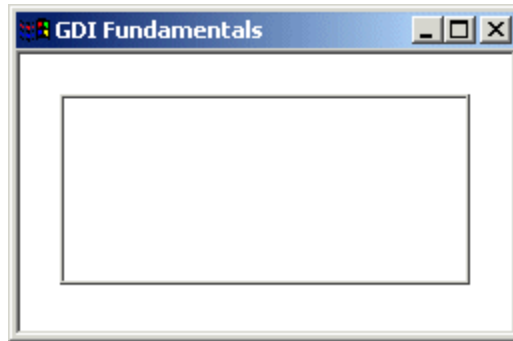
Here is an example:

```
//-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

    RECT Recto = {20, 20, 225, 115};

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        DrawEdge(hDC, &Recto, BDR_RAISEDOUTER | BDR_SUNKENINNER, BF_RECT);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
    }
}
```

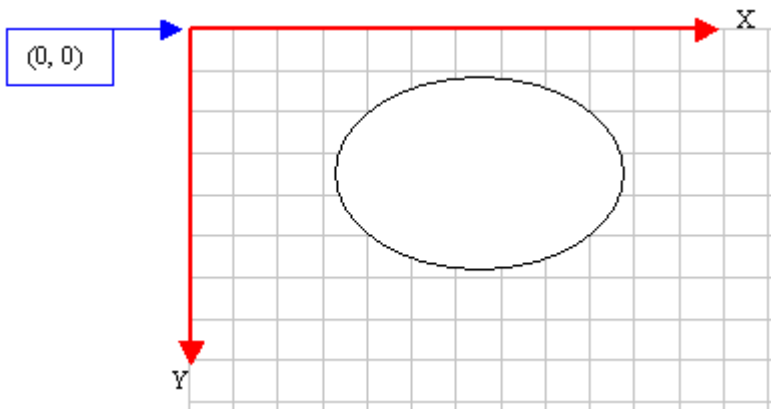
```
        break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
```



## Ellipses and Circles

---

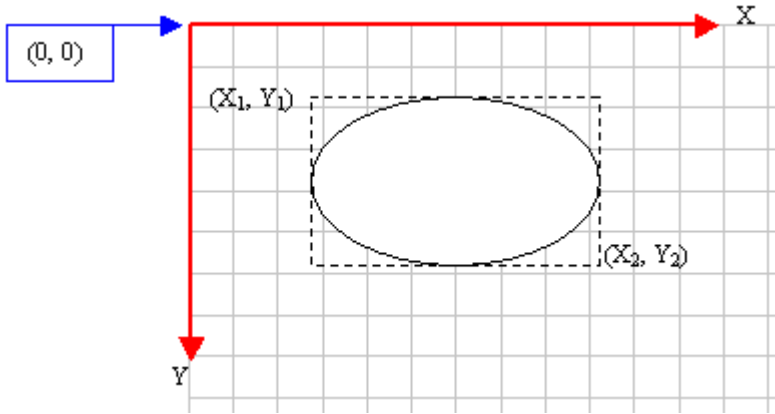
An ellipse is a closed continuous line whose points are positioned so that two points exactly opposite each other have the exact same distant from a central point. It can be illustrated as follows:



Because an ellipse can fit in a rectangle, in GDI programming, an ellipse is defined with regards to a rectangle it would fit in. Therefore, to draw an ellipse, you specify its rectangular corners. The syntax used to do this is:

```
BOOL Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);
```

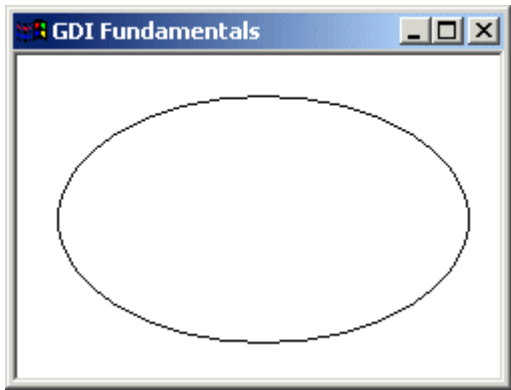
The arguments of this function play the same roll as those of the **Rectangle()** function:



Here is an example:

```
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

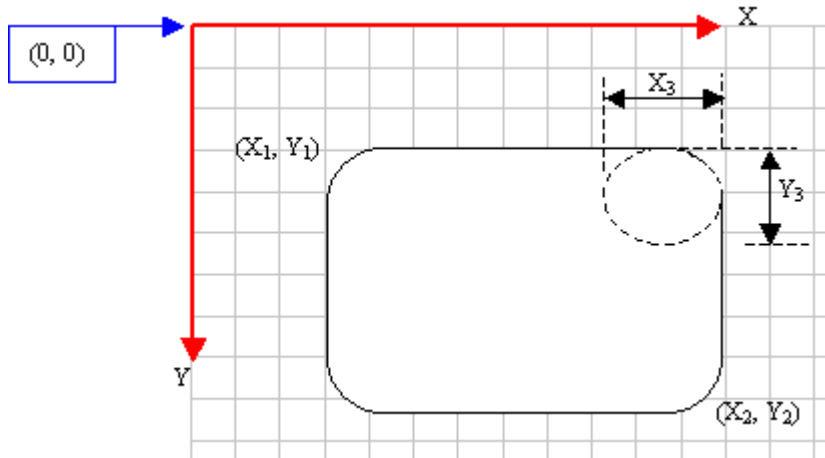
    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        Ellipse(hDC, 20, 20, 226, 144);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----
```



## Round Rectangles and Round Squares

---

A rectangle qualifies as round if its corners do not form straight angles but rounded corners. It can be illustrated as follows:



To draw such a rectangle, you can use the **RoundRect()** function. Its syntax is:

```
BOOL RoundRect (HDC hdc,
                int nLeftRect, int nTopRect, int nRightRect, int nBottomRect,
                int nWidth, int nHeight );
```

When this member function executes, the rectangle is drawn from the (*nLeftRect*, *nTopRect*) to the (*nRightRect*, *nBottomRect*) points. The corners are rounded by an ellipse whose width would be *nWidth* and the ellipse's height would be *nHeight*.

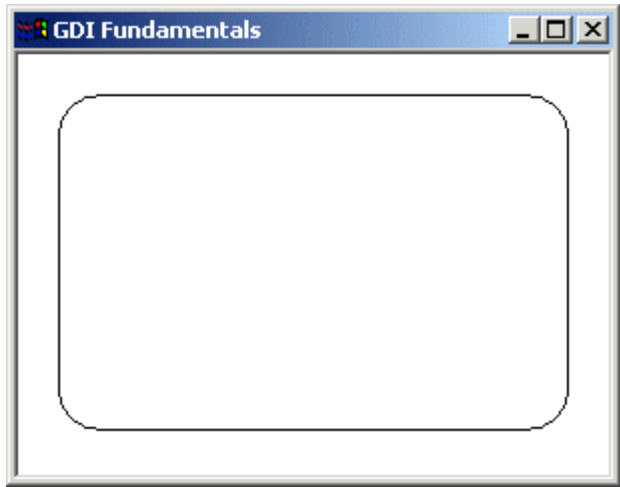
Here is an example:

```
//-----  
-----  
LRESULT CALLBACK WndProc (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hDC;  
    PAINTSTRUCT Ps;  
  
    switch (Msg)  
    {  
    case WM_PAINT:  
        hDC = BeginPaint (hWnd, &Ps);  
        RoundRect (hDC, 20, 20, 275, 188, 42, 38);  
        EndPaint (hWnd, &Ps);  
        break;  
    case WM_DESTROY:  
        PostQuitMessage (WM_QUIT);  
        break;  
    default:  
        return DefWindowProc (hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}
```

```

}
//-----
-----

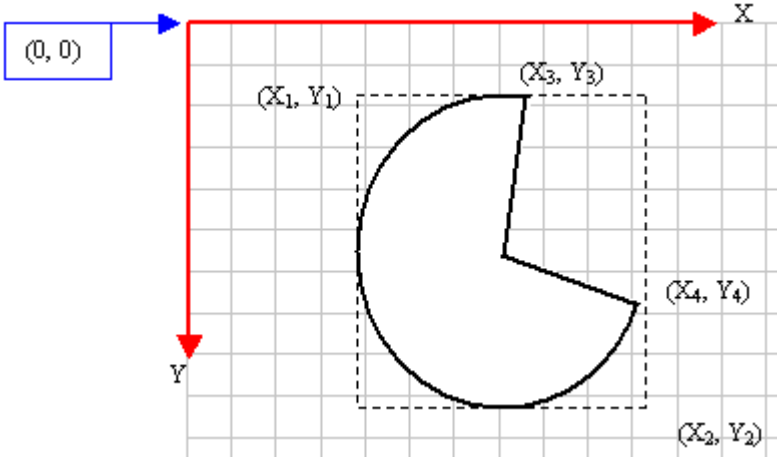
```



A round square is a square whose corners are rounded.

**Pies**

A pie is a fraction of an ellipse delimited by two lines that span from the center of the ellipse to one side each. It can be illustrated as follows:



To draw a pie, you can use the **Pie()** function whose syntax is:

```

BOOL Pie(HDC hdc,
         int nLeftRect, int nTopRect, int nRightRect, int nBottomRect,
         int nXRadial1, int nYRadial1, int nXRadial2, int nYRadial2);

```

The (nLeftRect, nTopRect) point determines the upper-left corner of the rectangle in which the ellipse that represents the pie fits. The (nRightRect, nBottomRect) point is the bottom-right corner of the rectangle.

The (nXRadial1, nYRadial1) point species the end point of the pie.

To complete the pie, a line is drawn from (nXRadial1, nYRadial1) to the center and from the

center to the (*nXRadial2*, *nYRadial2*) points.

Here is an example:

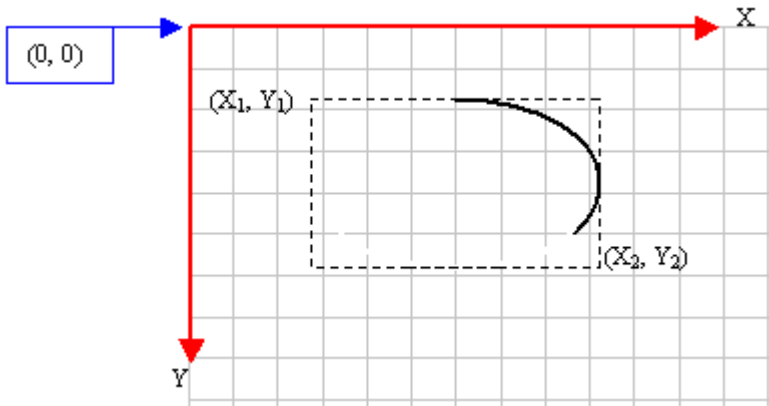
```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hDC;  
    PAINTSTRUCT Ps;  
  
    switch(Msg)  
    {  
    case WM_PAINT:  
        hDC = BeginPaint(hWnd, &Ps);  
        Pie(hDC, 40, 20, 226, 144, 155, 32, 202, 115);  
        EndPaint(hWnd, &Ps);  
        break;  
    case WM_DESTROY:  
        PostQuitMessage(WM_QUIT);  
        break;  
    default:  
        return DefWindowProc(hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}  
//-----  
-----
```



## Arcs

---

An arc is a portion or segment of an ellipse, meaning an arc is a non-complete ellipse. Because an arc must conform to the shape of an ellipse, it is defined as it fits in a rectangle and can be illustrated as follows:



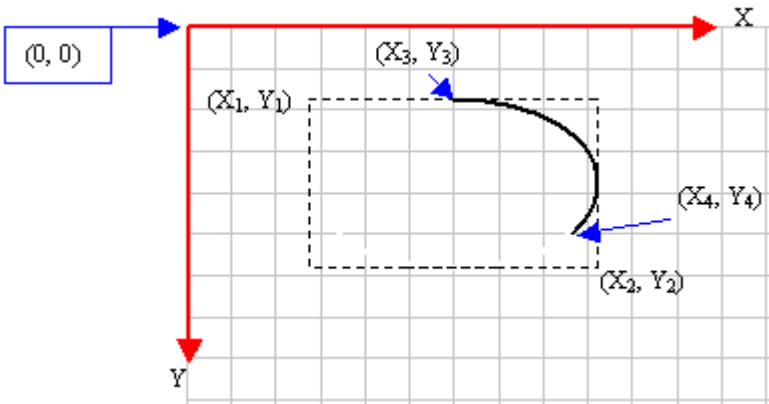
To draw an arc, you can use the **Arc()** function whose syntax is:

```

BOOL Arc(HDC hdc,
         int nLeftRect, int nTopRect, int nRightRect, int nBottomRect,
         int nXStartArc, int nYStartArc, int nXEndArc, int nYEndArc);

```

Besides the left (*nLeftRect*, *nTopRect*) and the right (*nRightRect*, *nBottomRect*) borders of the rectangle in which the arc would fit, an arc must specify where it starts and where it ends. These additional points are set as the (*nXStartArc*, *nYStartArc*) and (*nXEndArc*, *nYEndArc*) points of the figure. Based on this, the above arc can be illustrated as follows:



Here is an example:

```

//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

    switch(Msg)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &Ps);
        Arc(hdc, 20, 20, 226, 144, 202, 115, 105, 32);

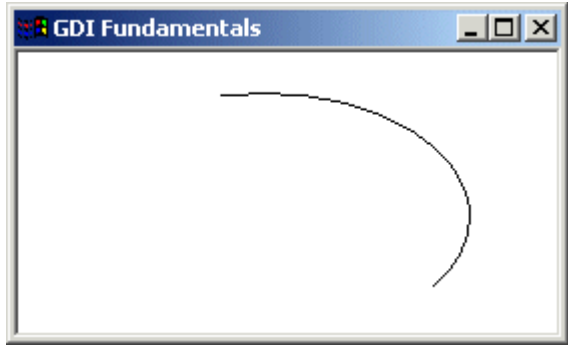
```



```

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



Besides the **Arc()** function, the GDI library provides the **ArcTo()** member function used to draw an arc. Its syntax is:

```

BOOL ArcTo(HDC hdc,
           int nLeftRect, int nTopRect, int nRightRect, int nBottomRect,
           int nXRadial1, int nYRadial1, int nXRadial2, int nYRadial2);

```

This function uses the same arguments as **Arc()**. The difference is that while **Arc()** starts drawing at  $(nXRadial1, nYRadial1)$ , **ArcTo()** does not inherently control the drawing starting point. It refers to the current point, exactly like the **LineTo()** (and the **PolylineTo()**) function. Therefore, if you want to specify where the drawing should start, can call **MoveToEx()** before **ArcTo()**. Here is an example:

```

//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

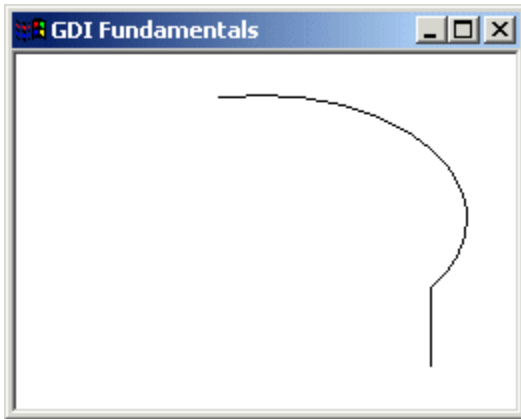
    switch(Msg)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &Ps);
        MoveToEx(hdc, 207, 155, NULL);
        ArcTo(hdc, 20, 20, 226, 144, 202, 115, 105, 32);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    }
}

```

```

default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



## The Arc's Direction

---

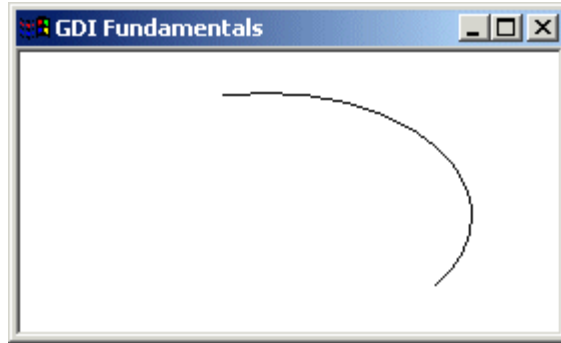
Here is an arc we drew earlier with a call to **Arc()**:

```

//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        Arc(hDC, 20, 20, 226, 144, 202, 115, 105, 32);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



You may wonder why the arc is drawn to the right side of a vertical line that would cross the center of the ellipse instead of the left. This is because the drawing of an arc is performed from right to left or from bottom to top, in the opposite direction of the clock. This is known as the counterclockwise direction. To control this orientation, the GDI library is equipped with the **SetArcDirection()** function. Its syntax is:

```
int SetArcDirection(HDC hdc, int ArcDirection);
```

This function specifies the direction the **Arc()** function should follow from the starting to the end points. The argument passed as *ArcDirection* controls this orientation. It can have the following values:

Value	Orientation
AD_CLOCKWISE	The figure is drawn clockwise
AD_COUNTERCLOCKWISE	The figure is drawn counterclockwise

The default value of the direction is **AD\_COUNTERCLOCKWISE**. Therefore, this would be used if you do not specify a direction. Here is an example that uses the same values as above with a different orientation:

```
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

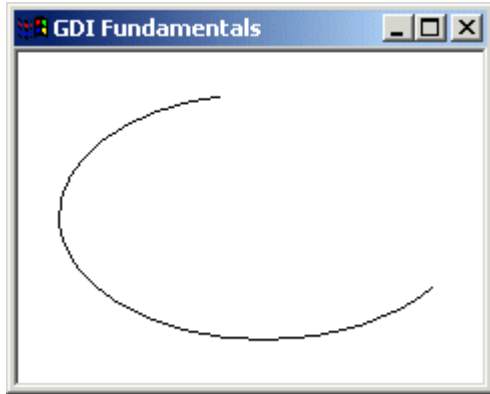
    switch(Msg)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &Ps);
        SetArcDirection(hdc, AD_CLOCKWISE);
        Arc(hdc, 20, 20, 226, 144, 202, 115, 105, 32);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:

```

```

        return DefWindowProc (hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



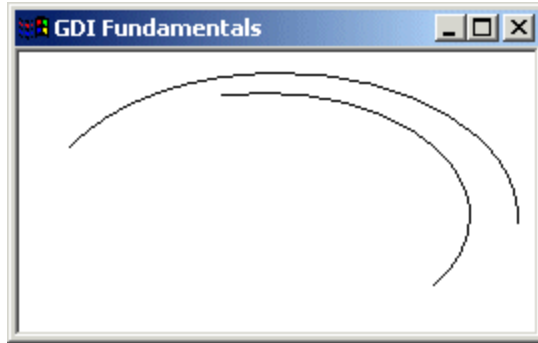
After calling **SetArcDirection()** and changing the previous direction, all drawings would use the new direction to draw arcs using **Arc()** or **ArcTo()** and other figures (such as chords, ellipses, pies, and rectangles). Here is an example:

```

//-----
-----
LRESULT CALLBACK WndProc (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

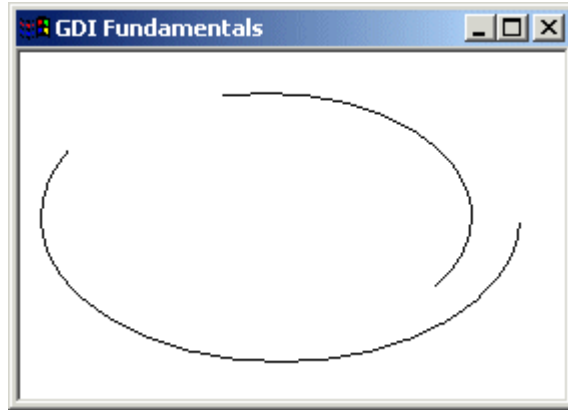
    switch (Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint (hWnd, &Ps);
        SetArcDirection (hDC, AD_COUNTERCLOCKWISE);
        Arc (hDC, 20, 20, 226, 144, 202, 115, 105, 32);
        Arc (hDC, 10, 10, 250, 155, 240, 85, 24, 48);
        EndPaint (hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage (WM_QUIT);
        break;
    default:
        return DefWindowProc (hWnd, Msg, wParam, lParam);
    }
    return 0;
}
//-----
-----

```



If you want to change the direction, you must call **SetArcDirection()** with the desired value. Here is an example;

```
//-----  
-----  
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HDC hDC;  
    PAINTSTRUCT Ps;  
  
    switch(Msg)  
    {  
    case WM_PAINT:  
        hDC = BeginPaint(hWnd, &Ps);  
        SetArcDirection(hDC, AD_COUNTERCLOCKWISE);  
        Arc(hDC, 20, 20, 226, 144, 202, 115, 105, 32);  
        SetArcDirection(hDC, AD_CLOCKWISE);  
        Arc(hDC, 10, 10, 250, 155, 240, 85, 24, 48);  
        EndPaint(hWnd, &Ps);  
        break;  
    case WM_DESTROY:  
        PostQuitMessage(WM_QUIT);  
        break;  
    default:  
        return DefWindowProc(hWnd, Msg, wParam, lParam);  
    }  
    return 0;  
}  
//-----  
-----
```



At any time, you can find out the current direction used. This is done by calling the **GetArcDirection()** function. Its syntax is:

```
int GetArcDirection(HDC hdc);
```

This function returns the current arc direction as **AD\_CLOCKWISE** or **AD\_COUNTERCLOCKWISE**.

## Angular Arcs

---

You can (also) draw an arc using the **AngleArc()** function. Its syntax is:

```
BOOL AngleArc(HDC hdc, int X, int Y, DWORD dwRadius,
              FLOAT eStartAngle, FLOAT eSweepAngle);
```

This function draws a line and an arc connected. The arc is based on a circle and not an ellipse. This implies that the arc fits inside a square and not a rectangle. The circle that would be the base of the arc is defined by its center located at  $C(X, Y)$  with a radius of *dwRadius*. The arc starts at an angle of *eStartAngle*. The angle is based on the x axis and must be positive. That is, it must range from  $0^\circ$  to  $360^\circ$ . If you want to specify an angle that is below the x axis, such as  $-15^\circ$ , use  $360^\circ - 15^\circ = 345^\circ$ . The last argument, *eSweepAngle*, is the angular area covered by the arc.

The **AngleArc()** function does not control where it starts drawing. This means that it starts at the origin, unless a previous call to **MoveToEx()** specified the beginning of the drawing.

Here is an example:

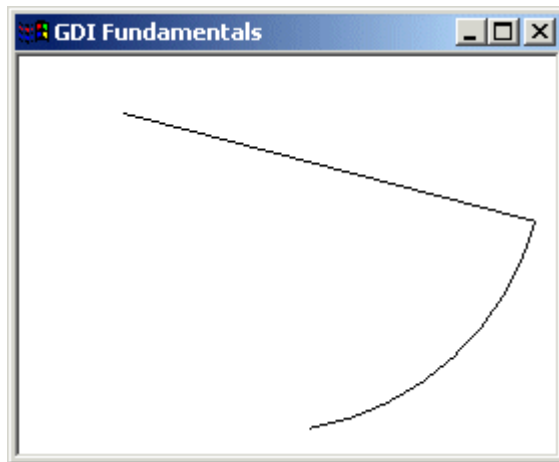
```
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

    switch(Msg)
    {
    case WM_PAINT:
```

```

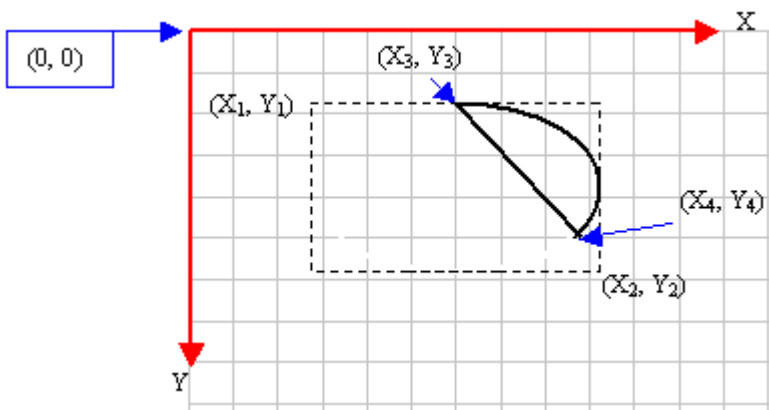
    HDC = BeginPaint(hWnd, &Ps);
    MoveToEx(hDC, 52, 28, NULL);
    AngleArc(hDC, 120, 45, 142, 345, -65);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----

```



## Chords

The arcs we have drawn so far are considered open figures because they are made of a line that has a beginning and an end (unlike a circle or a rectangle that do not). A chord is an arc whose two ends are connected by a straight line. In other words, a chord is an ellipse that is divided by a straight line from one side to another:



To draw a chord, you can use the Chord() function. Its syntax is as follows:

```

BOOL Chord(HDC hdc,

```

```
int nLeftRect, int nTopRect, int nRightRect, int nBottomRect,
int nXRadial1, int nYRadial1, int nXRadial2, int nYRadial2);
```

The *nLeftRect*, *nTopRect*, *nRightRect*, and *nBottomRect* are the coordinates of the rectangle in which the chord of the circle would fit.

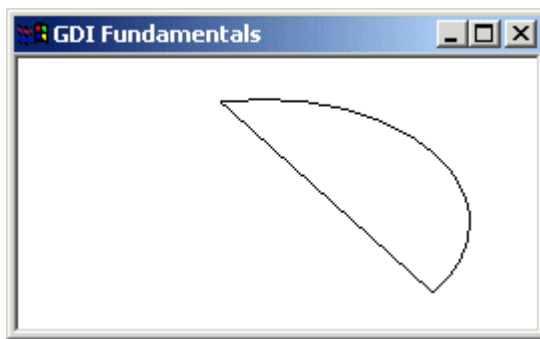
These *nXRadial1* and *nYRadial1* coordinates specify where the arc that holds the chord starts.

To complete the chord, a line is drawn from (*nXRadial1*, *nYRadial1*) to (*nXRadial2*, *nYRadial2* ).

Here is an example:

```
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hDC;
    PAINTSTRUCT Ps;

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        Chord(hDC, 20, 20, 226, 144, 202, 115, 105, 32);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```

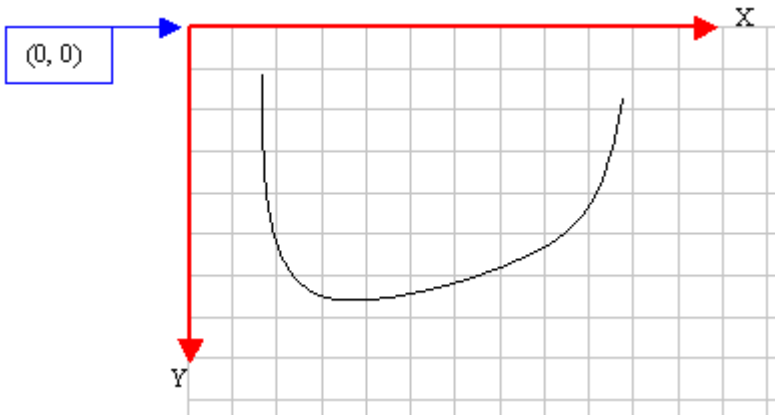


## Bézier Curves

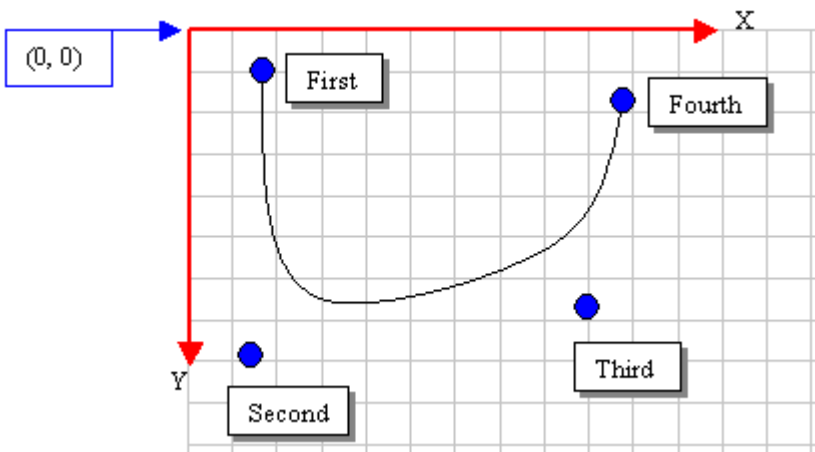
---

A bézier line is an arc that is strictly based on a set number of points instead of on an ellipse. A bézier curve uses at least four points to draw on. A bézier line with four points can be illustrated as follows:





To draw this line (with four points), the compiler would draw a curve from the first to the fourth points. Then it would bend the curve by bringing each middle (half-center) side close to the second and the third points respectively, of course without touching those second and third points. For example, the above bézier curve could have been drawn using the following four points:



**PolyBezier()**: To draw a bézier curve, the GDI library provides the **PolyBezier()** function. Its syntax is:

```
BOOL PolyBezier(HDC hdc, CONST POINT *lppt, DWORD cPoints);
```

The *lppt* argument is an array of **POINT** values. The *cPoints* argument specifies the number of points that will be used to draw the line. Here is an example:

```
//-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

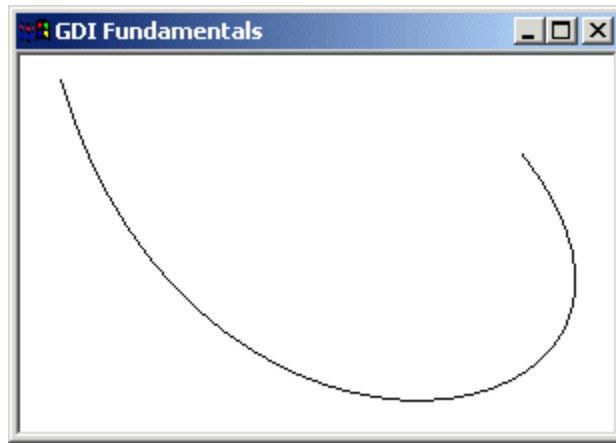
    POINT Pt[4] = { { 20, 12 }, { 88, 246 }, { 364, 192 }, { 250, 48 } };

    switch(Msg)
    {
```

```

case WM_PAINT:
    hDC = BeginPaint(hWnd, &Ps);
    PolyBezier(hDC, Pt, 4);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----

```



**PolyBezierTo():** The **PolyBezier()** function requires at least four points to draw its curve. This is because it needs to know where to start drawing. Another way you can control where the curve would start is by using the **PolyBezierTo()** function. Its syntax is:

```

BOOL PolyBezierTo(HDC hdc, CONST POINT *lppt, DWORD cCount);

```

The **PolyBezierTo()** function draws a bézier curve. Its first argument is a pointer to an array of **POINT** values. This member function requires at least three points. It starts drawing from the current line to the third point. If you do not specify the current line, it would consider the origin (0, 0). The first and the second lines are used to control the curve. The *cCount* argument is the number of points that would be considered. Here is an example:

```

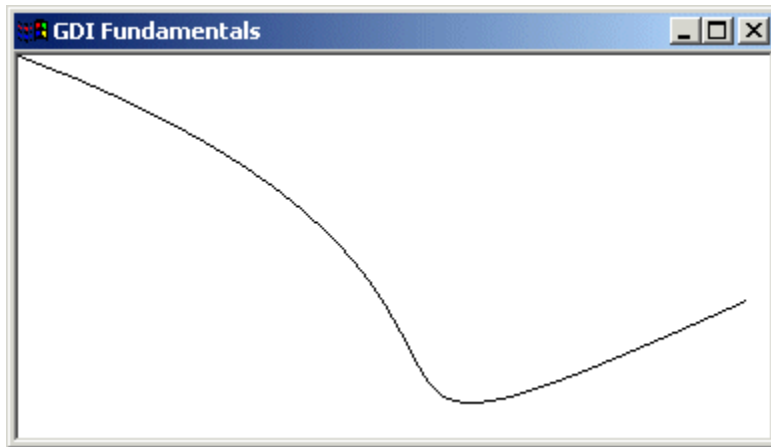
//-----
-----
LRESULT CALLBACK WndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT Ps;

    POINT Pt[4] = { { 320, 120 }, { 88, 246 }, { 364, 122 } };

    switch(Msg)
    {
    case WM_PAINT:

```

```
    hDC = BeginPaint(hWnd, &Ps);
    PolyBezierTo(hDC, Pt, 3);
    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}
//-----
-----
```



## GDI Colors

---

The color is one the most fundamental objects that enhances the aesthetic appearance of an object. The color is a non-spatial object that is added to an object to modify some of its visual aspects. The MFC library, combined with the Win32 API, provides various actions you can use to take advantage of the various aspects of colors.

Three numeric values are used to create a color. Each one of these values is 8 bits. The first number is called red. The second is called green. The third is called blue:

	Bits							
<b>Red</b>	7	6	5	4	3	2	1	0
<b>Green</b>	7	6	5	4	3	2	1	0
<b>Blue</b>	7	6	5	4	3	2	1	0

Converted to decimal, each one of these numbers would produce:

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$

$$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$

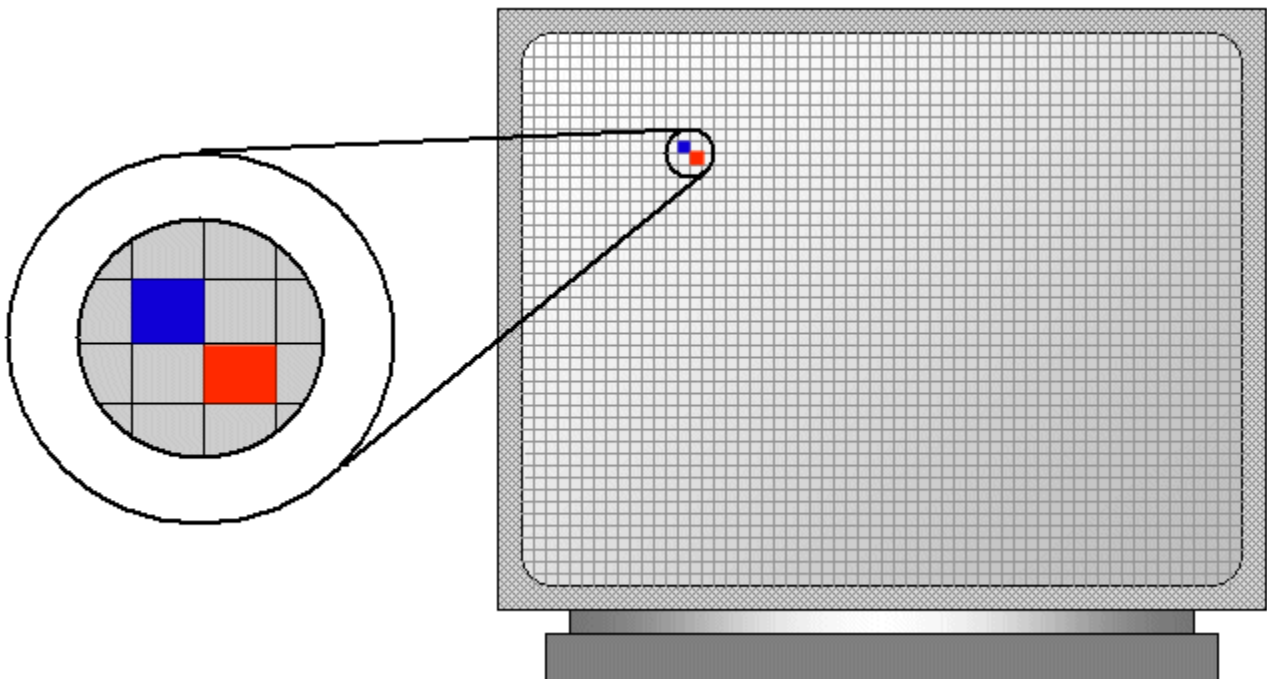
$$= 255$$

Therefore, each number can have a value that ranges from 0 to 255 in the decimal system. These three numbers are combined to produce a single number as follows:

Color	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>Value</b>	<b>Blue</b>								<b>Green</b>								<b>Red</b>							

Converted to decimal, this number has a value of  $255 * 255 * 255 = 16581375$ . This means that we can have approximately 16 million colors available. The question that comes to mind is how we use these colors, to produce what effect.

Your computer monitor has a surface that resembles a series of tinny horizontal and vertical lines. The intersection of a one horizontal line and a vertical line is called a pixel. This pixel holds, carries, or displays one color.



As the pixels close to each other have different colors, the effect is a wonderful distortion that creates an aesthetic picture. It is by changing the colors of pixels that you produce the effect of color variances seen on pictures and other graphics.

## The Color as a Data Type

Microsoft Windows considers that a color is a 32-bit numeric value. Therefore, a color is actually a combination of 32 bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

# Bitmaps

---

## Introduction



A bitmap is a series of points (bits) arranged like a map so that, when put together, they produce a picture that can be written to, copied from, re-arranged, changed, manipulated, or stored as a computer file. Bitmaps are used to display pictures on graphical applications, word processors, database files, or audience presentations. To display its product on a device such as a monitor or a printer, a bitmap holds some properties and follows a set of rules.

There are various types of bitmaps, based on the number of colors that the bitmap can display. First of all, a bitmap can be monochrome, in which case each pixel corresponds to 1 bit. A bitmap can also be colored. The number of colors that a bitmap can display is equal to 2 raised to the number of bits/pixel. For example, a simple bitmap uses only 4 bits/pixel or 4 bpp can handle only  $2^4 = 16$  colors. A more enhanced bitmap that requires 8 bpp can handle  $2^8 = 256$  colors. Bitmaps are divided in two categories that control their availability to display on a device.

A device-independent bitmap (DIB) is a bitmap that is designed to be loaded on any application or display on any device and produce the same visual effect. To make this possible, such a bitmap contains a table of colors that describes how the colors of the bitmap should be used on pixels when displaying it. The characteristics of a DIB are defined by the BITMAPINFO structure.

A device-dependent bitmap (DDB) is a bitmap created from the BITMAP structure using the dimensions of the bitmap.

## Bitmap Creation

Unlike the other GDI tools, creating a bitmap usually involves more steps. For example, you may want to create a bitmap to display on a window. You may create another bitmap to paint a geometric area, in which case the bitmap would be used as a brush.

Before creating a bitmap as a GDI object, you should first have a brush. You can do this by defining an array of unsigned hexadecimal numbers. Such a bitmap can be used for a brush.

One way you can use a bitmap is to display a picture on a window. To do this, you must first have a picture resource. Although the image editors of both Borland C++ Builder and Microsoft Visual C++ are meant to help with regular application resources, they have some limitations. Nevertheless, once your bitmap is ready, call the **LoadBitmap()** function. Its syntax is:

```
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);
```

The *hInstance* argument is the instance of the application that contains the bitmap you want to use

The *lpBitmapName* argument is the string that determines where the bitmap file is located. If you had imported the bitmap, you can use the MAKEINTRESOURCE macro to convert this string.

Before selecting the newly created bitmap object, allocate a block of computer memory that would hold the bitmap. You can then copy it to the actual device. This job can be taken care of by the **CreateCompatibleDC()** function. Its syntax is:

```
HBITMAP CreateCompatibleBitmap(HDC hdc, int nWidth, int nHeight);
```

This function takes a pointer to a device context. If it is successful, it returns TRUE or a non-zero value. If it is not, it returns FALSE or 0.

1. For an example, start MSVC to create a new Win32 Project and name it Bitmap1
2. Create it as a Windows Application and Empty Project
3. Save this bitmap in your computer as exercise.bmp



4. On the main menu, click either (MSVC 6) Insert -> Resource... or (MSVC .Net) Project -> Add Resource...
5. Click the Import... button
6. Locate the above picture from your computer and open or import it
7. In the Properties window, change its ID to IDB\_EXERCISING
8. Save the resource. If you are using MSVC 6, save the resource script as Bitmap1.rc then add it to your project (Project -> Add To Project -> Files... Make sure you select the file with rc extension)
9. Add a new C++ (Source) File named Exercise and implement it as follows:

```
#include <windows.h>
#include "Resource.h"

HINSTANCE hInst;
LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam,
LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
```

```

static char szAppName[] = "BitmapIntro";
MSG          Msg;

    hInst          = hInstance;
WndCls.cbSize    = sizeof(WndCls);
WndCls.style     = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
WndCls.lpfWndProc = WindProcedure;
WndCls.cbClsExtra = 0;
WndCls.cbWndExtra = 0;
WndCls.hInstance = hInst;
WndCls.hIcon     = LoadIcon(NULL, IDI_APPLICATION);
WndCls.hCursor   = LoadCursor(NULL, IDC_ARROW);
WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndCls.lpszMenuName = NULL;
WndCls.lpszClassName = szAppName;
WndCls.hIconSm    = LoadIcon(hInstance, IDI_APPLICATION);
RegisterClassEx(&WndCls);

CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
               szAppName,
               "Bitmaps Fundamentals",
               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               CW_USEDEFAULT,
               CW_USEDEFAULT,
               CW_USEDEFAULT,
               CW_USEDEFAULT,
               NULL,
               NULL,
               hInstance,
               NULL);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage( &Msg);
}

return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC hDC, MemDCExercising;
    PAINTSTRUCT Ps;
    HBITMAP bmpExercising;

    switch(Msg)
    {
        case WM_DESTROY:
            PostQuitMessage(WM_QUIT);
            break;
        case WM_PAINT:
            hDC = BeginPaint(hWnd, &Ps);

            // Load the bitmap from the resource

```

```

        bmpExercising = LoadBitmap(hInst,
MAKEINTRESOURCE(IDB_EXERCISING));
        // Create a memory device compatible with the above DC
variable
        MemDCEexercising = CreateCompatibleDC(hDC);
        // Select the new bitmap
        SelectObject(MemDCEexercising, bmpExercising);

        // Copy the bits from the memory DC into the current dc
        BitBlt(hDC, 10, 10, 450, 400, MemDCEexercising, 0, 0,
SRCCOPY);

        // Restore the old bitmap
        DeleteDC(MemDCEexercising);
        DeleteObject(bmpExercising);
        EndPaint(hWnd, &Ps);
        break;
default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}

```

10. Test

the

application



11. Return to your programming environment

## GDI Tools - Fonts



### Introduction

A font is a list of symbols that can be drawn on a device context to produce a symbol. A font is designed by an artist but usually follows a specific pattern. For example a font designed to produce symbols readable in the English language must be designed by a set of predetermined



and agreed upon symbols. These English symbols are grouped in an entity called the English alphabet. When designing such a font, the symbols created must conform to that language. This also implies that one font can be significantly different from another and a font is not necessarily a series of readable symbols.

Just like everything else in the computer, a font must have a name. To accommodate the visual needs, a font is designed to assume different sizes.

## Font Selection

---

Before using a font to draw a symbol on a device, the font must have been installed. Microsoft Windows installs many fonts during setup. To handle its various assignments, the operating system uses a particular font known as the System Font. This is the font used to display the menu items and other labels for resources in applications. If you want to use a different font to draw text in your application, you must select it.

Selecting a font, as well as selecting any other GDI object, is equivalent to specifying the characteristics of a GDI object you want to use. To do this, you must first create the object, unless it exists already. To select an object, pass it as a pointer to the **SelectObject()** function. The syntax of this function is:

```
HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hgdiobj);
```

This function takes as argument the font you want to use, *hgdiobj*. It returns a pointer to the font that was previously selected. If there was a problem when selecting the font, the function returns NULL. As you can see, you must first have a font you want to select.

## Regular Font Creation

---

A font in Microsoft Windows is stored as an **HFONT** value.

To Create a font, you can use the **CreateFont()** function. Its syntax is:

```
HFONT CreateFont(  
    int nHeight,  
    int nWidth,  
    int nEscapement,  
    int nOrientation,  
    int fnWeight,  
    DWORD fdwItalic,  
    DWORD fdwUnderline,  
    DWORD fdwStrikeOut,  
    DWORD fdwCharSet,  
    DWORD fdwOutputPrecision,  
    DWORD fdwClipPrecision,  
    DWORD fdwQuality,  
    DWORD fdwPitchAndFamily,  
    LPCTSTR lpszFace  
);
```

The *nHeight* argument is the height applied to the text.

The *nWidth* value is the desired width that will be applied on the text.

The *nEscapement* is the angle used to orient the text. The angle is calculated as a multiple of 0.1 and oriented counterclockwise.

The `nOrientation` is the angular orientation of the text with regards to the horizontal axis.

The `nWeight` is used to attempt to control the font weight of the text because it is affected by the characteristics of the font as set by the designer. It holds values that displays text from thin heavy bold. The possible values are:

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

The `bItalic` specifies whether the font will be italicized (TRUE) or not (FALSE).

The `bUnderline` is used to underline (TRUE) or not underline (FALSE) the text.

The `cStrikeOut` is specifies whether the text should be stroke out (TRUE) or not (FALSE) with a line.

The `nCharSet`, specifies the character set used. The possible values are:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

The `nOutPrecision` controls the amount precision used to evaluate the numeric values used on this function for the height, the width, and angles. It can have one of the following values: `OUT_CHARACTER_PRECIS`, `OUT_STRING_PRECIS`, `OUT_DEFAULT_PRECIS`, `OUT_STROKE_PRECIS`, `OUT_DEVICE_PRECIS`, `OUT_TT_PRECIS`, `OUT_RASTER_PRECIS`

If some characters may be drawn outside of the area in which they are intended, the `nClipPrecision` is used to specify how they may be clipped. The possible value used are `CLIP_CHARACTER_PRECIS`, `CLIP_MASK`, `CLIP_DEFAULT_PRECIS`, `CLIP_STROKE_PRECIS`, `CLIP_ENCAPSULATE`, `CLIP_TT_ALWAYS`, `CLIP_LH_ANGLES`.

The `nQuality` specifies how the function will attempt to match the font's characteristics. The

possible values are DEFAULT\_QUALITY, PROOF\_QUALITY, and DRAFT\_QUALITY. The nPitchAndFamily specifies the category of the font used. It combines the pitch and the family the intended font belongs to. The pitch can be specified with DEFAULT\_PITCH, VARIABLE\_PITCH, or FIXED\_PITCH. The pitch is combined using the bitwise OR operator with one of the following values:

Constant	Description
FF_DECORATIVE	Used for a decorative or fancy font
FF_DONTCARE	Let the compiler specify
FF_MODERN	Modern fonts that have a constant width
FF_ROMAN	Serif fonts with variable width
FF_SCRIPT	Script-like fonts
FF_SWISS	Sans serif fonts with variable width

The lpszFacename is the name of the font used.

Once you have created a font, you can select it into the device context and use it for example to draw text. After using a font, you should delete it to reclaim the memory space its variable was using. This is done by calling the **DeleteObject()** function.

Here is an example:

```
#include <windows.h>

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
    static char szAppName[] = "ExoFont";
    MSG        Msg;

    WndCls.cbSize           = sizeof(WndCls);
    WndCls.style            = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
    WndCls.lpfnWndProc     = WindProcedure;
    WndCls.cbClsExtra      = 0;
    WndCls.cbWndExtra      = 0;
    WndCls.hInstance       = hInstance;
    WndCls.hIcon            = LoadIcon(NULL, IDI_APPLICATION);
    WndCls.hCursor         = LoadCursor(NULL, IDC_ARROW);
    WndCls.hbrBackground   = (HBRUSH) GetStockObject(WHITE_BRUSH);
    WndCls.lpszMenuName    = NULL;
    WndCls.lpszClassName   = szAppName;
    WndCls.hIconSm         = LoadIcon(hInstance, IDI_APPLICATION);
    RegisterClassEx(&WndCls);
}
```

```

CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
               szAppName, "Fonts Fundamentals",
               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               CW_USEDEFAULT, CW_USEDEFAULT, 450, 220,
               NULL, NULL, hInstance, NULL);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage( &Msg);
}

return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HFONT        font;

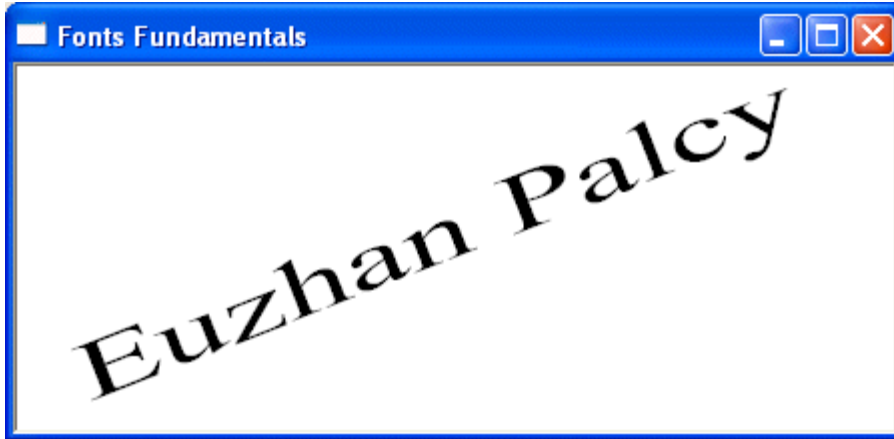
    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        font = CreateFont(46, 28, 215, 0,
                          FW_NORMAL, FALSE, FALSE, FALSE,
                          ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                          CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                          DEFAULT_PITCH | FF_ROMAN,
                          "Times New Roman");

        SelectObject(hDC, font);
        TextOut(hDC, 20, 128, "Euzhan Palcy", 12);
        DeleteObject(font);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```



Remember that once an object such as a font has been selected, it remains in the device context until further notice. For example, if you have created and selected a font, any text you draw would follow the characteristics of that font. If you want another font, you must change the previously selected font.

The computer uses the default black color to draw the text. If you want to draw text with a different color, you can first call the **SetTextColor()** function and specify the color of your choice.

## Logical Font Creation

---

The **CreateFont()** function is used to specify all characteristics of a font in one step. Alternatively, if you want to specify each font property, you can declare a **LOGFONT** variable and initialize it. It is defined as follows:

```
typedef struct tagLOGFONT {
LONG lfHeight;
LONG lfWidth;
LONG lfEscapement;
LONG lfOrientation;
LONG lfWeight;
BYTE lfItalic;
BYTE lfUnderline;
BYTE lfStrikeOut;
BYTE lfCharSet;
BYTE lfOutPrecision;
BYTE lfClipPrecision;
BYTE lfQuality;
BYTE lfPitchAndFamily;
TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT;
```

This time, you do not have to provide a value for each member of the structure and even if you do, you can supply values in the order of your choice. For any member whose value is not specified, the compiler would use a default value but you may not like some the result. Therefore, you should specify as many values as possible.

After initializing the **LOGFONT** variable, call the **CreateFontIndirect()** function. Its syntax is:

```
BOOL CreateFontIndirect(const LOGFONT* lpLogFont);
```

When calling this member function, pass the **LOGFONT** variable as a pointer, *lpLogFont*.

To select the newly created font, call the **SelectObject()** function. Once done, you can use the new font as you see fit. Here is an example:

```
LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HFONT        font;
    LOGFONT      LogFont;

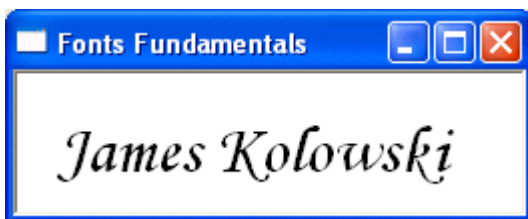
    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        LogFont.lfStrikeOut = 0;
        LogFont.lfUnderline = 0;
        LogFont.lfHeight = 42;
        LogFont.lfEscapement = 0;
        LogFont.lfItalic = TRUE;

        font = CreateFontIndirect(&LogFont);
        SelectObject(hDC, font);
        TextOut(hDC, 20, 18, "James Kolowski", 14);

        DeleteObject(font);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```



## Font Retrieval

---

If some text is displaying and you want to get the font properties of that text, you can call the **GetObject()** function. Its syntax is:

```
int GetObject(HGDIOBJ hgdiobj, int cbBuffer, LPVOID lpvObject);
```

# Pens

## The Fundamentals of a Pen

As mentioned already, in order to draw, two primary objects are needed: a platform and a tool. So far, we've seen the platform, called a device context. We introduced the device context class as `HDC`. The device context is a container for the platform on which the drawing is performed and the necessary tools to draw on it.

A pen is a tool used to draw lines and curves on a device context. In the graphics programming, a pen is also used to draw the borders of a geometric closed shape such as a rectangle or a polygon.

To make it an efficient tool, a pen must produce some characteristics on the lines it is asked to draw. These characteristics can range from the width of the line drawn to their colors, from the pattern applied to the level of visibility of the line. To manage these properties, Microsoft Windows considers two types of pens: cosmetic and geometric.




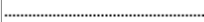

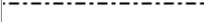

- A pen is referred to as cosmetic when it can be used to draw only simple lines of a fixed width, less than or equal to 1 pixel.
- A pen is geometric when it can assume different widths and various ends.

## Creating and Selecting a Pen

To create a pen, you can call the **CreatePen()** function. Its syntax is:

```
HPEN CreatePen(int fnPenStyle, int nWidth, COLORREF crColor);
```

The *fnPenStyle* argument is characteristic is referred to as the style of the pen. The possible values of this argument are:

Value	Illustration	Description
PS_SOLID		A continuous solid line
PS_DASH		A continuous line with dashed interruptions
PS_DOT		A line with a dot interruption at every other pixel
PS_DASHDOT		A combination of alternating dashed and dotted points
PS_DASHDOTDOT		A combination of dash and double dotted interruptions
PS_NULL		No visible line
PS_INSIDEFRAME		A line drawn just inside of the border of a closed shape

When creating, as cosmetic or geometric, use the bitwise OR operator to combine one or more of the above styles with the `PS_SOLID` style. To specify the type of pen, use the `PS_COSMETIC` or `PS_GEOMETRIC` style. The following are the possible values:

- **PS\_COSMETIC**: used to create a cosmetic pen
- **PS\_GEOMETRIC**: used to create a geometric pen

If you are creating a cosmetic pen, you can also add (bitwise OR) the **PS\_ALTERNATE** style to set the pen to draw the other side of the line.

The *nWidth* argument is the width used to draw the lines or borders of a closed shape. A cosmetic pen can have only 1 pixel. If you specify a higher width, it would be ignored. A geometric pen can have a width of 1 or more. The line can only be solid or null. This means that, if you specify the style as **PS\_DASH**, **PS\_DOT**, **PS\_DASHDOT**, **PS\_DASHDOTDOT** but set a width higher than 1, the line would be drawn as solid.

The default color of pen on the device context is black. If you want to control the color, specify the desired *crColor* argument.

After creating a pen, you can select it into the desired device context variable and then use it as you see in the next example drawing a rectangle. Here is an example:

```
#include <windows.h>

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
    static char szAppName[] = "ExoPen";
    MSG        Msg;

    WndCls.cbSize      = sizeof(WndCls);
    WndCls.style       = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
    WndCls.lpfnWndProc = WindProcedure;
    WndCls.cbClsExtra  = 0;
    WndCls.cbWndExtra  = 0;
    WndCls.hInstance  = hInstance;
    WndCls.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
    WndCls.hCursor     = LoadCursor(NULL, IDC_ARROW);
    WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndCls.lpszMenuName = NULL;
    WndCls.lpszClassName = szAppName;
    WndCls.hIconSm     = LoadIcon(hInstance, IDI_APPLICATION);
    RegisterClassEx(&WndCls);

    CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                  szAppName, "Pens Fundamentals",
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  CW_USEDEFAULT, CW_USEDEFAULT, 420, 220,
                  NULL, NULL, hInstance, NULL);

    while( GetMessage(&Msg, NULL, 0, 0) )
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }

    return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{

```



```

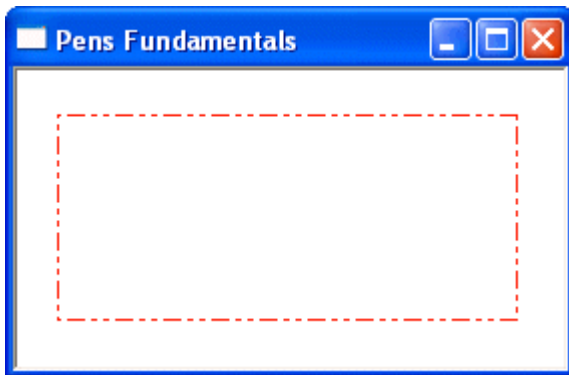
HDC          hDC;
PAINTSTRUCT Ps;
HPEN        hPen;

switch(Msg)
{
case WM_PAINT:
    hDC = BeginPaint(hWnd, &Ps);

    hPen = CreatePen(PS_DASHDOTDOT, 1, RGB(255, 25, 5));
    SelectObject(hDC, hPen);
    Rectangle(hDC, 20, 22, 250, 125);

    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}

```



Once a pen has been selected, any drawing performed and that uses a pen would use the currently selected pen. If you want to use a different pen, you can create a new pen. After using a pen, between exiting the function that created it, you should get rid of it and restore the pen that was selected previously. This is done by the `DeleteObject()` function as follows:

```

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HPEN        hPen;

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        hPen = CreatePen(PS_DASHDOTDOT, 1, RGB(255, 25, 5));
        SelectObject(hDC, hPen);
        Rectangle(hDC, 20, 22, 250, 125);
    }
}

```

```

        DeleteObject(hPen);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```

The Win32 API provides the LOGPEN structure that you can use to individually specify each characteristics of a **LOGPEN** structure is defined as follows:

```

typedef struct tagLOGPEN {
    UINT lopnStyle;
    POINT lopnWidth;
    COLORREF lopnColor;
} LOGPEN, *PLOGPEN;

```

To use this structure, declare a variable of **LOGPEN** type or a pointer. Then initialize each member of the structure. If you do not, its default values would be used and the line not be visible.

The *lopnStyle* argument follows the same rules we reviewed for the *nPenStyle* argument of the **CreatePen()** function.

The *lopnWidth* argument is provided as a POINT value. Only the POINT::x value is used.

The *lopnColor* argument is a color and can be provided following the rules we reviewed for colors.

After initializing the **LOGPEN** variable, call the **CreatePenIndirect()** function to create a pen. The syntax of the **CreatePenIndirect()** function is:

```

HPEN CreatePenIndirect(CONST LOGPEN *lplgpn);

```

The **LOGPEN** value is passed to this method as a pointer. After this call, the new pen is available and can be used in a device context variable for use. Here is an example:

```

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HPEN         hPen;
    LOGPEN       LogPen;
    POINT        Pt = { 1, 105 };

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);
        LogPen.lopnStyle = PS_SOLID;
        LogPen.lopnWidth = Pt;
        LogPen.lopnColor = RGB(235, 115, 5);

        hPen = CreatePenIndirect(&LogPen);
    }
}

```

```

SelectObject (hDC, hPen);

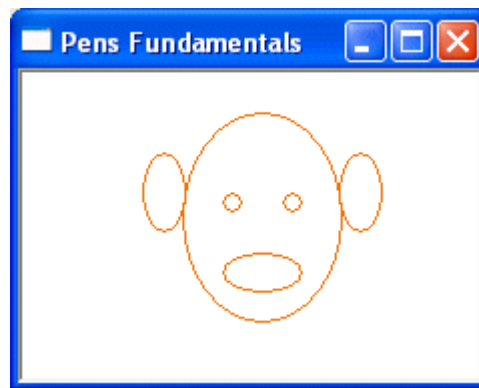
Ellipse (hDC, 60, 40, 82, 80);
Ellipse (hDC, 80, 20, 160, 125);
Ellipse (hDC, 158, 40, 180, 80);

Ellipse (hDC, 100, 60, 110, 70);
Ellipse (hDC, 130, 60, 140, 70);
Ellipse (hDC, 100, 90, 140, 110);

DeleteObject (hPen);

    EndPaint (hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage (WM_QUIT);
    break;
default:
    return DefWindowProc (hWnd, Msg, wParam, lParam);
}
return 0;
}

```



## Retrieving a Pen

If you want to know the currently selected pen used on a device context, you can call the **GetObject()** member

# Brushes

---

## Introduction

A brush is a drawing tool used to fill out closed shapes or the interior of lines. Using a brush is like picking up a bucket of paint and pouring it somewhere. In the case of computer graphics, the area where you position the brush is called the brush origin. The color or pattern that the brush holds would be used to fill the whole area until the brush finds a limit set by some rule.

A brush can be characterized by its color (if used), its pattern used to fill the area, or a picture (bitmap) used as the brush.

Because there can be so many variations of brushes, there are different functions for the various possible types of brushes you would need. The easiest brush you can create is made of a color.

## Solid Brushes

---

A brush is referred to as solid if it is made of a color simply used to fill a closed shape. To create a solid brush, call the `CreateSolidBrush()` function. Its syntax is:

```
HBRUSH CreateSolidBrush(COLORREF crColor);
```

The color to provide as the `crColor` argument follows the rules we reviewed for colors.

To use the newly created brush, you can select it into the device context by calling the **SelectObject()** function. Once this is done. Any closed shape you draw (ellipse, rectangle, polygon) would be filled with the color specified. After using the brush, you can delete it and restore the previous brush. Here is an example:

```
#include <windows.h>

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
    static char szAppName[] = "ExoBrush";
    MSG        Msg;

    WndCls.cbSize      = sizeof(WndCls);
    WndCls.style       = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
    WndCls.lpfnWndProc = WindProcedure;
    WndCls.cbClsExtra  = 0;
    WndCls.cbWndExtra  = 0;
    WndCls.hInstance   = hInstance;
    WndCls.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
    WndCls.hCursor      = LoadCursor(NULL, IDC_ARROW);
    WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndCls.lpszMenuName = NULL;
    WndCls.lpszClassName = szAppName;
    WndCls.hIconSm      = LoadIcon(hInstance, IDI_APPLICATION);
    RegisterClassEx(&WndCls);

    CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                  szAppName, "GDI Brushes Fundamentals",
                  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                  CW_USEDEFAULT, CW_USEDEFAULT, 420, 220,
                  NULL, NULL, hInstance, NULL);

    while( GetMessage(&Msg, NULL, 0, 0) )
    {
        TranslateMessage(&Msg);
        DispatchMessage( &Msg);
    }
}
```

```

    return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HBRUSH       NewBrush;

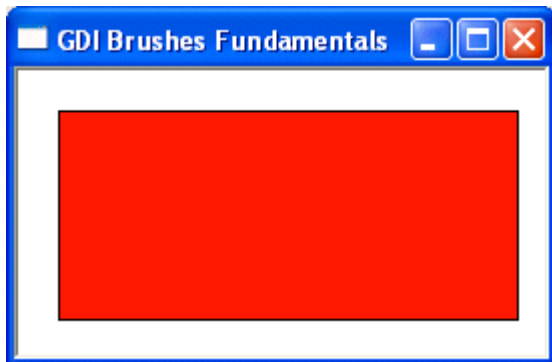
    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        NewBrush = CreateSolidBrush( RGB(250, 25, 5) );

        SelectObject(hDC, NewBrush);
        Rectangle(hDC, 20, 20, 250, 125);
        DeleteObject(NewBrush);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```



Once a brush has been selected, it would be used on all shapes that are drawn under it, until you delete or change it. Here is an example:

```

#include <windows.h>

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{

```

```

WNDCLASSEX WndCls;
static char szAppName[] = "ExoBrush";
MSG        Msg;

WndCls.cbSize      = sizeof(WndCls);
WndCls.style       = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
WndCls.lpfnWndProc = WindProcedure;
WndCls.cbClsExtra  = 0;
WndCls.cbWndExtra  = 0;
WndCls.hInstance   = hInstance;
WndCls.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
WndCls.hCursor     = LoadCursor(NULL, IDC_ARROW);
WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndCls.lpszMenuName = NULL;
WndCls.lpszClassName = szAppName;
WndCls.hIconSm     = LoadIcon(hInstance, IDI_APPLICATION);
RegisterClassEx(&WndCls);

CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
               szAppName, "GDI Brushes Fundamentals",
               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
               CW_USEDEFAULT, CW_USEDEFAULT, 400, 280,
               NULL, NULL, hInstance, NULL);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage( &Msg);
}

return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC        hDC;
    PAINTSTRUCT Ps;
    HBRUSH     NewBrush;
    POINT      Pt[3];

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        NewBrush = CreateSolidBrush(RGB(255, 2, 5));
        SelectObject(hDC, NewBrush);

        // Top Triangle
        Pt[0].x = 125; Pt[0].y = 10;
        Pt[1].x = 95; Pt[1].y = 70;
        Pt[2].x = 155; Pt[2].y = 70;

        Polygon(hDC, Pt, 3);
    }
}

```

```

// Left Triangle
Pt[0].x = 80; Pt[0].y = 80;
Pt[1].x = 20; Pt[1].y = 110;
Pt[2].x = 80; Pt[2].y = 140;

Polygon(hdc, Pt, 3);

// Bottom Triangle
Pt[0].x = 95; Pt[0].y = 155;
Pt[1].x = 125; Pt[1].y = 215;
Pt[2].x = 155; Pt[2].y = 155;

Polygon(hdc, Pt, 3);

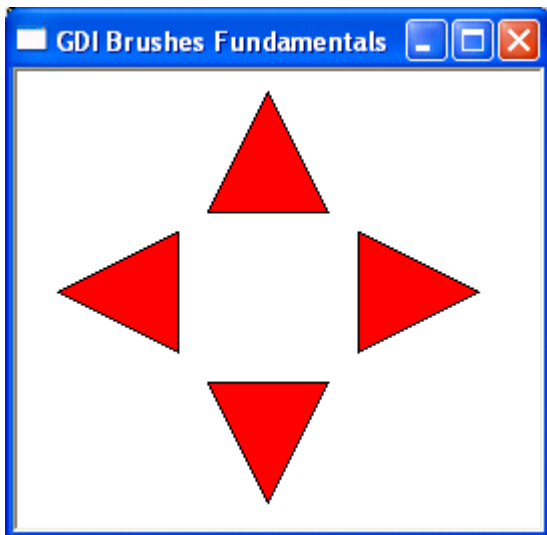
// Right Triangle
Pt[0].x = 170; Pt[0].y = 80;
Pt[1].x = 170; Pt[1].y = 140;
Pt[2].x = 230; Pt[2].y = 110;

Polygon(hdc, Pt, 3);

DeleteObject(NewBrush);

        EndPaint(hWnd, &Ps);
        break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}

```



If you want to use a different brush, you should create a new one. Here is an example:

```
LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT Msg,
```

```

                                WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HBRUSH BrushGreen = CreateSolidBrush( RGB(0, 125, 5) );
    HBRUSH BrushRed   = CreateSolidBrush( RGB(255, 2, 5) );
    HBRUSH BrushYellow = CreateSolidBrush( RGB(250, 255, 5) );
    HBRUSH BrushBlue  = CreateSolidBrush( RGB(0, 2, 255) );

    POINT        Pt[3];

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        // Top Triangle
        Pt[0].x = 125; Pt[0].y = 10;
        Pt[1].x = 95;  Pt[1].y = 70;
        Pt[2].x = 155; Pt[2].y = 70;

        SelectObject(hDC, BrushGreen);
        Polygon(hDC, Pt, 3);

        // Left Triangle
        Pt[0].x = 80; Pt[0].y = 80;
        Pt[1].x = 20; Pt[1].y = 110;
        Pt[2].x = 80; Pt[2].y = 140;

        SelectObject(hDC, BrushRed);
        Polygon(hDC, Pt, 3);

        // Bottom Triangle
        Pt[0].x = 95; Pt[0].y = 155;
        Pt[1].x = 125; Pt[1].y = 215;
        Pt[2].x = 155; Pt[2].y = 155;

        SelectObject(hDC, BrushYellow);
        Polygon(hDC, Pt, 3);

        // Right Triangle
        Pt[0].x = 170; Pt[0].y = 80;
        Pt[1].x = 170; Pt[1].y = 140;
        Pt[2].x = 230; Pt[2].y = 110;

        SelectObject(hDC, BrushBlue);
        Polygon(hDC, Pt, 3);

        DeleteObject(BrushGreen);
        DeleteObject(BrushRed);
        DeleteObject(BrushYellow);
        DeleteObject(BrushBlue);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);

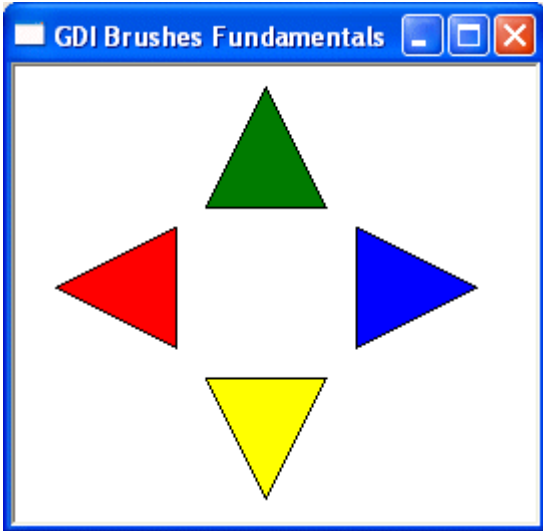
```



```

        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```



## Hatched Brushes

A hatch brush is one that uses a drawn hatch pattern to regularly fill an area. Microsoft Windows provides 6 preset patterns for such a brush. To create a hatched brush, you can call the **CreateHatchBrush()** function. Its syntax is:

```
HBRUSH CreateHatchBrush(int fnStyle, COLORREF clrref);
```

The *fnStyle* argument specifies the hatch style that must be used to fill the area. The possible values to use are **HS\_BDIAGONAL**, **HS\_CROSS**, **HS\_DIAGCROSS**, **HS\_FDIAGONAL**, **HS\_HORIZONTAL**, or **HS\_VERTICAL**.

The *clrref* argument specifies the color applied on the drawn pattern.

Here is an example:

```

#include <windows.h>

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
    static char szAppName[] = "ExoBrush";
    MSG        Msg;

    WndCls.cbSize        = sizeof(WndCls);
    WndCls.style         = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
    WndCls.lpfnWndProc   = WindProcedure;
    WndCls.cbClsExtra    = 0;

```

```

WndCls.cbWndExtra      = 0;
WndCls.hInstance      = hInstance;
WndCls.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
WndCls.hCursor        = LoadCursor(NULL, IDC_ARROW);
WndCls.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndCls.lpszMenuName   = NULL;
WndCls.lpszClassName  = szAppName;
WndCls.hIconSm        = LoadIcon(hInstance, IDI_APPLICATION);
RegisterClassEx(&WndCls);

CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
              szAppName, "Hatch Brushes",
              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
              CW_USEDEFAULT, CW_USEDEFAULT, 520, 230,
              NULL, NULL, hInstance, NULL);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage( &Msg);
}

return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HBRUSH       brBDiagonal  = CreateHatchBrush(HS_BDIAGONAL,   RGB(0, 0, 255));
    HBRUSH       brCross      = CreateHatchBrush(HS_CROSS,       RGB(200, 0, 0));
    HBRUSH       brDiagCross  = CreateHatchBrush(HS_DIAGCROSS,   RGB(0, 128, 0));
    HBRUSH       brFDiagonal  = CreateHatchBrush(HS_FDIAGONAL,   RGB(0, 128, 192));
    HBRUSH       brHorizontal = CreateHatchBrush(HS_HORIZONTAL,  RGB(255, 128, 0));
    HBRUSH       brVertical   = CreateHatchBrush(HS_VERTICAL,    RGB(255, 0, 255));

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        SelectObject(hDC, brBDiagonal);
        RoundRect(hDC, 20, 30, 160, 80, 10, 10);

        SelectObject(hDC, brFDiagonal);
        RoundRect(hDC, 180, 30, 320, 80, 10, 10);

        SelectObject(hDC, brDiagCross);
        RoundRect(hDC, 340, 30, 480, 80, 10, 10);

        SelectObject(hDC, brVertical);
        RoundRect(hDC, 20, 120, 160, 170, 10, 10);

        SelectObject(hDC, brHorizontal);
        RoundRect(hDC, 180, 120, 320, 170, 10, 10);
    }
}

```

```

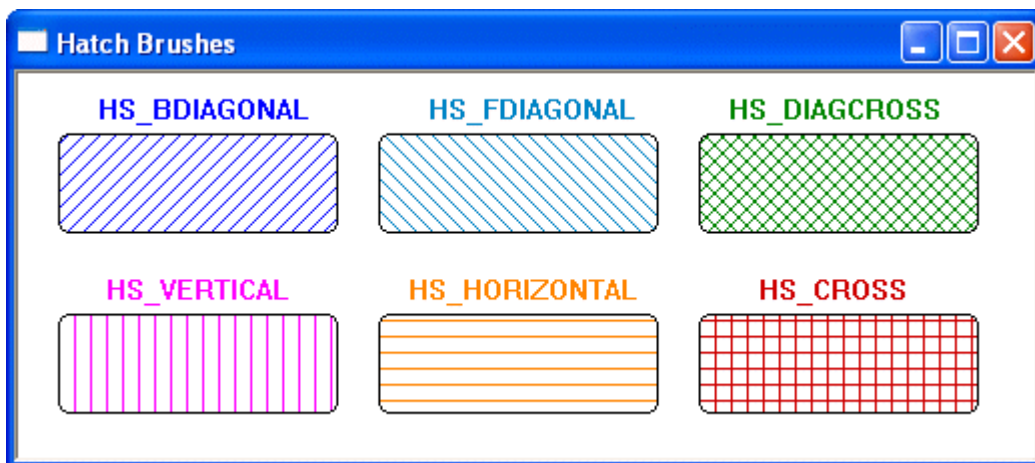
SelectObject(hDC, brCross);
RoundRect(hDC, 340, 120, 480, 170, 10, 10);

SetTextColor(hDC, RGB(0, 0, 255));
TextOut(hDC, 40, 10, "HS_BDIAGONAL", 12);
SetTextColor(hDC, RGB(0, 128, 192));
TextOut(hDC, 205, 10, "HS_FDIAGONAL", 12);
SetTextColor(hDC, RGB(0, 128, 0));
TextOut(hDC, 355, 10, "HS_DIAGCROSS", 12);
SetTextColor(hDC, RGB(255, 0, 255));
TextOut(hDC, 44, 100, "HS_VERTICAL", 11);
SetTextColor(hDC, RGB(255, 128, 0));
TextOut(hDC, 195, 100, "HS_HORIZONTAL", 13);
SetTextColor(hDC, RGB(200, 0, 0));
TextOut(hDC, 370, 100, "HS_CROSS", 8);

DeleteObject(brBDiagonal);
DeleteObject(brCross);
DeleteObject(brDiagCross);
DeleteObject(brFDiagonal);
DeleteObject(brHorizontal);
DeleteObject(brVertical);

    EndPaint(hWnd, &Ps);
    break;
case WM_DESTROY:
    PostQuitMessage(WM_QUIT);
    break;
default:
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
return 0;
}

```



## Patterned Brushes

A pattern brush is one that uses a bitmap or (small) picture to fill out an area. To create DDB bitmap, you can first create an array of WORD values. Then call the **CreateBitmap()** function to initialize it. As this makes the bitmap ready, call the **CreatePatternBrush()** function to initialize the brush. The syntax of this function is:

```
HBRUSH CreatePatternBrush(HBITMAP hbmp);
```

Once the brush has been defined, you can select it into a device context and use it as you see fit. For example, you can use it to fill a shape. Here is an example:

```
LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                                WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HBITMAP      BmpBrush;
    HBRUSH       brBits;
    WORD         wBits[] = { 0x00, 0x22, 0x44, 0x88, 0x00, 0x22, 0x44,
0x88,
                                0x22, 0x44, 0x88, 0x00, 0x22, 0x44, 0x88, 0x00,
                                0x44, 0x88, 0x00, 0x22, 0x44, 0x88, 0x00, 0x22,
                                0x88, 0x00, 0x22, 0x44, 0x88, 0x00, 0x22, 0x44 };

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

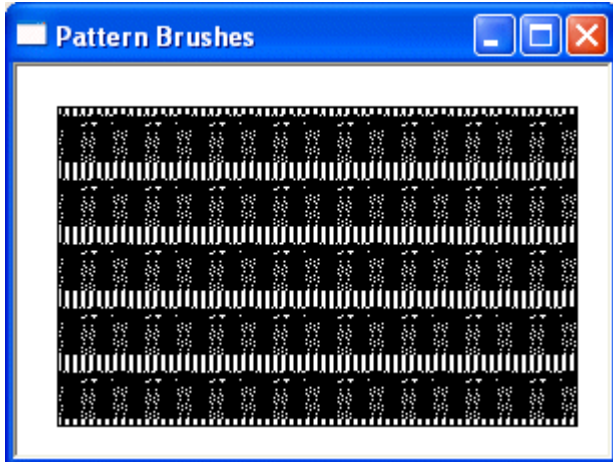
        BmpBrush = CreateBitmap(32, 32, 1, 1, wBits);
        brBits = CreatePatternBrush(BmpBrush);

        SelectObject(hDC, brBits);

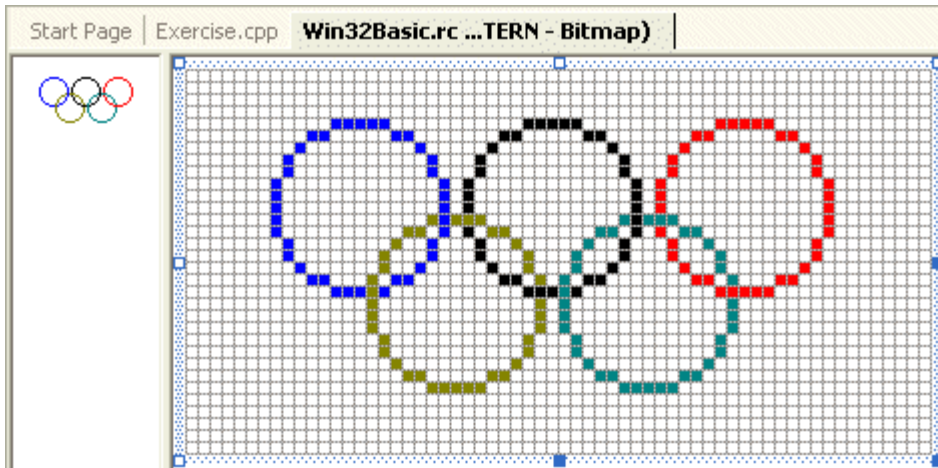
        Rectangle(hDC, 20, 20, 280, 180);

        DeleteObject(BmpBrush);

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}
```



Another technique you can use to create a pattern brush consists of using a bitmap resource. Before creating a pattern, you must first have a picture, which can be done by creating a bitmap. For example, imagine you create the following bitmap identified as IDB\_PATTERN:



To create a brush based on a bitmap, you can first load the bitmap either using `LoadBitmap()` or `CreateBitmap()`. Once the bitmap is ready, call the `CreatePatternBrush()` function to initialize it. This will allow you to get an `HBRUSH` that you can then select into the device context and use it as you see fit.

Here is an example:

```
#include <windows.h>
#include "resource.h"

HINSTANCE hInst;
LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndCls;
    static char szAppName[] = "ExoBrush";
```

```

MSG          Msg;

    hInst      = hInstance;
WndCls.cbSize      = sizeof(WndCls);
WndCls.style       = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
WndCls.lpfWndProc  = WindProcedure;
WndCls.cbClsExtra  = 0;
WndCls.cbWndExtra  = 0;
WndCls.hInstance  = hInst;
WndCls.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
WndCls.hCursor     = LoadCursor(NULL, IDC_ARROW);
WndCls.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
WndCls.lpszMenuName = NULL;
WndCls.lpszClassName = szAppName;
WndCls.hIconSm     = LoadIcon(hInstance, IDI_APPLICATION);
RegisterClassEx(&WndCls);

CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
              szAppName, "Pattern Brushes",
              WS_OVERLAPPEDWINDOW | WS_VISIBLE,
              CW_USEDEFAULT, CW_USEDEFAULT, 500, 240,
              NULL, NULL, hInstance, NULL);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}

return static_cast<int>(Msg.wParam);
}

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;
    PAINTSTRUCT Ps;
    HBITMAP      BmpBrush;
    HBRUSH       brPattern;
    HPEN         hPen = CreatePen(PS_SOLID, 1, RGB(255, 255, 255));

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

        BmpBrush = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_PATTERN));
        brPattern = CreatePatternBrush(BmpBrush);

        SelectObject(hDC, hPen);
        SelectObject(hDC, brPattern);

        Rectangle(hDC, 5, 3, 380, 280);

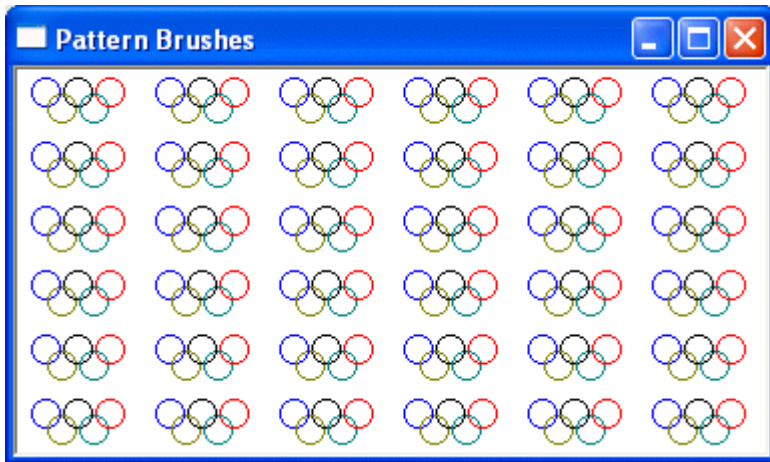
        DeleteObject(hPen);
        DeleteObject(BmpBrush);
    }
}

```

```

        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```



You can use this same approach to paint an area with a more sophisticated picture.

## Logical Brushes

The Win32 library provides the **LOGBRUSH** structure that can be used to create a brush by specifying its characteristics. **LOGBRUSH** is defined as follows:

```

typedef struct tagLOGBRUSH {
    UINT lbStyle;
    COLORREF lbColor;
    LONG lbHatch;
} LOGBRUSH, *PLOGBRUSH;

```

The `lbStyle` member variable specifies the style applied on the brush.

The `lbColor` is specified as a `COLORREF` value.

The `lbHatch` value represents the hatch pattern used on the brush.

To use this structure, declare and initialize a `LOGBRUSH` variable. Once the variable is ready, you can pass it to the **CreateBrushIndirect()** function. Its syntax is:

```

HBRUSH CreateBrushIndirect(CONST LOGBRUSH *lplb);

```

The **CreateBrushIndirect()** function returns an **HBRUSH** value that you can select into the device context and use it as you see fit. Here is an example:

```

LRESULT CALLBACK WindProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    HDC          hDC;

```

```

    PAINTSTRUCT Ps;
    HBRUSH      brLogBrush;
    LOGBRUSH    LogBrush;

    switch(Msg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &Ps);

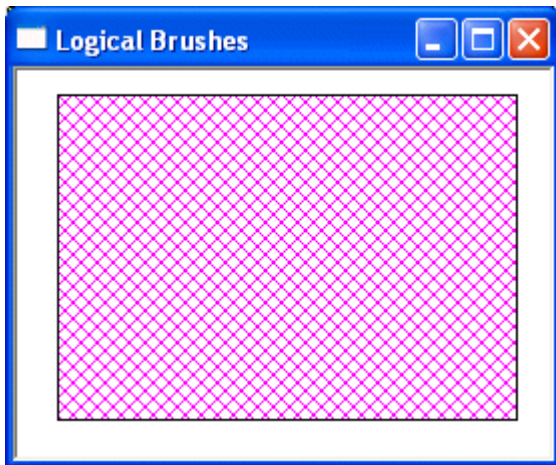
        LogBrush.lbStyle = BS_HATCHED;
        LogBrush.lbColor = RGB(255, 0, 255);
        LogBrush.lbHatch = HS_DIAGCROSS;

        brLogBrush = CreateBrushIndirect(&LogBrush);
        SelectObject(hDC, brLogBrush);

        Rectangle(hDC, 20, 12, 250, 175);

        DeleteObject(brLogBrush);
        EndPaint(hWnd, &Ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;
    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    return 0;
}

```



## Win32 Controls - Month Calendar

### The Month Calendar Control

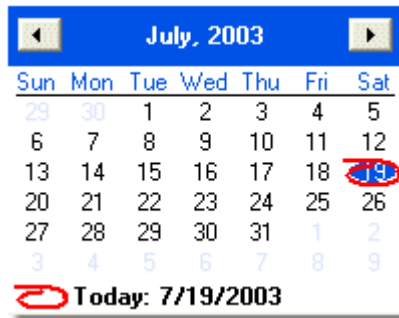




## **Introduction**

---

The Win32 API provides a control used to select dates on a colorful calendar. The dates used and the way they display are based on the Regional Settings of the Control Panel. It may also depend on the operating system:

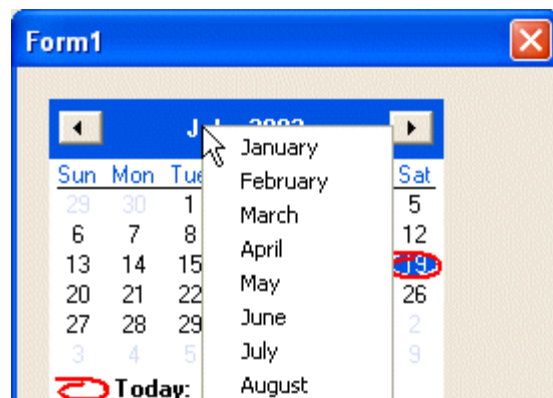


This convenient control is called Month Calendar. The title bar of the control displays two buttons and two labels. The left button allows the user to select the previous month by clicking the button. The left label displays the currently selected month. The right label displays the year of the displayed date. The right button is used to get to the next month.

The calendar can be configured to display more than one month. Here is an example that displays two months:



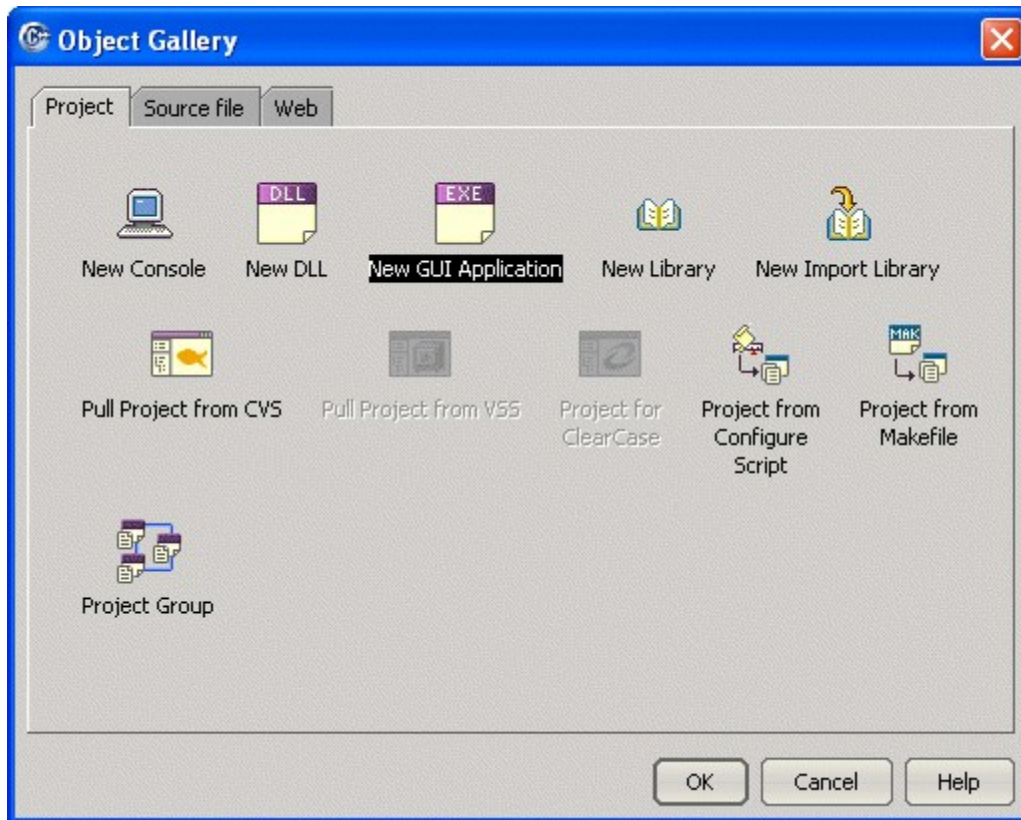
If the control is displaying more than one month, the buttons would increment or decrement by the previous or next month in the list. For example, if the control is displaying April and May, if the user clicks the left button, the control would display March and April. If the control is displaying April and May and the user clicks the right button, the control would display May and June. Also, to select any month of the current year, the user can click the name of the month, which displays the list of months and this allows the user to click the desired month:



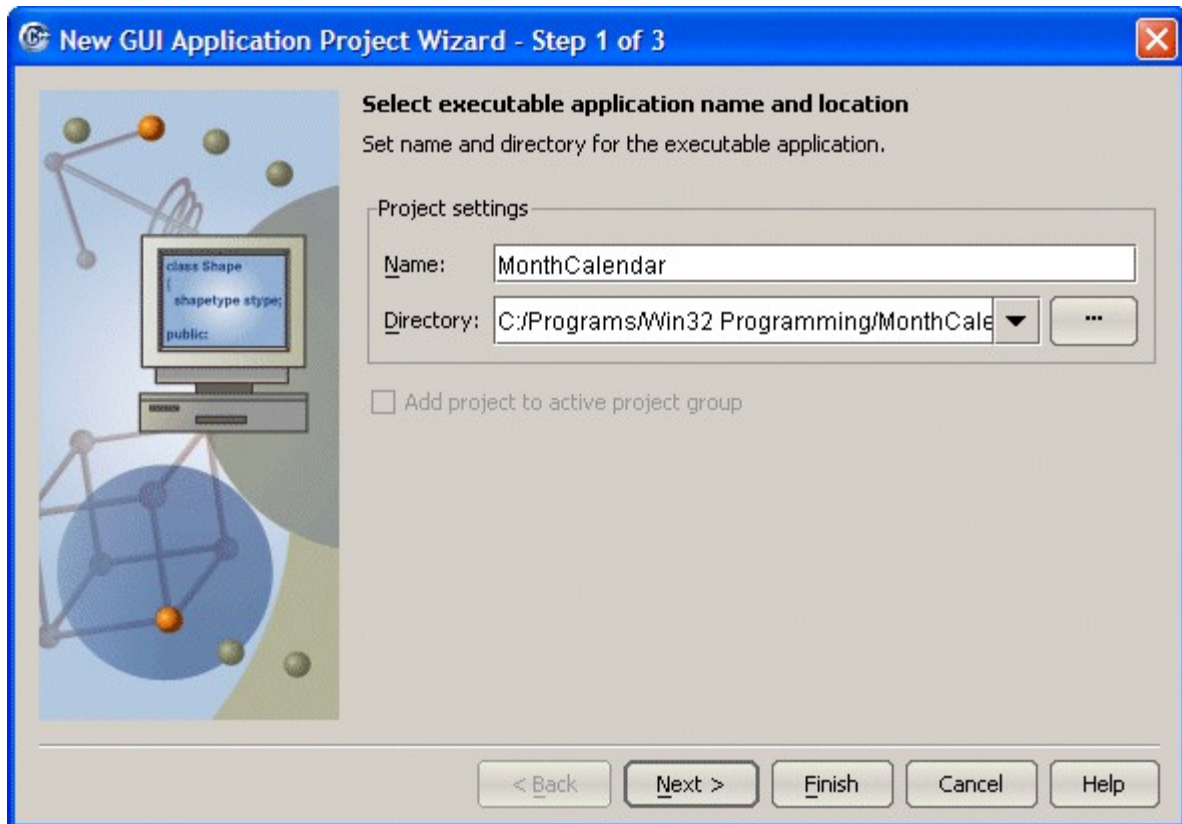
## ◆ Practical Learning: Creating the Application

---

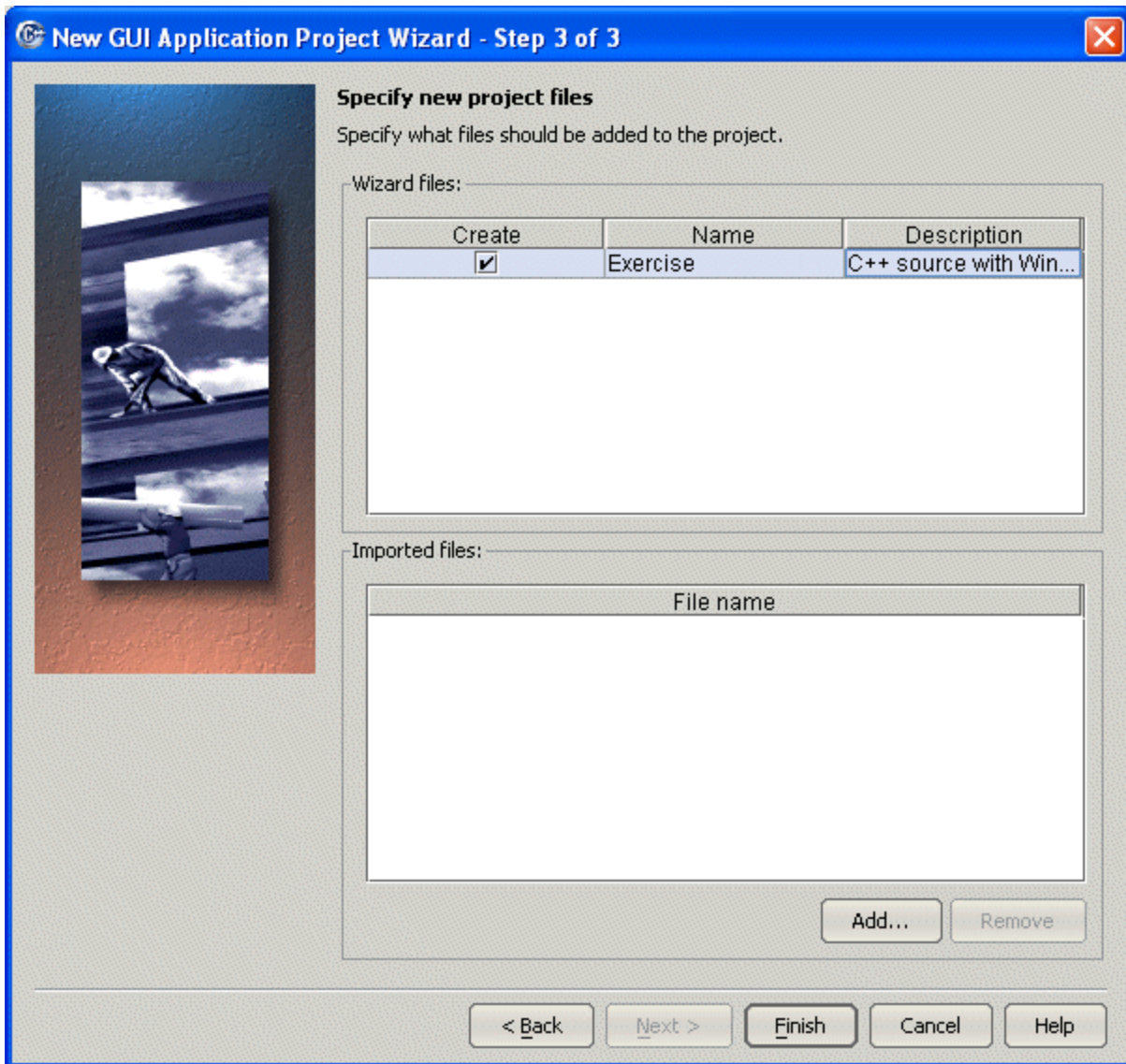
1. Because Borland **C++BuilderX** is free, we are going to use it.  
Start Borland C++BuilderX and, on the main menu, click File -> New...



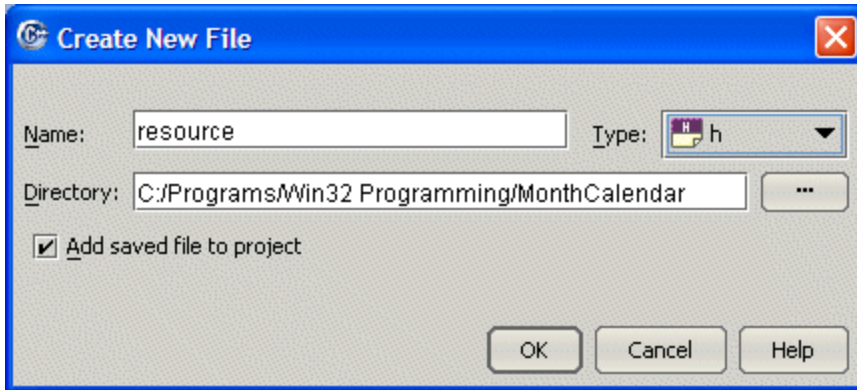
2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the New GUI Application Project Wizard - Step 1 of 3, in the Directory edit box of the Project Settings section, type the path you want. Otherwise, type **C:\Programs\Win32 Programming**
4. In the Name edit box, type **MonthCalendar**



5. Click Next
6. In the New GUI Application Project Wizard - Step 2 of 3, accept the defaults and click Next
7. In the New GUI Application Project Wizard - Step 3 of 3, click the check box under **Create**
8. Select **Untitled** under the Name column header. Type **Exercise** to replace the name and press Tab



9. Click Finish
10. To create the resource header file, on the main menu, click File -> New File...
11. In the Create New File dialog box, change the contents of the Name edit box with **resource**
12. In the Type combo box, select h



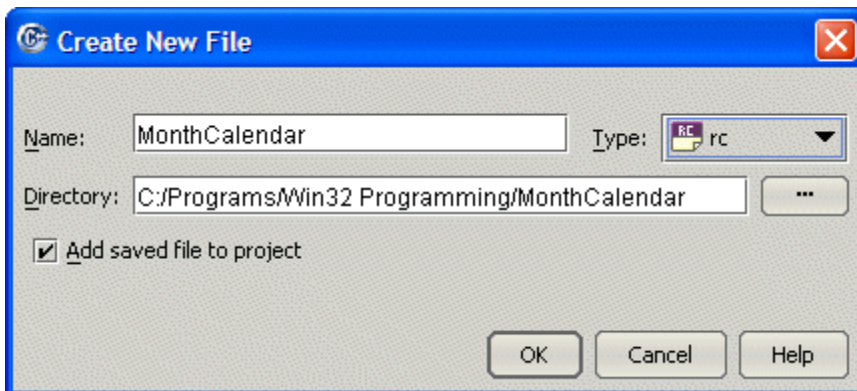
13. Click OK

14. In the file, type **#define IDD\_CONTROLSDLG 101**

15. To create the rc resource file, on the main menu of C++ Builder X, click File -> New File...

16. In the Create New File dialog box, change the contents of the Name edit box to **MonthCalendar**

17. In the Type combo box, select **rc**



18. Click OK

19. In the empty file, type the following (the referenced header file will be created next):

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 260, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON "Close", IDCANCEL, 202,14,50,14
END
```

20.

In the left frame, double-click **Exercise.cpp** and change the file to the following:

21.

```

#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"
//-----
//-----
HWND hWnd;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
//-----
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    DialogBox(hInstance, MAKEINTRESOURCE(IDD_CONTROLSDLG),
             hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return 0;
}
//-----
//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            return TRUE;
        }
        break;
    }

    return FALSE;
}
//-----
//-----

```

Press F9 to test the application

22. Click Close to dismiss the dialog box

## Month Calendar Creation

---

The Month Calendar control is part of the family of Common Controls defined in the comctl32.dll library. Therefore, before including it in your application, you must register it by calling the **InitCommonControlsEx()** function and assign the **ICC\_DATE\_CLASSES** value to

the **INITCOMMONCONTROLSEX::dwICC** member variable.

As always, there are two main ways you can create a control. You can use a resource script. If you use a script, the formula to follow is:

The easiest way to create a combo box is through a resource script. The syntax used to create the control in a script is:

```
CONTROL string, id, "SysMonthCal32", style, x, y, width, height
```

You must start with the **CONTROL** class to indicate that you are going to create a control from the **Comctl32.dll** family of controls.

The string factor can be anything. It is for controls that would need it. The Month Calendar control doesn't need it. Therefore, you can pass it as an empty string "" or provide any string you want.

The *id* is the number used to identify the control in a resource header file.

To indicate that you are creating a Month Calendar control, pass the class name as **SysMonthCal32**.

The *style* and the *extended-style* factors are used to configure and prepare the behavior of the combo box.

The *x* measure is its horizontal location with regards to the control's origin, which is located in the top left corner of the window that is hosting the combo box

The *y* factor is the distance from control's origin, which is located in the top left corner of the window that is hosting the combo box, to the top-left side of the combo box

The *width* and the *height* specify the dimensions of the control

Instead of using a script, to create a Month Calendar control, you can use the **CreateWindowEx()** function and pass the name of the class as **MONTHCAL\_CLASS**.

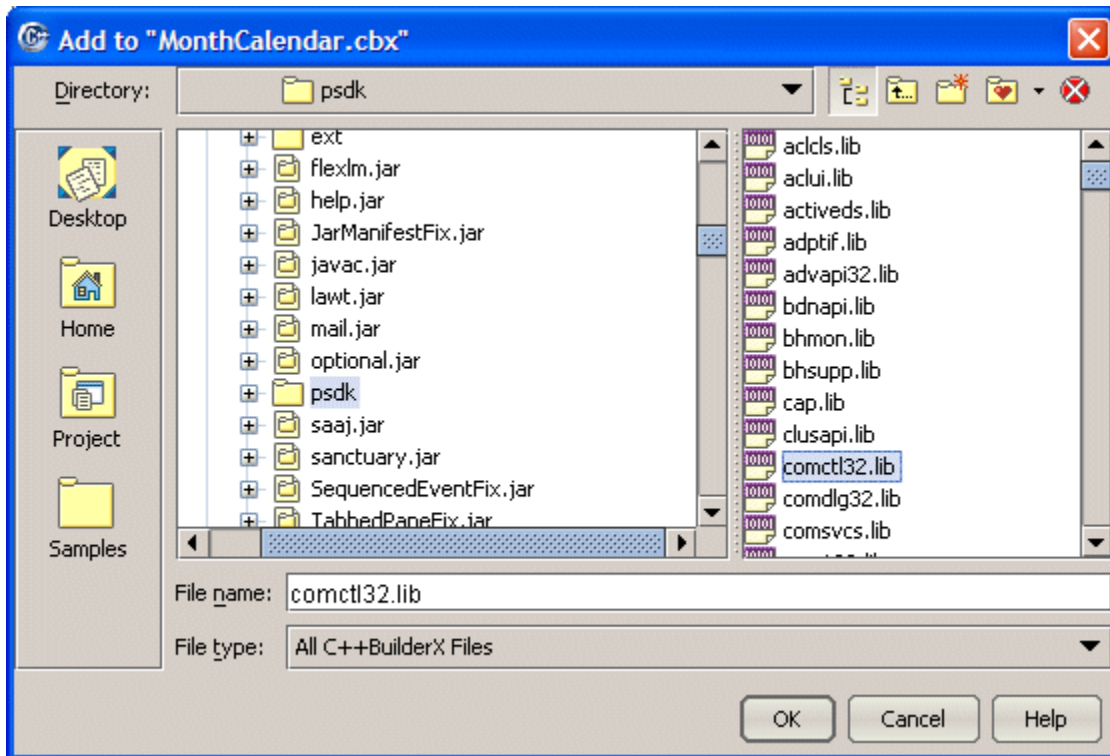
## Practical Learning: Creating the Control in a Script

---

1. To include the comctl32.dll library in your project, on the main menu of C++BuilderX, click Project -> Add Files
2. Navigate to *Drive:\C++BuilderX\lib\psdk* folder and display its content



- In the right frame, click comctl32.lib



- Click OK
- Change the resource.h header file as follows:

```
#define IDD_CONTROLSDLG 101
#define IDC_MONTHCALENDAR 102
```

- Change the MonthCalendar.rc resource script as follows:

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 200, 120
STYLE_DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    CONTROL        "", IDC_MONTHCALENDAR, "SysMonthCal32",
                  WS_CHILD | WS_TABSTOP, 10, 10, 118, 96
    DEFPUSHBUTTON  "Close", IDCANCEL, 140, 10, 50, 14
END
```

- Change the Exercise.cpp source file as follows:

```
#include <windows.h>
#include <commctrl.h>

#ifdef __BORLANDC__
    #pragma argsused
```

```

#endif

#include "resource.h"
//-----
-----
HWND hWnd;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
-----
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    DialogBox(hInstance, MAKEINTRESOURCE(IDD_CONTROLSDLG),
hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    INITCOMMONCONTROLSEX InitCtrlEx;

    InitCtrlEx.dwSize = sizeof(INITCOMMONCONTROLSEX);
    InitCtrlEx.dwICC = ICC_PROGRESS_CLASS;
    InitCommonControlsEx(&InitCtrlEx);

    return 0;
}
//-----
-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,
LPARAM lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            return TRUE;

        }
        break;
    }

    return FALSE;
}
//-----
-----

```

8. Test the application

## **The Properties of a Month Calendar Control**

---

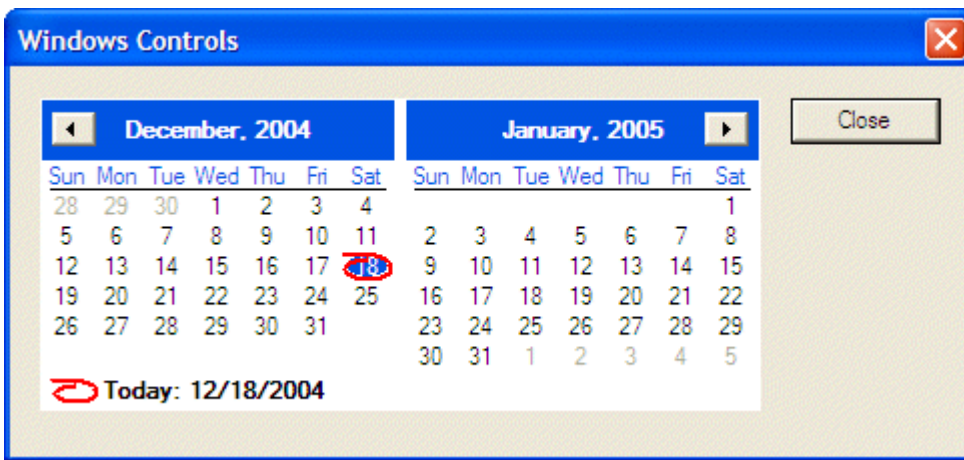
### **The Effects of the Control Dimensions**

---

After creating a Month Calendar control, it displays the current month and maybe only one month. To display more than one month, set the width of the control to provide enough space. Here is an example:

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 320, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    CONTROL        "", IDC_MONTHCALENDAR, "SysMonthCal32",
                  WS_CHILD | WS_TABSTOP, 10, 10, 240, 96
    DEFPUSHBUTTON  "Close", IDCANCEL, 260, 10, 50, 14
END
```



In the same way, you can increase the height to display many months.

## Colors: The Background of the Title Bar

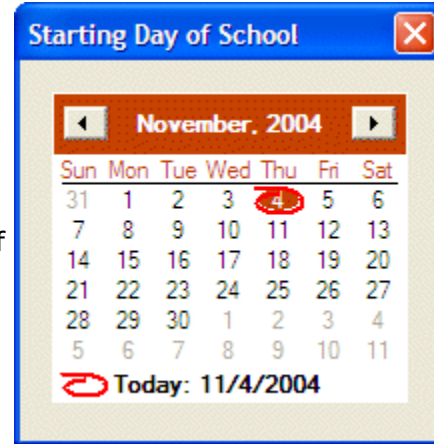
To make it a highly visual object, a calendar uses different colors to represent the background, week days, the background of the title bar, the text of the title bar, the text of the days of the previous month, and the text of the days of the subsequent month. Of course, you can programmatically change these colors. Although any color is allowed in any category, you should make sure that the calendar is still reasonably appealing and usable.

To change the colors of the Month Calendar control, you can call the **MonthCal\_SetColor()** function. Its syntax is:

```
COLORREF MonthCal_SetColor(HWND hwndMC, INT iColor, COLORREF clr);
```

The first argument is a handle to the Month Calendar control whose color you want to set or change.

By default, the title of the Month Calendar control appears on top of a blue background. If you want a different color, pass the *iColor* argument as **MCSC\_TITLEBK**, and pass the color of your choice as the *clr* argument.



Here is an example:

```
//-----
---
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HWND hWndMonthCal;

    hWndMonthCal = GetDlgItem(hWndDlg, IDC_MONTHCALENDAR);

    switch(Msg)
    {
    case WM_INITDIALOG:
        MonthCal_SetColor(hWndMonthCal, MCSC_TITLEBK, RGB(205, 50, 5));

        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            return TRUE;
        }
        break;
    }

    return FALSE;
}
//-----
---
```

Alternatively, you can send the **MCM\_SETCOLOR** message. The syntax to use would be:

```
lResult = SendMessage(HWND hWnd, MCM_SETCOLOR, WPARAM wParam, LPARAM lParam);
```

In this case, pass the *wParam* argument as **MCSC\_TITLEBK**, and pass the color of your choice as the *lParam* argument.

## ◆ Practical Learning: Changing the Background of the Title Bar

---

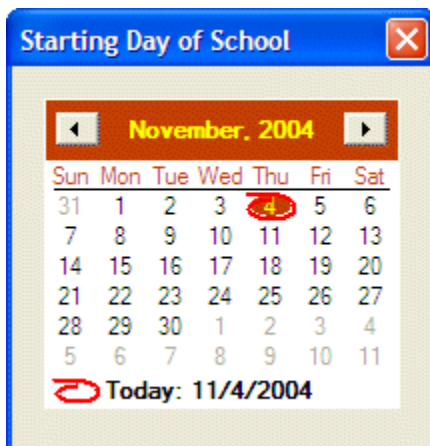
1. To set the background color of the title bar, change the procedure as follows:

```
//-----  
-----  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,  
LPARAM lParam)  
{  
    HWND hWndMonthCal;  
  
    hWndMonthCal = GetDlgItem(hWndDlg, IDC_MONTHCALENDAR);  
  
    switch(Msg)  
    {  
    case WM_INITDIALOG:  
        SendMessage(hWndMonthCal, MCM_SETCOLOR, MCSC_TITLEBK,  
RGB(205, 50, 5));  
  
        return TRUE;  
  
    case WM_COMMAND:  
        switch(wParam)  
        {  
        case IDCANCEL:  
            EndDialog(hWndDlg, 0);  
            return TRUE;  
        }  
        break;  
    }  
  
    return FALSE;  
}  
//-----  
-----
```

2. Test the application

### Colors: The Caption of the Title Bar

---



By default, the labels on the title bar display in a white color. To change the color used to paint the text of the labels, you can call the **MonthCal\_SetColor()** macro. Alternatively, you can send the **MCM\_SETCOLOR** message, pass the *wParam* argument as **MCSC\_TITLETEXT**, and pass the color of your choice as the *lParam* argument.

The names of weekdays use the same color as the color set when passing the **MCSC\_TITLETEXT** value as the *wParam* or the *iColor* argument.

Under the names of the week, there is a horizontal line used as the separator. By default, this line separator is painted in black but it uses the same color as the numeric values of the days of the selected month.

## ◆ Practical Learning: Changing the Color of the Title Bar Caption

---

1. To set the background color of the title bar, change the procedure as follows:

```
//-----  
-----  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,  
LPARAM lParam)  
{  
    HWND hWndMonthCal;  
  
    hWndMonthCal = GetDlgItem(hWndDlg, IDC_MONTHCALENDAR);  
  
    switch(Msg)  
    {  
    case WM_INITDIALOG:  
        SendMessage(hWndMonthCal, MCM_SETCOLOR, MCSC_TITLEBK,  
RGB(205, 50, 5));  
        MonthCal_SetColor(hWndMonthCal, MCSC_TITLETEXT, RGB(255,  
250, 5));  
  
        return TRUE;  
  
    case WM_COMMAND:  
        switch(wParam)  
        {  
        case IDCANCEL:  
            EndDialog(hWndDlg, 0);  
            return TRUE;  
        }  
        break;  
    }  
  
    return FALSE;  
}  
//-----  
-----
```

2. Test the application

## Colors: The Caption of the Title Bar

---

Under the names of the week and their line separator, the numeric days of the month are listed. These days display in a different color. To specify the color of the days of the current month, you can call the **MonthCal\_SetColor()** macro or send a **MCSC\_TEXT** message. Pass the *iColor* or the *wParam* as **MCSC\_TEXT** and pass the color of your choice as the *lParam* or the *Clr* argument.

## ◆ Practical Learning: Changing the Color of the Current Month Days

---

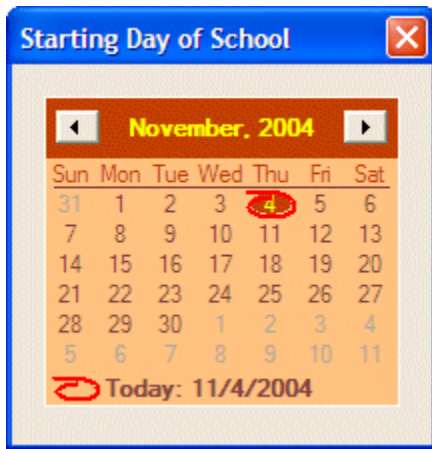
1. To set the background color of the title bar, change the procedure as follows:

```
//-----  
-----  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,  
LPARAM lParam)  
{  
    HWND hWndMonthCal;  
  
    hWndMonthCal = GetDlgItem(hWndDlg, IDC_MONTHCALENDAR);  
  
    switch(Msg)  
    {  
    case WM_INITDIALOG:  
        SendMessage(hWndMonthCal, MCM_SETCOLOR, MCSC_TITLEBK,  
RGB(205, 50, 5));  
        MonthCal_SetColor(hWndMonthCal, MCSC_TITLETEXT, RGB(255,  
250, 5));  
        MonthCal_SetColor(hWndMonthCal, MCSC_TEXT, RGB(128, 0,  
5));  
  
        return TRUE;  
  
    case WM_COMMAND:  
        switch(wParam)  
        {  
        case IDCANCEL:  
            EndDialog(hWndDlg, 0);  
            return TRUE;  
        }  
        break;  
    }  
  
    return FALSE;  
}  
//-----  
-----
```

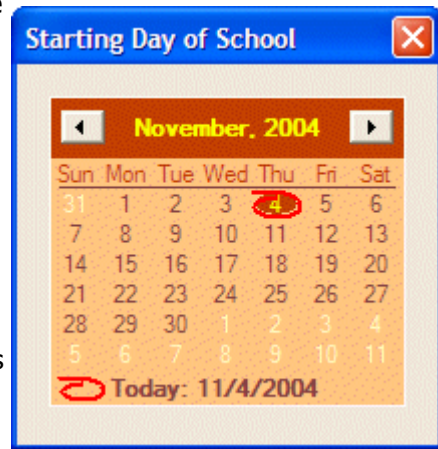
2. Test the application

## **Colors: Days of the Trailing Months**

---



The numbers of the days of the month display in two colors. The real days of the selected month display, by default, in a black color as the WindowText system color. To change the color of the days of the other months, you can call the **MonthCal\_SetColor()** macro, pass the *iColor* argument as **MCSC\_TRAILINGTEXT**, and pass the color of your choice as the *clr* argument. Alternatively, you can send the **MCM\_SETCOLOR** message



## Colors: The Body Background

Under the line separator, the numeric days of the month are listed. By default, the numeric days of the control display above a white background which is the Window system color. To change the background color of the area where the numeric days appear, you can call the **MonthCal\_SetColor()** macro or send a **MCM\_SETCOLOR** message, pass the *iColor* or the *wParam* argument as **MCSC\_MONTHBK**, and pass the color of your choice as the *Clr* or the *lParam* argument.

## Today's Effect

The Month Calendar control is used to let the user know today's date in two ways. On the calendar, today's date is circled by a hand-drawn ellipse. In the bottom section of the calendar, today's date is also displayed as a sentence:



To programmatically hide the today label, you can apply the **MCS\_NOTODAY** style. Here is an example:

```
#include <commctrl.h>
#include "resource.h"

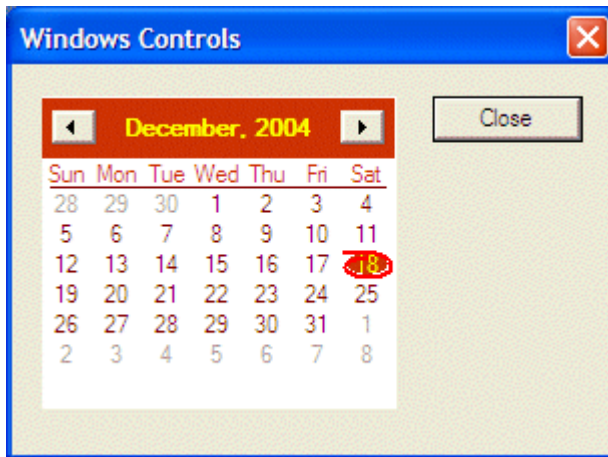
IDD_CONTROLSDLG DIALOG 260, 200, 200, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
```



```

CONTROL          "", IDC_MONTHCALENDAR, "SysMonthCal32",
                WS_CHILD | WS_TABSTOP | MCS_NOTODAY |
MCS_NOTODAY, 10, 10, 118, 96
                DEFPUSHBUTTON "Close", IDCANCEL, 140, 10, 50, 14
END

```



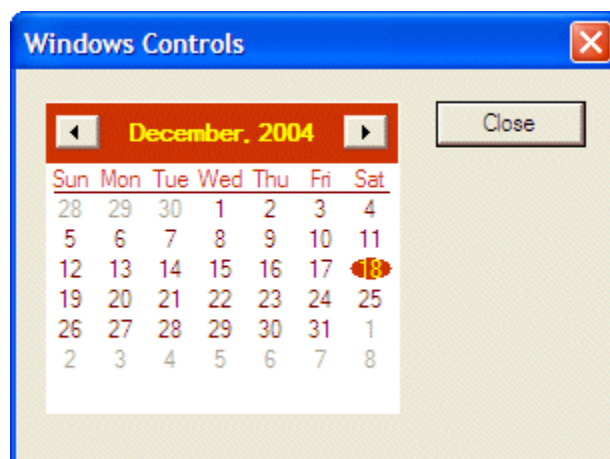
We also mentioned that today date appears with a circle. If you want to hide just the circle, apply the **MCS\_NOTODAYCIRCLE** style:

```

#include <commctrl.h>
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 200, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    CONTROL          "", IDC_MONTHCALENDAR, "SysMonthCal32",
                    WS_CHILD | WS_TABSTOP | MCS_NOTODAY |
                    MCS_NOTODAY | MCS_NOTODAYCIRCLE, 10, 10, 118, 96
    DEFPUSHBUTTON    "Close", IDCANCEL, 140, 10, 50, 14
END

```



As you may know already, to programmatically change the style of a control, for example to show or hide the today label, you can call the **SetWindowLong()** function.

## The First Day of the Week

---

Under the title bar, the short names of week days display, using the format set in Control Panel. In US English, the first day is Sunday. If you want to start with a different day, you can call the **MonthCal\_SetFirstDayOfWeek()** macro. Its syntax is:

```
DWORD MonthCal_SetFirstDayOfWeek(HWND hwndMC, INT iDay);
```

Alternatively, you can send the **MCM\_SETFIRSTDAYOFWEEK** message using the following syntax:

```
LRESULT = SendMessage(HWND hWndControl, MCM_SETFIRSTDAYOFWEEK, WPARAM wParam, LPARAM lParam);
```

The first argument is the handle to the Month Calendar control whose first day you want to set.

The *wParam* argument is not used and can be passed as 0.

The *iDay* or the *lParam* argument is a constant integer of the following values:

Value	Weekday
0	Monday
1	Tuesday
2	Wednesday
3	Thursday
4	Friday
5	Saturday
6	Sunday

If the calendar is already functioning, to find what its first day of the week is, you can call the **MonthCal\_GetFirstDayOfWeek()** macro. Its syntax is:

```
DWORD MonthCal_GetFirstDayOfWeek(HWND hwndMC);
```

To get the same information, you can send an **MCM\_GETFIRSTDAYOFWEEK** message. Neither the *wParam* nor the *lParam* argument is used. Therefore, they must be passed as 0.

Both the **MonthCal\_GetFirstDayOfWeek()** macro and the **SendMessage()** function that carries the **MCM\_GETFIRSTDAYOFWEEK** message returns a value of the above table.

## The Selected Day

---

At any time, a particular date is selected and has an ellipse with the same color as the background of the title bar. By default, the selected date is today's date. When the user clicks the calendar, a date is selected. You also can programmatically select a day. To do this, you can call the **MonthCal\_SetCurSel()** macro whose syntax is:

```
BOOL MonthCal_SetCurSel(HWND hwndMC, LPSYSTEMTIME lpSysTime);
```

Alternatively, you can send a **MCM\_SETCURSEL** message. The syntax used is:

```
LRESULT = SendMessage(HWND hWndControl, MCM_SETCURSEL, WPARAM wParam, LPARAM lParam);
```

The *wParam* argument of the **SendMessage()** function is not used and can be passed as 0.

The *lpSysTime* and the *lParam* arguments are passed as pointers to the Win32's **SYSTEMTIME** structure and therefore carry the returning value of the function.

After a day has been selected, whether by you or the user, to find out what day was selected, you can call the **MonthCal\_GetCurSel()** macro whose syntax is:

```
BOOL MonthCal_GetCurSel(HWND hwndMC, LPSYSTEMTIME lpSysTime);
```

To get the same information, you can send a **MCM\_GETCURSEL** message using the following syntax:

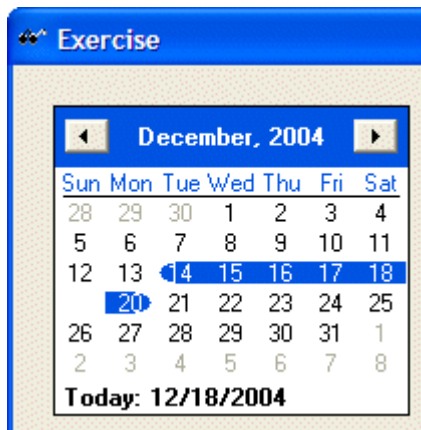
```
lResult = SendMessage(HWND hWndControl, MCM_GETCURSEL, WPARAM wParam, LPARAM lParam);
```

## The Multiple Day Selection Option

When the user clicks the Month Calendar control, one date is selected. To control whether the user can select one or more dates, you can apply the **MCS\_MULTISELECT** style:

```
#include <commctrl.h>
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 200, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    CONTROL        "", IDC_MONTHCALENDAR, "SysMonthCal32",
                  WS_CHILD | WS_TABSTOP | MCS_NOTODAY |
                  MCS_NOTODAY | MCS_NOTODAYCIRCLE | MCS_MULTISELECT,
                  10, 10, 118, 96
    DEFPUSHBUTTON  "Close", IDCANCEL, 140, 10, 50, 14
END
```



With this property set, the user can select many days in the calendar. You also can programmatically select a range of days. To do this, you can call the **MonthCal\_SetSelRange()** macro whose syntax is:

```
BOOL MonthCal_SetSelRange(HWND hwndMC, LPSYSTEMTIME lprgSysTimeArray);
```

To perform the same operation, you can send a **MCM\_SETSELRANGE** message using the following syntax:

```
lResult = SendMessage(HWND hWndControl, MCM_SETSELRANGE, WPARAM wParam, LPARAM lParam);
```

After the user has selected a range date on the calendar, to find out what that range is, you can call the **MonthCal\_GetSelRange()** macro.

As mentioned already, to change the month and subsequently the year of the calendar, the user can click the buttons continuously. To control the allowable dates the user can navigate from and to, you can call the **MonthCal\_SetRange()** macro.

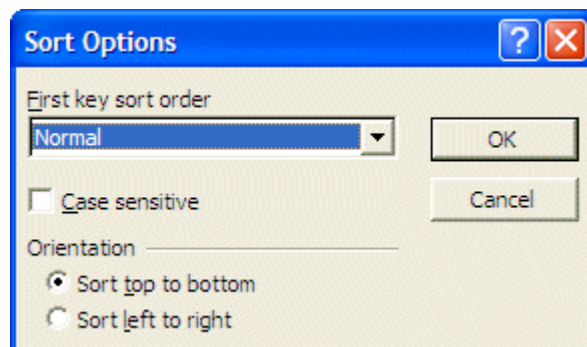
## Windows Controls: Combo Boxes

### Introduction to Combo Boxes

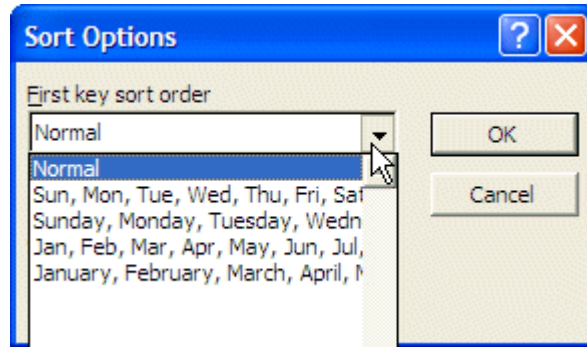
#### Overview

A combo box is a Windows control made of two sections. There are two main types of combo boxes: drop down and simple. Each is made of two sections.

The most commonly used combo box is called drop down. On the left side, it is made of an edit box. On the right side, it is equipped with a down-pointing arrow:

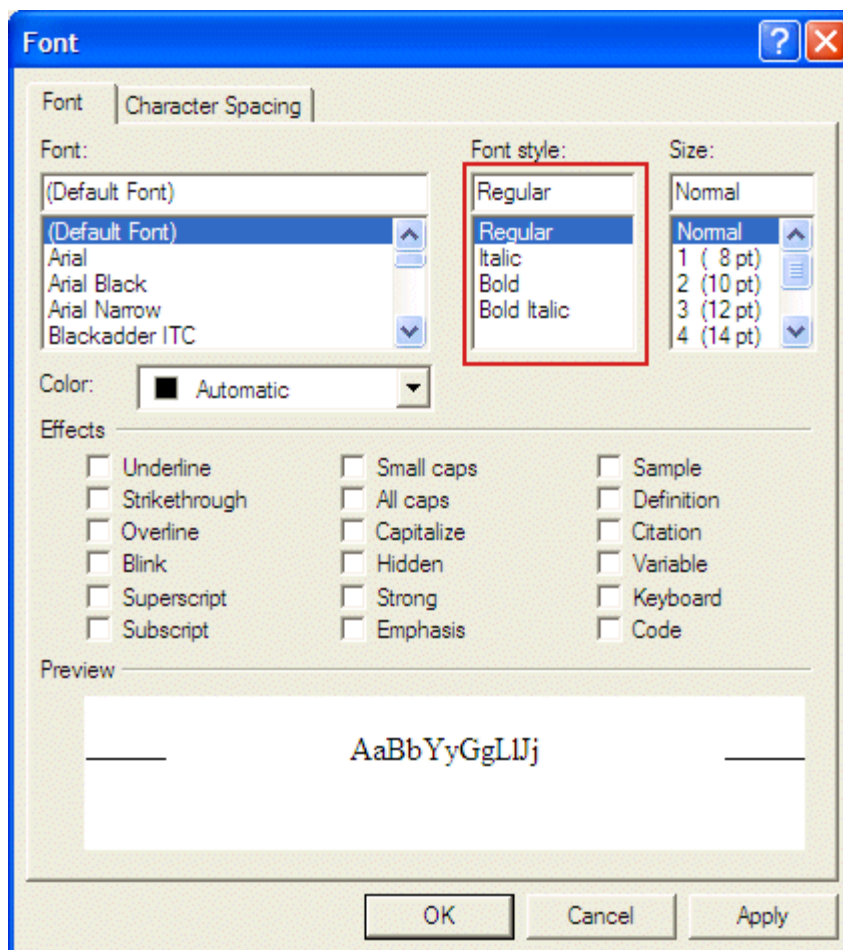


To use it, the user must click the arrow. This opens a list:



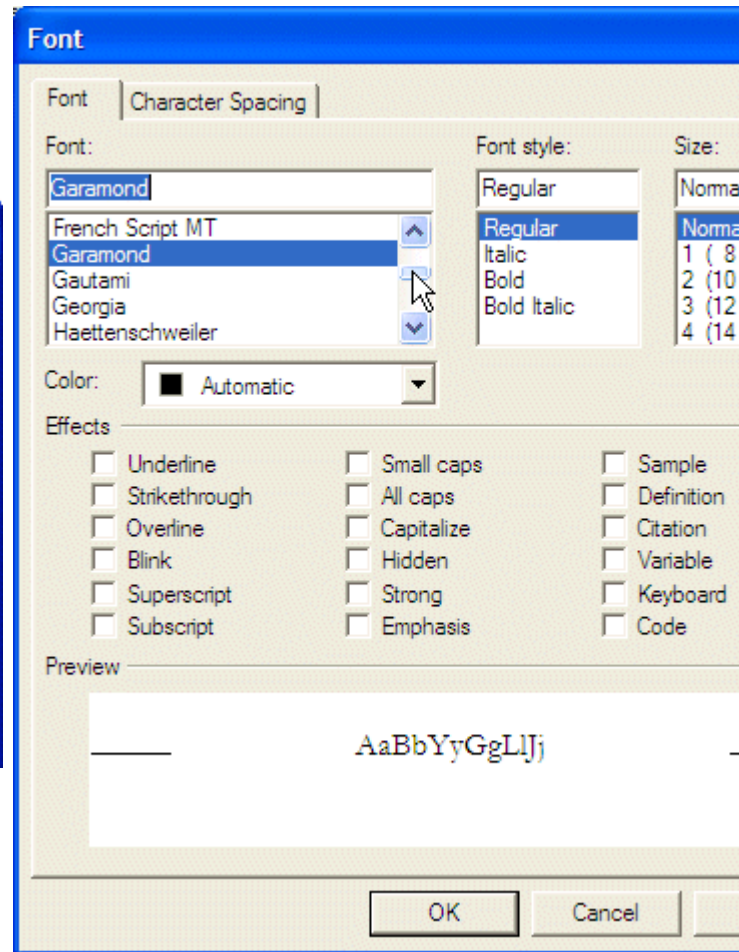
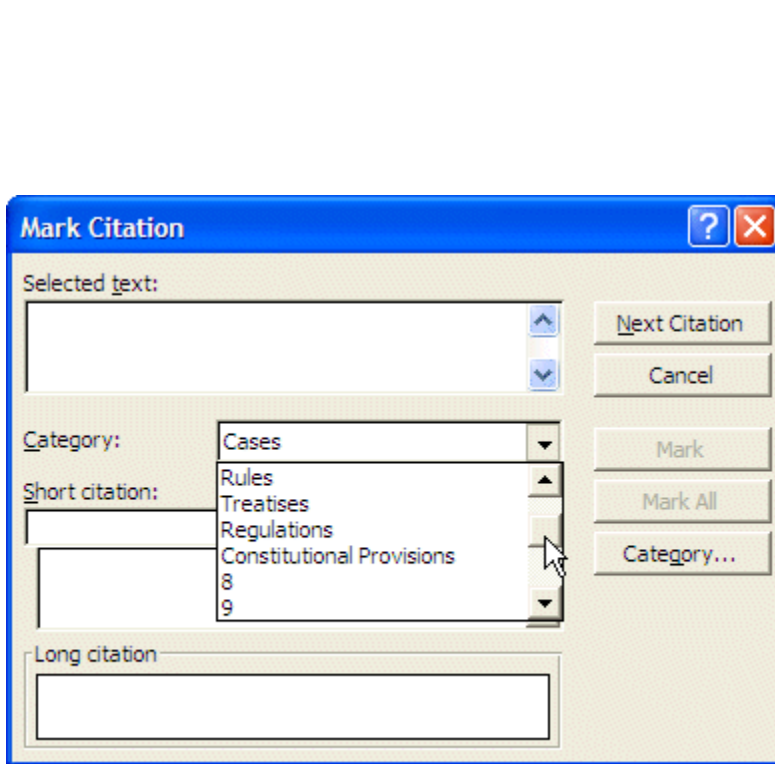
After locating the desired item in the list, the user can click it. The item clicked becomes the new one displaying in the edit part of the control. If the user doesn't find the desired item in the list, he or she can click the down-pointing arrow or press Esc. This hides the list and the control displays as before. The user can also display the list by giving focus to the control and then pressing Alt + down arrow key.

The second general type of combo box is referred to as simple. This type is also made of two sections but, instead of a down-pointing arrow used to display the list, it shows its list all the time:



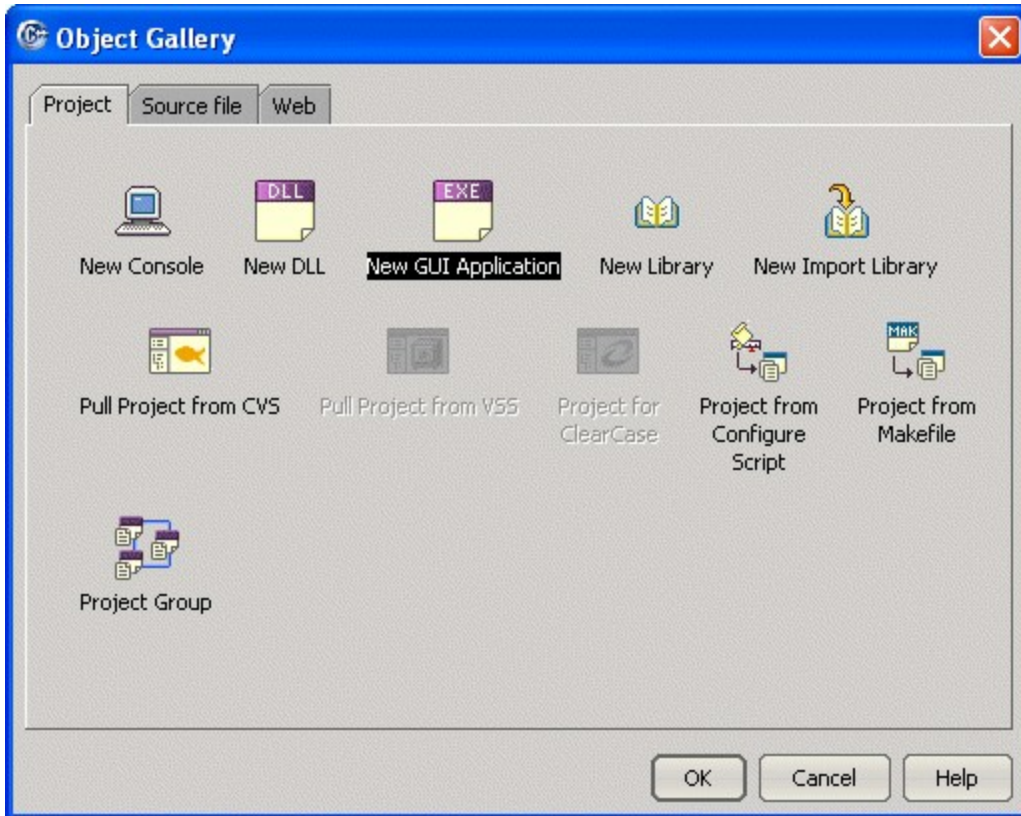
This time, to select an item, the user can simply locate it in the list and click it.

In both types of combo boxes, if the list is too long for the allocated space, when it displays, the list part is equipped with a vertical scroll bar. This allows the user to navigate up and down in the list to locate the desired item:

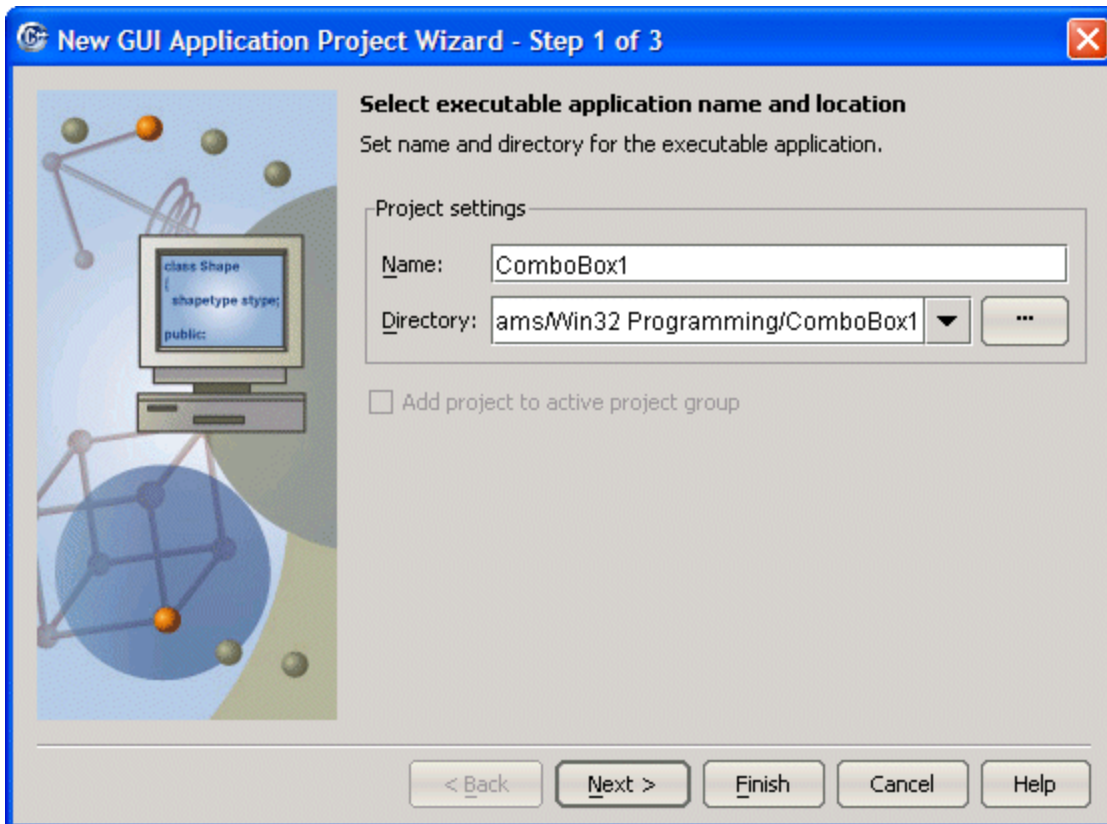


## ◆ Practical Learning: Creating the Application

1. Because Borland **C++BuilderX** is free, we are going to use it. Start Borland C++Builder X and, on the main menu, click File -> New...

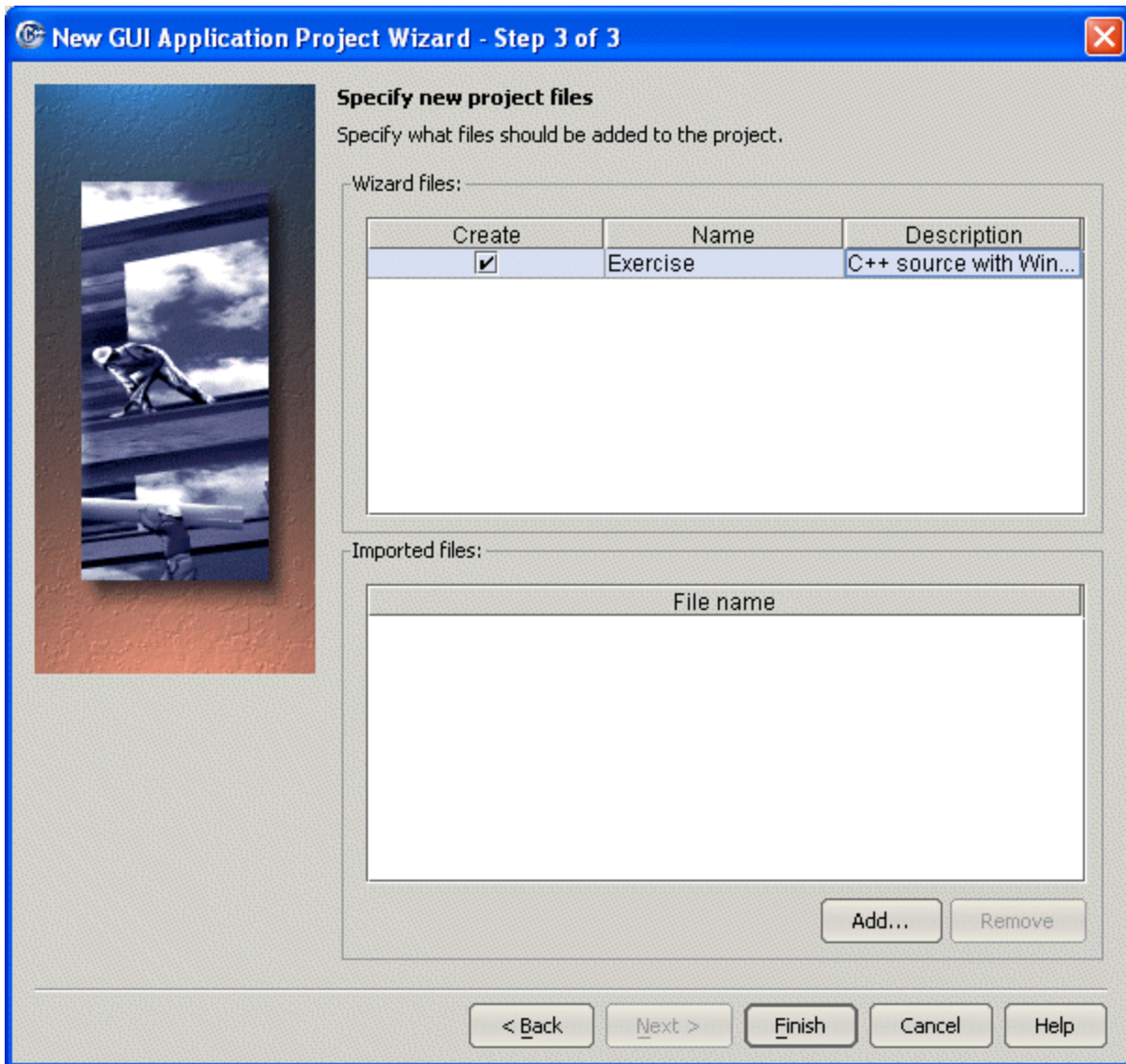


2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the New GUI Application Project Wizard - Step 1 of 3, in the Directory edit box of the Project Settings section, type the path you want. Otherwise, type **C:\Programs\Win32 Programming**
4. In the Name edit box, type **ComboBox1**

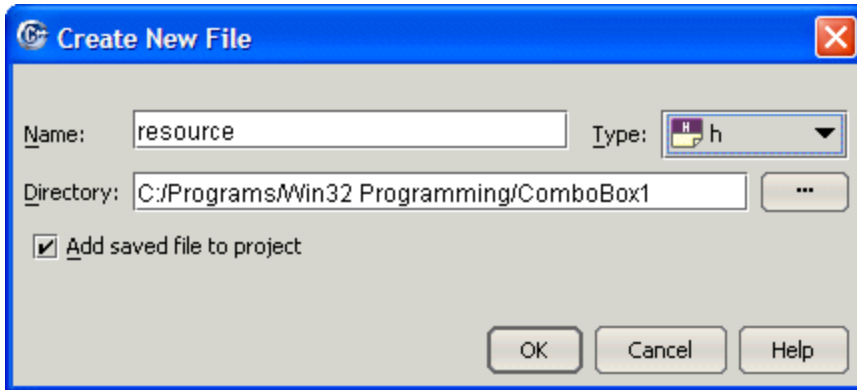


5. Click Next
6. In the New GUI Application Project Wizard - Step 2 of 3, accept the defaults and click Next
7. In the New GUI Application Project Wizard - Step 3 of 3, click the check box under **Create**
8. Select **Untitled** under the Name column header. Type **Exercise** to replace the name and press Tab





9. Click Finish
10. To create the resource header file, on the main menu, click File -> New File...
11. In the Create New File dialog box, change the contents of the Name edit box with **resource**
12. In the Type combo box, select h



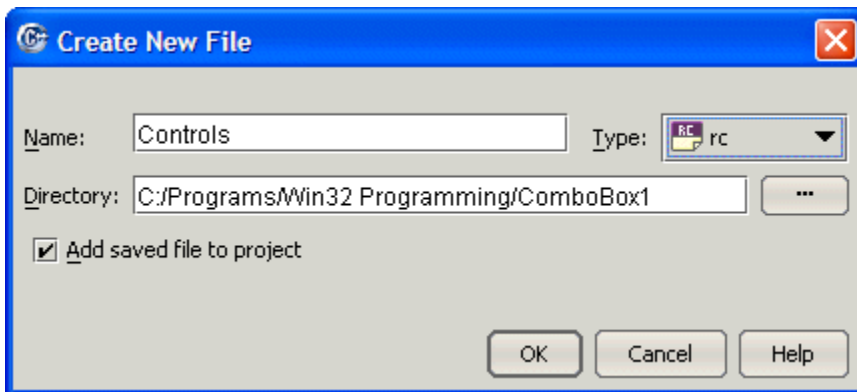
13. Click OK

14. In the file, type **#define IDD\_CONTROLSDLG 101**

15. To create the rc resource file, on the main menu of C++BuilderX, click File -> New File...

16. In the Create New File dialog box, change the contents of the Name edit box to **Controls**

17. In the Type combo box, select **rc**



18. Click OK

19. In the empty file, type the following (the referenced header file will be created next):

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 180, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON "Close", IDCANCEL, 120, 100, 50, 14
END
```

20.

In the left frame, double-click **Exercise.cpp** and change the file to the following:

```

#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"
//-----
HWND hWnd;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
 lParam);
//-----
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    DialogBox(hInstance, MAKEINTRESOURCE(IDD_CONTROLSDLG),
             hWnd, reinterpret_cast<DLGPROC>(DlgProc));

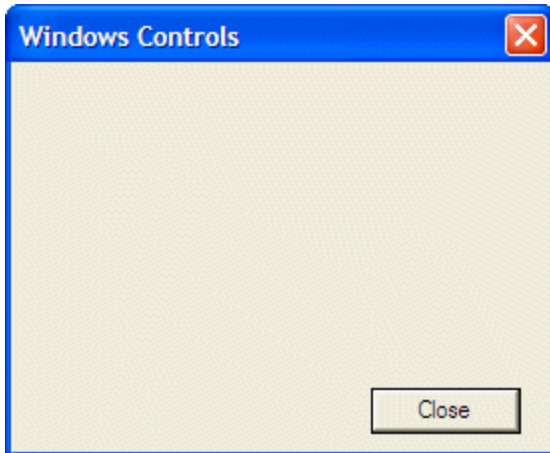
    return 0;
}
//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,
LPARAM lParam)
{
    switch(Msg)
    {
        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:
            switch(wParam)
            {
                case IDCANCEL:
                    EndDialog(hWndDlg, 0);
                    return TRUE;
            }
            break;

        return FALSE;
    }
}
//-----

```

21. Press F9 to test the application



22. Click Close to dismiss the dialog box

## Creating a Combo Box

---

There are two main ways you can create a combo box. You can write code or use a script. To create a combo box with code, you can first create a Windows class that defines an **HWND** handle and implements the assignments of a combo box.

The easiest way to create a combo box is through a resource script. The syntax used to create the control in a script is:

```
COMBOBOX id, x, y, width, height [, style [, extended-style]]
```

You must specify **COMBOBOX** as the class of this control

The *id* is the number used to identify the control in a resource header file

The *x* measure is its horizontal location with regards to the control's origin, which is located in the top left corner of the window that is hosting the combo box

The *y* factor is the distance from control's origin, which is located in the top left corner of the window that is hosting the combo box, to the top-left side of the combo box

The *width* and the *height* specify the dimensions of the combo box

The optional *style* and the *extended-style* factors are used to configure and prepare the behavior of the combo box.

## ◆ Practical Learning: Creating a Combo Box

---

1. To create an identifier for the combo box, open the resource header file and modify it as follows:

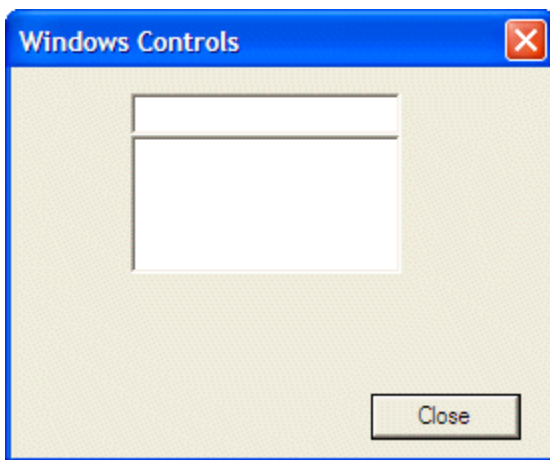
```
2.     #define IDD_CONTROLSDLG 101
       #define IDD_SIZE_CBO   102
```

To create the combo box, open the resource script and change it as follows:

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 180, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON    "Close", IDCANCEL, 120, 100, 50, 14
    COMBOBOX         IDD_SIZE_CBO, 40, 8, 90, 80
END
```

3. Test the application



4. Click Close to dismiss the dialog box

## Characteristics of a Combo Box

---

### Windows Styles of a Combo Box

---

Like all the other windows, to create a combo box programmatically, you can call the **CreateWindow()** or the **CreateWindowEx()** function. The syntax used is:

```
HWND CreateWindow( "COMBOBOX",           HWND CreateWindowEx( Extended Style,
                    "Default String",     "COMBOBOX",
                    style,                "Default String",
                    x,                    style,
                    Y,                    x,
                    width,                Y,
                    height,               width,
                    parent,               height,
                    menu,                 parent,
                    instance,             menu,
                    Optional Parameter    instance,
                                           Optional Parameter
);                                       );
```

The first argument of the **CreateWindow()** or the second argument of the **CreateWindowEx()** functions must be **COMBOBOX** passed as a string.

The second argument of the **CreateWindow()** or the third argument of the **CreateWindowEx()** functions specifies a string that would display in the edit part of the combo box when the control appears. If the control is created with certain styles we will review here, this string would not appear even if you specify it. You can also omit it and pass the argument as **NULL** or "" since there are other ways you can set the default string.

Like every other Windows control, a combo box' appearance and behavior are controlled by a set of properties called styles . The primary properties of a combo box are those controlled by the operating system and shared by all controls. You can use them to set the visibility, availability, and parenthood, etc, of the combo box. If you create a combo box using a resource script, since you would include it in a **DIALOG** section of the script, the dialog box is automatically made its parent. Otherwise, to specify that the combo box is hosted by another control, get the handle of the host and pass it as the *parent* parameter. You must also set or add the **WS\_CHILD** bit value to the *style* parameter. If you want the combo box to appear when its parent comes up, add the **WS\_VISIBLE** style using the bitwise | operator.

If you want the combo box to receive focus as a result of the user pressing the Tab key, add the **WS\_TABSTOP** style.

The location of a combo box is specified by the *x* and *y* parameters whose values are based on the origin, located in the top-left corner of the dialog box or the window that is hosting the combo box.

## ◆ Practical Learning: Programmatically Creating a Combo Box

---

1. To programmatically create a combo box, modify the Exercise.cpp file as follows:

```

#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"
//-----
HWND hWnd;
HWND hWndComboBox;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    hInst = hInstance;

    DialogBox(hInstance, MAKEINTRESOURCE(IDD_CONTROLSDLG),
        hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return 0;
}
//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,
LPARAM lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        hWndComboBox = CreateWindow("COMBOBOX",
            NULL,
            WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
            60, 62, 136, 60,
            hWndDlg,
            NULL,
            hInst,
            NULL);

        if( !hWndComboBox )
        {
            MessageBox(hWndDlg,
                "Could not create the combo box",
                "Failed Control Creation",
                MB_OK);
            return FALSE;
        }
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDCANCEL:

```

## Categories of a Combo Box

---

As mentioned already, there are two big categories of combo boxes: simple and drop down. The category is specified by the style of the control. By default, that is, if you don't specify a category, the combo box is created as simple, as you can see on the above screen shot. Otherwise, the style of the simple combo box is **CBS\_SIMPLE**. To create a combo box that display a down-pointing arrow and displays its list only when requested, add the **CBS\_DROPDOWN** instead. Here is an example:

```
CreateWindow("COMBOBOX",
            NULL,
            WS_CHILD | WS_VISIBLE | WS_TABSTOP | CBS_DROPDOWN,
            60, 62, 136, 60,
            hWndDlg,
            NULL,
            hInst,
            NULL);
```

Don't use both styles on the same combo box.

A combo box as a Windows control presents many other styles. Most of these styles are related to operations performed on the control. For this reason, we will review them when the relations operations are addressed.

### ◆ Practical Learning: Using Combo Box Styles

---

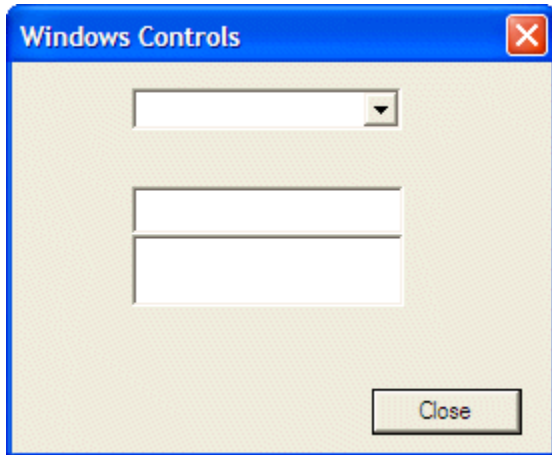
1. To make the combo box drop down, open the resource script and change it as follows:

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 180, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON    "Close", IDCANCEL, 120, 100, 50, 14
    COMBOBOX         IDD_SIZE_CBO, 40, 8, 90, 60, WS_TABSTOP |
CBS_DROPDOWN
END
```



2. Test the application



3. Close it and return to your programming environment

## Operations on Combo Boxes

---

### Creation-Related Messages

---

When creating the combo boxes above, we specified their location through *x* and *y* followed by their dimensions through the width and the height. If a combo box has already been created and you want to get its coordinates, you can send the **CB\_GETDROPPEDCONTROLRECT** message in the **SendMessage()** function. The third argument, *wParam*, is not used. The fourth argument, *lParam*, carries a **RECT** pointer that will return the coordinates of the combo box.

### Adding Items to the List

---

After creating a combo box, the first operation that probably comes in mind is to fill it with items the user would select from. On this issue, there are two types of lists used on combo boxes: regular or owner-draw.

A regular list of a combo box displays a normal list of strings. This is the simplest. To create a string to add to the list, you can call the **SendMessage()** function passing the second argument as **CB\_ADDSTRING**. The syntax used is:

```
lResult = SendMessage( (HWND) hWndControl,  
                      (UINT) CB_ADDSTRING,  
                      (WPARAM) wParam,  
                      (LPARAM) lParam );
```

The third argument is not used. The fourth argument is the string that will be added to the control. Here are examples:

```
//-----  
---  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{
```

```

const char *ComboBoxItems[] = { "Sri Lanka", "El Salvador", "Botswana",
                                "France", "Cuba" };

switch(Msg)
{
case WM_INITDIALOG:
    hWndComboBox = CreateWindow("COMBOBOX",
                                NULL,
                                WS_CHILD | WS_VISIBLE,
                                60, 62, 136, 60,
                                hWndDlg,
                                NULL,
                                hInst,
                                NULL);

    if( !hWndComboBox )
    {
        MessageBox(hWndDlg,
                    "Could not create the combo box",
                    "Failed Control Creation",
                    MB_OK);
        return FALSE;
    }

    SendMessage(hWndComboBox,
                 CB_ADDSTRING,
                 0,
                 reinterpret_cast<LPARAM>( (LPCTSTR) ComboBoxItems [0]
    ));

    SendMessage(hWndComboBox,
                 CB_ADDSTRING,
                 0,
                 reinterpret_cast<LPARAM>( (LPCTSTR) ComboBoxItems [1]
    ));

    SendMessage(hWndComboBox,
                 CB_ADDSTRING,
                 0,
                 reinterpret_cast<LPARAM>( (LPCTSTR) ComboBoxItems [2]
    ));

    SendMessage(hWndComboBox,
                 CB_ADDSTRING,
                 0,
                 reinterpret_cast<LPARAM>( (LPCTSTR) ComboBoxItems [3]
    ));

    SendMessage(hWndComboBox,
                 CB_ADDSTRING,
                 0,
                 reinterpret_cast<LPARAM>( (LPCTSTR) ComboBoxItems [4]
    ));

    return TRUE;

case WM_COMMAND:
    switch(wParam)
    {
    case IDCANCEL:
        EndDialog(hWndDlg, 0);

```

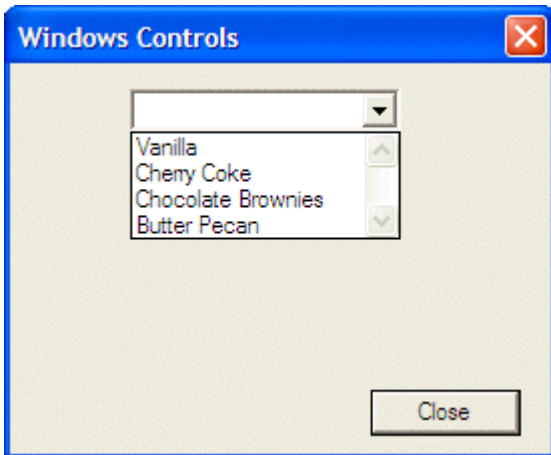
```

        return TRUE;
    }
    break;
}

return FALSE;
}
//-----
---
```

If the adding operation succeeds, the **SendMessage()** function returns the index of the item that was added. If it fails, it returns the **CB\_ERR** error.

When a list has been created, if it contains more items than the allocated space can display, you should provide a way for the user to navigate entirely in the list. This is usually done with a vertical scroll bar. To equip a combo box with a vertical scroll bar, add the **WS\_VSCROLL** value to its list of styles. If you add this style but the list is not too long, the scroll bar would not display. If you insist on displaying a vertical scroll bar even if the list is not too long, add the **CBS\_DISABLENOSCROLL** style. If you do this and if the list is short, it would appear with a disabled vertical scroll bar:



The **CB\_ADDSTRING** message we use above allows us to add a string to the list. You can use the same message to let the user either create a list or add an item to it.

## ◆ Practical Learning: Creating the List

---

1. Change the resource.h header file as follows:

```
#define IDD_CONTROLSDLG 101
#define IDD_COUNTRIES_CBO 102
```

2. To add a vertical scroll bar to the first combo box, Add **WS\_VSCROLL** to its style list in the resource script as follows:

```
#include "resource.h"

IDD_CONTROLSDLG DIALOG 260, 200, 180, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
```

```

FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON    "Close", IDCANCEL, 120, 100, 50, 14
    LTEXT            "Country:", IDC_STATIC, 10, 10, 25, 8
    COMBOBOX        IDD_COUNTRIES_CBO, 40, 8, 90, 60,
                    WS_TABSTOP | WS_VSCROLL | CBS_DROPDOWN
END

```

3. To create the list of items for the combo box, modify the Exercise.cpp file as follows:

```

#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"
//-----
HWND hWnd;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
//-----

int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    hInst = hInstance;

    DialogBox(hInstance, MAKEINTRESOURCE(IDD_CONTROLSDLG),
              hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return 0;
}
//-----

LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam,
LPARAM lParam)
{
    HWND cboCountries;

    const char *Countries[] = { "Sri Lanka", "El Salvador",
                                "Botswana",
                                "France", "Cuba", "South Africa",
                                "Australia", "Russia", "Jamaica",
                                "Great Britain", "Senegal", "Bangla
Desh" };

    switch(Msg)
    {
    case WM_INITDIALOG:
        cboCountries = GetDlgItem(hWndDlg, IDD_COUNTRIES_CBO);

        for(int Count = 0; Count < 12; Count++)
        {
            SendMessage(cboCountries,

```

```

        CB_ADDSTRING,
        0,
        reinterpret_cast<LPARAM>((LPCTSTR) Countries [C
ount]));
    }

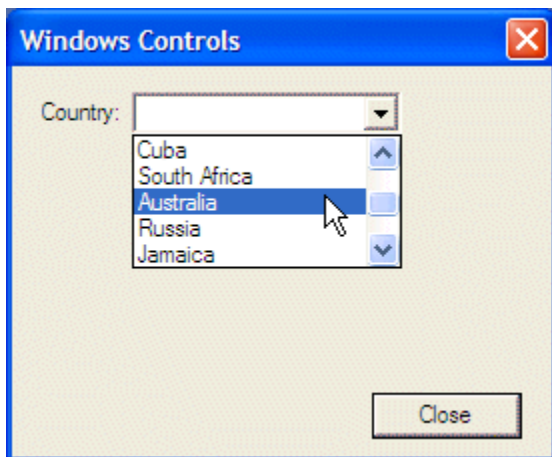
    return TRUE;

case WM_COMMAND:
    switch(wParam)
    {
    case IDCANCEL:
        EndDialog(hWndDlg, 0);
        return TRUE;
    }
    break;
}

return FALSE;
}
//-----
-----

```

#### 4. Test the application



#### 5. Return to your programming environment

## Sorting Items in the List

---

As the items are added to the list of a combo box, they create a C++ type of array indexed so that the first item of the list has a numeric position of 0, the second is 1, etc. This index will allow you to perform various operations on items by locating the desired one based on its index.

The items of a combo box are cumulatively added to the list, that is, in order. If you want to arrange them in alphabetical order, when creating the combo box, add the **CBS\_SORT** style. When this style is set, items are automatically and appropriately inserted in the order based on the language of the computer as set in Control Panel. Therefore, every time you add an item to the list using the **CB\_ADDSTRING** message, the list is sorted to rearrange it.

## Removing Items from the List

---

As opposed to adding an item, if the list already contains strings and you want to remove an item from the list, call the **SendMessage()** function, passing the message type as **CB\_DELETESTRING** and the index of the undesired item as the *lParam* value.

Here is an example that deletes the fourth item of the list:

```
SendMessage(cboCountries, CB_DELETESTRING, 3, 0);
```

If you want to remove all items from the combo box, call the **SendMessage()** function with the **CB\_RESETCONTENT** message. The two accompanying arguments, *wParam* and *lParam*, are not used.

## Selecting an Item

---

Once the list of a combo box has been created, the user can click an item in the list to select it. To select an item in a drop down combo box, the user can click the down-pointing arrow or press Alt + down arrow key. Any of these actions causes the list of a drop down combo box to display. You also, as the programmer, can display the list programmatically any time, even if the user clicks another control. To display the list, you can send a **CB\_SHOWDROPDOWN** message in the **SendMessage()** function. The *wParam* argument carries a **TRUE** value if you want to display the list. The *lParam* argument is not used. Here is an example:

```
SendMessage(cboCountries, CB_SHOWDROPDOWN, TRUE, 0);
```

An item that is selected in the list is commonly referred to as the current selection. You also as the programmer can select an item at any time. For example, after creating the list, you may want to specify a default item to be shown in the edit part of the control.

To select an item from the list, you can send the **CB\_SETCURSEL** message in the **SendMessage()** function. As the items are 0-based, to select a particular item, pass its index as the *wParam* parameter. The *lParam* argument is not used.

Here is an example that selects the fourth item in the list:

```
SendMessage(cboCountries, CB_SETCURSEL, 3, 0);
```

An alternative to selecting a string in the combo box is to send a **WM\_SETTEXT** message to the combo box using the **SendMessage()** function. In this case, the *wParam* argument is not used. The *lParam* argument carries the string to select. Here is an example:

```
SendMessage(cboCountries, WM_SETTEXT, 0, (LPARAM)Countries[5]);
```

So far, we have mentioned that there were two categories of combo boxes: simple and drop down. We also saw that the user could click the arrow of a drop down style to display the list. The drop down version has another characteristic: it allows the user to click the edit part of the control and start typing. This can let user find a match of a string based on the typed letters. In some cases, you may want the user to only be able to select an item in the list without typing it. In this case, the combo box provides the **CBS\_DROPDOWNLIST** style. When this style has been added, the edit part of the control is disabled or eliminated; it becomes only a place holder for a selected item.

## The Width of a Combo Box

---

When creating our combo boxes, we learned to specify their width. After a combo box has been created, if you want to find its width, you can call the **SendMessage()** function with the **CB\_GETDROPPEDWIDTH** message. Both the *wParam* and the *lParam* arguments are not used. If the **SendMessage()** function succeeds in its request, it returns the current width of the combo box.

Sometimes after adding an item to the list, you may find out that it is long than the width of the combo box can display. Although there are various ways you can resize a control, the combo box provides its own and (very) easy mechanism to change its width. This can be done by sending the **CB\_SETDROPPEDWIDTH** message through the **SendMessage()** function. The *wParam* argument is used to specify the new desired width. The *lParam* argument is not used.

```
SendMessage(cboCountries, CB_SETDROPPEDWIDTH, 240, 0);
```

## Getting the Selected Item

---

At any time, to find out what item has been selected or what item is displaying in the edit part of the control, you can call the **SendMessage()** function with the **CB\_GETCURSEL** message. When the function is called with this message, it checks the combo box first. If an item is currently selected, it returns it. If no item is selected, this function returns **CB\_ERR**.

An alternative to getting the string that is selected in the combo box is to send the **WM\_GETTEXT** to the combo box using the **SendMessage()** function. The *wParam* is the length of the string. The *lParam* argument is the variable into which the string will be copied.

## Combo Box Events

---

### List Display Events

---

When a combo box is created as drop down, in order to select an item from the list, the user must display the list. This is done either by clicking the down-pointing arrow or by pressing Alt and the down arrow key (keyboard). However this is done, when the user decides to display the list, just before the list is displayed, the combo box sends a **CBN\_DROPDOWN** message to the dialog box or the object that is hosting the combo box. You can do some last minute processing before the list is displayed. For example, you can intercept this message and prevent the list from displaying, or you can update it, anything.

Since this message comes from a child control, the dialog box processes it as a **WM\_COMMAND** message. The combo box that sends this message can be identified using the low word of the *wParam* argument of the procedure. In the following example, the message is intercepted and it is used to let the user know before playing the list:

```
//-----  
---  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM  
lParam)  
{  
    HWND cboCountries;  
  
    const char *Countries[] = { "Sri Lanka", "El Salvador", "Botswana",
```

```

        "France", "Cuba", "South Africa",
        "Australia", "Russia", "Jamaica",
        "Great Britain", "Senegal", "Bangla
Desh" };

switch(Msg)
{
case WM_INITDIALOG:
    cboCountries = GetDlgItem(hWndDlg, IDD_COUNTRIES_CBO);

    for(int Count = 0; Count < 12; Count++)
    {
        SendMessage(cboCountries,
                    CB_ADDSTRING,
                    0,
                    reinterpret_cast<LPARAM>((LPCTSTR)Countries[Count]
));
    }

    SendMessage(cboCountries, CB_SETCURSEL, 3, 0);
    break;

case WM_COMMAND: // Windows Controls processing
    switch(LOWORD(wParam)) // This switch identifies the control
    {
    case IDD_COUNTRIES_CBO: // If the combo box sent the message,
        switch(HIWORD(wParam)) // Find out what message it was
        {
        case CBN_DROPDOWN: // This means that the list is about to
display
                MessageBox(hWndDlg, "A request to display the list has
been made",
                        "Display Notification", MB_OK);
                break;
        }
        break;
    case IDCANCEL:
        EndDialog(hWndDlg, 0);
        return TRUE;
    }
    break;
}

return FALSE;
}
//-----
---
```

We have mentioned that, as opposed to the user displaying the list, you also can display it by sending the **CB\_SHOWDROPDOWN** message. At any time, you can find out whether the list is currently displaying by sending the **CB\_GETDROPPEDSTATE** message to the combo box.

After the user has finished using the list of a combo box, he or she can close it. This can be done by selecting an item, clicking the down-pointing arrow, or pressing Esc. In all cases, this action closes or hides the list and then the combo box sends a **CBN\_CLOSEUP** message to its parent. Here is an example:



```

//-----
---
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    HWND cboCountries;

    const char *Countries[] = { "Sri Lanka", "El Salvador", "Botswana",
                                "France", "Cuba", "South Africa",
                                "Australia", "Russia", "Jamaica",
                                "Great Britain", "Senegal", "Bangla
Desh" };

    switch(Msg)
    {
    case WM_INITDIALOG:
        cboCountries = GetDlgItem(hWndDlg, IDD_COUNTRIES_CBO);

        for(int Count = 0; Count < 12; Count++)
        {
            SendMessage(cboCountries,
                        CB_ADDSTRING,
                        0,
                        reinterpret_cast<LPARAM>((LPCTSTR)Countries[Count]
));
        }

        SendMessage(cboCountries, CB_SETCURSEL, 3, 0);
        break;

    case WM_COMMAND: // Windows Controls processing
        switch(LOWORD(wParam)) // This switch identifies the control
        {
        case IDD_COUNTRIES_CBO: // If the combo box sent the message,
            switch(HIWORD(wParam)) // Find out what message it was
            {
            case CBN_DROPDOWN: // This means that the list is about to
display
                MessageBox(hWndDlg, "A request to display the list has
been made",
                            "Display Notification", MB_OK);
                break;
            case CBN_CLOSEUP:
                MessageBox(hWndDlg, "The list will be closed",
                            "List Close Notification", MB_OK);
                break;
            }
        }
        break;
    case IDCANCEL:
        EndDialog(hWndDlg, 0);
        return TRUE;
    }
    break;
}

return FALSE;
}

```

//-----  
---

## Selection-Related Events

After the user has selected an item from the list, the combo box sends a **CBN\_SELCHANGE** message to its parent and then the list closes. You can either this event or the **CBN\_CLOSEUP** related event to find out what the user selected, if any.

If the user displays the list and clicks an item, the list is retracted. If the selection was successful (sometimes something could go wrong), the combo box sends a **CBN\_SELENDOK** message. On the other hand, if the user displays the list but doesn't select an item and then either clicks the down-pointing arrow, clicks somewhere else, or presses Esc, the combo box lets its parent know that nothing was selected. To do this, it sends a **CBN\_SELENDUNCANCEL** message.

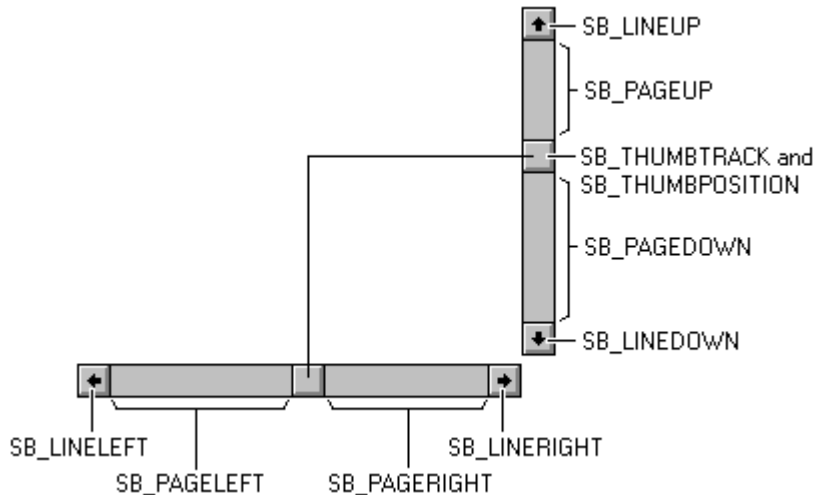
# Windows Controls: Scroll Bars

## Introduction to Scroll Bars

### Overview



A scroll bar is an object that allows the user to navigate either left and right or up and down, either on a document or on a section of the window. A scroll bar appears as a long bar with a (small) button at each end. Between these buttons, there is a moveable bar called a thumb. To scroll, the user can click one of the buttons or grab the thumb and drag it:



## Types of Scroll Bars

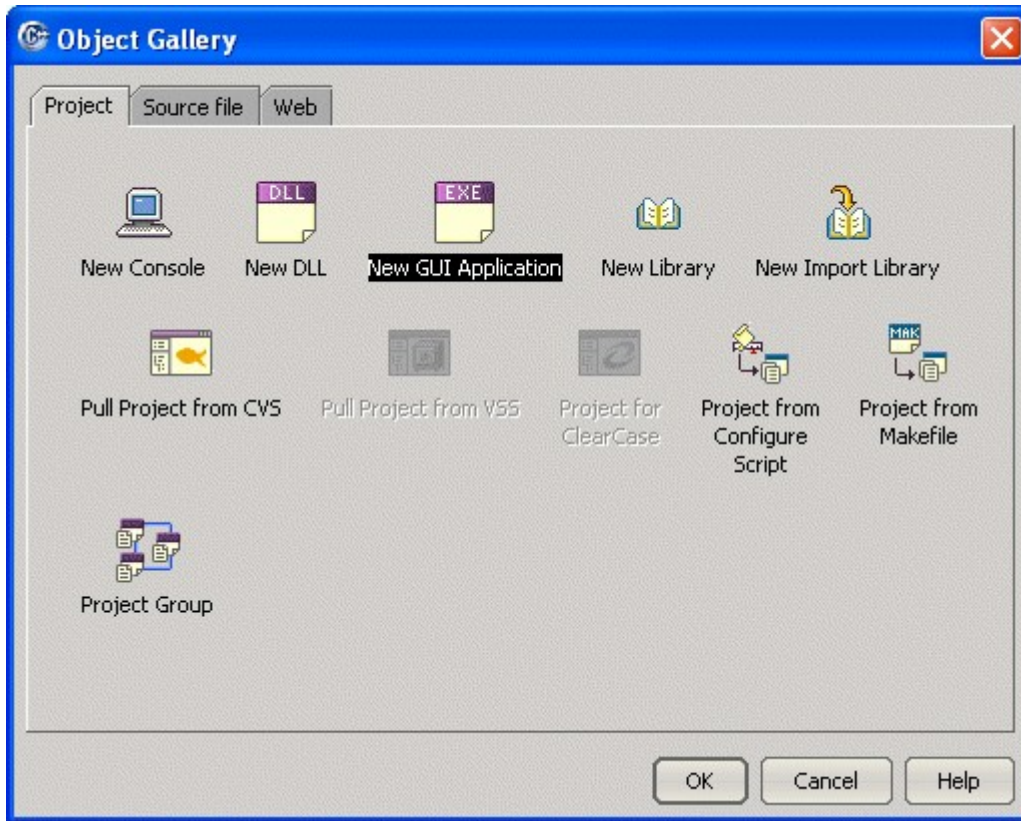
There are two types of scroll bars: vertical or horizontal. A vertical scroll bar allows the user to navigate up and down on a document or a section of a window. A horizontal scroll bar allows the user to navigate left and right on a document or a section of a window.

As far as **Microsoft Windows** is concerned, there are two categories of scroll bars: automatic and control-based.

## ◆ Practical Learning: Introducing Scroll Bars

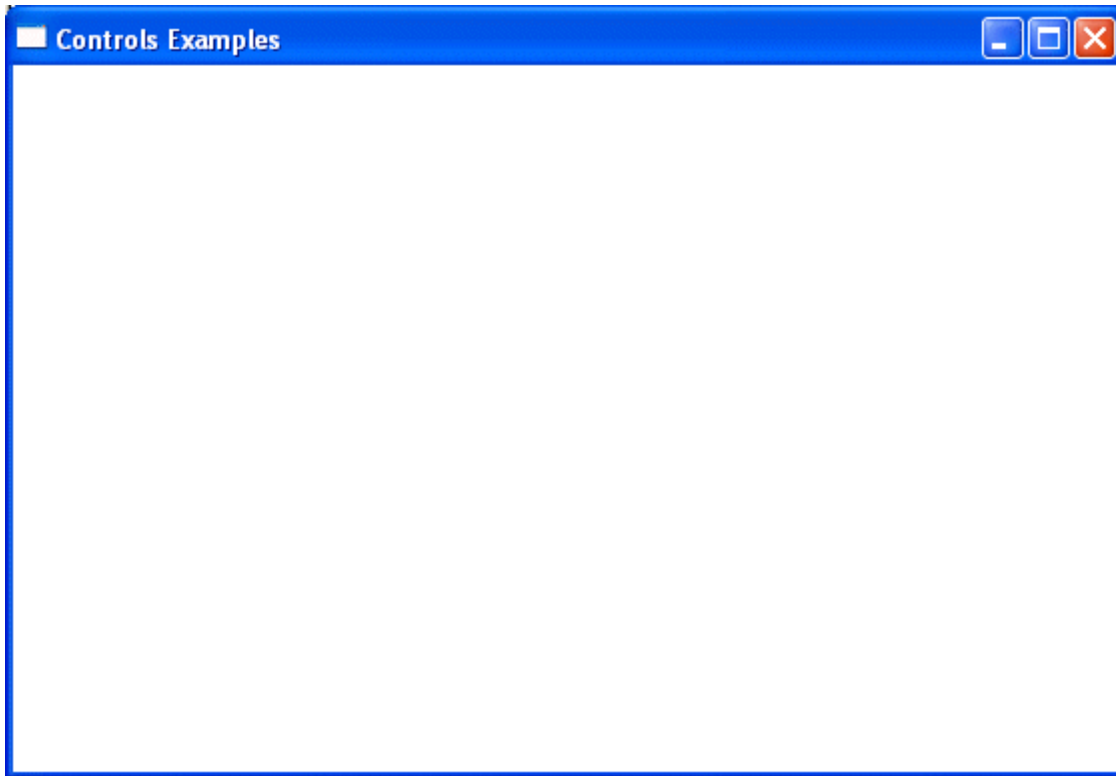
---

1. Because Borland **C++ Builder X** is free, we are going to use it. Start Borland C++ Builder X and, on the main menu, click File -> New...



2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the New GUI Application Project Wizard - Step 1 of 3, in the Directory edit box of the Project Settings section, type the path you want. Otherwise, type **C:\Programs\Win32 Programming**
4. In the Name edit box, type **ScrollBars**





## Automatic Scroll Bars

---

Some controls need a scroll bar to efficiently implement their functionality. The primary example is the edit control, which is used to display text. On that control, when the text is too long, the user needs to be able to scroll down and up to access the document fully. In the same way, if the text is too wide, the user needs to be able to scroll left and right to view the whole document.

When creating a text-based document or window, you can easily ask that one or both scroll bars be added. Of course, an edit control must be able to handle multiple lines of text. This is taken care of by adding the **ES\_MULTILINE** flag to its styles. Then:

- To add a vertical scroll bar to the window, add the **WS\_VSCROLL** flag to the *Style* argument of the **CreateWindow()** or the **CreateWindowEx()** function.
- To add a horizontal scroll bar to the window, add the **WS\_HSCROLL** flag to the *Style* argument of the **CreateWindow()** or the **CreateWindowEx()** function.
- To make the vertical scroll bar appear when necessary, that is, when the document is too long, add the **ES\_AUTOVSCROLL** style
- To make the horizontal scroll bar appear as soon as at least one line of the document is too wide, add the **ES\_AUTOVSCROLL** style

Of course, you can use only one, two, three or all four styles.

## ◆ Practical Learning: Automatically Handling Scroll Bars

---

1. To create a small editor with its scroll bars, modify the procedure as follows:

```

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                               WPARAM wParam, LPARAM lParam)
{
    static HWND hWndEdit;

    switch(Msg)
    {
    case WM_CREATE:

        hWndEdit = CreateWindow("EDIT", // We are creating an Edit
control
                                NULL, // Leave the control empty
                                WS_CHILD | WS_VISIBLE | WS_HSCROLL |
                                WS_VSCROLL | ES_LEFT |
ES_MULTILINE |
                                ES_AUTOHSCROLL | ES_AUTOVSCROLL,
below take care of the size    0, 0, 0, 0, // Let the WM_SIZE message
                                hWnd,
                                0,
                                hInst,
                                NULL);

        return 0;

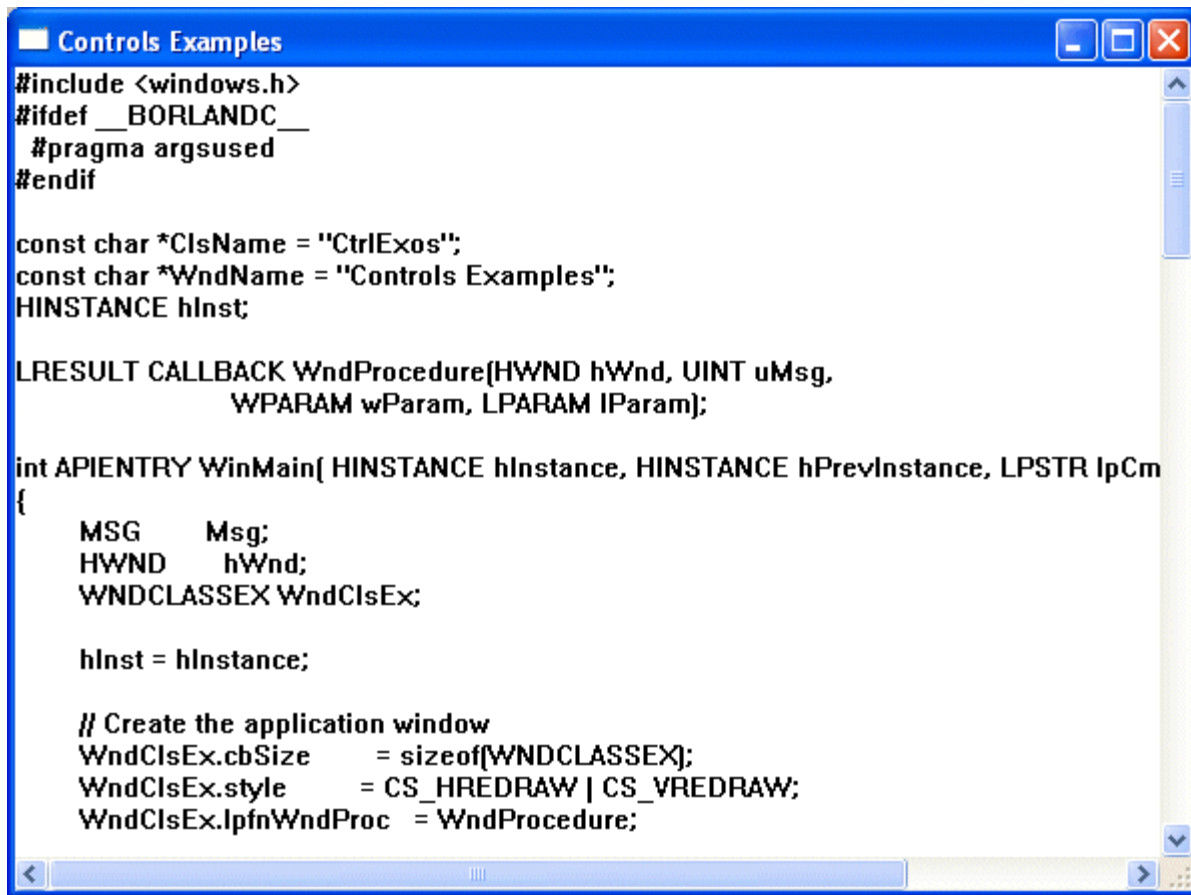
    case WM_SETFOCUS:
        SetFocus(hWndEdit);
        return 0;

    case WM_SIZE:
        MoveWindow(hWndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam),
TRUE);
        return 0;

    case WM_DESTROY:
        // If the user has finished, then close the window
        PostQuitMessage(WM_QUIT);
        break;
    default:
        // Process the left-over messages
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
    // If something was not done, let it go
    return 0;
}

```

2. }  
Test the application



```
#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

const char *ClsName = "CtrlExos";
const char *WndName = "Controls Examples";
HINSTANCE hInst;

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);

int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmd
{
    MSG     Msg;
    HWND    hWnd;
    WNDCLASSEX WndClsEx;

    hInst = hInstance;

    // Create the application window
    WndClsEx.cbSize      = sizeof(WNDCLASSEX);
    WndClsEx.style       = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc = WndProcedure;
```

## Control-Based Scroll Bars

---

### Introduction

Microsoft Windows provides another type of scroll bar. Treated as its own control, a scroll bar is created like any other window and can be positioned anywhere on its host.

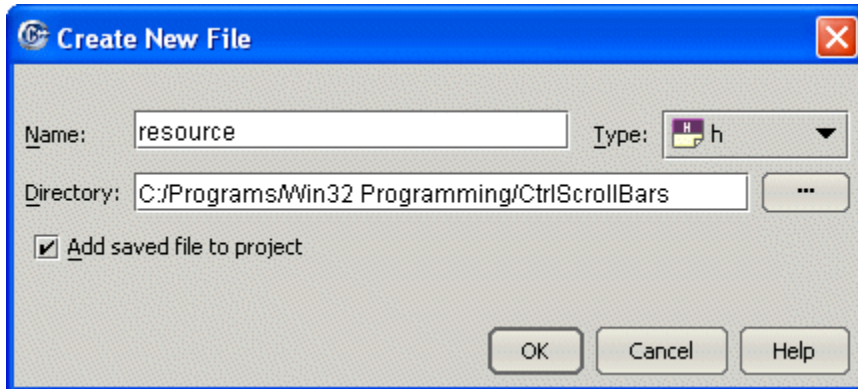
To create a scroll bar as a Windows control, call the **CreateWindow()** or the **CreateWindowEx()** functions and specify the class name as **SCROLLBAR**.

### ◆ Practical Learning: Using Scroll Bar Controls

---

1. Start a new GUI Application and name it **CtrlScrollBars**
2. Create its accompanying file as Exercise.cpp
3. To create a resource header file, on the main menu, click File -> New File...
4. In the Create New File dialog box, in the Name, type **resource**

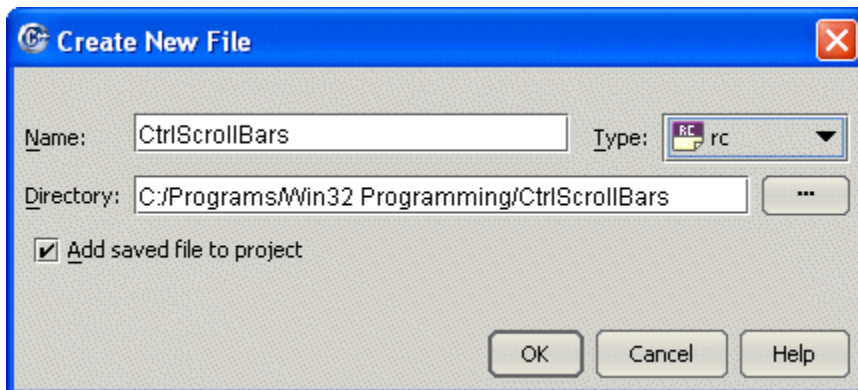
5. In the Type combo box, select h



6. Click OK
7. In the file, type:

```
#define IDD_CONTROLS_DLG 101
#define IDC_CLOSE_BTN 1000
```

8. To create a resource script, on the main menu, click File -> New File...
9. In the Create New File dialog box, in the Name, type **CtrlScrollBars**
10. In the Type combo box, select rc



11. Click OK
12. In the file, type:

```
#include "resource.h"

IDD_CONTROLS_DLG DIALOG DISCARDABLE 200, 150, 235, 151
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON "&Close", IDC_CLOSE_BTN, 178, 7, 50, 14
END
```



13. Display the Exercise.cpp file and change it as follows:

```
#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"

//-----
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----

int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
        hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return 0;
}
//-----

LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
    WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
            case IDC_CLOSE_BTN:
                EndDialog(hWndDlg, 0);
                return TRUE;
        }
        break;

    case WM_CLOSE:
        PostQuitMessage(WM_QUIT);
        break;
    }

    return FALSE;
}
//-----
```

## 14. Test the application

### Scroll Bar Creation

---

The easiest way to add a scroll bar to a project is through the resource script of the project. The syntax to follow is:

```
SCROLLBAR id, x, y, width, height [[, style [[, ExtendedStyle]]]]
```

This declaration starts with the **SCROLLBAR** keyword as the name of the class that creates a scroll bar.

The *id* is the identification of the control

The *x* parameter is the Left parameter of the control

The *y* parameter is the Top parameter of the control

The *width* parameter is the distance from the left to the right border of the control

The *height* parameter is the distance from the top to the bottom border of the control

These 5 parameters are required. Here is an example:

```
SCROLLBAR IDC_SCROLLBAR1,10,45,215,11
```

Alternatively, to create a scroll bar, you can use either the **CreateWindow()** or the **CreateWindowEx()** functions, specifying the class name as **SCROLLBAR**. Here is an example:

```
//-----  
---  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,  
                          WPARAM wParam, LPARAM lParam)  
{  
    switch(Msg)  
    {  
    case WM_INITDIALOG:  
        CreateWindowEx(0L,  
                       "SCROLLBAR",  
                       NULL, // There is no text to display  
                       WS_CHILD | WS_VISIBLE,  
                       50,  
                       20,  
                       220,  
                       21,  
                       hWndDlg,  
                       NULL,  
                       hInst,  
                       NULL);  
  
        return TRUE;  
  
    case WM_CLOSE:  
        PostQuitMessage(WM_QUIT);  
        break;  
    }  
}
```

```

    return FALSE;
}
//-----
---
```

A third alternative is to create your own class. Here is an example:

```

#include <windows.h>
#include "Resource.h"
//-----
---
class WScrollBar
{
public:
    WScrollBar();
    HWND Create(HWND parent, HINSTANCE hinst,
                DWORD dStyle = WS_CHILD | WS_VISIBLE,
                int x = 0, int y = 0, int width = 200, int height = 20);
    virtual ~WScrollBar();
    HWND hWndScrollBar;
private:
};
//-----
---
WScrollBar::WScrollBar()
{
}
//-----
---
WScrollBar::~WScrollBar()
{
}
//-----
---
HWND WScrollBar::Create(HWND parent, HINSTANCE hinst,
                        DWORD dStyle, int x, int y, int width, int height)
{
    hWndScrollBar = CreateWindow("SCROLLBAR", NULL, dStyle,
                                x, y, width, height, parent,
                                NULL, hinst, NULL);

    return hWndScrollBar;
}
//-----
---
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
---
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
```

```

        hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return FALSE;
}
//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
                        WPARAM wParam, LPARAM lParam)
{
    WScrollBar ScrBar;

    switch(Msg)
    {
    case WM_INITDIALOG:
        ScrBar.Create(hWndDlg, hInst);
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            break;
        }
        break;

    case WM_CLOSE:
        PostQuitMessage(WM_QUIT);
        break;
    }

    return FALSE;
}
//-----

```

## ◆ Practical Learning: Creating a Scroll Bar

---

1. Before creating the scroll bar control, modify the resource.h file as follows:

```

#define IDD_CONTROLS_DLG 101
#define IDC_CLOSE_BTN 1000
#define IDC_SCROLLER 1001

```

2. To add a scroll bar to the dialog box, add the following line:

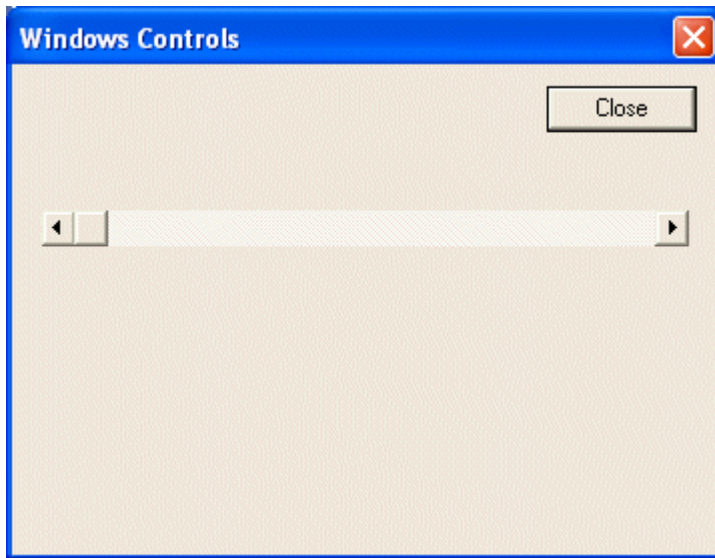
```

#include "resource.h"

IDD_CONTROLS_DLG DIALOG DISCARDABLE 200, 150, 235, 151
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON        "&Close", IDC_CLOSE_BTN, 178, 7, 50, 14
    SCROLLBAR         IDC_SCROLLER, 10, 45, 215, 11
END

```

3. Test the application



4. Return to your programming environment

## Scroll Bar Characteristics

---

As mentioned already, there are two categories of scroll bars. The desired category is set using one of the scroll bar styles. As you can see from the above picture, the default orientation of a scroll bar is horizontal whose value is **SBS\_HORZ**. If you want to produce a vertical scroll bar, you can OR the **SBS\_VERT** style in addition to Windows styles but do not OR both **SBS\_HORZ** and **SBS\_VERT** styles on the same control. Here is an example that creates a vertical scroll bar control. Here is an example that creates a vertical scroll bar:

```
//-----  
---  
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,  
                          WPARAM wParam, LPARAM lParam)  
{  
    WScrollBar ScrBar;  
  
    switch(Msg)  
    {  
    case WM_INITDIALOG:  
        ScrBar.Create(hWndDlg, hInst, WS_CHILD | WS_VISIBLE |  
SBS_VERT,  
                      20, 10, 20, 200);  
        return TRUE;  
  
    case WM_CLOSE:  
        PostQuitMessage(WM_QUIT);  
        break;  
    }  
  
    return FALSE;  
}  
//-----  
---
```

This would produce:



## ◆ Practical Learning: Creating a Vertical Scroll Bar

---

1. To make the scroll bar vertical, change the script as follows:

```
#include "resource.h"

IDD_CONTROLS_DLG DIALOG DISCARDABLE 200, 150, 235, 151
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON        "&Close", IDC_CLOSE_BTN, 178, 7, 50, 14
    SCROLLBAR         IDC_SCROLLER, 10, 10, 15, 130, SBS_VERT
END
```

2. Test the application

## Scroll Bar Functions

---

The **SCROLLBAR** class provides various functions that can be used to set its limits or change its position, etc. To use the scroll bar as a control, you should know its limits. These define the minimum and the maximum values that the thumb can navigate to. To set this range of values, you can call the **SetScrollRange()** function. Its syntax is:

```
BOOL SetScrollRange(HWND hWnd, int nBar, int nMinPos, int nMaxPos, BOOL bRedraw);
```

Here is an example:

```
#include <windows.h>
#include "Resource.h"
//-----
class WScrollBar
{
```

```

public:
    WScrollBar();
    HWND Create(HWND parent, HINSTANCE hinst,
                DWORD dStyle = WS_CHILD | WS_VISIBLE,
                int x = 0, int y = 0, int width = 200, int
height = 20);
    virtual ~WScrollBar();
    HWND hWndScrollBar;
    BOOL SetScrollRange(int min = 0, int max = 100, BOOL redraw =
TRUE);
private:
};
//-----
---
WScrollBar::WScrollBar()
{
}
//-----
---
WScrollBar::~WScrollBar()
{
}
//-----
---
HWND WScrollBar::Create(HWND parent, HINSTANCE hinst,
                        DWORD dStyle, int x, int y, int width, int
height)
{
    hWndScrollBar = CreateWindow("SCROLLBAR", NULL, dStyle,
                                x, y, width, height, parent, NULL, hinst,
NULL);
    return hWndScrollBar;
}
//-----
---
BOOL WScrollBar::SetScrollRange(int min, int max, BOOL redraw)
{
    BOOL SSR = ::SetScrollRange(hWndScrollBar, SB_CTL, min, max, TRUE);

    return SSR;
}
//-----
---
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
---
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
              hWnd, reinterpret_cast<DLGPROC>(DlgProc));
}

```

```

        return FALSE;
    }
//-----
---
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
                        WPARAM wParam, LPARAM lParam)
{
    WScrollBar ScrBar;

    switch(Msg)
    {
    case WM_INITDIALOG:
        ScrBar.Create(hWndDlg, hInst, WS_CHILD | WS_VISIBLE |
SBS_VERT,
                        20, 10, 20, 200);
        ScrBar.SetScrollRange(0, 224);

        return TRUE;

    case WM_CLOSE:
        PostQuitMessage(WM_QUIT);
        break;
    }

    return FALSE;
}
//-----
---

```

If the limits of the controls have already been set, you can find them out by calling the **GetScrollRange()** function. Its syntax is:

```

BOOL GetScrollRange(HWND hWnd, int nBar, LPINT lpMinPos, LPINT lpMaxPos);

```

Here is an example:

```

//-----
---
class WScrollBar
{
public:
    WScrollBar();
    HWND Create(HWND parent, HINSTANCE hinst,
                DWORD dStyle = WS_CHILD | WS_VISIBLE,
                int x = 0, int y = 0, int width = 200, int
height = 20);
    virtual ~WScrollBar();
    HWND hWndScrollBar;
    BOOL SetScrollRange(int min = 0, int max = 100, BOOL redraw =
TRUE);
    BOOL GetScrollRange(int *min, int *max) const;
private:
};
//-----
---
WScrollBar::WScrollBar()

```



```

{
}
//-----
---
WScrollBar::~WScrollBar()
{
}
//-----
---
HWND WScrollBar::Create(HWND parent, HINSTANCE hinst,
                        DWORD dStyle, int x, int y, int width, int height)
{
    hWndScrollBar = CreateWindow("SCROLLBAR", NULL, dStyle,
                                x, y, width, height, parent,
                                NULL, hinst, NULL);

    return hWndScrollBar;
}
//-----
---
BOOL WScrollBar::SetScrollRange(int min, int max, BOOL redraw)
{
    BOOL SSR = ::SetScrollRange(hWndScrollBar, SB_CTL, min, max, TRUE);

    return SSR;
}
//-----
---
BOOL WScrollBar::GetScrollRange(int *min, int *max) const
{
    BOOL GSR = ::GetScrollRange(hWndScrollBar, SB_CTL, min, max);

    return GSR;
}
//-----
---

```

When a window that has a scroll bar control comes up, the thumb is positioned in its minimum value. You may want the control to be positioned somewhere else than that. This attribute is taken care of by the **SetScrollPos()** function. Its syntax:

```
int SetScrollPos(HWND hWnd, int nBar, int nPos, BOOL bRedraw);
```

You can implement and use this function as follows:

```

#include <windows.h>
#include "Resource.h"
//-----
---
class WScrollBar
{
public:
    WScrollBar();
    HWND Create(HWND parent, HINSTANCE hinst,
                DWORD dStyle = WS_CHILD | WS_VISIBLE,
                int x = 0, int y = 0, int width = 200, int
height = 20);
    virtual ~WScrollBar();

```

```

        HWND hWndScrollBar;
        BOOL SetScrollRange(int min = 0, int max = 100, BOOL redraw =
TRUE);
        BOOL GetScrollRange(int *min, int *max) const;
        int SetScrollPos(const int pos);
private:
};
//-----
---
WScrollBar::WScrollBar()
{
}
//-----
---
WScrollBar::~WScrollBar()
{
}
//-----
---
HWND WScrollBar::Create(HWND parent, HINSTANCE hinst,
        DWORD dStyle, int x, int y, int width, int height)
{
        hWndScrollBar = CreateWindow("SCROLLBAR", NULL, dStyle,
        x, y, width, height, parent, NULL, hinst,
NULL);
        return hWndScrollBar;
}
//-----
---
BOOL WScrollBar::SetScrollRange(int min, int max, BOOL redraw)
{
        BOOL SSR = ::SetScrollRange(hWndScrollBar, SB_CTL, min, max, TRUE);

        return SSR;
}
//-----
---
BOOL WScrollBar::GetScrollRange(int *min, int *max) const
{
        BOOL GSR = ::GetScrollRange(hWndScrollBar, SB_CTL, min, max);

        return GSR;
}
//-----
---
int WScrollBar::SetScrollPos(const int position)
{
        int SSP = ::SetScrollPos(hWndScrollBar, SB_CTL, position, TRUE);

        return SSP;
}
//-----
---
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);

```

```

//-----
---
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
             hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return FALSE;
}
//-----
---
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
                        WPARAM wParam, LPARAM lParam)
{
    WScrollBar ScrBar;

    switch(Msg)
    {
    case WM_INITDIALOG:
        ScrBar.Create(hWndDlg, hInst, WS_CHILD | WS_VISIBLE |
SBS_VERT,
                    20, 10, 20, 200);
        ScrBar.SetScrollRange(0, 224);
        ScrBar.SetScrollPos(88);

        return TRUE;

    case WM_CLOSE:
        PostQuitMessage(WM_QUIT);
        break;

    }

    return FALSE;
}
//-----
---

```

## Practical Learning: Initializing a Scroll Bar

---

1. To set the scroll range and initial position of the control, initialize it in the procedure as follows:

```

//-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
                        WPARAM wParam, LPARAM lParam)
{
    HWND hWndScroller;
    SCROLLINFO si;

    hWndScroller = GetDlgItem(hWndDlg, IDC_SCROLLER);

    switch(Msg)
    {
    case WM_INITDIALOG:
        ZeroMemory(&si, sizeof(si));
        si.cbSize = sizeof(si);
        si.fMask = SIF_RANGE | SIF_PAGE | SIF_POS;
        si.nMin = 0;
        si.nMax = 240;
        si.nPage = 10;
        si.nPos = 54;
        SetScrollInfo(hWndScroller, SB_CTL, &si, TRUE);

        return TRUE;

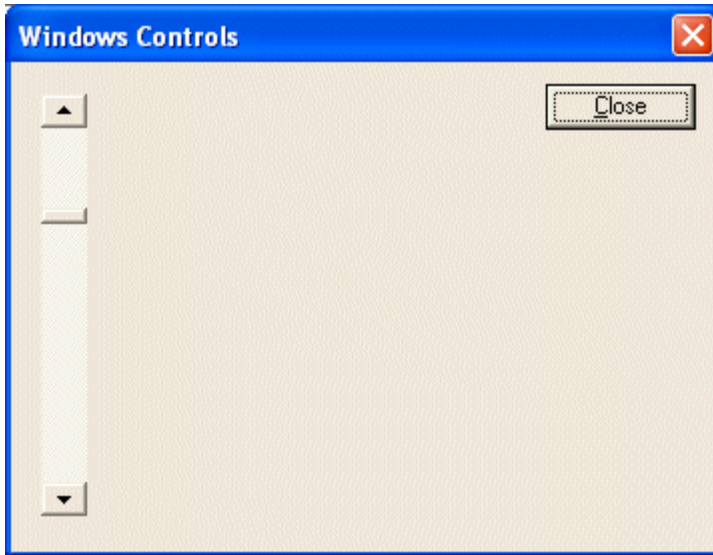
    case WM_COMMAND:
        switch(wParam)
        {
            case IDC_CLOSE_BTN:
                EndDialog(hWndDlg, 0);
                return TRUE;
        }
        break;

    case WM_CLOSE:
        PostQuitMessage(WM_QUIT);
        break;
    }

    return FALSE;
}
//-----

```

2. Test the application



## Scroll Bar Messages

---

To use a scroll bar, a person either clicks one of its buttons, clicks and hold the mouse on one of its buttons, drags the thumb, or clicks an area on either side of the thumb. Any of these actions sends a message that specifies the item the user clicked or dragged. To support this, each category of scroll bar sends the appropriate message.

### ◆ Practical Learning: Processing Scroll Bar Messages

---

1. Modify the resource header as follows:

```
#define IDD_CONTROLS_DLG 101
#define IDC_CLOSE_BTN 1000
#define IDC_SCROLLER 1001
#define IDC_LABEL 1002
```

2. Create a label in the resource script file as follows:

```
#include "resource.h"

IDD_CONTROLS_DLG DIALOG DISCARDABLE 200, 150, 235, 151
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Windows Controls"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON "&Close", IDC_CLOSE_BTN, 178, 7, 50, 14
    SCROLLBAR IDC_SCROLLER, 10, 10, 15, 130, SBS_VERT
    LTEXT "000", IDC_LABEL, 40, 68, 13, 8
END
```

3. To process a few messages of the scroll bar, change the procedure as follows:

```

#include <windows.h>
#include <cstdio>
using namespace std;

#ifdef __BORLANDC__
    #pragma argsused
#endif

#include "resource.h"
//-----
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----

int APIENTRY WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow )
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
        hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return 0;
}
//-----

LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
    WPARAM wParam, LPARAM lParam)
{
    HWND hWndScroller;
    SCROLLINFO si;
    int CurPos;
    char strPosition[20];

    hWndScroller = GetDlgItem(hWndDlg, IDC_SCROLLER);

    switch(Msg)
    {
    case WM_INITDIALOG:
        CurPos = 0;

        ZeroMemory(&si, sizeof(si));
        si.cbSize = sizeof(si);
        si.fMask = SIF_RANGE | SIF_PAGE | SIF_POS;
        si.nMin = 0;
        si.nMax = 240;
        si.nPage = 10;
        si.nPos = 54;
        SetScrollInfo(hWndScroller, SB_CTL, &si, TRUE);

        sprintf(strPosition, "%d", si.nPos);
        SetDlgItemText(hWndDlg, IDC_LABEL, strPosition);

        return TRUE;
    }
}

```

# Windows Controls: Progress Bars

## Overview

---



A progress bar is a Windows control that displays (small) rectangles that are each filled with a color. These (small) rectangles are separate but adjacent each other so that, as they display, they produce a bar. To have the effect of a progress bar, not all these rectangles display at the same time. Instead, a numeric value specifies how many of these (small) rectangles can display at one time.

There are two types of progress bars and various characteristics they can have. Although most progress bars are horizontal, the control can assume a vertical position. We mentioned that a progress bar is made of small colored rectangles. These rectangles can display distinctively from each other although they are always adjacent. Alternatively, these rectangles can be "glued" to produce a smooth effect, in which case they would not appear distinct.

1. Start a new Win32 Project and name it **ProgressTime**
2. Create it as a **Windows Application** and **Empty Project**
3. Display the Add Resource dialog box and double-click Dialog
4. Resize it to 320 x 166
5. Change its ID to **IDD\_CONTROLS\_DLG**
6. Change its **Caption** to **Progress Bar Example**
7. Change its **X Pos** to **200** and change its **Y Pos** to 180
8. Save All
9. Add a new C++ (Source) File and name it **Exercise**
10. Implement it as follows:

```

#include <windows.h>
#include "Resource.h"

//-----
-----
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return FALSE;
}
//-----
-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDOK:
            EndDialog(hWndDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            return TRUE;
        }
        break;
    }

    return FALSE;
}
//-----
-----

```

11. Test the application and return to your programming environment

## **Progress Bar Creation**

Because a progress control belongs to the family of Common Controls, before using it, call the `InitCommonControlsEx()` function. When initializing the `INITCOMMONCONTROLSEX` structure, assign the `ICC_PROGRESS_CLASS` to its `dwICC` member variable.



To create a progress bar, call the **CreateWindowEx()** function and specify the class name as **PROGRESS\_CLASS**.

1. To add a progress to the progress, change the Exercise.cpp source file as follows:

```

#include <windows.h>
#include <commctrl.h>
#include "Resource.h"

//-----
-----
HWND hWnd;
HINSTANCE hInst;
LRESULT CALLBACK DlgProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM
lParam);
//-----
-----
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    hInst = hInstance;

    DialogBox(hInst, MAKEINTRESOURCE(IDD_CONTROLS_DLG),
        hWnd, reinterpret_cast<DLGPROC>(DlgProc));

    return FALSE;
}
//-----
-----
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
WPARAM wParam, LPARAM lParam)
{
    INITCOMMONCONTROLSEX InitCtrlEx;

    InitCtrlEx.dwSize = sizeof(INITCOMMONCONTROLSEX);
    InitCtrlEx.dwICC = ICC_PROGRESS_CLASS;
    InitCommonControlsEx(&InitCtrlEx);

    switch(Msg)
    {
    case WM_INITDIALOG:
        CreateWindowEx(0, PROGRESS_CLASS, NULL,
            WS_CHILD | WS_VISIBLE,
            20, 20, 260, 17,
            hWndDlg, NULL, hInst, NULL);
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDOK:
            EndDialog(hWndDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hWndDlg, 0);
            return TRUE;
        }
        break;
    }

    return FALSE;
}

```

```
SendMessage(hProgress, PBM_GETRANGE, (WPARAM) (MAKELPARAM) (BOOL),
(LPARAM) (MAKELPARAM) (PPBRANGE, ppBRange));
```

## Techniques of Using Menus

---

### Introduction

---

This example show how to create an application with a main menu, how to add an item to the system menu, and how to create a context-sensitive menu activated when the user right-clicks somewhere.

### Resource Header

---

```
#define IDS_APP_NAME 1
#define IDR_MAIN_MENU 101
#define IDR_POPUP 102
#define IDM_FILE_EXIT 40001
#define IDM_SMALL 40002
#define IDM_MEDIUM 40003
#define IDM_LARGE 40004
#define IDM_JUMBO 40005
#define IDM_HELP_ABOUT 40006
```

### Resource Script

---

```
#include "resource.h"

////////////////////////////////////
//
// Menu
//

IDR_MAIN_MENU MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", IDM_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...\tF1", IDM_HELP_ABOUT
    END
END

IDR_POPUP MENU
BEGIN
    POPUP "_POPUP_"
    BEGIN
        MENUITEM "&Small", IDM_SMALL
        MENUITEM "&Medium", IDM_MEDIUM
        MENUITEM "&Large", IDM_LARGE
    END
END
```

```

        MENUITEM SEPARATOR
        MENUITEM "&Jumbo",
                                IDM_JUMBO
    END
END

////////////////////////////////////
//
// String Table
//

STRINGTABLE
BEGIN
    IDM_FILE_EXIT           "Closes the application\nClose"
    IDM_SMALL               "Selects a small pizza"
    IDM_MEDIUM              "Selects a medium pizza"
    IDM_LARGE                "Makes up a large pizza"
    IDM_JUMBO                "Builds an extra-large pizza"
    IDM_HELP_ABOUT          "About this application"
END

STRINGTABLE
BEGIN
    IDS_APP_NAME            "MenuApplied"
END

```

## Source Code

---

```

#include <windows.h>
#include "resource.h"

HINSTANCE hInst;
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    hInst = hInstance;

    const char *ClsName = "MenuApplied";
    const char *WndName = "Techniques of Using Menus";

    // Create the application window
    WndClsEx.cbSize       = sizeof(WNDCLASSEX);
    WndClsEx.style        = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc  = WndProcedure;
    WndClsEx.cbClsExtra   = 0;
    WndClsEx.cbWndExtra   = 0;
    WndClsEx.hIcon        = LoadIcon(NULL, IDI_WARNING);

```

```

WndClsEx.hCursor      = LoadCursor(NULL, IDC_ARROW);
WndClsEx.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1);
WndClsEx.lpszMenuName = MAKEINTRESOURCE(IDR_MAIN_MENU);
WndClsEx.lpszClassName = ClsName;
WndClsEx.hInstance    = hInstance;
WndClsEx.hIconSm      = LoadIcon(NULL, IDI_WARNING);

RegisterClassEx(&WndClsEx);

hWnd = CreateWindow(ClsName,
                   WndName,
                   WS_OVERLAPPEDWINDOW,
                   200,
                   160,
                   460,
                   320,
                   NULL,
                   NULL,
                   hInstance,
                   NULL);

if( !hWnd )
    return 0;

ShowWindow(hWnd, SW_SHOWNORMAL);
UpdateWindow(hWnd);

while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}

return 0;
}

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    // Handle to a menu. This will be used with the context-sensitive
menu
    HMENU hMenu;
    // Handle to the system menu
    HMENU hSysMenu;
    // Handle to the context menu that will be created
    HMENU hMenuTrackPopup;

switch(Msg)
{
    case WM_CREATE:
        // To modify the system menu, first get a handle to it
        hSysMenu = GetSystemMenu(hWnd, FALSE);
        // This is how to add a separator to a menu
        InsertMenu(hSysMenu, 2, MF_SEPARATOR, 0, "-");
        // This is how to add a menu item using a string
        AppendMenu(hSysMenu, MF_STRING, 1, "Practical Techniques");

```

```

// This is how to add a menu item using a defined
identifier AppendMenu(hSysMenu, MF_STRING, IDM_HELP_ABOUT,
"About...");
return 0;

case WM_COMMAND:
switch(LOWORD(wParam))
{
case IDM_LARGE:
MessageBox(hWnd, "Menu Item Selected = Large",
"Message", MB_OK);
break;

case IDM_FILE_EXIT:
PostQuitMessage(WM_QUIT);
break;
}
return 0;

case WM_CONTEXTMENU:
// Get a handle to the popup menu using its resource
if( (hMenu = LoadMenu(hInst, MAKEINTRESOURCE(IDR_POPUP)))
== NULL )
return 0;

// Get a handle to the first shortcut menu
hMenuTrackPopup = GetSubMenu(hMenu, 0);

// Display the popup menu when the user right-clicks
TrackPopupMenu(hMenuTrackPopup,
TPM_LEFTALIGN | TPM_RIGHTBUTTON,
LOWORD(lParam),
HIWORD(lParam),
0,
hWnd,
NULL);

break;

case WM_DESTROY:
PostQuitMessage(WM_QUIT);
break;

default:
return DefWindowProc(hWnd, Msg, wParam, lParam);
}

return 0;
}

```

## A Popup Window

---



## Introduction

A window is as popup when it is relatively small, is equipped with a short than usual title bar, is equipped with only the **System** button on its title bar, and has thin borders. Such a window is usually used to accompany or assist another big the main window of an application. This means that a **popup window** is hardly used by itself or as the main application. Based on this description, there are some characteristics you should apply to the frame to make a popup window.

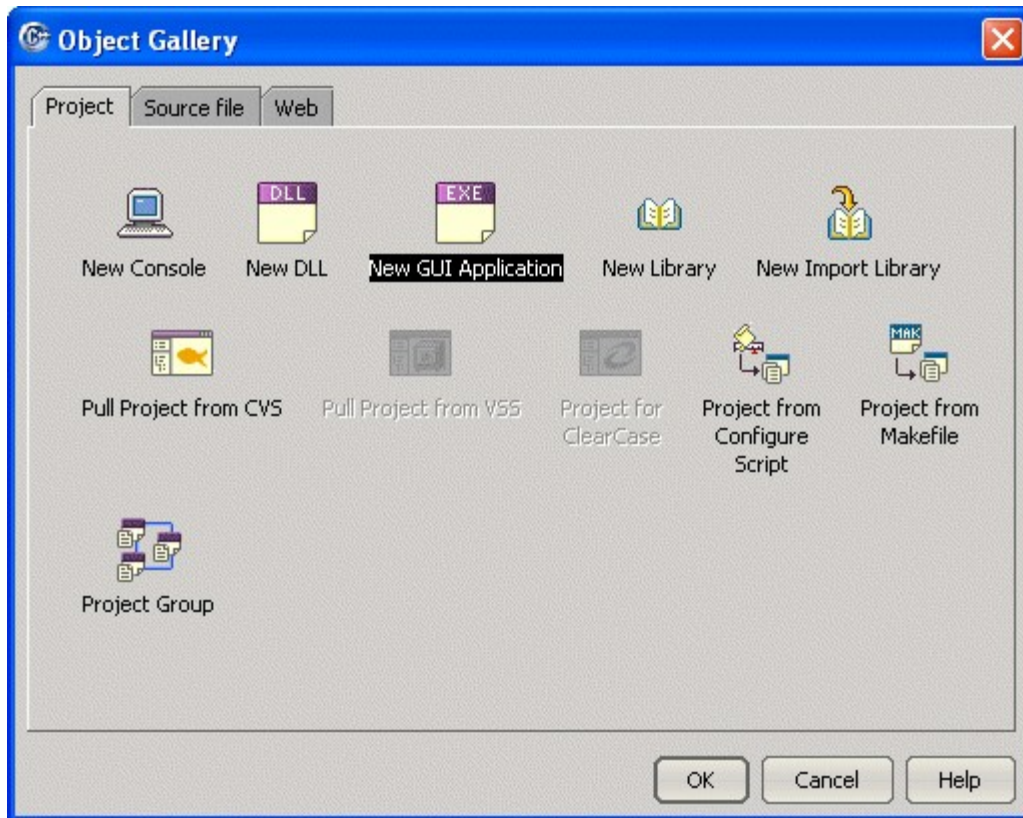
To create a popup window, it should have the **WS\_POPUPWINDOW** and the **WS\_CAPTION** styles. It is a relatively small rectangle. It may also have the **WS\_EX\_TOOLWINDOW** extended style.

In this exercise, to illustrate our point, we will create a frame-based application where a popup window is the main

1.

### ◆ Practical Learning: Introducing Tables

Start your programming environment. For this example, we will use Borland C++BuilderX. Therefore, start Borland C++BuilderX and, on the main menu, click File -> New...



2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the New GUI Application Project Wizard - Step 1 of 3, in the Directory edit box of the Project Settings s type the path you want. Otherwise, type **C:\Programs\Win32 Programming**
4. In the Name edit box, type **Popup1** and click Next
5. In the New GUI Application Project Wizard - Step 2 of 3, accept the defaults and click Next
6. In the New GUI Application Project Wizard - Step 3 of 3, click the check box under Create
7. Select Untitled under the Name column header. Type **Exercise** to replace the name and press Tab
8. Click Finish
9. Display the Exercise.cpp file and change it as follows:



```

#include <windows.h>
#ifdef __BORLANDC__
    #pragma argsused
#endif
//-----
---
const char *ClsName = "WndProperties";
const char *WndName = "Popup Window";
//-----
---
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);
//-----
---
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    // Create the application window
    WndClsEx.cbSize       = sizeof(WNDCLASSEX);
    WndClsEx.style        = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc  = WndProcedure;
    WndClsEx.cbClsExtra   = 0;
    WndClsEx.cbWndExtra   = 0;
    WndClsEx.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor      = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName = NULL;
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance    = hInstance;
    WndClsEx.hIconSm      = LoadIcon(NULL, IDI_APPLICATION);

    // Register the application
    RegisterClassEx(&WndClsEx);

    // Create the window object
    hWnd = CreateWindowEx(WS_EX_TOOLWINDOW,
                        ClsName, WndName,
                        WS_POPUPWINDOW | WS_CAPTION,
                        200, 120, 200, 320,
                        NULL, NULL, hInstance, NULL);

    // Find out if the window was created
    if( !hWnd ) // If the window was not created,
        return 0; // stop the application

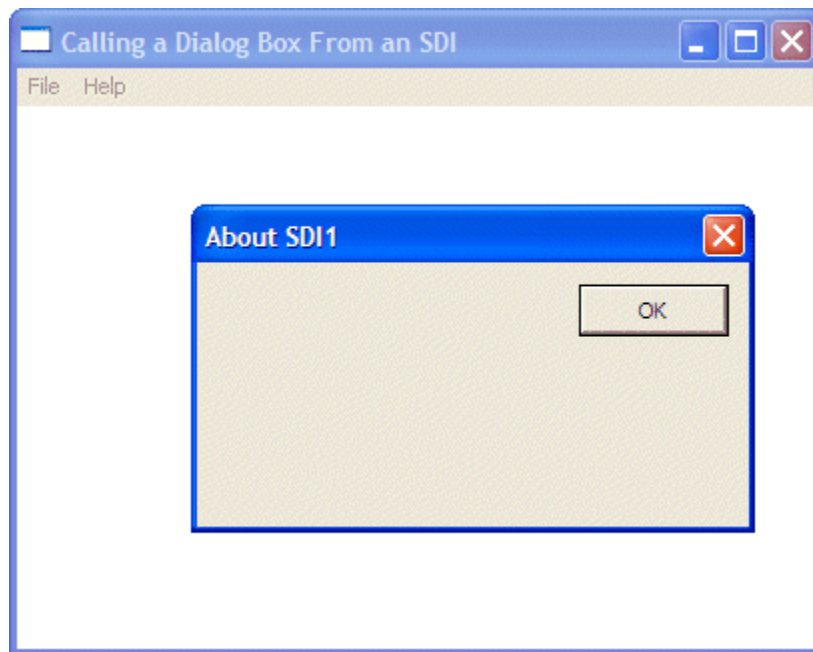
    // Display the window to the user
    ShowWindow(hWnd, SW_SHOWNORMAL);
    UpdateWindow(hWnd);

    // Decode and treat the messages
    // as long as the application is running
    while( GetMessage(&Msg, NULL, 0, 0) )
    {

```

# Calling a Dialog Box From an SDI

---



## Introduction

---

A Single Document Interface (SDI) is the type of application that presents a frame-based window equipped with a title bar, a menu, and thick borders. In most cases the one-frame window is enough to support the application.

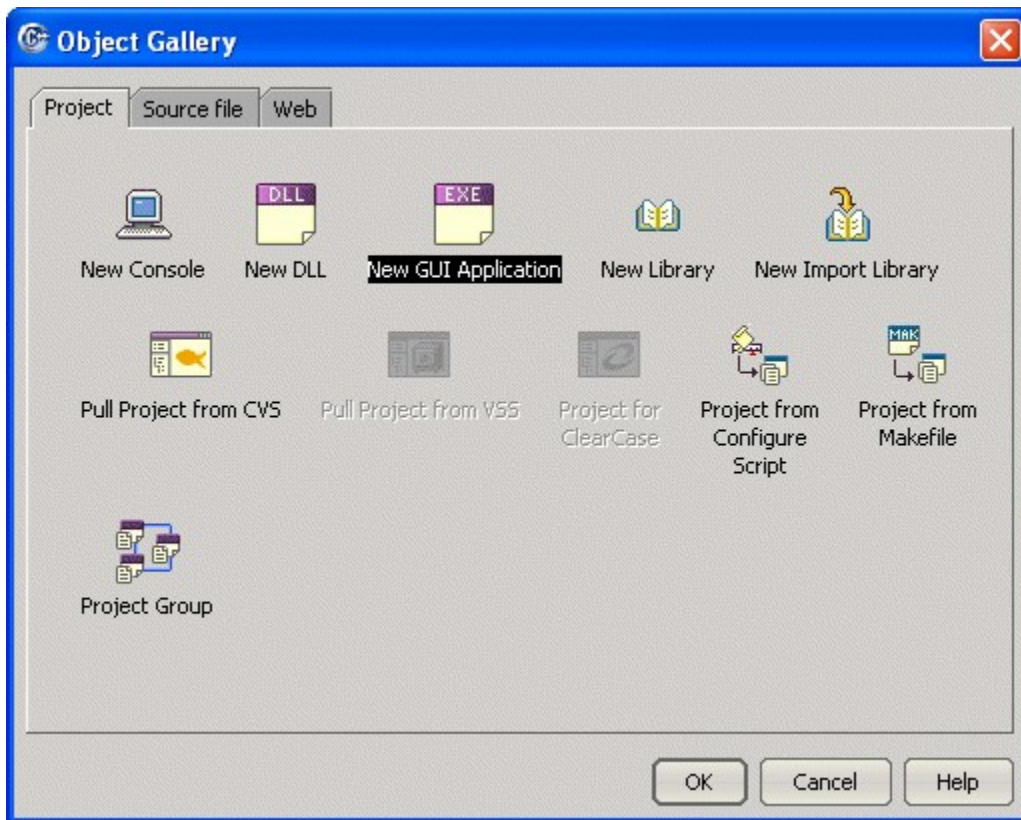
To provide support and various options to the frame of an SDI, you can add other dependent windows such as dialog boxes. When such a dialog box is needed, you can provide a menu item that the user would use to display the dialog box. Just as you can add one dialog box to an application, in the same way, you can add as many dialog boxes as you judge necessary to your SDI application and provide a way for the user to display them.

To display a dialog from an SDI, the easiest and most usual way consists of creating a menu item on the main menu. When implementing the event of the menu item, as normally done in C++, in the source file of the class that would call it, include the header file of the dialog box. In the event that calls the dialog box, first declare a variable based on the name of the class of the dialog box.

## ◆ Practical Learning: Calling a Dialog Box From an SDI

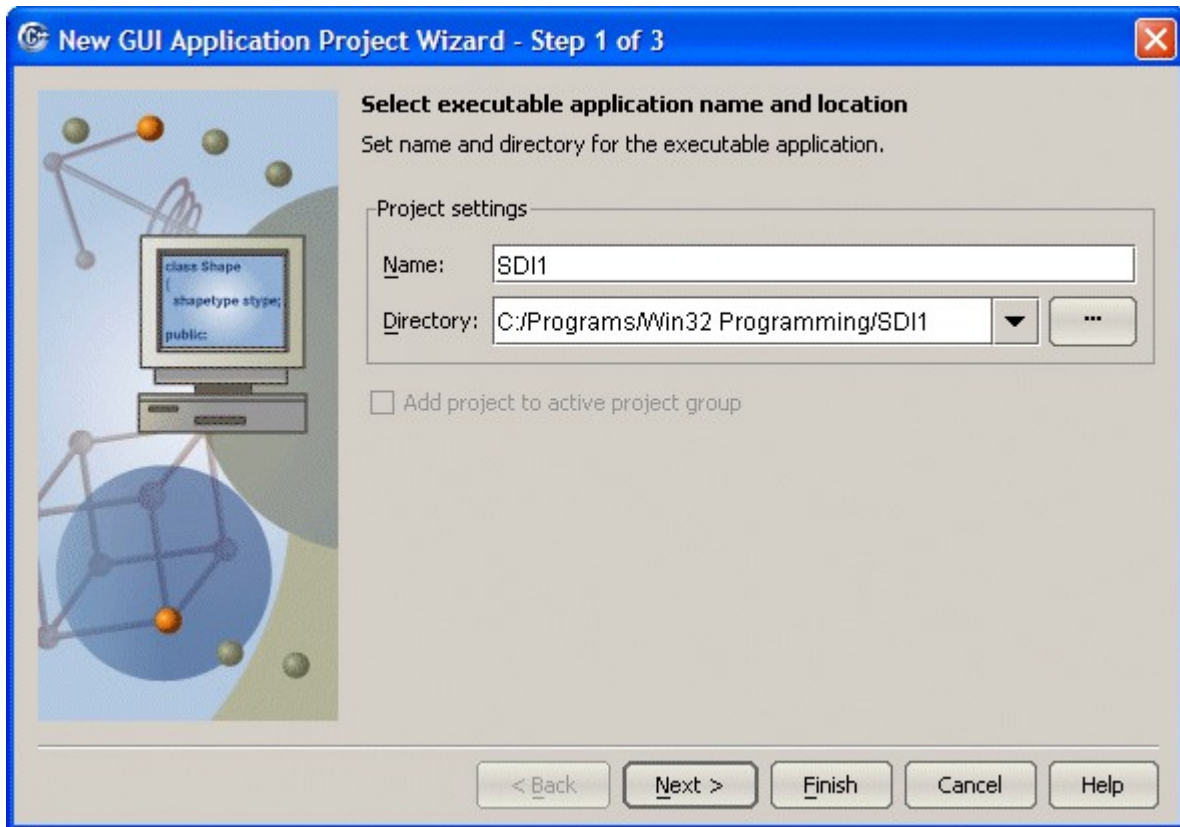
---

1. Start your programming environment. For this example, we will use Borland C++BuilderX.  
Therefore, start Borland C++BuilderX and, on the main menu, click File -> New...



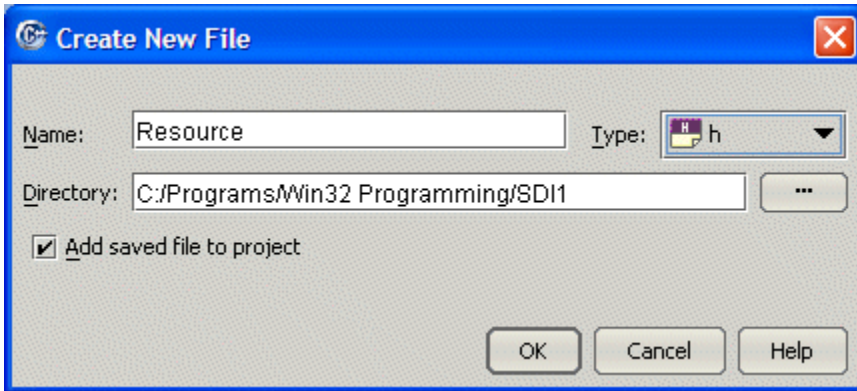
2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the New GUI Application Project Wizard - Step 1 of 3, in the Directory edit box of the Project Settings section, type the path you want. Otherwise, type **C:\Programs\Win32 Programming**

4. In the Name edit box, type **SDI1**



5. Click Next
6. In the New GUI Application Project Wizard - Step 2 of 3, accept the defaults and click Next
7. In the New GUI Application Project Wizard - Step 3 of 3, click the check box under Create
8. Select Untitled under the Name column header. Type **Exercise** to replace the name and press Tab
9. Click Finish
10. To create a resource header file, on the main menu, click File -> New File...
11. In the Create New File dialog box, in the Name, type **Resource**

12. In the Type combo box, select h, and click OK



13. In the file, type:

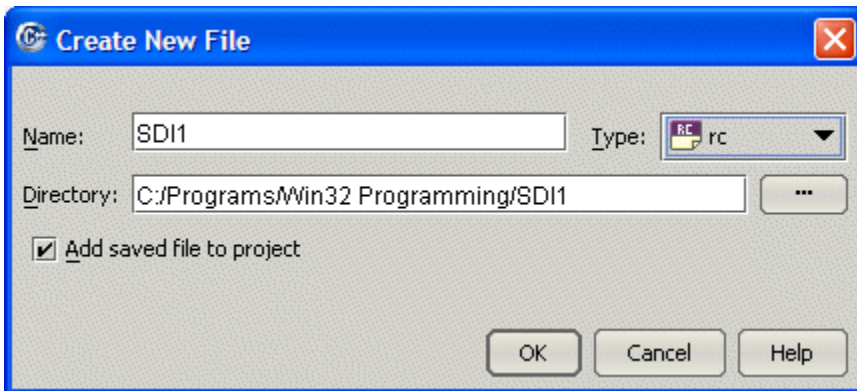
```
#define IDR_MAIN_MENU      1001
#define IDM_FILE_EXIT      1002
#define IDM_HELP_ABOUT     1003

#define IDD_ABOUTDLGBOX    1010
```

14. To create a resource script, on the main menu, click File -> New File...

15. In the Create New File dialog box, in the Name, type **SDI1**

16. In the Type combo box, select rc, and click OK



17. In the file, type:

```
#include "Resource.h"

// Main Menu
IDR_MAIN_MENU MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",      IDM_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
```

```

        MENUITEM "&About...", IDM_HELP_ABOUT
    END
END

// Dialog Box: About
IDD_ABOUTDLGBOX DIALOGEX 0, 0, 184, 81
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP |
WS_CAPTION |
    WS_SYSMENU
CAPTION "About SDI1"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON "OK",IDOK,127,7,50,16,WS_GROUP
END

```

18. Display the Exercise.cpp file and change it as follows:

```

#include <windows.h>
#include "Resource.h"

#ifdef __BORLANDC__
    #pragma argsused
#endif
//-----
---
HINSTANCE hInst;
const char *ClsName = "CallingDlg";
const char *WndName = "Calling a Dialog Box From an SDI";
//-----
---
LRESULT CALLBACK WndProcedure(HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg,
                          WPARAM wParam, LPARAM lParam);
//-----
---
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine, int nCmdShow )
{
    MSG        Msg;
    HWND       hWnd;
    WNDCLASSEX WndClsEx;

    // Create the application window
    WndClsEx.cbSize        = sizeof(WNDCLASSEX);
    WndClsEx.style         = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc   = WndProcedure;
    WndClsEx.cbClsExtra    = 0;
    WndClsEx.cbWndExtra    = 0;
    WndClsEx.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    WndClsEx.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    WndClsEx.lpszMenuName  = MAKEINTRESOURCE(IDR_MAIN_MENU);
    WndClsEx.lpszClassName = ClsName;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

```

```

// Register the application
RegisterClassEx(&WndClsEx);
hInst = hInstance;

// Create the window object
hWnd = CreateWindow(ClsName,
                   WndName,
                   WS_OVERLAPPEDWINDOW,
                   CW_USEDEFAULT,
                   CW_USEDEFAULT,
                   CW_USEDEFAULT,
                   CW_USEDEFAULT,
                   NULL,
                   NULL,
                   hInstance,
                   NULL);

// Find out if the window was created
if( !hWnd ) // If the window was not created,
           return 0; // stop the application

// Display the window to the user
ShowWindow(hWnd, SW_SHOWNORMAL);
UpdateWindow(hWnd);

// Decode and treat the messages
// as long as the application is running
while( GetMessage(&Msg, NULL, 0, 0) )
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}

return Msg.wParam;
}
//-----
---
```

---

```

LRESULT CALLBACK WndProcedure(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDM_FILE_EXIT:
            PostQuitMessage(WM_QUIT);
            break;
        case IDM_HELP_ABOUT:
            DialogBox(hInst,
                     MAKEINTRESOURCE(IDD_ABOUTDLGBOX),
                     hWnd,
                     reinterpret_cast<DLGPROC>(DlgProc));
            break;
        }
    }
}

```

```

        return 0;

// If the user wants to close the application
case WM_DESTROY:
    // then close it
    PostQuitMessage(WM_QUIT);
    break;
default:
    // Process the left-over messages
    return DefWindowProc(hWnd, Msg, wParam, lParam);
}
// If something was not done, let it go
return 0;
}
//-----
---
```

```

LRESULT CALLBACK DlgProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM
lParam)
{
    switch(Msg)
    {
    case WM_INITDIALOG:
        return TRUE;

    case WM_COMMAND:
        switch(wParam)
        {
        case IDOK:
            EndDialog(hWndDlg, 0);
            return TRUE;

        }
        break;
    }

    return FALSE;
}
//-----
---
```

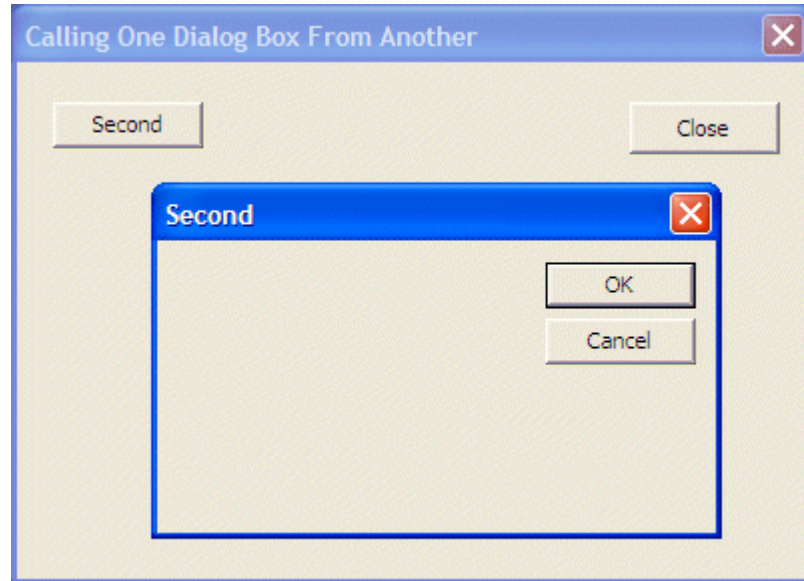
19.

20. Test the application

## Calling One Dialog Box From Another Dialog Box

---





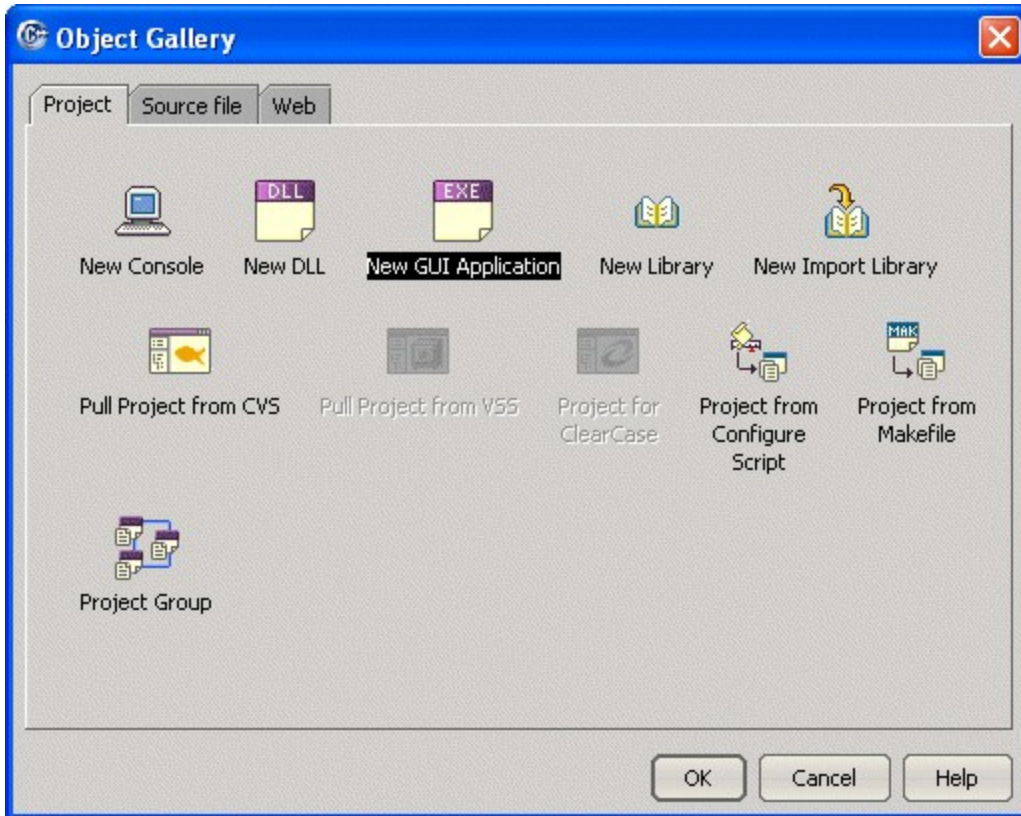
## Introduction

It is unusual for a dialog-based application. This is because a dialog box is usually made to complement a frame-based application. In the same way, if you create an application made of various dialog boxes, at one time, you may want the application to be able to call one dialog box from another. To start, you would create the necessary dialog boxes.

1.

### ◆ Practical Learning: Calling One Dialog Box From Another

Start your programming environment. For this example, we will use Borland C++BuilderX. Therefore, start Borland C++BuilderX and, on the main menu, click File -> New...



2. In the Object Gallery dialog box, click New GUI Application and click OK
3. In the Name edit box, type **Primary1** and click Next
4. In the New GUI Application Project Wizard - Step 2 of 3, accept the defaults and click Next
5. In the New GUI Application Project Wizard - Step 3 of 3, click the check box under Create
6. Select Untitled under the Name column header. Type **Exercise** to replace the name and press Tab
7. Click Finish
8. To create a resource header file, on the main menu, click File -> New File...
9. In the Create New File dialog box, in the Name, type **Resource**
10. In the Type combo box, select h, and click OK
11. In the file, type:

```
#define IDD_PRIMARY_DLG 102
#define IDD_SECOND_DLG 104
#define IDC_SECOND_BTN 106
```

12. script, on the main menu, click File -> New File...

To create a res

13. In the Create New File dialog box, in the Name, type **Primary1**
14. In the Type combo box, select rc, and click OK
15. In the file, type:

```

#include "Resource.h"

/* -- Dialog Box: Primary -- */

IDD_PRIMARY_DLG DIALOGEX 0, 0, 263, 159
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP |
WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Calling One Dialog Box From Another"
FONT 8, "MS Shell Dlg"
BEGIN
    PUSHBUTTON        "Close", IDCANCEL, 204, 12, 50, 16
    PUSHBUTTON        "Second", IDC_SECOND_BTN, 12, 12, 50, 14
END

/* -- Dialog Box: Second -- */

IDD_SECOND_DLG DIALOGEX 0, 0, 186, 90
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP |
WS_CAPTION |
    WS_SYSMENU
CAPTION "Second"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON    "OK", IDOK, 129, 7, 50, 14
    PUSHBUTTON        "Cancel", IDCANCEL, 129, 24, 50, 14
END

```

16. file and change it as follows:

Display the Exe

```

#include <windows.h>
#include "Resource.h"

#ifdef __BORLANDC__
    #pragma argsused
#endif
//-----
---
HWND hWnd;
HINSTANCE hInst;
const char *ClsName = "CallingDlg";
const char *WndName = "Calling One Dialog Box From Another";
LRESULT CALLBACK DlgProcPrimary(HWND hWndDlg, UINT Msg,
                                WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK DlgProcSecondary(HWND hWndDlg, UINT Msg,
                                   WPARAM wParam, LPARAM lParam);
//-----
---
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    DialogBox(hInstance, MAKEINTRESOURCE(IDD_PRIMARY_DLG),
             hWnd, (DLGPROC)DlgProcPrimary);

    hInst = hInstance;
    return FALSE;
}
//-----
---
LRESULT CALLBACK DlgProcPrimary(HWND hWndDlg, UINT Msg,
                                WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:
            switch(wParam)
            {
                case IDC_SECOND_BTN:
                    DialogBox(hInst, MAKEINTRESOURCE(IDD_SECOND_DLG),
                             hWnd, (DLGPROC)DlgProcSecondary);
                    return FALSE;
                case IDCANCEL:
                    EndDialog(hWndDlg, 0);
                    return TRUE;
            }
            break;
    }

    return FALSE;
}
//-----
---
LRESULT CALLBACK DlgProcSecondary(HWND hWndDlg, UINT Msg,
                                   WPARAM wParam, LPARAM lParam)

```

# A Child Window Attached to a Frame



## Introduction

This is an example of creating and attaching a child window, like a toolbox, to the main frame of an application.

## Resource Header

```
#define IDD_TOOLBOX_DLG          101
#define IDR_MAIN_MENU           102
#define IDM_FILE_EXIT           40001
#define IDM_VIEW_TOOLBOX       40002
```

## Resource Script

```
#include "resource.h"

//////////////////////////////////////
//
// Dialog
//

IDD_TOOLBOX_DLG DIALOG DISCARDABLE 0, 0, 86, 249
STYLE DS_MODALFRAME | WS_CHILD
FONT 8, "MS Sans Serif"
BEGIN
END
```

```

////////////////////////////////////
//
// Menu
//

IDR_MAIN_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",          IDM_FILE_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbox",      IDM_VIEW_TOOLBOX, CHECKED
    END
END

////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    IDM_FILE_EXIT          "Closes the application"
    IDM_VIEW_TOOLBOX      "Toggles the presence and disappearance of the toolbox\nShow/Hide Toolbox"
END

#endif // English (U.S.) resources
////////////////////////////////////

```

## Source Code

---

```

#include <windows.h>
#include "resource.h"

HINSTANCE hInst;
LPTSTR strAppName = "WndFrame";
LPTSTR WndName = "Attaching a child window to an application's frame";
LPTSTR strToolbox = "WndFloater";

HWND hWndMainFrame, hWndToolbox;

LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM
                              lParam);
LRESULT CALLBACK ToolboxProc(HWND hWnd, UINT Msg,
                              WPARAM wParam, LPARAM
                              lParam);
//-----
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    RECT rect;
    WNDCLASSEX WndClsEx;

    WndClsEx.cbSize = sizeof(WNDCLASSEX);

```

```

WndClsEx.style      = CS_HREDRAW | CS_VREDRAW;
WndClsEx.lpfWndProc = MainWndProc;
WndClsEx.cbClsExtra = 0;
WndClsEx.cbWndExtra = 0;
WndClsEx.hIcon      = LoadIcon(NULL, IDI_APPLICATION);
WndClsEx.hCursor    = LoadCursor(NULL, IDC_ARROW);
WndClsEx.hbrBackground = static_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
WndClsEx.lpszMenuName = MAKEINTRESOURCE(IDR_MAIN_MENU);
WndClsEx.lpszClassName = strAppName;
WndClsEx.hInstance  = hInstance;
WndClsEx.hIconSm    = LoadIcon(NULL, IDI_APPLICATION);

if (!RegisterClassEx(&WndClsEx))
    return (FALSE);

hInst = hInstance;

hWndMainFrame = CreateWindow(strAppName,
                             WndName,
                             WS_OVERLAPPEDWINDOW,
                             CW_USEDEFAULT,
                             CW_USEDEFAULT,
                             CW_USEDEFAULT,
                             CW_USEDEFAULT,
                             NULL,
                             NULL,
                             hInstance,
                             NULL);

if( !hWndMainFrame )
    return (FALSE);

// Create a child window based on the available dialog box
hWndToolbox = CreateDialog(hInst,
                           MAKEINTRESOURCE(IDD_TOOLBOX_DLG),
                           hWndMainFrame,
                           (DLGPROC)ToolboxProc);

ShowWindow (hWndToolbox, SW_SHOW);
ShowWindow(hWndMainFrame, nCmdShow);

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

return 0;
}
//-----
LRESULT CALLBACK ToolboxProc(HWND hWndDlg, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_INITDIALOG:
            return TRUE;
    }
}

```

```

        return FALSE;
    }
//-----
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg,
                             WPARAM wParam, LPARAM lParam)
{
    HMENU hMenu;
    RECT rctMainWnd, rctToolbox;
    UINT ToolboxMenuState;

    switch(Msg)
    {
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDM_VIEW_TOOLBOX:
            hMenu = GetMenu(hWndMainFrame);
            ToolboxMenuState = GetMenuState(hMenu, IDM_VIEW_TOOLBOX,
MF_BYCOMMAND);
            if( LOBYTE(ToolboxMenuState) & MF_CHECKED )
            {
                CheckMenuItem(hMenu, IDM_VIEW_TOOLBOX,
MF_BYCOMMAND | MF_UNCHECKED);
                ShowWindow(hWndToolbox, SW_HIDE);
            }
            else
            {
                CheckMenuItem(hMenu, IDM_VIEW_TOOLBOX,
MF_BYCOMMAND | MF_CHECKED);
                ShowWindow(hWndToolbox, SW_SHOW);
            }
            break;

        case IDM_FILE_EXIT:
            PostQuitMessage(WM_QUIT);
            return 0;
        };
        break;

    case WM_SIZE:
        GetClientRect(hWndMainFrame, &rctMainWnd);
        GetWindowRect(hWndToolbox, &rctToolbox);

        SetWindowPos(hWndToolbox,
            HWND_TOP,
            rctMainWnd.left,
            rctMainWnd.top,
            rctToolbox.right - rctToolbox.left,
            rctMainWnd.bottom,
            SWP_NOACTIVATE | SWP_NOOWNERZORDER);

        break;

    case WM_DESTROY:
        PostQuitMessage(WM_QUIT);
        break;

    default:
        return DefWindowProc(hWnd, Msg, wParam, lParam);
    }
}

```



```

    }
    return 0;
}
//-----

```

# A Multiple Document Interface (MDI)

---

## Introduction

---

This is an example of an MDI.

## Resource Header

---

```

#define IDR_MAIN_MENU          101
#define IDM_FILE_NEW          40001
#define IDM_FILE_CLOSE        40002
#define IDM_FILE_EXIT         40003
#define IDM_WINDOW_TILE       40004
#define IDM_WINDOW_CASCADE    40005
#define IDM_WINDOW_ICONS      40006
#define IDM_WINDOW_CLOSE_ALL  40007

```

## Resource Script

---

```

#include "resource.h"

////////////////////////////////////
//
// Menu
//

IDR_MAIN_MENU MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N",          40001
        MENUITEM SEPARATOR
        MENUITEM "&Close",                40002
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                  40003
    END
    POPUP "&Window"
    BEGIN
        MENUITEM "&Tile",                  40004
        MENUITEM "C&ascade",                40005
        MENUITEM "Arrange &Icons",          40006
        MENUITEM "CI&ose All",              40007
    END
END

////////////////////////////////////
//
// String Table

```

```
//
STRINGTABLE
BEGIN
    IDM_FILE_NEW          "Creates a new document\nNew"
    IDM_FILE_CLOSE        "Closes the current document\nClose"
    IDM_FILE_EXIT         "Closes the application\nExit"
    IDM_WINDOW_TILE       "Arranges child windows in a tile format"
    IDM_WINDOW_CASCADE    "Arranges child windows in a cascade format"
    IDM_WINDOW_ICONS      "Arranges the icons of minimized child windows"
    IDM_WINDOW_CLOSE_ALL  "Closes all child windows"
END
```

## Source Code

---

```
#include <windows.h>
#include "resource.h"

const char MainClassName[] = "MainMDIWnd";
const char ChildClassName[] = "MDIChildWnd";

const int StartChildrenNo = 994;

HWND hWndMainFrame = NULL;
HWND hWndChildFrame = NULL;

BOOL CreateNewDocument(HINSTANCE hInstance);
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT msg,
                               WPARAM wParam, LPARAM
lParam);
BOOL CreateNewDocument(HINSTANCE hInstance);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX WndClsEx;
    MSG Msg;

    WndClsEx.cbSize          = sizeof(WNDCLASSEX);
    WndClsEx.style           = 0;
    WndClsEx.lpfWndProc      = MainWndProc;
    WndClsEx.cbClsExtra     = 0;
    WndClsEx.cbWndExtra     = 0;
    WndClsEx.hInstance      = hInstance;
    WndClsEx.hIcon           = LoadIcon(NULL, IDI_ASTERISK);
    WndClsEx.hCursor        = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    WndClsEx.lpszMenuName   = MAKEINTRESOURCE(IDR_MAIN_MENU);
    WndClsEx.lpszClassName  = MainClassName;
    WndClsEx.hIconSm        = LoadIcon(NULL, IDI_ASTERISK);

    if(!RegisterClassEx(&WndClsEx))
    {
        MessageBox(NULL,
                   "Window Registration Failed!", "Error!",
                   MB_OK);
        return 0;
    }

    if( !CreateNewDocument(hInstance) )
```

```

        return 0;

    hWndMainFrame = CreateWindowEx(0L,
        MainClassName,
        "Multiple Document Application",
        WS_OVERLAPPEDWINDOW |
WS_CLIPCHILDREN,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL);

    if(hWndMainFrame == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hWndMainFrame, nCmdShow);
    UpdateWindow(hWndMainFrame);

    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        if (!TranslateMDISysAccel(hWndChildFrame, &Msg))
        {
            TranslateMessage(&Msg);
            DispatchMessage(&Msg);
        }
    }
    return 0;
}

HWND CreateNewMDIChild(HWND hMDIClient)
{
    MDICREATESTRUCT mcs;
    HWND NewWnd;

    mcs.szTitle = "Untitled";
    mcs.szClass = ChildClassName;
    mcs.hOwner = GetModuleHandle(NULL);
    mcs.x = mcs.cx = CW_USEDEFAULT;
    mcs.y = mcs.cy = CW_USEDEFAULT;
    mcs.style = MDIS_ALLCHILDSTYLES;

    NewWnd = (HWND)SendMessage(hMDIClient, WM_MDICREATE, 0, (LONG)&mcs);

    if( !NewWnd )
    {
        MessageBox(NULL,
            "Error creating child window",
            "Creation Error",
            MB_OK);
    }
    return NewWnd;
}

```

```

}

LRESULT CALLBACK MainWndProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
    switch(Msg)
    {
        case WM_CREATE:
        {
            CLIENTCREATESTRUCT ccs;

            ccs.hWindowMenu = GetSubMenu(GetMenu(hWnd), 2);
            ccs.idFirstChild = StartChildrenNo;

            hWndChildFrame = CreateWindowEx(WS_EX_CLIENTEDGE,
                "MDIClient",

            NULL,

            WS_CHILD | WS_CLIPCHILDREN | WS_VSCROLL
                | WS_HSCROLL | WS_VISIBLE,
                CW_USEDEFAULT,

            CW_USEDEFAULT,

            680,

            460,

            hWnd,

            (HMENU)IDM_FILE_NEW,

            GetModuleHandle(NULL),

            (LPVOID)&ccs);

            if(hWndChildFrame == NULL)
                MessageBox(hWnd, "Could not create MDI client.",
"Error", MB_OK | MB_ICONERROR);
        }
        CreateNewMDIChild(hWndChildFrame);
        break;
        case WM_SIZE:
        {
            HWND hWndMDI;
            RECT rctClient;

            GetClientRect(hWnd, &rctClient);

            hWndMDI = GetDlgItem(hWnd, IDM_FILE_NEW);
            SetWindowPos(hWndMDI, NULL, 0, 0, rctClient.right,
rctClient.bottom, SWP_NOZORDER);
        }
        break;
        case WM_CLOSE:
            DestroyWindow(hWnd);
        break;
        case WM_DESTROY:
    }
}

```

```

        PostQuitMessage(0);
break;
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case IDM_FILE_NEW:
            CreateNewMDIChild(hWndChildFrame);
            break;
        case IDM_FILE_CLOSE:
            {
                HWND hChild =
(HWND)SendMessage(hWndChildFrame, WM_MDIGETACTIVE,0,0);
                if(hChild)
                {
                    SendMessage(hChild, WM_CLOSE, 0,
0);
                }
            }
            break;
        case IDM_FILE_EXIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0);
            break;
        case IDM_WINDOW_TILE:
            SendMessage(hWndChildFrame, WM_MDITILE,
0, 0);
            break;

        case IDM_WINDOW_CASCADE:
            SendMessage(hWndChildFrame,
WM_MDICASCADE, 0, 0);
            break;

        case IDM_WINDOW_ICONS:
            SendMessage(hWndChildFrame,
WM_MDIICONARRANGE, 0, 0);
            break;

        case IDM_WINDOW_CLOSE_ALL:
            {
                HWND hWndCurrent;

                do {
                    hWndCurrent =
(HWND)SendMessage(hWndChildFrame, WM_MDIGETACTIVE,0,0);
                    SendMessage(hWndCurrent,
WM_CLOSE, 0, 0);
                }while(hWndCurrent);
            }
            break;

        default:
            {
                if(LOWORD(wParam) >= StartChildrenNo)
                {
                    DefFrameProc(hWnd,
hWndChildFrame, WM_COMMAND, wParam, lParam);
                }
                else
                {

```

```

        HWND hWndCurrent =
        (HWND)SendMessage(hWndChildFrame, WM_MDIGETACTIVE,0,0);
        if( hWndCurrent )
        {
            SendMessage(hWndCurrent,
            WM_COMMAND, wParam, lParam);
        }
    }
    break;
    default:
        return DefFrameProc(hWnd, hWndChildFrame, Msg, wParam,
        lParam);
    }
    return 0;
}

LRESULT CALLBACK ChildWndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CREATE:
            break;

        case WM_MDIACTIVATE:
            {
                HMENU hMenu, hFileMenu;
                UINT EnableFlag;

                hMenu = GetMenu(hWndMainFrame);
                if(hwnd == (HWND)lParam)
                {
                    EnableFlag = MF_ENABLED;
                }
                else
                {
                    EnableFlag = MF_GRAYED;
                }

                EnableMenuItem(hMenu, 1, MF_BYPOSITION | EnableFlag);
                EnableMenuItem(hMenu, 2, MF_BYPOSITION | EnableFlag);

                hFileMenu = GetSubMenu(hMenu, 0);

                EnableMenuItem(hFileMenu, IDM_FILE_CLOSE, MF_BYCOMMAND |
                EnableFlag);
                EnableMenuItem(hFileMenu, IDM_WINDOW_CLOSE_ALL,
                MF_BYCOMMAND | EnableFlag);

                DrawMenuBar(hWndMainFrame);
            }
            break;

        default:
            return DefMDIChildProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}

```

```

BOOL CreateNewDocument(HINSTANCE hInstance)
{
    WNDCLASSEX WndClsEx;

    WndClsEx.cbSize          = sizeof(WNDCLASSEX);
    WndClsEx.style          = CS_HREDRAW | CS_VREDRAW;
    WndClsEx.lpfnWndProc    = ChildWndProc;
    WndClsEx.cbClsExtra     = 0;
    WndClsEx.cbWndExtra     = 0;
    WndClsEx.hInstance     = hInstance;
    WndClsEx.hIcon          = LoadIcon(NULL, IDI_WARNING);
    WndClsEx.hCursor        = LoadCursor(NULL, IDC_ARROW);
    WndClsEx.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
    WndClsEx.lpszMenuName   = NULL;
    WndClsEx.lpszClassName = ChildClassName;
    WndClsEx.hIconSm        = LoadIcon(NULL, IDI_WARNING);

    if(!RegisterClassEx(&WndClsEx))
    {
        MessageBox(NULL,
                   "There was a problem when attempting to create a document",
                   "Application Error",
                   MB_OK);

        return FALSE;
    }
    else
        return TRUE;
}

```