

Programming 2D Games

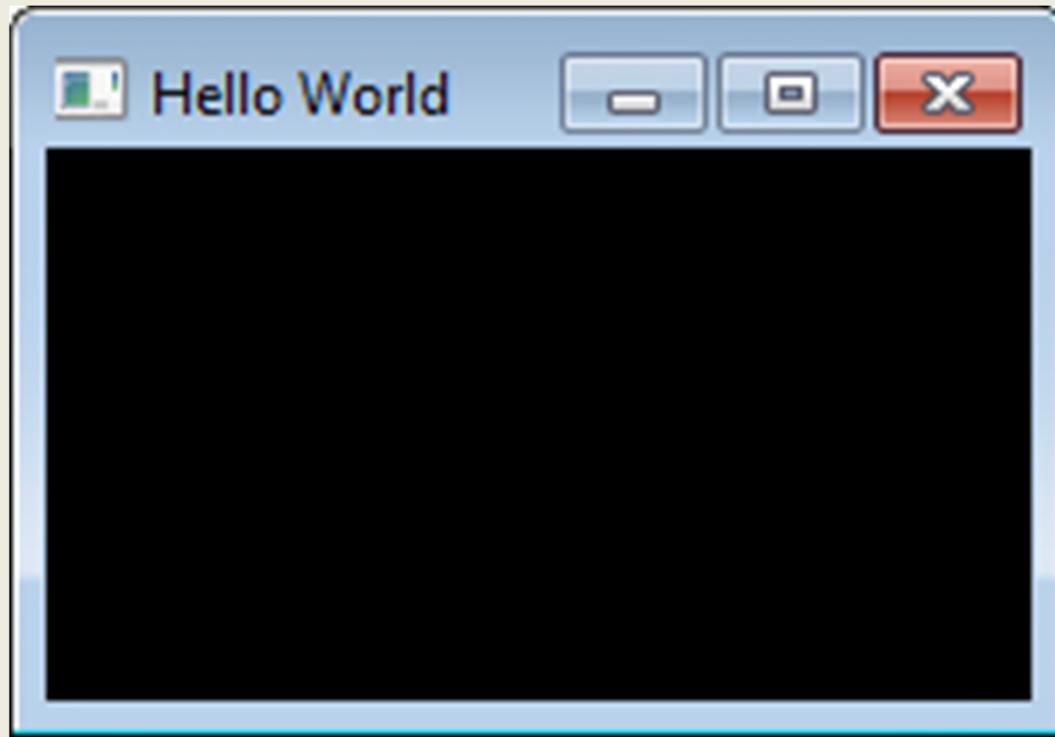
Chapter 2:

Windows Programming Fundamentals

Windows Programming Fundamentals

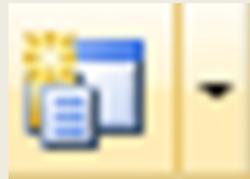
- Windows Application Programming Interface (API).
- The Windows API may also be referred to as WinAPI, Win32 API or just Win32.
- The API provides access to many of the inner workings of Windows.

“Hello World” Windows Style

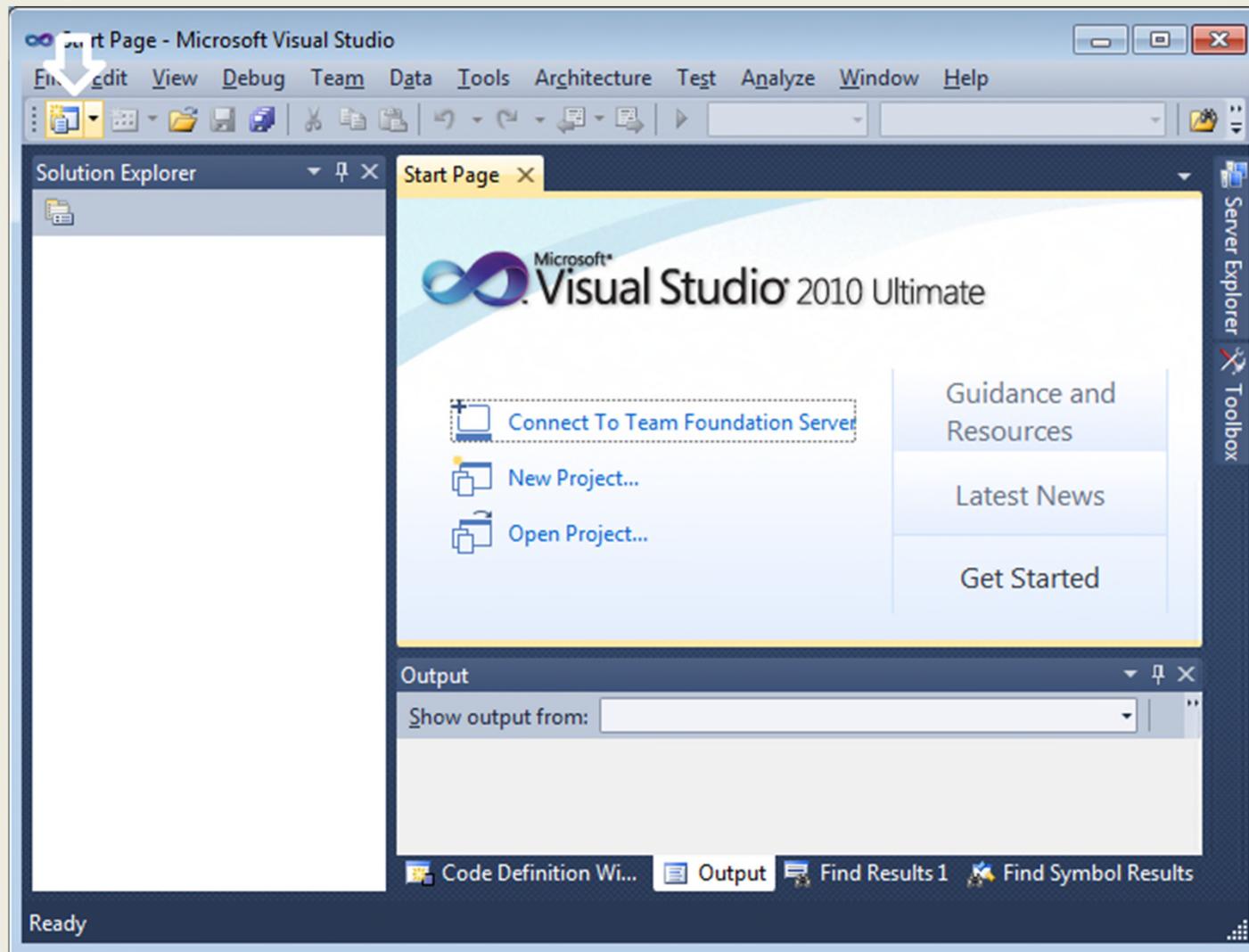


Getting Started with Visual Studio

- Create a new project by selecting *File* → *New* → *Project* from the menu or by clicking the “New Project” button on the toolbar



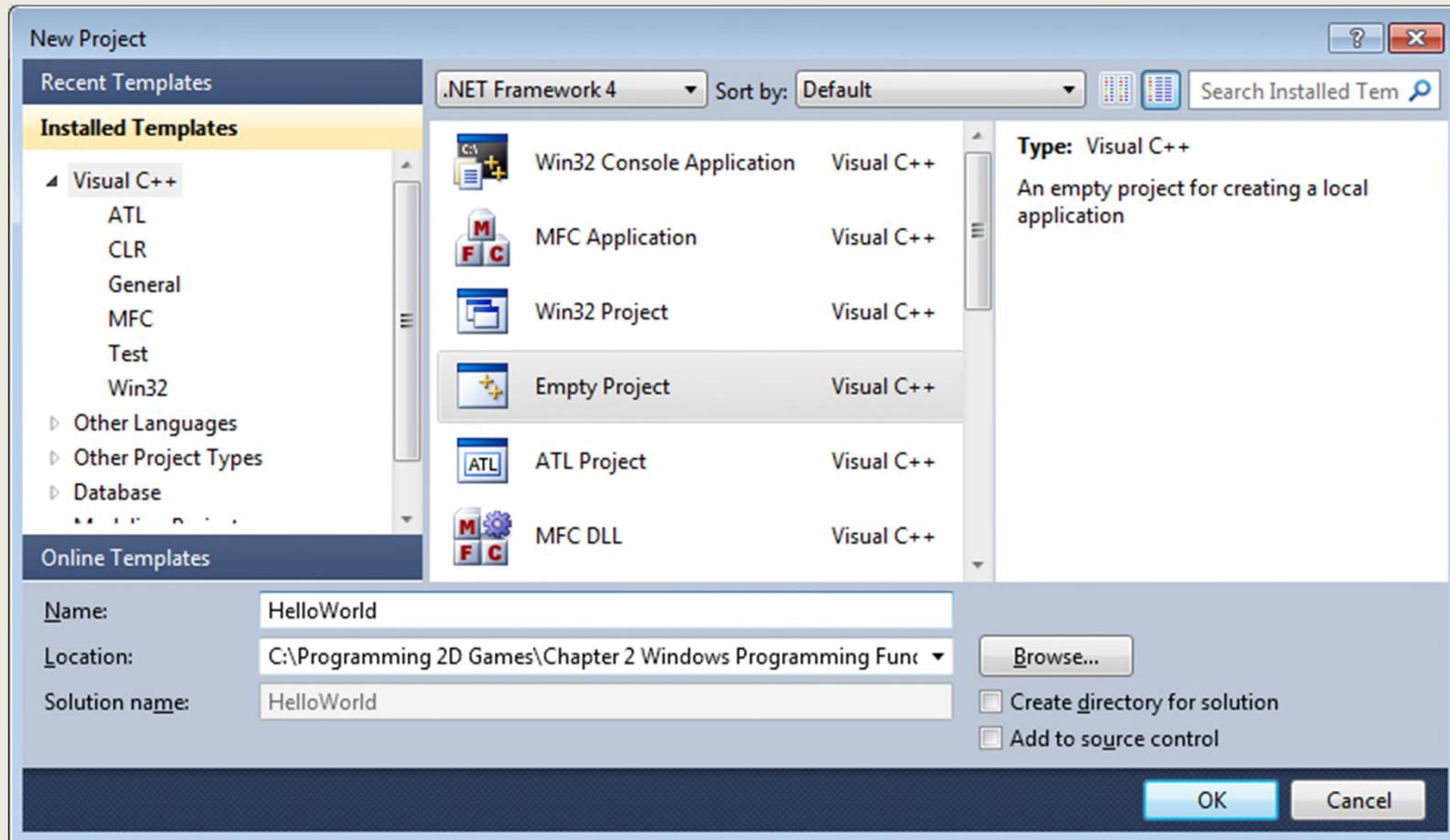
Getting Started with Visual Studio



Getting Started with Visual Studio

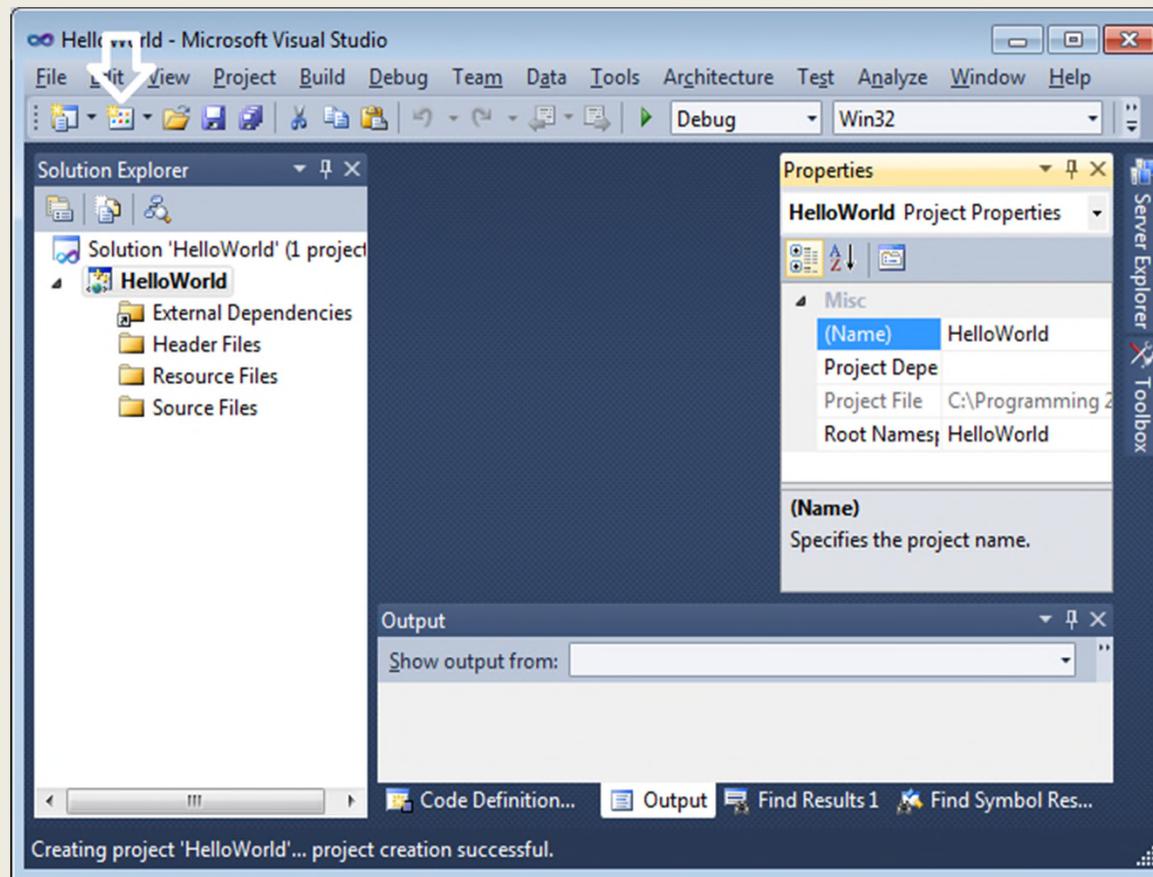
- Select Visual C++ as the project type in the left pane. In the center pane select “Empty Project”.
- Name the project “HelloWorld.” (The project name is also the name given to the executable file when the project is compiled.) The Solution name defaults to the project name.
- Clear the checkbox labeled “Create directory for solution.”
- Specify the location where the project should be created and click OK.

Getting Started with Visual Studio



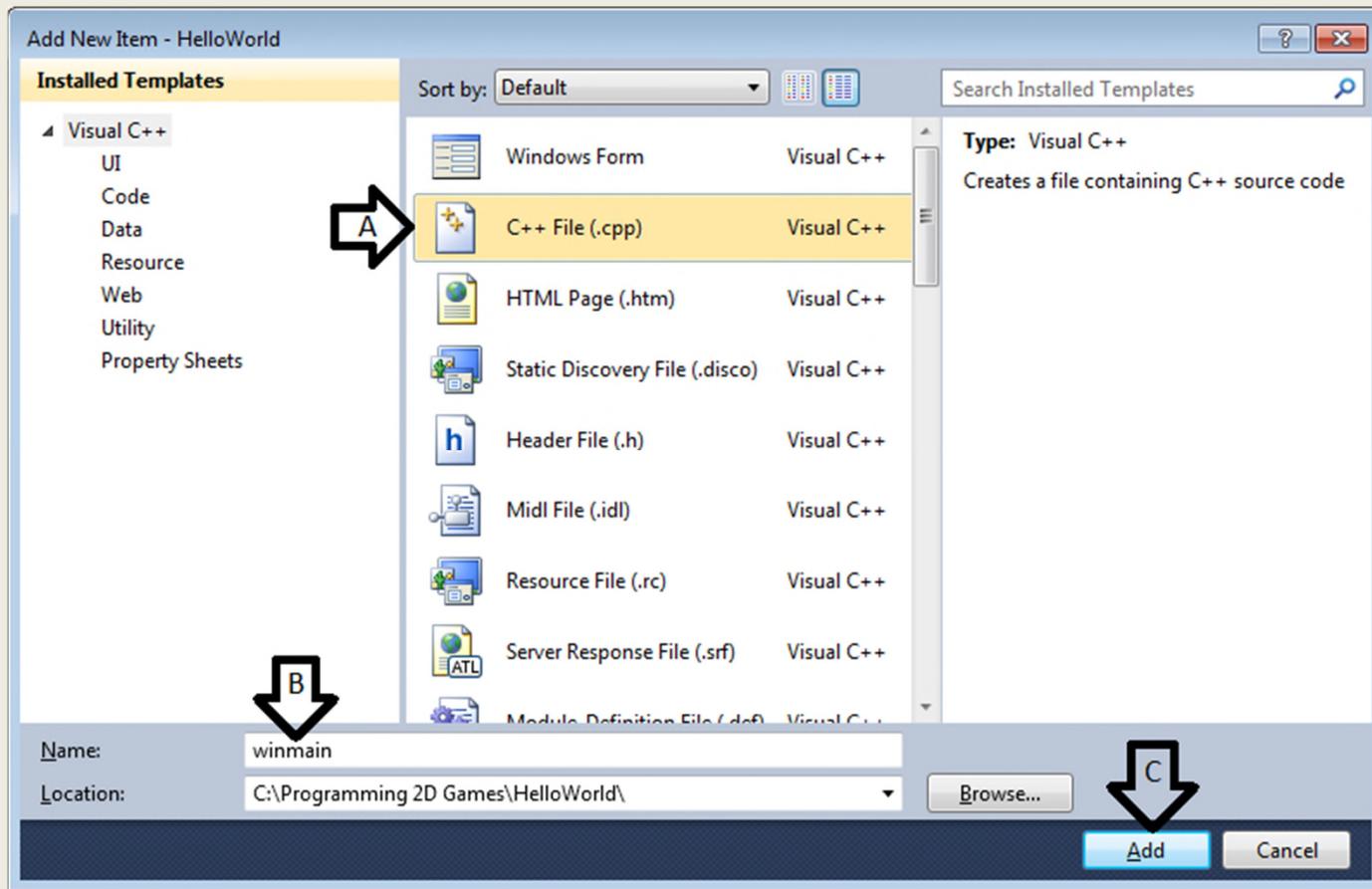
Getting Started with Visual Studio

- Add a source file to the empty project by clicking the “Add New Item” button on the toolbar.



Getting Started with Visual Studio

- Select “C++ File (.cpp)” in the center pane (A)
- Name the item “winmain” (B) and click the *Add* button (C).



“Hello World” Windows Style

- **WinMain** is the starting point of a Windows program.
- The windows.h header file is required.
- The directive `#define WIN32_LEAN_AND_MEAN` is used to prevent unwanted files from being included.

“Hello World” Windows Style

- The WinMain function:

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

- The return type is `int`.
- `WINAPI` is a calling convention that specifies parameter passing protocols.
- The `WinMain` parameters are typically not used to control the appearance of our window in game programs.

The Window Class

- A window class must be created before we can display our window.
- The window class defines features of the window.
- The window features are contained in a `WNDCLASSEX` structure.
- Once we have the structure configured to our liking we typically do not need to make any changes.

The Window Class

```
WNDCLASSEX wcx;  
HWND hwnd;  
// Parameters that describe the main window.  
wcx.cbSize = sizeof(wcx);           // Size of structure  
wcx.style = CS_HREDRAW | CS_VREDRAW; // Redraw if size changes  
wcx.lpfnWndProc = WinProc;          // Points to window procedure  
wcx.cbClsExtra = 0;                 // No extra class memory  
wcx.cbWndExtra = 0;                 // No extra window memory  
wcx.hInstance = hInstance;         // Handle to instance  
wcx.hIcon = NULL;  
wcx.hCursor = LoadCursor(NULL, IDC_ARROW); // Predefined arrow  
wcx.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH); // Background brush  
wcx.lpszMenuName = NULL;            // Name of menu resource  
wcx.lpszClassName = CLASS_NAME;    // Name of window class  
wcx.hIconSm = NULL;                 // Small class icon  
// Register the window class  
if (RegisterClassEx(&wcx) == 0)    // If error  
    return false;
```

Our message handler

The Window Class

- The Window class must be registered with Windows using the RegisterClassEx function.

```
// Register the Window class
// RegisterClassEx returns 0 on error
if (RegisterClassEx(&wcx) == 0)    // If error
    return false;
```

CreateWindow function

- The CreateWindow function is called to create the window.

```
HWND CreateWindow(  
    LPCTSTR    lpClassName,  
    LPCTSTR    lpWindowName,  
    DWORD      dsStyle,  
    int        x,  
    int        y,  
    int        nWidth,  
    int        nHeight,  
    HWND       hWndParent,  
    HMENU      hMenu,  
    HINSTANCE  hInstance,  
    LPVOID     lpParam  
);
```

CreateWindow function parameters

- `lpClassName`. A pointer to a NULL-terminated string containing the window class name. This name must match the name used in the `lpszClassName` member in the `CreateWindowClass` function.
- `lpWindowName`. The text that appears in the title bar.
- `dsStyle`. The style of window to create. Such as:
 - `WS_OVERLAPPEDWINDOW`. Creates a resizable window with the familiar controls.
 - `WS_OVERLAPPED`. Creates a fixed size window with no controls. This is the style we will most often use for windowed games.
 - `WS_EX_TOPMOST | WS_VISIBLE | WS_POPUP`: These are three styles combined with the OR `|` operator. This is the style we will use for full-screen games.

CreateWindow function parameters

- `x`, `y`. The coordinates of the top-left corner of the window.
- `nWidth`. The width of the window in pixels.
- `nHeight`. The height of the window in pixels.
- `hWndParent`. The parent window. Normally our games will not have a parent window.
- `hMenu`. The window menu.
- `hInstance`. The application identifier from the window class.
- `lpParam`. Additional window parameters.

Message Loop

- Windows communicates with our program by sending it messages.
- A loop in `WinMain` is used to check for messages.
- If our application is going to accept character input it needs to call the `TranslateMessage` function inside the message loop.
- `TranslateMessage` converts virtual-key messages into character messages.
- The messages are sent to our `WinProc` function for processing by the `DispatchMessage` function.

Message Loop

```
// Main message loop
int done = 0;
while (!done)
{
    // Check for Windows messages
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        // Look for quit message
        if (msg.message == WM_QUIT)
            done = 1;
        // Decode and pass messages on to WinProc
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;
```

Be sure to use
PeekMessage, not
GetMessage

WinProc Function

- The `WinProc` function is used to process messages.
- The name used for this function must match the name specified in the `WNDCLASSEX` structure.
- We respond to desired messages by placing code in `WinProc`.
- Any messages we ignore will be handled by Windows.
- The `WM_DESTROY` message is sent to our application when our window is being destroyed.
- `PostQuitMessage(0)` sends a `WM_QUIT` message to our program which ends the message loop in `WinMain`.

WinProc Function

```
//=====
// Window event callback function
//=====
LRESULT WINAPI WinProc( HWND hWnd, UINT msg,
                       WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            // Tell Windows to kill this program
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
```

Device Context

- Windows supports output to a variety of devices.
- The output is the same regardless of the output device.
- Output device independence is possible because of the *graphics device interface* (GDI).
- The GDI is a dynamic-link library that, together with a device driver, enables applications to output to different devices in the same manner.
- Access to an output device is done through a *device context* (DC).
- A DC is a structure that defines a graphics object and its properties. Windows created a DC when it created our window.

Keyboard Input with Windows API

- The two types of keyboard input used by games are:
 - Text. We want to know which character the user pressed.
 - Keyboard as game controller. We want to know which combination of keys is currently pressed.
- Windows sends several messages related to key presses.

WM_CHAR Message

- The WM_CHAR message is sent when a character is typed on the keyboard.
- To read the typed character we add a WM_CHAR message handler to our WinProc function.
- The wParam contains the character code.

WM_CHAR Message

```
LRESULT WINAPI WinProc( HWND hwnd, UINT msg,
                        WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            // Send WM_QUIT message
            PostQuitMessage(0);
            return 0;
        // A character was entered by the keyboard
        case WM_CHAR:
            // The character is in wParam
            switch (wParam) {
                // Process the character
```

WM_KEYDOWN, WM_KEYUP message

- The WM_KEYDOWN and WM_KEYUP messages allow us to use the keyboard like a giant game controller.
- Each time a key is pressed a WM_KEYDOWN message is sent.
- When the key is released a WM_KEYUP message is sent.
- The virtual key code is contained in wParam.
- We save the state of each key as *true* or *false* in the vkKeys array.

WM_KEYDOWN, WM_KEYUP message

- Virtual key codes are different from the character codes we get from WM_CHAR messages.
- Each key on the keyboard has an assigned virtual key code.
- For a complete list of virtual key codes look in the WinUser.h file.
- Virtual key constants begin with the prefix VK_
- The arrow keys are: VK_LEFT, VK_UP, VK_RIGHT, VK_DOWN.
- To test for a right arrow press use:
`if (vkKeys[VK_RIGHT])`

Using a Mutex to Prevent Multiple Instances

- If a user runs multiple instances of our game undesirable results may occur.
- We can use a *mutex* to prevent multiple instances of our game from running.
- A mutex is an object that may be owned by only one thread at a time.
- If our game creates a mutex then any subsequent attempts to create the same mutex will fail.
- The mutex is created with a call to the `CreateMutex` function.

Using a Mutex to Prevent Multiple Instances

```
//=====
// Checks for another instance of the current application
// Returns: true if another instance is found
//         false if this is the only one
//=====
bool AnotherInstance()
{
    HANDLE ourMutex;
    // Attempt to create a mutex using our unique string
    ourMutex = CreateMutex(NULL, true,
        "Use_a_different_string_here_for_each_program");
    if (GetLastError() == ERROR_ALREADY_EXISTS)
        return true;           // Another instance was found

    return false;             // We are the only instance
}
```

Multitasking in Windows

- Multiple applications and internal processes may be running in Windows at any given time.
- Our game will be given access to a processor for brief amounts of time (on the order of 1 to 20 milliseconds).
- This presents a challenge when we want our game to have smooth animation. We will see how to overcome this challenge in later chapters.

Chapter Review

- `WinMain` is the starting point for a Windows application.
- The `WNDCLASSEX` structure describes our window class.
- The `CreateWindow` function creates the window.
- Windows sends our program messages.
- The `WinProc` function processes Windows messages.
- The `WM_CHAR` message is used to read character input.
- `WM_KEYDOWN` and `WM_KEYUP` messages are used to read the keyboard like a game controller.
- A `Mutex` may be used to prevent multiple instances or a program from running.
- Windows uses multitasking to run programs in short time slices.