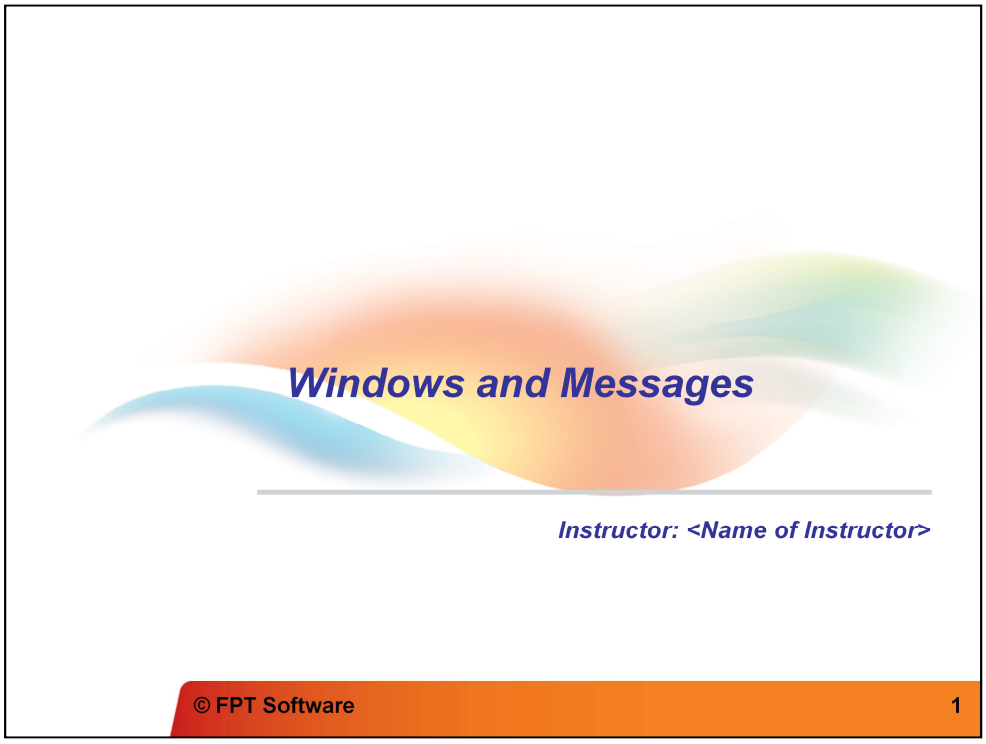


Latest updated by: HanhTT1

Agenda

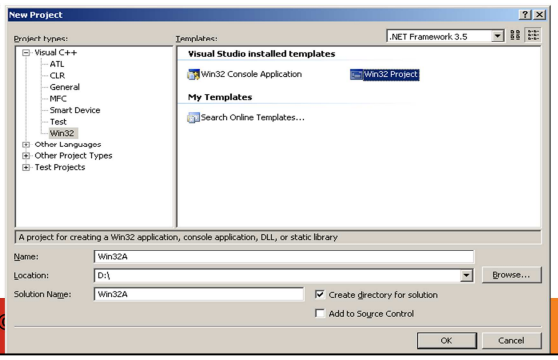
- Getting started
- Windows Programming Model
- Messages
- Hook

Getting started

- Windows enviroment
- The first C program in Windows enviroment

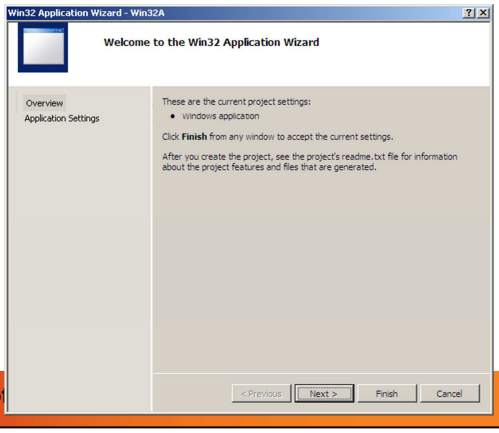
First program

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



First program - cont

- In the location, type the path where the application should be stored, such as D:
- In the Name edit box, type the name of the application as **Win32A** and click OK



Structure of Windows program

- WinMain() function
- Windows Procedure
- Other functions

Basic program flow

- Register window class
- Create window from window class
- Show window
- Create message loop for getting messages from operation system, of from application to itself and hanldling these messages

The main window class

- WinMain tasks:
  - ✓ Declare window class
  - ✓ Register window class
  - ✓ Create and display
  - ✓ Get notifications

```
int APIENTRY WinMain(HINSTANCE  hInstance,
                     HINSTANCE  hPrevInstance,
                     LPTSTR     lpCmdLine,
                     int        nCmdShow)
```

- *hInstance*, is a handle to the instance of the program you are writing
- *hPrevInstance*, is used if your program had any previous instance. If not, this argument can be ignored
- *lpCmdLine*, is a string that represents all items used on the command line to compile the application
- *nCmdShow*, controls how the window you are building will be displayed

Window class

- Windows defined window class as a struct
- Name of struct : **WNDCLASS**
- List fields of **WNDCLASS** struct :
  - **UINT** *style*
  - **WNDPROC** *lpfnWndProc*
  - **int** *cbClsExtra*
  - **int** *cbWndExtra*
  - **HANDLE** *hInstance*
  - **HICON** *hIcon*
  - **HCURSOR** *hCursor*
  - **HBRUSH** *hbrBackground*
  - **LPCTSTR** *lpszMenuName*
  - **LPCTSTR** *lpszClassName*

Window Extra-Memory

- When an application has been launched and is displaying on the screen, which means an instance of the application has been created, the operating system allocates an amount of memory space for that application to use. If you think that your application's instance will need more memory than that, you can request that extra memory bytes be allocated to it. Otherwise, you can let the operating system handle this instance memory issue and initialize the *cbWndExtra* member variable to 0

```
wndClass.cbClsExtra = 0;
```

The Main Window's Style

- The *style* member variable specifies the primary operations applied on the window class. The actual available styles are constant values. For example, if a user moves a window or changes its size, you would need the window to be redrawn to get its previous characteristics. To redraw the window horizontally, you would apply the **CS_HREDRAW**. In the same way, to redraw the window vertically, you can apply the **CS_VREDRAW**.
- The styles are combined using the bitwise OR (|) operator. The **CS_HREDRAW** and the **CS_VREDRAW** styles can be combined and assigned to the style member variable

```
wndClass.style = CS_HREDRAW | CS_VREDRAW
```

The Application's Instance

- Creating an application is equivalent to creating an instance for it. To communicate to the **WinMain()** function that you want to create an instance for your application, which is, to make it available as a resource, assign the **WinMain()**'s *hInstance* argument to your **WNDCLASS** variable

```
wndClass.hInstance = hInstance
```

Message Processing

- The name of the window procedure must be assigned to the *lpfnWndProc* member variable of the **WNDCLASS** or **WNDCLASSEX** variable

```
wndClass.lpfnWndProc = WndProc

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;
}
```

The Application Main Icon

- An icon can be used to represent an application in My Computer or Windows Explorer. To assign this small picture to your application, you can either use an existing icon or design your own. To make your programming a little faster, Microsoft Windows installs a few icons. The icon is assigned to the *hIcon* member variable using the **LoadIcon()** function

  wndClass.hIcon = HICON LoadIcon(HINSTANCE *hInstance*,

  LPCTSTR *lpIconName*);

- The *hInstance* argument is a handle to the file in which the icon was created
- The *lpIconName* is the name of the icon to be loaded. This name is added to the resource file when you create the icon resource
- If you designed your own icon (you should make sure you design a 32x32 and a 16x16 versions, even for convenience), to use it, specify the *hInstance* argument of the **LoadIcon()** function to the instance of your application. Then use the **MAKEINTRESOURCE** macro to convert its identifier to a null-terminated string. This can be done as follows:

  WndCls.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_STAPLE));

Introduction to Cursors

- A cursor is used to locate the position of the mouse pointer on a document or the screen. To use a cursor, call the Win32 **LoadCursor()** function:

  HCURSOR LoadCursor(HINSTANCE *hInstance*, LPCTSTR *lpCursorName*)

- Example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
                    WNDCLASSEX WndClsEx;
                    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.hInstance = hInstance;
                    WndClsEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
                    return 0;
}
```

The Application's Main Menu

- If you want the window to display a menu, first create or design the resource menu (we will eventually learn how to do this). After creating the menu, assign its name to the *lpszMenuName* name to your **WNDCLASS** or **WNDCLASSEX** variable. Otherwise, pass this argument as NULL
- Example:

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
                    WNDCLASSEX WndClsEx;
                    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 = GetStockObject(WHITE_BRUSH);
                    WndClsEx.lpszMenuName = NULL;
                    WndClsEx.hInstance = hInstance;
                    WndClsEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
                    return 0;
}
```

The Window's Class Name

- To create a window, you must provide its name as everything else in the computer has a name. The class name of your main window must be provided to the *lpszClassName* member variable of your **WNDCLASS** or **WNDCLASSEX** variable. You can provide the name to the variable or declare a global null-terminated string
- Example:

```
ATOM MyRegisterClass(HINSTANCE hInstance) {
        WNDCLASSEX wcex;
        wcex.cbSize = sizeof(WNDCLASSEX);
        wcex.style                      = CS_HREDRAW | CS_VREDRAW;
        wcex.lpfnWndProc                = WndProc;
        wcex.cbWndExtra                 = 0;
        wcex.hInstance                  = hInstance;
        wcex.hIcon                      = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WIN32A));
        wcex.hCursor                    = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground              = (HBRUSH)(COLOR_WINDOW+1);
        wcex.lpszMenuName               = MAKEINTRESOURCE(IDC_WIN32A);
        wcex.lpszClassName              = szWindowClass;
        wcex.hIconSm                    = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
        return RegisterClassEx(&wcex);
}
```

Register window class

- Declare an struct variable that has type is **WNDCLASS** like below :
  - WNDCLASS wndClass;
- Set value for window class's fields
  - wndClass.style = CS_HREDRAW | CS_VREDRAW
  - **wndClass.lpfnWndProc = WndProc;**
  - wndClass.cbClsExtra = 0;
  - wndClass.cbWndExtra = 0;
  - wndClass.hInstance = hInstance;
  - wndClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  - wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
  - wndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
  - wndClass.lpSzMenuName = NULL;
  - **wndClass.lpSzClassName = szAppName;**

Register window class

- Register window class :

```
if (!RegisterClass( &wndClass))  {
      MessageBox ( NULL, TEXT("This program requires Windows NT")
                        , szAppName, MB_ICONERROR);

      return 0;
}
```

Create Window

- Window class defines common properties for one window. Base on window class, we could create different windows by call Win32 APPI : **CreateWindow()** function
- **HWND CreateWindow(**
  - **LPCTSTR** *lpClassName*,
  - **LPCTSTR** *lpWindowName*,
  - **DWORD** *dwStyle*,
  - **int** *x*,
  - **int** *y*,
  - **int** *nWidth*,
  - **int** *nHeight*,
  - **HWND** *hWndParent*,
  - **HMENU** *hMenu*,
  - **HANDLE** *hInstance*,
  - **PVOID** *lpParam* **);**

Show Window

- After create window, window is not appeared on screen, we must call **ShowWindow()** function for display window on screen :
- **ShowWindow(hWnd, iCmdShow);**
- For update client area of window, we must call **UpdateWindow()** function for re-draw client area of window on screen :
- **UpdateWindow(hWnd);**

Messages Loop

- After window has been created, displayed on screen, application is ready for handle all of events are occurred by interaction between user and application
- When an event occurs, Windows translates event to a “message” then place this message in application’s message queue
- For handle events, application must get messages from message queue then send messages to message handlers that are defined in window procedure

Message Loop

- Get messages from message queue :

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

- The **MSG** struct contains message information from message queue

```
typedef struct tagMSG {
HWND    hwnd ;      // handle of windows that get message
UINT                message ;    // message identifier
WPARAM  wParam ;    // additional information about message
LPARAM  lParam ;    // additional information about message
. . .
} MSG;
```

- For wParam and lParam, the exact meaning depends on the value of the message member. For WM_LBUTTONDOWN it is the state of the Ctrl or Shift keys, and the mouse coordinates.

Window's Message Decoding (1)

- Once a window has been created, the user can use it. This is done by the user clicking things with the mouse or pressing keys on the keyboard. A message that a window sends is received by the application. This application must analyze, translate, and decode the message to know what object sent the message and what the message consists of. To do this, the application uses the **GetMessage()** function.

  BOOL GetMessage(LPMSG *lpMsg*, HWND *hWnd*, UINT *wMsgFilterMin*, UINT *wMsgFilterMax*)

  - ✓ *lpMsg* argument is a pointer to the **MSG** structure
  - ✓ *hWnd argument identifies which window sent the message. If you want the messages of all windows to be processed, pass this argument as NULL.*
  - ✓ *wMsgFilterMin* and the *wMsgFilterMax* arguments specify the message values that will be treated. If you want all messages to be considered, pass each of them as 0

Window's Message Decoding (2)

- Once a message has been sent, the application analyzes it using the **TranslateMessage()** function. This function takes as argument the **MSG** object that was passed to the **GetMessage()** function and analyzes it. If this function successfully translates the message, it returns TRUE. If it cannot identify and translate the message, it returns FALSE

  BOOL TranslateMessage(CONST MSG *_lpMsg_)

- Once a message has been decoded, the application must send it to the window procedure. This is done using the **DispatchMessage()** function. This **DispatchMessage()** function sends the *lpMsg* message to the window procedure. The window procedure processes it and sends back the result, which becomes the return value of this function

  LRESULT DispatchMessage(CONST MSG *_lpMsg_)

Window procedure

- Contains message hanlder for handling event occurs of application
- Prototype :

```
LRESUTL CALLBACK WndProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARMA   lParam)
```

Window procedure

- Window procedure is always called from Windows
- One program could call window procedure by call Win32 API function : **SendMessage()**
- Basic flow of message handling in window procedure :

```
Switch (Message id) {
  Case WM_CREATE :
        // handle message WM_CREATE
        Return 0;
  Case [Other message] :
        // hanlde message
        Return 0;
  Case WM_PAINT :
        // hanlde message WM_PAINT
        Return 0;
  Case WM_DESTROY :
        // hanlde message WM_DESTROY
        Return 0;
}
Return DefWindowProc(hWnd, iMsg, wParam, lParam);
```

Messages, Messages, and Messages

Message	***Sent When***
WM_CHAR	A character is input from the keyboard.
WM_COMMAND	The user selects an item from a menu, or a control sends a notification to its parent.
WM_CREATE	A window is created.
WM_DESTROY	A window is destroyed.
WM_LBUTTONDOWN	The left mouse button is pressed.
WM_LBUTTONUP	The left mouse button is released.
WM_MOUSEMOVE	The mouse pointer is moved.
WM_PAINT	A window needs repainting.
WM_QUIT	The application is about to terminate.
WM_SIZE	A window is resized.

Sending Messages

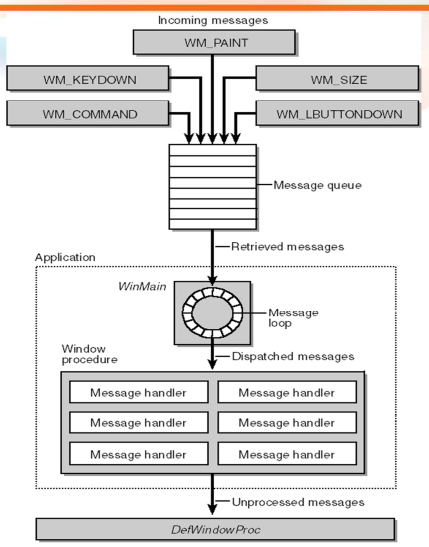
- The messages we have used so far belong to specific events generated at a particular time by a window. Sometimes in the middle of doing something, you may want to send a message regardless of what is going on. This is made possible by a function called **SendMessage()**
  LRESULT SendMessage(HWND *hWnd*,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

- ❑ 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

- 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*)

Summary


Incoming messages
WM_PAINT
WM_KEYDOWN
WM_SIZE
WM_COMMAND
WM_LBUTTONDOWN
Message queue
Retrieved messages
Application
WinMain
Message loop
Window procedure
Dispatched messages
Message handler
Message handler
Message handler
Message handler
Message handler
Message handler
Unprocessed messages
DefWindowProc

Hooks

- Overview about hook
- Hook Types
- Hook Scopes
- Win32 API Related to Hook
- Sample code to implement hook
- Hook Example

Overview about hook

- Hook is a technique to intercept events, such as messages, mouse actions and keystrokes before these events are sent to windows procedure
- Hook procedure: a function that intercepts a particular type of event is known as hook procedure. After hook procedure processes events, it can modify or discard the event.
- Hook chain: A *hook chain* is a list of pointers to special, application-defined callback functions called *hook procedures*
- Each type of hook enable an application monitor a different aspect of the system's message-handling mechanism
  Refer to below link for detail about each hook type and corresponding hook procedure
  http://msdn.microsoft.com/en-us/library/windows/desktop/ms644959(v=vs.85).aspx
- Hooks tend to slow down the system because they increase the amount of processing the system must perform for each message => Only use when necessary.

A *hook chain* is a list of pointers to special, application-defined callback functions called *hook procedures*. When a message occurs that is associated with a particular type of hook, the system passes the message to each hook procedure referenced in the hook chain, one after the other. The action a hook procedure can take depends on the type of hook involved. The hook procedures for some types of hooks can only monitor messages; others can modify messages or stop their progress through the chain, preventing them from reaching the next hook procedure or the destination window

Hook Types

Below is some common hook type

- **WH_CALLWNDPROC**: monitor messages sent to window procedures
- **WH_GETMESSAGE**: monitor messages about to be returned by the GetMessage or PeekMessage function
- **WH_KEYBOARD_LL:** monitor keyboard input events about to be posted in a thread input queue
- **WH_KEYBOARD**: monitor message traffic for WM_KEYDOWN and WM_KEYUP messages about to be returned by the GetMessage or PeekMessage function
- **WH_MOUSE_LL**: monitor mouse input events about to be posted in a thread input queue
- **WH_MOUSE:** monitor mouse messages about to be returned by the GetMessage or PeekMessage function.

Hook Scopes

- Global Hooks, process all input of the appropriate type for the entire OS and must be in a DLL
- Thread Hooks, process all input of the appropriate type for that process or thread

Name	Scope	Hook procedure
WH_CALLWNDPROC	Global/Thread	CallWndProc
WH_GETMESSAGE	Global/Thread	GetMsgProc
WH_KEYBOARD_LL	Global	LowLevelKeyboardProc
WH_KEYBOARD	Global/Thread	KeyboardProc
WH_MOUSE_LL	Global	LowLevelMouseProc
WH_MOUSE	Global/Thread	MouseProc

Win32 API Related to Hook

Hook procedure

```
LRESULT CALLBACK HookProc( int nCode, WPARAM wParam,
    LPARAM lParam ) {
    ..........................
}
```

Below is some common hook type

- **SetWindowsHookEx**: Installs an application-defined hook procedure into a hook chain
- **UnhookWindowsHookEx**: Removes a hook procedure installed in a hook chain
- **CallNextHookEx**: Passes the hook information to the next hook procedure in the current hook chain

Note:

- –Don't use the non-Ex versions, as they are for windows 3.x
- –Include windows.h header when use.

Win32 API Related to Hook

SetWindowsHookEx

–Return handle to the hook procedure when success

–If the function fails, the return value is **NULL**

```
HHOOK SetWindowsHookEx (
        int             idHook,         //type of hook procedure
        HOOKPROC        lpfn,           //pointer to the hook procedure
        HINSTANCE       hMod,           //handle to the DLL containing the hook procedure
        DWORD           dwThreadId      //thread id
);
```

•The *hMod* parameter must be set to NULL if

– *dwThreadId* specifies a thread created by the current process

– *lpfn* is within the code associated with the current process

•If *lpfn* in a dll, hMod must be the instance of the DLL

Win32 API Related to Hook

UnhookWindowsHookEx

- Return nonzero to the hook procedure when success
- Return zero when function fails

```
BOOL UnhookWindowsHookEx (
        HHOOK   hhk         //handle to the hook to be removed
);
```

•Hooks don’t attach and detach immediately after SetWindowsHookEx and UnhookWindowsHookEx. Because windows hooks are all about messages, the hook isn't really mapped or unmapped until an adequate event happens.

Win32 API Related to Hook

CallNextHookEx

- If you want to have windows continue processing the hook, your processing function should return this function
- To discard return 0

```
LRESULT CallNextHookEx (
    HHOOK     hhk,        //just pass NULL for it
    int       nCode,      //hook code
    WPARAM    wParam,     //value passed to the current hook procedure
    LPARAM    lParam      //value passed to the current hook procedure
);
```

Step to implement hook

- Step 1: Create hook procedure. Prototype of hook procedure is followed hook type. Content of hook procedure is as below

```
// Hook procedure
LRESULT CALLBACK [FUNCTION_NAME]( int nCode, WPARAM wParam, LPARAM lParam )
{
    //if press keyboard
    if ( nCode == [CODE] )
    {
        ...
    }
    //call next Hook procedure in Hook chain or just return 0 for discard
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}
```

- [FUNCTION_NAME] : is name of hook procedure
- [CODE] : is hook code

Step to implement hook

- Step 2: Implement method to install hook. Set [THREAD_ID] to NULL if want to create global hook

```
//Install hook
void InstallHook(HINSTANCE [DLL_HANDLE])
{
    //Global hook
    [HOOK_HANDLE] = SetWindowsHookEx([HOOK_TYPE], [HOOK_PROCEDURE], [DLL_HANDLE], NULL);
    //Thread hook use function from library
    [HOOK_HANDLE] = SetWindowsHookEx([HOOK_TYPE], [HOOK_PROCEDURE], [DLL_HANDLE], [THREAD_ID]);
}
```

- [HOOK_HANDLE] : is kept to use when released hook
- [HOOK_PROCEDURE] : is address of hook procedure
- [DLL_HANDLE] : is handle of dll that contains hook procedure
- [THREAD_ID] : is thread id of thread want to hook

Step to implement hook

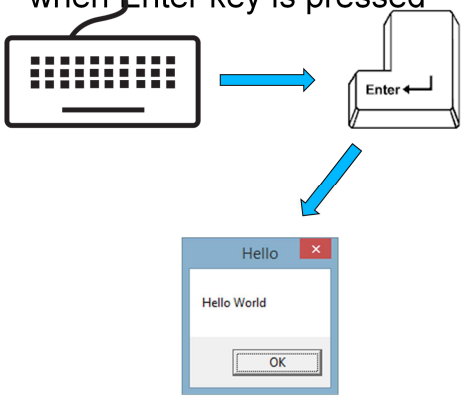
- Step 3: Implement method to release hook

```
//Release Hook
void ReleaseHook()
{
    UnhookWindowsHookEx([HOOK_HANDLE]);
}
```

- [HOOK_HANDLE] : is handle fo hook want to release

Example

- Example: Install global hook to show message box when Enter key is pressed



Example

- Step 1: Create a dll (Ex: TestHook.dll) to contain
  - hook procedure
  - Method to install hook
  - Method to release hook
- Step 2: Create a MFC dialog, in OnInitDialog method, load TestHook.dll and call method to install hook
- Step 3: On OnDestroy method of dialog application, call method to release hook

Example

Sample code of TestHook.dll

```
// Hook procedure
LRESULT CALLBACK InputKey( int nCode, WPARAM wParam, LPARAM lParam )
{
    //if press keyboard
    if ( nCode == HC_ACTION )
    {
        //if press Enter key
        if( wParam == 13 )
        {
            MessageBox( NULL , _T("Hello World") , _T("Hello") , NULL);
            return 0;
        }
    }
    //call next Hook procedure in Hook chain or just return 0 for discard
    return CallNextHookEx(NULL, nCode, wParam, lParam);
}

//Install hook
void InstallHook(HINSTANCE hDll)
{
    g_hHook = SetWindowsHookEx(WH_KEYBOARD, InputKey, hDll, NULL);
}

//Release Hook
void ReleaseHook()
{
    UnhookWindowsHookEx(g_hHook);
}
```

Example

- Sample code of MFC application

```
BOOL CTestHookAppDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    //Get handle of TestHook dll
    HINSTANCE hDll = GetModuleHandle(_T("TestHook"));
    //Install hook
    InstallHook(hDll);

    return TRUE;
}

void CTestHookAppDlg::OnDestroy()
{
    //Release hook
    ReleaseHook();
}
```

Hungary notation

Prefix	*Data Type*
b	BOOL
c or *ch*	char
clr	COLORREF
cx, cy	Horizontal or vertical distance
dw	DWORD
h	Handle
l	LONG
n	int
p	Pointer
sz	Zero-terminated string
w	WORD
wnd	CWnd
str	CString
m_	**class member variable**

Note:
Prefixes can be combined:
pszName
m_nAge


Q&
A