RogerVillela

Journal

Volume 0x0001

RogerVillela Journal – Volume 0x0001

INTRODUCTION

The **RogerVillela Journal (RVJ Services)** is a group of technical services with educational purpose. Among these professional services we have this publication about Microsoft Windows programming environment, concepts and the use of APIs. The purpose of this publication is to put on the hands of the students, hobbyists, and professional software engineer's around the world, practical and objective knowledge of intermediate and advanced levels about Microsoft Windows operating system environment. Technical materials for Microsoft Windows native programming are using C, C++ and Assembly programming languages, and are based on Intel IA-32 and Intel 64 (x64) hardware architectures. The technical materials about Common Language Runtime (CLR) are using C# programming language, C++/CLI projection, and MSIL. Technical materials about Windows Runtime (WinRT) execution environment are using C++/CX projection, C++ programming language, and C++/WinRT. The Microsoft development tools used are Microsoft Visual Studio integrated development environment (IDE) and Microsoft Windows SDK tools. For the IDE are used the sample solutions (.sln) and respective sample projects (.*proj). The Intel Parallel Studio with Intel C++ tools are also used in two forms, integrated with Microsoft Visual Studio and via command line. Initially, the knowledge areas of Microsoft Windows operating system covered by **RogerVillela Journal (RVJ)** are Windows Architecture and Engineering implementation, with Memory Management, Processes and Threads, and Debugging resources having a special attention. As C, C++ and Assembly programming languages are used for the sample projects, publications also have technical information about them, about C++/CLI projection for CLR execution environment and C++/CX projection for the WinRT execution environment, when applicable.

by Roger Villela

**Author and Technical Educator**

RogerVillela Journal

ORGANIZATION

Within each of this publication we have lessons or experiments for the use in the process of learnings about the Microsoft Windows software development using C++ programming language for who of you that are exploring how is the inner workings of these two technological contexts. We will learn that, despite the common perception that the only commercial valid investment is on most recent Microsoft Windows implementation versions, this is not the real life. We will see that even recent developed Application Programming Interfaces (API's) was made with support for Microsoft Windows 2000 or Microsoft Windows XP as the minimum implementation version that is supported by the API. The important commercial lesson here is that we have a broadly spectrum of Microsoft Windows implementation versions that we can offer commercial services to and to obtain the return of our investments and be more profitable for our life plans, that certainly go through the professional area. Together, the C++ programming language and Microsoft Windows API put available these commercial opportunities.
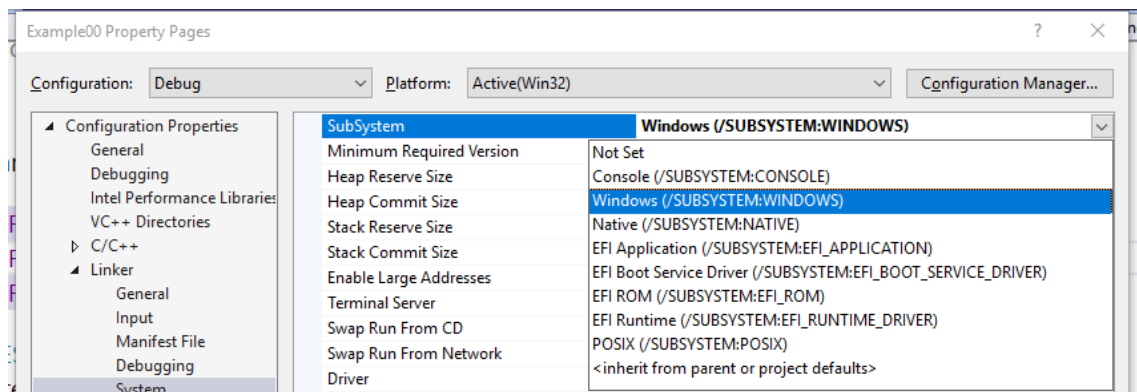
SOLUTIONS AND PROJECTS

All code available are sample projects for demonstrative and educational purposes, and are used gradually on presentation of concepts, and technology resources. These sample code and sample projects are gradually changed and often according to their purpose in commercial products such as courses and sequence of free or marketed publications, such as books and changes in technologies.

Within this publication we have two sample solutions, the **SecureZeroMemory** and the **IsWindows7SP1OrGreater**. The **SecureZeroMemory** sample solution has one sample project named **Example00**, for the *Microsoft Windows Console subsystem*.
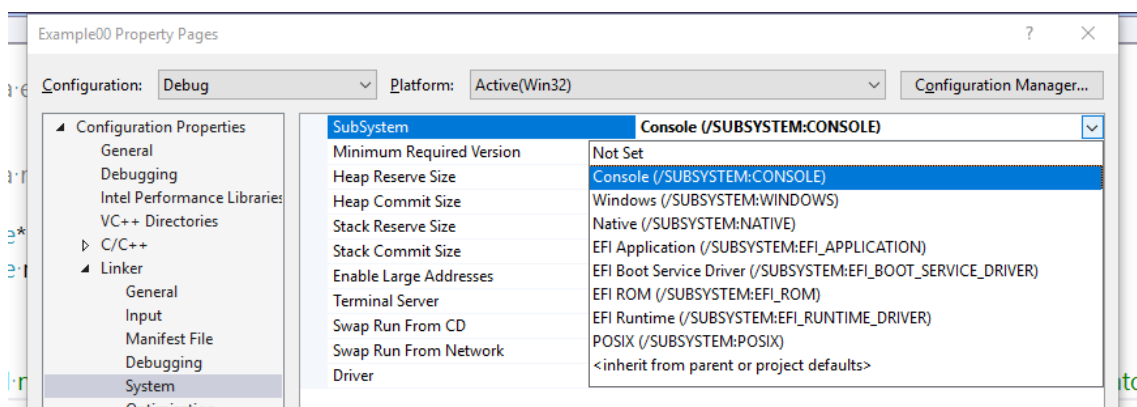
The **IsWindows7SP1OrGreater** sample solution has one sample project named **Example00** for *Microsoft Windows subsystem (GUI and DLL)*.

For the sample solutions and sample projects, this publication is using Microsoft Visual Studio 2019 and will update to new versions of Microsoft Visual Studio product as soon as the RTM be available. Despite of this fact, most of source code in sample projects compiles in earlier versions of Microsoft C++ compiler, with changes where applicable and with the appropriated configured development environment. The source code files are created to shows Microsoft Windows features and details about the inner workings of it, with the practical and applicable purpose of expanding the knowledge of the people interested in this type of details, even if you are not a regular C++ programming language or direct Windows API customer. For example, features of C++ programming language are used and explained as appropriated, in more or less details, when in any source code file. One typical example is the use of the constant expression qualifier **constexpr** (C++11/C++14) that is supported by modern implementations of C++ compilers. When offering commercial support services for an earlier implementation version of Microsoft C++ compiler that is based on a source code that is using the constant expression qualifier **constexpr**, we will need to do adjustments in our source code base so that it can offer support for both, the earlier Microsoft C++ compiler implementation version that does not offer the support for the constant expression qualifier **constexpr**, and more modern Microsoft C++ implementation version that offers support for the constant expression qualifier **constexpr**. This possible scenario of the uses of the keywords **constexpr** and **const** should not be made via something like "text copy and replace" in anyway. We should use, for example, the **#define** macro directive and create replacement based on business rules, such as the Microsoft C++ implementation version. The following excerpt of code shows an example of this possibility using **#define** directive in conjunction with a checking of the Microsoft Visual C++ implementation version and take a decision about the use of keywords **constexpr** or **const** when using Microsoft Visual Studio 2015 / Microsoft Visual C++ 2015 (version 1900) the minimum version being used for compilation. Using the automatically defined **_MSC_VER** macro we can check the Microsoft Visual C++ implementation version and define compatibility rules on our source code files:

```
#if _MSC_VER  >= 1900

#define CONSTEXPR

#if defined(CONSTEXPR)
#define CONST_OR_CONSTEXPR constexpr
#else
#define CONST_OR_CONSTEXPR const
#endif

#endif
```

So, this kind of information is also available within my commercial publications such as my books or within the source code file in most cases because it is more practical.

PATH(S)

- o <install_folder>\RVJ\Platforms\Windows\Code

SOLUTIONS

- SYSTEM SERVICES - MEMORY MANAGEMENT - GENERAL

  - o WE – Windows\Windows.System.MemoryManagement\General\

    - C

      - CopyMemory\CopyMemory.sln

    - M

      - MoveMemory\MoveMemory.sln

    - S

      - SecureZeroMemory\SecureZeroMemory.sln

- SYSTEM SERVICES – INFORMATION – VERSION HELPER FUNCTIONS

  - o WE – Windows\Windows.System.Information\

    - I

      - IsWindows7SP1OrGreater\IsWindows7SP1OrGreater.sln

HEADER FILES AND MICROSOFT WINDOWS API

NOTE: All information for this topic is based on the header files of the Microsoft Windows SDK 10.0.17763.132 or more recent.

Not every person working with C or C++ programming language like the idea of "include file", but it is a fact.

In C programming language the **.h extension** is commonly used as part of the name of an "include file", but this is not a rule, it is a convention described in the programming language specification. In C++ programming language an "include file" was assumed **not having any extension**, but this is also a convention described in the programming language specification, and in fact, is common to find "include files" with **.inc extension** or **.hpp extension**, for example. But as in every API is necessary to start from some point and the Microsoft Windows API should have support for C programming language and for C++ programming language standards and conventions, so we will find "include files" with **.h extension** as a more common choice because of the compatibility with both programming languages. But we will also find the support for specializations and peculiarities, such as **.inc extension** used in certain types of files. Another interesting item we can find when using the **Project -> Add New Item… (Ctrl+Shift+A)** dialog box window that have the header file with the **.h extension**, as default, but the item **Midl File (.idl)**, it is also a type of "include file" used for Component Object Model (COM) interface definition. This stands for that the Microsoft Windows API is not build of only "traditional" C/C++ files, the API also have specialized files that pertain to the specialized and advanced Microsoft Windows operating system technologies.
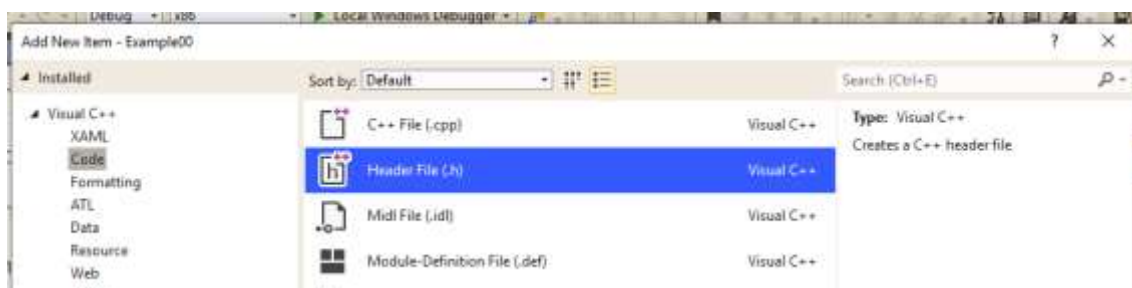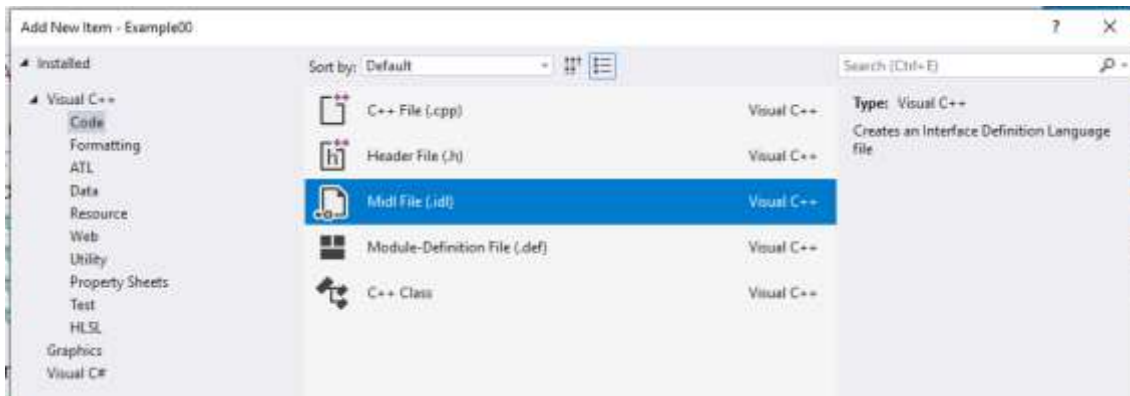
From now on, we will be using the abstract term "include file" as meaning a file that is included via an accepted scheme of use of Microsoft Windows SDK tools, and by which the file content is also recognized and accepted by such groups of tools. The "header file" term will be used for a typical C programming language and C++ programming language supported include files 🗁, with **.h file extension** or with **no file extension**. Any specific "include file", such as with **.idl extension** will be described as used for the first time, and when necessary.

On the Microsoft Windows API, the header files not only introduce symbols but also produces a map for these symbols and set up the relationship among themselves. In general, when using header files, we can have a lot of misinformation or confusion, so it is important to understand the proper kind of symbols that certain header files are introducing into the scope, because even the compiler taking care of resolution of the sequence for introduction of the symbols into the scope, putting the header files in a proper organization is an important programming practice that become even more important when applied in complex and advanced API's, such as Microsoft Windows API.

The Microsoft Windows API have a series of **master header files**.

These **master header files** introduce a large group of symbols , and so, functionalities that are associated through these symbols. By symbols we should understands not only a name of a constants, for example, but also functions and data types, complex or not, such as class types and enums, and as such other kinds that make a demand.

Currently, for the Windows API the master of the master header files is the **Windows.h**, and for the typical use of Microsoft Windows API, we need just of this header file, believe or not. The **Windows.h** header file is a root, or master, of the header files, and through it, a sequence of other header files and symbols are included or not, depending of the conditions presented in certain *conditional macros*.

Let's understand this, doing an exploratory exercise.

Will be using the Microsoft Visual 2017 and Microsoft Visual Studio 2019, and the C++ source code is the same and it is presented in the following listing:

main.cpp

```cpp
#pragma region Include files
#include <Windows.h>
#include <cstdint>
#pragma endregion


#pragma region Namespaces
using namespace std;
#pragma endregion


#pragma region Window Procedure Function

LRESULT CALLBACK MyWindowProcedure( HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam ) {

	constexpr char16_t* const defaultWindowText { u"Hello, Microsoft Windows 10!" };

	LRESULT processResult {};


	switch ( message ) {

	case WM_DESTROY: {

		/*

		When an application processes this message, it should return zero.

		*/

		processResult = FALSE;
```

```
                PostQuitMessage( 0i32 );



        }; break;

        case WM_PAINT: {

                constexpr  SIZE_T rectSize { sizeof( RECT ) };
                constexpr  SIZE_T paintstructSize { sizeof( PAINTSTRUCT ) };
                HANDLE const processHeap { GetProcessHeap() };

                LPPAINTSTRUCT  ps { reinterpret_cast<
LPPAINTSTRUCT >( HeapAlloc( processHeap, HEAP_ZERO_MEMORY, paintstructSize ) ) };
                LPRECT       rect { reinterpret_cast< LPRECT >( HeapAlloc( processHeap,
HEAP_ZERO_MEMORY, rectSize ) ) };

                if ( ( ps != nullptr ) && ( rect != nullptr ) ) {

                        HDC       hdc { BeginPaint( hWindow, ps ) };
                        GetClientRect( hWindow, rect );

                        DrawText( hdc, reinterpret_cast<
LPCWSTR >( defaultWindowText ), -1i32, rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER );

                        EndPaint( hWindow, ps );

                        SecureZeroMemory( reinterpret_cast< PVOID >( ps ),
paintstructSize );

                        SecureZeroMemory( reinterpret_cast< PVOID >( rect ), rectSize );

                };


                HeapFree( processHeap, {}, reinterpret_cast< LPVOID >( ps ) ), ps = nullptr;
                HeapFree( processHeap, {}, reinterpret_cast< LPVOID >( rect ) ), rect =
nullptr;

                /*

                When application process the WM_PAINT message, it should set the
return value for FALSE or ZERO value.
```

The WM_PAINT message is sent by the system and should not be sent explicitly by the application, that is via SendMessage() function for example.

The system sends the WM_PAINT message when there are no other messages in the application's message queue.

The GetMessage() function determines which message to dispatch and returns the WM_PAINT message when there are no other messages in the application's message queue.

The DispatchMessage() function determines where to send the WM_PAINT, that is, for which window procedure in what application that will process that message.

```
                */
                processResult = FALSE;

        }; break;

        case WM_CLOSE: {

                /*

                When an application processes this message, it should return zero.

                */

                processResult = FALSE;

                /*DefWindowProc( hWindow, message, wParam, lParam );*/
                DestroyWindow( hWindow );


        }; break;

                /*
```

The WM_QUIT message is never received through the window procedure because it is not associated with any window. The WM_QUIT message is only retrieved from message queue by the GetMessage() function or by the PeekMessage() function.

Also, we should never post the WM_QUIT message using the PostMessage() function, we always MUST use the PostQuitMessage() function.

```
        case WM_QUIT: break;
```

```
                */

        default: processResult = DefWindowProc( hWindow, message, wParam, lParam );
break;

        };


        return processResult;

};
#pragma endregion

int APIENTRY wWinMain( _In_ HINSTANCE hThisInstance, _In_opt_ HINSTANCE
hPrevInstance, _In_ LPWSTR szCmdLine, _In_ int iCmdShow ) {


        UNREFERENCED_PARAMETER( hPrevInstance );
        UNREFERENCED_PARAMETER( szCmdLine );
        UNREFERENCED_PARAMETER( iCmdShow );


        constexpr char16_t* const applicationName { u"HelloWindowClass" };
        constexpr char16_t* const windowCaption { u"The Hello Program." };
        constexpr SIZE_T wndclassSizeInBytes { sizeof( WNDCLASSEX ) };
        constexpr SIZE_T msgSizeInBytes { sizeof( MSG ) };


        int32_t messageResult {};


        HANDLE const defaultProcessHeap { GetProcessHeap() };
        HWND        hWindow {};
        LPMSG       message { reinterpret_cast< LPMSG >( HeapAlloc( defaultProcessHeap,
HEAP_ZERO_MEMORY, msgSizeInBytes ) ) };
        LPWNDCLASSEX     windowClass { reinterpret_cast<
LPWNDCLASSEX >( HeapAlloc( defaultProcessHeap, HEAP_ZERO_MEMORY,
wndclassSizeInBytes ) ) };

        if ( windowClass != nullptr ) {

#pragma region Task 00 - Create main window.
```

```
windowClass->cbSize = wndclassSizeInBytes;
windowClass->style = ( CS_HREDRAW | CS_VREDRAW );
windowClass->lpfnWndProc = &MyWindowProcedure;

/*windowClass->cbClsExtra = 0;
windowClass->cbWndExtra = 0;*/

windowClass->hInstance = hThisInstance; /* Instance of process. */
windowClass->hIcon = reinterpret_cast< HICON >( LoadImage( nullptr,
IDI_APPLICATION, IMAGE_ICON, {}, {}, LR_DEFAULTSIZE | LR_LOADMAP3DCOLORS |
LR_LOADTRANSPARENT ) );
windowClass->hCursor = reinterpret_cast<
HCURSOR >( LoadImage( nullptr, IDC_ARROW, IMAGE_CURSOR, {}, {}, LR_DEFAULTSIZE |
LR_LOADMAP3DCOLORS | LR_LOADTRANSPARENT ) );
windowClass->hbrBackground = reinterpret_cast<
HBRUSH >( GetStockObject( WHITE_BRUSH ) );
/*windowClass->lpszMenuName = nullptr;*/
windowClass->lpszClassName = reinterpret_cast<
LPCWSTR >( applicationName );

if ( !RegisterClassEx( reinterpret_cast< WNDCLASSEX const
* >( windowClass ) ) ) {
MessageBox( nullptr, TEXT( "This program requires Microsoft
Windows NT!" ),
reinterpret_cast< LPCWSTR >( applicationName ),
MB_ICONERROR );
} else {

hWindow = CreateWindowEx( {}, reinterpret_cast<
LPCTSTR >( applicationName ),
reinterpret_cast< LPCTSTR >( windowCaption ),
WS_OVERLAPPEDWINDOW | WS_HSCROLL |
WS_VSCROLL,

CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
nullptr,
nullptr,
hThisInstance,
nullptr );
```

```cpp
        if ( hWindow != nullptr ) {

                        ShowWindow( hWindow, SW_SHOWDEFAULT );
                        UpdateWindow( hWindow );

                        if ( message != nullptr ) {

                                while ( messageResult =
GetMessage( reinterpret_cast< LPMSG >( message ), nullptr, 0ui32, 0ui32 ) ) {

                                        TranslateMessage( reinterpret_cast<
MSG* >( message ) );

                                        DispatchMessage( reinterpret_cast<
MSG* >( message ) );

                                };

                        };
                };

        }; /* end create main window */
#pragma endregion


        DestroyIcon( windowClass->hIcon );
        DestroyCursor( windowClass->hCursor );


        windowClass = reinterpret_cast<
LPWNDCLASSEX >( SecureZeroMemory( reinterpret_cast< LPVOID >( windowClass ),
wndclassSizeInBytes ) );

    };

    HeapFree( defaultProcessHeap, {}, reinterpret_cast< LPVOID >( message ) ),
message = nullptr;
    HeapFree( defaultProcessHeap, {}, reinterpret_cast< LPVOID >( windowClass ) ),
windowClass = nullptr;


    return messageResult;
```
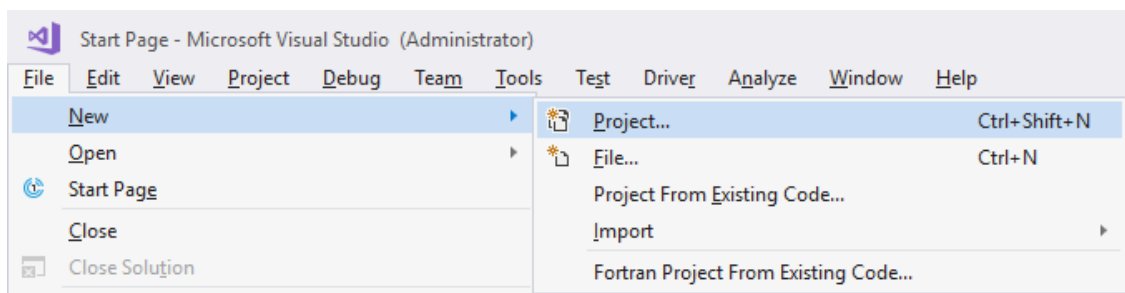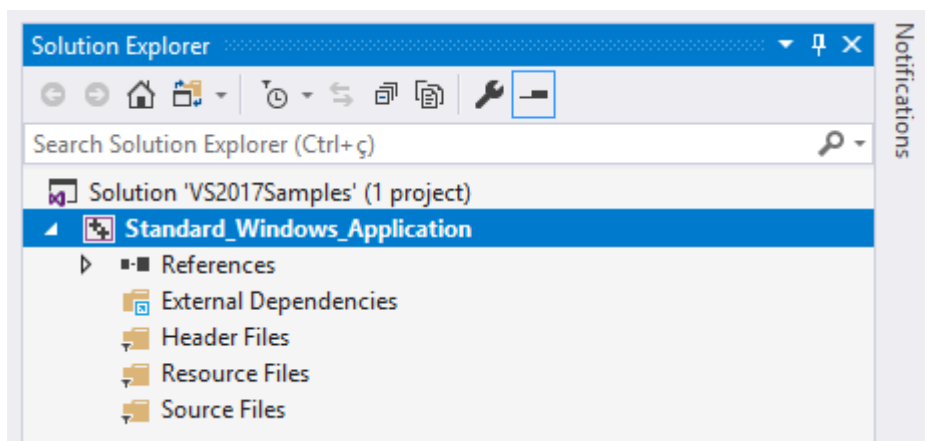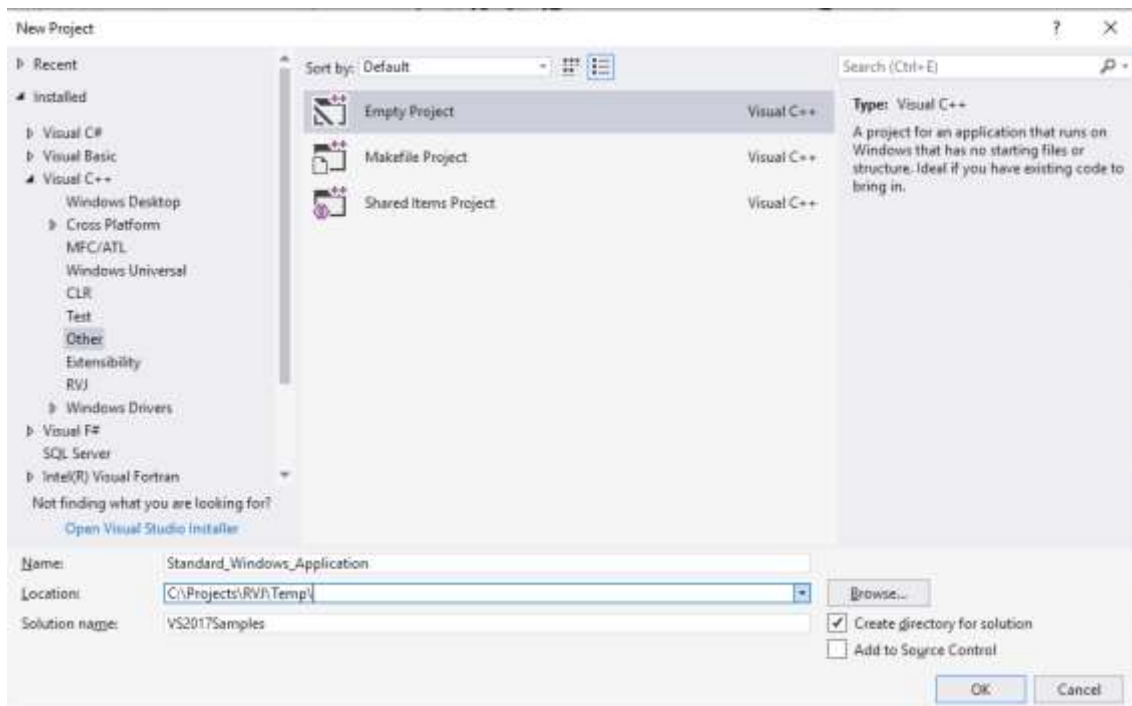
Here is the sequence that we must follow to create the solution and the project using **Microsoft Visual Studio 2017**:
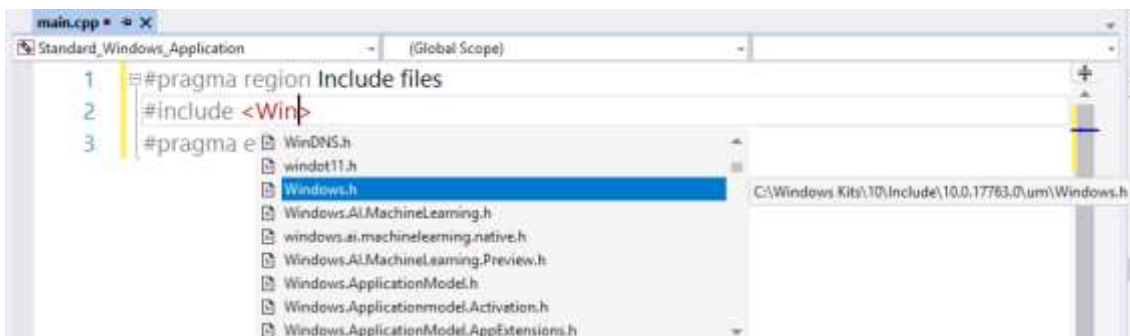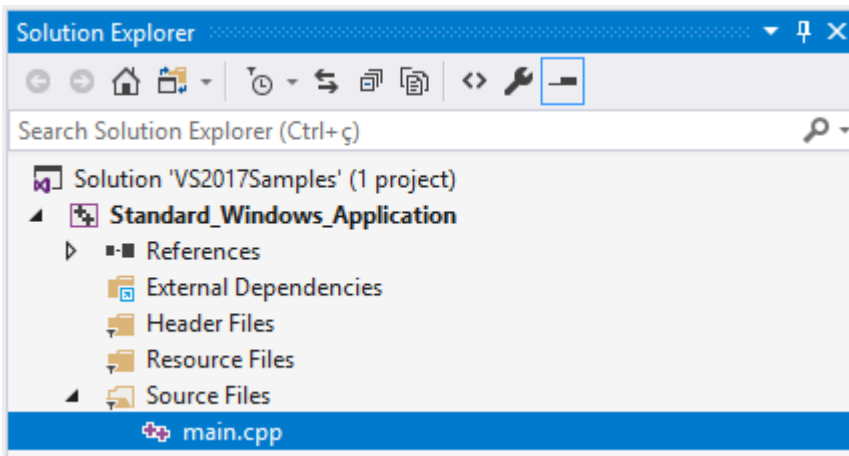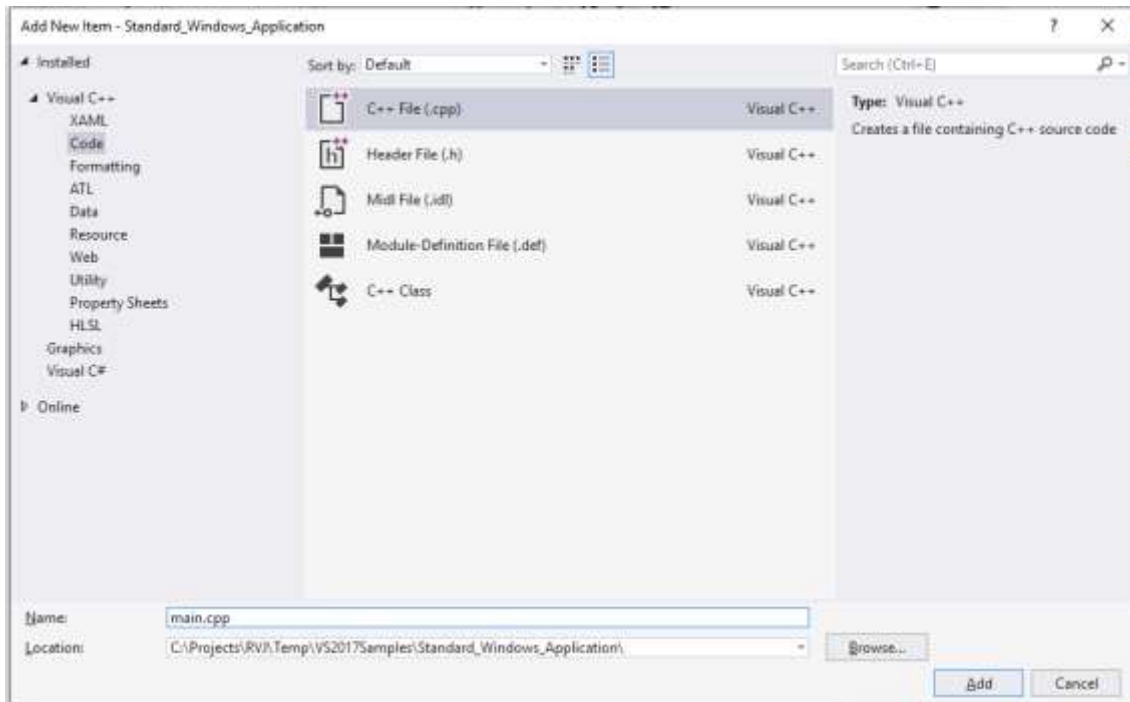
1. Click on **File->New->Project…** or use the **Ctrl+Shift+N** shortcut.

2. On **New Project** dialog window, choose **Visual C++** category.

3. Choose **Visual C++ -> Other** subcategory .

4. On the right side of the window we will find a template named **Empty Project.**

5. Choose the template named **Empty Project**.

6. Give the name **Standard_Windows_Application** as the project name.

7. Give the name **VS2017Samples** as the solution name.

8. Use a temporary folder such as **C:\Projects\RVJ\Temp** or another that you want.

9. Click on the button **OK** to create the solution and the project.

10. On the **Solution Explorer** window, right-click on the project name and choose **Add -> New Item…** or press **Ctrl+Shift+A** to insert a new source code file.

11. On **Add New Item** dialog box window that appears, on the left-side choose the **Visual C++ -> Code** as category and subcategory, respectively.

12. On the list of items, choose **C++ File (.cpp)** for the template.

13. Give the **main.cpp** as the name for the new source code file.

14. On the bottom of the window, click on the **Add** button.

15. A new empty C++ source code file is inserted on the project.

Here are the screens to help with the instructions:

Here is the sequence that we must follow to create the solution and the project using **Microsoft Visual Studio 2019**:

1. Click on **File->New->Project…** or use the **Ctrl+Shift+N** shortcut.

2. The new create project dialog window of Microsoft Visual Studio 2019 is presented.

3. On the top we have the filters the **Language**, the **Platform**, and the **Project type** for choosing the template. We also have a search box when we can input few words to find the right template.

4. Choose the following values for the filters:

a. **Language -> C++**

b. **Platform -> Windows**

c. **Project type -> Console**

5. With filters applied the **Empty Project** template appears as the first in the result list, at least on my installation. 😊

6. Click on the template named **Empty Project**. The template is selected.

7. Click on the **Next** button at the bottom of window.

8. The next presented screen is the **Configure your new project** window.

9. Give the name **Standard_Windows_Application** as the project name.

10. Give the name **VS2019Samples** as the solution name.

11. Use a temporary folder such as **C:\Projects\RVJ\Temp** or another that you want.

12. Click on the button **Create** at the bottom of the screen to create the solution and the project.

13. On the **Solution Explorer** window, right-click on the project name and choose **Add -> New Item…** or press **Ctrl+Shift+A** to insert the new source code file.

14. On **Add New Item** dialog box window that appears, on the left-side choose the **Visual C++ -> Code** as category and subcategory, respectively.

15. On the list of items, choose **C++ File (.cpp)** for the template.

16. Give the **main.cpp** as the name for the new source code file.

17. On the bottom of the window, click on the **Add** button.

18. A new empty C++ source code file is inserted on the project.

Here are the screens to help with the instructions:
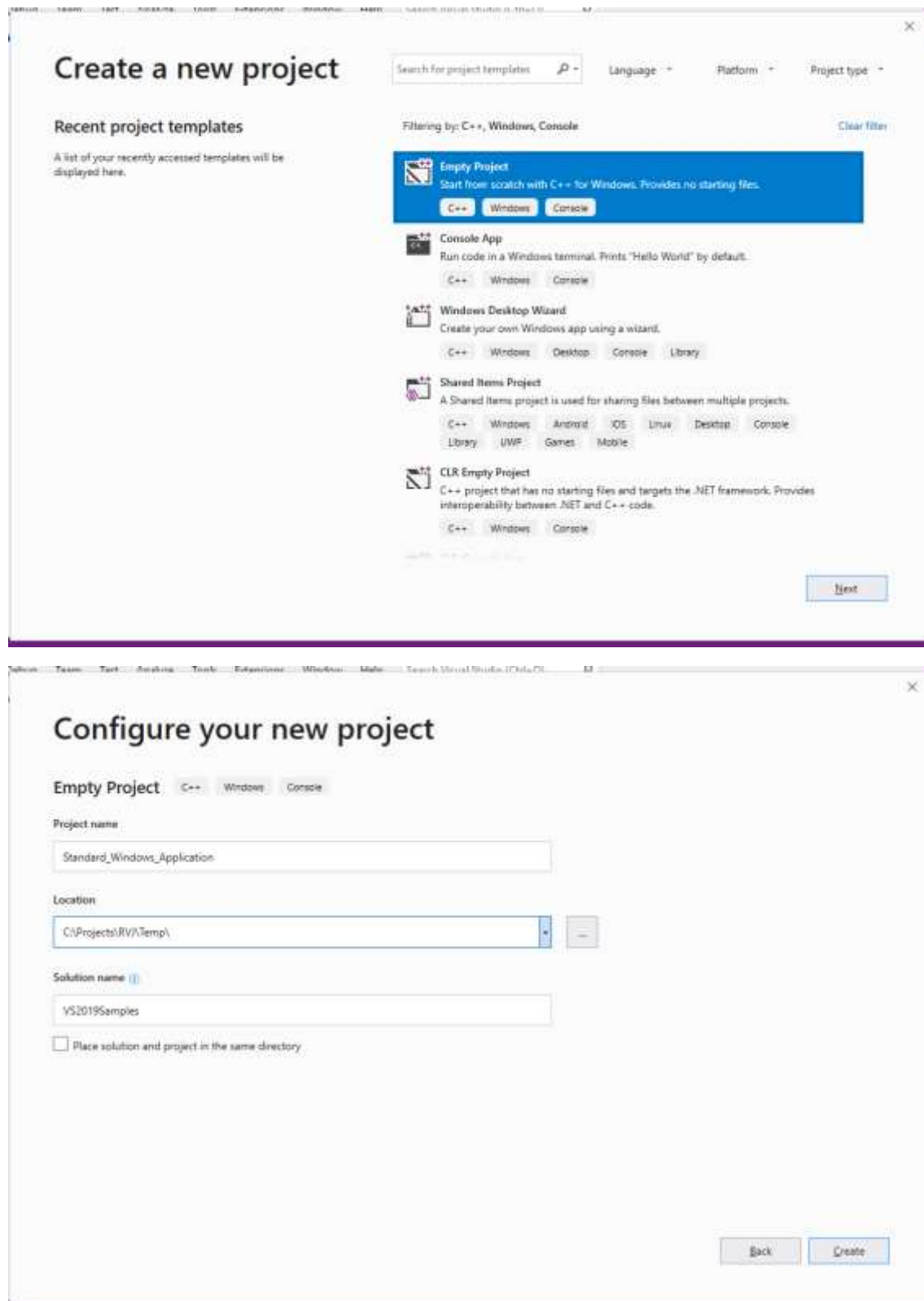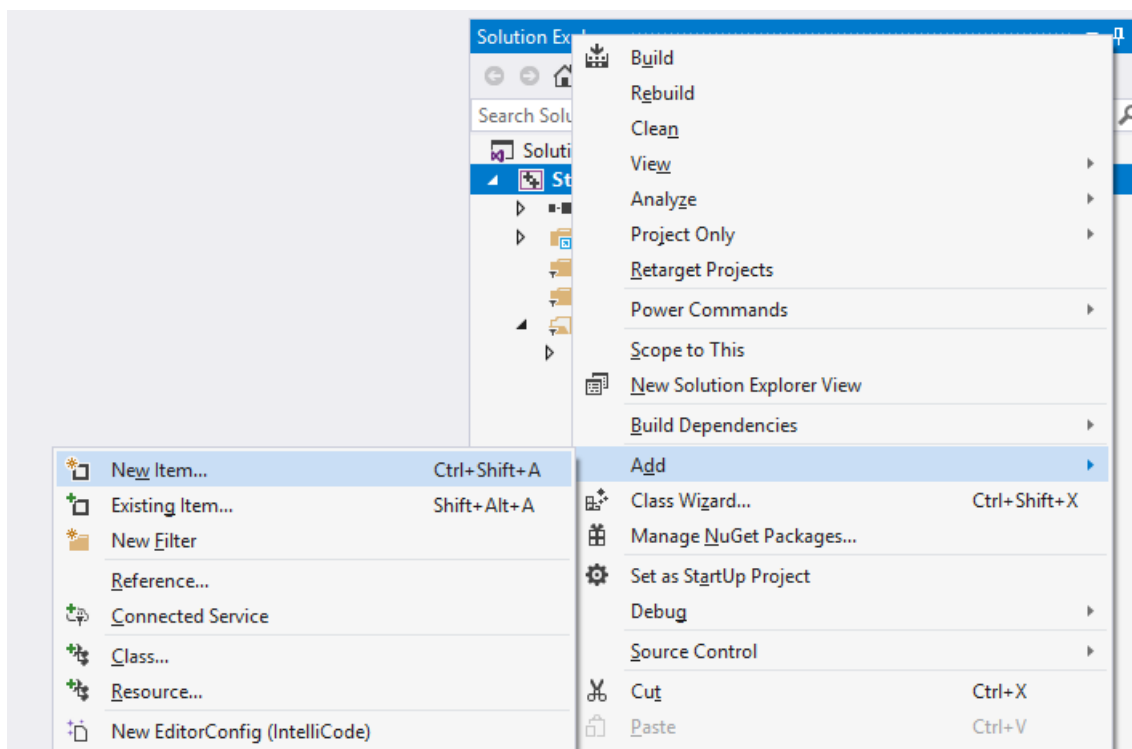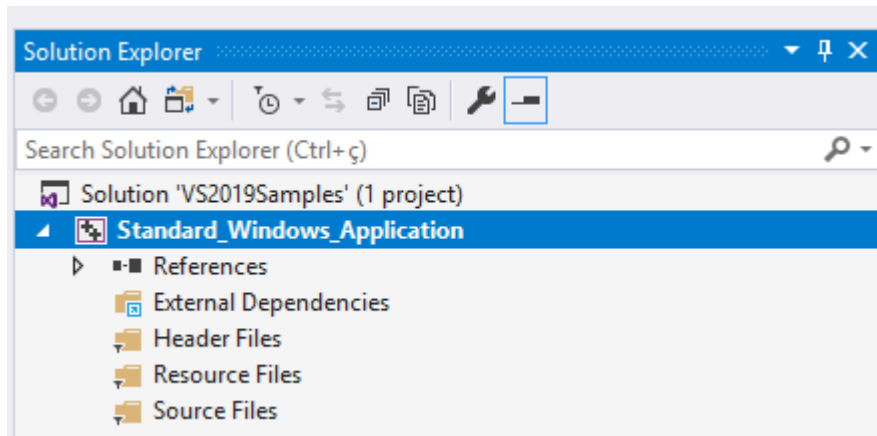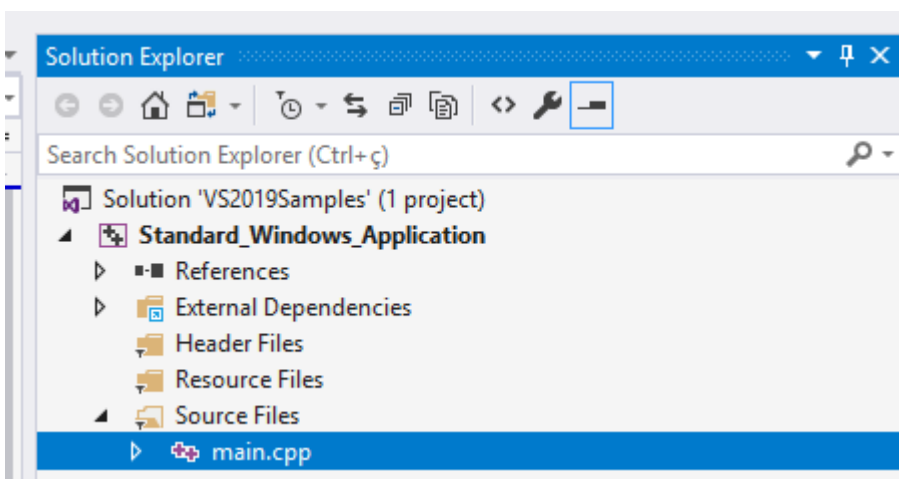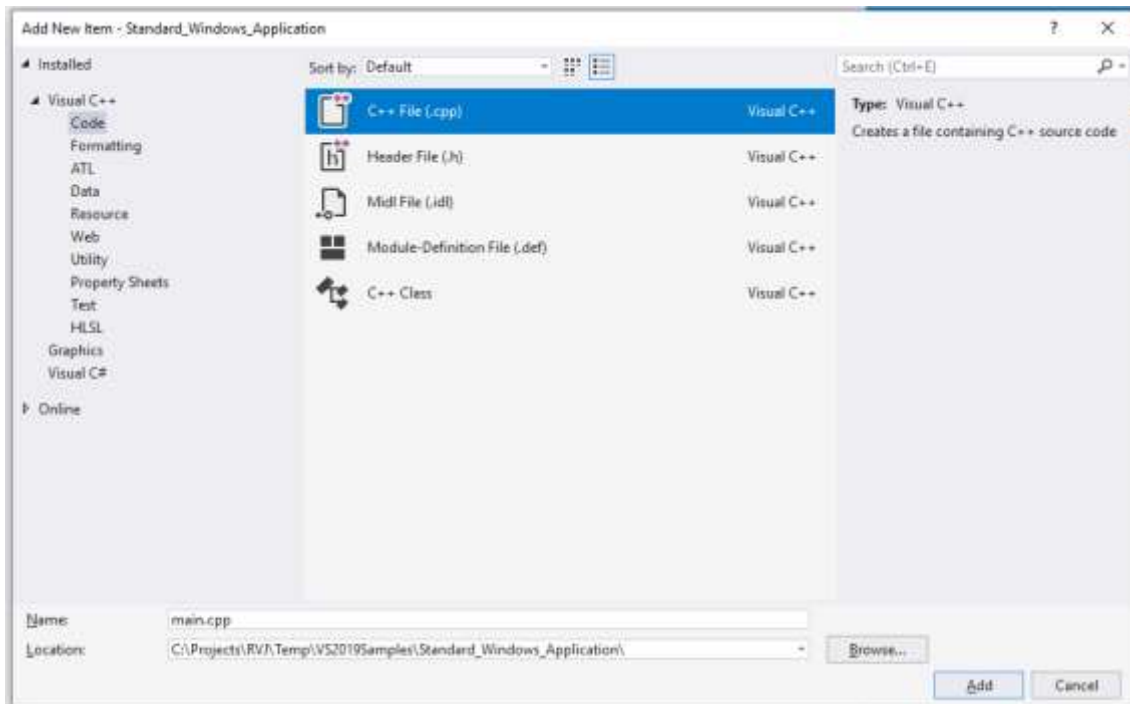
Now that with the solution and the project for our lessons was created, let's return for the conversion about the header files in the Microsoft Windows API.

With every Microsoft Windows SDK update we have new header files, header files that are updated, replaced or removed, so this is an especially important fact to keep in mind when compiling any code base. The Microsoft Windows SDK adopt the politics of preserving header file names for compatibility and to helps the updating of the header files content for the developers. Of course, with evolution of the Microsoft Windows SDK there are the latest content included in appropriated header files as demanded. For example, the concept of API FAMILY is something relatively recent in Microsoft Windows API and Microsoft Windows programming world. Opening the header file **Windows.h**, the text at the top of the file includes

a header file named **WinApiFamily.h**, and after that we have a section comment with distinct information about changing the 0001 build version number and informing that the **Windows.h** is the master include file for the development of applications for the Microsoft Windows operating system. In the following listing we have all current content of the **Windows.h** header file, currently no more than three hundred (300) lines of code, including all comments. This sounds a surprise considering the importance of the header file. But this is one of the characteristics of the Microsoft Windows API, the *contextual modularity*. Reading the content of **Windows.h** header file we will find symbols in the form of C++ macros, and these macros define information for various contexts that are supported by the Microsoft Windows API such as hardware architecture, Microsoft C/C++ compiler version and respective features, and inclusion of other infrastructural header files, for example. We must be aware that this are not the only group of header files, it is the start point for the most infrastructural header files of the Windows API and that expose sequence of system services such as memory management, process and thread management, certain infrastructure services for Graphical User Interface (GUI), Dynamic-link library (DLL) management, synchronization for thread management, locally and remote communication between processes via the interprocess communication (IPC) and the remote procedure call (RPC), Windows Services that are processes running on the background executing complex tasks, and others.

Windows.h header file

#include <winapifamily.h>

/*++ BUILD Version: 0001   Increment this if a change has global effects

Copyright (c) Microsoft Corporation. All rights reserved.

Module Name:

    windows.h

Abstract:

    Master include file for Windows applications.

--*/

```
#ifndef _WINDOWS_
#define _WINDOWS_


#include <sdkddkver.h>


#ifndef _INC_WINDOWS
#define _INC_WINDOWS


#if defined (_MSC_VER) && (_MSC_VER >= 1020)
#pragma once
#endif


#pragma region Application Family or OneCore Family
#if WINAPI_FAMILY_PARTITION(WINAPI_PARTITION_APP | WINAPI_PARTITION_SYSTEM)


/*  If defined, the following flags inhibit definition
 *     of the indicated items.
 *
 *  NOGDICAPMASKS     - CC_*, LC_*, PC_*, CP_*, TC_*, RC_
 *  NOVIRTUALKEYCODES - VK_*
 *  NOWINMESSAGES     - WM_*, EM_*, LB_*, CB_*
 *  NOWINSTYLES       - WS_*, CS_*, ES_*, LBS_*, SBS_*, CBS_*
 *  NOSYSMETRICS      - SM_*
 *  NOMENUS           - MF_*
 *  NOICONS           - IDI_*
 *  NOKEYSTATES       - MK_*
 *  NOSYSCOMMANDS     - SC_*
 *  NORASTEROPS       - Binary and Tertiary raster ops
```

```
 *  NOSHOWWINDOW      - SW_*

 *  OEMRESOURCE       - OEM Resource values

 *  NOATOM            - Atom Manager routines

 *  NOCLIPBOARD       - Clipboard routines

 *  NOCOLOR           - Screen colors

 *  NOCTLMGR          - Control and Dialog routines

 *  NODRAWTEXT        - DrawText() and DT_*

 *  NOGDI             - All GDI defines and routines

 *  NOKERNEL          - All KERNEL defines and routines

 *  NOUSER            - All USER defines and routines

 *  NONLS             - All NLS defines and routines

 *  NOMB              - MB_* and MessageBox()

 *  NOMEMMGR          - GMEM_*, LMEM_*, GHND, LHND, associated routines

 *  NOMETAFILE        - typedef METAFILEPICT

 *  NOMINMAX          - Macros min(a,b) and max(a,b)

 *  NOMSG             - typedef MSG and associated routines

 *  NOOPENFILE        - OpenFile(), OemToAnsi, AnsiToOem, and OF_*

 *  NOSCROLL          - SB_* and scrolling routines

 *  NOSERVICE         - All Service Controller routines, SERVICE_ equates, etc.

 *  NOSOUND           - Sound driver routines

 *  NOTEXTMETRIC      - typedef TEXTMETRIC and associated routines

 *  NOWH              - SetWindowsHook and WH_*

 *  NOWINOFFSETS      - GWL_*, GCL_*, associated routines

 *  NOCOMM            - COMM driver routines

 *  NOKANJI           - Kanji support stuff.

 *  NOHELP            - Help engine interface.

 *  NOPROFILER        - Profiler interface.

 *  NODEFERWINDOWPOS  - DeferWindowPos routines

 *  NOMCX             - Modem Configuration Extensions

 */
```

```
#if defined(RC_INVOKED) && !defined(NOWINRES)

#include <winresrc.h>

#else

#if defined(RC_INVOKED)
/* Turn off a bunch of stuff to ensure that RC files compile OK. */
#define NOATOM
#define NOGDI
#define NOGDICAPMASKS
#define NOMETAFILE
#define NOMINMAX
#define NOMSG
#define NOOPENFILE
#define NORASTEROPS
#define NOSCROLL
#define NOSOUND
#define NOSYSMETRICS
#define NOTEXTMETRIC
#define NOWH
#define NOCOMM
#define NOKANJI
#define NOCRYPT
#define NOMCX
#endif

#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) && defined(_M_IX86)
#define _X86_
#if !defined(_CHPE_X86_ARM64_) && defined(_M_HYBRID)
```

```
#define _CHPE_X86_ARM64_

#endif

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) &&
defined(_M_AMD64)

#define _AMD64_

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) && defined(_M_ARM)

#define _ARM_

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) &&
defined(_M_ARM64)

#define _ARM64_

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) && defined(_M_M68K)

#define _68K_

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_IA64_)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) && defined(_M_MPPC)

#define _MPPC_

#endif


#if !defined(_68K_) && !defined(_MPPC_) && !defined(_X86_) && !defined(_M_IX86)
&& !defined(_AMD64_) && !defined(_ARM_) && !defined(_ARM64_) && defined(_M_IA64)
```

```
#if !defined(_IA64_)
#define _IA64_
#endif /* !_IA64_ */
#endif


#ifndef _MAC
#if defined(_68K_) || defined(_MPPC_)
#define _MAC
#endif
#endif


#if defined (_MSC_VER)
#if ( _MSC_VER >= 800 )
#ifndef __cplusplus
#pragma warning(disable:4116)      /* TYPE_ALIGNMENT generates this - move it */
                       /* outside the warning push/pop scope. */
#endif
#endif
#endif


#ifndef RC_INVOKED
#if    ( _MSC_VER >= 800 )
#pragma warning(disable:4514)
#ifndef __WINDOWS_DONT_DISABLE_PRAGMA_PACK_WARNING__
#pragma warning(disable:4103)
#endif
#if _MSC_VER >= 1200
#pragma warning(push)
#endif
#pragma warning(disable:4001)
#pragma warning(disable:4201)
```

```
#pragma warning(disable:4214)
#endif
#include <excpt.h>
#include <stdarg.h>
#endif /* RC_INVOKED */


#include <windef.h>
#include <winbase.h>
#include <wingdi.h>
#include <winuser.h>
#if !defined(_MAC) || defined(_WIN32NLS)
#include <winnls.h>
#endif
#ifndef _MAC
#include <wincon.h>
#include <winver.h>
#endif
#if !defined(_MAC) || defined(_WIN32REG)
#include <winreg.h>
#endif
#ifndef _MAC
#include <winnetwk.h>
#endif


#ifndef WIN32_LEAN_AND_MEAN
#include <cderr.h>
#include <dde.h>
#include <ddeml.h>
#include <dlgs.h>
#ifndef _MAC
#include <lzexpand.h>
```

```
#include <mmsystem.h>
#include <nb30.h>
#include <rpc.h>
#endif
#include <shellapi.h>
#ifndef _MAC
#include <winperf.h>
#include <winsock.h>
#endif
#ifndef NOCRYPT
#include <wincrypt.h>
#include <winefs.h>
#include <winscard.h>
#endif

#ifndef NOGDI
#ifndef _MAC
#include <winspool.h>
#ifdef INC_OLE1
#include <ole.h>
#else
#include <ole2.h>
#endif /* !INC_OLE1 */
#endif /* !MAC */
#include <commdlg.h>
#endif /* !NOGDI */
#endif /* WIN32_LEAN_AND_MEAN */

#include <stralign.h>

#ifdef _MAC
```

```
#include <winwlm.h>
#endif


#ifdef INC_OLE2
#include <ole2.h>
#endif /* INC_OLE2 */


#ifndef _MAC
#ifndef NOSERVICE
#include <winsvc.h>
#endif


#if(WINVER >= 0x0400)
#ifndef NOMCX
#include <mcx.h>
#endif /* NOMCX */


#ifndef NOIME
#include <imm.h>
#endif
#endif /* WINVER >= 0x0400 */
#endif


#ifndef RC_INVOKED
#if    ( _MSC_VER >= 800 )
#if _MSC_VER >= 1200
#pragma warning(pop)
#else
#pragma warning(default:4001)
#pragma warning(default:4201)
```

```
#pragma warning(default:4214)
/* Leave 4514 disabled.  It's an unneeded warning anyway. */
#endif
#endif
#endif /* RC_INVOKED */


#endif /* RC_INVOKED */


#endif /* WINAPI_FAMILY_PARTITION(WINAPI_PARTITION_APP |
WINAPI_PARTITION_SYSTEM) */
#pragma endregion


#endif /* _INC_WINDOWS */


#endif /* _WINDOWS_ */
```

There are other important header files that are included through this sequence introduced by the **Windows.h** header file. As we can see in the graphic, the header files **WinApiFamily.h** and **SdkDDKVer.h** are also included as part of the **Windows.h** header file.

Windows.h

WinApiFamily.h   Sdkddkver.h

Opening the **WinApiFamily.h** header file, we can read this text at the top of the file:

/*

Copyright (c) Microsoft Corporation. All rights reserved.

Module Name:

    winapifamily.h

Abstract:

    Master include file for API family partitioning.

*/

This is another master header file part of the Microsoft Windows SDK. The **WinApiFamily.h** header file explain for us that the Microsoft Windows operating is not just one, isolated product, it is a family of products that are based on the Microsoft Windows operating system technologies.

But for better understand the journey and the dimension of the Windows API, look at this part of the Microsoft official documentation. Here we have a series of technologies of the Microsoft Windows family that are accessible via Windows API:

https://docs.microsoft.com/en-us/windows/desktop/api/index

On the left side we have a list with technological context, such as the technologies for use on the construction and manipulation of a backup system. When we click on any of these names,

we are redirected for a specialized documentation page that includes, among others, information about the header files that we should include in our projects when developing a custom solution using a specific technological context, such as backup system for example:

If we click in a technological context such as *Backup*, we have a page that, among others information, shows the names of important header files that expose Windows API's recommended or required for use in the development of a custom backup system, and in the interaction with the advanced backup services currently implemented by Microsoft Windows client, embedded, IoT and server. Now, looking for the names of the header files showed in the documentation page we have **winbase.h** and **winnt.h**, and opening again the installed version that we are using of the **Windows.h** header file and looking in the listing of the header files that are included, we will find **winbase.h** header file, but not **winnt.h** header file. We will not find **winnt.h** header file directly cited in some #include directive in the **Windows.h** header file by two fundamental reasons, because of reorganization of the header files in Microsoft Windows SDK and because **winnt.h** is a more infrastructural header file yet than **winbase.h** header file, and this means that another header file in the sequence is including **winnt.h** header file.

Open the solution **VS2017Samples** or **VS2019Samples**, the project **Standard_Windows_Application** is also loaded. In the **Solution Explorer** window expand the **External Dependencies** node in the **Standard_Windows_Application** project and scrolling through the list we will see the name of the **winnt.h** header file.
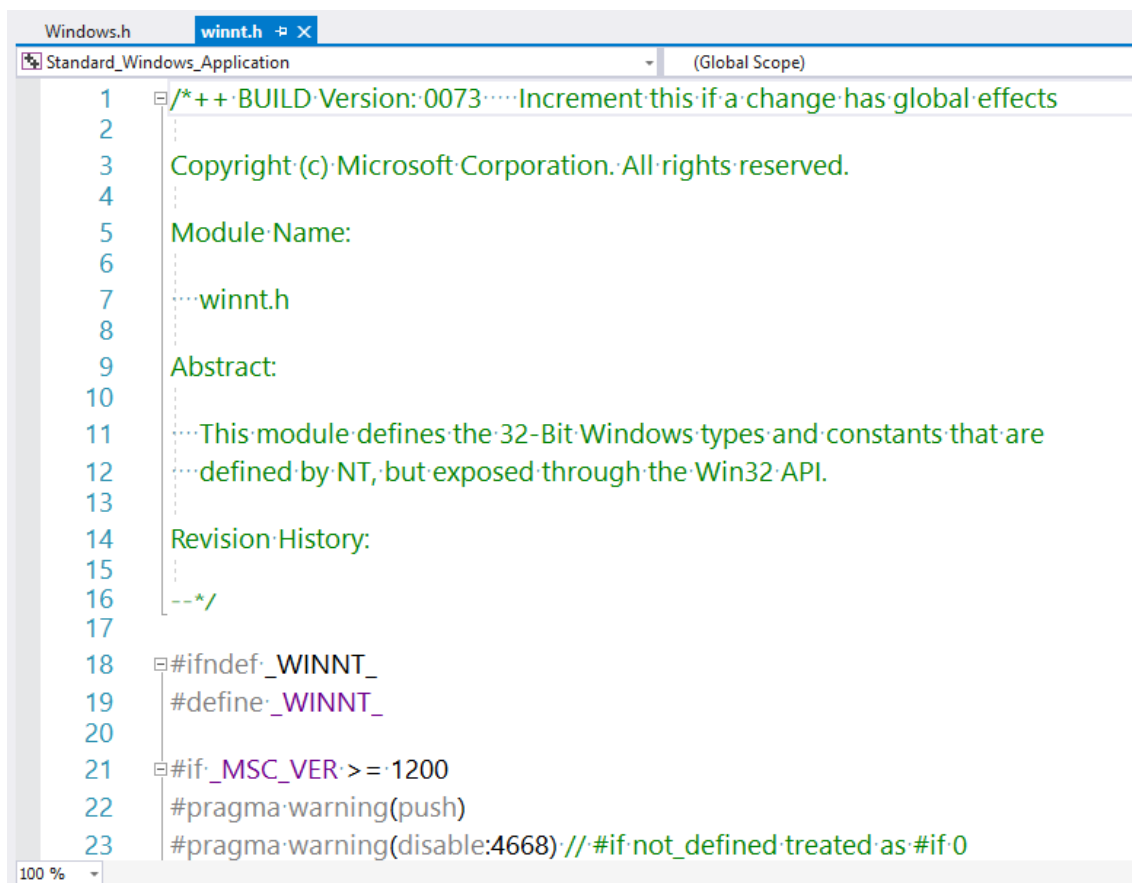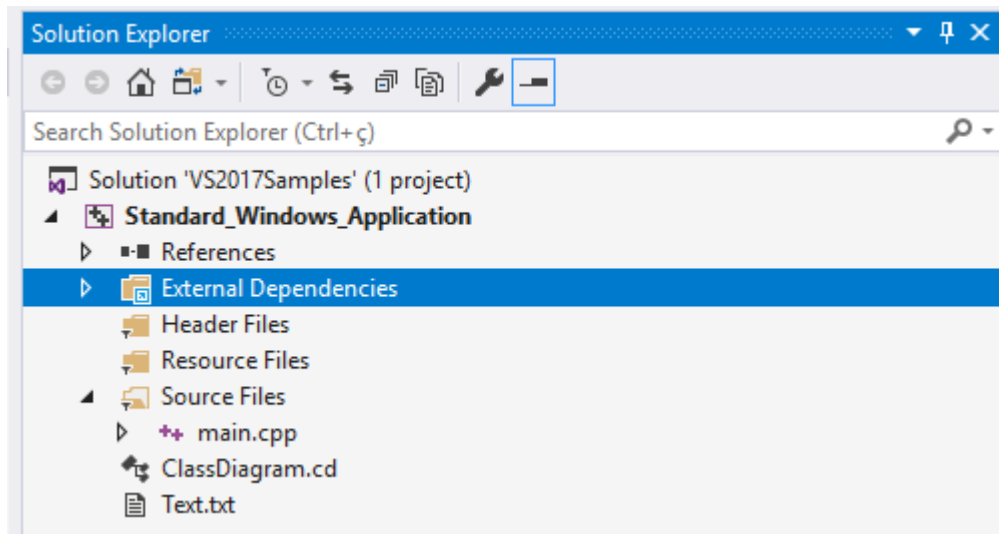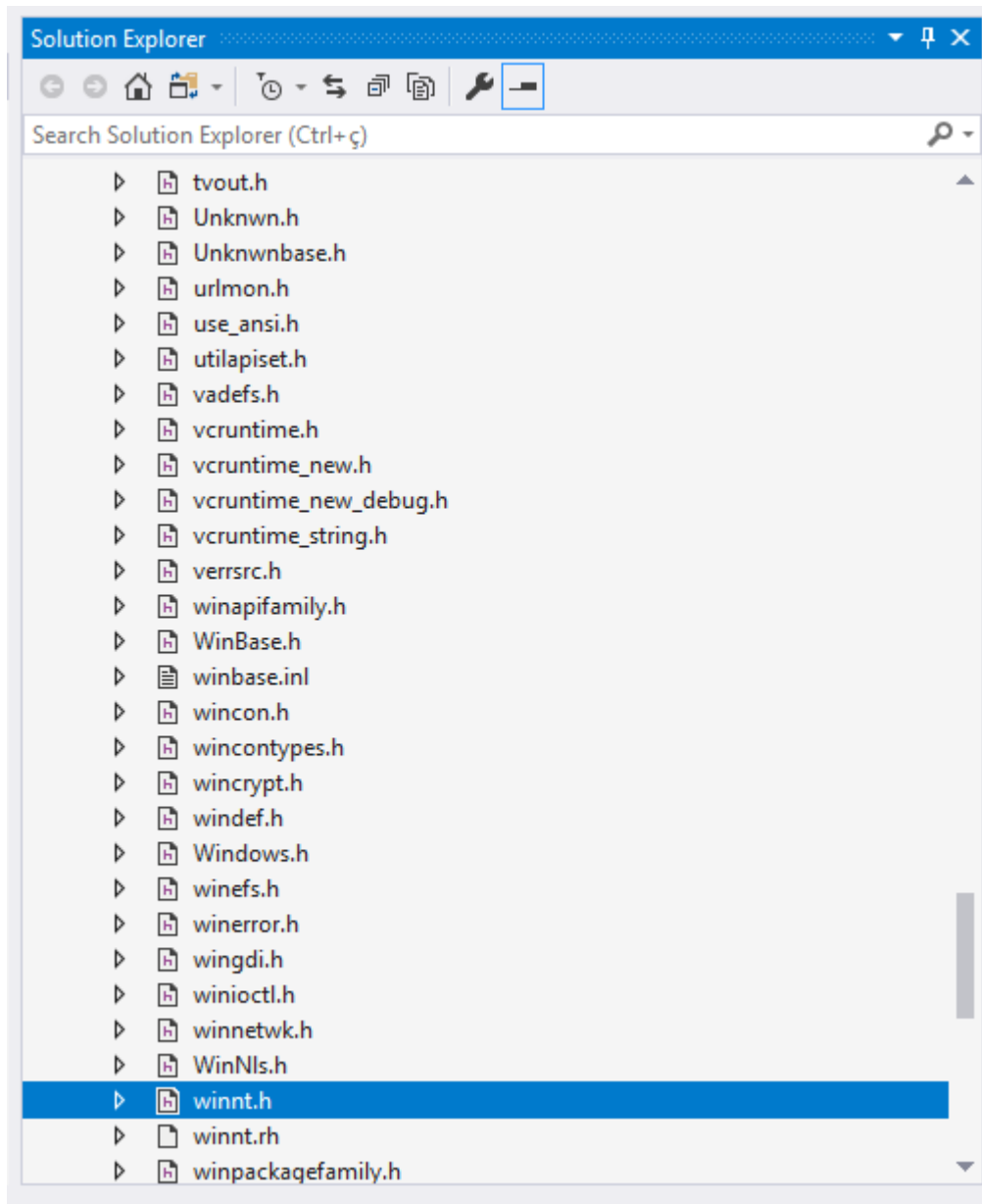
Currently the **winnt.h** header file have about twenty two thousand and five hundred (22.500) of source lines, including all the comments.

With the **winnt.h** header file opened we can use the dropdown lists to view scopes and data types, and to view functions of the Windows API that are declared or defined in the header file:

_HARDWARE_COUNTER_DATA

Struct: _HARDWARE_COUNTER_DATA

Use the dropdown to view and navigate to all scopes/types in this file (winnt.h).
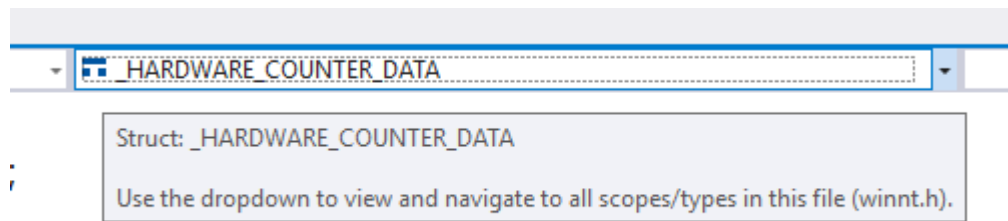
NFORMATION ·*PACTIVATION CONTEXT DETAILED INFORM

_MEMORY_BASIC_INFORMATION

_MEMORY_BASIC_INFORMATION
_MEMORY_BASIC_INFORMATION32
_MEMORY_BASIC_INFORMATION64
_MESSAGE_RESOURCE_BLOCK
_MESSAGE_RESOURCE_DATA
_MESSAGE_RESOURCE_ENTRY
_MONITOR_DISPLAY_STATE
_NEON128
_NETWORK_APP_INSTANCE_EA
_NON_PAGED_DEBUG_INFO
_NT_TIB
_NT_TIB::__unnamed_union_0035_31

MATION

Use the dropdown to view and navigate to all the functions in (Global Scope).

InterlockedXor16(SHORT volatile * Destination, SHORT Value)
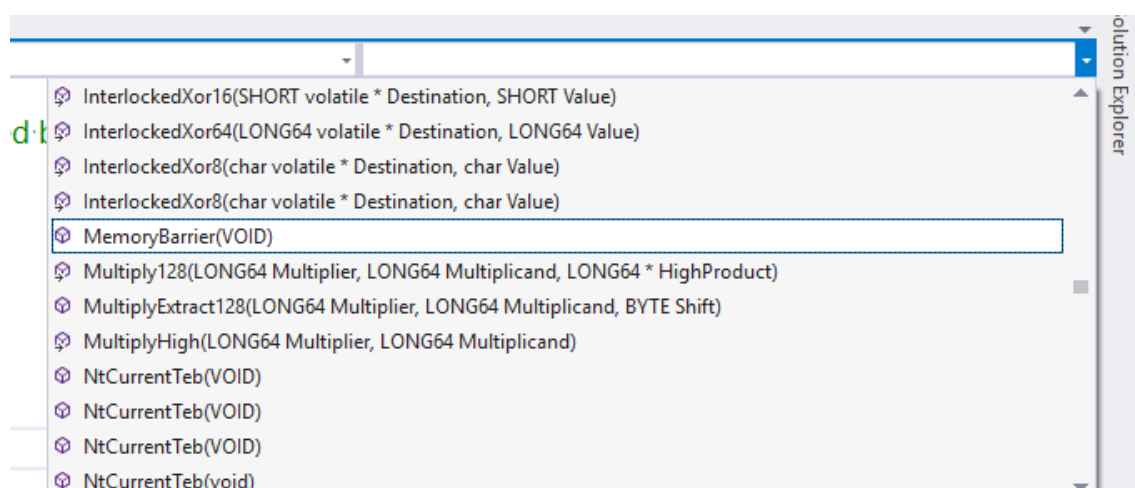InterlockedXor64(LONG64 volatile * Destination, LONG64 Value)
InterlockedXor8(char volatile * Destination, char Value)
InterlockedXor8(char volatile * Destination, char Value)
MemoryBarrier(VOID)
Multiply128(LONG64 Multiplier, LONG64 Multiplicand, LONG64 * HighProduct)
MultiplyExtract128(LONG64 Multiplier, LONG64 Multiplicand, BYTE Shift)
MultiplyHigh(LONG64 Multiplier, LONG64 Multiplicand)
NtCurrentTeb(VOID)
NtCurrentTeb(VOID)
NtCurrentTeb(VOID)
NtCurrentTeb(void)

Still with the **winnt.h** header file opened we can learn more specific details about the Windows API elements before continuing talking about the **winbase.h** header file content.

As example, we will talk about two functions and one data structure, but we will not talk about the purpose of the functions or data structure, but about aspects that we commonly find in the declaration and definition of functions and data structures in the Windows API.

In the dropdown list for the functions, choose the **MemoryBarrier()** function. The *#pragma directives* are more than common, they are a requirement, and we will not find a single function or data structure in the Windows API declared or defined outside of the influence of #pragma directives, directly or indirectly. The #pragma directives are used for indicating machine-specific or operating-specific compiler features, this is also used for code analysis for example. With the help of macros, conditions are also included on the decisions for the use of the #pragma directives and respective values. Another common resource used in the Windows API is an **inline function**, and the **MemoryBarrier()** is an inline implemented function. The **FORCEINLINE** is not a C keyword, C++ keyword, inline assembly or an intrinsic function, but it is macro also defined in the **winnt.h** header file for a special decision about the implementation of an inline function. This is the entire block that defines the function:

```
#if !defined(MIDL_PASS) && defined(_M_IX86)

#if !defined(_M_CEE_PURE) && !defined(_M_HYBRID_X86_ARM64)

#pragma prefast(push)
#pragma warning(push)
#pragma prefast(disable: 6001 28113, "The barrier variable is accessed only to create a side effect.")
#pragma warning(disable: 4793)
FORCEINLINE
VOID
MemoryBarrier (
    VOID
    )
{
```

```
    LONG Barrier;


    InterlockedOr(&Barrier, 0);
    return;
}


#pragma warning(pop)
#pragma prefast(pop)


#endif /* !_M_CEE_PURE || !_M_HYBRID_X86_ARM64*/
```

If we select the **FORCEINLINE** macro and press **F12 (Go To Definition)**, we will have the definition of the macro presented in the code editor of the Microsoft Visual Studio IDE. The **FORCEINLINE** macro will have the value **__forceinline** or **__inline** depending of the minimum version of the Microsoft C/C++ compiler in use. If the Microsoft C/C++ compiler in use for build is **greater than to or at least equal to 1200**, the **__forceinline** keyword is used because this version of Microsoft C/C++ compiler have support for this keyword. If the Microsoft C/C++ compiler version number in use for build is **less than 1200**, the **__inline** keyword is used because these earlier versions of Microsoft C/C++ compiler does not have support for the **__forceinline** keyword and the respective functionality. The **__inline** keyword is used to suggest to the compiler that it is important, and even critical, get the generated assembly source block for this function and insert into every point where a calling to respective function was found, effectively replacing the calls for the function. But, the **__inline** keyword is a suggestion to the compiler, and the compiler can opt to ignore completely, the judgment is made by the compiler. On the other hand, the **__forceinline** keyword is not a suggestion to the compiler, it is a command to the compiler. In this case, the compiler will trust in the judgment of who takes this decision, assuming that, using the inline implementation for the function is always the better solution for the scenarios.

Inline function is a technique for potential optimization, but not a guarantee, because the algorithm used to implement the function should be efficient for the purpose.
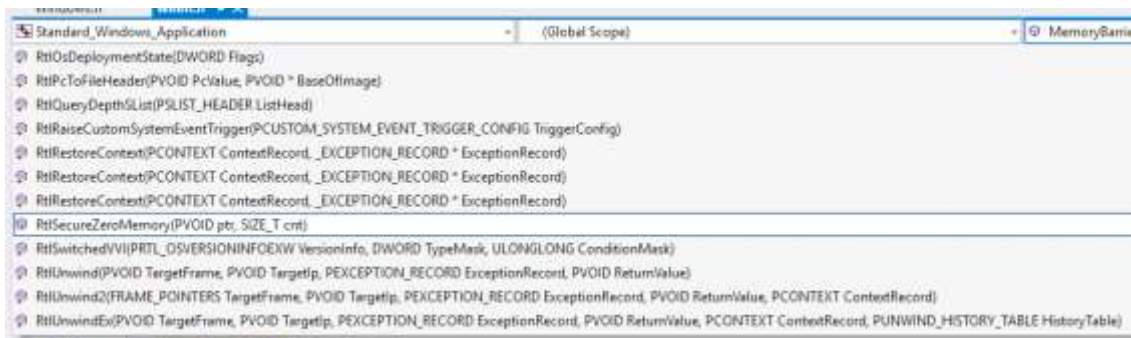
```
#ifndef FORCEINLINE
#if (_MSC_VER >= 1200)
#define FORCEINLINE __forceinline
#else
```

```
#define FORCEINLINE __inline
#endif
#endif
```

Now, just for see how this is an important technique for potential optimization, we will look other function, also implemented as inline. Again, open the dropdown list with the function names and choose the **RtlSecureZeroMemory()** function:


```
#if !defined(MIDL_PASS)

FORCEINLINE
PVOID
RtlSecureZeroMemory(
    _Out_writes_bytes_all_(cnt) PVOID ptr,
    _In_ SIZE_T cnt
    )
{
    volatile char *vptr = (volatile char *)ptr;

#if defined(_M_AMD64)

    __stosb((PBYTE )((DWORD64)vptr), 0, cnt);

#else
```

```
    while (cnt) {

#if !defined(_M_CEE) && (defined(_M_ARM) || defined(_M_ARM64))

        __iso_volatile_store8(vptr, 0);

#else

        *vptr = 0;

#endif

        vptr++;
        cnt--;
    }

#endif // _M_AMD64

    return ptr;
}

#endif
```

We find again the **FORCEINLINE** macro, in this case on the implementation of the **RtlSecureZeroMemory()** function. The implementation of this function is extremely performance sensitive, so the function implementation also uses other powerful programming technique and supported feature of Microsoft C/C++ compilers called **Compiler Intrinsics**. Most functions that we use in our applications are stored in libraries. So, to use a function we do a reference to the binary file, the DLL or LIB, that store the functions and data structures that we will use in our softwares. With Microsoft C/C++ compilers there are functions that have special characteristics, and they are not available in any specific external library that we can do a reference, like a DLL or LIB, these special functions are built in the compiler, and that means **Intrinsic Functions**, **Compiler Intrinsics** or just **Intrinsics**.

If a function is an *intrinsic*, the block of code for that function is inserted inline, preferentially, to avoid the overhead of a function call. Inserted inline or not, the *intrinsics* permits a highly efficient machine instructions to be emitted for that function.

| Hardware architecture	Page with the list of intrinsics
ARM	**Intrinsics for ARM, click here.**
X86	**Intrinsics for x86, click here.**
X64(amd64)	**Intrinsics for x64 (amd64), click here.**
Intrinsics available on hardware architectures. *This is the page with the list of intrinsics that are available with the x86, x64 (amd64) and ARM architectures.*	**Intrinsics available on x86, x64 (amd64), and ARM, click here.**
Alphabetic listing of Intrinsic functions.	**Intrinsics in alphabetic order (A-Z), click here.**
Intel Intrinsics guide.	**https://software.intel.com/sites/landingpage/IntrinsicsGuide/#**

We must be aware that the use of *intrinsics* affects the portability because they are specific for hardware architectures and requires support of the compiler. The Microsoft Visual C++ offers certain *intrinsics* that can be used in more than one hardware architecture, such as the list of *intrinsics* that are available on the x86, x64 (amd64) and ARM hardware architectures.

Continuing to talk about **RtlSecureZeroMemory() function**, we are seen an example of implementation technically advanced, that is, use *intrinsics*, inline implementation via keywords that alter the standard behavior of the compiler via suggestions or requirements, and **#ifdef** / **#endif** directives, all in a short block of code. The important here is understanding and accepting that flexibility is what we will find all the time in the Windows API.

These same development resources are all available for us, for use in our source code and development of our software libraries. For example, we can use *intrinsics*, inline implementation, **#pragma** directives, **#ifdef** / **#endif** directives and much more we will find when look to the Microsoft Visual C++ product tools and Microsoft C/C++ compilers features. We cannot forget that these distinctly features of the Microsoft Visual C++ product tools and Microsoft C/C++ compilers are all available in addition of all C and C++ programming language features, standard and Microsoft-specific.

Now let us see a data struct and interesting development aspects. Still using the **winnt.h** header file, open the dropdown list with the data structures and chooses **_MEMORY_BASIC_INFORMATION** and the definition is showed in the code editor of IDE:
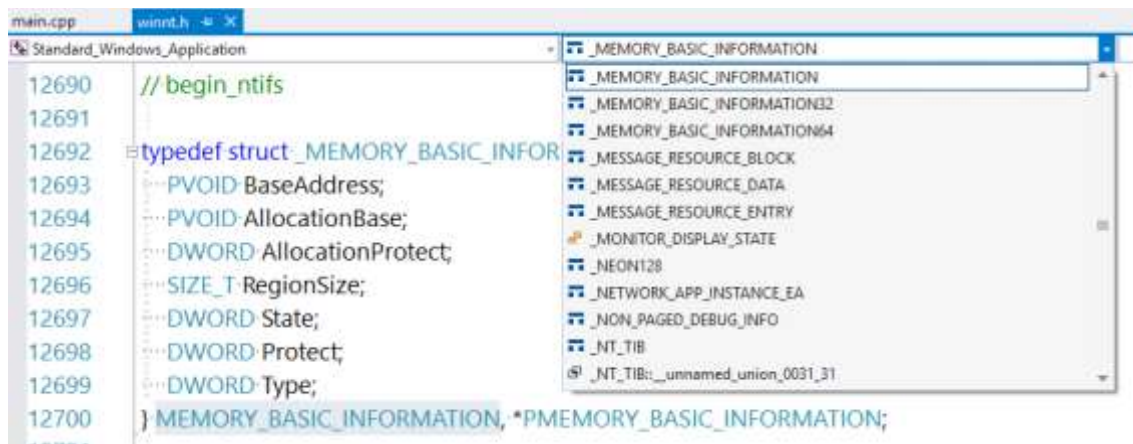
The typedef keyword

The use of the C/C++ **typedef** keyword is one more significant resource used through all Windows API. The fundamental idea of a **typedef** is to create new types using the same base definition and permits one or more levels of indirection to facilitates the evolution of the data type and the API's that are using this base type. In our example for the Windows API we have what we could see as an "internal" data struct, that is, the definition for the **_MEMORY_BASIC_INFORMATION**. We know that by default, any C/C++ data struct defined using the C/C++ **struct** keyword is public. Also, by default are public any members of such data struct defined with the C/C++ **struct** keyword.

Naming scheme for custom data types and members

These so-called "internal" data types and functions usually have a peculiar scheme for their "internal" names. Typically, the name of data structs defined with C/C++ **struct** keyword begins with one or more underline characters and uses uppercase for all letters in the name, the public struct members defined within the data struct uses Pascal-case style for naming their members. But within these data structs we can also find members with names using a slightly different scheme for naming. This way is used for showing that the member has a distinct acting on the data struct composition. For example, the 64-bit specialized implementation named **_MEMORY_BASIC_INFORMATION64**, has two data members, the

__**alignment1**__ and __**alignment2**__, used to help the compiler on keep user-defined data precisely aligned on 16-bit blocks of data distributed sequentially on memory. To inform the compiler of this demand about the alignment, the **_MEMORY_BASIC_INFORMATION64** is declared using __**declspec(align(16))**__, that is, the __**declspec**__ keyword with the **align** Microsoft-specific storage-class attribute and this informs the compiler to emit instructions for deal with the alignment tasks.

```
typedef struct DECLSPEC_ALIGN(16) _MEMORY_BASIC_INFORMATION64 {
    ULONGLONG BaseAddress;
    ULONGLONG AllocationBase;
    DWORD     AllocationProtect;
    DWORD     __alignment1;
    ULONGLONG RegionSize;
    DWORD     State;
    DWORD     Protect;
    DWORD     Type;
    DWORD     __alignment2;
} MEMORY_BASIC_INFORMATION64, *PMEMORY_BASIC_INFORMATION64;
```

DATA TYPE ALIGNMENT

The RAM memory is organized in addresses in the range of 0 until N-1, and with four gigabytes of memory with have addresses (2^32-1) in the range of 0 to 4,294,967,295, and each memory address has one byte of size, so, if we need to store eight bytes, we need eight memory addresses, preferentially in sequence, that is, memory address 1, memory address 2, memory address 3, and so on. For simple data types, the compiler assigns addresses that are multiples of the size in bytes of the data type. For example, the **bool** and **char** are aligned on one byte boundaries, **short** on two bytes, **int** on four bytes, **long long**, **double** and **long double** on eight bytes. The compilers attempt to emit instructions to allocate data in a way that prevents misalignment.

Now, for better learning about alignment, we are using a custom data struct for hypothetical scenarios. Initially without any member:

```
#pragma region Alignment for a struct
struct _Special_Data_For_Aligment { } Alignment_Example;
#pragma endregion
```

The return value using the **sizeof** operator is 1 (one) byte:

```
int32_t sizeOfStructInBytes { sizeof( Alignment_Example ) };
```

Now, using the same custom data struct but declared using the **alignas** C++11 operator:

#pragma region Alignment for a struct
struct alignas(16) _Special_Data_For_Aligment { } Alignment_Example;
#pragma endregion

The return value using the **sizeof** operator are 16 (sixteen) bytes:

int32_t sizeOfStructInBytes { sizeof(Alignment_Example) };

For custom data structures defined using C/C++ struct, class or other keywords, the compiler automatically tries to align to the nearest number power of two. As a hypothetical scenario, if a custom data struct has eleven bytes of size in total, the compiler emits one or more complementary data members until achieve the nearest number power of two for the total size adequated for accommodation in memory of instances of the custom data struct. For this scenario of a custom data struct with eleven bytes size in total, except if a specific alignment was explicitly defined for the custom data struct, the total size for the alignment will be the nearest number power of two considering the base size in bytes used for the alignment. To better understand this scenario, here we have a custom data struct with three data members with the eleven size in bytes in total, **double (8)**, **char16_t (2)** and **char(1)**:

struct _Special_Data_For_Aligment {

      double dataMemberOne {};
      char16_t dataMemberTwo {};
      char dataMemberThree {};

} Alignment_Example;

In this example we are using the **std::alignment_of<T>** class, available via **<type_traits>** header file, and with it we can query the alignment of a data type, that is for our hypothetical scenario a custom data struct:

```
constexpr int32_t sizeOfStructInBytes { sizeof( Alignment_Example ) };
constexpr int32_t alignmentSize { alignment_of<_Special_Data_For_Aligment>::value };
```

As we can view, the size for the alignment is based on the size of the **double** type, that have the bigger size among all data members. On our hypothetical scenario, the nearest number power of two that can be used for the alignment of an instance of the custom **data struct**, is the number sixteen, as we can read in the image of the **Locals** window. But, as we seen in the example, exist the possibility to inform the compiler of that a specific alignment value, different of default, should be used for certain custom **data structs**. When the data is stored in a memory address that is aligned with its size, we have a data that is *naturally aligned*, otherwise it is misaligned. For Microsoft C/C++ compilers, a compiler option for a default alignment is available. We can see an image with this option and the values that can be used:
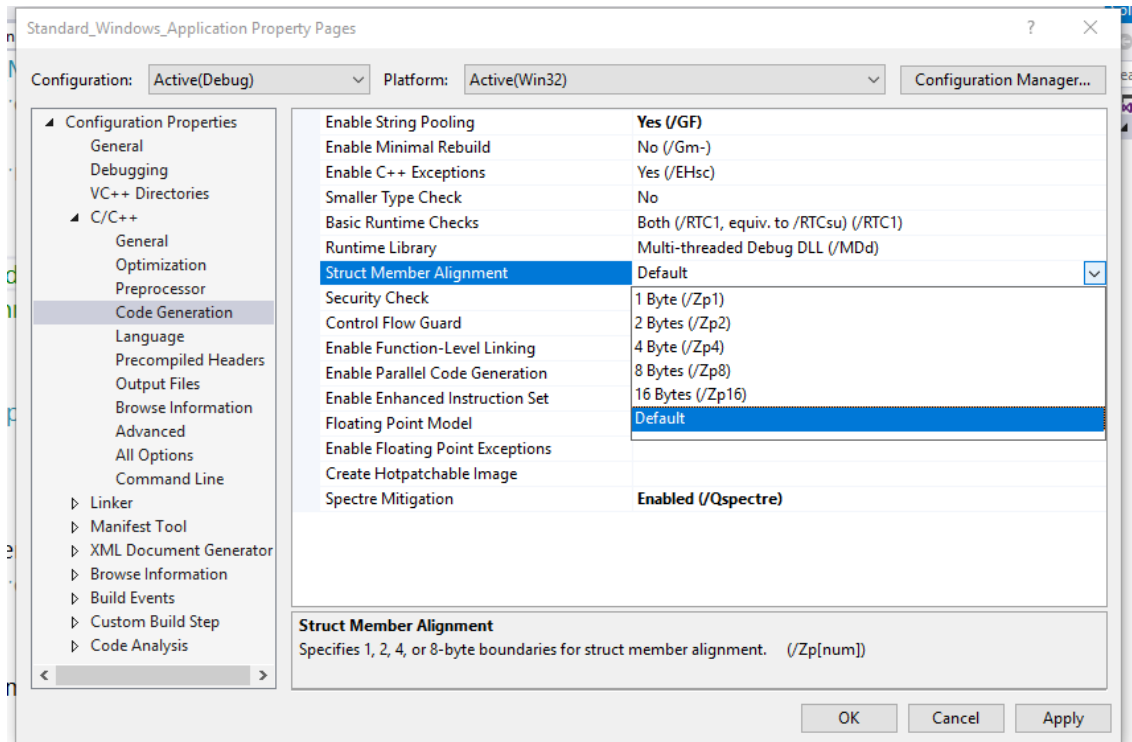
Calculates the natural alignment of a memory address

A CPU executes instructions that work on data stored in memory addresses. In addition to its address in memory, a single data also has a size that is expressed in bytes. A data is *naturally aligned* if the addresses are aligned to its size and misaligned otherwise. For example, an 8-byte floating-point is *naturally aligned* if the address used to identify it, is aligned to 8. To check if a specific memory address is naturally aligned with some number, we must check if the remainder of a division is zero. We must use this formula **(memory address % + alignment size)**, if the result is 0 (zero) the data type is **naturally aligned** with the **alignment size**.

constexpr uint32_t sizeOfStructInBytes { sizeof(Alignment_Example) };

constexpr uint32_t alignmentSize { alignment_of<_Special_Data_For_Aligment>::value };

_Special_Data_For_Aligment*   startAddress   {   (addressof<   _Special_Data_For_Aligmen Alignment_Example)) };

_Special_Data_For_Aligment* endAddress { (addressof< _Special_Data_For_Aligmen Alignment_Example) + (sizeOfStructInBytes - 1ui32)) };

bool isAligned { (reinterpret_cast< uint32_t >(endAddress) % alignmentSize) == 0ui32 };

startAddress = nullptr, endAddress = nullptr;

On the following image we can see the boolean variable isAligned with the true value as the result of the verification for *naturally aligned*.
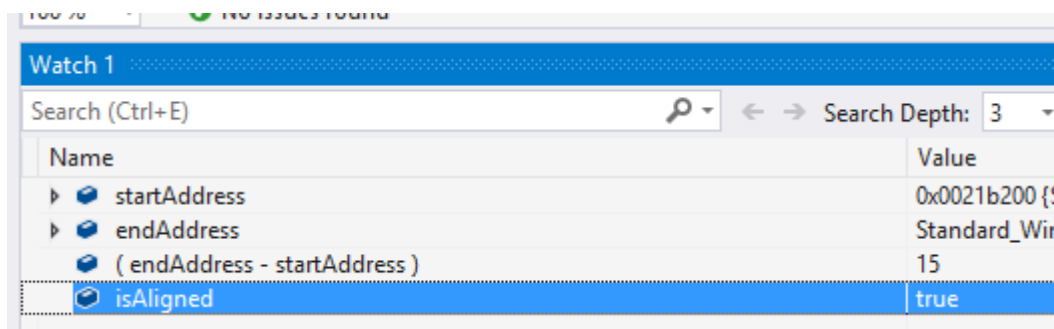
Microsoft Visual Studio 2015 and earlier versions

Prior to Microsoft Visual Studio 2015 we have to use two of Microsoft-specific keywords as options to specify alignment different of default. These keywords are:

- the __**alignof** operator keyword.
- the __**declspec(alignas)**, that is, the **alignas** type specifier keyword.

Since Microsoft Visual Studio 2015, we should use the C++11 standard keywords, because this also helps code portability. These new keywords behave in the same way as the Microsoft-specific extensions:

- the **alignof** keyword (C++11).

- the **alignas** keyword (C++11).

When working with more low-level options available in any C/C++ compiler, such as specific for the alignment, we must be conscious that this is a way to take advantage of a hardware specific features, and this create a less portable code, but potentially more powerful results.

A new public Windows data type

With the new **data struct** defined, the next step is to consider it to be a **typedef**. As a new *Windows Data Type* also means creates new public names for the new "internal" data type that reflects the base data type. In our example the **_MEMORY_BASIC_INFORMATION** is the "internal" data type, but as part of the Windows API, we should preserve the flexibility required by updates. Here we have **MEMORY_BASIC_INFORMATION** and ***PMEMORY_BASIC_INFORMATION**.

The **MEMORY_BASIC_INFORMATION** is the public *Windows Data Type* that applications should use in their source codes. Except for scenarios that are explicitly described on Microsoft official documentation or paper, we should never consider use the "internal" name, like **_MEMORY_BASIC_INFORMATION**. Using the "internal" name in our source codes will difficult our work because the updates that Microsoft eventually needs to do in the Windows API. The **_MEMORY_BASIC_INFORMATION** is a public data struct as seen by the programming language, we known that, but this is another special characteristic of the work with Windows API, discipline and knowledge. Not everything that we view or that we can access, we should use.

In the case of data types defined with C/C++ **struct** keyword, is also common the access via pointer. This is also another common characteristic of the Windows API *Windows Data Types*, and in our example, we can find the ***PMEMORY_BASIC_INFORMATION** pointer data type, that is a new type for the **_MEMORY_BASIC_INFORMATION** data struct. If we look through header files of the Microsoft Windows API, we will find these same implementation patterns. If we are interested in use the ***PMEMORY_BASIC_INFORMATION** pointer or **MEMORY_BASIC_INFORMATION** data struct, you can define in your custom API new typedefs for the **_MEMORY_BASIC_INFORMATION** "internal" data type.

```
#pragma region My Custom typedefs for _MEMORY_BASIC_INFORMATION internal Windows Data Type.
typedef _MEMORY_BASIC_INFORMATION MemoryBasicInformation, * PtrToMemoryBasicInformation;
#pragma endregion

/*  All fields initialized with their default  values. */
MemoryBasicInformation memBasicInfo {};
SIZE_T sizeInBytes { sizeof( MemoryBasicInformation ) };

/*  All fields initialized with zero value. */
PtrToMemoryBasicInformation ptrToMemBasicInfo {
```

```
reinterpret_cast < PtrToMemoryBasicInformation >( HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMO
sizeof( MemoryBasicInformation ) ) )

};

        /*

        ... All logic of the software ...

        */

/*

Release of resources.

Include tasks like:

-> remove from memory blocks information that can potentially be security critical, such as Ids, nam
cryptographic related data, as example.
-> free allocated memory blocks.

*/
if ( ptrToMemBasicInfo != nullptr ) {

            SecureZeroMemory( reinterpret_cast< PVOID >( &memBasicInfo ), ( ( SIZE_T ) size
MemoryBasicInformation ) ) );
            SecureZeroMemory( reinterpret_cast< PVOID >( ptrToMemBasicInfo ), ( ( SIZE_T ) size
MemoryBasicInformation ) ) );

            HeapFree( GetProcessHeap(), 0i32, reinterpret_cast < LPVOID >( ptrToMemBasicInfo ) );

};

ptrToMemBasicInfo = nullptr;
```

WINBASE.H HEADER FILE

OK, now we have seen that header files are a critical piece of any API written in C/C++ programming language, and the Microsoft Windows API is written in C/C++. Recently was placed available by Microsoft an API for the WinRT platform entirely make of header files, the C++/WinRT. This is an excerpt of the Microsoft official documentation about the C++/WinRT. We can read the documentation accessing this web address: https://docs.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/ .

As we can read in this introductory text excerpt of Microsoft official documentation about the C++/WinRT projection, **it was made using the standard modern C++17 language resources, it is a C++ language projection for Windows Runtime APIs, implemented as a header-file-based library, AND designed to provide us with first-class access to the modern Windows API.**

C++/WinRT is an entirely standard modern C++17 language projection for Windows Runtime (WinRT) APIs, implemented as a header-file-based library, and designed to provide you with first-class access to the modern Windows API. With C++/WinRT, you can author and consume Windows Runtime APIs using any standards-compliant C++17 compiler. The Windows SDK includes C++/WinRT; it was introduced in version 10.0.17134.0 (Windows 10, version 1803).

C++/WinRT is for any developer interested in writing beautiful and fast code for Windows. Here's why.

With this in mind, we can better understand the importance of the header files in the organization, maintenance, and evolution of the Microsoft Windows development as a software platform.

Now, just to finish this part of the tour through the Windows API, opening the **winbase.h** header file, we will see that it has about ten thousand (10.000) lines of code, including all the comments. As scenario for example of functions, data structs, and compiler features we was using the items in the **winbase.h** header file that are for backup systems. Checking the Microsoft official documentation for the Windows API about the backup system, we see the **winbase.h** and **winnt.h** header files cited as required for working with this technology of Microsoft Windows API. If we click in the links with the names of the **winbase.h** and **winnt.h** header files, we will see functions, data structs and other items that are available for work with the backup system technology. We must be aware that APIs on Microsoft Windows API are updated with a vigorous rhythm, so it is common to find information on the documentation that is not rigorously in sync

with the Windows API version that we are using in our project. The inverted scenario it is also true. My recommendation is to create a private knowledge base, using Microsoft OneNote for example, Microsoft Word and Microsoft Visual Studio projects with comments, to help through the years, and this recommendation is also valid for WinRT and .NET Framework / .NET Core, and programming languages for examples.

For access this example of the Windows API for backup system as appears in the following image, we should access this page https://docs.microsoft.com/en-us/windows/desktop/api/_backup/ . Here are the links for the pages with documentation for the header files https://docs.microsoft.com/en-us/windows/desktop/api/winbase/index and https://docs.microsoft.com/en-us/windows/desktop/api/winnt/index .

If we are interested in specific information about backup technologies of Microsoft operation system, we should access this page https://docs.microsoft.com/en-us/windows/desktop/backup .

See you on next publication!!! 😊 Thank you!!! 😊