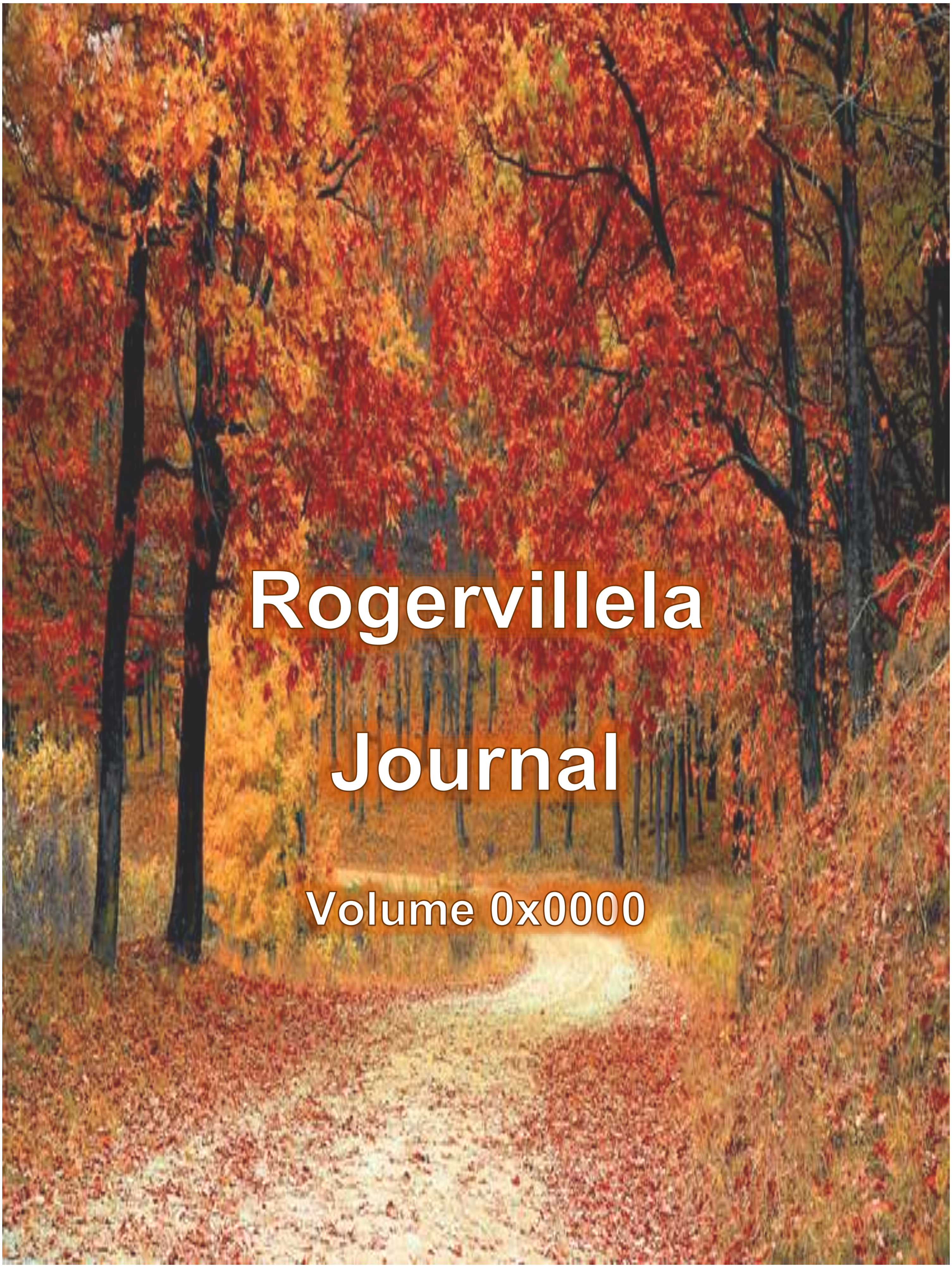


# Rogervillela Journal

Volume 0x0000

# TABLE OF CONTENTS

# INTRODUCTION

by Roger Villela
Author and Technical Educator

RogerVillela Journal

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.

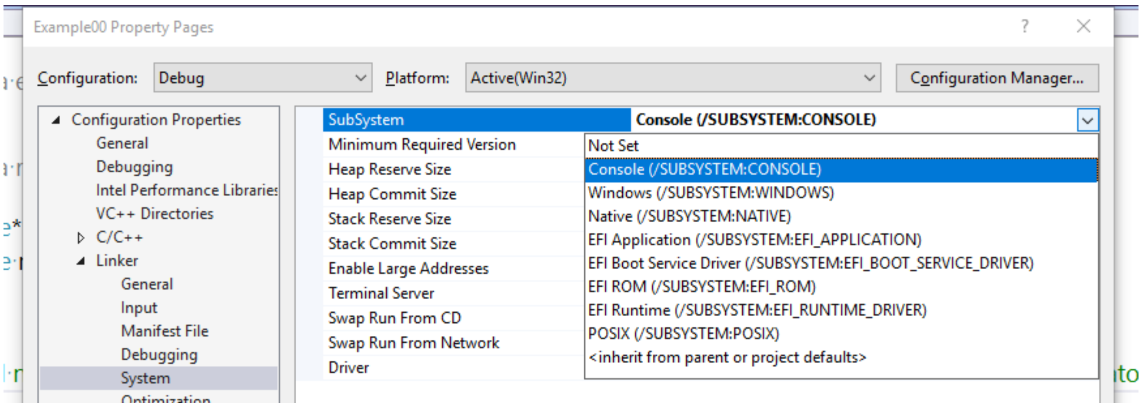
# ORGANIZATION

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*.



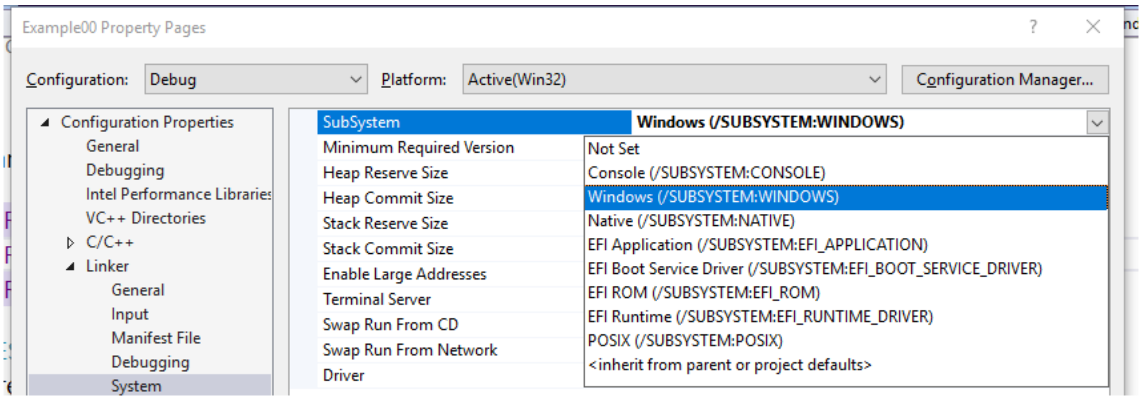
Example00 Property Pages

Configuration: Debug  Platform: Active(Win32)  Configuration Manager...

SubSystem: **Console (/SUBSYSTEM:CONSOLE)**

Not Set
Console (/SUBSYSTEM:CONSOLE)
Windows (/SUBSYSTEM:WINDOWS)
Native (/SUBSYSTEM:NATIVE)
EFI Application (/SUBSYSTEM:EFI_APPLICATION)
EFI Boot Service Driver (/SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER)
EFI ROM (/SUBSYSTEM:EFI_ROM)
EFI Runtime (/SUBSYSTEM:EFI_RUNTIME_DRIVER)
POSIX (/SUBSYSTEM:POSIX)
<inherit from parent or project defaults>

The **IsWindows7SP1OrGreater** sample solution has one sample project named **Example00** for *Microsoft Windows subsystem (GUI and DLL)*.



Example00 Property Pages

Configuration: Debug  Platform: Active(Win32)  Configuration Manager...

SubSystem: **Windows (/SUBSYSTEM:WINDOWS)**

Not Set
Console (/SUBSYSTEM:CONSOLE)
Windows (/SUBSYSTEM:WINDOWS)
Native (/SUBSYSTEM:NATIVE)
EFI Application (/SUBSYSTEM:EFI_APPLICATION)
EFI Boot Service Driver (/SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER)
EFI ROM (/SUBSYSTEM:EFI_ROM)
EFI Runtime (/SUBSYSTEM:EFI_RUNTIME_DRIVER)
POSIX (/SUBSYSTEM:POSIX)
<inherit from parent or project defaults>

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 appropriate 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)

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

## SOLUTIONS

- SYSTEM SERVICES - MEMORY MANAGEMENT - GENERAL
    - WE – Windows\Windows.System.MemoryManagement\General\
        - C
            - CopyMemory\CopyMemory.sln
        - S
            - SecureZeroMemory\SecureZeroMemory.sln
- SYSTEM SERVICES – INFORMATION – VERSION HELPER FUNCTIONS
    - WE – Windows\Windows.System.Information\
        - I
            - IsWindows7SP1OrGreater\IsWindows7SP1OrGreater.sln

# ABOUT MICROSOFT WINDOWS API

It is a public set of **data types**, **data structures**, and **functions** that we should use for developing our software components, software libraries, and software applications. An API is the public face of the complex engineering details for the inner workings of any software or hardware platform.

When working with **Microsoft Windows API** the first set of aspects that we face are the various **data types**, **data structures**, **functions** and the **concepts** that are directly or indirectly associated with these elements. So, beginning from exploring some of the main header files of the Microsoft Windows API, helps us on acquiring knowledge about standard elements, patterns and engineering implementation details of the infrastructure that are the base of any technology on Microsoft Windows operation system per se, also used for the implementation of specialized execution environments like CLR and WinRT, and other Microsoft products developed having Microsoft Windows operating system as platform, such as Microsoft SQL Server, Microsoft Office 365, Microsoft Visual Studio, Microsoft Azure Cloud platform, just to cite a few of them. As the Microsoft Windows operating system, the Microsoft Windows API was conceived and constructed to be platform agnostic and to be updated over the time using this same principle. Currently, the Windows API has main support for 32-bit and 64-bit hardware platforms and their respective Microsoft Windows implementations, but in a not so distant time 😊, the Windows API also had support for 16-bit hardware platforms and the Microsoft Windows implementations. So, when we open and read any base header file of the Microsoft Windows API, we will find the use of **typedef** keyword of C/C++ programming languages for the definition of specific **data types** and **data structures**, and any of these are also a *Windows Data Type*. The denotation *Windows Data Type* is used for **data types** and **data structures** that was designed, implemented and placed available to represent any aspect of the infrastructure of Microsoft Windows operating system, the high level specialized functionalities, and that can be aggregated to the Microsoft Windows operating system. For example, we have fundamental *Windows Data Types* like BOOL, BYTE, WORD, DWORD, INT, UINT, FLOAT, and even the VOID, and here is an excerpt of Microsoft Windows SDK header files with examples of fundamental *Windows Data Types*:

```
typedef unsigned long DWORD;

typedef int BOOL;

typedef unsigned char BYTE;

typedef unsigned short WORD;
typedef float FLOAT;

typedef int INT;

typedef unsigned int UINT;

#ifndef VOID

#define VOID void

typedef char CHAR;

typedef short SHORT;

typedef long LONG;

#if !defined(MIDL_PASS)

typedef int INT;

#endif

#endif

//

// UNICODE (Wide Character) types

//


#ifndef _MAC

typedef wchar_t WCHAR; // wc, 16-bit UNICODE character

#else

//some Macintosh compilers don't define wchar_t in a convenient location, or define it as a char

typedef unsigned short WCHAR; // wc, 16-bit UNICODE character

#endif
```

We also have *Windows Data Types* that are pointers to these other *Windows Data Types*, and these are also defined using **typedef** keyword of C++ programming languages:

```
typedef FLOAT            *PFLOAT;

typedef BOOL near          *PBOOL;

typedef BOOL far          *LPBOOL;

typedef BYTE near          *PBYTE;

typedef BYTE far          *LPBYTE;
typedef int near          *PINT;
typedef int far           *LPINT;
typedef WORD near         *PWORD;
typedef WORD far          *LPWORD;
typedef long far          *LPLONG;
typedef DWORD near        *PDWORD;
typedef DWORD far         *LPDWORD;
typedef void far          *LPVOID;
typedef CONST void far    *LPCVOID;
typedef unsigned int      *PUINT;
```

**(*) Do not worry about the words *near* and *far*, the purpose is related with the differences between 16-bit and 32-bit of Microsoft Windows implementation on 16-bit and 32-bit hardware platforms and will be explained through specialized topics on the publications about memory management. The important to remember for now is that *near* and *far* ARE NOT KEYWORDS of Microsoft C/C++ implementations, are just macros defined in a header file of Microsoft Windows SDK.**

Opening the **SecureZeroMemory** sample solution and the **main.cpp** source code file of the **Example00** sample project, we can see the use of *Windows Data Types*, and as integrant of the Windows API for memory management, we have the *SecureZeroMemory() function*, part of the *General* category. The *SecureZeroMemory() function* is declared and defined with the return value of PVOID (pointer to void) *Windows Data Type*, with the first parameter using the PVOID (pointer to void) *Windows Data Type*, and with the last parameter using the SIZE_T *Windows Data Type*. The SIZE_T *Windows Data Type* is defined C++ data type *unsigned long* for 32-bit platforms and Microsoft C++ specific data type *unsigned __int64* for 64-bit platforms:

```
PVOID SecureZeroMemory( PVOID pointer, SIZE_T count );
```

On the signature of **SecureZeroMemory()** function we have PVOID and SIZE_T *Windows Data Types*. The *Windows Data Type* PVOID, as we can guess by name 😊, is a pointer to the void built-in fundamental data type, that is part of the C/C++ programming languages. The PVOID *Windows Data Type* is defined as:

```
typedef void *PVOID;
```

The SIZE_T *Windows Data Type* is a **typedef** that is defined based on another *Windows Data Type*, the **ULONG_PTR**.

```
// SIZE_T used for counts or ranges which need to span the range of a pointer.  SSIZE_T is the signed
// variation.
typedef ULONG_PTR SIZE_T, *PSIZE_T;
typedef LONG_PTR SSIZE_T, *PSSIZE_T;
```

The ULONG *Windows Data Type* means *unsigned long*, a reference to its C/C++ fundamental base data type, and a description that helps known which base type is. The ULONG *Windows Data Type* is defined for 32-bit and 64-bit implementations, depending on the target Microsoft Windows implementation. Here we have an excerpt of the block of code that defines the ULONG_PTR *Windows Data Type*:

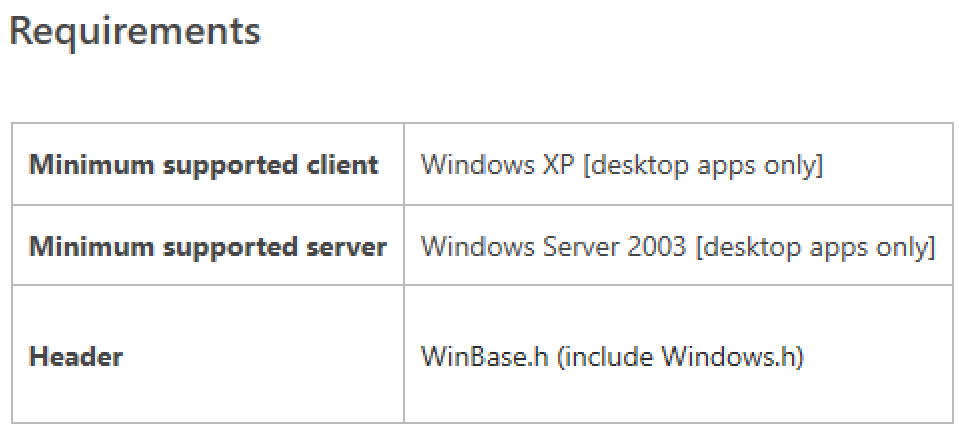
```
#if defined(_WIN64)
typedef unsigned __int64 ULONG_PTR, *PULONG_PTR;
#else
typedef _W64 unsigned long ULONG_PTR, *PULONG_PTR;
#endif
```

If the target implementation is a 64-bit Microsoft Windows implementation, the ULONG *Windows Data Type* is defined as the Microsoft C++ specific data type **unsigned __int64**. If the target implementation is a 32-bit Microsoft Windows implementation, the ULONG *Windows Data Type* is defined as C++ data type **unsigned long**.

**(*) The __int64 is a Microsoft-specific data type that is part of a group called *Microsoft sized integers*. For example, we have __int8, __int16, __int32, and the __int64 sized integer data types. These data types are supported only by Microsoft C/C++ compilers and the purpose is the portability of the code. Obviously, if we are creating a source code base for work with different**

**compilers, we will need to do an arrangement in the source code to check which is the compiler in use.**

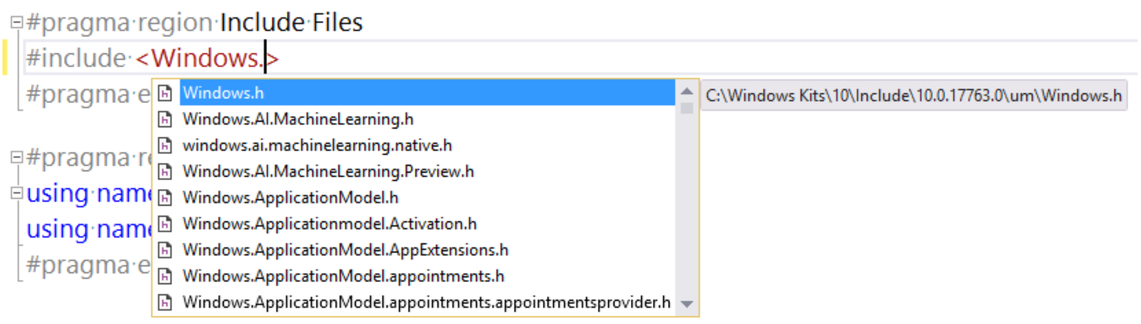
What we see here with these resources and features of compilers and programming languages, is a code implementation engineering technique for the creation of one or more layers of abstraction that helps us on the construction of the portability of large and complex source code bases at various levels. For example, when reading Microsoft official documentation, we can find information on the bottom of the pages with resumed information about the supported Microsoft Windows implementations. Here we have an example for the *SecureZeroMemory() Windows API function*:



Requirements

| | |
|---|---|
| **Minimum supported client** | Windows XP [desktop apps only] |
| **Minimum supported server** | Windows Server 2003 [desktop apps only] |
| **Header** | WinBase.h (include Windows.h) |

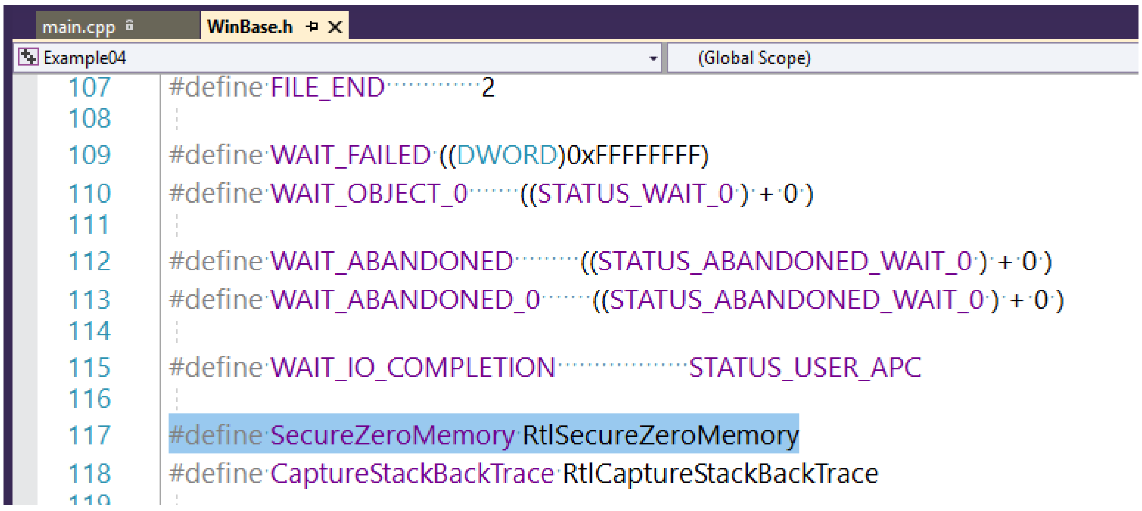
These are called **MINIMUM REQUIREMENTS** and inform about the minimum client and minimum server Microsoft Windows and platforms, such as Microsoft WinRT, that this Windows API element, the *SecureZeroMemory() function* in this example, is supported. These bottom page notes also inform the kind of **TARGET ENVIRONMENT** that this Windows API element is available for, and in this example the *SecureZeroMemory() function* can directly be used only within software applications that target the Microsoft Windows desktop environment. In this example, this means that the *SecureZeroMemory() function* cannot directly be included and used in a WinRT component source code or in an UWP software application source code. As final details, these bottom page notes also inform about the Microsoft Windows SDK header file that this function is declared or implemented depending of the case. On these information's about the *SecureZeroMemory() function*, it is declared within the *WinBase.h* header file that is included by *Windows.h* header file, and this means that we should access this function including *Windows.h* header file in our source code and not *WinBase.h* header file, except when explicitly indicated or required by the implementation model of the software component or software application. In these following images we can see the declaration of the *SecureZeroMemory() function*, in fact a macro, and an excerpt of the implementation, made by the *RtlSecureZeroMemory() function* in the *WinNT.h* header file. On the topic **About SecureZeroMemory() function** we can find the details. The initials **Rtl** means **Runtime Library** that is another concept that we talk about through the publications, and for now this is enough:



```
#pragma region Include Files
#include <Windows.>
#pragma e
#pragma r
using name
using name
#pragma e
```

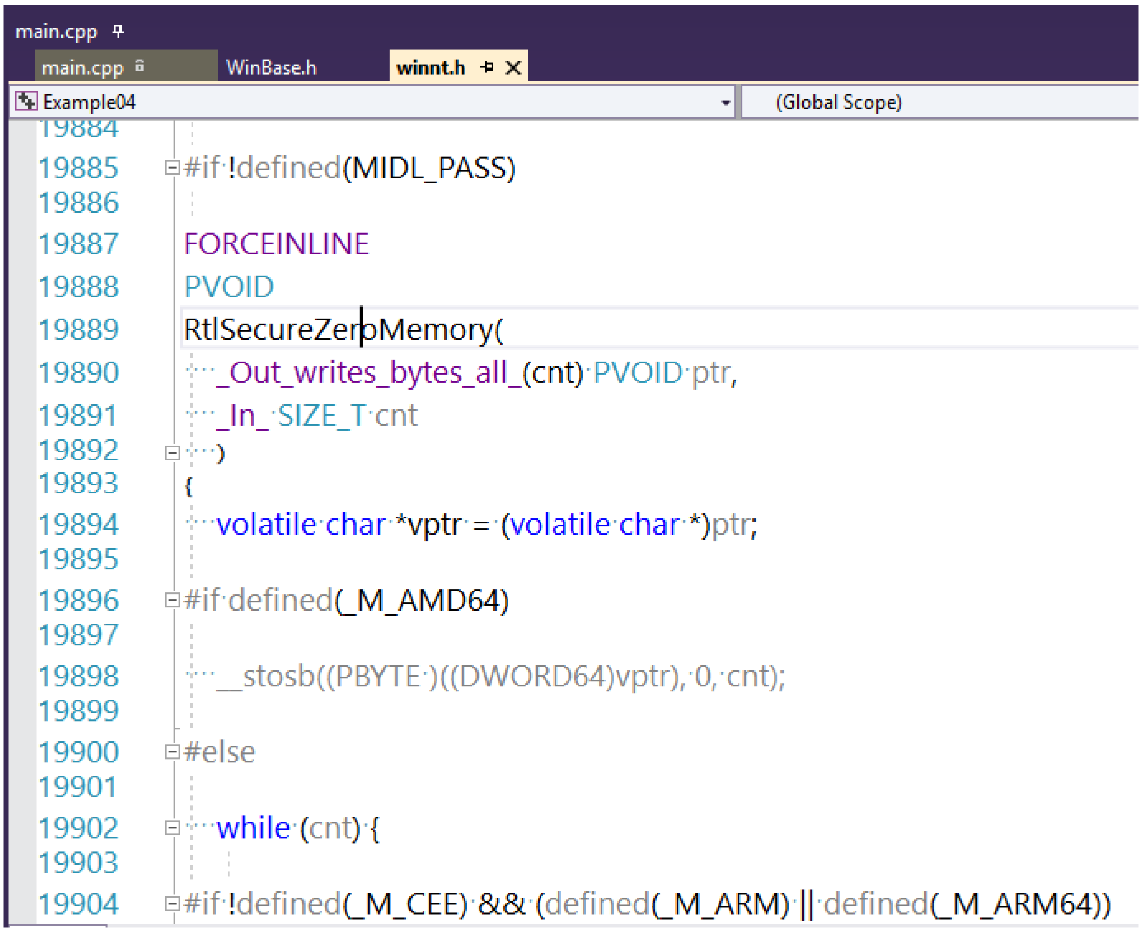
```
Windows.h
Windows.AI.MachineLearning.h
windows.ai.machinelearning.native.h
Windows.AI.MachineLearning.Preview.h
Windows.ApplicationModel.h
Windows.Applicationmodel.Activation.h
Windows.ApplicationModel.AppExtensions.h
Windows.ApplicationModel.appointments.h
Windows.ApplicationModel.appointments.appointmentsprovider.h
```

C:\Windows Kits\10\Include\10.0.17763.0\um\Windows.h

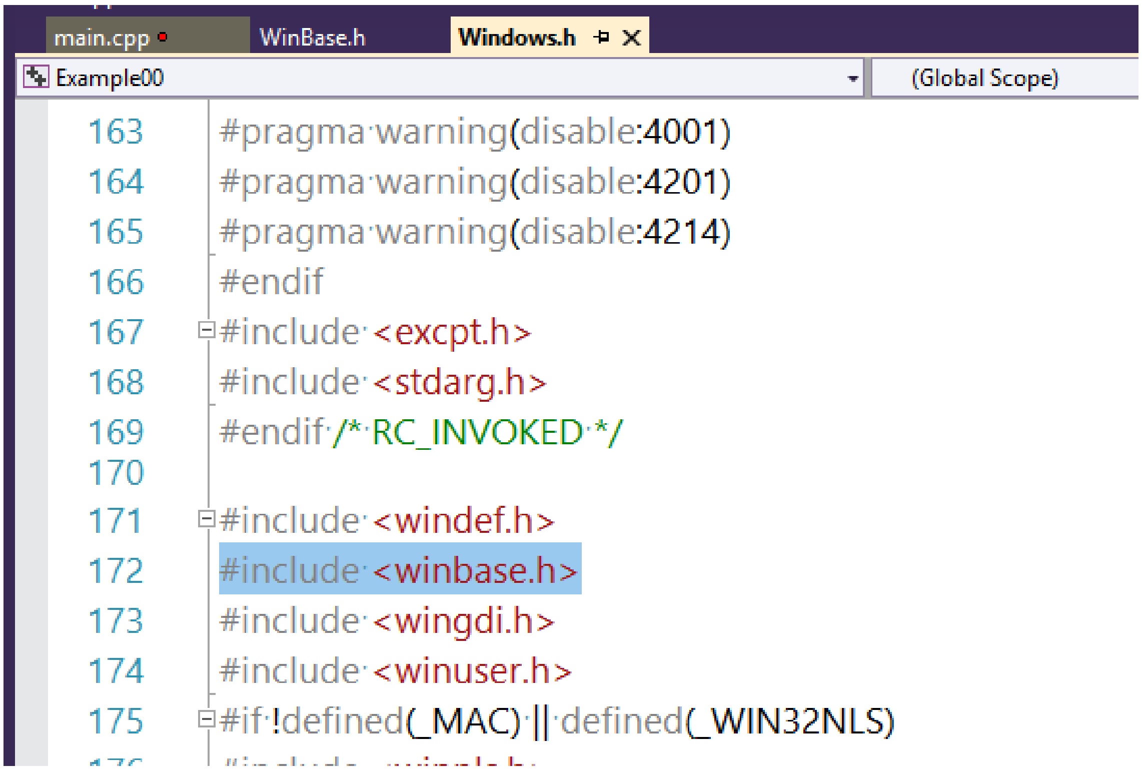


main.cpp | WinBase.h | Example04 | (Global Scope)

```
#define FILE_END            2

#define WAIT_FAILED ((DWORD)0xFFFFFFFF)
#define WAIT_OBJECT_0       ((STATUS_WAIT_0 ) + 0 )

#define WAIT_ABANDONED         ((STATUS_ABANDONED_WAIT_0 ) + 0 )
#define WAIT_ABANDONED_0       ((STATUS_ABANDONED_WAIT_0 ) + 0 )

#define WAIT_IO_COMPLETION                  STATUS_USER_APC

#define SecureZeroMemory RtlSecureZeroMemory
#define CaptureStackBackTrace RtlCaptureStackBackTrace
```



```
#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))
```



```
#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)
```

Returning to the main.cpp source code file of *SecureZeroMemory->Example00* sample project, we can read the use of **reinterpret_cast<T>** keyword (available since C++98 with improvements introduced in C++11/C++17) that is supported by modern C++ compiler implementations, and the use of the typical casting operation supported by C++ compiler implementations. If we are using an implementation of a C++ compiler that supports modern C++ features such as **reinterpret_cast<T>** keyword, we should opt for it, else we can use the typical syntax for the cast operation. The fundamental purpose of **reinterpret_cast<T>** keyword is to help the compiler to understands and reinforce rules when working with pointer operations in scenarios where the bit patterns of pointer of type A should be converted to pointer of type B and understood as the pointer to type B from that point in execution scope. In the main.cpp source code file of the Example00 sample project, we have used the **reinterpret_cast<T>** with the **typedef** BaseType for the standard C++ data type *std::uint32_t*, and with the *Windows Data Type* PVOID:

```
typedef uint32_t BaseType;

BaseType x { 72ui32 };

BaseType* pointerToX { &x };



pointerToX = ( reinterpret_cast< BaseType* >( SecureZeroMemory( reinterpret_cast< PVOID >(
pointerToX ), ( ( SIZE_T ) sizeof( BaseType ) ) ) ));
```

If we had used casting syntax for the two situations where the **reinterpret_cast<T>** keyword was used, it works as well, and the same block of code appears like this:

```
typedef uint32_t BaseType;

BaseType x { 72ui32 };

BaseType* pointerToX { &x };


pointerToX = ( ( BaseType* ) ( SecureZeroMemory( ( PVOID )( pointerToX ), ( ( SIZE_T ) sizeof(
BaseType ) ) ) ));
```

On this expression with the call for the **SecureZeroMemory() function**, the only cast operation that should be explicitly expressed is because of the return of the **SecureZeroMemory() function**, that is, the PVOID *Windows Data Type*, in fact, a C++ *void** pointer aggregated data type, and the target pointer type pointerToX that is a pointer to BaseType*, in fact, a pointer to C++ standard data type *std::uint32_t*. The other two parameters also need a cast operation, but to help in the productivity the Microsoft C/C++ compilers offers the implicit cast feature, and this means that where the compilers could find the necessity of a cast operation, it tries to resolve automatically. When the implicit cast operation was not possible to be applied by the Microsoft C/C++ compiler, we should explicitly inform via the appropriated syntax and data types. On our example, if the casting operation is applied where the **reinterpret_cast<T>** keyword appears, it works as well, and the same block of code appears like this:

```
typedef uint32_t BaseType;

BaseType x { 72ui32 };

BaseType* pointerToX { &x };


pointerToX = ( ( BaseType* ) ( SecureZeroMemory( ( pointerToX ),  sizeof( BaseType ) ) ) );
```

Maybe now we are asking why not using the C/C++ fundamental built-in **data types** and **data structures** such as struct and class? Which is the reason to define new types for existing types or newly defined types?

Well, the primary answer here is portability.

Creating a gigantic product as an operating system requires that the design choices helps the **return of investment**, and an operating system is a **huge investment**, and we also have this type of design choices with the advanced data management platforms, such as Microsoft SQL Server and Microsoft Azure Cosmos DB. We can install the Microsoft SQL Server 2019 in a Microsoft Windows 10 Professional environment and in a Microsoft Windows Server 2019 Server, so, if we had created database tables, we can see the same fundamental data types on both installations of Microsoft SQL Server, and this same broadly available logic applies to queries and other data management objects that was made for this specialized data management platforms, of course we also have specialized product features that are made to work only in a specialized environment like the Microsoft Windows Server 2019, but this reflects how correctly the product is designed and implemented, separating technological features as the requirements of the client business expands. Using this abstract and broadly way of thinking, we have the motivation and the opportunity to create products that are more portable, and their functionalities will help on the return of the investments made on the platforms.

## More Specialized Windows Data Types

As the following excerpts of code with the definitions for SYSTEMTIME, FILE_INFO_BY_HANDLE_CLASS, CREATE_THREAD_DEBUG_INFO and CREATE_PROCESS_DEBUG_INFO **data structures** show, we also have specialized **data structures** for more high level elements and functionalities of the Microsoft Windows operating system. In particular, the FILE_INFO_BY_HANDLE_CLASS shows another interesting aspect of the works with the Microsoft Windows API, that is how to deal with the improvements between Microsoft Windows implementations and differences in the **data structures** and **data types**. In the definition of FILE_INFO_BY_HANDLE_CLASS **data structure** we can read **#if** / **#endif** directives for checking the Microsoft Windows main version number of the implementation and depending of this information elements are included or not included. For example, if we are implementing a specific feature to be part of our software application or our software library that uses a **data structure** for a respective feature of Microsoft Windows operating system that has differences between Microsoft Windows operating system implementation versions, it is our responsibility to check the differences on the **data structure** for each version of Microsoft Windows and prepare our software application or software library to work with such requirements. In the following excerpts of code, we can see this behavior in the definition of the FILE_INFO_BY_HANDLE_CLASS **data structure** where these checks are made via **#if** / **#endif** directives in the source code base being compiled for. First, the FILE_INFO_BY_HANDLE_CLASS **data structure** is only defined if we are compiling for the _WIN32_WINNT_LONGHORN as the minimum Microsoft Windows implementation version, this signifies that this **data structure** does not exist in preceding Microsoft Windows implementation versions. The subsequent checks for the data structure use the **#if** / **#endif** directives to verifies if the Microsoft Windows implementation versions that we are compiling for, are _WIN32_WINNT_WIN8 or _WIN32_WINNT_WIN10_RS1 to includes certain fields and respective values:

```
#if (_WIN32_WINNT >= _WIN32_WINNT_LONGHORN)
typedef enum _FILE_INFO_BY_HANDLE_CLASS {
    FileBasicInfo,
    FileStandardInfo,
    FileNameInfo,
    FileRenameInfo,
    FileDispositionInfo,
    FileAllocationInfo,
    FileEndOfFileInfo,
    FileStreamInfo,
    FileCompressionInfo,
    FileAttributeTagInfo,
    FileIdBothDirectoryInfo,
    FileIdBothDirectoryRestartInfo,
    FileIoPriorityHintInfo,
    FileRemoteProtocolInfo,
    FileFullDirectoryInfo,
    FileFullDirectoryRestartInfo,
#if (_WIN32_WINNT >= _WIN32_WINNT_WIN8)
    FileStorageInfo,
    FileAlignmentInfo,
    FileIdInfo,
    FileIdExtdDirectoryInfo,
    FileIdExtdDirectoryRestartInfo,
#endif
#if (_WIN32_WINNT >= _WIN32_WINNT_WIN10_RS1)
    FileDispositionInfoEx,
    FileRenameInfoEx,
#endif
```

```
    MaximumFileInfoByHandleClass
} FILE_INFO_BY_HANDLE_CLASS, *PFILE_INFO_BY_HANDLE_CLASS;
#endif
```

As we have read about, for the agnostic and abstract purpose of the operating system and the API **data types** on respect of the hardware target platform's, the uses of the **typedef** keyword is a constant choice, even for **data structures** that represent concepts that are more common in our day life, but not less important, like the representation of the system date and time:

```
//
// System time is represented with the following structure:
//
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```

Another important and interesting group with *Windows Data Types* is that used for the support of the diagnostics of the applications and the environment around these, like the debugging activity. Different from what most people might think, the debugging features are not only available at specialized software development environments. Debugging is a feature available as part of the operating system services and can be used like any other functionality. Of course, certain features require a prepared development environment for debugging, but we can do a lot even without a disciplined specialized development environment for debugging. Independently of the level of the discipline used for the configuration with focus on diagnostics, the importance of it is reflected on the existence of the specialized **data types**, **data structures** and **API's** for the most fundamental operating system objects, such the process and the thread operating system objects:

```
typedef struct _CREATE_THREAD_DEBUG_INFO {
  HANDLE hThread;
   LPVOID lpThreadLocalBase;
   LPTHREAD_START_ROUTINE lpStartAddress;
} CREATE_THREAD_DEBUG_INFO, *LPCREATE_THREAD_DEBUG_INFO;

typedef struct _CREATE_PROCESS_DEBUG_INFO {
   HANDLE hFile;
   HANDLE hProcess;
   HANDLE hThread;
   LPVOID lpBaseOfImage;
   DWORD dwDebugInfoFileOffset;
   DWORD nDebugInfoSize;
   LPVOID lpThreadLocalBase;
   LPTHREAD_START_ROUTINE lpStartAddress;
   LPVOID lpImageName;
   WORD fUnicode;
} CREATE_PROCESS_DEBUG_INFO, *LPCREATE_PROCESS_DEBUG_INFO;
```

These few examples of **data structures** are here to show the enormous and broadly spectrum covered by Microsoft Windows API, but it follows patterns since the conceptions until the implementation, because the Microsoft Windows API reflects the way of work of the Microsoft Windows operating system, so with each step in this tough learning curve, we are also learning about the inner workings of the architecture and implementation engineering of Microsoft Windows operating system.

# ABOUT MICROSOFT WINDOWS VERSION NUMBER

This is critical to understands and accept, every element of the Microsoft Windows API reflects aspects and services (functionalities) of the Microsoft Windows operating system product implementations. So, if we try to include **data structures** or **data types** that is not available or supported for such implementation, it is our fault and our responsibility for the consequences.

This topic is not an extensive discussion about the versioning mechanism of the Microsoft Windows operating system, but an introduction about aspects of the version numbers and the Windows API. The versioning is an extensive topic and when discussing about it we must include the CLR and WinRT, and this will be made gradually. 😊

Currently, the main version number of Microsoft Windows implementation is defined as simple macros via **#define** directive for hexadecimal numbers. We will find these patterns, that is, number values expressed in hexadecimal representation and associated with macros, not only within Microsoft Windows API, but for other Microsoft products and their respective C/C++ API's. In fact, we should follow this good practice of using hexadecimal numbers for every literal, and not only for C/C++ programming, but for CLR, UWP/WinRT coding too, independent of programming language choosed. Here we have examples of these number values expressed in hexadecimal representation and their associated macros:

```
//
//_WIN32_WINNT version constants
//
#define _WIN32_WINNT_NT4            0x0400
#define _WIN32_WINNT_WIN2K          0x0500
#define _WIN32_WINNT_WINXP          0x0501
#define _WIN32_WINNT_WS03           0x0502
#define _WIN32_WINNT_WIN6           0x0600
#define _WIN32_WINNT_VISTA          0x0600
#define_WIN32_WINNT_WS08            0x0600
#define _WIN32_WINNT_LONGHORN       0x0600
```

```
#define _WIN32_WINNT_WIN7               0x0601
#define _WIN32_WINNT_WIN8               0x0602
#define _WIN32_WINNT_WINBLUE            0x0603
#define _WIN32_WINNT_WINTHRESHOLD       0x0A00 /* ABRACADABRA_THRESHOLD*/
#define _WIN32_WINNT_WIN10              0x0A00 /* ABRACADABRA_THRESHOLD*/
```

The uses of this programming technique using **#if** / **#endif** directives is applied at compile time ⌚, as we see in the definition of FILE_INFO_BY_HANDLE_CLASS enum. But we can also use a checking of "which is" the Microsoft Windows implementation version at run-time using Microsoft Windows API functions and **data structures** designed and implemented specifically for this task, like the new **Version Helper functions** of the Microsoft Windows API. The version helper functions are defined in the **versionhelpers.h** header file, and it was first distributed with Microsoft Windows SDK for Microsoft Windows 8.1. But the **versionhelpers.h** can also be used with projects that target earlier versions Microsoft Windows and support Microsoft Windows 2000 Professional or Microsoft Windows 2000 Server as the minimum Microsoft Windows implementations. If we look inside the **versionhelpers.h** header file, we will find various "Is" functions, like **IsWindowsXPOrGreater()**, **IsWindowsXPSP1OrGreater()**, **IsWindows7OrGreater()**, **IsWindows7SP1OrGreater()**, **IsWindows10OrGreater()**. Currently, except the **IsWindowsServer()** function that explicitly calls **VerifyVersionInfo()** function that is part of the Windows API, all these "Is" functions, calls just one function, the **IsWindowsVersionOrGreater()**, that also is part of the source code in the **versionhelpers.h** header file. Here is the full implementation of **IsWindowsXPOrGreater()** as is currently in the **versionhelpers.h** header file:

```
VERSIONHELPERAPI IsWindowsXPOrGreater() {
return IsWindowsVersionOrGreater(HIBYTE(_WIN32_WINNT_WINXP), LOBYTE(_WIN32_WINNT_WINXP),
0);


};
```

The **IsWindowsVersionOrGreater()** function is a kind of "all-in-one" function, that is, the other functions inform the same sequence of parameters but with different argument values for each specific Microsoft Windows version that is interested in. Internally, the "all-in-one" **IsWindowsVersionOrGreater()** function calls the **VerifyVersionInfo()** function that is part of Windows API and that is available since Microsoft Windows 2000, Professional and Server. Here is the full implementation of **IsWindowsVersionOrGreater()** as is currently in the **versionhelpers.h** header file:

```
VERSIONHELPERAPI IsWindowsVersionOrGreater(WORD wMajorVersion, WORD wMinorVersion, WORD
wServicePackMajor) {


    OSVERSIONINFOEXW osvi = { sizeof(osvi), 0, 0, 0, 0, {0}, 0, 0 };

    DWORDLONG  const dwlConditionMask = VerSetConditionMask( VerSetConditionMask(
VerSetConditionMask( 0, VER_MAJORVERSION, VER_GREATER_EQUAL), VER_MINORVERSION,
VER_GREATER_EQUAL), VER_SERVICEPACKMAJOR, VER_GREATER_EQUAL);


osvi.dwMajorVersion = wMajorVersion;

osvi.dwMinorVersion = wMinorVersion;

osvi.wServicePackMajor = wServicePackMajor;


return VerifyVersionInfoW(&osvi, VER_MAJORVERSION | VER_MINORVERSION |
VER_SERVICEPACKMAJOR, dwlConditionMask) != FALSE;

};
```

If we open the **IsWindows7SP1OrGreater** sample solution and the **main.cpp** source code file of the **Example00** sample project we can find an example of the use of **IsWindows7SP1OrGreater()** version helper function. In fact, this sample project is a kind of "Hello, World of Microsoft Windows", but using the GDI/GDI+ GUI.

## ABOUT SECUREZEROMEMORY FUNCTION

```
PVOID SecureZeroMemory( _In_ PVOID ptr, _In_ SIZE_T cnt );
```

The technical purpose of implementation of the **SecureZeroMemory()** function is to assign the zero value for each byte in a memory block and it is a safer implementation of the macro **ZeroMemory()**. In fact, the **SecureZeroMemory()** function is a macro for the **RtlSecureZeroMemory()** function.

```
#define SecureZeroMemory RtlSecureZeroMemory
```

**The implementation of the RtlSecureZeroMemory() function is provided inline.**

```
FORCEINLINE PVOID RtlSecureZeroMemory( PVOID ptr, _In_ SIZE_T cnt );
```

FORCEINLINE is a macro for the instruction __forceinline or __inline. The __inline and inline specifiers instructs the compiler to insert a copy of the body of the function instead of making the call. This means that the block of code in the assembly programming language that represents the function is inserted into each location where a call occurs for that function. When applied to a function, the __inline or inline specifiers, this does not mean that an automatic expansion occurs. The cost and benefit assessment of the expansion is accomplished by the compiler. When applied the __forceinline specifier, it is not the compiler that performs the evaluation but who wrote the code. That is, the uses of __forceinline specifier overlaps the compiler's analysis criterion and assumes the cost and benefit criterion of whoever wrote the code correctly. The definition of the macro FORCEINLINE includes a version check from Microsoft Visual Studio, and the definition uses the MSC_VER macro to verify that Microsoft Visual Studio used to compile the project is greater than or equal to Microsoft Visual Studio 2013. If positive, uses __forceinline, otherwise use __inline. But even though they seem like orders, __forceinline and __inline are suggestions for the compiler and can be ignored in the same way.

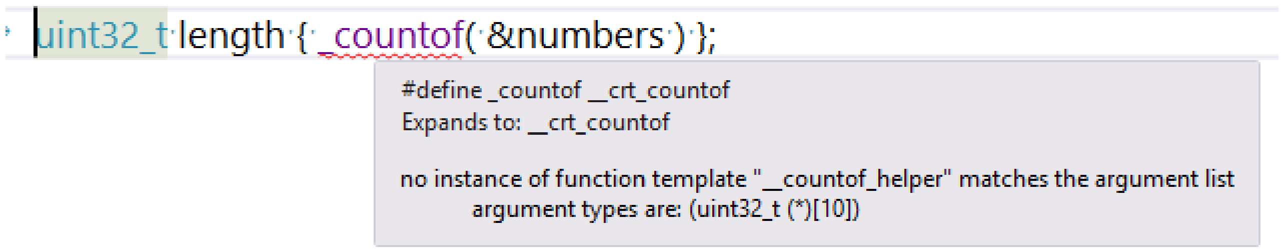
```
#ifndef FORCEINLINE
  #if (_MSC_VER >= 1200)
    #define FORCEINLINE __forceinline
  #else
```

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

It is important to remember that many of these functions, such as **RtlSecureZeroMemory() function**, are provided **inline** only when compiled in RELEASE mode. But **RtlSecureZeroMemory() function** is provided **inline** in DEBUG mode and in RELEASE mode. During the compilation process, the compilers perform a process called "optimization", and this "optimization" means the identification of an inefficient code pattern and substitution by a more efficient and safe code pattern. For the **ZeroMemory() function**, this optimization means that the compiler removes the calls to **ZeroMemory() function** and includes other code that performs the same function. If the use of **ZeroMemory() function** is critical to the application, is recommended to use the **SecureZeroMemory() function** instead of the **ZeroMemory() function**. The **SecureZeroMemory() function** is safer and currently it is not removed by the compilers. The use of **SecureZeroMemory() function** is recommended for use with dynamically allocated memory blocks and not for use with typical variables, however, the macro can also be used with statically allocated arrays, for example. The total number of bytes that all the elements in a static array occupy, can be obtained using the sizeof operator.

```
uint32_t* pointerToNumbers{};
uint32_t numbers[]{ {}, 1ui32, 2ui32, 3ui32, 4ui32, 5ui32, 6ui32, 7ui32, 8ui32, 9ui32 };
uint32_t totalOfBytes{ sizeof( numbers ) };
```

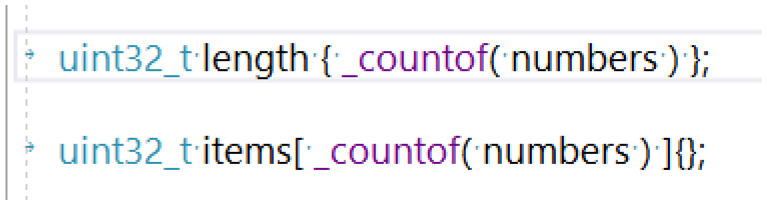
These two expressions calculate the total number of elements that exist on a statically allocated array. The second is a macro of the UCRT/CRT that does the same task, but in recent C++ implementations of Microsoft UCRT/CRT the macro is using a template-based implementation instead of more direct expression, like the first example. One advantage in the use of a template-based implementation is that helps the compiler in a more precising **data type** checking, for example, if we try to pass a pointer as argument value instead of an array, the compiler can check this even before the compilation, and with the direct expression, the compiler cannot does this, and we will get a runtime error of address access violation.



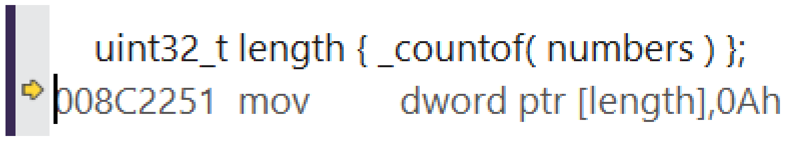
```
uint32_t length { _countof( &numbers ) };
```

```
#define _countof __crt_countof
Expands to: __crt_countof

no instance of function template "__countof_helper" matches the argument list
        argument types are: (uint32_t (*)[10])
```

The result of the **_countof()** macro is a constant value because the macro only can be used with statically allocated arrays. And when we look in the assembly code, we can read the constant value, in this example 0Ah, for the number of the elements:

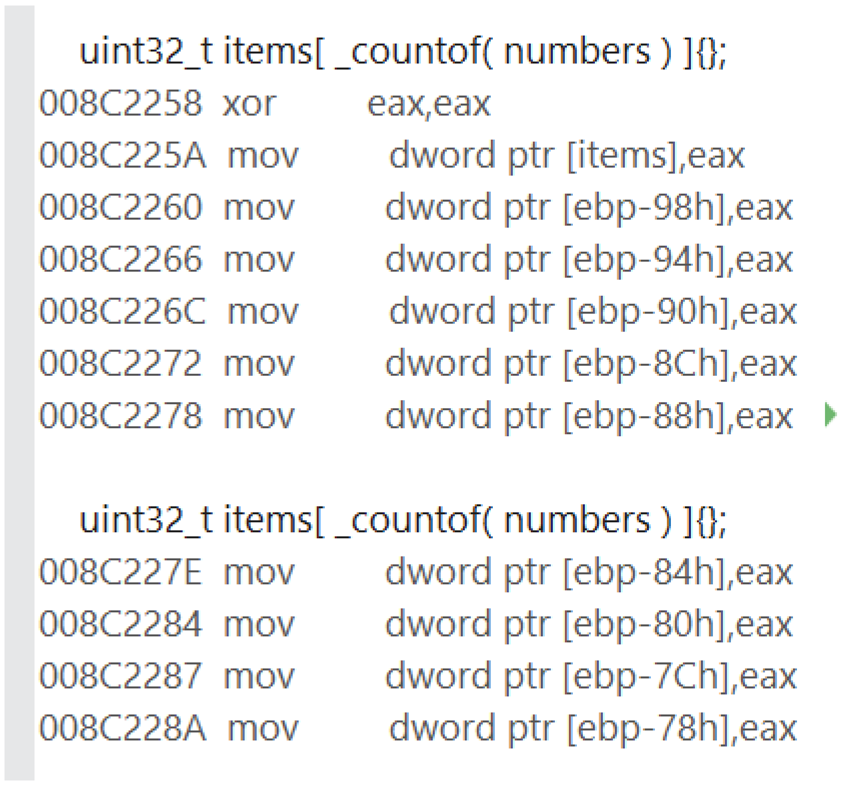


```
uint32_t length { _countof( numbers ) };

uint32_t items[ _countof( numbers ) ]{};
```



```
    uint32_t length { _countof( numbers ) };
008C2251  mov         dword ptr [length],0Ah
```

And the new array named *items* initialized with the number of elements informed by the constant value resultant of the **_countof()** macro call:



```
   uint32_t items[ _countof( numbers ) ]{};
008C2258  xor         eax,eax
008C225A  mov         dword ptr [items],eax
008C2260  mov         dword ptr [ebp-98h],eax
008C2266  mov         dword ptr [ebp-94h],eax
008C226C  mov         dword ptr [ebp-90h],eax
008C2272  mov         dword ptr [ebp-8Ch],eax
008C2278  mov         dword ptr [ebp-88h],eax

   uint32_t items[ _countof( numbers ) ]{};
008C227E  mov         dword ptr [ebp-84h],eax
008C2284  mov         dword ptr [ebp-80h],eax
008C2287  mov         dword ptr [ebp-7Ch],eax
008C228A  mov         dword ptr [ebp-78h],eax
```

So, when developing using C++ programming language, the recommendation is the use of the **_countof()** macro, instead of a direct expression:

```
uint32_t length{ sizeof ( numbers ) / sizeof ( numbers[ 0ui32 ] ) };
```

**`uint32_t length{ _countof( numbers ) };`**

```
Console::WriteLine( u"Number of bytes: %u\n\n", totalOfBytes );
Console::WriteLine( u"(BEFORE) numbers\n\n" );
```

Displays the contents of the array before using **SecureZeroMemory() function**.

```
uint32_t index{};
```

```
for ( ; index != length; index++ ) Console::WriteLine( u"[ %u ]: %u\n", index, numbers[ index ] );
```

Uses **SecureZeroMemory() function** on the array.

```
pointerToNumbers = ( reinterpret_cast< uint32_t* >( SecureZeroMemory( reinterpret_cast< PVOID >( numbers ),
SIZE_T ) totalOfBytes ) ) );
```

The *ptr* parameter is a pointer to the memory address where the buffer that must be filled starts. "First position" is a relative concept because the "start" may be different from the "zero" position. The *cnt* parameter indicates how many bytes should receive the value 0 (zero).

**What does the SecureZeroMemory() macro returns?**

Returns a pointer to the buffer, that is, the function returns a pointer to the address that was informed as the argument value to the *ptr* parameter. But in the signature of the function, the return is of the PVOID type, that is a **typedef** for a *void* * (void pointer), and PVOID type is defined in the header file **Winnt.h** as follows:

```
typedef void *PVOID;
```

Because a *void* pointer (*void) can point to any type there is no way to know which type is returned. Then we must perform the casting for the expected type as a return.

If everything worked out, displays the contents of the array, displays the memory addresses of the pointer returned by **SecureZeroMemory() function**, which is the memory address of the array numbers.

```
if ( ( pointerToNumbers != nullptr ) && ( pointerToNumbers == numbers ) ) {
OR
if ( ( pointerToNumbers ) && ( pointerToNumbers == numbers ) ) {
        Console::WriteLine( u"\npointerToNumbers: %#0x\n" "numbers: %#0x\n", pointerToNumbers, numbers );
```

Displays the contents of the array after using **SecureZeroMemory() function**.

```
Console::WriteLine( u"\n(AFTER) numbers\n\n" );
```

```
for ( index = {}; index != length; index++ ) Console::WriteLine( u"[ %u ]: %u\n", index, numbers[ index ] );
 };

pointerToNumbers = nullptr;
```

# RECOMMENDATIONS

Even though the **SecureZeroMemory()** macro is referenced in the Microsoft official documentation as a function, it is a macro for a real implementation function, named **RtlSecureZeroMemory() function**, and this is another characteristic of the Microsoft Windows API. Today this rarely used, if it is, in modern API's of the Microsoft Windows operating system, but we will find this kind of construction for scenarios like with **SecureZeroMemory()** macro. This is a not a "bad" thing per se, it is a just a possibility available via the programming language. Unfortunately, radical views that are against of this kind of programming construction, could lead to source code bases to ignore the use of the **SecureZeroMemory()** macro and chooses to directly call of the **RtlSecureZeroMemory() function**. It is important to be aware that there is no checking by compiler part to something like this, and for the compiler tools, at least until now, it is just another call for a normal function of a library, so, nothing wrongs will happens, and this works normally. But, as **SecureZeroMemory()** macro is created as the public face of the **RtlSecureZeroMemory() function**, we should respect this structure and purpose used by the Microsoft Windows API. If you are in doubt about the meaning of the **Rtl**, it is **Run-time Library**. However, in specific scenarios we will see that the non-use of programming constructions like **SecureZeroMemory()** macro is not only a recommendation, but it is a requirement. The code used on the development of kernel-mode device drivers, is one of these scenarios. In fact, the use of any kind of code for work at kernel context, is an extremely performance sensitive kind of code and shall follows more disciplined implementation strategies. We can learn more about the use of **RtlSecureZeroMemory() function** for kernel-mode device drivers, and kernel-mode code in general, reading the Microsoft official documentation for device drivers' development:

**https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-rtlsecurezeromemory**

In this page of the Microsoft official documentation for the use of the **RtlSecureZeroMemory() function** on the development of device drivers, we can read at bottom of the web page in the **Requirements** section, more detailed information about the **RtlSecureZeroMemory() function**. If we look at **Wdm.h** header file that is part of the Microsoft Windows Driver Development Kit, we will not find any declaration for a **SecureZeroMemory()** macro, but only the definition of the **RtlSecureZeroMemory() function** :

| | |
|---|---|
| **Minimum supported client** | Available in Windows Server 2003 and later versions of Windows. (Because the routine is declared inline, the body of the routine can be included in earlier versions of the operating system.) |
| **Target Platform** | Desktop |
| **Header** | wdm.h (include Wdm.h, Ntddk.h, Ntifs.h) |
| **IRQL** | Any level (See Remarks section) |

Here is the implementation as is in the **Wdm.h** header file:

```
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((PUCHAR)((ULONG64)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;
}
```

**Run-time Library routines: [https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/_kernel/#run-time-library-rtl-routines](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/_kernel/#run-time-library-rtl-routines)**

## THE NOTORIOUS "HELLO, WORLD!", BUT USING GDI+

No presentation text about software programming is complete without the notorious "Hello, World!" program, but in our case, we have a different "Hello, World!" and it is based on the GDI+ graphical API. When working with .NET platform, we known about the GDI+ when we are using the Microsoft Windows Forms that is part of the .NET Framework Class Library (FCL). This example is left here to be our starting point for the publications. For now, you can do experiments with the source code and explore the concepts that we are talking about on this publication. You should be conscious that the use of aspects of the GDI+ graphical system API is not to explain graphical programming, but infrastructure functionalities of the Microsoft Windows operating system such as the *message queue system* used for communication of events between applications and the operating system specialized services, and this is valid for other graphics system used by sample projects like WPF, UWP / WinRT or DirectX platform's. Using this "Hello, World!" as base code, we also begin to explore aspects of process, threads and memory management.

**Our "Hello, World!" is in fact the sample project Example00 that is part of the sample solution IsWindows7SP1OrGreater**. We will use this structure of source code for examples from now on, and we will begin to explore aspects of GDI+, or other graphical system, but keeping the focus on the purpose of the publication and not as guide for the graphical programming features of the GDI+ API or any other graphical system supported by Microsoft Windows operating system. We will also introduce the construction of Dynamic-link library (DLL) as part of reorganization of the structure of the sample project. Within this publication the companion source code is organized in three files, **Example00.h**, **Button.cpp** and **main.cpp**. This sample project compiles and runs "correctly" but has intentional bugs. One of the bugs is in the **main.cpp** source code file, more specifically in the processing of the **WM_PAINT window message**. If you have a basic experience with GDI+ you can opt by doing the necessary adjustment. If you are not interested in doing this now, you can comment all the source code for the processing **WM_PAINT window message** and the visible GUI bug will disappear. In the **Button.cpp**, for the **WM_PARENTNOTIFY and WM_CREATE window messages**, the source code is incomplete, and you can opt to include experimental code. There are other types of bugs, but they need knowledge about Microsoft Windows operating system inner workings to understands "the why?". Anyway, all these bugs will be fixed and required concepts explained through the publications and using comments in the source code.

## Example00.h header file

```
#pragma once



#pragma region Header files

#include <RVJ.Desktop.h>

#pragma endregion


#pragma region Use of constexpr where possible instead of const

/*


If the minimum MSVC (Microsoft Visual C++) is 1900, that is, Microsoft Visual C++ that is distributed with
Microsoft Visual Studio 2015, use constexpr where supported by the compiler.


*/



#if _MSC_VER  >= 1900


#define CONSTEXPR


#if defined(CONSTEXPR)

#define CONST_OR_CONSTEXPR constexpr

#else

#define CONST_OR_CONSTEXPR const

#endif
```

```
#endif

#pragma endregion

#pragma region String messages
CONST_OR_CONSTEXPR LPCWSTR MessageCaption { L"IsWindows7SP1OrGreater" };
CONST_OR_CONSTEXPR LPCWSTR MessageText { L"We are running on Microsoft Windows 7 Service Pack 1 or greater." };
#pragma endregion

#pragma region Constants for DPI manipulation
CONST_OR_CONSTEXPR int32_t Standard_96DPI { 96i32 };
#pragma endregion

LRESULT ManageButton( HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam );
BOOL CALLBACK ManagedChildWindows( HWND, LPARAM );
```

## Button.cpp

```
#pragma region Header Files
#include <RVJ.Desktop.h>
#include "Example00.h"
#pragma endregion
```

```
LRESULT ManageButton( HWND hWindow, UINT message, WPARAM wParam, LPARAM lParam ) {

        CONST_OR_CONSTEXPR SIZE_T MaxClassNameSize { 80ui32 };
        CONST_OR_CONSTEXPR WCHAR* ButtonClassName { L"Button" };
        CONST_OR_CONSTEXPR SIZE_T paintstructSize { sizeof( PAINTSTRUCT ) };

        LPCREATESTRUCT localStruct {};
        LRESULT result {};
        LPWSTR windowClassName { reinterpret_cast< LPWSTR >( HeapAlloc( GetProcessHeap(),
HEAP_ZERO_MEMORY, MaxClassNameSize ) ) };
        bool validClassName { ( GetClassName( hWindow, windowClassName, MaxClassNameSize ) >
0i32 ) };

        switch ( message ) {

                case WM_PARENTNOTIFY: {

                        switch ( LOWORD( wParam ) ) {

                                case WM_CREATE: {

                                        HWND childHandle = reinterpret_cast<HWND>( lParam
);

                                        localStruct = reinterpret_cast< LPCREATESTRUCT >(
lParam );
```

```
if ( validClassName ) {
    /*
    If it is a BUTTON window, adapt the size to content.
    */
    if ( !wcscmp( windowClassName, ButtonClassName ) ) {
        LPPAINTSTRUCT paintStruct { reinterpret_cast< LPPAINTSTRUCT >( HeapAlloc( GetProcessHeap(), HEAP_ZERO_MEMORY, paintstructSize ) ) };
        /*HDC deviceContext { BeginPaint( hWindow, paintStruct ) };*/

        /*EndPaint( hWindow, paintStruct );*/
        HeapFree( GetProcessHeap(), {}, reinterpret_cast< LPVOID >( paintStruct ) );
        paintStruct = nullptr;
        result = FALSE;
```

```
                                                        };
                                                    }; }; break;

                                            default: break;
                                    };

                            }; break;

                            default: break;
                    };

                    HeapFree( GetProcessHeap(), {}, reinterpret_cast< LPVOID >( windowClassName ) );
                    windowClassName = nullptr;

                    return result;

            };
```

## main.cpp

```
#ifdef __INTEL_COMPILER
#pragma warning( disable: 1079; )
#endif

#pragma region Header Files
```

```
#include <RVJ.Desktop.h>
#include "Example00.h"
#pragma endregion

#pragma region Namespaces
using namespace std;
#pragma endregion


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

        CONST_OR_CONSTEXPR SIZE_T MaxClassNameSize { 80ui32 };
        CONST_OR_CONSTEXPR WCHAR* ButtonClassName { L"Button" };
        CONST_OR_CONSTEXPR char16_t* const defaultWindowText { u"Hello, Microsoft Windows 10!"
};

        LRESULT processResult {};

        switch ( message ) {

                case WM_CREATE: {

                        LPWSTR windowClassName { reinterpret_cast< LPWSTR >( HeapAlloc(
GetProcessHeap(), HEAP_ZERO_MEMORY, MaxClassNameSize ) ) };
                        bool validClassName { ( GetClassName( hWindow, windowClassName,
MaxClassNameSize ) > 0i32 ) };
```

```
                    HeapFree( GetProcessHeap(), {}, reinterpret_cast< LPVOID >(
windowClassName ) );
                    windowClassName = nullptr;

                };
                                    break;

                case WM_SIZE: {

                    RECT clientRect {};

                    GetClientRect( hWindow, &clientRect );

                    EnumChildWindows( hWindow, &ManagedChildWindows, ( LPARAM )
&clientRect );

                    processResult = FALSE;

                }; break;

                case WM_PARENTNOTIFY: {

                    processResult = ManageButton( hWindow, message, wParam, lParam );
```

```
            };  break;

            case WM_COMMAND: {

                if ( HIWORD( wParam ) == BN_CLICKED ) {

                    if ( IsWindows7SP1OrGreater() ) MessageBox( nullptr,
MessageText, MessageCaption, MB_OK );

                };

                processResult = FALSE;

            }; break;

            case WM_PAINT: {

              CONST_OR_CONSTEXPR  SIZE_T rectSize { sizeof( RECT ) };
              CONST_OR_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 ) ) };

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

    SecureZeroMemory( reinterpret_cast< PVOID >( rect ), rectSize );
    HeapFree( processHeap, {}, reinterpret_cast< LPVOID >( rect ) );
    rect = nullptr;


}; break;


case WM_CLOSE: DestroyWindow( hWindow ); break;

case WM_DESTROY: {

        PostQuitMessage( 0i32 );

}; break;
```

```
        };

        processResult = DefWindowProc( hWindow, message, wParam, lParam );

        return processResult;
};



BOOL CALLBACK ManagedChildWindows( HWND handle, LPARAM lParam ) {

        BOOL result {};
        LPRECT clientRect { reinterpret_cast< LPRECT >( lParam ) };
        UINT dpiForWindow { GetDpiForWindow( handle ) };
        int32_t dpiX { MulDiv( ( clientRect->right >> 1i32 ), dpiForWindow, Standard_96DPI ) };
        int32_t dpiY { MulDiv( ( clientRect->bottom >> 1i32 ), dpiForWindow, Standard_96DPI ) };
        int32_t dpiWidth { MulDiv( 200, dpiForWindow, Standard_96DPI ) };
        int32_t dpiHeight { MulDiv( 100, dpiForWindow, Standard_96DPI ) };

        SetWindowPos( handle, nullptr, dpiX, dpiY, dpiWidth, dpiHeight, SWP_NOZORDER |
SWP_NOACTIVATE );
        //MoveWindow( handle, ( clientRect->right >> 1i32 ), ( clientRect->bottom >> 1i32 ), 200, 100,
TRUE );

        result = ShowWindow( handle, SW_SHOW );

        return result;
};
```

```
int WINAPI WinMain( _In_ HINSTANCE hThisInstance, _In_ HINSTANCE hPrevInstance,
        _In_opt_ LPSTR szCmdLine, _In_ int iCmdShow ) {

        CONST_OR_CONSTEXPR char16_t* const applicationName { u"HelloWindowClass" };
        CONST_OR_CONSTEXPR char16_t* const windowCaption { u"The Hello Program" };
        CONST_OR_CONSTEXPR SIZE_T wndclassSizeInBytes { sizeof( WNDCLASSEX ) };
        CONST_OR_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 ) ) };

        #pragma region Task 00 - Create main window.

        windowClass->cbSize = wndclassSizeInBytes;
        windowClass->style = ( CS_HREDRAW | CS_VREDRAW );
        windowClass->lpfnWndProc = &WndProc;
        /*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 );

			#pragma endregion

			#pragma region Task 01 - Create child window.
```

```
CreateWindowEx( {}, reinterpret_cast< LPCWSTR >( L"Button" ),
            reinterpret_cast< LPCWSTR >( windowCaption ),
            WS_TABSTOP | WS_CHILDWINDOW | BS_PUSHBUTTON,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_DEFAULT,
            CW_DEFAULT,
            hWindow,
            nullptr,
            hThisInstance,
            nullptr );

#pragma endregion

ShowWindow( hWindow, iCmdShow );
/*
AnimateWindow( hWindow, 200i32, AW_BLEND | AW_ACTIVATE | AW_SLIDE );
AnimateWindow( hWindow, 200i32, AW_HOR_POSITIVE );
AnimateWindow( hWindow, 200i32, AW_HOR_NEGATIVE );
AnimateWindow( hWindow, 200i32, AW_SLIDE );
AnimateWindow( hWindow, 200i32, AW_VER_POSITIVE );
AnimateWindow( hWindow, 200i32, AW_VER_NEGATIVE );
*/
UpdateWindow( hWindow );
```

```
            while ( messageResult = GetMessage( reinterpret_cast< LPMSG >( message ), nullptr,
0ui32, 0ui32 ) ) {
                TranslateMessage( reinterpret_cast< MSG* >( message ) );
                DispatchMessage( reinterpret_cast< MSG* >( message ) );
            };

        };

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

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

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

        message = nullptr;
        windowClass = nullptr;
        const_cast< HANDLE >( defaultProcessHeap ) = nullptr;

        return messageResult;

};
```

See you on next publication! 😉