Using the Windows Headers

Article • 08/19/2020

The header files for the Windows API enable you to create 32- and 64-bit applications. They include declarations for both Unicode and ANSI versions of the API. For more information, see Unicode in the Windows API. They use data types that enable you to build both 32- and 64-bit versions of your application from a single source code base. For more information, see Getting Ready for 64-bit Windows. Additional features include Header Annotations and STRICT Type Checking.

Visual C++ and the Windows Header Files

Microsoft Visual C++ includes copies of the Windows header files that were current at the time Visual C++ was released. Therefore, if you install updated header files from an SDK, you may end up with multiple versions of the Windows header files on your computer. If you do not ensure that you are using the latest version of the SDK header files, you will receive the following error code when compiling code that uses features that were introduced after Visual C++ was released: error C2065: undeclared identifier.

Macros for Conditional Declarations

Certain functions that depend on a particular version of Windows are declared using conditional code. This enables you to use the compiler to detect whether your application uses functions that are not supported on its target version(s) of Windows. To compile an application that uses these functions, you must define the appropriate macros. Otherwise, you will receive the C2065 error message.

The Windows header files use macros to indicate which versions of Windows support many programming elements. Therefore, you must define these macros to use new functionality introduced in each major operating system release. (Individual header files may use different macros; therefore, if compilation problems occur, check the header file that contains the definition for conditional definitions.) For more information, see SdkDdkVer.h.

The following table describes the preferred macros used in the Windows header files. If you define NTDDI_VERSION, you must also define _WIN32_WINNT.

Expand table

Minimum system required	Value for NTDDI_VERSION
Windows 10 1903 "19H1"	**NTDDI_WIN10_19H1** (0x0A000007)
Windows 10 1809 "Redstone 5"	**NTDDI_WIN10_RS5** (0x0A000006)
Windows 10 1803 "Redstone 4"	**NTDDI_WIN10_RS4** (0x0A000005)
Windows 10 1709 "Redstone 3"	**NTDDI_WIN10_RS3** (0x0A000004)
Windows 10 1703 "Redstone 2"	**NTDDI_WIN10_RS2** (0x0A000003)
Windows 10 1607 "Redstone 1"	**NTDDI_WIN10_RS1** (0x0A000002)
Windows 10 1511 "Threshold 2"	**NTDDI_WIN10_TH2** (0x0A000001)
Windows 10 1507 "Threshold"	**NTDDI_WIN10** (0x0A000000)
Windows 8.1	**NTDDI_WINBLUE** (0x06030000)
Windows 8	**NTDDI_WIN8** (0x06020000)
Windows 7	**NTDDI_WIN7** (0x06010000)
Windows Server 2008	**NTDDI_WS08** (0x06000100)
Windows Vista with Service Pack 1 (SP1)	**NTDDI_VISTASP1** (0x06000100)
Windows Vista	**NTDDI_VISTA** (0x06000000)
Windows Server 2003 with Service Pack 2 (SP2)	**NTDDI_WS03SP2** (0x05020200)
Windows Server 2003 with Service Pack 1 (SP1)	**NTDDI_WS03SP1** (0x05020100)
Windows Server 2003	**NTDDI_WS03** (0x05020000)
Windows XP with Service Pack 3 (SP3)	**NTDDI_WINXPSP3** (0x05010300)
Windows XP with Service Pack 2 (SP2)	**NTDDI_WINXPSP2** (0x05010200)
Windows XP with Service Pack 1 (SP1)	**NTDDI_WINXPSP1** (0x05010100)
Windows XP	**NTDDI_WINXP** (0x05010000)

The following tables describe other macros used in the Windows header files.

Expand table

Minimum system required	Minimum value for _WIN32_WINNT and WINVER
Windows 10	**_WIN32_WINNT_WIN10** (0x0A00)
Windows 8.1	**_WIN32_WINNT_WINBLUE** (0x0603)
Windows 8	**_WIN32_WINNT_WIN8** (0x0602)
Windows 7	**_WIN32_WINNT_WIN7** (0x0601)
Windows Server 2008	**_WIN32_WINNT_WS08** (0x0600)
Windows Vista	**_WIN32_WINNT_VISTA** (0x0600)
Windows Server 2003 with SP1, Windows XP with SP2	**_WIN32_WINNT_WS03** (0x0502)
Windows Server 2003, Windows XP	**_WIN32_WINNT_WINXP** (0x0501)

Expand table

Minimum version required	Minimum value of _WIN32_IE
Internet Explorer 11.0	**_WIN32_IE_IE110** (0x0A00)
Internet Explorer 10.0	**_WIN32_IE_IE100** (0x0A00)
Internet Explorer 9.0	**_WIN32_IE_IE90** (0x0900)
Internet Explorer 8.0	**_WIN32_IE_IE80** (0x0800)
Internet Explorer 7.0	**_WIN32_IE_IE70** (0x0700)
Internet Explorer 6.0 SP2	**_WIN32_IE_IE60SP2** (0x0603)
Internet Explorer 6.0 SP1	**_WIN32_IE_IE60SP1** (0x0601)
Internet Explorer 6.0	**_WIN32_IE_IE60** (0x0600)
Internet Explorer 5.5	**_WIN32_IE_IE55** (0x0550)
Internet Explorer 5.01	**_WIN32_IE_IE501** (0x0501)
Internet Explorer 5.0, 5.0a, 5.0b	**_WIN32_IE_IE50** (0x0500)

Setting WINVER or _WIN32_WINNT

You can define these symbols by using the #define statement in each source file, or by specifying the /D compiler option supported by Visual C++.

For example, to set WINVER in your source file, use the following statement:

```
#define WINVER 0x0502
```

To set _WIN32_WINNT in your source file, use the following statement:

```
#define _WIN32_WINNT 0x0502
```

To set _WIN32_WINNT using the /D compiler option, use the following command:

cl -c /D_WIN32_WINNT=0x0502 *source*.**cpp**

For information on using the /D compiler option, see /D (preprocessor definitions).

Note that some features introduced in the latest version of Windows may be added to a service pack for a previous version of Windows. Therefore, to target a service pack, you may need to define _WIN32_WINNT with the value for the next major operating system release. For example, the **GetDllDirectory** function was introduced in Windows Server 2003 and is conditionally defined if _WIN32_WINNT is 0x0502 or greater. This function was also added to Windows XP with SP1. Therefore, if you were to define _WIN32_WINNT as 0x0501 to target Windows XP, you would miss features that are defined in Windows XP with SP1.

Controlling Structure Packing

Projects should be compiled to use the default structure packing, which is currently 8 bytes because the largest integral type is 8 bytes. Doing so ensures that all structure types within the header files are compiled into the application with the same alignment the Windows API expects. It also ensures that structures with 8-byte values are properly aligned and will not cause alignment faults on processors that enforce data alignment.

For more information, see /Zp (struct member alignment) or pack.

Faster Builds with Smaller Header Files

You can reduce the size of the Windows header files by excluding some of the less common API declarations as follows:

- Define WIN32_LEAN_AND_MEAN to exclude APIs such as Cryptography, DDE, RPC, Shell, and Windows Sockets.

  ```
  #define WIN32_LEAN_AND_MEAN
  ```

- Define one or more of the NO*api* symbols to exclude the API. For example, NOCOMM excludes the serial communication API. For a list of support NO*api* symbols, see Windows.h.

  ```
  #define NOCOMM
  ```

Related topics

Windows SDK download site

Header Annotations

Article • 06/22/2022

[This topic describes the annotations supported in the Windows headers through Windows 7. If you are developing for Windows 8, you should use the annotations described in SAL Annotations.]

Header annotations describe how a function uses its parameters and return value. These annotations have been added to many of the Windows header files to help you ensure that you are calling the Windows API correctly. If you enable code analysis, which is available starting with the Visual Studio 2005, the compiler will produce level 6000 warnings if you are not calling these functions per the usage described through the annotations. You can also add these annotations in your own code to ensure that it is being called correctly. To enable code analysis in Visual Studio, see the documentation for your version of Visual Studio.

These annotations are defined in Specstrings.h. They are built on primitives that are part of the Standard Annotation Language (SAL) and implemented using `_declspec("SAL_*")`.

There are two classes of annotations: buffer annotations and advanced annotations.

Buffer Annotations

Buffer annotations describe how functions use their pointers and can be used to detect buffer overruns. Each parameter may use zero or one buffer annotation. A buffer annotation is constructed with a leading underscore and the components described in the following sections.

Buffer size	Description
(*size*)	Specifies the total size of the buffer. Use with _bcount and _ecount; do not use with _part. This value is the accessible space; it may be less than the allocated space.
(*size*,*length*)	Specifies the total size and initialized length of the buffer. Use with _bcount_part and _ecount_part. The total size may be less than the allocated space.

Buffer size units	Description
_bcount	The buffer size is in bytes.
_ecount	The buffer size is in elements.

Direction	Description
_in	The function reads from the buffer. The caller provides the buffer and initializes it.
_inout	The function both reads from and writes to buffer. The caller provides the buffer and initializes it. If used with _deref, the buffer may be reallocated by the function.
_out	The function writes to the buffer. If used on the return value or with _deref, the function provides the buffer and initializes it. Otherwise, the caller provides the buffer and the function initializes it.

Indirection	Description
_deref	Dereference the parameter to obtain the buffer pointer. This parameter may not be **NULL**.
_deref_opt	Dereference the parameter to obtain the buffer pointer. This parameter can be **NULL**.

Initialization	Description
_full	The function initializes the entire buffer. Use only with output buffers.
_part	The function initializes part of the buffer, and explicitly indicates how much. Use only with output buffers.

Required or optional buffer	Description
_opt	This parameter can be **NULL**.

The following example shows the annotations for the **GetModuleFileName** function. The *hModule* parameter is an optional input parameter . The *lpFilename* parameter is an output parameter; its size in characters is specified by the *nSize* parameter and its length includes the **null**-terminating character. The *nSize* parameter is an input parameter.

C++

```
DWORD
WINAPI
GetModuleFileName(
    __in_opt HMODULE hModule,
    __out_ecount_part(nSize, return + 1) LPTSTR lpFilename,
    __in DWORD nSize
    );
```

The following are the annotations defined in Specstrings.h. Use the information in the tables above to interpret their meaning.

__bcount(*size*)
__bcount_opt(*size*)
__deref_bcount(*size*)
__deref_bcount_opt(*size*)
__deref_ecount(*size*)
__deref_ecount_opt(*size*)
__deref_in
__deref_in_bcount(*size*)
__deref_in_bcount_opt(*size*)
__deref_in_ecount(*size*)
__deref_in_ecount_opt(*size*)
__deref_in_opt
__deref_inout
__deref_inout_bcount(*size*)
__deref_inout_bcount_full(*size*)
__deref_inout_bcount_full_opt(*size*)
__deref_inout_bcount_opt(*size*)
__deref_inout_bcount_part(*size*,*length*)
__deref_inout_bcount_part_opt(*size*,*length*)
__deref_inout_ecount(*size*)
__deref_inout_ecount_full(*size*)
__deref_inout_ecount_full_opt(*size*)
__deref_inout_ecount_opt(*size*)
__deref_inout_ecount_part(*size*,*length*)
__deref_inout_ecount_part_opt(*size*,*length*)
__deref_inout_opt
__deref_opt_bcount(*size*)
__deref_opt_bcount_opt(*size*)
__deref_opt_ecount(*size*)
__deref_opt_ecount_opt(*size*)
__deref_opt_in
__deref_opt_in_bcount(*size*)
__deref_opt_in_bcount_opt(*size*)
__deref_opt_in_ecount(*size*)
__deref_opt_in_ecount_opt(*size*)
__deref_opt_in_opt
__deref_opt_inout
__deref_opt_inout_bcount(*size*)
__deref_opt_inout_bcount_full(*size*)
__deref_opt_inout_bcount_full_opt(*size*)

__deref_opt_inout_bcount_opt(*size*)
__deref_opt_inout_bcount_part(*size*,*length*)
__deref_opt_inout_bcount_part_opt(*size*,*length*)
__deref_opt_inout_ecount(*size*)
__deref_opt_inout_ecount_full(*size*)
__deref_opt_inout_ecount_full_opt(*size*)
__deref_opt_inout_ecount_opt(*size*)
__deref_opt_inout_ecount_part(*size*,*length*)
__deref_opt_inout_ecount_part_opt(*size*,*length*)
__deref_opt_inout_opt
__deref_opt_out
__deref_opt_out_bcount(*size*)
__deref_opt_out_bcount_full(*size*)
__deref_opt_out_bcount_full_opt(*size*)
__deref_opt_out_bcount_opt(*size*)
__deref_opt_out_bcount_part(*size*,*length*)
__deref_opt_out_bcount_part_opt(*size*,*length*)
__deref_opt_out_ecount(*size*)
__deref_opt_out_ecount_full(*size*)
__deref_opt_out_ecount_full_opt(*size*)
__deref_opt_out_ecount_opt(*size*)
__deref_opt_out_ecount_part(*size*,*length*)
__deref_opt_out_ecount_part_opt(*size*,*length*)
__deref_opt_out_opt
__deref_out
__deref_out_bcount(*size*)
__deref_out_bcount_full(*size*)
__deref_out_bcount_full_opt(*size*)
__deref_out_bcount_opt(*size*)
__deref_out_bcount_part(*size*,*length*)
__deref_out_bcount_part_opt(*size*,*length*)
__deref_out_ecount(*size*)
__deref_out_ecount_full(*size*)
__deref_out_ecount_full_opt(*size*)
__deref_out_ecount_opt(*size*)
__deref_out_ecount_part(*size*,*length*)
__deref_out_ecount_part_opt(*size*,*length*)
__deref_out_opt
__ecount(*size*)
__ecount_opt(*size*)

__in
__in_bcount(*size*)
__in_bcount_opt(*size*)
__in_ecount(*size*)
__in_ecount_opt(*size*)
__in_opt
__inout
__inout_bcount(*size*)
__inout_bcount_full(*size*)
__inout_bcount_full_opt(*size*)
__inout_bcount_opt(*size*)
__inout_bcount_part(*size*,*length*)
__inout_bcount_part_opt(*size*,*length*)
__inout_ecount(*size*)
__inout_ecount_full(*size*)
__inout_ecount_full_opt(*size*)
__inout_ecount_opt(*size*)
__inout_ecount_part(*size*,*length*)
__inout_ecount_part_opt(*size*,*length*)
__inout_opt
__out
__out_bcount(*size*)
__out_bcount_full(*size*)
__out_bcount_full_opt(*size*)
__out_bcount_opt(*size*)
__out_bcount_part(*size*,*length*)
__out_bcount_part_opt(*size*,*length*)
__out_ecount(*size*)
__out_ecount_full(*size*)
__out_ecount_full_opt(*size*)
__out_ecount_opt(*size*)
__out_ecount_part(*size*,*length*)
__out_ecount_part_opt(*size*,*length*)
__out_opt

Advanced Annotations

Advanced annotations provide additional information about the parameter or return value. Each parameter or return value may use zero or one advanced annotation.

Annotation	Description
__blocksOn(*resource*)	The functions blocks on the specified resource.
__callback	The function can be used as a function pointer.
__checkReturn	Callers must check the return value.
__format_string	The parameter is a string that contains printf-style % markers.
__in_awcount(*expr*,*size*)	If the expression is true at exit, the size of the input buffer is specified in bytes. If the expression is false, the size is specified in elements.
__nullnullterminated	The buffer may be accessed up to and including the first sequence of two **null** characters or pointers.
__nullterminated	The buffer may be accessed up to and including the first **null** character or pointer.
__out_awcount(*expr*,*size*)	If the expression is true at exit, the size of the output buffer is specified in bytes. If the expression is false, the size is specified in elements.
__override	Specifies C#-style override behavior for virtual methods.
__reserved	The parameter is reserved for future use and must be zero or **NULL**.
__success(*expr*)	If the expression is true at exit, the caller can rely on all guarantees specified by other annotations. If the expression is false, the caller cannot rely on the guarantees. This annotation is automatically added to functions that return an **HRESULT** value.
__typefix(*ctype*)	Treat the parameter as the specified type rather than its declared type.

The following examples show the buffer and advanced annotations for the DeleteTimerQueueTimer, FreeEnvironmentStrings, and UnhandledExceptionFilter functions.

C++

```
__checkReturn
BOOL
WINAPI
DeleteTimerQueueTimer(
    __in_opt HANDLE TimerQueue,
    __in     HANDLE Timer,
    __in_opt HANDLE CompletionEvent
    );

BOOL
WINAPI
FreeEnvironmentStrings(
    __in __nullnullterminated LPTCH
```

```
    );

__callback
LONG
WINAPI
UnhandledExceptionFilter(
    __in struct _EXCEPTION_POINTERS *ExceptionInfo
    );
```

Related topics

SAL Annotations

Walkthrough: Analyzing C/C++ Code for Defects

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

STRICT Type Checking

Article • 08/23/2019

The Windows.h header file contains definitions, macros, and structures to help you write source code that is portable between versions of Windows. Some of these features are enabled when you define the STRICT symbol when building your application. The following topics explain the advantages of STRICT features and how using them affects the writing of code.

- Enabling STRICT
- Disabling STRICT
- STRICT Compliance

Feedback

Enabling STRICT

Article • 02/08/2021

When you define the **STRICT** symbol, you enable features that require more care in declaring and using types. This helps you write more portable code. This extra care will also reduce your debugging time. Enabling **STRICT** redefines certain data types so that the compiler does not permit assignment from one type to another without an explicit cast. This is especially helpful with Windows code. Errors in passing data types are reported at compile time instead of causing fatal errors at run time.

With Visual C++, **STRICT** type checking is defined by default.

To define **STRICT** on a file-by-file basis, insert a **#define** statement before including Windows.h:

```
C++

#define STRICT
#include <windows.h>
```

When **STRICT** is defined, data type definitions change as follows:

- Specific handle types are defined to be mutually exclusive; for example, you will not be able to pass an **HWND** where an **HDC** type argument is required. Without **STRICT**, all handles are defined as **HANDLE**, so the compiler does not prevent you from using one type of handle where another type is expected.
- All callback function types (such as dialog procedures, window procedures, and hook procedures) are defined with full prototypes. This prevents you from declaring callback functions with incorrect parameter lists.
- Parameter and return value types that should use a generic pointer are declared correctly as **LPVOID** instead of as **LPSTR** or another pointer type.
- The **COMSTAT** structure is declared according to the ANSI standard.

Related topics

Disabling STRICT

STRICT Compliance

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Disabling STRICT

Article • 08/23/2019

To disable **STRICT** type checking, define the symbol name **NO_STRICT**. In Visual C/C++, you can also specify this definition on the command line or in a makefile by specifying **/DNO_STRICT** as a compiler option.

To define **NO_STRICT** on a file-by-file basis, insert a **#define** statement before including Windows.h:

C++

```
#define NO_STRICT
#include <windows.h>
```

For best results, you should also set the warning level for error messages to at least **/W3**. This is always advisable with applications for Windows, because a coding practice that causes a warning (for example, passing the wrong number of parameters) usually causes a fatal error at run time if it is not corrected.

Related topics

Enabling STRICT

STRICT Compliance

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

STRICT Compliance

Article • 08/19/2020

Some source code that compiles successfully might produce error messages when you enable **STRICT** type checking. The following sections describe the minimal requirements for making your code compile when **STRICT** is enabled. Additional steps are recommended, especially to produce portable code.

General Requirements

The principal requirement is that you must declare correct handle types and function pointers instead of relying on more general types. You cannot use one handle type where another is expected. This also means that you may have to change function declarations and use more type casts.

For best results, the generic **HANDLE** type should be used only when necessary.

Declaring Functions Within Your Application

Make sure all application functions are declared. Placing all function declarations in an include file is recommended because you can easily scan your declarations and look for parameter and return types that should be changed.

If you use the **/Zg** compiler option to create header files for your functions, remember that you will get different results depending on whether you have enabled **STRICT** type checking. With **STRICT** disabled, all handle types generate the same base type. With **STRICT** enabled, they generate different base types. To avoid conflict, you need to re-create the header file each time you enable or disable **STRICT**, or edit the header file to use the types **HWND**, **HDC**, **HANDLE**, and so on, instead of the base types.

Any function declarations that you copied from Windows.h into your source code may have changed, and your local declaration may be out of date. Remove your local declaration.

Types that Require Casts

Some functions have generic return types or parameters. For example, the SendMessage function returns data that may be any number of types, depending on the context. When you see any of these functions in your source code, make sure that

you use the correct type cast and that it is as specific as possible. The following list is an example of these functions.

- LocalLock
- GlobalLock
- GetWindowLong
- SetWindowLong
- SendMessage
- DefWindowProc
- SendDlgItemMessage

When you call SendMessage, DefWindowProc, or SendDlgItemMessage, you should first cast the result to type **UINT_PTR**. You need to take similar steps for any function that returns an **LRESULT** or **LONG_PTR** value, where the result contains a handle. This is necessary for writing portable code because the size of a handle varies, depending on the version of Windows. The (**UINT_PTR**) cast ensures proper conversion. The following code shows an example in which **SendMessage** returns a handle to a brush:

```
C++

HBRUSH hbr;

hbr = (HBRUSH)(UINT_PTR)SendMessage(hwnd, WM_CTLCOLOR, ..., ...);
```

The **CreateWindow** and **CreateWindowEx** parameter *hmenu* is sometimes used to pass an integer control identifier (ID). In this case, you must cast the ID to an **HMENU** type:

```
C++

HWND hwnd;
int id;

hwnd = CreateWindow(
        TEXT("Button"), TEXT("OK"), BS_PUSHBUTTON,
        x, y, cx, cy, hwndParent,
        (HMENU)id,    // Cast required here
        hinst,
        NULL);
```

Additional Considerations

To get the most benefit from **STRICT** type checking, there are additional guidelines you should follow. Your code will be more portable in future versions of Windows if you make the following changes.

The types **WPARAM**, **LPARAM**, **LRESULT**, and **LPVOID** are *polymorphic data types*. They hold different kinds of data at different times, even when **STRICT** type checking is enabled. To get the benefit of type checking, you should cast values of these types as soon as possible. (Note that message crackers automatically recast *wParam* and *lParam* for you in a portable way.)

Take special care to distinguish **HMODULE** and **HINSTANCE** types. Even with **STRICT** enabled, they are defined as the same base type. Most kernel module management functions use **HINSTANCE** types, but there are a few functions that return or accept only **HMODULE** types.

Related topics

Disabling STRICT

Enabling STRICT

Feedback

Windows Data Types

Article • 11/07/2024

The data types supported by Windows are used to define function return values, function and message parameters, and structure members. They define the size and meaning of these elements. For more information about the underlying C/C++ data types, see Data Type Ranges.

The following table contains the following types: character, integer, Boolean, pointer, and handle. The character, integer, and Boolean types are common to most C compilers. Most of the pointer-type names begin with a prefix of P or LP. Handles refer to a resource that has been loaded into memory.

For more information about handling 64-bit integers, see Large Integers.

Expand table

Data type	Description
`APIENTRY`	The calling convention for system functions. This type is declared in WinDef.h as follows: `#define APIENTRY WINAPI`
`ATOM`	An atom. For more information, see About Atom Tables. This type is declared in WinDef.h as follows: `typedef WORD ATOM;`
`BOOL`	A Boolean variable (should be **TRUE** or **FALSE**). This type is declared in WinDef.h as follows: `typedef int BOOL;`
`BOOLEAN`	A Boolean variable (should be **TRUE** or **FALSE**). This type is declared in WinNT.h as follows: `typedef BYTE BOOLEAN;`
`BYTE`	A byte (8 bits). This type is declared in WinDef.h as follows: `typedef unsigned char BYTE;`
`CALLBACK`	The calling convention for callback functions. This type is declared in WinDef.h as follows: `#define CALLBACK __stdcall` **CALLBACK**, **WINAPI**, and **APIENTRY** are all used to define functions with the __stdcall calling convention. Most functions in the Windows API are declared using **WINAPI**. You may wish to use **CALLBACK** for the callback

Data type	Description
	functions that you implement to help identify the function as a callback function.
`CCHAR`	An 8-bit Windows (ANSI) character. This type is declared in WinNT.h as follows: `typedef char CCHAR;`
`CHAR`	An 8-bit Windows (ANSI) character. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef char CHAR;`
`COLORREF`	The red, green, blue (RGB) color value (32 bits). See **COLORREF** for information on this type. This type is declared in WinDef.h as follows: `typedef DWORD COLORREF;`
`CONST`	A variable whose value is to remain constant during execution. This type is declared in WinDef.h as follows: `#define CONST const`
`DWORD`	A 32-bit unsigned integer. The range is 0 through 4294967295 decimal. This type is declared in IntSafe.h as follows: `typedef unsigned long DWORD;`
`DWORDLONG`	A 64-bit unsigned integer. The range is 0 through 18446744073709551615 decimal. This type is declared in IntSafe.h as follows: `typedef unsigned __int64 DWORDLONG;`
`DWORD_PTR`	An unsigned long type for pointer precision. Use when casting a pointer to a long type to perform pointer arithmetic. (Also commonly used for general 32-bit parameters that have been extended to 64 bits in 64-bit Windows.) This type is declared in BaseTsd.h as follows: `typedef ULONG_PTR DWORD_PTR;`
`DWORD32`	A 32-bit unsigned integer. This type is declared in BaseTsd.h as follows: `typedef unsigned int DWORD32;`
`DWORD64`	A 64-bit unsigned integer. This type is declared in BaseTsd.h as follows: `typedef unsigned __int64 DWORD64;`
`FLOAT`	A floating-point variable. This type is declared in WinDef.h as follows: `typedef float FLOAT;`

Data type	Description
`HACCEL`	A handle to an accelerator table. This type is declared in WinDef.h as follows: `typedef HANDLE HACCEL;`
`HALF_PTR`	Half the size of a pointer. Use within a structure that contains a pointer and two small fields. This type is declared in BaseTsd.h as follows: Expand table **C++** `#ifdef _WIN64` ` typedef int HALF_PTR;` `#else` ` typedef short HALF_PTR;` `#endif`
`HANDLE`	A handle to an object. This type is declared in WinNT.h as follows: `typedef PVOID HANDLE;`
`HBITMAP`	A handle to a bitmap. This type is declared in WinDef.h as follows: `typedef HANDLE HBITMAP;`
`HBRUSH`	A handle to a brush. This type is declared in WinDef.h as follows: `typedef HANDLE HBRUSH;`
`HCOLORSPACE`	A handle to a color space. This type is declared in WinDef.h as follows: `typedef HANDLE HCOLORSPACE;`
`HCONV`	A handle to a dynamic data exchange (DDE) conversation. This type is declared in Ddeml.h as follows: `typedef HANDLE HCONV;`

Data type	Description
`HCONVLIST`	A handle to a DDE conversation list. This type is declared in Ddeml.h as follows: `typedef HANDLE HCONVLIST;`
`HCURSOR`	A handle to a cursor. This type is declared in WinDef.h as follows: `typedef HICON HCURSOR;`
`HDC`	A handle to a device context (DC). This type is declared in WinDef.h as follows: `typedef HANDLE HDC;`
`HDDEDATA`	A handle to DDE data. This type is declared in Ddeml.h as follows: `typedef HANDLE HDDEDATA;`
`HDESK`	A handle to a desktop. This type is declared in WinDef.h as follows: `typedef HANDLE HDESK;`
`HDROP`	A handle to an internal drop structure. This type is declared in ShellApi.h as follows: `typedef HANDLE HDROP;`
`HDWP`	A handle to a deferred window position structure. This type is declared in WinUser.h as follows: `typedef HANDLE HDWP;`
`HENHMETAFILE`	A handle to an enhanced metafile. This type is declared in WinDef.h as follows: `typedef HANDLE HENHMETAFILE;`
`HFILE`	A handle to a file opened by **OpenFile**, not **CreateFile**. This type is declared in WinDef.h as follows:

Data type	Description
	`typedef int HFILE;`
`HFONT`	A handle to a font. This type is declared in WinDef.h as follows: `typedef HANDLE HFONT;`
`HGDIOBJ`	A handle to a GDI object. This type is declared in WinDef.h as follows: `typedef HANDLE HGDIOBJ;`
`HGLOBAL`	A handle to a global memory block. This type is declared in WinDef.h as follows: `typedef HANDLE HGLOBAL;`
`HHOOK`	A handle to a hook. This type is declared in WinDef.h as follows: `typedef HANDLE HHOOK;`
`HICON`	A handle to an icon. This type is declared in WinDef.h as follows: `typedef HANDLE HICON;`
`HINSTANCE`	A handle to an instance. This is the base address of the module in memory. **HMODULE** and **HINSTANCE** are the same today, but represented different things in 16-bit Windows. This type is declared in WinDef.h as follows: `typedef HANDLE HINSTANCE;`
`HKEY`	A handle to a registry key. This type is declared in WinDef.h as follows: `typedef HANDLE HKEY;`
`HKL`	An input locale identifier. This type is declared in WinDef.h as follows:

Data type	Description
	`typedef HANDLE HKL;`
`HLOCAL`	A handle to a local memory block. This type is declared in WinDef.h as follows: `typedef HANDLE HLOCAL;`
`HMENU`	A handle to a menu. This type is declared in WinDef.h as follows: `typedef HANDLE HMENU;`
`HMETAFILE`	A handle to a metafile. This type is declared in WinDef.h as follows: `typedef HANDLE HMETAFILE;`
`HMODULE`	A handle to a module. This is the base address of the module in memory. **HMODULE** and **HINSTANCE** are the same in current versions of Windows, but represented different things in 16-bit Windows. This type is declared in WinDef.h as follows: `typedef HINSTANCE HMODULE;`
`HMONITOR`	A handle to a display monitor. This type is declared in WinDef.h as follows: `if(WINVER >= 0x0500) typedef HANDLE HMONITOR;`
`HPALETTE`	A handle to a palette. This type is declared in WinDef.h as follows: `typedef HANDLE HPALETTE;`
`HPEN`	A handle to a pen. This type is declared in WinDef.h as follows: `typedef HANDLE HPEN;`
`HRESULT`	The return codes used by COM interfaces. For more information, see Structure of the COM Error Codes. To test an **HRESULT** value, use the **FAILED** and **SUCCEEDED** macros. This type is declared in WinNT.h as follows:

Data type	Description
	`typedef LONG HRESULT;`
`HRGN`	A handle to a region. This type is declared in WinDef.h as follows: `typedef HANDLE HRGN;`
`HRSRC`	A handle to a resource. This type is declared in WinDef.h as follows: `typedef HANDLE HRSRC;`
`HSZ`	A handle to a DDE string. This type is declared in Ddeml.h as follows: `typedef HANDLE HSZ;`
`HWINSTA`	A handle to a window station. This type is declared in WinDef.h as follows: `typedef HANDLE WINSTA;`
`HWND`	A handle to a window. This type is declared in WinDef.h as follows: `typedef HANDLE HWND;`
`INT`	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in WinDef.h as follows: `typedef int INT;`
`INT_PTR`	A signed integer type for pointer precision. Use when casting a pointer to an integer to perform pointer arithmetic. This type is declared in BaseTsd.h as follows: Expand table **C++** `#if defined(_WIN64)` `typedef __int64 INT_PTR;`

Data type	Description
	C++ ``` #else typedef int INT_PTR; #endif ```
`INT8`	An 8-bit signed integer. This type is declared in BaseTsd.h as follows: `typedef signed char INT8;`
`INT16`	A 16-bit signed integer. This type is declared in BaseTsd.h as follows: `typedef signed short INT16;`
`INT32`	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in BaseTsd.h as follows: `typedef signed int INT32;`
`INT64`	A 64-bit signed integer. The range is -9223372036854775808 through 9223372036854775807 decimal. This type is declared in BaseTsd.h as follows: `typedef signed __int64 INT64;`
`LANGID`	A language identifier. For more information, see Language Identifiers. This type is declared in WinNT.h as follows: `typedef WORD LANGID;`
`LCID`	A locale identifier. For more information, see Locale Identifiers. This type is declared in WinNT.h as follows: `typedef DWORD LCID;`
`LCTYPE`	A locale information type. For a list, see Locale Information Constants. This type is declared in WinNls.h as follows: `typedef DWORD LCTYPE;`
`LGRPID`	A language group identifier. For a list, see **EnumLanguageGroupLocales**.

Data type	Description
	This type is declared in WinNls.h as follows: `typedef DWORD LGRPID;`
`LONG`	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal. This type is declared in WinNT.h as follows: `typedef long LONG;`
`LONGLONG`	A 64-bit signed integer. The range is -9223372036854775808 through 9223372036854775807 decimal. This type is declared in WinNT.h as follows: Expand table **C++** `#if !defined(_M_IX86)` `  typedef __int64 LONGLONG;` `#else` `  typedef double LONGLONG;` `#endif`
`LONG_PTR`	A signed long type for pointer precision. Use when casting a pointer to a long to perform pointer arithmetic. This type is declared in BaseTsd.h as follows: Expand table **C++** `#if defined(_WIN64)` `  typedef __int64 LONG_PTR;` `#else` `  typedef long LONG_PTR;` `#endif`
`LONG32`	A 32-bit signed integer. The range is -2147483648 through 2147483647 decimal.

Data type	Description
	This type is declared in BaseTsd.h as follows: `typedef signed int LONG32;`
`LONG64`	A 64-bit signed integer. The range is -9223372036854775808 through 9223372036854775807 decimal. This type is declared in BaseTsd.h as follows: `typedef __int64 LONG64;`
`LPARAM`	A message parameter. This type is declared in WinDef.h as follows: `typedef LONG_PTR LPARAM;`
`LPBOOL`	A pointer to a **BOOL**. This type is declared in WinDef.h as follows: `typedef BOOL far *LPBOOL;`
`LPBYTE`	A pointer to a **BYTE**. This type is declared in WinDef.h as follows: `typedef BYTE far *LPBYTE;`
`LPCOLORREF`	A pointer to a **COLORREF** value. This type is declared in WinDef.h as follows: `typedef DWORD *LPCOLORREF;`
`LPCSTR`	A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef __nullterminated CONST CHAR *LPCSTR;`
`LPCTSTR`	An **LPCWSTR** if **UNICODE** is defined, an **LPCSTR** otherwise. For more information, see Windows Data Types for Strings. This type is declared in WinNT.h as follows: Expand table

Data type	Description
	C++ ``` #ifdef UNICODE typedef LPCWSTR LPCTSTR; #else typedef LPCSTR LPCTSTR; #endif ```
`LPCVOID`	A pointer to a constant of any type. This type is declared in WinDef.h as follows: `typedef CONST void *LPCVOID;`
`LPCWSTR`	A pointer to a constant null-terminated string of 16-bit Unicode characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef CONST WCHAR *LPCWSTR;`
`LPDWORD`	A pointer to a **DWORD**. This type is declared in WinDef.h as follows: `typedef DWORD *LPDWORD;`
`LPHANDLE`	A pointer to a **HANDLE**. This type is declared in WinDef.h as follows: `typedef HANDLE *LPHANDLE;`
`LPINT`	A pointer to an **INT**. This type is declared in WinDef.h as follows: `typedef int *LPINT;`
`LPLONG`	A pointer to a **LONG**. This type is declared in WinDef.h as follows: `typedef long *LPLONG;`
`LPSTR`	A pointer to a null-terminated string of 8-bit Windows (ANSI) characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows:

Data type	Description
	`typedef CHAR *LPSTR;`
`LPTSTR`	An **LPWSTR** if **UNICODE** is defined, an **LPSTR** otherwise. For more information, see Windows Data Types for Strings. This type is declared in WinNT.h as follows: Expand table **C++** ``` #ifdef UNICODE typedef LPWSTR LPTSTR; #else typedef LPSTR LPTSTR; #endif ```
`LPVOID`	A pointer to any type. This type is declared in WinDef.h as follows: `typedef void *LPVOID;`
`LPWORD`	A pointer to a **WORD**. This type is declared in WinDef.h as follows: `typedef WORD *LPWORD;`
`LPWSTR`	A pointer to a null-terminated string of 16-bit Unicode characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef WCHAR *LPWSTR;`
`LRESULT`	Signed result of message processing. This type is declared in WinDef.h as follows: `typedef LONG_PTR LRESULT;`
`PBOOL`	A pointer to a **BOOL**. This type is declared in WinDef.h as follows: `typedef BOOL *PBOOL;`

Data type	Description
`PBOOLEAN`	A pointer to a **BOOLEAN**. This type is declared in WinNT.h as follows: `typedef BOOLEAN *PBOOLEAN;`
`PBYTE`	A pointer to a **BYTE**. This type is declared in WinDef.h as follows: `typedef BYTE *PBYTE;`
`PCHAR`	A pointer to a **CHAR**. This type is declared in WinNT.h as follows: `typedef CHAR *PCHAR;`
`PCSTR`	A pointer to a constant null-terminated string of 8-bit Windows (ANSI) characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef CONST CHAR *PCSTR;`
`PCTSTR`	A **PCWSTR** if **UNICODE** is defined, a **PCSTR** otherwise. For more information, see Windows Data Types for Strings. This type is declared in WinNT.h as follows: Expand table **C++** `#ifdef UNICODE` ` typedef LPCWSTR PCTSTR;` `#else` ` typedef LPCSTR PCTSTR;` `#endif`
`PCWSTR`	A pointer to a constant null-terminated string of 16-bit Unicode characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef CONST WCHAR *PCWSTR;`
`PDWORD`	A pointer to a **DWORD**.

Data type	Description
	This type is declared in WinDef.h as follows: `typedef DWORD *PDWORD;`
`PDWORDLONG`	A pointer to a DWORDLONG. This type is declared in WinNT.h as follows: `typedef DWORDLONG *PDWORDLONG;`
`PDWORD_PTR`	A pointer to a DWORD_PTR. This type is declared in BaseTsd.h as follows: `typedef DWORD_PTR *PDWORD_PTR;`
`PDWORD32`	A pointer to a DWORD32. This type is declared in BaseTsd.h as follows: `typedef DWORD32 *PDWORD32;`
`PDWORD64`	A pointer to a DWORD64. This type is declared in BaseTsd.h as follows: `typedef DWORD64 *PDWORD64;`
`PFLOAT`	A pointer to a FLOAT. This type is declared in WinDef.h as follows: `typedef FLOAT *PFLOAT;`
`PHALF_PTR`	A pointer to a HALF_PTR. This type is declared in BaseTsd.h as follows: Expand table **C++** `#ifdef _WIN64` ` typedef HALF_PTR *PHALF_PTR;` `#else` ` typedef HALF_PTR *PHALF_PTR;` `#endif`

Data type	Description
`PHANDLE`	A pointer to a **HANDLE**. This type is declared in WinNT.h as follows: `typedef HANDLE *PHANDLE;`
`PHKEY`	A pointer to an **HKEY**. This type is declared in WinDef.h as follows: `typedef HKEY *PHKEY;`
`PINT`	A pointer to an **INT**. This type is declared in WinDef.h as follows: `typedef int *PINT;`
`PINT_PTR`	A pointer to an **INT_PTR**. This type is declared in BaseTsd.h as follows: `typedef INT_PTR *PINT_PTR;`
`PINT8`	A pointer to an **INT8**. This type is declared in BaseTsd.h as follows: `typedef INT8 *PINT8;`
`PINT16`	A pointer to an **INT16**. This type is declared in BaseTsd.h as follows: `typedef INT16 *PINT16;`
`PINT32`	A pointer to an **INT32**. This type is declared in BaseTsd.h as follows: `typedef INT32 *PINT32;`
`PINT64`	A pointer to an **INT64**. This type is declared in BaseTsd.h as follows: `typedef INT64 *PINT64;`
`PLCID`	A pointer to an **LCID**. This type is declared in WinNT.h as follows:

Data type	Description
	`typedef PDWORD PLCID;`
`PLONG`	A pointer to a **LONG**. This type is declared in WinNT.h as follows: `typedef LONG *PLONG;`
`PLONGLONG`	A pointer to a **LONGLONG**. This type is declared in WinNT.h as follows: `typedef LONGLONG *PLONGLONG;`
`PLONG_PTR`	A pointer to a **LONG_PTR**. This type is declared in BaseTsd.h as follows: `typedef LONG_PTR *PLONG_PTR;`
`PLONG32`	A pointer to a **LONG32**. This type is declared in BaseTsd.h as follows: `typedef LONG32 *PLONG32;`
`PLONG64`	A pointer to a **LONG64**. This type is declared in BaseTsd.h as follows: `typedef LONG64 *PLONG64;`
`POINTER_32`	A 32-bit pointer. On a 32-bit system, this is a native pointer. On a 64-bit system, this is a truncated 64-bit pointer. This type is declared in BaseTsd.h as follows: Expand table **C++** `#if defined(_WIN64)` `#define POINTER_32 __ptr32` `#else` `#define POINTER_32` `#endif`
`POINTER_64`	A 64-bit pointer. On a 64-bit system, this is a native pointer. On a 32-bit system, this is a sign-extended 32-bit pointer.

Data type	Description
	Note that it is not safe to assume the state of the high pointer bit. This type is declared in BaseTsd.h as follows: Expand table **C++** ``` #if (_MSC_VER >= 1300) #define POINTER_64 __ptr64 #else #define POINTER_64 #endif ```
`POINTER_SIGNED`	A signed pointer. This type is declared in BaseTsd.h as follows: `#define POINTER_SIGNED __sptr`
`POINTER_UNSIGNED`	An unsigned pointer. This type is declared in BaseTsd.h as follows: `#define POINTER_UNSIGNED __uptr`
`PSHORT`	A pointer to a **SHORT**. This type is declared in WinNT.h as follows: `typedef SHORT *PSHORT;`
`PSIZE_T`	A pointer to a **SIZE_T**. This type is declared in BaseTsd.h as follows: `typedef SIZE_T *PSIZE_T;`
`PSSIZE_T`	A pointer to a **SSIZE_T**. This type is declared in BaseTsd.h as follows: `typedef SSIZE_T *PSSIZE_T;`
`PSTR`	A pointer to a null-terminated string of 8-bit Windows (ANSI) characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows:

Data type	Description
	`typedef CHAR *PSTR;`
`PTBYTE`	A pointer to a **TBYTE**. This type is declared in WinNT.h as follows: `typedef TBYTE *PTBYTE;`
`PTCHAR`	A pointer to a **TCHAR**. This type is declared in WinNT.h as follows: `typedef TCHAR *PTCHAR;`
`PTSTR`	A **PWSTR** if **UNICODE** is defined, a **PSTR** otherwise. For more information, see Windows Data Types for Strings. This type is declared in WinNT.h as follows: Expand table **C++** `#ifdef UNICODE` ` typedef LPWSTR PTSTR;` `#else typedef LPSTR PTSTR;` `#endif`
`PUCHAR`	A pointer to a **UCHAR**. This type is declared in WinDef.h as follows: `typedef UCHAR *PUCHAR;`
`PUHALF_PTR`	A pointer to a **UHALF_PTR**. This type is declared in BaseTsd.h as follows: Expand table **C++** `#ifdef _WIN64` ` typedef UHALF_PTR *PUHALF_PTR;` `#else`

Data type	Description
	C++ ``` typedef UHALF_PTR *PUHALF_PTR; #endif ```
`PUINT`	A pointer to a **UINT**. This type is declared in WinDef.h as follows: `typedef UINT *PUINT;`
`PUINT_PTR`	A pointer to a **UINT_PTR**. This type is declared in BaseTsd.h as follows: `typedef UINT_PTR *PUINT_PTR;`
`PUINT8`	A pointer to a **UINT8**. This type is declared in BaseTsd.h as follows: `typedef UINT8 *PUINT8;`
`PUINT16`	A pointer to a **UINT16**. This type is declared in BaseTsd.h as follows: `typedef UINT16 *PUINT16;`
`PUINT32`	A pointer to a **UINT32**. This type is declared in BaseTsd.h as follows: `typedef UINT32 *PUINT32;`
`PUINT64`	A pointer to a **UINT64**. This type is declared in BaseTsd.h as follows: `typedef UINT64 *PUINT64;`
`PULONG`	A pointer to a **ULONG**. This type is declared in WinDef.h as follows: `typedef ULONG *PULONG;`
`PULONGLONG`	A pointer to a **ULONGLONG**. This type is declared in WinDef.h as follows:

Data type	Description
	`typedef ULONGLONG *PULONGLONG;`
`PULONG_PTR`	A pointer to a **ULONG_PTR**. This type is declared in BaseTsd.h as follows: `typedef ULONG_PTR *PULONG_PTR;`
`PULONG32`	A pointer to a **ULONG32**. This type is declared in BaseTsd.h as follows: `typedef ULONG32 *PULONG32;`
`PULONG64`	A pointer to a **ULONG64**. This type is declared in BaseTsd.h as follows: `typedef ULONG64 *PULONG64;`
`PUSHORT`	A pointer to a **USHORT**. This type is declared in WinDef.h as follows: `typedef USHORT *PUSHORT;`
`PVOID`	A pointer to any type. This type is declared in WinNT.h as follows: `typedef void *PVOID;`
`PWCHAR`	A pointer to a **WCHAR**. This type is declared in WinNT.h as follows: `typedef WCHAR *PWCHAR;`
`PWORD`	A pointer to a **WORD**. This type is declared in WinDef.h as follows: `typedef WORD *PWORD;`
`PWSTR`	A pointer to a null-terminated string of 16-bit Unicode characters. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef WCHAR *PWSTR;`
`QWORD`	A 64-bit unsigned integer.

Data type	Description
	This type is declared as follows: `typedef unsigned __int64 QWORD;`
`SC_HANDLE`	A handle to a service control manager database. For more information, see SCM Handles. This type is declared in WinSvc.h as follows: `typedef HANDLE SC_HANDLE;`
`SC_LOCK`	A lock to a service control manager database. For more information, see SCM Handles. This type is declared in WinSvc.h as follows: `typedef LPVOID SC_LOCK;`
`SERVICE_STATUS_HANDLE`	A handle to a service status value. For more information, see SCM Handles. This type is declared in WinSvc.h as follows: `typedef HANDLE SERVICE_STATUS_HANDLE;`
`SHORT`	A 16-bit integer. The range is -32768 through 32767 decimal. This type is declared in WinNT.h as follows: `typedef short SHORT;`
`SIZE_T`	The maximum number of bytes to which a pointer can point. Use for a count that must span the full range of a pointer. This type is declared in BaseTsd.h as follows: `typedef ULONG_PTR SIZE_T;`
`SSIZE_T`	A signed version of **SIZE_T**. This type is declared in BaseTsd.h as follows: `typedef LONG_PTR SSIZE_T;`
`TBYTE`	A **WCHAR** if **UNICODE** is defined, a **CHAR** otherwise. This type is declared in WinNT.h as follows: Expand table

Data type	Description
	C++ ``` #ifdef UNICODE typedef WCHAR TBYTE; #else typedef unsigned char TBYTE; #endif ```
`TCHAR`	A **WCHAR** if **UNICODE** is defined, a **CHAR** otherwise. This type is declared in WinNT.h as follows: Expand table **C++** ``` #ifdef UNICODE typedef WCHAR TCHAR; #else typedef char TCHAR; #endif ```
`UCHAR`	An unsigned **CHAR**. This type is declared in WinDef.h as follows: `typedef unsigned char UCHAR;`
`UHALF_PTR`	An unsigned **HALF_PTR**. Use within a structure that contains a pointer and two small fields. This type is declared in BaseTsd.h as follows: Expand table **C++** ``` #ifdef _WIN64 typedef unsigned int UHALF_PTR; #else ```

Data type	Description
	C++ `  typedef unsigned short UHALF_PTR;` `#endif`
`UINT`	An unsigned **INT**. The range is 0 through 4294967295 decimal. This type is declared in WinDef.h as follows: `typedef unsigned int UINT;`
`UINT_PTR`	An unsigned **INT_PTR**. This type is declared in BaseTsd.h as follows: Expand table **C++** `#if defined(_WIN64)` `  typedef unsigned __int64 UINT_PTR;` `#else` `  typedef unsigned int UINT_PTR;` `#endif`
`UINT8`	An unsigned **INT8**. This type is declared in BaseTsd.h as follows: `typedef unsigned char UINT8;`
`UINT16`	An unsigned **INT16**. This type is declared in BaseTsd.h as follows: `typedef unsigned short UINT16;`
`UINT32`	An unsigned **INT32**. The range is 0 through 4294967295 decimal. This type is declared in BaseTsd.h as follows: `typedef unsigned int UINT32;`
`UINT64`	An unsigned **INT64**. The range is 0 through 18446744073709551615 decimal. This type is declared in BaseTsd.h as follows:

Data type	Description
	`typedef unsigned __int64 UINT64;`
`ULONG`	An unsigned LONG. The range is 0 through 4294967295 decimal. This type is declared in WinDef.h as follows: `typedef unsigned long ULONG;`
`ULONGLONG`	A 64-bit unsigned integer. The range is 0 through 18446744073709551615 decimal. This type is declared in WinNT.h as follows: Expand table **C++** `#if !defined(_M_IX86)` `  typedef unsigned __int64 ULONGLONG;` `#else` `  typedef double ULONGLONG;` `#endif`
`ULONG_PTR`	An unsigned LONG_PTR. This type is declared in BaseTsd.h as follows: Expand table **C++** `#if defined(_WIN64)` `  typedef unsigned __int64 ULONG_PTR;` `#else` `  typedef unsigned long ULONG_PTR;` `#endif`
`ULONG32`	An unsigned LONG32. The range is 0 through 4294967295 decimal. This type is declared in BaseTsd.h as follows: `typedef unsigned int ULONG32;`
`ULONG64`	An unsigned LONG64. The range is 0 through 18446744073709551615 decimal.

Data type	Description
	This type is declared in BaseTsd.h as follows: `typedef unsigned __int64 ULONG64;`
`UNICODE_STRING`	A Unicode string. This type is declared in Winternl.h as follows: Expand table **C++** ``` typedef struct _UNICODE_STRING { USHORT Length; USHORT MaximumLength; PWSTR Buffer; } UNICODE_STRING; typedef UNICODE_STRING *PUNICODE_STRING; typedef const UNICODE_STRING *PCUNICODE_STRING; ```
`USHORT`	An unsigned **SHORT**. The range is 0 through 65535 decimal. This type is declared in WinDef.h as follows: `typedef unsigned short USHORT;`
`USN`	An update sequence number (USN). This type is declared in WinNT.h as follows: `typedef LONGLONG USN;`
`VOID`	Any type. This type is declared in WinNT.h as follows: `#define VOID void`
`WCHAR`	A 16-bit Unicode character. For more information, see Character Sets Used By Fonts. This type is declared in WinNT.h as follows: `typedef wchar_t WCHAR;`
`WINAPI`	The calling convention for system functions. This type is declared in WinDef.h as follows:

Data type	Description
	`#define WINAPI __stdcall` **CALLBACK**, **WINAPI**, and **APIENTRY** are all used to define functions with the __stdcall calling convention. Most functions in the Windows API are declared using **WINAPI**. You may wish to use **CALLBACK** for the callback functions that you implement to help identify the function as a callback function.
`WORD`	A 16-bit unsigned integer. The range is 0 through 65535 decimal. This type is declared in WinDef.h as follows: `typedef unsigned short WORD;`
`WPARAM`	A message parameter. This type is declared in WinDef.h as follows: `typedef UINT_PTR WPARAM;`

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	BaseTsd.h; WinDef.h; WinNT.h

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Large Integers

Article • 08/19/2020

The large integer functions and structures originally provided support for 64-bit values on 32-bit Windows. Now, your C compiler may support 64-bit integers natively. For example, Microsoft Visual C++ supports the **__int64** sized integer type. For more information, see the documentation included with your C compiler.

For information on 64-bit integers on 64-bit Windows, see The New Data Types.

Large Integer Operations

Applications can multiply signed or unsigned 32-bit integers, generating 64-bit results, by using the **Int32x32To64** and **UInt32x32To64** functions. Applications can shift bits in 64-bit values to the left or right by using the **Int64ShllMod32**, **Int64ShraMod32**, and **Int64ShrlMod32** functions. These functions provide logical and arithmetic shifting.

Applications can also multiply and divide 32-bit values in a single operation by using the **MulDiv** function. Although the result of the operation is a 32-bit value, the function stores the intermediate result as a 64-bit value, so that information is not lost when large 32-bit values are multiplied and divided.

Large Integer Reference

- Large Integer Functions
- Large Integer Structures

Feedback

Large Integer Functions

Article • 08/23/2019

The following functions are used with large integers.

In this section

Function	Description
Int32x32To64	Multiplies two signed 32-bit integers, returning a signed 64-bit integer result.
Int64ShllMod32	Performs a left logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for left logical shifts where the shift count is in the range 0-31.
Int64ShraMod32	Performs a right arithmetic shift operation on a signed 64-bit integer value. The function provides improved shifting code for right arithmetic shifts where the shift count is in the range 0-31.
Int64ShrlMod32	Performs a right logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for right logical shifts where the shift count is in the range 0-31.
MulDiv	Multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value.
Multiply128	Multiplies two 64-bit integers to produce a 128-bit integer.
MultiplyExtract128	Multiplies two 64-bit integers to produce a 128-bit integer, shifts the product to the right by the specified number of bits, and returns the low 64 bits of the result.
MultiplyHigh	Multiplies two 64-bit integers to produce a 128-bit integer and gets the high 64 bits.
PopulationCount64	Counts the number of one bits (population count) in a 64-bit unsigned integer.
ShiftLeft128	Shifts 128-bit left.
ShiftRight128	Shifts 128-bit right.
UInt32x32To64	Multiplies two unsigned 32-bit integers, returning an unsigned 64-bit integer result.
UnsignedMultiply128	Multiplies two unsigned 64-bit integers to produce an unsigned 128-bit integer.

Function	Description
UnsignedMultiplyExtract128	Multiplies two unsigned 64-bit integers to produce an unsigned 128-bit integer, shifts the product to the right by the specified number of bits, and returns the low 64 bits of the result.
UnsignedMulitplyHigh	Multiplies two 64-bit integers to produce a 128-bit integer and gets the high unsigned 64 bits.

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Int32x32To64 macro (winnt.h)

07/09/2025

Multiplies two signed 32-bit integers, returning a signed 64-bit integer result. The function performs optimally on 32-bit Windows.

Syntax

```
C++

LONGLONG Int32x32To64(
  [in]  LONG a,
  [in]  LONG b
);
```

Parameters

`[in] a`

The first signed 32-bit integer for the multiplication operation.

`[in] b`

The second signed 32-bit integer for the multiplication operation.

Return value

Type: **LONGLONG**

The return value is the signed 64-bit integer result of the multiplication operation.

Remarks

This function is implemented on all platforms by optimal inline code: a single multiply instruction that returns a 64-bit result.

Please note that the function's return value is a 64-bit value, not a LARGE_INTEGER structure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

Large Integers

UInt32x32To64

Int64ShllMod32 macro (winnt.h)

07/09/2025

Performs a left logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for left logical shifts where the shift count is in the range 0-31.

Syntax

```
C++

ULONGLONG Int64ShllMod32(
  [in]  ULONGLONG a,
  [in]  DWORD b
);
```

Parameters

`[in] a`

The unsigned 64-bit integer to be shifted.

`[in] b`

The shift count in the range 0-31.

Return value

Type: **ULONGLONG**

The return value is the unsigned 64-bit integer result of the left logical shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a left logical shift operation on an unsigned value, the value's bits move to the left, and vacated bits on the right side of the value are set to zero.

A compiler can generate optimal code for a left logical shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown, the compiler must assume the worst case, leading to non-optimal code: code that calls a

subroutine, or code that is inline but branches. By restricting the shift count to the range 0-31, the **Int64ShllMod32** function lets the compiler generate optimal or near-optimal code.

Please note that the **Int64ShllMod32** function's *Value* parameter and return value are 64-bit values, not LARGE_INTEGER structures.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

Int64ShraMod32

Int64ShrlMod32

Large Integers

Int64ShraMod32 macro (winnt.h)

07/09/2025

Performs a right arithmetic shift operation on a signed 64-bit integer value. The function provides improved shifting code for right arithmetic shifts where the shift count is in the range 0-31.

Syntax

```
C++

LONGLONG Int64ShraMod32(
  [in]  LONGLONG a,
  [in]  DWORD b
);
```

Parameters

`[in] a`

The signed 64-bit integer to be shifted.

`[in] b`

The shift count in the range 0-31.

Return value

Type: **LONGLONG**

The return value is the signed 64-bit integer result of the right arithmetic shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a right arithmetic shift operation on a signed value, the value's bits move to the right, and vacated bits on the left side of the value are set to the value of the sign bit.

A compiler can generate optimal code for a right arithmetic shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown,

the compiler must assume the worst case, leading to non-optimal code: code that calls a subroutine, or code that is inline but branches. By restricting the shift count to the range 0-31, the **Int64ShraMod32** function lets the compiler generate optimal or near-optimal code.

Please note that the **Int64ShraMod32** function's *Value* parameter and return value are 64-bit values, not LARGE_INTEGER structures.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

Int64ShllMod32

Int64ShrlMod32

Large Integers

Int64ShrlMod32 macro (winnt.h)

07/09/2025

Performs a right logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for right logical shifts where the shift count is in the range 0-31.

Syntax

```
C++

ULONGLONG Int64ShrlMod32(
  [in]  ULONGLONG a,
  [in]  DWORD b
);
```

Parameters

`[in] a`

The unsigned 64-bit integer to be shifted.

`[in] b`

The shift count in the range 0-31.

Return value

Type: **ULONGLONG**

The return value is the unsigned 64-bit integer result of the right logical shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a right logical shift operation on an unsigned value, the value's bits move to the right, and vacated bits on the left side of the value are set to zero.

A compiler can generate optimal code for a right logical shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown, the

compiler must assume the worst case, leading to non-optimal code: code that calls a subroutine, or code that is inline but branches. By restricting the shift count to the range 0-31, the **Int64ShrlMod32** function lets the compiler generate optimal or near-optimal code.

> **Note** The **Int64ShrlMod32** function's *Value* parameter and return value are 64-bit values, not **LARGE_INTEGER** structures.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

Int64ShllMod32

Int64ShraMod32

Large Integers

MulDiv function (winbase.h)

02/22/2024

Multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value. The final result is rounded to the nearest integer.

Syntax

```
C++

int MulDiv(
  [in] int nNumber,
  [in] int nNumerator,
  [in] int nDenominator
);
```

Parameters

`[in] nNumber`

The multiplicand.

`[in] nNumerator`

The multiplier.

`[in] nDenominator`

The number by which the result of the multiplication operation is to be divided.

Return value

If the function succeeds, the return value is the result of the multiplication and division, rounded to the nearest integer. If the result is a positive half integer (ends in .5), it is rounded up. If the result is a negative half integer, it is rounded down.

If either an overflow occurred or *nDenominator* was 0, the return value is -1.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Int32x32To64

Large Integers

UInt32x32To64

Multiply128 function (winnt.h)

Article02/22/2024

Multiplies two 64-bit integers to produce a 128-bit integer.

Syntax

```
C++

LONG64 Multiply128(
  [in]  LONG64 Multiplier,
  [in]  LONG64 Multiplicand,
  [out] LONG64 *HighProduct
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

`[out] HighProduct`

The high 64 bits of the product.

Return value

The low 64 bits of the product.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

__mul128

MultiplyExtract128 function (winnt.h)

Article02/22/2024

Multiplies two 64-bit integers to produce a 128-bit integer, shifts the product to the right by the specified number of bits, and returns the low 64 bits of the result.

Syntax

```
C++

LONG64 MultiplyExtract128(
  [in] LONG64 Multiplier,
  [in] LONG64 Multiplicand,
  [in] BYTE   Shift
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

`[in] Shift`

The number of bits to shift.

Return value

The low 64 bits of the result.

Requirements

Expand table

Requirement	Value
Target Platform	Windows

Requirement	Value
Header	winnt.h

MultiplyHigh function (winnt.h)

Article02/22/2024

Multiplies two 64-bit integers to produce a 128-bit integer and gets the high 64 bits.

Syntax

```
C++

LONGLONG MultiplyHigh(
  [in] LONG64 Multiplier,
  [in] LONG64 Multiplicand
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

Return value

The high 64 bits of the product.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

PopulationCount64 function (winnt.h)

Article02/22/2024

Counts the number of one bits (population count) in a 64-bit unsigned integer.

Syntax

```
C++

DWORD64 PopulationCount64(
  [in] DWORD64 operand
);
```

Parameters

`[in] operand`

The operand.

Return value

The count of one bits.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

ShiftLeft128 function (winnt.h)

Article02/22/2024

Shifts 128-bit left.

Syntax

```
C++
```

```
DWORD64 ShiftLeft128(
  DWORD64 LowPart,
  DWORD64 HighPart,
  BYTE    Shift
);
```

Parameters

`LowPart`

The low 64 bits.

`HighPart`

The high 64 bits.

`Shift`

Bytes to shift.

Return value

The shifted bits.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

ShiftRight128 function (winnt.h)

Article02/22/2024

Shifts 128-bit right.

Syntax

```
C++

DWORD64 ShiftRight128(
  DWORD64 LowPart,
  DWORD64 HighPart,
  BYTE    Shift
);
```

Parameters

`LowPart`

The low 64 bits.

`HighPart`

The high 64 bits.

`Shift`

Bytes to shift.

Return value

The shifted bits.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

UInt32x32To64 macro (winnt.h)

07/09/2025

Multiplies two unsigned 32-bit integers, returning an unsigned 64-bit integer result. The function performs optimally on 32-bit Windows.

Syntax

```
C++

LONGLONG Int32x32To64(
  [in]  LONG a,
  [in]  LONG b
);
```

Parameters

`[in] a`

The first unsigned 32-bit integer for the multiplication operation.

`[in] b`

The second unsigned 32-bit integer for the multiplication operation.

Return value

Type: **LONGLONG**

The return value is the signed 64-bit integer result of the multiplication operation.

Remarks

This function is implemented on all platforms by optimal inline code: a single multiply instruction that returns a 64-bit result.

Please note that the function's return value is a 64-bit value, not a LARGE_INTEGER structure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

Int32x32To64

Large Integers

UnsignedMultiply128 function (winnt.h)

Article02/22/2024

Multiplies two unsigned 64-bit integers to produce an unsigned 128-bit integer.

Syntax

```
C++
```

```
DWORD64 UnsignedMultiply128(
  [in]  DWORD64 Multiplier,
  [in]  DWORD64 Multiplicand,
  [out] DWORD64 *HighProduct
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

`[out] HighProduct`

The high 64 bits of the product.

Return value

The low 64 bits of the product.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

_umu128

UnsignedMultiplyExtract128 function (winnt.h)

Article02/22/2024

Multiplies two unsigned 64-bit integers to produce an unsigned 128-bit integer, shifts the product to the right by the specified number of bits, and returns the low 64 bits of the result.

Syntax

```
C++

DWORD64 UnsignedMultiplyExtract128(
  [in] DWORD64 Multiplier,
  [in] DWORD64 Multiplicand,
  [in] BYTE    Shift
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

`[in] Shift`

The number of bits to shift.

Return value

The low 64 bits of the result.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

UnsignedMultiplyHigh function (winnt.h)

Article02/22/2024

Multiplies two 64-bit integers to produce a 128-bit integer and gets the high unsigned 64 bits.

Syntax

```
C++
```

```
ULONGLONG UnsignedMultiplyHigh(
  [in] DWORD64 Multiplier,
  [in] DWORD64 Multiplicand
);
```

Parameters

`[in] Multiplier`

The first integer.

`[in] Multiplicand`

The second integer.

Return value

The high 64 bits of the product.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

Large Integer Structures

Article • 02/08/2021

The following structures represent large integers:
[**LARGE_INTEGER**](/windows/win32/api/winnt/ns-winnt-large_integer-r1)
[**ULARGE_INTEGER**](/windows/win32/api/winnt/ns-winnt-ularge_integer-r1)

Feedback

LARGE_INTEGER union (winnt.h)

Article02/22/2024

Represents a 64-bit signed integer value.

> **Note** Your C compiler may support 64-bit integers natively. For example, Microsoft Visual C++ supports the **__int64** sized integer type. For more information, see the documentation included with your C compiler.

Syntax

```
C++
```

```
typedef union _LARGE_INTEGER {
  struct {
    DWORD LowPart;
    LONG  HighPart;
  } DUMMYSTRUCTNAME;
  struct {
    DWORD LowPart;
    LONG  HighPart;
  } u;
  LONGLONG QuadPart;
} LARGE_INTEGER;
```

Members

`DUMMYSTRUCTNAME`

`DUMMYSTRUCTNAME.LowPart`

`DUMMYSTRUCTNAME.HighPart`

`u`

`u.LowPart`

`u.HighPart`

`QuadPart`

A signed 64-bit integer.

Remarks

The **LARGE_INTEGER** structure is actually a union. If your compiler has built-in support for 64-bit integers, use the **QuadPart** member to store the 64-bit integer. Otherwise, use the **LowPart** and **HighPart** members to store the 64-bit integer.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winnt.h

See also

ULARGE_INTEGER

Feedback

ULARGE_INTEGER union (winnt.h)

Article02/22/2024

Represents a 64-bit unsigned integer value.

> **Note** Your C compiler may support 64-bit integers natively. For example, Microsoft Visual C++ supports the **__int64** sized integer type. For more information, see the documentation included with your C compiler.

Syntax

```
C++

typedef union _ULARGE_INTEGER {
  struct {
    DWORD LowPart;
    DWORD HighPart;
  } DUMMYSTRUCTNAME;
  struct {
    DWORD LowPart;
    DWORD HighPart;
  } u;
  ULONGLONG QuadPart;
} ULARGE_INTEGER;
```

Members

`DUMMYSTRUCTNAME`

`DUMMYSTRUCTNAME.LowPart`

`DUMMYSTRUCTNAME.HighPart`

`u`

`u.LowPart`

`u.HighPart`

`QuadPart`

An unsigned 64-bit integer.

Remarks

The **ULARGE_INTEGER** structure is actually a union. If your compiler has built-in support for 64-bit integers, use the **QuadPart** member to store the 64-bit integer. Otherwise, use the **LowPart** and **HighPart** members to store the 64-bit integer.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winnt.h

See also

FILETIME

LARGE_INTEGER

SYSTEMTIME