Security Best Practices for C++

Visual Studio 2015 Other Versions

This article contains information about security tools and practices. Using them does not make applications immune from attack, but it makes successful attacks less likely.

Visual C++ Security Features

These security features are built into the Visual C++ compiler and linker:

/guard (Enable Control Flow Guard)

Causes the compiler to to analyze control flow for indirect call targets at compile time, and then to insert code to verify the targets at runtime.

/GS (Buffer Security Check)

Instructs the compiler to insert overrun detection code into functions that are at risk of being exploited. When an overrun is detected, execution is stopped. By default, this option is on.

/SAFESEH (Image has Safe Exception Handlers)

Instructs the linker to include in the output image a table that contains the address of each exception handler. At run time, the operating system uses this table to make sure that only legitimate exception handlers are executed. This helps prevent the execution of exception handlers that are introduced by a malicious attack at run time. By default, this option is off.

/NXCOMPAT, /NXCOMPAT (Compatible with Data Execution Prevention)

These compiler and linker options enable Data Execution Prevention (DEP) compatibility. DEP guards the CPU against the execution of non-code pages.

/analyze (Code Analysis)

This compiler option activates code analysis that reports potential security issues such as buffer overrun, un-initialized memory, null pointer dereferencing, and memory leaks. By default, this option is off. For more information, see Code Analysis for C/C++ Overview.

/DYNAMICBASE (Use address space layout randomization)

This linker option enables the building of an executable image that can be loaded at different locations in memory at the beginning of execution. This option also makes the stack location in memory much less predictable.

Security-Enhanced CRT

The C Runtime Library (CRT) has been augmented to include secure versions of functions that pose security risks—for example, the unchecked strcpy string copy function. Because the older, nonsecure versions of these functions are deprecated, they cause compile-time warnings. We encourage you to use the secure versions of these CRT functions instead of suppressing the compilation warnings. For more information, see Security Features in the CRT.

SafeInt Library

SafeInt Library helps prevent integer overflows and other exploitable errors that might occur when the application performs mathematical operations. The SafeInt library includes the SafeInt Class, the SafeIntException Class, and several SafeInt Functions.

The SafeInt class protects against integer overflow and divide-by-zero exploits. You can use it to handle comparisons between values of different types. I provides two error handling policies. The default policy is for the SafeInt class to throw a SafeIntException class exception to report why a mathematical operation cannot be completed. The second policy is for the SafeInt class to stop program execution. You can also define a custom policy.

Each SafeInt function protects one mathematical operation from an exploitable error. You can use two different kinds of parameters without converting them to the same type. To protect multiple mathematical operations, use the SafeInt class.

Checked Iterators

A checked iterator enforces container boundaries. By default, when a checked iterator is out of bounds, it generates an exception and ends program execution. A checked iterator provides other levels of response that depend on values that are assigned to preprocessor defines such as \_SECURE\_SCL\_THROWS and \_ITERATOR\_DEBUG\_LEVEL. For example, at \_ITERATOR\_DEBUG\_LEVEL=2, a checked iterator provides comprehensive correctness checks in debug mode, which are made available by using asserts. For more information, see Checked Iterators.

Code Analysis for Managed Code

Code Analysis for Managed Code, also known as FxCop, checks assemblies for conformance to the.NET Framework design guidelines. FxCop analyzes the code and metadata in each assembly to check for defects in the following areas:

Library design

Localization

Naming conventions

Performance

Security

Windows Application Verifier

The Application Verifier (AppVerifier) can help you identify potential application compatibility, stability, and security issues.

The AppVerifier monitors how an application uses the operating system. It watches the file system, registry, memory, and APIs while the application is running, and recommends source-code fixes for issues that it uncovers.

You can use the AppVerifier to:

Test for potential application compatibility errors that are caused by common programming mistakes.

Examine an application for memory-related issues.

Identify potential security issues in an application.

The AppVerifier is part of the Application Compatibility Toolkit, which is available from the Application Compatibility on the TechNet web site.

.NET Framework Security Features

NIB: Configuring Security Policy describes guidelines and tools for adjusting the .NET Framework security policies.

Windows User Accounts

Using Windows user accounts that belong to the Administrators group exposes developers and--by extension--customers to security risks. For more information, see Running as a Member of the Users Group and How User Account Control (UAC) Affects Your Application.

/guard (Enable Control Flow Guard)

**Visual Studio 2015**

Enable compiler generation of Control Flow Guard security checks.

[Syntax](javascript:void(0))

/guard:cf[-]

[Remarks](javascript:void(0))

The **/guard:cf** option causes the compiler to analyze control flow for indirect call targets at compile time, and then to insert code to verify the targets at runtime. By default, **/guard:cf** is off and must be explicitly enabled. To explicitly disable this option, use **/guard:cf-**.

When the **/guard:cf** Control Flow Guard (CFG) option is specified, the compiler and linker insert extra runtime security checks to detect attempts to compromise your code. During compiling and linking, all indirect calls in your code are analyzed to find every location that the code can reach when it runs correctly. This information is stored in extra structures in the headers of your binaries. The compiler also injects a check before every indirect call in your code that ensures the target is one of the verified locations. If the check fails at runtime on a CFG-aware operating system, the operating system closes the program.

A common attack on software takes advantage of bugs in handling extreme or unexpected inputs. Carefully crafted input to the application may overwrite a location that contains a pointer to executable code. This can be used to redirect control flow to code controlled by the attacker. The CFG runtime checks do not fix the data corruption bugs in your executable. They instead make it more difficult for an attacker to use them to execute arbitrary code. CFG is a mitigation tool that prevents calls to locations other than function entry points in your code. It's similar to how Data Execution Prevention (DEP), [/GS](https://msdn.microsoft.com/en-us/library/8dbf701c.aspx) stack checks, and [/DYNAMICBASE](https://msdn.microsoft.com/en-us/library/bb384887.aspx) and [/HIGHENTROPYVA](https://msdn.microsoft.com/en-us/library/dn195771.aspx) address space layout randomization (ASLR) lower the chances that your code becomes an exploit vector.

The **/guard:cf** option must be passed to both the compiler and linker to build code that uses the CFG exploit mitigation technique. If your binary is built by using a single cl command, the compiler passes the option to the linker. If you compile and link separately, the option must be set on both the compiler and linker commands. The /DYNAMICBASE linker option is also required. To verify that your binary has CFG data, use the dumpbin /headers /loadconfig command. CFG-enabled binaries have Guard in the list of EXE or DLL characteristics, and Guard Flags include CF Instrumented and FID table present.

The **/guard:cf** option is incompatible with [/ZI](https://msdn.microsoft.com/en-us/library/958x11bc.aspx) (Edit and Continue debug information) or [/clr](https://msdn.microsoft.com/en-us/library/k8d11d4s.aspx) (Common Language Runtime Compilation).

Code compiled by using **/guard:cf** can be linked to libraries and object files that are not compiled by using the option. Only this code, when also linked by using the **/guard:cf** option and run on a CFG-aware operating system, has CFG protection. Because code compiled without the option will not stop an attack, we recommend that you use the option on all the code you compile. There is a small runtime cost for CFG checks, but the compiler analysis attempts to optimize away the checks on indirect jumps that can be proven to be safe.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see [How to: Open Project Property Pages](https://msdn.microsoft.com/en-us/library/e79xc5h1.aspx).
2. Select **Configuration Properties**, **C/C++**, **Code Generation**.
3. Select the **Control Flow Guard** property.
4. In the dropdown control, choose **Yes** to enable Control Flow Guard, or **No** to disable it.

# /GS (Buffer Security Check)

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

Detects some buffer overruns that overwrite a function's return address, exception handler address, or certain types of parameters. Causing a buffer overrun is a technique used by hackers to exploit code that does not enforce buffer size restrictions.

## [Syntax](javascript:void(0))

/GS[-]

## [Remarks](javascript:void(0))

**/GS** is on by default. If you expect your application to have no security exposure, use **/GS-**. For more information about **/GS**, see [Compiler Security Checks In Depth](http://go.microsoft.com/fwlink/?linkid=7260). For more information about suppressing buffer overrun detection, see [safebuffers](https://msdn.microsoft.com/en-us/library/dd778695.aspx).

### [Security Checks](javascript:void(0))

On functions that the compiler recognizes as subject to buffer overrun problems, the compiler allocates space on the stack before the return address. On function entry, the allocated space is loaded with a security cookie that is computed once at module load. On function exit, and during frame unwinding on 64-bit operating systems, a helper function is called to make sure that the value of the cookie is still the same. A different value indicates that an overwrite of the stack may have occurred. If a different value is detected, the process is terminated.

### [GS Buffers](javascript:void(0))

A buffer overrun security check is performed on a GS buffer. A GS buffer can be one of these:

* An array that is larger than 4 bytes, has more than two elements, and has an element type that is not a pointer type.
* A data structure whose size is more than 8 bytes and contains no pointers.
* A buffer allocated by using the [\_alloca](https://msdn.microsoft.com/en-us/library/wb1s57t5.aspx) function.
* Any class or structure that contains a GS buffer.

For example, the following statements declare GS buffers.

char buffer[20];

int buffer[20];

struct { int a; int b; int c; int d; } myStruct;

struct { int a; char buf[20]; };

However, the following statements do not declare GS buffers. The first two declarations contain elements of pointer type. The third and fourth statements declare arrays whose size is too small. The fifth statement declares a structure whose size on an x86 platform is not more than 8 bytes.

char \*pBuf[20];

void \*pv[20];

char buf[4];

int buf[2];

struct { int a; int b; };

### [Initialize the Security Cookie](javascript:void(0))

The **/GS** compiler option requires that the security cookie be initialized before any function that uses the cookie is run. The security cookie must be initialized on entry to an EXE or DLL. This is done automatically if you use the default CRT entry points (mainCRTStartup, wmainCRTStartup, WinMainCRTStartup, wWinMainCRTStartup, or \_DllMainCRTStartup). If you use an alternate entry point, you must manually initialize the security cookie by calling [\_\_security\_init\_cookie](https://msdn.microsoft.com/en-us/library/ms235362.aspx).

### [What Is Protected](javascript:void(0))

The **/GS** compiler option protects the following items:

* The return address of a function call.
* The address of an exception handler for a function.
* Vulnerable function parameters.

On all platforms, **/GS** attempts to detect buffer overruns into the return address. Buffer overruns are more easily exploited on platforms such as x86 and x64, which use calling conventions that store the return address of a function call on the stack.

On x86, if a function uses an exception handler, the compiler injects a security cookie to protect the address of the exception handler. The cookie is checked during frame unwinding.

**/GS** protects vulnerable parameters that are passed into a function. A vulnerable parameter is a pointer, a C++ reference, a C-structure (C++ POD type) that contains a pointer, or a GS buffer.

A vulnerable parameter is allocated before the cookie and local variables. A buffer overrun can overwrite these parameters. And code in the function that uses these parameters could cause an attack before the function returns and the security check is performed. To minimize this danger, the compiler makes a copy of the vulnerable parameters during the function prolog and puts them below the storage area for any buffers.

The compiler does not make copies of vulnerable parameters in the following situations:

* Functions that do not contain a GS buffer.
* Optimizations ([/O options](https://msdn.microsoft.com/en-us/library/k1ack8f1.aspx)) are not enabled.
* Functions that have a variable argument list (...).
* Functions that are marked with [naked](https://msdn.microsoft.com/en-us/library/h5w10wxs.aspx).
* Functions that contain inline assembly code in the first statement.
* A parameter is used only in ways that are less likely to be exploitable in the event of a buffer overrun.

### [What Is Not Protected](javascript:void(0))

The **/GS** compiler option does not protect against all buffer overrun security attacks. For example, if you have a buffer and a vtable in an object, a buffer overrun could corrupt the vtable.

Even if you use **/GS**, always try to write secure code that has no buffer overruns.

### To set this compiler option in Visual Studio

1. In **Solution Explorer**, right-click the project and then click **Properties**.

For more information, see [How to: Open Project Property Pages](https://msdn.microsoft.com/en-us/library/e79xc5h1.aspx).

1. In the **Property Pages** dialog box, click the **C/C++** folder.
2. Click the **Code Generation** property page.
3. Modify the **Buffer Security Check** property.

### To set this compiler option programmatically

* See [BufferSecurityCheck](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.vcclcompilertool.buffersecuritycheck.aspx).

## Example

This sample overruns a buffer. This causes the application to fail at runtime.

// compile with: /c /W1

#include <cstring>

#include <stdlib.h>

#pragma warning(disable : 4996) // for strcpy use

// Vulnerable function

void vulnerable(const char \*str) {

char buffer[10];

strcpy(buffer, str); // overrun buffer !!!

// use a secure CRT function to help prevent buffer overruns

// truncate string to fit a 10 byte buffer

// strncpy\_s(buffer, \_countof(buffer), str, \_TRUNCATE);

}

int main() {

// declare buffer that is bigger than expected

char large\_buffer[] = "This string is longer than 10 characters!!";

vulnerable(large\_buffer);

}

/SAFESEH (Image has Safe Exception Handlers)

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

/SAFESEH[:NO]

When **/SAFESEH** is specified, the linker will only produce an image if it can also produce a table of the image's safe exception handlers. This table specifies for the operating system which exception handlers are valid for the image.

**/SAFESEH** is only valid when linking for x86 targets. **/SAFESEH** is not supported for platforms that already have the exception handlers noted. For example, on x64 and ARM, all exception handlers are noted in the PDATA. ML64.exe has support for adding annotations that emit SEH information (XDATA and PDATA) into the image, allowing you to unwind through ml64 functions. See [MASM for x64 (ml64.exe)](https://msdn.microsoft.com/en-us/library/hb5z4sxd.aspx) for more information.

If **/SAFESEH** is not specified, the linker will produce an image with a table of safe exceptions handlers if all modules are compatible with the safe exception handling feature. If any modules were not compatible with safe exception handling feature, the resulting image will not contain a table of safe exception handlers. If [/SUBSYSTEM](https://msdn.microsoft.com/en-us/library/fcc1zstk.aspx) specifies WINDOWSCE or one of the EFI\_\* options, the linker will not attempt to produce an image with a table of safe exceptions handlers, as neither of those subsystems can make use of the information.

If **/SAFESEH:NO** is specified, the linker will not produce an image with a table of safe exceptions handlers even if all modules are compatible with the safe exception handling feature.

The most common reason for the linker not to be able to produce an image is because one or more of the input files (modules) to the linker was not compatible with the safe exception handlers feature. A common reason for a module to not be compatible with safe exception handlers is because it was created with a compiler from a previous version of Visual C++.

You can also register a function as a structured exception handler by using [.SAFESEH](https://msdn.microsoft.com/en-us/library/16aexws6.aspx).

It is not possible to mark an existing binary as having safe exception handlers (or no exception handlers); information on safe exception handling must be added at build time.

The linker's ability to build a table of safe exception handlers depends on the application using the C runtime library. If you link with [/NODEFAULTLIB](https://msdn.microsoft.com/en-us/library/3tz4da4a.aspx)and you want a table of safe exception handlers, you need to supply a load config struct (such as can be found in loadcfg.c CRT source file) that contains all the entries defined for Visual C++. For example:

#include <windows.h>

extern DWORD\_PTR \_\_security\_cookie; /\* /GS security cookie \*/

/\*

\* The following two names are automatically created by the linker for any

\* image that has the safe exception table present.

\*/

extern PVOID \_\_safe\_se\_handler\_table[]; /\* base of safe handler entry table \*/

extern BYTE \_\_safe\_se\_handler\_count; /\* absolute symbol whose address is

the count of table entries \*/

typedef struct {

DWORD Size;

DWORD TimeDateStamp;

WORD MajorVersion;

WORD MinorVersion;

DWORD GlobalFlagsClear;

DWORD GlobalFlagsSet;

DWORD CriticalSectionDefaultTimeout;

DWORD DeCommitFreeBlockThreshold;

DWORD DeCommitTotalFreeThreshold;

DWORD LockPrefixTable; // VA

DWORD MaximumAllocationSize;

DWORD VirtualMemoryThreshold;

DWORD ProcessHeapFlags;

DWORD ProcessAffinityMask;

WORD CSDVersion;

WORD Reserved1;

DWORD EditList; // VA

DWORD\_PTR \*SecurityCookie;

PVOID \*SEHandlerTable;

DWORD SEHandlerCount;

} IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2;

const IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2 \_load\_config\_used = {

sizeof(IMAGE\_LOAD\_CONFIG\_DIRECTORY32\_2),

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

0,

&\_\_security\_cookie,

\_\_safe\_se\_handler\_table,

(DWORD)(DWORD\_PTR) &\_\_safe\_se\_handler\_count

};

To set this linker option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see [Setting Visual C++ Project Properties](https://msdn.microsoft.com/en-us/library/669zx6zc.aspx).
2. Select the **Linker** folder.
3. Select the **Command Line** property page.
4. Enter the option into the **Additional Options** box.

/NXCOMPAT

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

/NXCOMPAT[:NO]

[Remarks](javascript:void(0))

Indicates that an executable was tested to be compatible with the Windows Data Execution Prevention feature.

For more information, see [/NXCOMPAT (Compatible with Data Execution Prevention)](https://msdn.microsoft.com/en-us/library/ms235442.aspx).

/analyze (Code Analysis)

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

Enables code analysis and control options.

[Syntax](javascript:void(0))

/analyze[-][:WX-][:log filename][:quiet][:stacksize number][:max\_paths number][:only]

[Arguments](javascript:void(0))

/analyze

Turns on analysis in the default mode. Analysis output goes to the **Output** window like other error messages. Use **/analyze-** to explicitly turn off analysis.

/analyze:WX-

Specifying **/analyze:WX-** means that code analysis warnings are not treated as errors when you compile by using **/WX**. For more information, see [/w, /Wn, /WX, /Wall, /wln, /wdn, /wen, /won (Warning Level)](https://msdn.microsoft.com/en-us/library/thxezb7y.aspx).

/analyze:log *filename*

Detailed analyzer results are written as XML to the file that is specified by *filename*.

/analyze:quiet

Turns off analyzer output to the **Output** window.

/analyze:stacksize *number*

The *number* parameter that is used with this option specifies the size, in bytes, of the stack frame for which warning [C6262](https://msdn.microsoft.com/en-us/library/7yhee2f0.aspx) is generated. If this parameter is not specified, the stack frame size is 16KB by default.

/analyze:max\_paths *number*

The *number* parameter that is used with this option specifies the maximum number of code paths to be analyzed. If this parameter is not specified, the number is 256 by default. Larger values perform more thorough checking, but the analysis might take longer.

/analyze:only

Typically, the compiler generates code and does more syntax checking after it runs the analyzer. The **/analyze:only** option turns off this code generation pass; this makes analysis faster but compile errors and warnings that might have been discovered by the code generation pass of the compiler are not emitted. If the program is not free of code-generation errors, analysis results might be unreliable; therefore, we recommend that you use this option only if the code already passes code-generation syntax checking without errors.

[Remarks](javascript:void(0))

For more information, see [Code Analysis for C/C++ Overview](https://msdn.microsoft.com/en-us/library/d3bbz7tz.aspx) and [Code Analysis for C/C++ Warnings](https://msdn.microsoft.com/en-us/library/a5b9aa09.aspx).

/DYNAMICBASE (Use address space layout randomization)

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

Specifies whether to generate an executable image that can be randomly rebased at load time by using the address space layout randomization (ASLR) feature of Windows Vista.

[Syntax](javascript:void(0))

/DYNAMICBASE[:NO]

[Remarks](javascript:void(0))

By default, /DYNAMICBASE is on.

This option modifies the header of an executable to indicate whether the application should be randomly rebased at load time.

Address space layout randomization is supported on Windows Vista.

To set this linker option in Visual Studio

1. Open the project **Property Pages** dialog box. For more information, see [How to: Open Project Property Pages](https://msdn.microsoft.com/en-us/library/e79xc5h1.aspx).
2. Expand the **Configuration Properties** node.
3. Expand the **Linker** node.
4. Select the **Advanced** property page.
5. Modify the **Randomized Base Address** property.

To set this linker option programmatically

1. See [RandomizedBaseAddress](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.vclinkertool.randomizedbaseaddress.aspx).

# VCLinkerTool.RandomizedBaseAddress Property

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

Gets or sets a value controls whether to generate an executable image that can be randomly rebased at load time by using the address space layout randomization (ASLR) feature of Windows Vista.

**Namespace:**  [Microsoft.VisualStudio.VCProjectEngine](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.aspx)  
**Assembly:** Microsoft.VisualStudio.VCProjectEngine (in Microsoft.VisualStudio.VCProjectEngine.dll)

## [Syntax](javascript:void(0))

C#

[**C++**](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.vclinkertool.randomizedbaseaddress.aspx?cs-save-lang=1&cs-lang=cpp#code-snippet-1)

[**F#**](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.vclinkertool.randomizedbaseaddress.aspx?cs-save-lang=1&cs-lang=fsharp#code-snippet-1)

[**VB**](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.vclinkertool.randomizedbaseaddress.aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

enumRandomizedBaseAddressBOOL RandomizedBaseAddress { get; set; }

#### Property Value

Type: [Microsoft.VisualStudio.VCProjectEngine.enumRandomizedBaseAddressBOOL](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.enumrandomizedbaseaddressbool.aspx)

A member of the [enumRandomizedBaseAddressBOOL](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.vcprojectengine.enumrandomizedbaseaddressbool.aspx) enumeration.

SafeInt Library

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

The SafeInt library helps prevent integer overflows that might result when the application performs mathematical operations.

In This Section

|  |  |
| --- | --- |
| **Section** | **Description** |
| [SafeInt Class](https://msdn.microsoft.com/en-us/library/dd570021.aspx) | This class protects against integer overflows. |
| [SafeInt Functions](https://msdn.microsoft.com/en-us/library/dd575188.aspx) | Functions that can be used without creating a **SafeInt** object. |
| [SafeIntException Class](https://msdn.microsoft.com/en-us/library/dd570024.aspx) | A class of exceptions related to the **SafeInt** class. |

# SafeInt Class

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

Extends the integer primitives to help prevent integer overflow and lets you compare different types of integers.

## [Syntax](javascript:void(0))

template<typename T, typename E = \_SAFEINT\_DEFAULT\_ERROR\_POLICY>

class SafeInt;

## [Parameters](javascript:void(0))

|  |  |
| --- | --- |
| **Template** | **Description** |
| T | The type of integer or Boolean parameter that **SafeInt** replaces. |
| E | An enumerated data type that defines the error handling policy. |
| U | The type of integer or Boolean parameter for the secondary operand. |
| **Parameter** | **Description** |
| [in] rhs | An input parameter that represents the value on the right side of the operator in several stand-alone functions. |
| [in] i | An input parameter that represents the value on the right side of the operator in several stand-alone functions. |
| [in] bits | An input parameter that represents the value on the right side of the operator in several stand-alone functions. |

## [Members](javascript:void(0))

### [Public Constructors](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Description** |
| [SafeInt::SafeInt](https://msdn.microsoft.com/en-us/library/dd572129.aspx) | Default constructor. |

### [Assignment Operators](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Syntax** |
| = | template<typename U>  SafeInt<T,E>& operator= (const U& rhs) |
| = | SafeInt<T,E>& operator= (const T& rhs) throw() |
| = | template<typename U>  SafeInt<T,E>& operator= (const SafeInt<U, E>& rhs) |
| = | SafeInt<T,E>& operator= (const SafeInt<T,E>& rhs) throw() |

### [Casting Operators](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Syntax** |
| bool | operator bool() throw() |
| char | operator char() const |
| signed char | operator signed char() const |
| unsigned char | operator unsigned char() const |
| \_\_int16 | operator \_\_int16() const |
| unsigned \_\_int16 | operator unsigned \_\_int16() const |
| \_\_int32 | operator \_\_int32() const |
| unsigned \_\_int32 | operator unsigned \_\_int32() const |
| long | operator long() const |
| unsigned long | operator unsigned long() const |
| \_\_int64 | operator \_\_int64() const |
| unsigned \_\_int64 | operator unsigned \_\_int64() const |
| wchar\_t | operator wchar\_t() const |

### [Comparison Operators](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Syntax** |
| < | template<typename U>  bool operator< (U rhs) const throw() |
| < | bool operator< (SafeInt<T,E> rhs) const throw() |
| >= | template<typename U>  bool operator>= (U rhs) const throw() |
| >= | Bool operator>= (SafeInt<T,E> rhs) const throw() |
| > | template<typename U>  bool operator> (U rhs) const throw() |
| > | Bool operator> (SafeInt<T,E> rhs) const throw() |
| <= | template<typename U>  bool operator<= (U rhs) const throw() |
| <= | bool operator<= (SafeInt<T,E> rhs) const throw() |
| == | template<typename U>  bool operator== (U rhs) const throw() |
| == | bool operator== (bool rhs) const throw() |
| == | bool operator== (SafeInt<T,E> rhs) const throw() |
| != | template<typename U>  bool operator!= (U rhs) const throw() |
| != | bool operator!= (bool b) const throw() |
| != | bool operator!= (SafeInt<T,E> rhs) const throw() |

### [Arithmetic Operators](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Syntax** |
| + | const SafeInt<T,E>& operator+ () const throw() |
| - | SafeInt<T,E> operator- () const |
| ++ | SafeInt<T,E>& operator++ () |
| -- | SafeInt<T,E>& operator-- () |
| % | template<typename U>  SafeInt<T,E> operator% (U rhs) const |
| % | SafeInt<T,E> operator% (SafeInt<T,E> rhs) const |
| %= | template<typename U>  SafeInt<T,E>& operator%= (U rhs) |
| %= | template<typename U>  SafeInt<T,E>& operator%= (SafeInt<U, E> rhs) |
| \* | template<typename U>  SafeInt<T,E> operator\* (U rhs) const |
| \* | SafeInt<T,E> operator\* (SafeInt<T,E> rhs) const |
| \*= | SafeInt<T,E>& operator\*= (SafeInt<T,E> rhs) |
| \*= | template<typename U>  SafeInt<T,E>& operator\*= (U rhs) |
| \*= | template<typename U>  SafeInt<T,E>& operator\*= (SafeInt<U, E> rhs) |
| / | template<typename U>  SafeInt<T,E> operator/ (U rhs) const |
| / | SafeInt<T,E> operator/ (SafeInt<T,E> rhs ) const |
| /= | SafeInt<T,E>& operator/= (SafeInt<T,E> i) |
| /= | template<typename U>  SafeInt<T,E>& operator/= (U i) |
| /= | template<typename U>  SafeInt<T,E>& operator/= (SafeInt<U, E> i) |
| + | SafeInt<T,E> operator+ (SafeInt<T,E> rhs) const |
| + | template<typename U>  SafeInt<T,E> operator+ (U rhs) const |
| += | SafeInt<T,E>& operator+= (SafeInt<T,E> rhs) |
| += | template<typename U>  SafeInt<T,E>& operator+= (U rhs) |
| += | template<typename U>  SafeInt<T,E>& operator+= (SafeInt<U, E> rhs) |
| - | template<typename U>  SafeInt<T,E> operator- (U rhs) const |
| - | SafeInt<T,E> operator- (SafeInt<T,E> rhs) const |
| -= | SafeInt<T,E>& operator-= (SafeInt<T,E> rhs) |
| -= | template<typename U>  SafeInt<T,E>& operator-= (U rhs) |
| -= | template<typename U>  SafeInt<T,E>& operator-= (SafeInt<U, E> rhs) |

### [Logical Operators](javascript:void(0))

|  |  |
| --- | --- |
| **Name** | **Syntax** |
| ! | bool operator !() const throw() |
| ~ | SafeInt<T,E> operator~ () const throw() |
| << | template<typename U>  SafeInt<T,E> operator<< (U bits) const throw() |
| << | template<typename U>  SafeInt<T,E> operator<< (SafeInt<U, E> bits) const throw() |
| <<= | template<typename U>  SafeInt<T,E>& operator<<= (U bits) throw() |
| <<= | template<typename U>  SafeInt<T,E>& operator<<= (SafeInt<U, E> bits) throw() |
| >> | template<typename U>  SafeInt<T,E> operator>> (U bits) const throw() |
| >> | template<typename U>  SafeInt<T,E> operator>> (SafeInt<U, E> bits) const throw() |
| >>= | template<typename U>  SafeInt<T,E>& operator>>= (U bits) throw() |
| >>= | template<typename U>  SafeInt<T,E>& operator>>= (SafeInt<U, E> bits) throw() |
| & | SafeInt<T,E> operator& (SafeInt<T,E> rhs) const throw() |
| & | template<typename U>  SafeInt<T,E> operator& (U rhs) const throw() |
| &= | SafeInt<T,E>& operator&= (SafeInt<T,E> rhs) throw() |
| &= | template<typename U>  SafeInt<T,E>& operator&= (U rhs) throw() |
| &= | template<typename U>  SafeInt<T,E>& operator&= (SafeInt<U, E> rhs) throw() |
| ^ | SafeInt<T,E> operator^ (SafeInt<T,E> rhs) const throw() |
| ^ | template<typename U>  SafeInt<T,E> operator^ (U rhs) const throw() |
| ^= | SafeInt<T,E>& operator^= (SafeInt<T,E> rhs) throw() |
| ^= | template<typename U>  SafeInt<T,E>& operator^= (U rhs) throw() |
| ^= | template<typename U>  SafeInt<T,E>& operator^= (SafeInt<U, E> rhs) throw() |
| | | SafeInt<T,E> operator| (SafeInt<T,E> rhs) const throw() |
| | | template<typename U>  SafeInt<T,E> operator| (U rhs) const throw() |
| |= | SafeInt<T,E>& operator|= (SafeInt<T,E> rhs) throw() |
| |= | template<typename U>  SafeInt<T,E>& operator|= (U rhs) throw() |
| |= | template<typename U>  SafeInt<T,E>& operator|= (SafeInt<U, E> rhs) throw() |

## [Remarks](javascript:void(0))

The **SafeInt** class protects against integer overflow in mathematical operations. For example, consider adding two 8-bit integers: one has a value of 200 and the second has a value of 100. The correct mathematical operation would be 200 + 100 = 300. However, because of the 8-bit integer limit, the upper bit will be lost and the compiler will return 44 (300 - 28) as the result. Any operation that depends on this mathematical equation will generate unexpected behavior.

The **SafeInt** class checks whether an arithmetic overflow occurs or whether the code tries to divide by zero. In both cases, the class calls the error handler to warn the program of the potential problem.

This class also lets you compare two different types of integers as long as they are **SafeInt** objects. Typically, when you perform a comparison, you must first convert the numbers to be the same type. Casting one number to another type often requires checks to make sure that there is no loss of data.

The Operators table in this topic lists the mathematical and comparison operators supported by the **SafeInt** class. Most mathematical operators return a **SafeInt** object of type T.

Comparison operations between a **SafeInt** and an integral type can be performed in either direction. For example, both SafeInt<int>(x) < yand y > SafeInt<int>(x) are valid and will return the same result.

Many binary operators do not support using two different **SafeInt** types. One example of this is the **&** operator. SafeInt<T, E> & int is supported, but SafeInt<T, E> & SafeInt<U, E> is not. In the latter example, the compiler does not know what type of parameter to return. One solution to this problem is to cast the second parameter back to the base type. By using the same parameters, this can be done withSafeInt<T, E> & (U)SafeInt<U, E>.

|  |
| --- |
| **System_CAPS_noteNote** |
| For any bitwise operations, the two different parameters should be the same size. If the sizes differ, the compiler will throw an [ASSERT (MFC)](https://msdn.microsoft.com/en-us/library/ew16s3zc.aspx)exception. The results of this operation cannot be guaranteed to be accurate. To resolve this issue, cast the smaller parameter until it is the same size as the larger parameter. |

For the shift operators, shifting more bits than exist for the template type will throw an ASSERT exception. This will have no effect in release mode. Mixing two types of SafeInt parameters is possible for the shift operators because the return type is the same as the original type. The number on the right side of the operator only indicates the number of bits to shift.

When you perform a logical comparison with a SafeInt object, the comparison is strictly arithmetic. For example, consider these expressions:

* SafeInt<uint>((uint)~0) > -1
* ((uint)~0) > -1

The first statement resolves to **true**, but the second statement resolves to **false**. The bitwise negation of 0 is 0xFFFFFFFF. In the second statement, the default comparison operator compares 0xFFFFFFFF to 0xFFFFFFFF and considers them to be equal. The comparison operator for the **SafeInt**class realizes that the second parameter is negative whereas the first parameter is unsigned. Therefore, although the bit representation is identical, the **SafeInt** logical operator realizes that the unsigned integer is larger than -1.

Be careful when you use the **SafeInt** class together with the **?:** ternary operator. Consider the following line of code.

Int x = flag ? SafeInt<unsigned int>(y) : -1;

The compiler converts it to this:

Int x = flag ? SafeInt<unsigned int>(y) : SafeInt<unsigned int>(-1);

If flag is **false**, the compiler throws an exception instead of assigning the value of -1 to x. Therefore, to avoid this behavior, the correct code to use is the following line.

Int x = flag ? (int) SafeInt<unsigned int>(y) : -1;

T and U can be assigned a Boolean type, character type, or integer type. The integer types can be signed or unsigned and any size from 8 bits to 64 bits.

|  |
| --- |
| **System_CAPS_noteNote** |
| Although the **SafeInt** class accepts any kind of integer, it performs more efficiently with unsigned types. |

E is the error handling mechanism that **SafeInt** uses. Two error handling mechanisms are provided with the SafeInt library. The default policy is**SafeIntErrorPolicy\_SafeIntException**, which throws a [SafeIntException Class](https://msdn.microsoft.com/en-us/library/dd570024.aspx) exception when an error occurs. The other policy is**SafeIntErrorPolicy\_InvalidParameter**, which stops the program if an error occurs.

There are two options to customize the error policy. The first option is to set the parameter E when you create a **SafeInt**. Use this option when you want to change the error handling policy for just one **SafeInt**. The other option is to define **\_SAFEINT\_DEFAULT\_ERROR\_POLICY** to be your customized error-handling class before you include the **SafeInt** library. Use this option when you want to change the default error handling policy for all instances of the **SafeInt** class in your code.

|  |
| --- |
| **System_CAPS_noteNote** |
| A customized class that handles errors from the SafeInt library should not return control to the code that called the error handler. After the error handler is called, the result of the **SafeInt** operation cannot be trusted. |

## Requirements

**Header:** safeint.h

**Namespace:** msl::utilities

SafeInt Functions

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

The SafeInt library provides several functions that you can use without creating an instance of the [SafeInt Class](https://msdn.microsoft.com/en-us/library/dd570021.aspx). If you want to protect a single mathematical operation from integer overflow, you can use these functions. If you want to protect multiple mathematical operations, you should create **SafeInt** objects. It is more efficient to create **SafeInt** objects than to use these functions multiple times.

These functions enable you to compare or perform mathematical operations on two different types of parameters without having to convert them to the same type first.

Each of these functions has two template types: *T* and *U*. Each of these types can be a Boolean, character, or integral type. Integral types can be signed or unsigned and any size from 8 bits to 64 bits.

In This Section

|  |  |
| --- | --- |
| **Function** | **Description** |
| [SafeAdd](https://msdn.microsoft.com/en-us/library/dd578442.aspx) | Adds two numbers and protects against overflow. |
| [SafeCast](https://msdn.microsoft.com/en-us/library/dd578445.aspx) | Casts one type of parameter to another type. |
| [SafeDivide](https://msdn.microsoft.com/en-us/library/dd578496.aspx) | Divides two numbers and protects against dividing by zero. |
| [SafeEquals](https://msdn.microsoft.com/en-us/library/dd578451.aspx), [SafeGreaterThan](https://msdn.microsoft.com/en-us/library/dd578441.aspx), [SafeGreaterThanEquals](https://msdn.microsoft.com/en-us/library/dd578443.aspx),[SafeLessThan](https://msdn.microsoft.com/en-us/library/dd578492.aspx), [SafeLessThanEquals](https://msdn.microsoft.com/en-us/library/dd578501.aspx), [SafeNotEquals](https://msdn.microsoft.com/en-us/library/dd578439.aspx) | Compares two numbers. These functions enable you to compare two different types of numbers without changing their types. |
| [SafeModulus](https://msdn.microsoft.com/en-us/library/dd578495.aspx) | Performs the modulus operation on two numbers. |
| [SafeMultiply](https://msdn.microsoft.com/en-us/library/dd578491.aspx) | Multiplies two numbers together and protects against overflow. |
| [SafeSubtract](https://msdn.microsoft.com/en-us/library/dd578500.aspx) | Subtracts two numbers and protects against overflow. |

SafeInt Functions

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

The SafeInt library provides several functions that you can use without creating an instance of the [SafeInt Class](https://msdn.microsoft.com/en-us/library/dd570021.aspx). If you want to protect a single mathematical operation from integer overflow, you can use these functions. If you want to protect multiple mathematical operations, you should create **SafeInt** objects. It is more efficient to create **SafeInt** objects than to use these functions multiple times.

These functions enable you to compare or perform mathematical operations on two different types of parameters without having to convert them to the same type first.

Each of these functions has two template types: *T* and *U*. Each of these types can be a Boolean, character, or integral type. Integral types can be signed or unsigned and any size from 8 bits to 64 bits.

In This Section

|  |  |
| --- | --- |
| **Function** | **Description** |
| [SafeAdd](https://msdn.microsoft.com/en-us/library/dd578442.aspx) | Adds two numbers and protects against overflow. |
| [SafeCast](https://msdn.microsoft.com/en-us/library/dd578445.aspx) | Casts one type of parameter to another type. |
| [SafeDivide](https://msdn.microsoft.com/en-us/library/dd578496.aspx) | Divides two numbers and protects against dividing by zero. |
| [SafeEquals](https://msdn.microsoft.com/en-us/library/dd578451.aspx), [SafeGreaterThan](https://msdn.microsoft.com/en-us/library/dd578441.aspx), [SafeGreaterThanEquals](https://msdn.microsoft.com/en-us/library/dd578443.aspx),[SafeLessThan](https://msdn.microsoft.com/en-us/library/dd578492.aspx), [SafeLessThanEquals](https://msdn.microsoft.com/en-us/library/dd578501.aspx), [SafeNotEquals](https://msdn.microsoft.com/en-us/library/dd578439.aspx) | Compares two numbers. These functions enable you to compare two different types of numbers without changing their types. |
| [SafeModulus](https://msdn.microsoft.com/en-us/library/dd578495.aspx) | Performs the modulus operation on two numbers. |
| [SafeMultiply](https://msdn.microsoft.com/en-us/library/dd578491.aspx) | Multiplies two numbers together and protects against overflow. |
| [SafeSubtract](https://msdn.microsoft.com/en-us/library/dd578500.aspx) | Subtracts two numbers and protects against overflow. |

Related Sections

|  |  |
| --- | --- |
| **Section** | **Description** |
| [SafeInt Class](https://msdn.microsoft.com/en-us/library/dd570021.aspx) | The **SafeInt** class. |
| [SafeIntException Class](https://msdn.microsoft.com/en-us/library/dd570024.aspx) | The exception class specific to the SafeInt library. |

# SafeIntException Class

**Visual Studio 2015**

[Other Versions](javascript:;)

https://msdn.microsoft.com/Areas/Epx/Content/Images/ImageSprite.png?v=636042219474060904

The **SafeInt** class uses **SafeIntException** to identify why a mathematical operation cannot be completed.

## [Syntax](javascript:void(0))

class SafeIntException;

## [Members](javascript:void(0))

### [Public Constructors](javascript:void(0))

[SafeIntException::SafeIntException](https://msdn.microsoft.com/en-us/library/dd572131.aspx)

Creates a **SafeIntException** object.

## [Remarks](javascript:void(0))

The [SafeInt Class](https://msdn.microsoft.com/en-us/library/dd570021.aspx) is the only class that uses the **SafeIntException** class.

## [Inheritance Hierarchy](javascript:void(0))

[SafeIntException Class](https://msdn.microsoft.com/en-us/library/dd570024.aspx)

## Requirements

**Header:** safeint.h

**Namespace:** msl::utilities