# Getting started

## System requirements

* Windows 7 or later x86 or x64
* Development tools
  + Visual Studio 2012
    - .NET framework 4.5
    - MSBuild 4
    - Link.exe for Windows target
* Any supported SSH client for Linux and FreeBSD targets
  + [Bitvise Tunnelier](http://www.bitvise.com/tunnelier-download)
  + [TeraTerm](http://en.sourceforge.jp/projects/ttssh2)
* Virtualization software (optional, allows run compiler and generated executable on same physical machine)
  + [VMWare Workstation](https://www.vmware.com/products/workstation/)
  + [VMWare Player](https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0)

## Installation

* Run provided installer on Windows with Visual Studio 2012
* Install SSH client + terminal emulator
  + <http://en.sourceforge.jp/projects/ttssh2>, set PATH environment to be able to run ttermpro.exe from command line
  + <http://www.bitvise.com/tunnelier-download>
* Generate RSA key pair, passwords not supported by internal SSH client
  + Generate key pair on client, upload public key to appropriate place on SSH server ($HOME/.ssh/authorized\_keys)
* Ensure you can access remote machine with SSH client

### Windows environment

Use %PROGRAMFILES% instead of %PROGRAMFILES(x86)% on 32-bit Windows

* Default compiler directory “%PROGRAMFILES(x86)%\MSBuild\Bamelg\”
* Default IDE integration directory – “%PROGRAMFILES(x86)%\Microsoft Visual Studio 11.0\Common7\IDE\Extensions"
* Default ICE reporting directory – “%LOCALAPPDATA%\Bamelg\ICE Reports\”

MSBuild user-defined property defaults “%LOCALAPPDATA%\Bamelg\Defaults.build”

### Linux build bot

Install required packages

|  |
| --- |
| yum instal gcc valgrind libuuid-devel libcap-devel libaio-devel lksctp-tools-devel jemalloc-devel gperftools-devel google-perftools perf |

Optional packages

|  |
| --- |
| yum instal giflib-devel libjpeg-turbo-devel pango-devel cairo-devel atk-devel gtk2-devel |

## Build with MSBuild

### Build options

* Default compilation mode is cross-compilation which means you build executable module for non-host processor and operating system
  + After succeeded compilation object file appears in output directory
* For Linux and FreeBSD targets linking is performed on remote machine with gcc installed (internally it uses ld with bootstap object files with C runtime)
  + link.exe used for Windows targets
* MSBuild used for build process. Most notable parameters:
* NativeTargetPlatform – target platform for executable. Can be **FreeBSD**, **Linux**, **Windows**, **MacOS**.
* NativeTargetPlatformVersion – target platform version. OSes of one family can have differences in system-level API which can affect on how to properly write programs for that platform (like poll, epoll). NativeTargetPlatformand NativeTargetPlatformVersionhelps to select proper platfrom-specific folders, for example “C:\Program Files (x86)\MSBuild\Bamelg\Modules\BCL\Platform\Linux\2.4 - X86\_32\Platform”
* ModulesFolder – location of all available modules
* ModularBuild – build modules separately to improve compile times on small changes
* GenerateAssemblySource – generate assembly “$(AssemblyName).s” file
* GenerateObjectFile – generate object “$(AssemblyName).o” file
* RootNamespace – default namespace name for some internally generated types, like EmbeddedResourcesType
* CStringCodepage – default code page for AnsiString and CStringstored in executable
* EmbeddedResourcesType – name of type to hold all embedded resources as fixed arrays of bytes
* AssemblyName – name of output executable without extension, which is platform-dependent
* TargetCpu – target CPU architecture
  + X86\_32 – x86 with 32-bit pointers
  + X86\_64 – x64 with 64-bit pointers
  + PowerPC\_32 – x64 with 32-bit pointers
  + PowerPC\_64 – x64 with 64-bit pointers
  + ARM\_32 – ARM with 32-bit pointers
  + ARM\_64 – ARM with 64-bit pointers
  + MIPS\_32 – MIPS with 32-bit pointers
  + MIPS\_64 – MIPS with 64-bit pointers
* TargetCpuName – a specific chip in the target CPU architecture to generate code for. Used by backend to determine which extended instruction sets will be used during lowering code to assembly commands.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| amdfam10 | athlon | athlon-4 | athlon-fx | athlon-mp | athlon-tbird | athlon-xp | athlon64 |
| athlon64-sse3 | atom | bdver1 | bdver2 | btver1 | c3 | c3-2 | core-avx-i |
| core-avx2 | core2 | corei7 | corei7-avx | generic | i386 | i486 | i586 |
| i686 | k6 | k6-2 | k6-3 | k8 | k8-sse3 | Nehalem | nocona |
| opteron | opteron-sse3 | penryn | pentium | pentium-m | pentium-mmx | pentium2 | pentium3 |
| pentium3m | pentium4 | pentium4m | pentiumpro | prescott | westmere | winchip-c6 | winchip2 |
| x86-64 | yonah | 440 | 450 | 601 | 602 | 603 | 603e |
| 603ev | 604 | 604e | 620 | 7400 | 7450 | 750 | 970 |
| a2 | g3 | g4 | g4+ | g5 | ppc | ppc64 | arm1020e |
| arm1020t | arm1022e | arm10e | arm10tdmi | arm1136j-s | arm1136jf-s | arm1156t2-s | arm1156t2f-s |
| arm1176jz-s | arm1176jzf-s | arm710t | arm720t | arm7tdmi | arm7tdmi-s | arm8 | arm810 |
| arm9 | arm920 | arm920t | arm922t | arm926ej-s | arm940t | arm946e-s | arm966e-s |
| arm968e-s | arm9e | arm9tdmi | cortex-a8 | cortex-a9 | cortex-a9-mp | cortex-m0 | cortex-m3 |
| cortex-m4 | ep9312 | iwmmxt | mpcore | mpcorenovfp | strongarm | strongarm110 | strongarm1100 |
| strongarm1110 | xscale | mips32 | mips32r2 | mips64 | mips64r2 |  |  |

* StartProgram – program running after build have been succeeded
* CmdArgs – command line arguments passed to $(StartProgram)executable
* DebugSymbols – generate DWARF symbols (for gdb/valgrind/backtrace etc.) for executable file
* OptimizationLevel– optimization level. Optimization level used to determine which optimization passes will be executed on program intermediate representation while lowering code to assembly commands.
  + 0 optimal for Debug builds (no optimization)
  + 1 optimal for Release builds
  + 3,4 optimal for Distribution builds
* DefineConstants – compilation constants used by methods marked with ConditionalAttribute
* RunUnitTests – run unit tests before transfer control to application entry point
* ImplicitTLS – modern targets support optimized TLS access to variables instead of TlsGetValue or pthread\_getspecific. Some old operating systems (like Linux 2.4.\*) has no implicit TLS support in executable loader leading to TLS variables sharing between threads.
* StripCallerContextExpressions – performs stripping of method caller info to reduce executable size.
  + Assert.IsTrue(), Assert.AreEqual() and other tracing methods with “CallerContext& callercontext = nullref” parameters will be unable to show correct source location
* Parameters for remote link and run. Remote machine accessed via SSH.
  + RemoteDebugMachine – address of remote machine to login, link and run executable via SSH
  + RemoteExecutablePath – remote app path which passed to linker as destination file
  + RemoteOutputPath – path to copy file from $(RemoteExecutablePath)
  + SshKeyFile – path to RSA private key. [Create](https://help.github.com/articles/generating-ssh-keys/) [key](http://www.bitvise.com/public-keys-in-ssh) if necessary. First line of file should look like “-----BEGIN RSA PRIVATE KEY-----”
  + SshUser – remote user name
  + SshPort – TCP port for SSH. Default is 22.
  + SshCheckServerFingerprint – when true performs check server host key with cached entries in HKCU\Software\SimonTatham\PuTTY\SshHostKeys
  + SshShellCommand – command to run executable
  + SshPrefferedKeyExchangeAlgorithm – comma separated list of following values:
    - diffie-hellman-group-exchange-sha256
    - diffie-hellman-group-exchange-sha1
    - diffie-hellman-group1-sha1
    - diffie-hellman-group14-sha1
  + SshPrefferedMacAlgorithm – comma separated list of following values:
    - aes128-cbc
    - aes192-cbc
    - aes256-cbc
    - aes128-ctr
    - aes192-ctr
    - aes256-ctr
    - arcfour128
    - arcfour256
    - arcfour
    - none
  + SshPrefferedEncryptionAlgorithm – comma separated list of following values:
    - hmac-sha1-etm@openssh.com
    - hmac-sha1
    - hmac-sha2-256-etm@openssh.com
    - hmac-sha2-256
    - hmac-sha2-512-etm@openssh.com
    - hmac-sha2-512,hmac-sha1-96-etm@openssh.com
    - hmac-sha1-96
    - hmac-md5-etm@openssh.com
    - hmac-md5
    - hmac-ripemd160-etm@openssh.com
    - hmac-ripemd160
    - hmac-ripemd160@openssh.com
    - hmac-md5-96-etm@openssh.com
    - hmac-md5-96
    - none

### Global overrides

Default values for most of build properties can be set using “%LOCALAPPDATA%\Bamelg\Defaults.build” file. For example

|  |
| --- |
| <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  <PropertyGroup>  <PlatformToolset>v120</PlatformToolset>  <SshKeyFile>c:\connect.rsa.pvk</SshKeyFile>  <SshUser>root</SshUser>  <SshPrefferedEncryptionAlgorithm>none,arcfour</SshPrefferedEncryptionAlgorithm>  </PropertyGroup>  </Project> |

### Command line

|  |
| --- |
| :: Build with 32-bit MSBuild and 32-bit Bamelg compiler  "%windir%\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe" /property:VisualStudioVersion=11.0 /verbosity:d /t:Build App.bmproj  :: ReBuild with 64-bit MSBuild and 64-bit Bamelg compiler  "%windir%\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe" /property:VisualStudioVersion=11.0 /verbosity:d /t:ReBuild App.bmproj  :: Reformat with 32-bit 12 MSBuild  "c:\Program Files (x86)\MSBuild\12.0\Bin\MSBuild.exe"" /property:VisualStudioVersion=11.0 /verbosity:d /t:Reformat App.bmproj  :: Build with 64-bit 12 MSBuild  "c:\Program Files (x86)\MSBuild\12.0\Bin\amd64\MSBuild.exe" /property:VisualStudioVersion=11.0 /verbosity:d /t:Build App.bmproj |

## Performance

See <http://bamelg.com/performance> for details

# Notes

## Predefined project configurations

* Debug
  + No optimizations , slow code generation speed
  + Defined constants **TRACE**, **DEBUG**
  + No dead code stripped, no metadata stripped
  + DWARF debug information embedded
  + Large executable
* Release
  + Lightweight optimizations, fast code generation speed
  + Some code and metadata stripped
  + Medium-sized executable
    - [UPX](http://upx.sourceforge.net/) can be used to compress executable file
* Distribution
  + Aggressive optimizations, slow code generation speed
    - You can additionally set StripCallerContextExpressions to true
  + Unreachable code and metadata stripped
  + Executable size varies. In most cases resulting executable is large than Release build due to inlining.
    - [UPX](http://upx.sourceforge.net/) can be used to compress executable file on most platforms

In addition, you can easily add custom build types by manually edit project file (see MSBuild reference).

## Type system

* Reference types
  + Derived from RuntimeObjectBase
    - class types

|  |
| --- |
| class Test { } |

* + - delegate types

|  |
| --- |
| public delegate void Action(); |

* + - dynamic array types

|  |
| --- |
| byte[] buffer = new[55] byte; |

* Value types
  + Pointers

|  |
| --- |
| protected void\* \_buffer; |

* + Functors

|  |
| --- |
| public functor void Action(); |

* + Fixed arrays

|  |
| --- |
| protected byte[128] \_buffer; |

* + struct types

|  |
| --- |
| struct Test { } |

* + Built-in numeric types
    - byte, sbyte
    - short, ushort
      * short\_littleendian, ushort\_ littleendian
      * short\_bigendian, ushort\_bigendian
    - int, uint
      * int\_littleendian, uint\_ littleendian
      * int\_bigendian, uint\_bigendian
    - long, ulong
      * long\_littleendian, ulong\_ littleendian
      * long\_bigendian, ulong\_bigendian
    - float, double
      * float\_littleendian, double\_ littleendian
      * float\_bigendian, double\_bigendian

## Exceptions

Exceptions in Bamelg:

* Are something like condition system in SmallTalk and Common Lisp, not like C++/C#/Java exceptions
* Did not unwind stack
  + Execution flow continues after exception handler has being invoked
  + Program flow able to take custom action (continue,restart, or modify fields and exit from current method) after exception handler has being invoked
* Designed to raise/catch/re-raise recoverable errors/conditions
  + No NullReferenceException – it’s program error, and source code must be modified accordingly
  + No StackOverflowException – it’s program/configuration error
    - Specify more pages for fibers
    - Increase stacks size for threads
    - Flatten recursive calls or limit recursion depth
* Platform/runtime-independent
* Designed to be fast and thread/fiber-safe

## Debugging

* Use optimization level 0 for debug builds
  + Methods, properties will not be inlined
  + Stack frame pointer will not omitted
* Use tools from valgrind suite to find memory errors, thread synchronization errors etc.
  + Use Valgrind.IsAttached to detect valgrind presence
* Use Assert.IsTrue() and Assert.AreEqual() to verify conditions in both DEBUG and RELEASE build modes
  + Use Debug.Assert() to verify conditions when DEBUG is defined ( in DEBUG mode )
  + Use Trace .Assert() to verify conditions when TRACE is defined
* Use gdb
  + Use Debugger.Break() to programmatically insert breakpoint
* Use Console.WriteLine(), or Trace.Write(), or custom tracing ( for example see EventLoop.bmlg ) to trace application flow

## Optimization

* Prior to any low-level optimization find and read good books
  + Check Agner Fog’s [site](http://www.agner.org/optimize/)
  + Read Intel and AMD optimization reference manuals
* Prior to any high-level optimization find and read good books too
* Do not implement premature optimizations but design app architecture to easily apply these optimizations later. High-level optimizations in most cases have greater effect than micro-optimizations.
* Increase temporal locality. Align data properly. Minimize cache pollution. Minimize page faults. Minimize working set. Minimize context switches.
* Avoid mixing of signed/unsigned operands for comparison operators ( <, >, <=, >=, ==, != ). Encoding of such operations takes more assembly instructions.
* Use right collection classes. Find proper balance between memory usage and lookup/modification performance.
  + Avoid resizing of collections (List<T>, Dictionary<K,V>, etc.). Use **Capacity** property or appropriate constructor to influence the initial buffer sizes. This optional **Capacity** property adjusts the amount of memory allocated. If **Capacity** properly predicted it avoids many allocations when adding elements that were not anticipated.
* If you are doing significant appending or replacing of strings, the StringBuilder type can improve performance
* Avoid heavy ToString() usage. implement IFormattable and use ToString( StringBuilder builder, string format ) instead
* Use proper language constructs:
  + Invocation of static methods is fastest
  + Invocation of instance methods is sligthly slower (hidden parameter passed for this)
  + Invocation of virtual methods is slower than instance methods (indirect call)
    - Use sealed keyword to assist compiler with devirtualization
  + Invocation of interface methods is slower than virtual methods (indirect call)
  + Invocation of delegates or events is slower than interface method call.
* Access to static fields is fastest. Access to instance fields is slower. Access to property depends of its body and inlining behavior.
* LLVM backend is good in inlining optimizations but it can reject big method. Use ForceInlineAttributeto force inlining.
* Use BackendHintReadOnlyAttribute and BackendHintReadNoneAttribute to allow aggressive CSE/DCE optimizations
* Use threads for computation-intensive tasks. Use fibers for IO-intensive tasks.
  + IO-intensive tasks can be prototyped with multithreading approach and then rewritten with fibers
  + Avoid thread blocking during fiber execution
* Use struct and class appropriately. If you are not sure then use profiler to make right decision.
* Use TLS properly. Minimize TLS field access count.
  + TLS field initializers helps to eliminate checking of variable for null value to initialize it at cost thread creation slow down
  + Use implicit TLS (see ThreadStaticAttribute) instead of explicit TLS due to performance and portability.
* Write your programs in event-driven way. Remove active delays. Minimize UNIX signal usage. In most cases
  + OS responsiveness will benefit of it
  + Mobile users will save more battery energy
* Minimize usage of dynamically linked methods
  + External methods did not participate in Inter-Procedural Optimizations (no inlining, no specialization, no splitting critical edges etc.)
  + External methods slow down loading of app
  + External methods always called indirectly
* Minimize serialization and synchronization instruction usage (like **atomic** increments/decrements operations used by RuntimeObjectBase.AddReference()/ReleaseReference())
  + Use Memory*.MultithreadedEnvironment* to change default reference counting mode (atomic/not atomic) for newly created objects
* Minimize OS synchronization primitives (mutex, semaphore, rwlock etc.) usage especially process-shared ones
  + Use lightweight primitives when possible (for example CRITICAL\_SECTION instead of Mutex/Semaphore on Win32 )
  + Use non-blocking synchronization. See [CAS](http://en.wikipedia.org/wiki/Compare-and-swap)
  + Try to separate synchronization from logic. Write most of your classes in MT-unsafe way and write multithread access adapter classes later.
* Assign proper TargetCpuName to utilize advanced processor instruction sets.
* Assign proper OptimizationLevel for release build
  + All valid code should work for OptimizationLevel=4. If something not work probably…
    - Your code and/or assumptions may be wrong. For example :
      * Variables in stack frame aggressively packed by backend and memory pointed by dangling pointers to dead variables will changed more frequently
      * Aggressive inlining and stack frame pointer omission affects on call tree which can give “unexpected” results for StackTrace.Print()
      * Reordering and packing of local variables can expose previously hidden cross-thread synchronization bugs
    - Something has have been miscompiled by compiler backend

## Fiber pitfalls

* Thread blocking
  + Fibers are useless without proper library support
    - Need for “event loop” to manage OS handle notifications
    - Blocking IO should be simulated by using non-blocking sockets or ASIO configured properly to deliver completion notifications to fiber manager
    - Blocking syscalls should be executed in ThreadPool
  + Do not use thread synchronization primitives. Use special fiber synchronization primitives like FiberMutex.
  + Do not use blocking system calls like Sleep(), nanosleep().
  + Do not use blocking IO
    - Page faults will block thread so be careful with swappable memory-mapped files.
* Thread-local global variables
  + Thread-local global variables shared between fibers so any fiber able to change **errno/LastError**. Fiber should manually save all required global state. For example:
    - errnosince it can be set in another fiber (in syscall wrapper)
    - GetLastResult() since SetLastResult() may be called in another fiber
    - GetCurrectDirectory() since SetCurrentDirectory() may be called in another fiber
      * CurrentDirectory is thread local on \*nixes, thread-shared on Windows

## Design your application properly

* Always write your programs in event-driven way.
* Do not abort or cancel operating system threads.
* Avoid suspending and resuming OS threads. In most cases, synchronization primitives provide much flexible thread blocking.
  + Primary use of suspend/resume is to perform execution context reading and modification

## Bad code

### Dangling pointers

|  |
| --- |
| int& Foo() {  int variable = 123;  // some code here  return variable; // return reference to dead variable  } |

|  |
| --- |
| int\* Foo() {  int variable = 123;  // some code here  return &variable; // return pointer to dead variable  } |

|  |
| --- |
| IFormattable Foo() {  int variable = 123;  // some code here  return variable; // return newly created interface to variable which will dead after end of scope  } |

|  |
| --- |
| IFormattable Foo() {  int variable = 123;  var result = ( IFormattable ) variable;  // some code here  return result; // ‘result’ only valid inside ‘Foo’  } |

These dangling pointers:

* Not **null**
* May be overwritten by ISR or by functions [called later] which stack frame intersects with old variable location
* May give access violation after thread or fiber terminated (SEGFAULT, memory cannot be read, etc.) since stack memory region is freed

|  |
| --- |
| C-like way to get dangling pointer with malloc/free: |
| var guid = ( Guid\* ) Memory.Allocate( typeof( Guid ).InstanceSize );  // ...  Memory.Deallocate( guid ); // 'guid' now becomes a dangling pointer. Need to write ‘guid=null;’  guid->ToString(); // use after free, program continues |
| Use of heapalloc and heapfree can fix problem: |
| var guid = heapalloc Guid;  // ...  heapfree guid; // 'guid' now null  guid->ToString(); // program aborts with AV since ‘guid’ == null after ‘heapfree’ |

See [article](http://en.wikipedia.org/wiki/Dangling_pointer) on Wikipedia for detailed explanation.

### Incorrect functor initialization with non-default calling convention

|  |
| --- |
| void\* methodRawPtr1 = bitcast<void\*>( new Functors.Action( SomeMethod ) );  void\* methodRawPtr2 = bitcast<void\*>( memberinfo( SomeMethod ).Address ); |

*SomeMethod* may have non-default calling convention and in first case we will get method address with calling convention of functor *Functors.Action*

In second case will get raw pointer to method code with CC of method *SomeMethod*. So *methodPtr1* may be not equal to *methodPtr2*

### Bad naming

* Name of types, properties and fields must be retained for platform bindings even naming style is bad.
* Code must be readable even without reading documentation
* Do not use abbreviations which is not well-known

|  |
| --- |
| void Foo() {  int LocalVariable; // localVariable  } |

|  |
| --- |
| void foo() { // Foo  } |

|  |
| --- |
| class test { // Test  } |

|  |
| --- |
| static int someProp { get; set; } // SomeProperty |

|  |
| --- |
| private static int field; // \_field |

|  |
| --- |
| public static int field; // Field |

|  |
| --- |
| public static int some\_old\_ported\_code; // SomeOldPortedCode |

|  |
| --- |
| public extern static int some\_c\_interop\_code; // do not change |

# Bamelg keywords

## abstract

Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes.

* An abstract class cannot be instantiated.
* An abstract class may contain abstract members.
* A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

|  |
| --- |
| abstract class A1 {  abstract void M1();  abstract int M2 { get; set; }  abstract int M3[int index] { get; set; }  abstract int this[int index] { get; set; }  abstract event Delegates.Action M5;  }  class C1 : A1 {  override void M1() { }  override int M2 { get { return Data[0]; } set { Data[0] = value; } }  override int M3[int index] { get { return Data[index]; } set { Data[index] = value; } }  override int this[int index] { get { return Data[index]; } set { Data[index] = value; } }  override event Delegates.Action M5;  int[] Data = new[10] int;  } |

## as

The as keyword used to perform conversions between compatible reference types.

|  |
| --- |
| using System;  namespace Test {  class Test {  static void Foo( RuntimeObjectBase someObject ) {  var utf16 = someObject as string;  if( utf16 != null ) {  Console.WriteLine( "1. someObject is 'Utf16String': {0}", utf16 );  return;  }  var utf32 = someObject as Utf32String;  if( utf32 != null ) {  Console.WriteLine( "2. someObject is 'Utf32String': {0}", utf32 );  return;  }  Console.WriteLine( "3. someObject is '{0}': {1}", someObject.GetType(), someObject );  }  [EntryPoint]  public static void Main() {  Foo( ( AnsiString ) "a" ); // 3. someObject is 'System.AnsiString': a  Foo( ( Utf8String ) "b" ); // 3. someObject is 'System.Utf8String': b  Foo( ( Utf16String ) "c" ); // 1. someObject is 'Utf16String': c  Foo( ( Utf32String ) "d" ); // 2. someObject is 'Utf32String': d  }  }  } |

## asm

The **asm** keyword used to define assembly block for methods, properties, indexed properties, and operators.

|  |
| --- |
| public static void Break() asm {  X86\_32 {  nop  icebp  ret  }  X86\_64 {  nop  icebp  ret  }  default {  Assert.Unreachable();  }  } |

|  |
| --- |
| public static float operator |( Quaternion& left, Quaternion& right ) asm {  X86\_32 {  feature( SSE1 ) {  if( thismethod.CallingConvention == CallingConvention.FastCallX86 ) {  // ecx - left  // edx - right  }  else {  mov ecx, [esp + 4] // left  mov edx, [esp + 8] // right  }  movaps xmm0, [ecx]  mulps xmm0, [edx]  xorps xmm1, xmm1  movhlps xmm1, xmm0  addps xmm0, xmm1  movaps xmm2, xmm0  shufps xmm2, xmm2, 00000001  addss xmm0, xmm2  mov eax, [esp]  movss [esp], xmm0  fld dword ptr [esp]  pop ecx  jmp eax  }  feature( AMD3DNow ) {  if( thismethod.CallingConvention == CallingConvention.FastCallX86 ) {  // ecx - left  // edx - right  }  else {  mov ecx, [esp + 4] // left  mov edx, [esp + 8] // right  }  movq mm0, [ecx]  movq mm1, [ecx + 8]  pfmul mm0, [edx]  pfmul mm1, [edx + 8]  pfadd mm0, mm1  mov eax, [esp]  pfacc mm0, mm0  movd [esp], mm0  femms  fld dword ptr [esp]  pop ecx  jmp eax  }  default {  if( thismethod.CallingConvention == CallingConvention.FastCallX86 ) {  // ecx - left  // edx - right  }  else {  mov ecx, [esp + 4] // left  mov edx, [esp + 8] // right  }  fld dword ptr [ecx]  fmul dword ptr [edx]  fld dword ptr [ecx + 4]  fmul dword ptr [edx + 4]  fld dword ptr [ecx + 8]  fmul dword ptr [edx + 8]  fld dword ptr [ecx + 12]  fmul dword ptr [edx + 12]  faddp st2, st0  faddp st2, st0  faddp st1, st0  retn  }  }  default {  return left.X \* right.X + left.Y \* right.Y + left.Z \* right.Z + left.W \* right.W;  }  } |

## atomic

The atomic keyword used to modify default behavior of loading and storing. Result of operation is a value prior to modification.

Syntax: atomic( expression [, AtomicOrdering = SequentiallyConsistent] [, SyncronizationScope = CrossThread] )

|  |
| --- |
| using System;  namespace Test {  class Test {  struct S1 { int SomeField; }  [EntryPoint]  private static void Main() {  int v1 = 0, v2 = 0;  var y = new S1 { SomeField = 6 };  int\* ptr = &y.SomeField;  int\* z = null;  Assert.IsTrue( atomic( z = ptr ) == null ); // atomic write, old value of 'z' is result. (compare not atomic)  Assert.IsTrue( atomic( z = ptr ) == ptr );  Assert.IsTrue( atomic( ++z ) == ptr ); // atomic increment, returns old value of 'z'  Assert.IsTrue( z == ptr + 1 );  Assert.IsTrue( atomic( --z ) == ptr + 1 ); // atomic decrement, returns old value of 'z'  Assert.IsTrue( z == ptr );  Assert.IsTrue( atomic( z += 10 ) == ptr ); // atomic add, returns old value of 'z'  Assert.IsTrue( z == ptr + 10 );  Assert.IsTrue( atomic( z -= 10 ) == ptr + 10 ); // atomic sub, returns old value of 'z'  Assert.IsTrue( z == ptr );  atomic( v1 = 1 ); // atomic write  Assert.AreEqual( 1, atomic( v1 = 2 ) ); // atomic write, old value of 'v1' is result  Assert.AreEqual( 2, atomic( ++v1 ) ); // atomic increment, old value of 'v1' is result (actual value is 3)  Assert.AreEqual( 3, atomic( --v1 ) ); // atomic decrement, old value of 'v1' is result (actual value is 2)  var x = atomic( \*ptr, Acquire ); // atomic read of data pointed by 'ptr' with 'Acquire' semantic and non-volatile write to 'x'  atomic( y.SomeField = \*ptr, Release ); // non-atomic read of data pointed by 'ptr' and atomic write to 'y.SomeField' with 'Release' semantic  atomic( y.SomeField = atomic( \*ptr, Acquire ), Release ); // atomic both read and write  atomic( y.SomeField = atomic( \*ptr ) ); // atomic both read and write with 'SequentiallyConsistent' semantic by default  Assert.AreEqual( 2, atomic( v1 ) );  Assert.AreEqual( 2, atomic( v1 += 3 ) ); // atomic add, old value of 'v1' is result (actual value is 5)  Assert.AreEqual( 5, atomic( v1 ) );  Assert.AreEqual( 5, atomic( v1 -= 2 ) ); // atomic add, old value of 'v1' is result (actual value is 3)  Assert.AreEqual( 3, atomic( v1 ) );  }  }  } |

|  |
| --- |
| using System;  using System.Runtime.CompilerServices;  namespace Example {  class Test {  [EntryPoint]  private static void Main() {  Method1();  Method1();  }  private static bool \_Method1OnceAction = false;  private static void Method1() {  if( !atomic( \_Method1OnceAction = true ) ) { // similar to 'pthread\_once'  Console.WriteLine( "Executed once in {0}!", thismethod );  }  Console.WriteLine( thismethod );  }  private static string \_lazySingleton = null;  private static string LazySingleton {  get {  var result = atomic( \_lazySingleton );  if( result == null ) // similar to 'pthread\_once'  result = atomic( \_lazySingleton = new string( "initialized" ) );  return result;  }  }  }  } |

See documents:

* [LLVM Atomics](http://llvm.org/docs/Atomics.html)
* GCC [part1](http://gcc.gnu.org/wiki/Atomic/GCCMM/Optimizations) , [part2](http://gcc.gnu.org/wiki/Atomic/GCCMM/Optimizations/Details)

## base

The base keyword is used to access members of the base class from within a derived class:

* Call a method on the base class that has been overridden by another method.

|  |
| --- |
| public override void Dispose() { Cleanup(); base.Dispose(); } |

* Specify which base-class constructor should be called when creating instances of the derived class.

|  |
| --- |
| public class CustomHttpServer : HttpServer {  public CustomHttpServer( EventLoop eventLoop )  : base( eventLoop ) {  }  } |

## basetype

The basetype keyword is an alias for base *type access* expression. It simplifies refactoring and copy paste.

|  |
| --- |
| using System;  namespace Test {  class B1 {  static int Field;  static int Foo() { return 123; }  }  class D1 : B1 {  static new int Field;  static new int Foo() { return basetype.Foo() \* 2; } // calls B1.Foo()  [EntryPoint]  private static void Main() {  Console.WriteLine( typeof( B1 ) ); // Test.B1  Console.WriteLine( typeof( basetype ) ); // Test.B1  Console.WriteLine( memberinfo( B1.Field ) ); // Test.B1.Field  Console.WriteLine( memberinfo( basetype.Field ) ); // Test.B1.Field  Assert.IsTrue( &basetype.Field == &B1.Field );  Assert.IsTrue( basetype.Foo() == B1.Foo() );  }  }  } |

## bitcast

The **bitcast** keyword used to convert instances of incompatible types on per-bit basis. Must be used rarely since it most dangerous operator in terms of type safety.

It can convert:

* Pointer instances to pointers, **intptr**, **uintptr**
* Class,delegate,dynamic array instances to pointers
* Struct instances to another structs
* Functor instances to pointers, **intptr**, **uintptr**
* **float** to **int, uint**
* **double** to **long, ulong**
* **Guid** to **byte**[16], **int**[4]
* **string** to **Utf8String** ([valgrind](http://valgrind.org/) will be extremely happy)

|  |
| --- |
| using System;  namespace Test {  class Test {  struct S1 { float X; }  struct S2 { uint X; }  [EntryPoint]  private static void Main() {  Console.WriteLine( "0x{0:X8}", bitcast<uint>( 1f ) ); // 0x3F800000  Console.WriteLine( "0x{0:X16}", bitcast<ulong>( 1.0 ) ); // 0x3FF0000000000000  S1 s1 = new S1 { X = 1f };  S2 s2 = bitcast<S2>( s1 );  Console.WriteLine( "0x{0:X8}", s2.X ); // 0x3F800000  Console.WriteLine( "0x{0:X8}", bitcast<S2>( new S1 { X = 1f } ).X ); // 0x3F800000  }  }  } |

## bool

The **bool** keyword is an alias of type System.Boolean.

|  |
| --- |
| using System;  namespace Test {  class Test {  public static bool IsCurrentYearLeap {  get {  return DateTime.IsLeapYear( DateTime.UtcNow.Year );  }  }  [EntryPoint]  private static void Main() {  bool b1 = true;  bool b2 = false;  bool b3 = IsCurrentYearLeap;  if( b3 )  Console.WriteLine( "Year {0} is leap", DateTime.UtcNow.Year );  else  Console.WriteLine( "Year {0} is not leap", DateTime.UtcNow.Year );  }  }  } |

## break

The **break** statement terminates the closest enclosing loop or **switch** statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( int i = 0; i <= 10; i++ ) {  if( i >= 5 ) break;  Console.WriteLine( i ); // 0 1 2 3 4  }  }  }  } |

## byte

The **byte** keyword is an alias of type System.Byte. **byte** is unsigned 8-bit integer with range 0 to 255.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  byte b = 123;  Console.WriteLine( b + 100 ); // 223  Console.WriteLine( b + 200 ); // 67, since ( byte )( 123 + 200 ) == 67. Note: C# promotes both operans to int  Console.WriteLine( checkwrap( b + 100 ) ); // false  Console.WriteLine( checkwrap( b + 200 ) ); // true  b = 12;  Console.WriteLine( b \* 10 ); // 120  Console.WriteLine( b \* 100 ); // 176, since ( byte )( 12 \* 100 ) == 67. Note: C# promotes both operans to int  Console.WriteLine( checkwrap( b \* 10 ) ); // false  Console.WriteLine( checkwrap( b \* 100 ) ); // true  }  }  } |

## case

## cast

Same as C-style cast expression used sometimes to reduce parenthesis number.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  byte b = 123;  Console.WriteLine( ( ( double ) b ).ToString() );  Console.WriteLine( cast<double>( b ).ToString() ); // same as previous line  }  }  } |

## catch

## char

The char keyword is an alias of type System.LittleEndianChar or System.BigEndianChar depending of TargetCpu.

## char\_bigendian

The char\_bigendian keyword is an alias of type System.BigEndianChar.

## char\_littleendian

## checkwrap

The checkwrap keyword used to detect overflow in arithmetic operations with built-in integer and pointer types.

* Overflow detection performed using the addition, subtraction and multiplication operations
* Code outside checkwrap expression ( nested method calls, property evaluation etc. ) not taken into account

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  byte b = 123;  Assert.AreEqual( ( byte ) 223, b + 100 ); // 223  Assert.AreEqual( ( byte ) 67, b + 200 ); // 67, since ( byte )( 123 + 200 ) == 67. Note: C# promotes both operans to int  Assert.IsFalse( checkwrap( b + 100 ) ); // false  Assert.IsTrue( checkwrap( b + 200 ) ); // true  b = 12;  Assert.AreEqual( ( byte ) 120, b \* 10 ); // 120  Assert.AreEqual( ( byte ) 176, b \* 100 ); // 176, since ( byte )( 12 \* 100 ) == 67. Note: C# promotes both operans to int  Assert.IsFalse( checkwrap( b \* 10 ) ); // false  Assert.IsTrue( checkwrap( b \* 100 ) ); // true  var ptr = bitcast<byte\*>( uintptr.MaxValue - 1 );  Assert.IsFalse( checkwrap( ptr + 1 ) );  Assert.IsTrue( checkwrap( ptr + 2 ) ); // ptr wrapped  Assert.IsFalse( checkwrap( ++ptr ) );  Assert.IsTrue( checkwrap( ptr += 1 ) ); // ptr wrapped  }  }  } |

## class

## const

## continue

The continuestatement passes control to the next iteration of the enclosing iteration statement ( for, foreach, while, do ) in which it appears.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( int i = 1; i <= 10; i++ ) {  if( i < 5 ) continue;  Console.WriteLine( i ); // 5 6 7 8 9 10  }  }  }  } |

## declaringtype

Thedeclaringtype keyword is an alias for enclosing *type access* expression. It simplifies refactoring and copy paste.

|  |
| --- |
| using System;  namespace Test {  class D1 {  static int Field;  static int Foo() { return 123; }  class E1 {  static int Field;  static int Foo() { return declaringtype.Foo() \* 2; } // calls B1.Foo()  [EntryPoint]  private static void Main() {  Console.WriteLine( typeof( D1 ) ); // Test.D1  Console.WriteLine( typeof( declaringtype ) ); // Test.D1  Console.WriteLine( typeof( thistype ) ); // Test.D1.E1  Console.WriteLine( memberinfo( D1.Field ) ); // Test.D1.Field  Console.WriteLine( memberinfo( declaringtype.Field ) ); // Test.D1.Field  Assert.IsTrue( &declaringtype.Field == &D1.Field );  Assert.IsTrue( declaringtype.Foo() == D1.Foo() );  }  }  }  } |

## default

Specifies the default value of the type parameter. This will be null for reference types and zero for value types.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( default( int ) == 0 ); // true  Console.WriteLine( default( int\* ) == null ); // true  Console.WriteLine( default( string ) == null ); // true  }  }  } |

## delegate

The delegate keyword used to declare delegates.

## do

The do statement executes a statement or a block of statements repeatedly while a conditional expression evaluates to true. The body of the loop must be enclosed in braces.

|  |
| --- |
| using System;  namespace Test {  class D1 {  [EntryPoint]  public static void Main() {  int x = 0;    do {  Console.WriteLine( x ); // 0 1 2  ++x;  } while( x < 3 );  }  }  } |

## double

## double\_bigendian

## double\_littleendian

## else

## enum

## enumvalue

|  |
| --- |
| using System;  namespace Example {  class Test {  enum TestEnum {  Element1 = 20,  Element2 = 10,  Element3 = 30  }  [EntryPoint]  static void Main() {  // NOTE: enum elements are ordered by value in metadata  TestEnum t1 = enumvalue<TestEnum>( 0 ); // extracts '0' element from 'TestEnum' metadata  // GetIndex() finds index of value 't1' in 'TestEnum' metadata  Console.WriteLine( "TestEnum[{0}]={1}", t1.GetIndex(), t1 );  var t2 = enumvalue<TestEnum>( 1 ); // extracts '1' element from 'TestEnum' metadata  Console.WriteLine( "TestEnum[{0}]={1}", t2.GetIndex(), t2 );  Console.WriteLine( "== enumerate all enum values ==" );  for( var i = 0; i < typeof( TestEnum ).ElementCount; ++i ) {  Console.WriteLine( "TestEnum[{0}]={1}", i, enumvalue<TestEnum>( i ) );  }  }  }  } |

## event

## explicit

## extern

## false

## finally

## float

## float\_bigendian

## float\_littleendian

## for

The for statement executes a statement or a block of statements repeatedly while a specified expression evaluates to true.

|  |
| --- |
| using System;  namespace Test {  class D1 {  [EntryPoint]  static void Main() {  for( int i = 0; i <= 3; ++i ) {  Console.WriteLine( i ); // 0 1 2 3  }  }  }  } |

## foreach

## functor

The functor keyword used to declare pointer to methods.

## goto

The goto statement transfers the program control directly to a labeled statement.

|  |
| --- |
| namespace Test {  class Test {  [System.EntryPoint]  public static void Main() {  goto SomeLabel;  System.Console.WriteLine( "Hello" ); // never called  SomeLabel:  System.Console.WriteLine( "World" );  }  }  } |

|  |
| --- |
| namespace Test {  class Test {  [System.EntryPoint]  public static void Main() {  int i = 10;  switch( i ) {  case 10: System.Console.Write( "Hello" ); goto case 5;  case 5: System.Console.Write( " World" ); goto default;  default: System.Console.WriteLine( " %USERNAME%!" ); break;  }  // "Hello World %USERNAME%!" printed at this point  }  }  } |

## heapalloc

The heapalloc expression provides syntax sugar for System.Runtime.Memory.Allocate/AllocateClear and takes into account type alignment.

|  |
| --- |
| using System;  using System.Runtime;  namespace Test {  class Test {  struct UnalignedStruct {  int Field;  UnalignedStruct( int value ) {  Field = value;  }  ~UnalignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  }  [Alignment( Boundary = 32 )]  struct AlignedStruct {  int Field;  ~AlignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  }  [EntryPoint]  public static void Main() {  // var p0 = ( UnalignedStruct\* ) System.Runtime.Memory.Allocate( sizeof( UnalignedStruct ) );  var p0 = heapalloc UnalignedStruct;  p0->Field = 1;  // allocate instance of 'UnalignedStruct' in heap with alignment 4  // UnalignedStruct\* p1 = ( UnalignedStruct\* ) System.Runtime.Memory.Allocate( sizeof( UnalignedStruct ) ); p1->Field = 1;  var p1 = heapalloc UnalignedStruct { Field = 2 };  // allocated instance of 'UnalignedStruct' in heap with alignment 32  // AlignedStruct\* p2 = ( AlignedStruct\* ) System.Runtime.Memory.AllocateAligned( ( uint ) typeof( AlignedStruct ).ContentSize, typeof( AlignedStruct ).ContentAlignment ); p2->Field = 2;  var p2 = heapalloc AlignedStruct { Field = 3 };  // allocate instance of 'UnalignedStruct' in heap with alignment 4, then call constructor 'UnalignedStruct(int)'  var p3 = heapalloc UnalignedStruct( 4 );  Console.WriteLine( "p0={0} p1={1} p2={2} p3={3}", p0, p1, p2, p3 );  Console.WriteLine( "{0} {1} {2} {3}", p0->Field, p1->Field, p2->Field, p3->Field ); // 1 2 3 4  heapfree p3; // ~UnalignedStruct( p3 ); System.Runtime.Memory.Deallocate( p3 );  heapfree p2; // ~AlignedStruct( p2 ); System.Runtime.Memory.DeallocateAligned( p2 );  heapfree p1; // ~UnalignedStruct( p1 ); System.Runtime.Memory.Deallocate( p1 );  heapfree p0; // ~UnalignedStruct( p0 ); System.Runtime.Memory.Deallocate( p0 );  }  }  } |

|  |
| --- |
| p0=04050F58 p1=04050F90 p2=04051000 p3=04050FC8  1 2 3 4  Test.Test.UnalignedStruct.#dtor()( 04050FC8 )  Test.Test.AlignedStruct.#dtor()( 04051000 )  Test.Test.UnalignedStruct.#dtor()( 04050F90 )  Test.Test.UnalignedStruct.#dtor()( 04050F58 ) |

## heapfree

The heapfree expression provides syntax sugar for System.Runtime.Memory.Deallocate/DeallocateAligned. See example for heapallockeyword.

## if

The if statement selects a statement for execution based on the value of a Boolean expression.

|  |
| --- |
| namespace Test {  class Test {  [System.EntryPoint]  public static void Main() {  int i = 10;  if( i < 5 ) System.Console.WriteLine( "i < 5" );  if( i >= 5 ) System.Console.WriteLine( "i >= 5" ); // i >= 5  if( i < 5 ) System.Console.WriteLine( "i < 5" );  else System.Console.WriteLine( "i >= 5" ); // i >= 5  }  }  } |

Also if statement can be used inside asm block with constant condition expression:

|  |
| --- |
| public static float operator |( Quaternion& left, Quaternion& right ) asm {  X86\_32 {  feature( SSE1 ) {  if( thismethod.CallingConvention == CallingConvention.FastCallX86 ) {  // ecx - left  // edx - right  }  else {  mov ecx, [esp + 4] // left  mov edx, [esp + 8] // right  }  movaps xmm0, [ecx]  … … … |

## implicit

## interface

## int

## int\_bigendian

## int\_littleendian

## intptr

The intptr type designed to be an integer whose size is platform-specific. intptr objects mostly used for interoperation with ASM/C/C++ code, for example it can be used to hold kernel-mode pointers or handles in userspace (since dereference is not intended).

On X86\_32, PowerPC\_32: sizeof( intptr ) == sizeof( void\* ) == 4 ; typeof( intptr ) == typeof( int )

On X86\_64, PowerPC\_64: sizeof( intptr ) == sizeof( void\* ) == 8 ; typeof( intptr ) == typeof( long )

## internal

## is

Checks if an object is compatible with a given type.

|  |
| --- |
| using System;  namespace Test {  class Test {  static void Foo( RuntimeObjectBase someObject ) {  if( someObject is string ) Console.WriteLine( "1. someObject is 'Utf16String'" );  else if( someObject is Utf32String ) Console.WriteLine( "2. someObject is 'Utf32String'" );  else Console.WriteLine( "3. someObject is '{0}'", someObject.GetType() );  }  [EntryPoint]  public static void Main() {  Foo( ( AnsiString ) "1" ); // 3. someObject is 'System.AnsiString'  Foo( ( Utf8String ) "1" ); // 3. someObject is 'System.Utf8String'  Foo( ( Utf16String ) "1" ); // 1. someObject is 'Utf16String'  Foo( ( Utf32String ) "1" ); // 2. someObject is 'Utf32String'  }  }  } |

## long

## long\_bigendian

## long\_littleendian

## memberinfo

The memberinfo keyword used to obtain metadata object for a member.

|  |
| --- |
| using System;  using System.Reflection;  using System.Runtime;  namespace Example {  class Test {  struct SomeStruct {  int Field1;  int Field2;  static int Field3;  event Delegates.Action Ev;  void Method() { }  int Prop { get { return 1; } }  int IndexedProp[int index] { get { return 2 \* index; } }  byte OverloadedIndexedProp[byte index] { get { return 2 \* index; } }  int OverloadedIndexedProp[int index] { get { return 2 \* index; } }  }  class SomeClass {  void OverloadedMethod( int x ) { }  void OverloadedMethod( float z ) { }  virtual void VirtualMethod1( int x ) { }  virtual void VirtualMethod2( float z ) { }  }  [EntryPoint]  private static void Main() {  // Field 'Example.Test.SomeStruct.Field1': byteoffset=0 isstatic=false type='int'  Print( memberinfo( SomeStruct.Field1 ) );  // Field 'Example.Test.SomeStruct.Field2': byteoffset=4 isstatic=false type='int'  Print( memberinfo( SomeStruct.Field2 ) );  // Field 'Example.Test.SomeStruct.Field3': byteoffset=138533612 isstatic=true type='int'  Print( memberinfo( SomeStruct.Field3 ) );  // Event 'Example.Test.SomeStruct.Ev': return='System.Delegates.Action'  Print( memberinfo( SomeStruct.Ev ) );  // Property 'Example.Test.SomeStruct.Prop': return='int'  Print( memberinfo( SomeStruct.Prop ) );  // Indexed property 'Example.Test.SomeStruct.IndexedProp[int]': return='int'  Print( memberinfo( SomeStruct.IndexedProp ) );  // Indexed property 'Example.Test.SomeStruct.OverloadedIndexedProp[byte]': return='byte'  Print( memberinfo( SomeStruct.OverloadedIndexedProp, byte ) );  // Indexed property 'Example.Test.SomeStruct.OverloadedIndexedProp[int]': return='int'  Print( memberinfo( SomeStruct.OverloadedIndexedProp, int ) );  // Method 'Example.Test.SomeStruct.Method()': vslot=0 address=081DB220 ParameterCount=0 return='void'  Print( memberinfo( SomeStruct.Method ) );  // Method 'Example.Test.SomeClass.OverloadedMethod(int)': vslot=0 address=081DB230 ParameterCount=1 return='void'  Print( memberinfo( SomeClass.OverloadedMethod, int ) );  // Method 'Example.Test.SomeClass.OverloadedMethod(float)': vslot=0 address=081DB250 ParameterCount=1 return='void'  Print( memberinfo( SomeClass.OverloadedMethod, float ) );  // Method 'Example.Test.SomeClass.VirtualMethod1(int)': vslot=3 address=081DB270 ParameterCount=1 return='void'  Print( memberinfo( SomeClass.VirtualMethod1, int ) );  // Method 'Example.Test.SomeClass.VirtualMethod2(float)': vslot=4 address=081DB290 ParameterCount=1 return='void'  Print( memberinfo( SomeClass.VirtualMethod2, float ) );  }  private static void Print( UserType.Field& field ) {  Console.WriteLine( "Field '{0}': byteoffset={1} isstatic={2} type='{3:L}'", field, field.ByteOffset, field.IsStatic, field.FieldType );  }  private static void Print( UserType.Property& property ) {  Console.WriteLine( "Property '{0}': return='{1:L}'", property, property.PropertyType );  }  private static void Print( UserType.Event& property ) {  Console.WriteLine( "Event '{0}': return='{1:L}'", property, property.EventType );  }  private static void Print( UserType.IndexedProperty& property ) {  Console.WriteLine( "Indexed property '{0}': return='{1:L}'", property, property.PropertyType );  }  private static void Print( UserType.Method& method ) {  Console.WriteLine( "Method '{0}': vslot={1} address={2} ParameterCount={3} return='{4:L}'", method, method.VirtualSlot, method.Address, method.ParameterCount, method.ReturnType );  }  }  } |

## namespace

The namespace keyword is used to declare a scope that contains types or nested namespaces.

|  |
| --- |
| namespace Example {  class A { } // Example.A  namespace NestedNamespace { // Example.NestedNamespace  class B { } // Example.NestedNamespace.B  }  } |

## new

The new keyword inside *new expression* used to allocate type instances on stack, invoke constructors and perform initialization.

|  |
| --- |
| using System;  using System.Runtime;  namespace Test {  class Test {  struct UnalignedStruct {  int Field;  UnalignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  UnalignedStruct( int value ) {  Console.WriteLine( "{0}( {1} ) value={2}", thismethod, &this, value );  Field = value;  }  ~UnalignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  }  [Alignment( Boundary = 32 )]  struct AlignedStruct {  int Field;  AlignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  ~AlignedStruct() {  Console.WriteLine( "{0}( {1} )", thismethod, &this );  }  }  [EntryPoint]  public static void Main() {  Console.WriteLine( "p0" );  // allocate instance of 'UnalignedStruct' in stack with alignment 4, no constructor call performed  var p0 = new UnalignedStruct { };  p0.Field = 1;  Console.WriteLine( "p1" );  // allocate instance of 'UnalignedStruct' in stack with alignment 4, no constructor call performed  var p1 = new UnalignedStruct { Field = 2 };  Console.WriteLine( "p2" );  // allocated instance of 'UnalignedStruct' in stack with alignment 32, no constructor call performed  var p2 = new AlignedStruct { Field = 3 };  Console.WriteLine( "p3" );  // allocate instance of 'UnalignedStruct' in stack with alignment 4, then call constructor 'UnalignedStruct(int)'  var p3 = new UnalignedStruct( 4 );  Console.WriteLine( "p4" );  // allocate instance of 'UnalignedStruct' in stack with alignment 4, then call constructor 'UnalignedStruct()'  var p4 = new UnalignedStruct();  Console.WriteLine( "p5" );  // allocate instance of 'UnalignedStruct[3]' in stack with alignment 4, then call constructor 'UnalignedStruct(int)' for each element  var p5 = new UnalignedStruct[3] { new UnalignedStruct( 1 ), new UnalignedStruct( 2 ), new UnalignedStruct( 3 ) };  Assert.IsTrue( p5[0].Field == 1 ); Assert.IsTrue( p5[1].Field == 2 ); Assert.IsTrue( p5[2].Field == 3 );  Console.WriteLine( "p6" );  // allocate instance of 'UnalignedStruct[3]' in stack with alignment 4, then call constructor 'UnalignedStruct(int)' for each element  var p6 = new UnalignedStruct[3]( 2 );  for( var i = 0; i < p6.Length; ++i ) Assert.IsTrue( p6[0].Field == 2 );  Console.WriteLine( "p6a" );  // allocate instance of 'UnalignedStruct[3]' in stack with alignment 4, then evaluate default initializer 'UnalignedStruct(3)' to initialize each element  var p6a = new UnalignedStruct[3] default( new UnalignedStruct( 3 ) );  for( var i = 0; i < p6a.Length; ++i ) Assert.IsTrue( p6a[0].Field == 3 );  Console.WriteLine( "p7" );  // allocate instance of 'UnalignedStruct[]', then call constructor 'UnalignedStruct(int)' for each element  var p7 = new[3] UnalignedStruct { new UnalignedStruct( 1 ), new UnalignedStruct( 2 ), new UnalignedStruct( 3 ) };  Assert.IsTrue( p7[0].Field == 1 ); Assert.IsTrue( p7[1].Field == 2 ); Assert.IsTrue( p7[2].Field == 3 );  Console.WriteLine( "p8" );  // allocate instance of 'UnalignedStruct[]', then call constructor 'UnalignedStruct(int)' for each element  var p8 = new[3] UnalignedStruct( 2 );  for( var i = 0; i < p8.Length; ++i ) Assert.IsTrue( p8[0].Field == 2 );  Console.WriteLine( "p8a" );  // allocate instance of 'UnalignedStruct[]', then evaluate default initializer 'UnalignedStruct(3)' to initialize each element  var p8a = new[3] UnalignedStruct default( new UnalignedStruct( 3 ) );  for( var i = 0; i < p8a.Length; ++i ) Assert.IsTrue( p8a[0].Field == 3 );  }  }  } |

When used as a modifier, the new keyword explicitly hides a member inherited from a base class.

|  |
| --- |
| using System;  namespace Test {  class Test {  struct A {  int Field;  void Method() { Console.WriteLine( "{0} Field={1}", thismethod, Field ); }  }  struct B : A {  new int Field; // hide Test.Test.A.Fieldcwith new field  new void Method() { Console.WriteLine( "{0} Field={1}", thismethod, Field ); } // hide Test.Test.A.Method with new method  }  [EntryPoint]  public static void Main() {  var b = new B { };  A& a = b; // a points to same location as &b  a.Field = 2;  b.Field = 3;  // Test.Test.A.Method() Field=2  a.Method();  // Test.Test.B.Method() Field=3  b.Method();  }  }  } |

## null

The null keyword is a literal that represents a null reference, one that does not refer to any object. null is the default value of reference-type and pointer-type variables.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int\* x = null, y;  Console.WriteLine( x == null ); // true  Console.WriteLine( y == null ); // true  Console.WriteLine( ( intptr ) x ); // 0  }  }  } |

## nullref

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int a = 10;  int& b = a;  int& c = nullref;  Console.WriteLine( b != nullref ); // true  Console.WriteLine( c == nullref ); // true  }  }  } |

## operator

## override

## private

## protected

## public

## readonly

## return

## sbyte

## sealed

## short

## short\_bigendian

## short\_littleendian

## sizeof

## stackalloc

## static

## struct

## string

## switch

## this

The this keyword refers to the current instance of the class or struct. The following are common uses of this:

* To qualify members hidden by similar names:

|  |
| --- |
| namespace Test {  class Test {  class SomeClass {  int Field;  public SomeClass() {  }  public void SetUnqualified( int Field ) { // wrong case! don write your code in that way  Field = Field; // self-assignment  }  public void SetQualified( int Field ) { // wrong case! don write your code in that way  this.Field = Field;  }  }  [System.EntryPoint]  public static void Main() {  var v = new SomeClass();  v.SetQualified( 123 ); // v.Field == 123  System.Console.WriteLine( v.Field ); // 123  v.SetUnqualified( 456 ); // v.Field still == 123  System.Console.WriteLine( v.Field ); // 123  }  }  } |

* To get current instance of the class or struct:

|  |
| --- |
| namespace Test {  class Test {  class SomeClass {  int Field;  public SomeClass() {  Field = 123;  }  public void Print() {  PrintImpl( this, "D4" ); // 0123  }  private static void PrintImpl( SomeClass value, string format ) {  var sb = new System.StringBuilder();  sb.Append( value.Field, format );  System.Console.WriteLine( sb );  }  }  [System.EntryPoint]  public static void Main() {  var v = new SomeClass();  v.Print();  }  }  } |

* To declare indexers:

|  |
| --- |
| namespace Test {  class Test {  class SomeClass {  int Field;  public SomeClass() {  Field = 16;  }  public int this[int index1, int index2] {  get { return Field \* index1 + index2; }  }  }  [System.EntryPoint]  public static void Main() {  var v = new SomeClass();  System.Console.WriteLine( v[4, 6] ); // 70  }  }  } |

## thistype

The thistype keyword is an alias for this *type access* expression. It simplifies refactoring and copy paste.

|  |
| --- |
| using System;  namespace Test {  class D1 {  static int Field;  static int Foo() { return 123; }  [EntryPoint]  private static void Main() {  Console.WriteLine( typeof( D1 ) ); // Test.D1  Console.WriteLine( typeof( thistype ) ); // Test.D1  // Console.WriteLine( typeof( this ) ); access to 'this' not valid in static method  Console.WriteLine( memberinfo( Field ) ); // Test.D1.Field  Console.WriteLine( memberinfo( D1.Field ) ); // Test.D1.Field  Console.WriteLine( memberinfo( thistype.Field ) ); // Test.D1.Field  }  }  } |

## thismethod

The thismethod keyword is an alias for memberinfo of this method.

|  |
| --- |
| using System;  namespace Test {  class D1 {  [EntryPoint]  private static void Main() {  Console.WriteLine( memberinfo( Main ) ); // Test.D1.Main()  Console.WriteLine( thismethod ); // Test.D1.Main()  Assert.IsTrue( memberinfo( Main ).Address == thismethod.Address );  Foo();  Assert.IsTrue( ( Property += 10 ) == 20 );  }  static void Foo() {  Console.WriteLine( thismethod ); // Test.D1.Foo()  }  static int Property {  get {  Console.WriteLine( thismethod ); // Test.D1.#get\_Property()  return 10;  }  set {  Console.WriteLine( thismethod ); // Test.D1.#set\_Property(int)  }  }  }  } |

## thisproperty

The thismethod keyword is an alias for memberinfo of this property.

|  |
| --- |
| using System;  namespace Test {  class D1 {  [EntryPoint]  private static void Main() {  Assert.AreEqual( typeof( int ), memberinfo( Property ).PropertyType );  Assert.IsTrue( ( Property += 10 ) == 20 );  }  static int Property {  get {  Console.WriteLine( memberinfo( Property ) ); // Test.D1.Property  Console.WriteLine( thisproperty ); // Test.D1.Property  Console.WriteLine( thismethod ); // Test.D1.#get\_Property()  Assert.AreEqual( typeof( int ), thisproperty.PropertyType );  return 10;  }  set {  Console.WriteLine( memberinfo( Property ) ); // Test.D1.Property  Console.WriteLine( thisproperty ); // Test.D1.Property  Console.WriteLine( thismethod ); // Test.D1.#set\_Property(int)  }  }  }  } |

## throw

## true

## try

## typeof

The typeof keyword used to obtain metadata object for a type.

|  |
| --- |
| using System;  using System.Reflection;  using System.Runtime;  namespace Example {  class Test {  struct SomeStruct {  byte \_trash1;  int X;  byte \_trash2;  double Y;  byte \_trash3;  float Z;  byte \_trash4;  }  class SomeClass {  byte \_trash1;  int X;  byte \_trash2;  double Y;  byte \_trash3;  float Z;  byte \_trash4;  }  enum SomeEnum {  El1 = 6,  El2 = 3,  El3 = 7,  El4 = 2,  }  functor void SomeFunctor( int x, int y );  delegate void SomeDelegate( int x, int y, int z );  [EntryPoint]  private static void Main() {  // User type 'int': size=4 stacksize=4 isclass=false alignment=4  // Field count = 0  Print( typeof( int ) ); // call Print( UserType& )  // User type 'Example.Test.SomeStruct': size=20 stacksize=20 isclass=false alignment=4  // Field count = 7  // \_trash1 at offset 16  // X at offset 0  // \_trash2 at offset 17  // Y at offset 4  // \_trash3 at offset 18  // Z at offset 12  // \_trash4 at offset 19  Print( typeof( Example.Test.SomeStruct ) ); // call Print( UserType& )  // User type 'Example.Test.SomeClass': size=40 stacksize=4 isclass=true alignment=4  // Field count = 7  // \_trash1 at offset 36  // X at offset 20  // \_trash2 at offset 37  // Y at offset 24  // \_trash3 at offset 38  // Z at offset 32  // \_trash4 at offset 39  Print( typeof( SomeClass ) ); // call Print( UserType& )  // Enum 'Example.Test.SomeEnum': size=4  // ElementCount count = 4  // El4 = 2  // El2 = 3  // El1 = 6  // El3 = 7  // == typesafe way to print all names/values ==  // El4 = 2  // El2 = 3  // El1 = 6  // El3 = 7  Print( typeof( SomeEnum ) ); // call Print( EnumType& )  Console.WriteLine( "== typesafe way to print all names/values ==" );  for( var i = 0; i < typeof( SomeEnum ).ElementCount; ++i ) {  SomeEnum element = SomeEnum.GetValue( i );  Console.WriteLine( " {0} = {1}", element, element.Value() );  }  // Functor 'Example.Test.SomeFunctor': ParameterCount=2  Print( typeof( SomeFunctor ) ); // call Print( FunctorType& )  // Delegate 'Example.Test.SomeDelegate': ParameterCount=3  Print( typeof( SomeDelegate ) ); // call Print( DelegateType& )  // Pointer 'int\*\*': Pointee='int\*'  Print( typeof( int\*\* ) ); // call Print( DelegateType& )  int x;  Assert.IsTrue( typeof( x ) == typeof( int ) );  }  private static void Print( UserType& type ) {  Console.WriteLine( "User type '{0:L}': size={1} stacksize={2} isclass={3} alignment={4}", type, type.ContentSize, type.InstanceSize, type.IsClass, type.ContentAlignment );  Console.WriteLine( " Field count = {0}", type.FieldCount );  foreach( var item in type.Fields ) {  Console.WriteLine( " {0} at offset {1}", item->Name, item->ByteOffset );  }  }  private static void Print( EnumType& type ) {  Console.WriteLine( "Enum '{0}': size={1}", type, type.InstanceSize );  Console.WriteLine( " ElementCount count = {0}", type.ElementCount );  for( var i = 0; i < type.ElementCount; ++i ) {  Console.WriteLine( " {0} = {1}", type.Names[i], type.GetUInt32Value( i ) );  }  }  private static void Print( FunctorType& type ) {  Console.WriteLine( "Functor '{0}': ParameterCount={1}", type, type.ParameterCount );  }  private static void Print( DelegateType& type ) {  Console.WriteLine( "Delegate '{0}': ParameterCount={1}", type, type.ParameterCount );  }  private static void Print( PointerType& type ) {  Console.WriteLine( "Pointer '{0:L}': Pointee='{1:L}'", type, type.Pointee );  }  }  } |

## uint

## uint\_bigendian

## uint\_littleendian

## uintptr

The uintptr type designed to be an integer whose size is platform-specific. uintptr objects mostly used for interoperation with ASM/C/C++ code, for example it can be used to hold kernel-mode pointers or handles in userspace (since dereference is not intended).

On X86\_32, PowerPC\_32: sizeof( uintptr ) == sizeof( void\* ) == 4 ; typeof( uintptr ) == typeof( uint )

On X86\_64, PowerPC\_64: sizeof( uintptr ) == sizeof( void\* ) == 8 ; typeof( uintptr ) == typeof( ulong )

## ulong

## ulong\_bigendian

## ulong\_littleendian

## ushort

## ushort\_bigendian

## ushort\_littleendian

## using

## vararg

The **vararg** keyword lets you specify a method parameter that takes a variable number of arguments.

|  |
| --- |
| using System;  using System.Runtime;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  // @{ parameter array constructed on stack  Print( 1, 2, 3 ); // 1 2 3  PrintDelegate( 1, 2, 3, 4 ); // 1 2 3 4  PrintDelegate2( 1, 2, 3, 4 ); // 2 3  // @}  var values = new int[5] { 100, 111, 122, 133, 144 };  Print( vararg( values, values.Length ) ); // parameter array constructed by 'vararg' expression  UnknownTypePass( 10, 10.5, "hello", ( Utf8String ) "world" ); // int=10 double=10.5 string=hello System.Utf8String=world  }  public static void PrintDelegate( vararg int parameters ) {  Print( vararg( parameters, parameters.Length ) ); // parameter array passed to another vararg method  }  public static void PrintDelegate2( vararg int parameters ) {  if( !Assert.IsTrue( parameters.Length >= 2 ) ) return;  Print( vararg( parameters + 1, parameters.Length - 2 ) ); // parameter array passed to another vararg method  }  public static void Print( vararg int parameters ) {  Console.WriteLine( "Length = {0}", parameters.Length );  for( int i = 0; i < parameters.Length; ++i )  Console.Write( "{0} ", parameters[i] );  Console.WriteLine();  }  public static void UnknownTypePass( vararg TypedReference parameters ) {  Console.WriteLine( "Length = {0}", parameters.Length );  for( int i = 0; i < parameters.Length; ++i )  Console.Write( "{0:L}={1} ", parameters[i].Type, parameters[i] );  Console.WriteLine();  }  }  } |

## virtual

The **virtual** keyword is used to modify a method, property, indexer, indexed property, or event declaration and allow for it to be overridden in a derived class.

|  |
| --- |
| using System;  using System.Runtime;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  var b = new B();  var a = ( A ) b;  a.NotVirtualMethod(); // Test.Test.A.NotVirtualMethod()  a.VirtualHiddenMethod(); // Test.Test.A.VirtualHiddenMethod()  a.VirtualOverridenMethod(); // Test.Test.B.VirtualOverridenMethod()  b.NotVirtualMethod(); // Test.Test.B.NotVirtualMethod()  b.VirtualHiddenMethod(); // Test.Test.B.VirtualHiddenMethod()  b.VirtualOverridenMethod(); // Test.Test.B.VirtualOverridenMethod()  }  class A {  A() { }  void NotVirtualMethod() { Console.WriteLine( thismethod ); }  virtual void VirtualHiddenMethod() { Console.WriteLine( thismethod ); }  virtual void VirtualOverridenMethod() { Console.WriteLine( thismethod ); }  }  class B : A {  B() { }  new void NotVirtualMethod() { Console.WriteLine( thismethod ); } // hide 'A.NotVirtualMethod' by name  new void VirtualHiddenMethod() { Console.WriteLine( thismethod ); } // hide 'A.VirtualMethod' by name  override void VirtualOverridenMethod() { Console.WriteLine( thismethod ); } // override 'A.VirtualOverridenMethod'  }  }  } |

## volatile

The volatile keyword used to modify default behavior of memory access. The optimizers may not change the number of volatile operations or change their order of execution relative to other volatile operations. No cross-thread synchronization performed.

|  |
| --- |
| using System;  namespace Test {  class Test {  struct S1 { int SomeField; }  [EntryPoint]  private static void Main() {  var y = new S1 { SomeField = 6 };  int\* ptr = &y.SomeField;  var x = volatile( \*ptr ); // volatile read of data pointed by 'ptr' and non-volatile write to 'x'  volatile( y.SomeField = \*ptr ); // non-volatile read of data pointed by 'ptr' and volatile write to 'y.SomeField'  volatile( y.SomeField = volatile( \*ptr ) ); // volatile read and volatile write  }  }  } |

## void

The **void** keyword is an alias for System.Void type.

|  |
| --- |
| using System;  using System.Runtime;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  void\* ptr = CRuntime.malloc( 16 );  // 'malloc' returned untyped pointer to memory region  // You can interpret this memory region in different ways by casting this pointer to appropriate type (like byte\*, char\*, short\*, int\*, Guid\*)  var typedPtr = ( byte\* ) ptr; // 'typedPtr' points to same memory region as 'ptr'  var typedPtr2 = ( short\* ) ptr; // 'typedPtr2' points to same memory region as 'ptr'  // \*ptr = 777; // error, 'ptr' is untyped  \*typedPtr2 = 0x6969; // write 'short'  Assert.IsTrue( \*typedPtr == 0x69 ); // read 'byte'  CRuntime.free( ptr );  Print();  // var ret = Print(); // error, Print() have no return  }  static void Print() { // method have no return  Console.WriteLine( "Print!" );  }  }  } |

## while

The **while** keyword used to define the *while statement*.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  while( true ) { // endless while  Console.WriteLine( "True!" );  }  }  }  } |

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 1;  while( x < 5 ) {  Console.Write( "{0} ", x ); // 1 2 3 4  ++x;  }  Console.WriteLine();  x = 1;  while( ++x < 5 )  Console.Write( "{0} ", x ); // 2 3 4  Console.WriteLine();  x = 1;  while( x++ < 5 )  Console.Write( "{0} ", x ); // 2 3 4 5  Console.WriteLine();  }  }  } |

## yield

The **yield** keyword used to:

* Define iterators. The **yield** keyword in return type signals to the compiler that the method is an iterator block. The compiler generates a **struct** or **class** to implement the behavior that is expressed in the iterator block.
* Return value from iterator. In a *yield return statement*, expression is evaluated and returned as a value to the enumerator object.
* Break iterator execution. In a *yield break statement*, control is unconditionally returned to the caller of the iterator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  foreach( var item in GetIterator() )  Console.Write( "{0} ", item ); // 1 3 5 6 7 8 9  Console.WriteLine();  foreach( var item in GetPowerIterator( 3, 4 ) )  Console.Write( "{0} ", item ); // 3 9 27 81  Console.WriteLine();  var manualIterator = GetPowerIterator( 4, 4 );  while( manualIterator.MoveNext() )  Console.Write( "{0} ", manualIterator.Current ); // 4 16 64 256  Console.WriteLine();  }  static yield<int> GetIterator() {  yield return 1;  yield return 3;  for( var i = 5; i < 10; ++i ) {  yield return i;  }  yield break;  yield return 100;  }  static yield<int> GetPowerIterator( int number, int exponent ) {  var result = 1;  for( var counter = 0; counter < exponent; ++counter ) {  result = result \* number;  yield return result;  }  }  }  } |

# Bamelg operators

**[]**

Square brackets **[]** are used for fixed arrays, dynamic arrays, indexers, indexed properties and pointers.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var someObject = new SomeClass();  int[] dynamicArray = new[3] int { 1, 2, 3 };  Console.WriteLine( dynamicArray[1] ); // 2  int[3] fixedArray = new int[3] { 1, 2, 3 };  Console.WriteLine( fixedArray[1] ); // 2  int\* ptr = &fixedArray[1];  Console.WriteLine( ptr[1] ); // 3  Console.WriteLine( someObject.SomeIndexedProperty[4] ); // 4  Console.WriteLine( someObject[5] ); // 25  }  class SomeClass {  SomeClass() { }  int this[int index] { get { return index \* index; } }  int SomeIndexedProperty[int index] { get { return index; } }  }  }  } |

**()**

* Order of operations in an expression:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int a = 1, b = 2, c = 3, d = 4;  Console.WriteLine( ( a | b ) & ( c ^ d ) ); // 3  Console.WriteLine( a | b & c ^ d ); // 7  }  }  } |

* Casts, or type conversions:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  double someVariable = 11.5;  Console.WriteLine( ( int ) someVariable ); // 11  }  }  } |

**.**

The dot operator (**.**) used for member access. The dot operator specifies a member of a type or namespace.

|  |
| --- |
| using System;  using System.IO; // namespace 'IO' in namespace 'System'  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( "Hello World" ); // Invoke method 'WriteLine' which declared in class 'Console' [in imported namespace 'System']  System.Console.WriteLine( "Hello World" ); // Invoke method 'WriteLine' which declared in class 'Console' which declared in namespace 'System'  var someVariable = new SomeClass();  someVariable.SomeField = 123; // assign to field 'SomeField'  someVariable.Foo(); // invoke member function 'Foo'  }  class SomeClass {  SomeClass() { }  int SomeField;  void Foo() { }  }  }  } |

**+**

The + operator can function as either a unary or a binary operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( +3 ); // 3  Console.WriteLine( +( 3 ) ); // 3  Console.WriteLine( 3 + 3 ); // 6  }  }  } |

**-**

The - operator can function as either a unary or a binary operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( -3 ); // -3  Console.WriteLine( -( 3 ) ); // -3  Console.WriteLine( 3 - 3 ); // 0  }  }  } |

**\***

* Multiplication operator (\*), which computes the product of its operands.

|  |
| --- |
| namespace Test {  class Test {  [System.EntryPoint]  public static void Main() {  System.Console.WriteLine( 1 \* 2 \* 3 ); // 6  }  }  } |

* Dereference operator, which allows reading and writing to a pointer.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var x = 11;  var y = &x;  Console.WriteLine( \*y ); // 11  \*y = 12; // write to memory pointed by pointer 'y', change value of 'x' to '12'  Console.WriteLine( x ); // 12  }  }  } |

**/**

The division operator divides its first operand by its second. All numeric types have predefined division operators.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( 100 / 10 ); // 10  Console.WriteLine( 100 / 10 / 2 ); // 5  Console.WriteLine( 100 \* 2 / 10 / 2 ); // 10  }  }  } |

**%**

The modulus operator computes the remainder after dividing its first operand by its second. All numeric types have predefined modulus operators.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( 19 % 5 ); // 4  Console.WriteLine( 19 % 5 % 3 ); // 1  }  }  } |

**&**

The & operator can function as either a unary or a binary operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var x = 5;  Console.WriteLine( x & 0 ); // 0  Console.WriteLine( x & 1 ); // 1  Console.WriteLine( x & 3 ); // 1  Console.WriteLine( x & 7 ); // 5  int\_littleendian xle = 5;  int\_bigendian xbe = 5;  Console.WriteLine( typeof( x & 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xle & 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xbe & 1 ) ); // System.BigEndianInt32  }  }  } |

**|**

Binary | operators are predefined for the integral types and bool. For integral types, | computes the bitwise OR of its operands. For bool operands, | computes the logical OR of its operands; that is, the result is false if and only if both its operands are false.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var x = 5;  Console.WriteLine( x | 0 ); // 5  Console.WriteLine( x | 1 ); // 5  Console.WriteLine( x | 3 ); // 7  int\_littleendian xle = 5;  int\_bigendian xbe = 5;  Console.WriteLine( typeof( x | 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xle | 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xbe | 1 ) ); // System.BigEndianInt32  }  }  } |

**^**

Binary ^ operators are predefined for the integral types and bool. For integral types, ^ computes the bitwise exclusive-OR of its operands. For bool operands, ^ computes the logical exclusive-or of its operands; that is, the result is true if and only if exactly one of its operands is true.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var x = 5;  Console.WriteLine( x ^ 0 ); // 5  Console.WriteLine( x ^ 1 ); // 4  Console.WriteLine( x ^ 3 ); // 6  Console.WriteLine( x ^ 7 ); // 2  int\_littleendian xle = 5;  int\_bigendian xbe = 5;  Console.WriteLine( typeof( x ^ 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xle ^ 1 ) ); // System.LittleEndianInt32  Console.WriteLine( typeof( xbe ^ 1 ) ); // System.BigEndianInt32  }  }  } |

**!**

The logical negation operator (!) is a unary operator that negates its operand. It defined for bool and returns true if and only if its operand is false.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  bool x = false;  Console.WriteLine( !x ); // true  Console.WriteLine( !!x ); // false  Console.WriteLine( !!!x ); // true  }  }  } |

**~**

The ~ operator performs a bitwise complement operation on its operand, which has the effect of reversing each bit.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var x = 5;  Console.WriteLine( "{0:X8}", x ); // 00000005  Console.WriteLine( "{0:B32}", x ); // 00000000000000000000000000000101  Console.WriteLine( "{0:X8}", ~x ); // FFFFFFFA  Console.WriteLine( "{0:B32}", ~x ); // 11111111111111111111111111111010  }  }  } |

**=**

The assignment operator (=) stores the value of its right-hand operand in the storage location, property, or indexer denoted by its left-hand operand and returns the value as its result. The operands must be of the same type or the right-hand operand must be implicitly convertible to the type of the left-hand operand.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int x = 5, y = 6;  double z;  x = y; // 'int' to 'int' assignment  z = y; // implicit conversion from 'int' to 'double'  Console.WriteLine( x ); // 6  Console.WriteLine( z ); // 6  }  }  } |

**<**

All numeric and enumeration types define a "less than" relational operator (<) that returns true if the first operand is less than the second, false otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( var i = 0; i < 6; ++i )  Console.WriteLine( i < 3 ); // true true true false false false  }  }  } |

**>**

All numeric and enumeration types define a "greater than" relational operator (>) that returns true if the first operand is greater than the second, false otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( var i = 0; i < 6; ++i )  Console.WriteLine( i > 3 ); // false false false false true true  }  }  } |

**<=**

All numeric and enumeration types define a "less than or equal" relational operator (<=) that returns true if the first operand is less than or equal to the second, false otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( var i = 0; i < 6; ++i )  Console.WriteLine( i <= 3 ); // true true true true false false  }  }  } |

**>=**

All numeric and enumeration types define a "greater than or equal" relational operator, >= that returns true if the first operand is greater than or equal to the second, false otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( var i = 0; i < 6; ++i )  Console.WriteLine( i >= 3 ); // false false false true true true  }  }  } |

**?:**

The conditional operator returns one of two values depending on the value of a Boolean expression

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  for( var i = 0; i < 6; ++i )  Console.Write( i < 3 ? "l" : "b" ); // lllbbb  Console.WriteLine();  }  }  } |

**++**

The increment operator (++) increments its operand by 1.

Prefix increment operation:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int x = 10;  int y = ++x; // the result of the operation is the value of the operand after it has been incremented  Console.WriteLine( "x={0} y={1}", x, y ); // x=11 y=11  y = ++x + ++x;  Console.WriteLine( "x={0} y={1}", x, y ); // x=13 y=25  }  }  } |

Postfix increment operation:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int x = 10;  int y = x++; // the result of the operation is the value of the operand before it has been incremented  Console.WriteLine( "x={0} y={1}", x, y ); // x=11 y=10  y = x++ + x++;  Console.WriteLine( "x={0} y={1}", x, y ); // x=13 y=22  }  }  } |

**--**

The decrement operator (--) decrements its operand by 1.

* Prefix decrement operation:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int x = 10;  int y = --x; // the result of the operation is the value of the operand after it has been decremented  Console.WriteLine( "x={0} y={1}", x, y ); // x=9 y=9  y = --x + --x;  Console.WriteLine( "x={0} y={1}", x, y ); // x=7 y=15  }  }  } |

* Postfix decrement operation:

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  int x = 10;  int y = x--; // the result of the operation is the value of the operand before it has been decremented  Console.WriteLine( "x={0} y={1}", x, y ); // x=9 y=10  y = x-- + x--;  Console.WriteLine( "x={0} y={1}", x, y ); // x=7 y=18  }  }  } |

**&&**

The conditional-AND operator performs a logical-AND of its bool operands, but only evaluates its second operand if necessary.

|  |
| --- |
| using System;  namespace Test {  class Test {  static bool GetTrue() {  Console.WriteLine( thismethod ); // Test.Test.GetTrue()  return true;  }  static bool GetFalse() {  Console.WriteLine( thismethod ); // Test.Test.GetFalse()  return false;  }  [EntryPoint]  public static void Main() {  Console.WriteLine( GetTrue() && GetTrue() ); // Test.Test.GetTrue() Test.Test.GetTrue() true  Console.WriteLine( GetTrue() && GetFalse() ); // Test.Test.GetTrue() Test.Test.GetFalse() false  Console.WriteLine( GetFalse() && GetTrue() ); // Test.Test.GetFalse() false  Console.WriteLine( GetFalse() && GetFalse() ); // Test.Test.GetFalse() false  if( GetTrue() && GetTrue() && GetTrue() )  Console.WriteLine( thismethod ); // Test.Test.GetTrue() Test.Test.GetTrue() Test.Test.GetTrue() Test.Test.Main()  }  }  } |

**||**

The conditional-OR operator performs a logical-OR of its bool operands, but only evaluates its second operand if necessary.

|  |
| --- |
| using System;  namespace Test {  class Test {  static bool GetTrue() {  Console.WriteLine( thismethod ); // Test.Test.GetTrue()  return true;  }  static bool GetFalse() {  Console.WriteLine( thismethod ); // Test.Test.GetFalse()  return false;  }  [EntryPoint]  public static void Main() {  Console.WriteLine( GetTrue() || GetTrue() ); // Test.Test.GetTrue() true  Console.WriteLine( GetTrue() || GetFalse() ); // Test.Test.GetTrue() true  Console.WriteLine( GetFalse() || GetTrue() ); // Test.Test.GetFalse() Test.Test.GetTrue() true  Console.WriteLine( GetFalse() || GetFalse() ); // Test.Test.GetFalse() Test.Test.GetFalse() false  if( GetTrue() || GetFalse() || GetTrue() )  Console.WriteLine( thismethod ); // Test.Test.GetTrue() Test.Test.Main()  }  }  } |

**<<**

The left-shift operator shifts its first operand left by the number of bits specified by its second operand.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  Console.WriteLine( x << 1 ); // 8  Console.WriteLine( x << 29 ); // -2147483648  Console.WriteLine( x << 30 ); // 0  }  }  } |

* Shift operations never cause overflows
* Sign of first operand not retained, sign bit shifted too.
* Second operand not masked and result is platform/codegen-dependent. LLVM can silently drop shift instructions with incorrect shift value after Constant Propagation pass.
* On **x86** platform last 5 bits ( cl & 31 ) are used to compute result

|  |
| --- |
| mov eax, 1  mov ecx, 33  shl al, cl // NOTE: al == 2 ! |

**>>**

The right-shift operator shifts its first operand right by the number of bits specified by its second operand.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  Console.WriteLine( x >> 1 ); // 2  Console.WriteLine( x >> 2 ); // 1  Console.WriteLine( x >> 3 ); // 0  x = -1;  Console.WriteLine( x >> 1 ); // 2147483647, since sign bit shifted too  }  }  } |

* Shift operations never cause overflows
* Sign of first operand not retained, sign bit shifted too.
* For arithmetic shift right use System.BitOperations.**SignedShiftRight**(). Note **SignedShiftRight**() divides a signed number by a power of two, rounding towards negative infinity. However, the **/** operator divides a signed number, rounding towards zero.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  Console.WriteLine( System.BitOperations.SignedShiftRight( 5, 1 ) ); // 2  Console.WriteLine( 5 / 2 ); // 2  Console.WriteLine( System.BitOperations.SignedShiftRight( -5, 1 ) ); // -3  Console.WriteLine( -5 / 2 ); // -2  }  }  } |

* Second operand not masked and result is platform/codegen-dependent. LLVM can silently drop shift instructions with incorrect shift value after Constant Propagation pass.
* On **x86** platform last 5 bits ( cl & 31 ) are used to compute result

|  |
| --- |
| mov eax, 2  mov ecx, 33  shr al, cl // NOTE: al == 1 ! |

**<<<**

The left-rotate operator rotates its first operand left by the number of bits specified by its second operand.

* Shift operations never cause overflows
* Sign of first operand not retained, sign bit shifted too.
* Second operand not masked and result is platform/codegen-dependent. LLVM can silently drop shift instructions with incorrect shift value after Constant Propagation pass.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  sbyte s8 = 1;  byte u8 = 1;  short s16 = 1;  ushort u16 = 1;  int s32 = 1;  uint u32 = 1;  long s64 = 1;  ulong u64 = 1;  for( var i = 0; i < 17; ++i ) {  Console.WriteLine( "========================== 1 <<< {0} ==========================", i );  Console.WriteLine( "sbyte: D={0,4} X={0:X2} B={0:B8}", s8 <<< ( sbyte ) i );  Console.WriteLine( "byte: D={0,4} X={0:X2} B={0:B8}", u8 <<< ( byte ) i );  Console.WriteLine( "short: D={0,6} X={0:X4} B={0:B16}", s16 <<< ( short ) i );  Console.WriteLine( "ushort: D={0,6} X={0:X4} B={0:B16}", u16 <<< ( ushort ) i );  Console.WriteLine( "int: D={0,11} X={0:X8} B={0:B32}", s32 <<< ( int ) i );  Console.WriteLine( "uint: D={0,11} X={0:X8} B={0:B32}", u32 <<< ( uint ) i );  Console.WriteLine( "long: D={0,20} X={0:X16} B={0:B64}", s64 <<< ( long ) i );  Console.WriteLine( "ulong: D={0,20} X={0:X16} B={0:B64}", u64 <<< ( ulong ) i );  }  }  }  } |

**>>>**

The right-rotate operator rotates its first operand right by the number of bits specified by its second operand.

* Shift operations never cause overflows
* Sign of first operand not retained.
* Second operand not masked and result is platform/codegen-dependent. LLVM can silently drop shift instructions with incorrect shift value after Constant Propagation pass.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  sbyte s8 = 2;  byte u8 = 2;  short s16 = 2;  ushort u16 = 2;  int s32 = 2;  uint u32 = 2;  long s64 = 2;  ulong u64 = 2;  for( var i = 0; i < 17; ++i ) {  Console.WriteLine( "========================== 2 >>> {0} ==========================", i );  Console.WriteLine( "sbyte: D={0,4} X={0:X2} B={0:B8}", s8 >>> ( sbyte ) i );  Console.WriteLine( "byte: D={0,4} X={0:X2} B={0:B8}", u8 >>> ( byte ) i );  Console.WriteLine( "short: D={0,6} X={0:X4} B={0:B16}", s16 >>> ( short ) i );  Console.WriteLine( "ushort: D={0,6} X={0:X4} B={0:B16}", u16 >>> ( ushort ) i );  Console.WriteLine( "int: D={0,11} X={0:X8} B={0:B32}", s32 >>> ( int ) i );  Console.WriteLine( "uint: D={0,11} X={0:X8} B={0:B32}", u32 >>> ( uint ) i );  Console.WriteLine( "long: D={0,20} X={0:X16} B={0:B64}", s64 >>> ( long ) i );  Console.WriteLine( "ulong: D={0,20} X={0:X16} B={0:B64}", u64 >>> ( ulong ) i );  }  }  }  } |

**==**

The equality operator returns true if its operands are equal, false otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var hello = new string( "Hello" );  Console.WriteLine( "Hello" == "Hello" ); // true, compile-time constant expression  Console.WriteLine( hello == "Hello" ); // true, since overloaded operator == returns true  Console.WriteLine( ( RuntimeObjectBase ) hello == ( RuntimeObjectBase ) "Hello" ); // false, object adresses not equal  }  }  } |

**!=**

The inequality operator returns false if its operands are equal, true otherwise.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  var hello = new string( "Hello" );  Console.WriteLine( "Hello" != "Hello" ); // false, compile-time constant expression  Console.WriteLine( hello != "Hello" ); // false, since overloaded operator != returns true  Console.WriteLine( ( RuntimeObjectBase ) hello != ( RuntimeObjectBase ) "Hello" ); // true, object adresses not equal  }  }  } |

**+=**

The addition assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 10;  x += 20; // x = x + 20  Console.WriteLine( x ); // 30  Console.WriteLine( ( x += 10 ) == 40 ); // true  Console.WriteLine( ( x += 10 ).ToString() ); // 50  }  }  } |

**-=**

The subtraction assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 50;  x -= 20; // x = x - 20  Console.WriteLine( x ); // 30  Console.WriteLine( ( x -= 10 ) == 20 ); // true  Console.WriteLine( ( x -= 10 ).ToString() ); // 10  }  }  } |

**\*=**

The binary multiplication assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 11;  x \*= 3; // x = x \* 3  Console.WriteLine( x ); // 33  Console.WriteLine( ( x \*= 3 ) == 99 ); // true  Console.WriteLine( typeof( x \*= 100 ) ); // System.LittleEndianInt32  Console.WriteLine( sizeof( x \*= 100 ) ); // 4  Console.WriteLine( x ); // 99, since typeof() & sizeof() only evaluate expression type/size  }  }  } |

**/=**

The division assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 99;  x /= 3; // x = x / 3  Console.WriteLine( x ); // 33  Console.WriteLine( ( x /= 3 ) + ( x /= 11 ) ); // 12  }  }  } |

**%=**

The modulus assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 99;  x %= 25; // x = x % 25  Console.WriteLine( x ); // 24  Console.WriteLine( x %= x %= 7 ); // 0  }  }  } |

**&=**

The AND assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 7;  x &= 5; // x = x & 5  Console.WriteLine( x ); // 5  Console.WriteLine( x &= x &= 4 ); // 4  }  }  } |

**|=**

The OR assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  x |= 1; // x = x | 1  Console.WriteLine( x ); // 5  Console.WriteLine( x |= 7 ); // 7  }  }  } |

**^=**

The exclusive-OR assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  x ^= 1; // x = x ^ 1  Console.WriteLine( x ); // 5  Console.WriteLine( x ^= 7 ); // 2  Console.WriteLine( x ^= 7 ); // 5  Console.WriteLine( x ^= x ^= 7 ); // 7  }  }  } |

**<<=**

The left-shift assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  x <<= 1; // x = x << 1  Console.WriteLine( x ); // 8  Console.WriteLine( x <<= 1 ); // 16  Console.WriteLine( x ); // 16  Console.WriteLine( x <<= 28 ); // 0  }  }  } |

**>>=**

The right-shift assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  private static void Main() {  int x = 4;  x >>= 1; // x = x >> 1  Console.WriteLine( x ); // 2  Console.WriteLine( x >>= 1 ); // 1  Console.WriteLine( x ); // 1  Console.WriteLine( x >>= 1 ); // 0  }  }  } |

**<<<=**

The left-rotate assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  sbyte s8 = 1;  byte u8 = 1;  short s16 = 1;  ushort u16 = 1;  int s32 = 1;  uint u32 = 1;  long s64 = 1;  ulong u64 = 1;  for( var i = 0; i < 17; ++i ) {  Console.WriteLine( "========================== 1 <<< {0} ==========================", i );  Console.WriteLine( "sbyte: D={0,4} X={0:X2} B={0:B8}", s8 );  Console.WriteLine( "byte: D={0,4} X={0:X2} B={0:B8}", u8 );  Console.WriteLine( "short: D={0,6} X={0:X4} B={0:B16}", s16 );  Console.WriteLine( "ushort: D={0,6} X={0:X4} B={0:B16}", u16 );  Console.WriteLine( "int: D={0,11} X={0:X8} B={0:B32}", s32 );  Console.WriteLine( "uint: D={0,11} X={0:X8} B={0:B32}", u32 );  Console.WriteLine( "long: D={0,20} X={0:X16} B={0:B64}", s64 );  Console.WriteLine( "ulong: D={0,20} X={0:X16} B={0:B64}", u64 );  s8 <<<= 1;  u8 <<<= 1;  s16 <<<= 1;  u16 <<<= 1;  s32 <<<= 1;  u32 <<<= 1;  s64 <<<= 1;  u64 <<<= 1;  }  }  }  } |

|  |
| --- |
| ========================== 1 <<< 0 ==========================  sbyte: D= 1 X=01 B=00000001  byte: D= 1 X=01 B=00000001  short: D= 1 X=0001 B=0000000000000001  ushort: D= 1 X=0001 B=0000000000000001  int: D= 1 X=00000001 B=00000000000000000000000000000001  uint: D= 1 X=00000001 B=00000000000000000000000000000001  long: D= 1 X=0000000000000001 B=0000000000000000000000000000000000000000000000000000000000000001  ulong: D= 1 X=0000000000000001 B=0000000000000000000000000000000000000000000000000000000000000001  ========================== 1 <<< 1 ==========================  sbyte: D= 2 X=02 B=00000010  byte: D= 2 X=02 B=00000010  short: D= 2 X=0002 B=0000000000000010  ushort: D= 2 X=0002 B=0000000000000010  int: D= 2 X=00000002 B=00000000000000000000000000000010  uint: D= 2 X=00000002 B=00000000000000000000000000000010  long: D= 2 X=0000000000000002 B=0000000000000000000000000000000000000000000000000000000000000010  ulong: D= 2 X=0000000000000002 B=0000000000000000000000000000000000000000000000000000000000000010  ========================== 1 <<< 2 ==========================  sbyte: D= 4 X=04 B=00000100  byte: D= 4 X=04 B=00000100  short: D= 4 X=0004 B=0000000000000100  ushort: D= 4 X=0004 B=0000000000000100  int: D= 4 X=00000004 B=00000000000000000000000000000100  uint: D= 4 X=00000004 B=00000000000000000000000000000100  long: D= 4 X=0000000000000004 B=0000000000000000000000000000000000000000000000000000000000000100  ulong: D= 4 X=0000000000000004 B=0000000000000000000000000000000000000000000000000000000000000100  ========================== 1 <<< 3 ==========================  sbyte: D= 8 X=08 B=00001000  byte: D= 8 X=08 B=00001000  short: D= 8 X=0008 B=0000000000001000  ushort: D= 8 X=0008 B=0000000000001000  int: D= 8 X=00000008 B=00000000000000000000000000001000  uint: D= 8 X=00000008 B=00000000000000000000000000001000  long: D= 8 X=0000000000000008 B=0000000000000000000000000000000000000000000000000000000000001000  ulong: D= 8 X=0000000000000008 B=0000000000000000000000000000000000000000000000000000000000001000  ========================== 1 <<< 4 ==========================  sbyte: D= 16 X=10 B=00010000  byte: D= 16 X=10 B=00010000  short: D= 16 X=0010 B=0000000000010000  ushort: D= 16 X=0010 B=0000000000010000  int: D= 16 X=00000010 B=00000000000000000000000000010000  uint: D= 16 X=00000010 B=00000000000000000000000000010000  long: D= 16 X=0000000000000010 B=0000000000000000000000000000000000000000000000000000000000010000  ulong: D= 16 X=0000000000000010 B=0000000000000000000000000000000000000000000000000000000000010000  ========================== 1 <<< 5 ==========================  sbyte: D= 32 X=20 B=00100000  byte: D= 32 X=20 B=00100000  short: D= 32 X=0020 B=0000000000100000  ushort: D= 32 X=0020 B=0000000000100000  int: D= 32 X=00000020 B=00000000000000000000000000100000  uint: D= 32 X=00000020 B=00000000000000000000000000100000  long: D= 32 X=0000000000000020 B=0000000000000000000000000000000000000000000000000000000000100000  ulong: D= 32 X=0000000000000020 B=0000000000000000000000000000000000000000000000000000000000100000  ========================== 1 <<< 6 ==========================  sbyte: D= 64 X=40 B=01000000  byte: D= 64 X=40 B=01000000  short: D= 64 X=0040 B=0000000001000000  ushort: D= 64 X=0040 B=0000000001000000  int: D= 64 X=00000040 B=00000000000000000000000001000000  uint: D= 64 X=00000040 B=00000000000000000000000001000000  long: D= 64 X=0000000000000040 B=0000000000000000000000000000000000000000000000000000000001000000  ulong: D= 64 X=0000000000000040 B=0000000000000000000000000000000000000000000000000000000001000000  ========================== 1 <<< 7 ==========================  sbyte: D=-128 X=80 B=10000000  byte: D= 128 X=80 B=10000000  short: D= 128 X=0080 B=0000000010000000  ushort: D= 128 X=0080 B=0000000010000000  int: D= 128 X=00000080 B=00000000000000000000000010000000  uint: D= 128 X=00000080 B=00000000000000000000000010000000  long: D= 128 X=0000000000000080 B=0000000000000000000000000000000000000000000000000000000010000000  ulong: D= 128 X=0000000000000080 B=0000000000000000000000000000000000000000000000000000000010000000  ========================== 1 <<< 8 ==========================  sbyte: D= 1 X=01 B=00000001  byte: D= 1 X=01 B=00000001  short: D= 256 X=0100 B=0000000100000000  ushort: D= 256 X=0100 B=0000000100000000  int: D= 256 X=00000100 B=00000000000000000000000100000000  uint: D= 256 X=00000100 B=00000000000000000000000100000000  long: D= 256 X=0000000000000100 B=0000000000000000000000000000000000000000000000000000000100000000  ulong: D= 256 X=0000000000000100 B=0000000000000000000000000000000000000000000000000000000100000000  ========================== 1 <<< 9 ==========================  sbyte: D= 2 X=02 B=00000010  byte: D= 2 X=02 B=00000010  short: D= 512 X=0200 B=0000001000000000  ushort: D= 512 X=0200 B=0000001000000000  int: D= 512 X=00000200 B=00000000000000000000001000000000  uint: D= 512 X=00000200 B=00000000000000000000001000000000  long: D= 512 X=0000000000000200 B=0000000000000000000000000000000000000000000000000000001000000000  ulong: D= 512 X=0000000000000200 B=0000000000000000000000000000000000000000000000000000001000000000  ========================== 1 <<< 10 ==========================  sbyte: D= 4 X=04 B=00000100  byte: D= 4 X=04 B=00000100  short: D= 1024 X=0400 B=0000010000000000  ushort: D= 1024 X=0400 B=0000010000000000  int: D= 1024 X=00000400 B=00000000000000000000010000000000  uint: D= 1024 X=00000400 B=00000000000000000000010000000000  long: D= 1024 X=0000000000000400 B=0000000000000000000000000000000000000000000000000000010000000000  ulong: D= 1024 X=0000000000000400 B=0000000000000000000000000000000000000000000000000000010000000000  ========================== 1 <<< 11 ==========================  sbyte: D= 8 X=08 B=00001000  byte: D= 8 X=08 B=00001000  short: D= 2048 X=0800 B=0000100000000000  ushort: D= 2048 X=0800 B=0000100000000000  int: D= 2048 X=00000800 B=00000000000000000000100000000000  uint: D= 2048 X=00000800 B=00000000000000000000100000000000  long: D= 2048 X=0000000000000800 B=0000000000000000000000000000000000000000000000000000100000000000  ulong: D= 2048 X=0000000000000800 B=0000000000000000000000000000000000000000000000000000100000000000  ========================== 1 <<< 12 ==========================  sbyte: D= 16 X=10 B=00010000  byte: D= 16 X=10 B=00010000  short: D= 4096 X=1000 B=0001000000000000  ushort: D= 4096 X=1000 B=0001000000000000  int: D= 4096 X=00001000 B=00000000000000000001000000000000  uint: D= 4096 X=00001000 B=00000000000000000001000000000000  long: D= 4096 X=0000000000001000 B=0000000000000000000000000000000000000000000000000001000000000000  ulong: D= 4096 X=0000000000001000 B=0000000000000000000000000000000000000000000000000001000000000000  ========================== 1 <<< 13 ==========================  sbyte: D= 32 X=20 B=00100000  byte: D= 32 X=20 B=00100000  short: D= 8192 X=2000 B=0010000000000000  ushort: D= 8192 X=2000 B=0010000000000000  int: D= 8192 X=00002000 B=00000000000000000010000000000000  uint: D= 8192 X=00002000 B=00000000000000000010000000000000  long: D= 8192 X=0000000000002000 B=0000000000000000000000000000000000000000000000000010000000000000  ulong: D= 8192 X=0000000000002000 B=0000000000000000000000000000000000000000000000000010000000000000  ========================== 1 <<< 14 ==========================  sbyte: D= 64 X=40 B=01000000  byte: D= 64 X=40 B=01000000  short: D= 16384 X=4000 B=0100000000000000  ushort: D= 16384 X=4000 B=0100000000000000  int: D= 16384 X=00004000 B=00000000000000000100000000000000  uint: D= 16384 X=00004000 B=00000000000000000100000000000000  long: D= 16384 X=0000000000004000 B=0000000000000000000000000000000000000000000000000100000000000000  ulong: D= 16384 X=0000000000004000 B=0000000000000000000000000000000000000000000000000100000000000000  ========================== 1 <<< 15 ==========================  sbyte: D=-128 X=80 B=10000000  byte: D= 128 X=80 B=10000000  short: D=-32768 X=8000 B=1000000000000000  ushort: D= 32768 X=8000 B=1000000000000000  int: D= 32768 X=00008000 B=00000000000000001000000000000000  uint: D= 32768 X=00008000 B=00000000000000001000000000000000  long: D= 32768 X=0000000000008000 B=0000000000000000000000000000000000000000000000001000000000000000  ulong: D= 32768 X=0000000000008000 B=0000000000000000000000000000000000000000000000001000000000000000  ========================== 1 <<< 16 ==========================  sbyte: D= 1 X=01 B=00000001  byte: D= 1 X=01 B=00000001  short: D= 1 X=0001 B=0000000000000001  ushort: D= 1 X=0001 B=0000000000000001  int: D= 65536 X=00010000 B=00000000000000010000000000000000  uint: D= 65536 X=00010000 B=00000000000000010000000000000000  long: D= 65536 X=0000000000010000 B=0000000000000000000000000000000000000000000000010000000000000000  ulong: D= 65536 X=0000000000010000 B=0000000000000000000000000000000000000000000000010000000000000000 |

**>>>=**

The right-rotate assignment operator.

|  |
| --- |
| using System;  namespace Test {  class Test {  [EntryPoint]  public static void Main() {  sbyte s8 = 2;  byte u8 = 2;  short s16 = 2;  ushort u16 = 2;  int s32 = 2;  uint u32 = 2;  long s64 = 2;  ulong u64 = 2;  for( var i = 0; i < 17; ++i ) {  Console.WriteLine( "========================== 2 >>> {0} ==========================", i );  Console.WriteLine( "sbyte: D={0,4} X={0:X2} B={0:B8}", s8 );  Console.WriteLine( "byte: D={0,4} X={0:X2} B={0:B8}", u8 );  Console.WriteLine( "short: D={0,6} X={0:X4} B={0:B16}", s16 );  Console.WriteLine( "ushort: D={0,6} X={0:X4} B={0:B16}", u16 );  Console.WriteLine( "int: D={0,11} X={0:X8} B={0:B32}", s32 );  Console.WriteLine( "uint: D={0,11} X={0:X8} B={0:B32}", u32 );  Console.WriteLine( "long: D={0,20} X={0:X16} B={0:B64}", s64 );  Console.WriteLine( "ulong: D={0,20} X={0:X16} B={0:B64}", u64 );  s8 >>>= 1;  u8 >>>= 1;  s16 >>>= 1;  u16 >>>= 1;  s32 >>>= 1;  u32 >>>= 1;  s64 >>>= 1;  u64 >>>= 1;  }  }  }  } |

|  |
| --- |
| ========================== 2 >>> 0 ==========================  sbyte: D= 2 X=02 B=00000010  byte: D= 2 X=02 B=00000010  short: D= 2 X=0002 B=0000000000000010  ushort: D= 2 X=0002 B=0000000000000010  int: D= 2 X=00000002 B=00000000000000000000000000000010  uint: D= 2 X=00000002 B=00000000000000000000000000000010  long: D= 2 X=0000000000000002 B=0000000000000000000000000000000000000000000000000000000000000010  ulong: D= 2 X=0000000000000002 B=0000000000000000000000000000000000000000000000000000000000000010  ========================== 2 >>> 1 ==========================  sbyte: D= 1 X=01 B=00000001  byte: D= 1 X=01 B=00000001  short: D= 1 X=0001 B=0000000000000001  ushort: D= 1 X=0001 B=0000000000000001  int: D= 1 X=00000001 B=00000000000000000000000000000001  uint: D= 1 X=00000001 B=00000000000000000000000000000001  long: D= 1 X=0000000000000001 B=0000000000000000000000000000000000000000000000000000000000000001  ulong: D= 1 X=0000000000000001 B=0000000000000000000000000000000000000000000000000000000000000001  ========================== 2 >>> 2 ==========================  sbyte: D=-128 X=80 B=10000000  byte: D= 128 X=80 B=10000000  short: D=-32768 X=8000 B=1000000000000000  ushort: D= 32768 X=8000 B=1000000000000000  int: D=-2147483648 X=80000000 B=10000000000000000000000000000000  uint: D= 2147483648 X=80000000 B=10000000000000000000000000000000  long: D=-9223372036854775808 X=8000000000000000 B=1000000000000000000000000000000000000000000000000000000000000000  ulong: D= 9223372036854775808 X=8000000000000000 B=1000000000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 3 ==========================  sbyte: D= 64 X=40 B=01000000  byte: D= 64 X=40 B=01000000  short: D= 16384 X=4000 B=0100000000000000  ushort: D= 16384 X=4000 B=0100000000000000  int: D= 1073741824 X=40000000 B=01000000000000000000000000000000  uint: D= 1073741824 X=40000000 B=01000000000000000000000000000000  long: D= 4611686018427387904 X=4000000000000000 B=0100000000000000000000000000000000000000000000000000000000000000  ulong: D= 4611686018427387904 X=4000000000000000 B=0100000000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 4 ==========================  sbyte: D= 32 X=20 B=00100000  byte: D= 32 X=20 B=00100000  short: D= 8192 X=2000 B=0010000000000000  ushort: D= 8192 X=2000 B=0010000000000000  int: D= 536870912 X=20000000 B=00100000000000000000000000000000  uint: D= 536870912 X=20000000 B=00100000000000000000000000000000  long: D= 2305843009213693952 X=2000000000000000 B=0010000000000000000000000000000000000000000000000000000000000000  ulong: D= 2305843009213693952 X=2000000000000000 B=0010000000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 5 ==========================  sbyte: D= 16 X=10 B=00010000  byte: D= 16 X=10 B=00010000  short: D= 4096 X=1000 B=0001000000000000  ushort: D= 4096 X=1000 B=0001000000000000  int: D= 268435456 X=10000000 B=00010000000000000000000000000000  uint: D= 268435456 X=10000000 B=00010000000000000000000000000000  long: D= 1152921504606846976 X=1000000000000000 B=0001000000000000000000000000000000000000000000000000000000000000  ulong: D= 1152921504606846976 X=1000000000000000 B=0001000000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 6 ==========================  sbyte: D= 8 X=08 B=00001000  byte: D= 8 X=08 B=00001000  short: D= 2048 X=0800 B=0000100000000000  ushort: D= 2048 X=0800 B=0000100000000000  int: D= 134217728 X=08000000 B=00001000000000000000000000000000  uint: D= 134217728 X=08000000 B=00001000000000000000000000000000  long: D= 576460752303423488 X=0800000000000000 B=0000100000000000000000000000000000000000000000000000000000000000  ulong: D= 576460752303423488 X=0800000000000000 B=0000100000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 7 ==========================  sbyte: D= 4 X=04 B=00000100  byte: D= 4 X=04 B=00000100  short: D= 1024 X=0400 B=0000010000000000  ushort: D= 1024 X=0400 B=0000010000000000  int: D= 67108864 X=04000000 B=00000100000000000000000000000000  uint: D= 67108864 X=04000000 B=00000100000000000000000000000000  long: D= 288230376151711744 X=0400000000000000 B=0000010000000000000000000000000000000000000000000000000000000000  ulong: D= 288230376151711744 X=0400000000000000 B=0000010000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 8 ==========================  sbyte: D= 2 X=02 B=00000010  byte: D= 2 X=02 B=00000010  short: D= 512 X=0200 B=0000001000000000  ushort: D= 512 X=0200 B=0000001000000000  int: D= 33554432 X=02000000 B=00000010000000000000000000000000  uint: D= 33554432 X=02000000 B=00000010000000000000000000000000  long: D= 144115188075855872 X=0200000000000000 B=0000001000000000000000000000000000000000000000000000000000000000  ulong: D= 144115188075855872 X=0200000000000000 B=0000001000000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 9 ==========================  sbyte: D= 1 X=01 B=00000001  byte: D= 1 X=01 B=00000001  short: D= 256 X=0100 B=0000000100000000  ushort: D= 256 X=0100 B=0000000100000000  int: D= 16777216 X=01000000 B=00000001000000000000000000000000  uint: D= 16777216 X=01000000 B=00000001000000000000000000000000  long: D= 72057594037927936 X=0100000000000000 B=0000000100000000000000000000000000000000000000000000000000000000  ulong: D= 72057594037927936 X=0100000000000000 B=0000000100000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 10 ==========================  sbyte: D=-128 X=80 B=10000000  byte: D= 128 X=80 B=10000000  short: D= 128 X=0080 B=0000000010000000  ushort: D= 128 X=0080 B=0000000010000000  int: D= 8388608 X=00800000 B=00000000100000000000000000000000  uint: D= 8388608 X=00800000 B=00000000100000000000000000000000  long: D= 36028797018963968 X=0080000000000000 B=0000000010000000000000000000000000000000000000000000000000000000  ulong: D= 36028797018963968 X=0080000000000000 B=0000000010000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 11 ==========================  sbyte: D= 64 X=40 B=01000000  byte: D= 64 X=40 B=01000000  short: D= 64 X=0040 B=0000000001000000  ushort: D= 64 X=0040 B=0000000001000000  int: D= 4194304 X=00400000 B=00000000010000000000000000000000  uint: D= 4194304 X=00400000 B=00000000010000000000000000000000  long: D= 18014398509481984 X=0040000000000000 B=0000000001000000000000000000000000000000000000000000000000000000  ulong: D= 18014398509481984 X=0040000000000000 B=0000000001000000000000000000000000000000000000000000000000000000  ========================== 2 >>> 12 ==========================  sbyte: D= 32 X=20 B=00100000  byte: D= 32 X=20 B=00100000  short: D= 32 X=0020 B=0000000000100000  ushort: D= 32 X=0020 B=0000000000100000  int: D= 2097152 X=00200000 B=00000000001000000000000000000000  uint: D= 2097152 X=00200000 B=00000000001000000000000000000000  long: D= 9007199254740992 X=0020000000000000 B=0000000000100000000000000000000000000000000000000000000000000000  ulong: D= 9007199254740992 X=0020000000000000 B=0000000000100000000000000000000000000000000000000000000000000000  ========================== 2 >>> 13 ==========================  sbyte: D= 16 X=10 B=00010000  byte: D= 16 X=10 B=00010000  short: D= 16 X=0010 B=0000000000010000  ushort: D= 16 X=0010 B=0000000000010000  int: D= 1048576 X=00100000 B=00000000000100000000000000000000  uint: D= 1048576 X=00100000 B=00000000000100000000000000000000  long: D= 4503599627370496 X=0010000000000000 B=0000000000010000000000000000000000000000000000000000000000000000  ulong: D= 4503599627370496 X=0010000000000000 B=0000000000010000000000000000000000000000000000000000000000000000  ========================== 2 >>> 14 ==========================  sbyte: D= 8 X=08 B=00001000  byte: D= 8 X=08 B=00001000  short: D= 8 X=0008 B=0000000000001000  ushort: D= 8 X=0008 B=0000000000001000  int: D= 524288 X=00080000 B=00000000000010000000000000000000  uint: D= 524288 X=00080000 B=00000000000010000000000000000000  long: D= 2251799813685248 X=0008000000000000 B=0000000000001000000000000000000000000000000000000000000000000000  ulong: D= 2251799813685248 X=0008000000000000 B=0000000000001000000000000000000000000000000000000000000000000000  ========================== 2 >>> 15 ==========================  sbyte: D= 4 X=04 B=00000100  byte: D= 4 X=04 B=00000100  short: D= 4 X=0004 B=0000000000000100  ushort: D= 4 X=0004 B=0000000000000100  int: D= 262144 X=00040000 B=00000000000001000000000000000000  uint: D= 262144 X=00040000 B=00000000000001000000000000000000  long: D= 1125899906842624 X=0004000000000000 B=0000000000000100000000000000000000000000000000000000000000000000  ulong: D= 1125899906842624 X=0004000000000000 B=0000000000000100000000000000000000000000000000000000000000000000  ========================== 2 >>> 16 ==========================  sbyte: D= 2 X=02 B=00000010  byte: D= 2 X=02 B=00000010  short: D= 2 X=0002 B=0000000000000010  ushort: D= 2 X=0002 B=0000000000000010  int: D= 131072 X=00020000 B=00000000000000100000000000000000  uint: D= 131072 X=00020000 B=00000000000000100000000000000000  long: D= 562949953421312 X=0002000000000000 B=0000000000000010000000000000000000000000000000000000000000000000  ulong: D= 562949953421312 X=0002000000000000 B=0000000000000010000000000000000000000000000000000000000000000000 |

**->**

The **->** operator combines pointer dereferencing and member access.

|  |
| --- |
| using System;  namespace Test {  class D1 {  struct S1 { int SomeField; }  [EntryPoint]  public static void Main() {  var s1 = new S1 { SomeField = 10 };  var p1 = &s1;  Assert.IsTrue( s1.SomeField == ( \*p1 ).SomeField );  Assert.IsTrue( s1.SomeField == p1->SomeField );  }  }  } |

The **->** operator cannot be overloaded.

**??**

The ?? operator returns the left-hand operand if it is not null, or else it returns the right operand.

|  |
| --- |
| using System;  namespace Test {  class D1 {  static int\* Foo( int param ) { return param > 0 ? ( int\* ) 0xA000 : null; }  [EntryPoint]  public static void Main() {  int\* ptr1 = Foo( 0 ), ptr2 = Foo( 1 );  int\* ptr3 = ptr1 ?? ptr2; // ptr3 = ptr1 != null ? ptr1 : ptr2;  Assert.IsTrue( ptr3 == ( int\* ) 0xA000 );  Console.WriteLine( null ?? null ?? "not null" ); // not null  }  }  } |