Swift Standard Library Operators

This chapter describes the operator declarations and corresponding global operator functions defined in the Swift standard library.

Overview

The tables below list the operators declared in the standard library, including their associativity and precedence group. Table 1 lists the prefix operators and Table 2 lists the infix operators. For more information about operation declarations, see Operator Declaration in The Swift Programming Language (Swift 3.0.1).

Table 1 Prefix operators

Operator	Description
!	Logical NOT
~	Bitwise NOT
+	Unary plus
-	Unary minus

Table 2 Infix operators

Operator	Description	Associativity	Precedence group
<<	Bitwise left shift	None	Bitwise shift
>>	Bitwise right shift	None	Bitwise shift
*	Multiply	Left associative	Multiplication
/	Divide	Left associative	Multiplication
%	Remainder	Left associative	Multiplication
&*	Multiply, ignoring overflow	Left associative	Multiplication
&	Bitwise AND	Left associative	Multiplication
+	Add	Left associative	Addition
-	Subtract	Left associative	Addition
&+	Add with overflow	Left associative	Addition

&-	Subtract with overflow	Left associative	Addition
|	Bitwise OR	Left associative	Addition
^	Bitwise XOR	Left associative	Addition
..<	Half-open range	None	Range formation
...	Closed range	None	Range formation
is	Type check	Left associative	Casting
as, as?, **and** as!	Type cast	Left associative	Casting
??	Nil coalescing	Right associative	Nil coalescing
<	Less than	None	Comparison
<=	Less than or equal	None	Comparison
>	Greater than	None	Comparison
>=	Greater than or equal	None	Comparison
==	Equal	None	Comparison
!=	Not equal	None	Comparison
===	Identical	None	Comparison
!==	Not identical	None	Comparison
~=	Pattern match	None	Comparison
&&	Logical AND	Left associative	Logical conjunction
||	Logical OR	Left associative	Logical disjunction
?:	Ternary conditional	Right associative	Ternary
=	Assign	Right associative	Assignment
*=	Multiply and assign	Right associative	Assignment
/=	Divide and assign	Right associative	Assignment
%=	Remainder and assign	Right associative	Assignment
+=	Add and assign	Right associative	Assignment
-=	Subtract and assign	Right associative	Assignment
<<=	Left bit shift and assign	Right associative	Assignment
>>=	Right bit shift and assign	Right associative	Assignment
&=	Bitwise AND and assign	Right associative	Assignment
|=	Bitwise OR and assign	Right associative	Assignment

^=	Bitwise XOR and assign	Right associative	Assignment

Symbols

Operators

`func !=<Element>(ArraySlice<Element>, ArraySlice<Element>)`

Returns `true` if the arrays do not contain the same elements.

`func !=<T>(T, T)`

Returns a Boolean value indicating whether the two arguments are not equal.

`func !=<T>(T, T)`

Returns a Boolean value indicating whether two values are not equal.

`func !=(Int8, Int8)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<Element>(Array<Element>, Array<Element>)`

Returns `true` if the arrays do not contain the same elements.

`func !=(Int16, Int16)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=(Int64, Int64)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<A, B>((A, B), (A, B))`

Returns a Boolean value indicating whether any corresponding components of the two tuples are not equal.

`func !=<A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))`

Returns a Boolean value indicating whether any corresponding components of the two tuples are not equal.

`func !=(UInt16, UInt16)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<Element>(ContiguousArray<Element>, ContiguousArray<Element>)`

Returns `true` if the arrays do not contain the same elements.

`func !=(UInt, UInt)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<T>(T, T)`

Returns a Boolean value indicating whether the two arguments are not equal.

`func !=<A, B, C>((A, B, C), (A, B, C))`

Returns a Boolean value indicating whether any corresponding components of the two tuples are not equal.

`func !=<T>(T?, T?)`

`func !=(UInt8, UInt8)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=(UInt32, UInt32)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=(UInt64, UInt64)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<T>(T?, _OptionalNilComparisonType)`

`func !=(Any.Type?, Any.Type?)`

Returns `false` iff `t0` is identical to `t1`; i.e. if they are both `nil` or they both represent the same type.

`func !=(Int32, Int32)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !=<T>(_OptionalNilComparisonType, T?)`

`func !=<A, B, C, D>((A, B, C, D), (A, B, C, D))`

Returns a Boolean value indicating whether any corresponding components of the two tuples are not equal.

`func !=<A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`

Returns a Boolean value indicating whether any corresponding components of the two tuples are not equal.

`func !=(Int, Int)`

Returns a Boolean value that indicates whether the two arguments have unequal values.

`func !==(AnyObject?, AnyObject?)`

Returns a Boolean value indicating whether two references point to different object instances.

`func %(Int8, Int8)`

`func %(Int16, Int16)`

`func %(UInt8, UInt8)`

`func %(UInt, UInt)`

`func %(Int, Int)`

`func %(Int64, Int64)`

`func %(UInt32, UInt32)`

`func %(Int32, Int32)`

`func %(UInt16, UInt16)`

`func %(UInt64, UInt64)`

`func %<T>(T, T)`

Divides `lhs` and `rhs`, returning the remainder and trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func %=<T>(inout T, T)`

Divides `lhs` and `rhs` and stores the remainder in `lhs`, trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func &(Int, Int)`

Returns the intersection of bits set in the two arguments.

`func &(UInt8, UInt8)`

Returns the intersection of bits set in the two arguments.

`func &(UInt16, UInt16)`

Returns the intersection of bits set in the two arguments.

`func &(UInt, UInt)`

Returns the intersection of bits set in the two arguments.

`func &(UInt32, UInt32)`

Returns the intersection of bits set in the two arguments.

`func &(Int32, Int32)`

Returns the intersection of bits set in the two arguments.

`func &(UInt64, UInt64)`

Returns the intersection of bits set in the two arguments.

`func &(Int64, Int64)`

Returns the intersection of bits set in the two arguments.

`func &(Int16, Int16)`

Returns the intersection of bits set in the two arguments.

`func &(Int8, Int8)`

Returns the intersection of bits set in the two arguments.

`func &*<T>(T, T)`

Multiplies `lhs` and `rhs`, silently discarding any overflow.

`func &+<T>(T, T)`

Adds `lhs` and `rhs`, silently discarding any overflow.

`func &-<T>(T, T)`

Subtracts `lhs` and `rhs`, silently discarding any overflow.

`func &=(inout UInt16, UInt16)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout Int8, Int8)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout Int32, Int32)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout UInt8, UInt8)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout UInt32, UInt32)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout UInt, UInt)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout Int16, Int16)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout Int, Int)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout UInt64, UInt64)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=(inout Int64, Int64)`

Calculates the intersection of bits set in the two arguments and stores the result in the first argument.

`func &=<T>(inout T, T)`

Calculates the intersections of bits sets in the two arguments and stores the result in the first argument.

`func *(Int64, Int64)`

`func *(Int32, Int32)`

`func *(UInt, UInt)`

`func *(Int16, Int16)`

`func *(UInt32, UInt32)`

`func *(UInt16, UInt16)`

`func *(Int8, Int8)`

`func *(UInt8, UInt8)`

`func *(Double, Double)`

`func *(UInt64, UInt64)`

`func *(Int, Int)`

`func *(Float, Float)`

`func *<T>(T, T)`

Multiplies `lhs` and `rhs`, returning the result and trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func *(Float80, Float80)`

`func *=(inout Int, Int)`

`func *=(inout Int16, Int16)`

`func *=(inout UInt8, UInt8)`

`func *=(inout UInt16, UInt16)`

`func *=(inout Double, Double)`

`func *=(inout Int64, Int64)`

`func *=(inout UInt, UInt)`

`func *=(inout Int8, Int8)`

`func *=(inout UInt32, UInt32)`

`func *=(inout Int32, Int32)`

`func *=(inout Float, Float)`

`func *=(inout UInt64, UInt64)`

`func *=<T>(inout T, T)`

Multiplies `lhs` and `rhs` and stores the result in `lhs`, trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func *=(inout Float80, Float80)`

`func +(Float)`

`func +(Double)`

`func +<T>(T)`

`func +(Float80)`

`func +(UInt64, UInt64)`

`func +(UInt64, UInt64)`

`func +(UInt8, UInt8)`

`func +<Pointee>(Int, UnsafeMutablePointer<Pointee>)`

`func +(Int8, Int8)`

`func +<Pointee>(Int, UnsafePointer<Pointee>)`

`func +<Pointee>(UnsafePointer<Pointee>, Int)`

`func +<T>(T, T.Stride)`

`func +(Float, Float)`

`func +(Int, Int)`

`func +(Int16, Int16)`

`func +(UInt16, UInt16)`

`func +<T>(T.Stride, T)`

`func +(Int64, Int64)`

`func +<Pointee>(UnsafeMutablePointer<Pointee>, Int)`

`func +(UInt32, UInt32)`

`func +(Double, Double)`

`func +(Int32, Int32)`

`func +(UInt, UInt)`

`func +<C, S>(S, C)`

Creates a new collection by concatenating the elements of a sequence and a collection.

`func +<C, S>(C, S)`

Creates a new collection by concatenating the elements of a collection and a sequence.

`func +<T>(T._DisallowMixedSignArithmetic, T)`

`func +<T>(T, T._DisallowMixedSignArithmetic)`

`func +<T>(T, T)`

Adds `lhs` and `rhs`, returning the result and trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func +<RRC1, RRC2>(RRC1, RRC2)`

Creates a new collection by concatenating the elements of two collections.

`func +<T>(T, T)`

`func +(Float80, Float80)`

`func +=<Pointee>(inout UnsafeMutablePointer<Pointee>, Int)`

`func +=(inout UInt32, UInt32)`

`func +=(inout UInt64, UInt64)`

`func +=(inout Int8, Int8)`

`func +=(inout Int64, Int64)`

`func +=(inout UInt16, UInt16)`

`func +=<Pointee>(inout UnsafePointer<Pointee>, Int)`

`func +=(inout Double, Double)`

`func +=(inout UInt, UInt)`

`func +=(inout Int16, Int16)`

`func +=(inout Int, Int)`

`func +=<T>(inout T, T.Stride)`

`func +=(inout Int32, Int32)`

`func +=(inout UInt8, UInt8)`

`func +=(inout Float, Float)`

`func +=<T>(inout T, T._DisallowMixedSignArithmetic)`

`func +=<T>(inout T, T)`

Adds `lhs` and `rhs` and stores the result in `lhs`, trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func +=<Element, C>(inout _ContiguousArrayBuffer<Element>, C)`

Append the elements of `rhs` to `lhs`.

`func +=<S>(inout Array<S.Iterator.Element>, S)`
: Appends the elements of a sequence to an array.

`func +=<S>(inout ContiguousArray<S.Iterator.Element>, S)`
: Appends the elements of a sequence to a `ContiguousArray` instance.

`func +=<S>(inout ArraySlice<S.Iterator.Element>, S)`
: Appends the elements of a sequence to an `ArraySlice` instance.

`func +=<C>(inout Array<C.Iterator.Element>, C)`
: Appends the elements of a collection to an array.

`func +=<C>(inout ArraySlice<C.Iterator.Element>, C)`
: Appends the elements of a collection to an `ArraySlice` instance.

`func +=<C>(inout ContiguousArray<C.Iterator.Element>, C)`
: Appends the elements of a collection to a `ContiguousArray` instance.

`func +=<T>(inout T, T)`

`func +=(inout Float80, Float80)`

`func -(Float)`

`func -(Double)`

`func -<T>(T)`

`func -(Float80)`

`func -(UInt, UInt)`

`func -(UInt64, UInt64)`

`func -(UInt8, UInt8)`

`func -<T>(T, T.Stride)`

`func -(Float, Float)`

```
func -<Pointee>(UnsafeMutablePointer<Pointee>, UnsafeMutablePointer<Po
intee>)
```

`func -<T>(T, T._DisallowMixedSignArithmetic)`

`func -(UInt16, UInt16)`

`func -<T>(T, T)`

`func -(Int64, Int64)`

```
func -(Double, Double)
```

```
func -<T>(T, T)
```

```
func -(Int, Int)
```

```
func -(Int8, Int8)
```

```
func -<Pointee>(UnsafeMutablePointer<Pointee>, Int)
```

```
func -(Int16, Int16)
```

```
func -(Int32, Int32)
```

```
func -(UInt32, UInt32)
```

```
func -<Pointee>(UnsafePointer<Pointee>, Int)
```

```
func -<Pointee>(UnsafePointer<Pointee>, UnsafePointer<Pointee>)
```

```
func -<T>(T, T)
```

Subtracts `lhs` and `rhs`, returning the result and trapping in case of arithmetic overflow (except in -Ounchecked builds).

```
func -(Float80, Float80)
```

```
func -=(inout Int16, Int16)
```

```
func -=(inout UInt, UInt)
```

```
func -=(inout UInt16, UInt16)
```

```
func -=(inout UInt32, UInt32)
```

```
func -=(inout UInt8, UInt8)
```

```
func -=<Pointee>(inout UnsafeMutablePointer<Pointee>, Int)
```

```
func -=(inout Int8, Int8)
```

```
func -=<Pointee>(inout UnsafePointer<Pointee>, Int)
```

```
func -=(inout Int, Int)
```

```
func -=(inout UInt64, UInt64)
```

```
func -=(inout Int64, Int64)
```

`func -=(inout Float, Float)`

`func -=(inout Double, Double)`

`func -=<T>(inout T, T.Stride)`

`func -=(inout Int32, Int32)`

`func -=<T>(inout T, T._DisallowMixedSignArithmetic)`

`func -=<T>(inout T, T)`

Subtracts `lhs` and `rhs` and stores the result in `lhs`, trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func -=<T>(inout T, T)`

`func -=(inout Float80, Float80)`

`func ...<Bound>(Bound, Bound)`

Returns a closed range that contains both of its bounds.

`func ...<Bound>(Bound, Bound)`

Returns a countable closed range that contains both of its bounds.

`func ..<<Bound>(Bound, Bound)`

Returns a half-open range that contains its lower bound but not its upper bound.

`func ..<<Bound>(Bound, Bound)`

Returns a countable half-open range that contains its lower bound but not its upper bound.

`func /(UInt32, UInt32)`

`func /(UInt64, UInt64)`

`func /(UInt16, UInt16)`

`func /(Int64, Int64)`

`func /(Int32, Int32)`

`func /(Int8, Int8)`

`func /(Float, Float)`

`func /(Int16, Int16)`

`func /(UInt, UInt)`

`func /(UInt8, UInt8)`

`func /(Double, Double)`

`func /(Int, Int)`

`func /<T>(T, T)`

Divides `lhs` and `rhs`, returning the result and trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func /(Float80, Float80)`

`func /=(inout Double, Double)`

`func /=(inout Float, Float)`

`func /=<T>(inout T, T)`

Divides `lhs` and `rhs` and stores the result in `lhs`, trapping in case of arithmetic overflow (except in -Ounchecked builds).

`func /=(inout Float80, Float80)`

`func <(UInt8, UInt8)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <<Pointee>(UnsafePointer<Pointee>, UnsafePointer<Pointee>)`

`func <(UInt, UInt)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <(Int64, Int64)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <(UInt16, UInt16)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <<T>(T, T)`

Compare two `Strideable`s.

`func <(Int8, Int8)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <(UInt64, UInt64)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <(Int, Int)`

Returns a Boolean value that indicates whether the first argument is less than the second argument.

`func <<A, B, C, D>((A, B, C, D), (A, B, C, D))`

Returns a Boolean value indicating whether the first tuple is ordered before the second in a lexicographical ordering.

`func <<A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`

Returns a Boolean value indicating whether the first tuple is ordered before the second in a lexicographical ordering.

```
func <<A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))
```

Returns a Boolean value indicating whether the first tuple is ordered before the second in a lexicographical ordering.

```
func <(Int32, Int32)
```

Returns a Boolean value that indicates whether the first argument is less than the second argument.

```
func <(Int16, Int16)
```

Returns a Boolean value that indicates whether the first argument is less than the second argument.

```
func <<Pointee>(UnsafeMutablePointer<Pointee>, UnsafeMutablePointer<Po
intee>)
```

```
func <(UInt32, UInt32)
```

Returns a Boolean value that indicates whether the first argument is less than the second argument.

```
func <<A, B>((A, B), (A, B))
```

Returns a Boolean value indicating whether the first tuple is ordered before the second in a lexicographical ordering.

```
func <<A, B, C>((A, B, C), (A, B, C))
```

Returns a Boolean value indicating whether the first tuple is ordered before the second in a lexicographical ordering.

```
func <<T>(T, T)
```

```
func <<T>(T, T)
```

```
func <(UnsafeRawPointer, UnsafeRawPointer)
```

```
func <(UnsafeMutableRawPointer, UnsafeMutableRawPointer)
```

```
func <<(Int32, Int32)
```

```
func <<(UInt32, UInt32)
```

```
func <<(Int, Int)
```

```
func <<(UInt8, UInt8)
```

```
func <<(Int64, Int64)
```

```
func <<(UInt64, UInt64)
```

```
func <<(Int16, Int16)
```

```
func <<(Int8, Int8)
```

`func <<(UInt, UInt)`

`func <<(UInt16, UInt16)`

`func <<=(inout UInt8, UInt8)`

`func <<=(inout UInt32, UInt32)`

`func <<=(inout UInt, UInt)`

`func <<=(inout Int64, Int64)`

`func <<=(inout Int16, Int16)`

`func <<=(inout UInt16, UInt16)`

`func <<=(inout Int32, Int32)`

`func <<=(inout Int, Int)`

`func <<=(inout Int8, Int8)`

`func <<=(inout UInt64, UInt64)`

`func <=(Int64, Int64)`
Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=<A, B, C, D>((A, B, C, D), (A, B, C, D))`
Returns a Boolean value indicating whether the first tuple is ordered before or the same as the second in a lexicographical ordering.

`func <=(UInt, UInt)`
Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=(UInt8, UInt8)`
Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=<A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`
Returns a Boolean value indicating whether the first tuple is ordered before or the same as the second in a lexicographical ordering.

`func <=<A, B>((A, B), (A, B))`
Returns a Boolean value indicating whether the first tuple is ordered before or the same as the second in a lexicographical ordering.

`func <=(Int32, Int32)`
Returns a Boolean value that indicates whether the first argument is less than or equal to the second

argument.

`func <=(Int16, Int16)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=(Int, Int)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=(UInt32, UInt32)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=(UInt64, UInt64)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=<A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))`

Returns a Boolean value indicating whether the first tuple is ordered before or the same as the second in a lexicographical ordering.

`func <=(UInt16, UInt16)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=(Int8, Int8)`

Returns a Boolean value that indicates whether the first argument is less than or equal to the second argument.

`func <=<T>(T, T)`

Returns a Boolean value indicating whether the value of the first argument is less than or equal to that of the second argument.

`func <=<A, B, C>((A, B, C), (A, B, C))`

Returns a Boolean value indicating whether the first tuple is ordered before or the same as the second in a lexicographical ordering.

`func <=<T>(T, T)`

`func <=<T>(T, T)`

`func ==<T>(T, T)`

`func ==<A, B, C>((A, B, C), (A, B, C))`

Returns a Boolean value indicating whether the corresponding components of two tuples are equal.

`func ==(Int32, Int32)`

Returns a Boolean value that indicates whether the two arguments have equal values.

`func ==(UInt8, UInt8)`

Returns a Boolean value that indicates whether the two arguments have equal values.

`func ==<T>(T?, T?)`

```
func ==<A, B, C, D>((A, B, C, D), (A, B, C, D))
```
Returns a Boolean value indicating whether the corresponding components of two tuples are equal.

```
func ==<Pointee>(AutoreleasingUnsafeMutablePointer<Pointee>, Autorelea
singUnsafeMutablePointer<Pointee>)
```

```
func ==(UInt32, UInt32)
```
Returns a Boolean value that indicates whether the two arguments have equal values.

```
func ==<T>(T?, _OptionalNilComparisonType)
```

```
func ==<A, B>((A, B), (A, B))
```
Returns a Boolean value indicating whether the corresponding components of two tuples are equal.

```
func ==(Int8, Int8)
```

Returns a Boolean value that indicates whether the two arguments have equal values.

```
func ==<T>(T, T)
```
Returns a Boolean value indicating whether the two arguments are equal.

```
func ==<A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))
```
Returns a Boolean value indicating whether the corresponding components of two tuples are equal.

```
func ==<Pointee>(UnsafeMutablePointer<Pointee>, UnsafeMutablePointer<P
ointee>)
```

```
func ==<Element>(ContiguousArray<Element>, ContiguousArray<Element>)
```
Returns `true` if these arrays contain the same elements.

```
func ==<Element>(Array<Element>, Array<Element>)
```
Returns `true` if these arrays contain the same elements.

```
func ==<Pointee>(UnsafePointer<Pointee>, UnsafePointer<Pointee>)
```

```
func ==(UInt16, UInt16)
```
Returns a Boolean value that indicates whether the two arguments have equal values.

```
func ==(Int, Int)
```
Returns a Boolean value that indicates whether the two arguments have equal values.

```
func ==<T>(_OptionalNilComparisonType, T?)
```

```
func ==<Element>(ArraySlice<Element>, ArraySlice<Element>)
```
Returns `true` if these arrays contain the same elements.

```
func ==(UInt64, UInt64)
```
Returns a Boolean value that indicates whether the two arguments have equal values.

```
func ==<Value, Element>(_HeapBuffer<Value, Element>, _HeapBuffer<Value
, Element>)
```

`func ==(Int16, Int16)`

Returns a Boolean value that indicates whether the two arguments have equal values.

`func ==<Header, Element>(ManagedBufferPointer<Header, Element>, ManagedBufferPointer<Header, Element>)`

`func ==(Int64, Int64)`

Returns a Boolean value that indicates whether the two arguments have equal values.

`func ==(UInt, UInt)`

Returns a Boolean value that indicates whether the two arguments have equal values.

`func ==<A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`

Returns a Boolean value indicating whether the corresponding components of two tuples are equal.

`func ==(Any.Type?, Any.Type?)`

Returns `true` iff `t0` is identical to `t1`; i.e. if they are both `nil` or they both represent the same type.

`func ==<T>(T, T)`

`func ==(UnsafeMutableRawPointer, UnsafeMutableRawPointer)`

`func ==(UnsafeRawPointer, UnsafeRawPointer)`

`func ===(AnyObject?, AnyObject?)`

Returns a Boolean value indicating whether two references point to the same object instance.

`func >(Int8, Int8)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func >(UInt8, UInt8)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func ><A, B, C>((A, B, C), (A, B, C))`

Returns a Boolean value indicating whether the first tuple is ordered after the second in a lexicographical

ordering.

`func >(UInt, UInt)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func ><T>(T, T)`

Returns a Boolean value indicating whether the value of the first argument is greater than that of the second argument.

`func >(UInt64, UInt64)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func >(Int, Int)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func ><A, B, C, D>((A, B, C, D), (A, B, C, D))`

Returns a Boolean value indicating whether the first tuple is ordered after the second in a lexicographical ordering.

`func >(Int64, Int64)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func ><A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))`

Returns a Boolean value indicating whether the first tuple is ordered after the second in a lexicographical ordering.

`func ><A, B>((A, B), (A, B))`

Returns a Boolean value indicating whether the first tuple is ordered after the second in a lexicographical ordering.

`func ><A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`

Returns a Boolean value indicating whether the first tuple is ordered after the second in a lexicographical ordering.

`func >(Int16, Int16)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func >(UInt16, UInt16)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func >(Int32, Int32)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func >(UInt32, UInt32)`

Returns a Boolean value that indicates whether the first argument is greater than the second argument.

`func ><T>(T, T)`

`func ><T>(T, T)`

`func >=(Int16, Int16)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=(Int32, Int32)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=<A, B, C, D>((A, B, C, D), (A, B, C, D))`

Returns a Boolean value indicating whether the first tuple is ordered after or the same as the second in a lexicographical ordering.

`func >=<A, B, C, D, E>((A, B, C, D, E), (A, B, C, D, E))`

Returns a Boolean value indicating whether the first tuple is ordered after or the same as the second in a lexicographical ordering.

`func >=(UInt8, UInt8)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=(Int8, Int8)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=<A, B, C>((A, B, C), (A, B, C))`

Returns a Boolean value indicating whether the first tuple is ordered after or the same as the second in a lexicographical ordering.

`func >=(Int64, Int64)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=(Int, Int)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=(UInt, UInt)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=<A, B>((A, B), (A, B))`

Returns a Boolean value indicating whether the first tuple is ordered after or the same as the second in a lexicographical ordering.

`func >=(UInt16, UInt16)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=<T>(T, T)`

Returns a Boolean value indicating whether the value of the first argument is greater than or equal to that of the second argument.

`func >=(UInt32, UInt32)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=(UInt64, UInt64)`

Returns a Boolean value that indicates whether the first argument is greater than or equal to the second argument.

`func >=<A, B, C, D, E, F>((A, B, C, D, E, F), (A, B, C, D, E, F))`

Returns a Boolean value indicating whether the first tuple is ordered after or the same as the second in a lexicographical ordering.

`func >=<T>(T, T)`

`func >=<T>(T, T)`

`func >>(Int32, Int32)`

`func >>(Int8, Int8)`

`func >>(UInt64, UInt64)`

`func >>(Int64, Int64)`

`func >>(UInt8, UInt8)`

`func >>(UInt, UInt)`

`func >>(Int, Int)`

`func >>(UInt32, UInt32)`

`func >>(UInt16, UInt16)`

`func >>(Int16, Int16)`

`func >>=(inout UInt64, UInt64)`

`func >>=(inout UInt16, UInt16)`

`func >>=(inout Int32, Int32)`

`func >>=(inout UInt32, UInt32)`

`func >>=(inout Int, Int)`

`func >>=(inout Int16, Int16)`

`func >>=(inout Int8, Int8)`

`func >>=(inout Int64, Int64)`

`func >>=(inout UInt8, UInt8)`

`func >>=(inout UInt, UInt)`

`func ??<T>(T?, () -> T)`

Performs a nil-coalescing operation, returning the wrapped value of an `Optional` instance or a default value.

`func ??<T>(T?, () -> T?)`

Performs a nil-coalescing operation, returning the wrapped value of an `Optional` instance or a default `Optional` value.

`func ^(Int16, Int16)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(UInt16, UInt16)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(Int32, Int32)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(Int8, Int8)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(UInt, UInt)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(Int64, Int64)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(UInt64, UInt64)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(UInt8, UInt8)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(Int, Int)`

Returns the bits that are set in exactly one of the two arguments.

`func ^(UInt32, UInt32)`

Returns the bits that are set in exactly one of the two arguments.

`func ^=(inout UInt32, UInt32)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout UInt, UInt)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout UInt16, UInt16)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout Int64, Int64)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout Int16, Int16)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout Int, Int)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout UInt8, UInt8)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout Int32, Int32)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout Int8, Int8)`

`func ^=(inout Int8, Int8)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=(inout UInt64, UInt64)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func ^=<T>(inout T, T)`

Calculates the bits that are set in exactly one of the two arguments and stores the result in the first argument.

`func |(Int8, Int8)`

Returns the union of bits set in the two arguments.

`func |(UInt64, UInt64)`

Returns the union of bits set in the two arguments.

`func |(UInt, UInt)`

Returns the union of bits set in the two arguments.

`func |(Int16, Int16)`

Returns the union of bits set in the two arguments.

`func |(Int64, Int64)`

Returns the union of bits set in the two arguments.

`func |(Int, Int)`

Returns the union of bits set in the two arguments.

`func |(Int32, Int32)`

Returns the union of bits set in the two arguments.

`func |(UInt16, UInt16)`

Returns the union of bits set in the two arguments.

`func |(UInt8, UInt8)`

Returns the union of bits set in the two arguments.

`func |(UInt32, UInt32)`

Returns the union of bits set in the two arguments.

`func |=(inout UInt32, UInt32)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout UInt64, UInt64)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout Int64, Int64)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout Int32, Int32)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout Int16, Int16)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout UInt8, UInt8)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout Int8, Int8)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout Int, Int)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout UInt16, UInt16)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=(inout UInt, UInt)`

Calculates the union of bits set in the two arguments and stores the result in the first argument.

`func |=<T>(inout T, T)`

Calculates the union of bits sets in the two arguments and stores the result in the first argument.

`func ~(UInt16)`

Returns the inverse of the bits set in the argument.

`func ~(UInt)`

Returns the inverse of the bits set in the argument.

`func ~(UInt32)`

Returns the inverse of the bits set in the argument.

`func ~(Int32)`

Returns the inverse of the bits set in the argument.

`func ~(Int)`

Returns the inverse of the bits set in the argument.

`func ~(Int8)`

Returns the inverse of the bits set in the argument.

`func ~(UInt8)`

Returns the inverse of the bits set in the argument.

`func ~(UInt64)`

Returns the inverse of the bits set in the argument.

`func ~(Int16)`

Returns the inverse of the bits set in the argument.

`func ~(Int64)`

Returns the inverse of the bits set in the argument.

`func ~=<T>(T, T)`

```
func ~=<T>(T, T)

func ~=<T>(_OptionalNilComparisonType, T?)
```