Structure

Int

A signed integer value type.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
@frozen
struct Int
```

Overview

On 32-bit platforms, `Int` is the same size as `Int32`, and on 64-bit platforms, `Int` is the same size as `Int64`.

Topics

Converting Integers

`init<T>(T)`

Creates a new instance from the given integer.

`init?<T>(exactly: T)`

`init<Other>(clamping: Other)`

Creates a new instance with the representable value that's closest to the given integer.

`init<T>(truncatingIfNeeded: T)`

Creates a new instance from the bit pattern of the given instance by sign-extending or truncating to fit this type.

`init(bitPattern: UInt)`

Creates a new instance with the same memory representation as the given value.

`init?(exactly: NSNumber)`

`init(truncating: NSNumber)`

Converting Floating-Point Values

`init<T>(T)`

`init(Double)`

Creates an integer from the given floating-point value, rounding toward zero.

`init(Float)`

Creates an integer from the given floating-point value, rounding toward zero.

`init(Float16)`

Creates an integer from the given floating-point value, rounding toward zero.

`init(Float80)`

Creates an integer from the given floating-point value, rounding toward zero.

`init(CGFloat)`

Converting with No Loss of Precision

These initializers result in `nil` if the value passed can't be represented without any loss of precision.

`init?<T>(exactly: T)`

`init?(exactly: Double)`

Creates an integer from the given floating-point value, if it can be represented exactly.

`init?(exactly: Float)`

Creates an integer from the given floating-point value, if it can be represented exactly.

`init?(exactly: Float16)`

Creates an integer from the given floating-point value, if it can be represented exactly.

`init?(exactly: Float80)`

Creates an integer from the given floating-point value, if it can be represented exactly.

Converting Strings

`init?(String)`

Creates a new integer value from the given string.

`init?<S>(S, radix: Int)`

Creates a new integer value from the given string and radix.

Creating a Random Integer

`static func random(in: Range<Self>) -> Self`

Returns a random value within the specified range.

`static func random<T>(in: Range<Self>, using: inout T) -> Self`

Returns a random value within the specified range, using the given generator as a source for randomness.

`static func random(in: ClosedRange<Self>) -> Self`

Returns a random value within the specified range.

`static func random<T>(in: ClosedRange<Self>, using: inout T) -> Self`

Returns a random value within the specified range, using the given generator as a source for randomness.

Performing Calculations

Integer Operators

Perform arithmetic and bitwise operations or compare values.

`func negate()`

Replaces this value with its additive inverse.

`func quotientAndRemainder(dividingBy: Self) -> (quotient: Self, remainder: Self)`

Returns the quotient and remainder of this value divided by the given value.

`func isMultiple(of: Self) -> Bool`

Returns `true` if this value is a multiple of the given value, and `false` otherwise.

Performing Calculations with Overflow

These methods return the result of an operation, and a flag indicating whether the operation overflowed the bounds of the type.

```
func addingReportingOverflow(Int) -> (partialValue: Int, overflow: Bool
)
```

Returns the sum of this value and the given value, along with a Boolean value indicating whether overflow occurred in the operation.

```
func subtractingReportingOverflow(Int) -> (partialValue: Int, overflow:
Bool)
```

Returns the difference obtained by subtracting the given value from this value, along with a Boolean value indicating whether overflow occurred in the operation.

```
func multipliedReportingOverflow(by: Int) -> (partialValue: Int,
overflow: Bool)
```

Returns the product of this value and the given value, along with a Boolean value indicating whether overflow occurred in the operation.

```
func dividedReportingOverflow(by: Int) -> (partialValue: Int, overflow:
Bool)
```

Returns the quotient obtained by dividing this value by the given value, along with a Boolean value indicating whether overflow occurred in the operation.

```
func remainderReportingOverflow(dividingBy: Int) -> (partialValue: Int,
overflow: Bool)
```

Returns the remainder after dividing this value by the given value, along with a Boolean value indicating whether overflow occurred during division.

Performing Double-Width Calculations

```
func multipliedFullWidth(by: Int) -> (high: Int, low: Int.Magnitude)
```

Returns a tuple containing the high and low parts of the result of multiplying this value by the given value.

```
func dividingFullWidth((high: Int, low: Int.Magnitude)) -> (quotient:
Int, remainder: Int)
```

Returns a tuple containing the quotient and remainder of dividing the given value by this value.

Finding the Sign and Magnitude

`var magnitude: UInt`

The magnitude of this value.

`typealias Magnitude`

A type that can represent the absolute value of any possible value of this type.

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

Returns the absolute value of the given number.

`func signum() -> Int`

Returns `-1` if this value is negative and `1` if it's positive; otherwise, `0`.

Accessing Numeric Constants

`static var zero: Self`

The zero value.

`static var min: Self`

The minimum representable integer in this type.

`static var max: Self`

The maximum representable integer in this type.

`static var isSigned: Bool`

A Boolean value indicating whether this type is a signed integer type.

Working with Byte Order

`var byteSwapped: Int`

A representation of this integer with the byte order swapped.

`var littleEndian: Self`

The little-endian representation of this integer.

`var bigEndian: Self`

The big-endian representation of this integer.

`init(littleEndian: Self)`

Creates an integer from its little-endian representation, changing the byte order if necessary.

`init(bigEndian: Self)`

Creates an integer from its big-endian representation, changing the byte order if necessary.

Working with Binary Representation

`static var bitWidth: Int`

The number of bits used for the underlying binary representation of values of this type.

`var bitWidth: Int`

The number of bits in the current binary representation of this value.

`var nonzeroBitCount: Int`

The number of bits equal to 1 in this value's binary representation.

`var leadingZeroBitCount: Int`

The number of leading zeros in this value's binary representation.

`var trailingZeroBitCount: Int`

The number of trailing zeros in this value's binary representation.

`var words: Int.Words`

A collection containing the words of this value's binary representation, in order from the least significant to most significant.

`struct Words`

A type that represents the words of this integer.

Working with Memory Addresses

These initializers create an integer with the bit pattern of the memory address of a pointer or class instance.

`init<P>(bitPattern: P?)`

Creates a new value with the bit pattern of the given pointer.

`init(bitPattern: ObjectIdentifier)`

Creates an integer that captures the full value of the given object identifier.

`init(bitPattern: OpaquePointer?)`

Creates a new value with the bit pattern of the given pointer.

Encoding and Decoding Values

`func encode(to: any Encoder) throws`

Encodes this value into the given encoder.

`init(from: any Decoder) throws`

Creates a new instance by decoding from the given decoder.

Describing an Integer

`var description: String`

A textual representation of this value.

`func hash(into: inout Hasher)`

Hashes the essential components of this value by feeding them into the given hasher.

`var customMirror: Mirror`

A mirror that reflects the `Int` instance.

Infrequently Used Functionality

`init()`

Creates a new value equal to zero.

`init(integerLiteral: Self)`

`typealias IntegerLiteralType`

A type that represents an integer literal.

`func distance(to: Int) -> Int`

Returns the distance from this value to the given value, expressed as a stride.

`func advanced(by: Int) -> Int`

Returns a value that is offset the specified distance from this value.

`typealias Stride`

A type that represents the distance between two values.

`var hashValue: Int`

The hash value.

Deprecated

~~`var customPlaygroundQuickLook: _PlaygroundQuickLook`~~

A custom playground Quick Look for the `Int` instance.

Deprecated

`init(NSNumber)`

SIMD-Supporting Types

`typealias SIMDMaskScalar`

`struct SIMD2Storage`

Storage for a vector of two integers.

`struct SIMD4Storage`

Storage for a vector of four integers.

`struct SIMD8Storage`

Storage for a vector of eight integers.

`struct SIMD16Storage`

Storage for a vector of 16 integers.

`struct SIMD32Storage`

Storage for a vector of 32 integers.

`struct SIMD64Storage`

Storage for a vector of 64 integers.

Operators

`static func != (Int, Int) -> Bool`

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

Calculates the result of shifting a value's binary representation the specified number of digits to the right, masking the shift amount to the type's bit width, and stores the result in the left-hand-side variable.

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

Returns the result of shifting a value's binary representation the specified number of digits to the left, masking the shift amount to the type's bit width, and stores the result in the left-hand-side variable.

`static func < (Int, Int) -> Bool`

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

`static func ^= (inout Int, Int)`

Stores the result of performing a bitwise XOR operation on the two given values in the left-hand-side variable.

`static func %= (inout Int, Int)`

Divides the first value by the second and stores the remainder in the left-hand-side variable.

`static func |= (inout Int, Int)`

Stores the result of performing a bitwise OR operation on the two given values in the left-hand-side variable.

Type Aliases

`typealias Specification`

`typealias UnwrappedType`

`typealias ValueType`

Type Properties

`static var defaultResolverSpecification: some ResolverSpecification`

Default Implementations

- AdditiveArithmetic Implementations
- AtomicRepresentable Implementations
- BinaryInteger Implementations
- CodingKeyRepresentable Implementations
- Comparable Implementations
- CustomReflectable Implementations

- Decodable Implementations
- Encodable Implementations
- Equatable Implementations
- ExpressibleByIntegerLiteral Implementations
- FixedWidthInteger Implementations
- Hashable Implementations
- SIMDScalar Implementations
- SignedInteger Implementations
- SignedNumeric Implementations
- Strideable Implementations

Relationships

Conforms To

`AdditiveArithmetic`
`AtomicRepresentable`
`BNNSGraph.Builder.SliceIndex`
`BinaryInteger`
`BindableData`
`BitwiseCopyable`
`CKRecordValueProtocol`
`CVarArg`
`CodingKeyRepresentable`
`Comparable`
`ConvertibleFromGeneratedContent`
`ConvertibleToGeneratedContent`
`Copyable`
`CustomReflectable`
`CustomStringConvertible`
`CustomURLRepresentationParameterConvertible`
`Decodable`
`Encodable`
`EntityIdentifierConvertible`

Equatable
ExpressibleByIntegerLiteral
FixedWidthInteger
Generable
Hashable
InstructionsRepresentable
LosslessStringConvertible
MLDataValueConvertible
MLIdentifier
MLTensorRangeExpression
MirrorPath
Numeric
Plottable
PrimitivePlottableProtocol
PromptRepresentable
RangeComparableProperty
SIMDScalar
Sendable
SendableMetatype
SignedInteger
SignedNumeric
Strideable

See Also

Standard Library

`struct` **Double**

A double-precision, floating-point value type.

`struct` **String**

A Unicode string value that is a collection of characters.

`struct` **Array**

An ordered, random-access collection.

`struct` **Dictionary**

A collection whose elements are key-value pairs.

Swift Standard Library

Solve complex problems and write high-performance, readable code.