IntPtr Struct

Reference

Definition

Namespace: System

Assembly: System.Runtime.dll

Represents a signed integer where the bit-width is the same as a pointer.

C#

```
public readonly struct IntPtr : IComparable<IntPtr>,
IEquatable<IntPtr>, IParsable<IntPtr>, ISpanParsable<IntPtr>,
System.Numerics.IAdditionOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.IAdditiveIdentity<IntPtr,IntPtr>,
System.Numerics.IBinaryInteger<IntPtr>,
System.Numerics.IBinaryNumber<IntPtr>,
System.Numerics.IBitwiseOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.IComparisonOperators<IntPtr,IntPtr,bool>,
System.Numerics.IDecrementOperators<IntPtr>,
System.Numerics.IDivisionOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.IEqualityOperators<IntPtr,IntPtr,bool>,
System.Numerics.IIncrementOperators<IntPtr>,
System.Numerics.IMinMaxValue<IntPtr>,
System.Numerics.IModulusOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.IMultiplicativeIdentity<IntPtr,IntPtr>,
System.Numerics.IMultiplyOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.INumber<IntPtr>, System.Numerics.INumberBase<IntPtr>,
System.Numerics.IShiftOperators<IntPtr,int,IntPtr>,
System.Numerics.ISignedNumber<IntPtr>,
System.Numerics.ISubtractionOperators<IntPtr,IntPtr,IntPtr>,
System.Numerics.IUnaryNegationOperators<IntPtr,IntPtr>,
System.Numerics.IUnaryPlusOperators<IntPtr,IntPtr>,
System.Runtime.Serialization.ISerializable
```

Inheritance Object → ValueType → IntPtr

Implements IComparable , IComparable<IntPtr> , IEquatable<IntPtr> , IFormattable , ISerializable , ISpanFormattable , IComparable<TSelf> , IEquatable<TSelf> , IParsable<IntPtr> , IParsable<TSelf> , ISpanParsable<IntPtr> , ISpanParsable<TSelf> , IAdditionOperators<IntPtr,IntPtr,IntPtr> , IAdditionOperators<TSelf,TSelf,TSelf> , IAdditiveIdentity<IntPtr,IntPtr> ,

IAdditiveIdentity<TSelf,TSelf> , IBinaryInteger<IntPtr> ,

IBinaryNumber<IntPtr> , IBinaryNumber<TSelf> ,

IBitwiseOperators<IntPtr,IntPtr,IntPtr> ,

IBitwiseOperators<TSelf,TSelf,TSelf> ,

IComparisonOperators<IntPtr,IntPtr,Boolean> ,

IComparisonOperators<TSelf,TSelf,Boolean> ,

IDecrementOperators<IntPtr> , IDecrementOperators<TSelf> ,

IDivisionOperators<IntPtr,IntPtr,IntPtr> ,

IDivisionOperators<TSelf,TSelf,TSelf> ,

IEqualityOperators<IntPtr,IntPtr,Boolean> ,

IEqualityOperators<TSelf,TOther,TResult> ,

IEqualityOperators<TSelf,TSelf,Boolean> , IIncrementOperators<IntPtr> ,

IIncrementOperators<TSelf> , IMinMaxValue<IntPtr> ,

IModulusOperators<IntPtr,IntPtr,IntPtr> ,

IModulusOperators<TSelf,TSelf,TSelf> ,

IMultiplicativeIdentity<IntPtr,IntPtr> , IMultiplicativeIdentity<TSelf,TSelf> ,

IMultiplyOperators<IntPtr,IntPtr,IntPtr> ,

IMultiplyOperators<TSelf,TSelf,TSelf> , INumber<IntPtr> ,

INumber<TSelf> , INumberBase<IntPtr> , INumberBase<TSelf> ,

IShiftOperators<IntPtr,Int32,IntPtr> , IShiftOperators<TSelf,Int32,TSelf> ,

ISignedNumber<IntPtr> , ISubtractionOperators<IntPtr,IntPtr,IntPtr> ,

ISubtractionOperators<TSelf,TSelf,TSelf> ,

IUnaryNegationOperators<IntPtr,IntPtr> ,

IUnaryNegationOperators<TSelf,TSelf> ,

IUnaryPlusOperators<IntPtr,IntPtr> , IUnaryPlusOperators<TSelf,TSelf>

Examples

The following example uses managed pointers to reverse the characters in an array. After it initializes a String object and gets its length, it does the following:

1. Calls the Marshal.StringToHGlobalAnsi method to copy the Unicode string to unmanaged memory as an ANSI (one-byte) character. The method returns an IntPtr object that points to the beginning of the unmanaged string. The Visual

Basic example uses this pointer directly; in the C++, F# and C# examples, it is cast to a pointer to a byte.

2. Calls the Marshal.AllocHGlobal method to allocate the same number of bytes as the unmanaged string occupies. The method returns an IntPtr object that points to the beginning of the unmanaged block of memory. The Visual Basic example uses this pointer directly; in the C++, F# and C# examples, it is cast to a pointer to a byte.

3. The Visual Basic example defines a variable named `offset` that is equal to the length of the ANSI string. It is used to determine the offset into unmanaged memory to which the next character in the ANSI string is copied. Because its starting value is the length of the string, the copy operation will copy a character from the start of the string to the end of the memory block.

   The C#, F# and C++ examples call the ToPointer method to get an unmanaged pointer to the starting address of the string and the unmanaged block of memory, and they add one less than the length of the string to the starting address of the ANSI string. Because the unmanaged string pointer now points to the end of the string, the copy operation will copy a character from the end of the string to the start of the memory block.

4. Uses a loop to copy each character from the string to the unmanaged block of memory.

   The Visual Basic example calls the Marshal.ReadByte(IntPtr, Int32) method to read the byte (or one-byte character) at a specified offset from the managed pointer to the ANSI string. The offset is incremented with each iteration of the loop. It then calls the Marshal.WriteByte(IntPtr, Int32, Byte) method to write the byte to the memory address defined by the starting address of the unmanaged block of memory plus `offset`. It then decrements `offset`.

   The C#, F# and C++ examples perform the copy operation, then decrement the pointer to the address of the next location in the unmanaged ANSI string and increment the pointer to the next address in the unmanaged block.

5. All examples call the Marshal.PtrToStringAnsi to convert the unmanaged memory block containing the copied ANSI string to a managed Unicode String object.

6. After displaying the original and reversed strings, all examples call the FreeHGlobal method to free the memory allocated for the unmanaged ANSI string and the unmanaged block of memory.

```
C#
```

```
using System;
using System.Runtime.InteropServices;

class NotTooSafeStringReverse
{
    static public void Main()
    {
        string stringA = "I seem to be turned around!";
        int copylen = stringA.Length;

        // Allocate HGlobal memory for source and destination strings
        IntPtr sptr = Marshal.StringToHGlobalAnsi(stringA);
        IntPtr dptr = Marshal.AllocHGlobal(copylen + 1);

        // The unsafe section where byte pointers are used.
        unsafe
        {
            byte *src = (byte *)sptr.ToPointer();
            byte *dst = (byte *)dptr.ToPointer();

            if (copylen > 0)
            {
                // set the source pointer to the end of the string
                // to do a reverse copy.
                src += copylen - 1;

                while (copylen-- > 0)
                {
                    *dst++ = *src--;
                }
                *dst = 0;
            }
        }
        string stringB = Marshal.PtrToStringAnsi(dptr);

        Console.WriteLine("Original:\n{0}\n", stringA);
        Console.WriteLine("Reversed:\n{0}", stringB);

        // Free HGlobal memory
        Marshal.FreeHGlobal(dptr);
        Marshal.FreeHGlobal(sptr);
    }
}

// The progam has the following output:
//
// Original:
// I seem to be turned around!
//
// Reversed:
// !dnuora denrut eb ot mees I
```

Remarks

The IntPtr type is designed to be an integer whose size is the same as a pointer. That is, an instance of this type is expected to be 32 bits in a 32-bit process and 64 bits in a 64-bit process.

The IntPtr type can be used by languages that support pointers and as a common means of referring to data between languages that do and do not support pointers.

IntPtr objects can also be used to hold handles. For example, instances of IntPtr are used extensively in the System.IO.FileStream class to hold file handles.

> **Note**
>
> Using **IntPtr** as a pointer or a handle is error prone and unsafe. It is simply an integer type that can be used as an interchange format for pointers and handles due to being the same size. Outside of specific interchange requirements, such as for passing data to a language that doesn't support pointers, a correctly typed pointer should be used to represent pointers and **SafeHandle** should be used to represent handles.

This type implements the ISerializable. In .NET 5 and later versions, this type also implements the IFormattable interfaces. In .NET 7 and later versions, this type also implements the IBinaryInteger<TSelf>, IMinMaxValue<TSelf>, and ISignedNumber<TSelf> interfaces.

In C# starting from version 9.0, you can use the built-in `nint` type to define native-sized integers. This type is represented by the IntPtr type internally and provides operations and conversions that are appropriate for integer types. For more information, see nint and nuint types.

In C# starting from version 11 and when targeting the .NET 7 or later runtime, `nint` is an alias for IntPtr in the same way that `int` is an alias for Int32.

Constructors

IntPtr(Int32)	Initializes a new instance of IntPtr using the specified 32-bit signed integer.
IntPtr(Int64)	Initializes a new instance of IntPtr using the specified 64-bit signed integer.

IntPtr(Void*)	Initializes a new instance of IntPtr using the specified pointer to an unspecified type.

Fields

Zero	A read-only field that represents a signed integer that has been initialized to zero.

Properties

MaxValue	Gets the largest possible value of IntPtr.
MinValue	Gets the smallest possible value of IntPtr.
Size	Gets the size of this instance.

Methods

Abs(IntPtr)	Computes the absolute of a value.
Add(IntPtr, Int32)	Adds an offset to a signed integer.
Clamp(IntPtr, IntPtr, IntPtr)	Clamps a value to an inclusive minimum and maximum value.
CompareTo(IntPtr)	Compares the current instance with another object of the same type and returns an integer that indicates whether the current instance precedes, follows, or occurs in the same position in the sort order as the other object.
CompareTo(Object)	Compares the current instance with another object of the same type and returns an integer that indicates whether the current instance precedes, follows, or occurs in the same position in the sort order as the other object.
CopySign(IntPtr, IntPtr)	Copies the sign of a value to the sign of another value.
CreateChecked<TOther>(TOther)	Creates an instance of the current type from a value, throwing an overflow exception for any values that fall outside the representable range of the current type.
CreateSaturating<TOther>(TOther)	Creates an instance of the current type from a value, saturating any values that fall outside the representable range of the current type.

CreateTruncating<TOther>(TOther)	Creates an instance of the current type from a value, truncating any values that fall outside the representable range of the current type.
DivRem(IntPtr, IntPtr)	Computes the quotient and remainder of two values.
Equals(IntPtr)	Indicates whether the current object is equal to another object of the same type.
Equals(Object)	Returns a value indicating whether this instance is equal to a specified object.
GetHashCode()	Returns the hash code for this instance.
IsEvenInteger(IntPtr)	Determines if a value represents an even integral number.
IsNegative(IntPtr)	Determines if a value is negative.
IsOddInteger(IntPtr)	Determines if a value represents an odd integral number.
IsPositive(IntPtr)	Determines if a value is positive.
IsPow2(IntPtr)	Determines if a value is a power of two.
LeadingZeroCount(IntPtr)	Computes the number of leading zeros in a value.
Log2(IntPtr)	Computes the log2 of a value.
Max(IntPtr, IntPtr)	Compares two values to compute which is greater.
MaxMagnitude(IntPtr, IntPtr)	Compares two values to compute which is greater.
Min(IntPtr, IntPtr)	Compares two values to compute which is lesser.
MinMagnitude(IntPtr, IntPtr)	Compares two values to compute which is lesser.
Parse(ReadOnlySpan<Char>, IFormatProvider)	Parses a span of characters into a value.
Parse(ReadOnlySpan<Char>, NumberStyles, IFormatProvider)	Converts the read-only span of characters representation of a number in a specified style and culture-specific format to its signed native integer equivalent.
Parse(String)	Converts the string representation of a number to its signed native integer equivalent.
Parse(String, IFormatProvider)	Converts the string representation of a number in a specified culture-specific format to its signed native integer equivalent.
Parse(String, NumberStyles)	Converts the string representation of a number in a specified style to its signed native integer equivalent.

Parse(String, NumberStyles, IFormatProvider)	Converts the string representation of a number in a specified style and culture-specific format to its signed native integer equivalent.
PopCount(IntPtr)	Computes the number of bits that are set in a value.
RotateLeft(IntPtr, Int32)	Rotates a value left by a given amount.
RotateRight(IntPtr, Int32)	Rotates a value right by a given amount.
Sign(IntPtr)	Computes the sign of a value.
Subtract(IntPtr, Int32)	Subtracts an offset from a signed integer.
ToInt32()	Converts the value of this instance to a 32-bit signed integer.
ToInt64()	Converts the value of this instance to a 64-bit signed integer.
ToPointer()	Converts the value of this instance to a pointer to an unspecified type.
ToString()	Converts the numeric value of the current IntPtr object to its equivalent string representation.
ToString(IFormatProvider)	Converts the numeric value of this instance to its equivalent string representation using the specified format and culture-specific format information.
ToString(String)	Converts the numeric value of the current IntPtr object to its equivalent string representation.
ToString(String, IFormat Provider)	Formats the value of the current instance using the specified format.
TrailingZeroCount(IntPtr)	Computes the number of trailing zeros in a value.
TryFormat(Span<Char>, Int32, ReadOnlySpan<Char>, IFormatProvider)	Tries to format the value of the current instance into the provided span of characters.
TryParse(ReadOnly Span<Char>, IFormatProvider, IntPtr)	Tries to parse a string into a value.
TryParse(ReadOnly Span<Char>, IntPtr)	Converts the read-only span of characters representation of a number to its signed native integer equivalent. A return value indicates whether the conversion succeeded.
TryParse(ReadOnly Span<Char>, NumberStyles, IFormatProvider, IntPtr)	Converts the read-only span of characters representation of a number in a specified style and culture-specific format to its signed native integer equivalent. A return value indicates whether the conversion succeeded.

TryParse(String, IFormatProvider, IntPtr)	Tries to parse a string into a value.
TryParse(String, IntPtr)	Converts the string representation of a number to its signed native integer equivalent. A return value indicates whether the conversion succeeded.
TryParse(String, NumberStyles, IFormatProvider, IntPtr)	Converts the string representation of a number in a specified style and culture-specific format to its signed native integer equivalent. A return value indicates whether the conversion succeeded.

Operators

Addition(IntPtr, Int32)	Adds an offset to a signed integer.
Equality(IntPtr, IntPtr)	Determines whether two specified instances of IntPtr are equal.
Explicit(Int32 to IntPtr)	Converts the value of a 32-bit signed integer to an IntPtr.
Explicit(Int64 to IntPtr)	Converts the value of a 64-bit signed integer to an IntPtr.
Explicit(IntPtr to Int32)	Converts the value of the specified IntPtr to a 32-bit signed integer.
Explicit(IntPtr to Int64)	Converts the value of the specified IntPtr to a 64-bit signed integer.
Explicit(IntPtr to Void*)	Converts the value of the specified IntPtr to a pointer to an unspecified type. This API is not CLS-compliant.
Explicit(Void* to IntPtr)	Converts the specified pointer to an unspecified type to an IntPtr. This API is not CLS-compliant.
Inequality(IntPtr, IntPtr)	Determines whether two specified instances of IntPtr are not equal.
Subtraction(IntPtr, Int32)	Subtracts an offset from a signed integer.

Explicit Interface Implementations

IAdditionOperators<IntPtr,IntPtr,IntPtr>.Addition(IntPtr, Int	Adds two values together to compute their sum.

Ptr)	
IAdditionOperators<IntPtr,IntPtr,IntPtr>.CheckedAddition(IntPtr, IntPtr)	Adds two values together to compute their sum.
IAdditiveIdentity<IntPtr,IntPtr>.AdditiveIdentity	Gets the additive identity of the current type.
IBinaryInteger<IntPtr>.GetByteCount()	Gets the number of bytes that will be written as part of TryWriteLittleEndian(Span<Byte>, Int32).
IBinaryInteger<IntPtr>.GetShortestBitLength()	Gets the length, in bits, of the shortest two's complement representation of the current value.
IBinaryInteger<IntPtr>.TryReadBigEndian(ReadOnlySpan<Byte>, Boolean, IntPtr)	
IBinaryInteger<IntPtr>.TryReadLittleEndian(ReadOnlySpan<Byte>, Boolean, IntPtr)	
IBinaryInteger<IntPtr>.TryWriteBigEndian(Span<Byte>, Int32)	Tries to write the current value, in big-endian format, to a given span.
IBinaryInteger<IntPtr>.TryWriteLittleEndian(Span<Byte>, Int32)	Tries to write the current value, in little-endian format, to a given span.
IBinaryNumber<IntPtr>.AllBitsSet	Gets an instance of the binary type in which all bits are set.
IBitwiseOperators<IntPtr,IntPtr,IntPtr>.BitwiseAnd(IntPtr, IntPtr)	Computes the bitwise-and of two values.
IBitwiseOperators<IntPtr,IntPtr,IntPtr>.BitwiseOr(IntPtr, IntPtr)	Computes the bitwise-or of two values.
IBitwiseOperators<IntPtr,IntPtr,IntPtr>.ExclusiveOr(IntPtr, IntPtr)	Computes the exclusive-or of two values.
IBitwiseOperators<IntPtr,IntPtr,IntPtr>.OnesComplement(IntPtr)	Computes the ones-complement representation of a given value.
IComparisonOperators<IntPtr,IntPtr,Boolean>.GreaterThan(IntPtr, IntPtr)	Compares two values to determine which is greater.
IComparisonOperators<IntPtr,IntPtr,Boolean>.Greater	Compares two values to determine which is greater or equal.

ThanOrEqual(IntPtr, IntPtr)	
IComparisonOperators<IntPtr,IntPtr,Boolean>.LessThan(IntPtr, IntPtr)	Compares two values to determine which is less.
IComparisonOperators<IntPtr,IntPtr,Boolean>.LessThanOrEqual(IntPtr, IntPtr)	Compares two values to determine which is less or equal.
IDecrementOperators<IntPtr>.CheckedDecrement(IntPtr)	Decrements a value.
IDecrementOperators<IntPtr>.Decrement(IntPtr)	Decrements a value.
IDivisionOperators<IntPtr,IntPtr,IntPtr>.Division(IntPtr, IntPtr)	Divides one value by another to compute their quotient.
IIncrementOperators<IntPtr>.CheckedIncrement(IntPtr)	Increments a value.
IIncrementOperators<IntPtr>.Increment(IntPtr)	Increments a value.
IMinMaxValue<IntPtr>.MaxValue	Gets the maximum value of the current type.
IMinMaxValue<IntPtr>.MinValue	Gets the minimum value of the current type.
IModulusOperators<IntPtr,IntPtr,IntPtr>.Modulus(IntPtr, IntPtr)	Divides two values together to compute their modulus or remainder.
IMultiplicativeIdentity<IntPtr,IntPtr>.MultiplicativeIdentity	Gets the multiplicative identity of the current type.
IMultiplyOperators<IntPtr,IntPtr,IntPtr>.CheckedMultiply(IntPtr, IntPtr)	Multiplies two values together to compute their product.
IMultiplyOperators<IntPtr,IntPtr,IntPtr>.Multiply(IntPtr, IntPtr)	Multiplies two values together to compute their product.
INumber<IntPtr>.MaxNumber(IntPtr, IntPtr)	Compares two values to compute which is greater and returning the other value if an input is `NaN`.

INumber<IntPtr>.MinNumber(IntPtr, IntPtr)	Compares two values to compute which is lesser and returning the other value if an input is `NaN`.
INumberBase<IntPtr>.IsCanonical(IntPtr)	Determines if a value is in its canonical representation.
INumberBase<IntPtr>.IsComplexNumber(IntPtr)	Determines if a value represents a complex number.
INumberBase<IntPtr>.IsFinite(IntPtr)	Determines if a value is finite.
INumberBase<IntPtr>.IsImaginaryNumber(IntPtr)	Determines if a value represents a pure imaginary number.
INumberBase<IntPtr>.IsInfinity(IntPtr)	Determines if a value is infinite.
INumberBase<IntPtr>.IsInteger(IntPtr)	Determines if a value represents an integral number.
INumberBase<IntPtr>.IsNaN(IntPtr)	Determines if a value is NaN.
INumberBase<IntPtr>.IsNegativeInfinity(IntPtr)	Determines if a value is negative infinity.
INumberBase<IntPtr>.IsNormal(IntPtr)	Determines if a value is normal.
INumberBase<IntPtr>.IsPositiveInfinity(IntPtr)	Determines if a value is positive infinity.
INumberBase<IntPtr>.IsRealNumber(IntPtr)	Determines if a value represents a real number.
INumberBase<IntPtr>.IsSubnormal(IntPtr)	Determines if a value is subnormal.
INumberBase<IntPtr>.IsZero(IntPtr)	Determines if a value is zero.
INumberBase<IntPtr>.MaxMagnitudeNumber(IntPtr, IntPtr)	Compares two values to compute which has the greater magnitude and returning the other value if an input is `NaN`.
INumberBase<IntPtr>.MinMagnitudeNumber(IntPtr, IntPtr)	Compares two values to compute which has the lesser magnitude and returning the other value if an input is `NaN`.
INumberBase<IntPtr>.One	Gets the value `1` for the type.

INumberBase<IntPtr>.Radix	Gets the radix, or base, for the type.
INumberBase<IntPtr>.TryConvertFromChecked<TOther>(TOther, IntPtr)	
INumberBase<IntPtr>.TryConvertFromSaturating<TOther>(TOther, IntPtr)	
INumberBase<IntPtr>.TryConvertFromTruncating<TOther>(TOther, IntPtr)	
INumberBase<IntPtr>.TryConvertToChecked<TOther>(IntPtr, TOther)	Tries to convert an instance of the the current type to another type, throwing an overflow exception for any values that fall outside the representable range of the current type.
INumberBase<IntPtr>.TryConvertToSaturating<TOther>(IntPtr, TOther)	Tries to convert an instance of the the current type to another type, saturating any values that fall outside the representable range of the current type.
INumberBase<IntPtr>.TryConvertToTruncating<TOther>(IntPtr, TOther)	Tries to convert an instance of the the current type to another type, truncating any values that fall outside the representable range of the current type.
INumberBase<IntPtr>.Zero	Gets the value `0` for the type.
ISerializable.GetObjectData(SerializationInfo, StreamingContext)	Populates a SerializationInfo object with the data needed to serialize the current IntPtr object.
IShiftOperators<IntPtr,Int32,IntPtr>.LeftShift(IntPtr, Int32)	Shifts a value left by a given amount.
IShiftOperators<IntPtr,Int32,IntPtr>.RightShift(IntPtr, Int32)	Shifts a value right by a given amount.
IShiftOperators<IntPtr,Int32,IntPtr>.UnsignedRightShift(IntPtr, Int32)	Shifts a value right by a given amount.
ISignedNumber<IntPtr>.NegativeOne	Gets the value `-1` for the type.
ISubtractionOperators<IntPtr,IntPtr,IntPtr>.CheckedSubtraction(IntPtr, IntPtr)	Subtracts two values to compute their difference.
ISubtractionOperators<IntPtr,IntPtr,IntPtr>.Subtraction(IntPtr, IntPtr)	Subtracts two values to compute their difference.
IUnaryNegationOperators<IntPtr,IntPtr>.CheckedUnary	Computes the checked unary negation of a value.

Negation(IntPtr)	
IUnaryNegationOperators<IntPtr,IntPtr>.UnaryNegation(IntPtr)	Computes the unary negation of a value.
IUnaryPlusOperators<IntPtr,IntPtr>.UnaryPlus(IntPtr)	Computes the unary plus of a value.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Thread Safety

This type is thread safe.

See also

- UIntPtr