System Library Reference

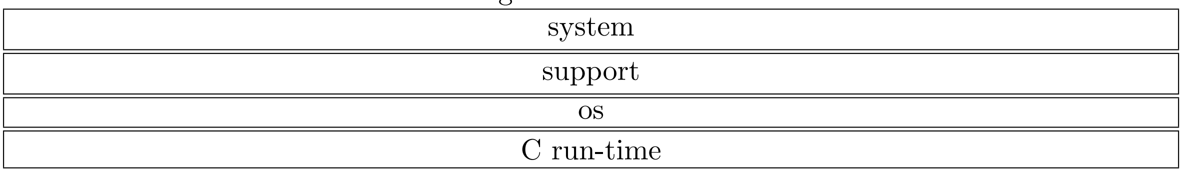
September 22, 2014

Contents

Description

The System library is the run-time library for Cmajor programs. The System library is built on top of the Support library that contains the run-time support for Cmajor language implementation (primarily support for exception handling). The Support library in turn is built on top of the Os library that provides interface to operating system services. The Os library is built on top of the C run-time library for the platform. In Windows this is the C run-time library of the **mingw_w64's gcc** compiler and in Linux this the C run-time library of the **GNU/Linux gcc** compiler. Figure 1 illustrates the layered architecture of Cmajor.

Figure 1: Libraries



Note: You may want to enable Previous View and Next View commands in the PDF viewer for comfortable browsing experience in this document.

Copyrights

Namespaces

Namespace	Description
System	Contains fundamental classes, functions and type definitions.
System.Collections	Contains collection classes and functions that operate on collections.
System.Concepts	Contains system library concepts.
System.IO	Contains classes and functions for doing input and output.
System.Text	Contains classes and functions for manipulating text.
System.Threading	Contains classes and functions for controlling multiple threads of execution.

1 System Namespace

Contains fundamental classes, functions and type definitions.

Figures 1.1 and 1.2 contain the classes in this namespace.

Figure 1.1: Class Diagram 1: Basic Classes

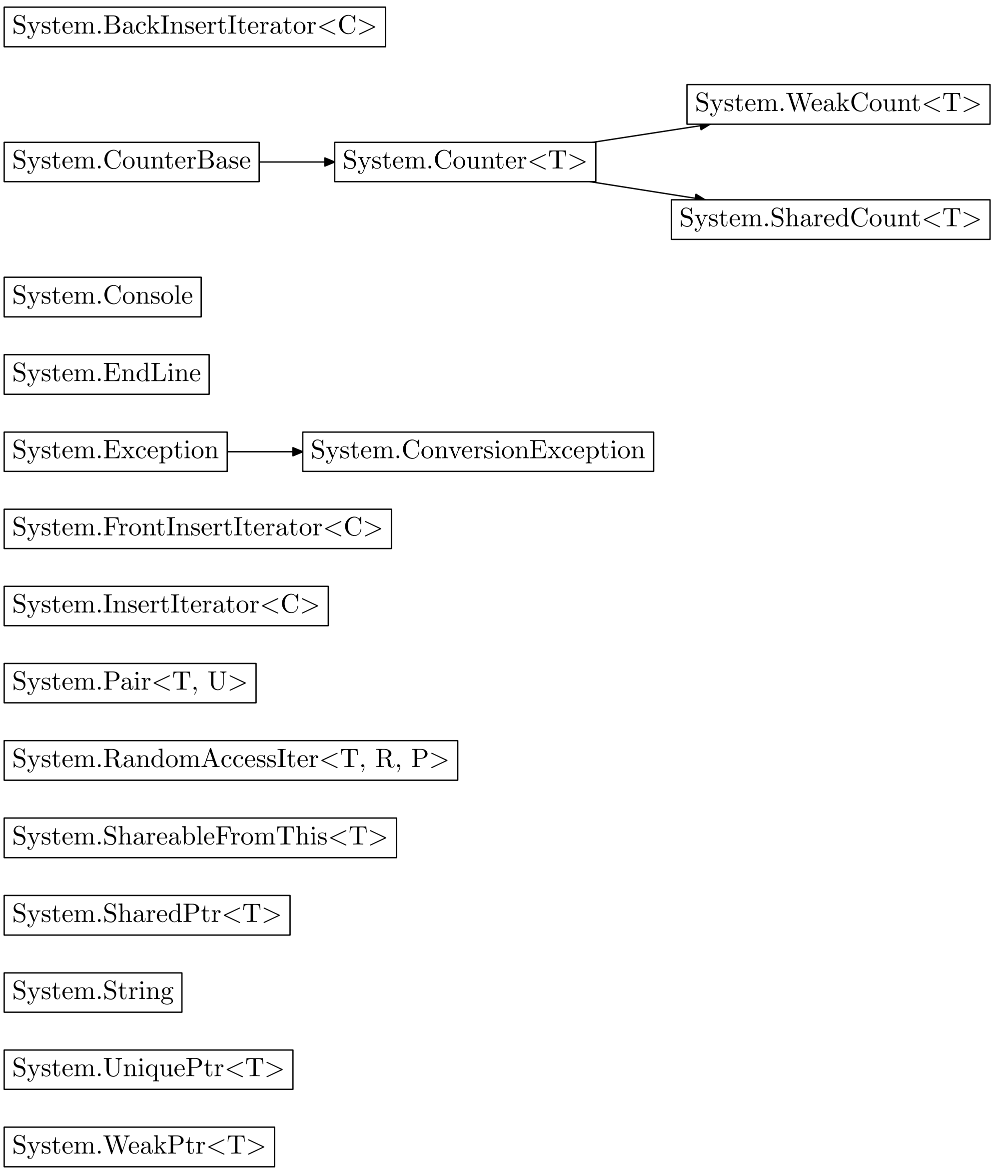


Figure 1.2: Class Diagram 2: Function Objects

1.1 Classes

Class	Description
BackInsertIterator<C>	An output iterator that inserts elements to the end of a back insertion sequence.
BinaryFun<Argument1, Argument2, Result>	A base class for binary function objects.
BinaryPred<Argument1, Argument2>	A base class for binary predicates.
Console	A static class that contains console input and output functions.
ConversionException	An exception thrown when a conversion function fails.
Counter<T>	A counter class that stores a pointer to the counted object.
CounterBase	An abstract base class for SharedCount<T> and WeakCount<T> that keeps track of use count and weak count.
Divides<T>	Division binary function object.
Duration	Represents a duration in nanoseconds.
EndLine	Represents an end of line character.
EqualTo<T>	An *equal to* relation.
EqualTo2<T, U>	An *equal to* binary predicate.
Exception	A base class for all exception classes.
FrontInsertIterator<C>	An output iterator that inserts elements to the front of a container.
Greater<T>	A *greater than* relation.
Greater2<T, U>	A *greater than* binary predicate.

GreaterOrEqualTo<T>	A *greater than or equal to* relation.
GreaterOrEqualTo2<T, U>	A *greater than or equal to* binary predicate.
Identity<T>	An identity unary function object.
InsertIterator<C>	An output iterator that inserts elements to given position of a container.
Less<T>	A *less than* relation.
Less2<T, U>	A *less than* binary predicate.
LessOrEqualTo<T>	A *less than or equal to* relation.
LessOrEqualTo2<T, U>	A *less than or equal to* binary predicate.
Minus<T>	A subtraction binary function object.
Multiplies<T>	A multiplication binary function object.
Negate<T>	A negation unary operation.
NotEqualTo<T>	An *not equal to* relation.
NotEqualTo2<T, U>	A *not equal to* binary predicate.
Pair<T, U>	A pair of values.
Plus<T>	An addition binary function object.
RandomAccessIter<T, R, P>	A random access iterator that contains a pointer to elements.
Rel<Argument>	A base class for relation function objects.
Remainder<T>	A remainder function object.
SelectFirst<T, U>	A function object for returning the first component of a pair.

SelectSecond<T, U>	A function object for returning the second component of a pair.
ShareableFromThis<T>	A class that implements the “shared from this” idiom.
SharedCount<T>	A handle to a Counter<T> that maintains the use count portion of the counter.
SharedPtr<T>	A shared pointer to an object.
String	A string of ASCII characters.
TimeError	An exception thrown when a time function fails.
TimePoint	Represents a point in time as specified nanoseconds elapsed since epoch.
Tracer	A utility class for tracing entry and exit of some operation.
UnaryFun<Argument, Result>	A base class for unary function objects.
UnaryPred<Argument>	A base class for unary predicates.
UniquePtr<T>	A unique pointer to an object.
WeakCount<T>	A handle to a Counter<T> that maintains the weak count portion of the counter.
WeakPtr<T>	Used to break cycles in shared ownership.
uhuge	128-bit unsigned integer type.
Date	

1.1.1 BackInsertIterator<C> Class

An output iterator that inserts elements to the end of a back insertion sequence.

Syntax

```
public class BackInsertIterator<C>;
```

Constraint

where C is BackInsertionSequence;

Model of

OutputIterator<T>

1.1.1.1 Remarks

BackInserter<C>(C&) is a helper function that returns a **BackInsertIterator<C>** for a back insertion sequence.

1.1.1.2 Example

```
using System;
using System.Collections;

//  Writes:
//  1, 2, 3

void main()
{
    Set<int> s;
    s.Insert(2);
    s.Insert(3);
    s.Insert(1);
//  Set is a sorted container, so it contains now 1, 2, 3...

    List<int> list; // list is a model of a back insertion sequence

//  Copy ints from s to the end of list using BackInsertIterator...
    Copy(s.CBegin(), s.CEnd(), BackInserter(list));

//  ForwardContainers containing ints can be put to OutputStream...
    Console.Out() << list << endl();
}
```

1.1.1.3 Type Definitions

Name	Type	Description

PointerType	BackInsertProxy<C>*	Pointer to implementation defined proxy type.
ReferenceType	BackInsertProxy<C>&	Reference to implementation defined proxy type.
ValueType	BackInsertProxy<C>	Implementation defined proxy type.

1.1.1.4 Member Functions

Member Function	Description
BackInsertIterator()	Constructor. Default constructs a back insert iterator.
BackInsertIterator(C&)	Constructor. Constructs a back insert iterator with a container.
operator++()	Advances the iterator to the next element.
operator->()	Returns a pointer to a proxy object that inserts values to the end of a container.
operator*()	Returns a reference to a proxy object that inserts values to the end of a container.

1.1.1.4.1 BackInsertIterator() Member Function

Constructor. Default constructs a back insert iterator.

Syntax

```
public nothrow BackInsertIterator();
```

1.1.1.4.2 BackInsertIterator(C&) Member Function

Constructor. Constructs a back insert iterator with a container.

Syntax

```
public nothrow BackInsertIterator(C& c);
```

Parameters

Name	Type	Description
c	C&	A container to which to insert elements.

1.1.1.4.3 operator++() Member Function

Advances the iterator to the next element.

Syntax

```
public nothrow BackInsertIterator<C>& operator++();
```

Returns

BackInsertIterator<C>&

Returns a reference to the iterator.

1.1.1.4.4 operator->() Member Function

Returns a pointer to a proxy object that inserts values to the end of a container.

Syntax

```
public nothrow ValueType* operator->();
```

Returns

ValueType*

Returns a pointer to a proxy object that inserts values to the end of a container.

1.1.1.4.5 operator*() Member Function

Returns a reference to a proxy object that inserts values to the end of a container.

Syntax

```
public nothrow ValueType& operator*();
```

Returns

ValueType&

Returns a reference to a proxy object that inserts values to the end of a container.

1.1.2 BinaryFun<Argument1, Argument2, Result> Class

A base class for binary function objects.

Syntax

```
public class BinaryFun<Argument1, Argument2, Result>;
```

Constraint

where Argument1 is Semiregular and Argument2 is Semiregular;

1.1.2.1 Remarks

A derived binary function inherits the type definitions of this base class and provides an implementation for the *operator()(FirstArgumentType, SecondArgumentType)* function.

1.1.2.2 Type Definitions

Name	Type	Description
FirstArgumentType	Argument1	The type of the first argument of the binary function.
ResultType	Result	The type of the result of the binary function.
SecondArgumentType	Argument2	The type of the second argument of the binary function.

1.1.3 BinaryPred<Argument1, Argument2> Class

A base class for binary predicates.

Syntax

```
public class BinaryPred<Argument1, Argument2>;
```

Constraint

where Argument1 is Semiregular and Argument2 is Semiregular;

Model of

BinaryPredicate<T>

Base Class

BinaryFun<Argument1, Argument2, bool>

1.1.3.1 Remarks

A binary predicate is a binary function whose application operator returns a truth value.

1.1.4 Console Class

A static class that contains console input and output functions.

Syntax

`public static class Console;`

1.1.4.1 Remarks

By default **Console** reads from the standard input stream and writes to the standard output stream of the process. It also contains an error stream that by default refers to the standard error stream of the process.

The input, output and error streams that the **Console** class uses can be set with the SetIn(UniquePtr<InputStream>&&), SetOut(UniquePtr<OutputStream>&&) and SetError(UniquePtr<O functions.

1.1.4.2 Example

```
using System;

void main()
{
    Console.WriteLine("Hello, World!");

    Console.WriteLine(10);

    int lineNumber = 1234;
    Console.Error() << "Error in line " << lineNumber << endl();

    Console.WriteLine("What's your name?");
    Console.Write("> ");
    string name = Console.ReadLine();

    Console.Out() << "Hello " << name << '!' << endl();
}
```

1.1.4.3 Member Functions

Member Function	Description
Error()	Returns a reference to the console error stream.
In()	Returns a reference to the console input stream.

Out()	Returns a reference to the console output stream.
ReadLine()	Reads a line of text from the console input stream and returns it.
ReadToEnd()	Returns a string that contains the rest of input of the console input stream.
SetError(UniquePtr<OutputStream>&&)	Sets the error stream the **Console** class uses.
SetIn(UniquePtr<InputStream>&&)	Sets the input stream the **Console** class uses.
SetOut(UniquePtr<OutputStream>&&)	Sets the output stream the **Console** class uses.
Write(const char*)	Writes a C-style string to the console output stream.
Write(const string&)	Writes a string to the console output stream.
Write(bool)	Writes a Boolean value to the console output stream.
Write(byte)	Writes a byte to the console output stream.
Write(char)	Writes a character to the console output stream.
Write(double)	Writes a double value to the console output stream.
Write(float)	Writes a float value to the console output stream.
Write(int)	Writes an int value to the console output stream.
Write(long)	Writes a long value to the console output stream.

Write(sbyte)	Writes a signed byte to the console output stream.
Write(short)	Writes a short value to the console output stream
Write(uint)	Writes an unsigned int to the console output stream.
Write(ulong)	Writes an unsigned long to the console output stream.
Write(ushort)	Writes an unsigned short to the console output stream.
WriteLine()	Writes a newline to the console output stream.
WriteLine(const string&)	Writes a string followed by a newline to the console output stream.
WriteLine(const char*)	Writes a C-style string followed by a newline to the console output stream.
WriteLine(bool)	Writes a Boolean value followed by a newline to the console output stream.
WriteLine(byte)	Writes a byte followed by a newline to the console output stream.
WriteLine(char)	Writes a character followed by a newline to the console output stream.
WriteLine(double)	Writes a double value followed by a newline to the console output stream.
WriteLine(float)	Writes a float value followed by a newline to the console output stream.
WriteLine(int)	Writes an int value followed by a newline to the console output stream.

WriteLine(long)	Writes a long value followed by a newline to the console output stream.
WriteLine(sbyte)	Writes a signed byte followed by a newline to the console output stream.
WriteLine(short)	Writes a short value followed by a newline to the console output stream.
WriteLine(uint)	Writes a unsigned int followed by a newline to the console output stream.
WriteLine(ulong)	Writes an unsigned long followed by a newline to the console output stream.
WriteLine(ushort)	Writes an unsigned short followed by a newline to the console output stream.

1.1.4.3.1 Error() Member Function

Returns a reference to the console error stream.

Syntax

```
public static OutputStream& Error();
```

Returns

OutputStream&

Returns a reference to the console error stream.

1.1.4.3.2 In() Member Function

Returns a reference to the console input stream.

Syntax

```
public static InputStream& In();
```

Returns

InputStream&

Returns a reference to the console input stream.

1.1.4.3.3 Out() Member Function

Returns a reference to the console output stream.

Syntax

```
public static OutputStream& Out();
```

Returns

OutputStream&

Returns a reference to the console output stream.

1.1.4.3.4 ReadLine() Member Function

Reads a line of text from the console input stream and returns it.

Syntax

```
public static string ReadLine();
```

Returns

string

Returns a line of text read from the console input stream.

1.1.4.3.5 ReadToEnd() Member Function

Returns a string that contains the rest of input of the console input stream.

Syntax

```
public static string ReadToEnd();
```

Returns

string

Returns a string that contains the rest of input of the console input stream.

1.1.4.3.6 SetError(UniquePtr<OutputStream>&&) Member Function

Sets the error stream the Console class uses.

Syntax

```
public static void SetError(UniquePtr<OutputStream>&& err_);
```

Parameters

Name	Type	Description
err_	UniquePtr<OutputStream>&&	A unique pointer to an output stream.

1.1.4.3.7 SetIn(UniquePtr<InputStream>&&) Member Function

Sets the input stream the Console class uses.

Syntax

```
public static void SetIn(UniquePtr<InputStream>&& in_);
```

Parameters

Name	Type	Description
in_	UniquePtr<InputStream>&&	A unique pointer to an input stream.

1.1.4.3.8 SetOut(UniquePtr<OutputStream>&&) Member Function

Sets the output stream the Console class uses.

Syntax

```
public static void SetOut(UniquePtr<OutputStream>&& out_);
```

Parameters

Name	Type	Description
out_	UniquePtr<OutputStream>&&	A unique pointer to an output stream.

Example

```
using System;
using System.IO;

void main()
{
    try
    {
        Console.SetOut(UniquePtr<OutputStream>(new OutputFileStream("
            trace.log")));
        Console.WriteLine("this goes to trace.log");
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.ToString() << endl();
    }
```

```
}
```

1.1.4.3.9 Write(const char*) Member Function

Writes a C-style string to the console output stream.

Syntax

```
public static void Write(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

1.1.4.3.10 Write(const string&) Member Function

Writes a string to the console output stream.

Syntax

```
public static void Write(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

1.1.4.3.11 Write(bool) Member Function

Writes a Boolean value to the console output stream.

Syntax

```
public static void Write(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

1.1.4.3.12 Write(byte) Member Function

Writes a byte to the console output stream.

Syntax

```
public static void Write(byte b);
```

Parameters

Name	Type	Description
b	byte	A byte to write.

1.1.4.3.13 Write(char) Member Function

Writes a character to the console output stream.

Syntax

```
public static void Write(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

1.1.4.3.14 Write(double) Member Function

Writes a double value to the console output stream.

Syntax

```
public static void Write(double d);
```

Parameters

Name	Type	Description
d	double	A double value to write.

1.1.4.3.15 Write(float) Member Function

Writes a float value to the console output stream.

Syntax

```
public static void Write(float f);
```

Parameters

Name	Type	Description
f	float	A float value to write.

1.1.4.3.16 Write(int) Member Function

Writes an int value to the console output stream.

Syntax

```
public static void Write(int i);
```

Parameters

Name	Type	Description
i	int	An int value to write.

1.1.4.3.17 Write(long) Member Function

Writes a long value to the console output stream.

Syntax

```
public static void Write(long l);
```

Parameters

Name	Type	Description
l	long	A long to write.

1.1.4.3.18 Write(sbyte) Member Function

Writes a signed byte to the console output stream.

Syntax

```
public static void Write(sbyte s);
```

Parameters

Name	Type	Description
s	sbyte	A signed byte to write.

1.1.4.3.19 Write(short) Member Function

Writes a short value to the console output stream

Syntax

```
public static void Write(short s);
```

Parameters

Name	Type	Description
s	short	A short value to write.

1.1.4.3.20 Write(uint) Member Function

Writes an unsigned int to the console output stream.

Syntax

```
public static void Write(uint u);
```

Parameters

Name	Type	Description
u	uint	An unsigned int to write.

1.1.4.3.21 Write(ulong) Member Function

Writes an unsigned long to the console output stream.

Syntax

```
public static void Write(ulong u);
```

Parameters

Name	Type	Description
u	ulong	An unsigned long to write.

1.1.4.3.22 Write(ushort) Member Function

Writes an unsigned short to the console output stream.

Syntax

```
public static void Write(ushort u);
```

Parameters

Name	Type	Description
u	ushort	An unsigned short to write.

1.1.4.3.23 WriteLine() Member Function

Writes a newline to the console output stream.

Syntax

```
public static void WriteLine();
```

1.1.4.3.24 WriteLine(const string&) Member Function

Writes a string followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

1.1.4.3.25 WriteLine(const char*) Member Function

Writes a C-style string followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

1.1.4.3.26 WriteLine(bool) Member Function

Writes a Boolean value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

1.1.4.3.27 WriteLine(byte) Member Function

Writes a byte followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(byte b);
```

Parameters

Name	Type	Description
b	byte	A byte to write.

1.1.4.3.28 WriteLine(char) Member Function

Writes a character followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

1.1.4.3.29 WriteLine(double) Member Function

Writes a double value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(double d);
```

Parameters

Name	Type	Description
d	double	A double value to write.

1.1.4.3.30 WriteLine(float) Member Function

Writes a float value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(float f);
```

Parameters

Name	Type	Description
f	float	A float value to write.

1.1.4.3.31 WriteLine(int) Member Function

Writes an int value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(int i);
```

Parameters

Name	Type	Description
i	int	An int value to write.

1.1.4.3.32 WriteLine(long) Member Function

Writes a long value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(long l);
```

Parameters

Name	Type	Description
l	long	A long value to write.

1.1.4.3.33 WriteLine(sbyte) Member Function

Writes a signed byte followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(sbyte s);
```

Parameters

Name	Type	Description
s	sbyte	A signed byte to write.

1.1.4.3.34 WriteLine(short) Member Function

Writes a short value followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(short s);
```

Parameters

Name	Type	Description
s	short	A short value to write.

1.1.4.3.35 WriteLine(uint) Member Function

Writes a unsigned int followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(uint u);
```

Parameters

Name	Type	Description
u	uint	An unsigned int to write.

1.1.4.3.36 WriteLine(ulong) Member Function

Writes an unsigned long followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(ulong u);
```

Parameters

Name	Type	Description
u	ulong	An unsigned long to write.

1.1.4.3.37 WriteLine(ushort) Member Function

Writes an unsigned short followed by a newline to the console output stream.

Syntax

```
public static void WriteLine(ushort u);
```

Parameters

Name	Type	Description
u	ushort	An unsigned short to write.

1.1.5 ConversionException Class

An exception thrown when a conversion function fails.

Syntax

```
public class ConversionException;
```

Base Class

Exception

1.1.5.1 Remarks

Conversion functions that throw **ConversionException** are: ParseInt(const string&), ParseUInt(const string&), ParseULong(const string&), ParseHex(const string&), ParseDouble(const string&) and ParseBool(const string&).

1.1.5.2 Example

```
using System;

//  Writes:
//  integer value cannot be parsed from input string '123.456'

void main()
{
    try
    {
        int x = ParseInt("123.456");
    }
    catch (const ConversionException& ex)
    {
        Console.WriteLine(ex.Message());
    }
}
```

1.1.5.3 Member Functions

Member Function	Description
ConversionException(const ConversionException&)	Constructor. Constructs a conversion exception with the given error message.
ConversionException(ConversionException&&)	Move constructor.
ConversionException(const string&)	Copy constructor.

operator=(const ConversionException&)	Copy assignment.
operator=(ConversionException&&)	Move assignment.
~ConversionException()	Destructor.

1.1.5.3.1 ConversionException(const ConversionException&) Member Function

Constructor. Constructs a conversion exception with the given error message.

Syntax

```
public nothrow ConversionException(const ConversionException& that);
```

Parameters

Name	Type	Description
that	const ConversionException&	An error message.

1.1.5.3.2 ConversionException(ConversionException&&) Member Function

Move constructor.

Syntax

```
public nothrow ConversionException(ConversionException&& that);
```

Parameters

Name	Type	Description
that	ConversionException&&	A conversion exception to move from.

1.1.5.3.3 ConversionException(const string&) Member Function

Copy constructor.

Syntax

```
public ConversionException(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	A conversion exception to copy.

1.1.5.3.4 operator=(const ConversionException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const ConversionException& that);
```

Parameters

Name	Type	Description
that	const ConversionException&	A conversion exception to assign.

1.1.5.3.5 operator=(ConversionException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(ConversionException&& that);
```

Parameters

Name	Type	Description
that	ConversionException&&	A conversion exception to move from.

1.1.5.3.6 ~ConversionException() Member Function

Destructor.

Syntax

```
public override nothrow ~ConversionException();
```

1.1.6 Counter<T> Class

A counter class that stores a pointer to the counted object.

Syntax

```
public class Counter<T>;
```

Base Class

CounterBase

1.1.6.1 Remarks

Base class for SharedCount<T> and WeakCount<T>.

1.1.6.2 Member Functions

Member Function	Description
Counter(T*)	Constructor. Stores a pointer to counted object.
Dispose()	Overridden. Deletes the counted object.

1.1.6.2.1 Counter(T*) Member Function

Constructor. Stores a pointer to counted object.

Syntax

```
public nothrow Counter(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to counted object.

1.1.6.2.2 Dispose() Member Function

Overridden. Deletes the counted object.

Syntax

```
public override nothrow void Dispose();
```

1.1.7 CounterBase Class

An abstract base class for SharedCount<T> and WeakCount<T> that keeps track of use count and weak count.

Syntax

```
public abstract class CounterBase;
```

1.1.7.1 Member Functions

Member Function	Description
CounterBase()	Constructor. Sets use count and weak count to one.
~CounterBase()	Destructor.
AddReference()	Increments use count and weak count.
Destruct()	Deletes this counter.
Dispose()	Abstract disposing function overridden in Counter<T> class.
GetUseCount() const	Returns the use count.
Release()	Decrements use count and weak count.
WeakAddReference()	Increments weak count.
WeakRelease()	Decrements weak count.

1.1.7.1.1 CounterBase() Member Function

Constructor. Sets use count and weak count to one.

Syntax

```
public nothrow CounterBase();
```

1.1.7.1.2 ~CounterBase() Member Function

Destructor.

Syntax

```
public virtual nothrow ~CounterBase();
```

1.1.7.1.3 AddReference() Member Function

Increments use count and weak count.

Syntax

```
public inline nothrow void AddReference();
```

1.1.7.1.4 Destruct() Member Function

Deletes this counter.

Syntax

```
public virtual nothrow void Destruct();
```

1.1.7.1.5 Dispose() Member Function

Abstract disposing function overridden in Counter<T> class.

Syntax

```
public abstract nothrow void Dispose();
```

1.1.7.1.6 GetUseCount() const Member Function

Returns the use count.

Syntax

```
public inline nothrow int GetUseCount() const;
```

Returns

int

Returns the use count.

1.1.7.1.7 Release() Member Function

Decrements use count and weak count.

Syntax

```
public inline nothrow void Release();
```

Remarks

If use count has gone to zero, deletes the counted object by calling Dispose() function. If the weak count also has gone to zero, deletes the counter object.

1.1.7.1.8 WeakAddReference() Member Function

Increments weak count.

Syntax

```
public inline nothrow void WeakAddReference();
```

1.1.7.1.9 WeakRelease() Member Function

Decrements weak count.

Syntax

```
public nothrow void WeakRelease();
```

Remarks

If weak count has gone to zero, calls Destruct() to delete this counter object.

1.1.8 Divides<T> Class

Division binary function object.

Syntax

`public class Divides<T>;`

Constraint

where T is MultiplicativeGroup;

Model of

BinaryOperation<T>

Base Class

BinaryFun<T, T, T>

1.1.8.1 Example

```
using System;
using System.Collections;

//   Writes:
//   2

void main()
{
    List<double> divisors;
    divisors.Add(2.0);
    divisors.Add(3.0);
    divisors.Add(4.0);
    double two = Accumulate(divisors.CBegin(), divisors.CEnd(), 48.0,
        Divides<double>());
    Console.WriteLine(two);
}
```

1.1.8.2 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns the first argument divided by the second argument.

1.1.8.2.1 operator()(const T&, const T&) const Member Function

Returns the first argument divided by the second argument.

Syntax

```
public inline nothrow T operator()(const T& a, const T& b) const;
```

Parameters

Name	Type	Description
a	const T&	Dividend.
b	const T&	Divisor.

Returns

T

Returns a/b.

1.1.9 Duration Class

Represents a duration in nanoseconds.

Syntax

```
public class Duration;
```

1.1.9.1 Member Functions

Member Function	Description
Duration()	Default constructor. Initializes the duration to zero nanoseconds.
Duration(Duration&&)	Move constructor.
Duration(const Duration&)	Copy constructor.
Duration(long)	Constructor. Initializes the duration to specified number of nanoseconds.
operator=(const Duration&)	Copy assignment.
operator=(Duration&&)	Move assignment.
FromHours(long)	Returns a duration of specified number of hours.
FromMicroseconds(long)	Returns duration of specified number of microseconds.
FromMilliseconds(long)	Returns duration of specified number of millseconds.
FromMinutes(long)	Returns duration of specified number of minutes.
FromNanoseconds(long)	Returns duration of specified number of nanoseconds.
FromSeconds(long)	Returns duration of specified number of seconds.
Hours() const	Returns total hours elapsed.
Microseconds() const	Returns total microseconds elapsed.
Milliseconds() const	Returns total milliseconds elapsed.

Minutes() const	Returns total minutes elapsed.
Nanoseconds() const	Returns total nanoseconds elapsed.
Rep() const	Returns total nanoseconds elapsed.
Seconds() const	Returns total seconds elapsed.

1.1.9.1.1 Duration() Member Function

Default constructor. Initializes the duration to zero nanoseconds.

Syntax

```
public nothrow Duration();
```

1.1.9.1.2 Duration(Duration&&) Member Function

Move constructor.

Syntax

```
public nothrow Duration(Duration&& that);
```

Parameters

Name	Type	Description
that	Duration&&	A duration to move from.

1.1.9.1.3 Duration(const Duration&) Member Function

Copy constructor.

Syntax

```
public nothrow Duration(const Duration& that);
```

Parameters

Name	Type	Description
that	const Duration&	A duration to copy from.

1.1.9.1.4 Duration(long) Member Function

Constructor. Initializes the duration to specified number of nanoseconds.

Syntax

```
public explicit nothrow Duration(long nanosecs_);
```

Parameters

Name	Type	Description
nanosecs_	long	Nanoseconds.

1.1.9.1.5 operator=(const Duration&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const Duration& that);
```

Parameters

Name	Type	Description
that	const Duration&	A duration to assign from.

1.1.9.1.6 operator=(Duration&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(Duration&& that);
```

Parameters

Name	Type	Description
that	Duration&&	A duration to move from.

1.1.9.1.7 FromHours(long) Member Function

Returns a duration of specified number of hours.

Syntax

```
public static nothrow Duration FromHours(long hours);
```

Parameters

Name	Type	Description
hours	long	Hours.

Returns

Duration

Returns a duration of specified number of hours.

1.1.9.1.8 FromMicroseconds(long) Member Function

Returns duration of specified number of microseconds.

Syntax

```
public static nothrow Duration FromMicroseconds(long microseconds);
```

Parameters

Name	Type	Description
microseconds	long	Microseconds.

Returns

Duration

Returns duration of specified number of microseconds.

1.1.9.1.9 FromMilliseconds(long) Member Function

Returns duration of specified number of millseconds.

Syntax

```
public static nothrow Duration FromMilliseconds(long milliseconds);
```

Parameters

Name	Type	Description
milliseconds	long	Milliseconds.

Returns

Duration

Returns duration of specified number of milliseconds.

1.1.9.1.10 FromMinutes(long) Member Function

Returns duration of specified number of minutes.

Syntax

```
public static nothrow Duration FromMinutes(long minutes);
```

Parameters

Name	Type	Description
minutes	long	Minutes.

Returns

Duration

Returns duration of specified number of minutes.

1.1.9.1.11 FromNanoseconds(long) Member Function

Returns duration of specified number of nanoseconds.

Syntax

```
public static nothrow Duration FromNanoseconds(long nanoseconds);
```

Parameters

Name	Type	Description
nanoseconds	long	Nanoseconds.

Returns

Duration

Returns duration of specified number of nanoseconds.

1.1.9.1.12 FromSeconds(long) Member Function

Returns duration of specified number of seconds.

Syntax

```
public static nothrow Duration FromSeconds(long seconds);
```

Parameters

Name	Type	Description
seconds	long	Seconds.

Returns

Duration

Returns duration of specified number of seconds.

1.1.9.1.13 Hours() const Member Function

Returns total hours elapsed.

Syntax

```
public nothrow long Hours() const;
```

Returns

long

Returns total hours elapsed.

1.1.9.1.14 Microseconds() const Member Function

Returns total microseconds elapsed.

Syntax

```
public nothrow long Microseconds() const;
```

Returns

long

Returns total microseconds elapsed.

1.1.9.1.15 Milliseconds() const Member Function

Returns total milliseconds elapsed.

Syntax

```
public nothrow long Milliseconds() const;
```

Returns

long

Returns total milliseconds elapsed.

1.1.9.1.16 Minutes() const Member Function

Returns total minutes elapsed.

Syntax

```
public nothrow long Minutes() const;
```

Returns

long

Returns total minutes elapsed.

1.1.9.1.17 Nanoseconds() const Member Function

Returns total nanoseconds elapsed.

Syntax

```
public nothrow long Nanoseconds() const;
```

Returns

long

Returns total nanoseconds elapsed.

1.1.9.1.18 Rep() const Member Function

Returns total nanoseconds elapsed.

Syntax

```
public inline nothrow long Rep() const;
```

Returns

long

Returns total nanoseconds elapsed.

1.1.9.1.19 Seconds() const Member Function

Returns total seconds elapsed.

Syntax

```
public nothrow long Seconds() const;
```

Returns

long

Returns total seconds elapsed.

1.1.9.2 Nonmember Functions

Function	Description
operator/(Duration, Duration)	Division of two durations.
operator==(Duration, Duration)	Compares two durations for equality.
operator<(Duration, Duration)	Compares two durations for less than relationship.
operator-(Duration, Duration)	Subtracts a duration from a duration.
operator%(Duration, Duration)	Computes the remainder of division of two durations.

operator+(Duration, Duration)	Computes the sum of two durations.
operator+(Duration, TimePoint)	Computes the sum of a duration and a time point and returns a time point.
operator*(Duration, Duration)	Computes the product of two durations.

1.1.9.2.1 operator/(Duration, Duration) Function

Division of two durations.

Syntax

```
public inline nothrow Duration operator/(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	Dividend.
right	Duration	Divisor.

Returns

Duration

Returns $left$ divided by $right$.

1.1.9.2.2 operator==(Duration, Duration) Function

Compares two durations for equality.

Syntax

```
public inline nothrow bool operator==(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

bool

Returns true, if the first duration is equal to the second duration, false otherwise.

1.1.9.2.3 operator<(Duration, Duration) Function

Compares two durations for less than relationship.

Syntax

```
public inline nothrow bool operator<(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

bool

Returns true, if the first duration is less than the second duration, false otherwise.

1.1.9.2.4 operator-(Duration, Duration) Function

Subtracts a duration from a duration.

Syntax

```
public inline nothrow Duration operator-(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

Duration

Returns $left - right$.

1.1.9.2.5 operator%(Duration, Duration) Function

Computes the remainder of division of two durations.

Syntax

```
public inline nothrow Duration operator%(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

Duration

Returns $left\%right$.

1.1.9.2.6 operator+(Duration, Duration) Function

Computes the sum of two durations.

Syntax

```
public inline nothrow Duration operator+(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

Duration

Returns $left + right$.

1.1.9.2.7 operator+(Duration, TimePoint) Function

Computes the sum of a duration and a time point and returns a time point.

Syntax

```
public inline nothrow TimePoint operator+(Duration d, TimePoint tp);
```

Parameters

Name	Type	Description
d	Duration	A duration.
tp	TimePoint	A time point.

Returns

TimePoint

Returns $d + tp$.

1.1.9.2.8 operator*(Duration, Duration) Function

Computes the product of two durations.

Syntax

```
public inline nothrow Duration operator*(Duration left, Duration right);
```

Parameters

Name	Type	Description
left	Duration	The first duration.
right	Duration	The second duration.

Returns

Duration

Returns $left \times right$.

1.1.10 EndLine Class

Represents an end of line character.

Syntax

```
public class EndLine;
```

1.1.10.1 Remarks

endl() function returns this for dispatching to operator<<(OutputStream&, EndLine) function.

1.1.10.2 Member Functions

Member Function	Description
EndLine()	Default constructor.
EndLine(const EndLine&)	Copy constructor.

1.1.10.2.1 EndLine() Member Function

Default constructor.

Syntax

```
public inline nothrow EndLine();
```

1.1.10.2.2 EndLine(const EndLine&) Member Function

Copy constructor.

Syntax

```
public inline nothrow EndLine(const EndLine& __parameter1);
```

Parameters

Name	Type	Description
__parameter1	const EndLine&	An end of line to copy.

1.1.11 EqualTo<T> Class

An *equal to* relation.

Syntax

```
public class EqualTo<T>;
```

Constraint

where T is Regular;

Model of

Relation<T>

Base Class

Rel<T>

1.1.11.1 Member Functions

Member Function	Description
operator()(const Domain&, const Domain&) const	Returns true if the first argument is equal to the second argument, false otherwise.

1.1.11.1.1 operator()(const Domain&, const Domain&) const Member Function

Returns true if the first argument is equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const Domain& left, const Domain& right)
const;
```

Parameters

Name	Type	Description
left	const Domain&	The first argument.
right	const Domain&	The second argument.

Returns

bool

Returns $left == right$.

1.1.12 EqualTo2<T, U> Class

An *equal to* binary predicate.

Syntax

```
public class EqualTo2<T, U>;
```

Constraint

where EqualityComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.12.1 Remarks

T and U are possibly different types, but can be compared for equality.

1.1.12.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is equal to the second argument, false otherwise.

1.1.12.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left == right$.

1.1.13 Exception Class

A base class for all exception classes.

Syntax

```
public class Exception;
```

1.1.13.1 Remarks

All possible Cmajor exceptions can be catched by catching the **Exception** .

1.1.13.2 Example

```
using System;
using System.IO;

// if file 'nonexistent.file' does not exist, writes a message like
// "System.IO.OpenFileException at 'C:/Programming/cmajor++/system/
    filestream.cm' line 105:
// could not open file 'nonexistent.file' for reading: No such file or
    directory"
// to the standard error stream.

void main()
{
    try
    {
        InputFileStream foo("nonexistent.file");
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.ToString() << endl();
    }
}
```

1.1.13.3 Member Functions

Member Function	Description
Exception()	Default constructor.
Exception(const string&)	Constructor. Constructs an exception with an error message.
Exception(const Exception&)	Copy constructor.
operator=(const Exception&)	Copy assignment.

~Exception()	Destructor.
File() const	Returns the path to the source file where the exception was thrown.
Line() const	Returns the source line number where the exception was thrown.
Message() const	Returns the error message.
SetCallStack(const string&)	Sets the call stack string.
SetFile(const string&)	Sets the source file.
SetLine(int)	Sets the source line number.
ToString() const	Returns the error message with the full name of the thrown exception class, and source file path and source line number where the exception was thrown.

1.1.13.3.1 Exception() Member Function

Default constructor.

Syntax

```
public nothrow Exception();
```

1.1.13.3.2 Exception(const string&) Member Function

Constructor. Constructs an exception with an error message.

Syntax

```
public nothrow Exception(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	An error message.

1.1.13.3.3 Exception(const Exception&) Member Function

Copy constructor.

Syntax

```
public nothrow Exception(const Exception& that);
```

Parameters

Name	Type	Description
that	const Exception&	An exception to copy.

1.1.13.3.4 operator=(const Exception&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const Exception& that);
```

Parameters

Name	Type	Description
that	const Exception&	An exception to assign.

1.1.13.3.5 ~Exception() Member Function

Destructor.

Syntax

```
public virtual nothrow ~Exception();
```

1.1.13.3.6 File() const Member Function

Returns the path to the source file where the exception was thrown.

Syntax

```
public nothrow const string& File() const;
```

Returns

const string&

Returns the path to the source file where the exception was thrown.

1.1.13.3.7 Line() const Member Function

Returns the source line number where the exception was thrown.

Syntax

```
public nothrow int Line() const;
```

Returns

int

Returns the source line number where the exception was thrown.

1.1.13.3.8 Message() const Member Function

Returns the error message.

Syntax

```
public nothrow const string& Message() const;
```

Returns

const string&

Returns the error message.

1.1.13.3.9 SetCallStack(const string&) Member Function

Sets the call stack string.

Syntax

```
public nothrow void SetCallStack(const string& callStack_);
```

Parameters

Name	Type	Description
callStack_	const string&	A call stack string.

1.1.13.3.10 SetFile(const string&) Member Function

Sets the source file.

Syntax

```
public nothrow void SetFile(const string& file_);
```

Parameters

Name	Type	Description
file_	const string&	A path to the source file.

1.1.13.3.11 SetLine(int) Member Function

Sets the source line number.

Syntax

```
public nothrow void SetLine(int line_);
```

Parameters

Name	Type	Description
line_	int	A source line number.

1.1.13.3.12 ToString() const Member Function

Returns the error message with the full name of the thrown exception class, and source file path and source line number where the exception was thrown.

Syntax

```
public virtual string ToString() const;
```

Returns

string

Returns the error message with the full name of the thrown exception class, and source file path and source line number where the exception was thrown.

1.1.14 FrontInsertIterator<C> Class

An output iterator that inserts elements to the front of a container.

Syntax

```
public class FrontInsertIterator<C>;
```

Constraint

where C is FrontInsertionSequence;

Model of

OutputIterator<T>

1.1.14.1 Remarks

FrontInserter<C>(C&) is a helper function that returns a **FrontInsertIterator<C>** for a container.

1.1.14.2 Example

```
using System;
using System.Collections;

//  Writes:
//  list: 3, 2, 1

void main()
{
    Set<int> s;
    s.Insert(2);
    s.Insert(1);
    s.Insert(3);
//  Set is a sorted container, so it contains now 1, 2, 3..

    List<int> list; // list is a model of a front insertion sequence

//  Copy each element in the set to the front of list using
    FrontInsertIterator...
    Copy(s.CBegin(), s.CEnd(), FrontInserter(list));

//  ForwardContainers containing ints can be put to OutputStream...
    Console.Out() << "list: " << list << endl();
}
```

1.1.14.3 Type Definitions

Name	Type	Description
PointerType	FrontInsertProxy<C>*	Pointer to implementation defined proxy type.
ReferenceType	FrontInsertProxy<C>&	Reference to implementation defined proxy type.
ValueType	FrontInsertProxy<C>	Implementation defined proxy type.

1.1.14.4 Member Functions

Member Function	Description
FrontInsertIterator()	Constructor. Default constructs a front insert iterator.
FrontInsertIterator(C&)	Copy constructor.
operator++()	Advances the iterator to the next element.
operator->()	Returns a pointer to a proxy object that inserts values at the beginning of a container.
operator*()	Returns a reference to a proxy object that inserts values at the beginning of a container.

1.1.14.4.1 FrontInsertIterator() Member Function

Constructor. Default constructs a front insert iterator.

Syntax

```
public nothrow FrontInsertIterator();
```

1.1.14.4.2 FrontInsertIterator(C&) Member Function

Copy constructor.

Syntax

```
public nothrow FrontInsertIterator(C& c);
```

Parameters

Name	Type	Description
c	C&	A front insert iterator to copy.

1.1.14.4.3 operator++() Member Function

Advances the iterator to the next element.

Syntax

```
public nothrow FrontInsertIterator<C>& operator++();
```

Returns

FrontInsertIterator<C>&

Returns a reference to the iterator.

1.1.14.4.4 operator->() Member Function

Returns a pointer to a proxy object that inserts values at the beginning of a container.

Syntax

```
public nothrow ValueType* operator->();
```

Returns

ValueType*

Returns a pointer to a proxy object that inserts values at the beginning of a container.

1.1.14.4.5 operator*() Member Function

Returns a reference to a proxy object that inserts values at the beginning of a container.

Syntax

```
public nothrow ValueType& operator*();
```

Returns

ValueType&

Returns a reference to a proxy object that inserts values at the beginning of a container.

1.1.15 Greater<T> Class

A *greater than* relation.

Syntax

`public class Greater<T>;`

Constraint

where T is LessThanComparable;

Model of

Relation<T>

Base Class

Rel<T>

1.1.15.1 Example

```
using System;
using System.Collections;

//  Writes:
//  3, 2, 1

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(2);
    list.Add(3);
    Sort(list, Greater<int>());
    Console.Out() << list << endl();
}
```

1.1.15.2 Example

```
using System;
using System.Collections;
using System.IO;

//  Writes:
//  foo, baz, bar

class A
{
    public A(): id()
```

```
    {
    }
    public A(const string& id_): id(id_)
    {
    }
    public nothrow inline const string& Id() const
    {
        return id;
    }
    private string id;
}

// Note: need only to provide less than operator for A -- compiler
   implements >, <= and >=.

public nothrow inline bool operator<(const A& left, const A& right)
{
    return left.Id() < right.Id();
}

public OutputStream& operator<<(OutputStream& s, const List<A>& list)
{
    bool first = true;
    for (const A& a : list)
    {
        if (first)
        {
            first = false;
        }
        else
        {
            s.Write(", ");
        }
        s.Write(a.Id());
    }
    return s;
}

void main()
{
    List<A> list;
    A foo("foo");
    list.Add(foo);
    A bar("bar");
    list.Add(bar);
    A baz("baz");
    list.Add(baz);
    Sort(list, Greater<A>());
    Console.Out() << list << endl();
}
```

1.1.15.3 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns true if the first argument is greater than the second argument, false otherwise.

1.1.15.3.1 operator()(const T&, const T&) const Member Function

Returns true if the first argument is greater than the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const T& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const T&	The second argument.

Returns

bool

Returns $left > right$.

1.1.16 Greater2<T, U> Class

A *greater than* binary predicate.

Syntax

```
public class Greater2<T, U>;
```

Constraint

where LessThanComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.16.1 Remarks

T and U are possibly different types, but can be compared for less than relationship.

1.1.16.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is greater than the second argument, false otherwise.

1.1.16.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is greater than the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left > right$.

1.1.17 GreaterOrEqualTo<T> Class

A *greater than or equal to* relation.

Syntax

```
public class GreaterOrEqualTo<T>;
```

Constraint

where T is LessThanComparable;

Model of

Relation<T>

Base Class

Rel<T>

1.1.17.1 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns true if the first argument is greater than or equal to the second argument, false otherwise.

1.1.17.1.1 operator()(const T&, const T&) const Member Function

Returns true if the first argument is greater than or equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const T& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const T&	The second argument.

Returns

bool

Returns $left >= right$.

1.1.18 GreaterOrEqualTo2<T, U> Class

A *greater than or equal to* binary predicate.

Syntax

```
public class GreaterOrEqualTo2<T, U>;
```

Constraint

where LessThanComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.18.1 Remarks

T and U are possible different types, but can be compared for less than relationship.

1.1.18.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is greater than or equal to the second argument, false otherwise.

1.1.18.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is greater than or equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left >= right$.

1.1.19 Identity<T> Class

An identity unary function object.

Syntax

```
public class Identity<T>;
```

Constraint

where T is Semiregular;

Model of

UnaryOperation<T>

Base Class

UnaryFun<T, T>

1.1.19.1 Member Functions

Member Function	Description
operator()(const T&) const	Returns the argument.

1.1.19.1.1 operator()(const T&) const Member Function

Returns the argument.

Syntax

```
public inline nothrow const T& operator()(const T& x) const;
```

Parameters

Name	Type	Description
x	const T&	An argument.

Returns

const T&

Returns x.

1.1.20 InsertIterator<C> Class

An output iterator that inserts elements to given position of a container.

Syntax

```
public class InsertIterator<C>;
```

Constraint

where C is InsertionSequence;

Model of

OutputIterator<T>

1.1.20.1 Remarks

System.Inserter.C.I.C.is.InsertionSequence.I.is.C.Iterator.C.ref.I is a helper function that returns a **InsertIterator<C>** for a container and position.

1.1.20.2 Example

```
using System;
using System.Collections;

//  Writes:
//  0, 1, 2, 3, 4

void main()
{
    Set<int> s;
    s.Insert(3);
    s.Insert(2);
    s.Insert(1);
//  Set is a sorted container, so it contains now 1, 2, 3...

    List<int> list;
    list.Add(0);
    list.Add(4);

//  Copy the set in the middle of the list using InsertIterator...
    Copy(s.CBegin(), s.CEnd(), Inserter(list, list.Begin() + 1));

//  ForwardContainers containing ints can be put to OutputStream...
    Console.Out() << list << endl();
}
```

1.1.20.3 Type Definitions

Name	Type	Description
PointerType	InsertProxy<C>*	Pointer to an implementation defined proxy type.
ReferenceType	InsertProxy<C>&	Reference to an implementation defined proxy type.
ValueType	InsertProxy<C>	An implementation defined proxy type.

1.1.20.4 Member Functions

Member Function	Description
InsertIterator()	Constructor. Default constructs an insert iterator.
InsertIterator(C&, C.Iterator)	Constructor. Constructs an insert iterator with the given container and container iterator.
operator++()	Advances the insert iterator to the next element.
operator->()	Returns a pointer to a proxy type that inserts values to the container.
operator*()	Returns a reference to a proxy type that inserts values to the container.

1.1.20.4.1 InsertIterator() Member Function

Constructor. Default constructs an insert iterator.

Syntax

```
public nothrow InsertIterator();
```

1.1.20.4.2 InsertIterator(C&, C.Iterator) Member Function

Constructor. Constructs an insert iterator with the given container and container iterator.

Syntax

```
public nothrow InsertIterator(C& c, C.Iterator i);
```

Parameters

Name	Type	Description

c	C&	A container to which to insert elements.
i	C.Iterator	An iterator pointing to a position in the container to which to insert elements.

1.1.20.4.3 operator++() Member Function

Advances the insert iterator to the next element.

Syntax

```
public nothrow InsertIterator<C>& operator++();
```

Returns

InsertIterator<C>&

Returns a reference to the iterator.

1.1.20.4.4 operator->() Member Function

Returns a pointer to a proxy type that inserts values to the container.

Syntax

```
public nothrow ValueType* operator->();
```

Returns

ValueType*

Returns a pointer to a proxy type that inserts values to the container.

1.1.20.4.5 operator*() Member Function

Returns a reference to a proxy type that inserts values to the container.

Syntax

```
public nothrow ValueType& operator*();
```

Returns

ValueType&

Returns a reference to a proxy type that inserts values to the container.

1.1.21 Less<T> Class

A *less than* relation.

Syntax

```
public class Less<T>;
```

Constraint

where T is LessThanComparable;

Model of

Relation<T>

Base Class

Rel<T>

1.1.21.1 Example

```
using System;
using System.Collections;

//  Writes:
//  1, 2, 3

void main()
{
    List<int> list;
    list.Add(3);
    list.Add(2);
    list.Add(1);
    Sort(list, Less<int>());
    Console.Out() << list << endl();
}
```

1.1.21.2 Example

```
using System;
using System.Collections;
using System.IO;

//  Writes:
//  bar, baz, foo

class A
{
    public A(): id()
```

```
    {
    }
    public A(const string& id_): id(id_)
    {
    }
    public nothrow inline const string& Id() const
    {
        return id;
    }
    private string id;
}

public nothrow inline bool operator<(const A& left, const A& right)
{
    return left.Id() < right.Id();
}

public OutputStream& operator<<(OutputStream& s, const List<A>& list)
{
    bool first = true;
    for (const A& a : list)
    {
        if (first)
        {
            first = false;
        }
        else
        {
            s.Write(", ");
        }
        s.Write(a.Id());
    }
    return s;
}

void main()
{
    List<A> list;
    A foo("foo");
    list.Add(foo);
    A bar("bar");
    list.Add(bar);
    A baz("baz");
    list.Add(baz);
    Sort(list, Less<A>());
    Console.Out() << list << endl();
}
```

1.1.21.3 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns true if the first argument is less than the second argument, false otherwise.

1.1.21.3.1 operator()(const T&, const T&) const Member Function

Returns true if the first argument is less than the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const T& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const T&	The second argument.

Returns

bool

Returns $left < right$.

1.1.22 Less2<T, U> Class

A *less than* binary predicate.

Syntax

```
public class Less2<T, U>;
```

Constraint

where LessThanComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.22.1 Remarks

T and U are possibly different types, but can be compared for less than relationship.

1.1.22.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is less than the second argument.

1.1.22.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is less than the second argument.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left < right$.

1.1.23 LessOrEqualTo<T> Class

A *less than or equal to* relation.

Syntax

```
public class LessOrEqualTo<T>;
```

Constraint

where T is LessThanComparable;

Model of

Relation<T>

Base Class

Rel<T>

1.1.23.1 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns true if the first argument is less than or equal to the second argument, false otherwise.

1.1.23.1.1 operator()(const T&, const T&) const Member Function

Returns true if the first argument is less than or equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const T& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const T&	The second argument.

Returns

bool

Returns $left <= right$.

1.1.24 LessOrEqualTo2<T, U> Class

A *less than or equal to* binary predicate.

Syntax

```
public class LessOrEqualTo2<T, U>;
```

Constraint

where LessThanComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.24.1 Remarks

T and U are possibly different types, but can be compared for less than relationship.

1.1.24.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is less than or equal to the second argument, false otherwise.

1.1.24.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is less than or equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left <= right$.

1.1.25 Minus<T> Class

A subtraction binary function object.

Syntax

```
public class Minus<T>;
```

Constraint

where T is AdditiveGroup;

Model of

BinaryFunction<T>

Base Class

BinaryFun<T, T, T>

1.1.25.1 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns the difference of the the first and second argument.

1.1.25.1.1 operator()(const T&, const T&) const Member Function

Returns the difference of the the first and second argument.

Syntax

```
public inline nothrow T operator()(const T& a, const T& b) const;
```

Parameters

Name	Type	Description
a	const T&	The first argument.
b	const T&	The second argument.

Returns

T

Returns $a - b$.

1.1.26 Multiplies<T> Class

A multiplication binary function object.

Syntax

```
public class Multiplies<T>;
```

Constraint

where T is MultiplicativeSemigroup;

Model of

BinaryOperation<T>

Base Class

BinaryFun<T, T, T>

1.1.26.1 Example

```
using System;
using System.Collections;

//  Writes:
//  48

void main()
{
    List<int> ints;
    ints.Add(2);
    ints.Add(4);
    ints.Add(6);
    int init = 1;
    int product = Accumulate(ints.CBegin(), ints.CEnd(), init, Multiplies
        <int>());
    Console.WriteLine(product);
}
```

1.1.26.2 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns the first argument multiplied with the second argument.

1.1.26.2.1 operator()(const T&, const T&) const Member Function

Returns the first argument multiplied with the second argument.

Syntax

```
public inline nothrow T operator()(const T& a, const T& b) const;
```

Parameters

Name	Type	Description
a	const T&	The first argument.
b	const T&	The second argument.

Returns

T

Returns $a * b$.

1.1.27 Negate<T> Class

A negation unary operation.

Syntax

```
public class Negate<T>;
```

Constraint

where T is AdditiveGroup;

Model of

UnaryOperation<T>

Base Class

UnaryFun<T, T>

1.1.27.1 Member Functions

Member Function	Description
operator()(const ResultType&) const	Returns the negation of its argument.

1.1.27.1.1 operator()(const ResultType&) const Member Function

Returns the negation of its argument.

Syntax

```
public inline nothrow ResultType operator()(const ResultType& a) const;
```

Parameters

Name	Type	Description
a	const ResultType&	The argument.

Returns

ResultType

Returns $-a$.

1.1.28 NotEqualTo<T> Class

An *not equal to* relation.

Syntax

```
public class NotEqualTo<T>;
```

Constraint

where T is Regular;

Model of

Relation<T>

Base Class

Rel<T>

1.1.28.1 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns true if the first argument is not equal to the second argument, false otherwise.

1.1.28.1.1 operator()(const T&, const T&) const Member Function

Returns true if the first argument is not equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const T& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const T&	The second argument.

Returns

bool

Returns $left \neq right$.

1.1.29 NotEqualTo2<T, U> Class

A *not equal to* binary predicate.

Syntax

```
public class NotEqualTo2<T, U>;
```

Constraint

where EqualityComparable<T, U>;

Model of

Relation<T, U, V>

Base Class

BinaryPred<T, U>

1.1.29.1 Remarks

T and U are possibly different types, but can be compared for equality.

1.1.29.2 Member Functions

Member Function	Description
operator()(const T&, const U&) const	Returns true if the first argument is not equal to the second argument, false otherwise.

1.1.29.2.1 operator()(const T&, const U&) const Member Function

Returns true if the first argument is not equal to the second argument, false otherwise.

Syntax

```
public inline nothrow bool operator()(const T& left, const U& right) const;
```

Parameters

Name	Type	Description
left	const T&	The first argument.
right	const U&	The second argument.

Returns

bool

Returns $left$!= $right$.

1.1.30 Pair<T, U> Class

A pair of values.

Syntax

```
public class Pair<T, U>;
```

Constraint

where T is Semiregular and U is Semiregular;

1.1.30.1 Remarks

The **Pair<T, U>** class is used for example in the system library algorithm EqualRange<I, T>(I, I, const T&) to return a pair of iterators, and in the system library Map<Key, Value, KeyCompare> class to compose a value type from a key type and a mapped type.

1.1.30.2 Member Functions

Member Function	Description
Pair()	Constructor. Constructs a pair with default values.
Pair(const T&, const U&)	Constructor. Constructs a pair with specified values.
Pair(T&&, U&&)	Constructor. Constructs a pair with given rvalues.

1.1.30.2.1 Pair() Member Function

Constructor. Constructs a pair with default values.

Syntax

```
public Pair();
```

1.1.30.2.2 Pair(const T&, const U&) Member Function

Constructor. Constructs a pair with specified values.

Syntax

```
public Pair(const T& first_, const U& second_);
```

Parameters

Name	Type	Description
first_	const T&	The first value.

second_	const U&	The second value.

1.1.30.2.3 Pair(T&&, U&&) Member Function

Constructor. Constructs a pair with given rvalues.

Syntax

```
public Pair(T&& first_, U&& second_);
```

Parameters

Name	Type	Description
first_	T&&	The first value.
second_	U&&	The second value.

1.1.30.3 Nonmember Functions

Function	Description
operator==<T, U>(const ArgumentType&, const ArgumentType&)	Compares two pairs for equality.
operator<<T, U>(const ArgumentType&, const ArgumentType&)	Compares two pairs for less than relationship.

1.1.30.3.1 operator==<T, U>(const ArgumentType&, const ArgumentType&) Function

Compares two pairs for equality.

Syntax

```
public nothrow bool operator==<T, U>(const ArgumentType& left, const ArgumentType&
right);
```

Constraint

where T is Regular and U is Regular;

Parameters

Name	Type	Description
left	const ArgumentType&	The first pair.
right	const ArgumentType&	The second pair.

Returns

bool

Returns true if the first component of the first pair is equal to the first component of the second pair and the second component of the first pair is equal to the second component of the second pair, false otherwise.

1.1.30.3.2 operator<<T, U>(const ArgumentType&, const ArgumentType&) Function

Compares two pairs for less than relationship.

Syntax

```
public nothrow bool operator<<T, U>(const ArgumentType& left, const ArgumentType&
right);
```

Constraint

where T is TotallyOrdered and U is TotallyOrdered;

Parameters

Name	Type	Description
left	const ArgumentType&	The first pair.
right	const ArgumentType&	The second pair.

Returns

bool

Returns true if the first component of the first pair is less than the first component of the second pair. Returns false if the first component of the second pair is less than the first component of the first pair. Returns true if the second component of the first pair is less than the second component of the second pair. Otherwise returns false.

1.1.31 Plus<T> Class

An addition binary function object.

Syntax

```
public class Plus<T>;
```

Constraint

where T is AdditiveSemigroup;

Model of

BinaryOperation<T>

Base Class

BinaryFun<T, T, T>

1.1.31.1 Example

```
using System;
using System.Collections;

//  Writes:
//  6

void main()
{
    List<int> ints;
    ints.Add(1);
    ints.Add(2);
    ints.Add(3);
    int init = 0;
    int sum = Accumulate(ints.CBegin(), ints.CEnd(), init, Plus<int>());
    Console.WriteLine(sum);
}
```

1.1.31.2 Member Functions

Member Function	Description
operator()(const ResultType&, const ResultType&) const	Returns the sum of the first argument and the second argument.

1.1.31.2.1 operator()(const ResultType&, const ResultType&) const Member Function

Returns the sum of the first argument and the second argument.

Syntax

```
public inline nothrow ResultType operator()(const ResultType& a, const ResultType&
b) const;
```

Parameters

Name	Type	Description
a	const ResultType&	the first argument.
b	const ResultType&	The second argument.

Returns

ResultType

Returns $a + b$.

1.1.32 RandomAccessIter<T, R, P> Class

A random access iterator that contains a pointer to elements.

Syntax

```
public class RandomAccessIter<T, R, P>;
```

Model of

RandomAccessIterator<T>

1.1.32.1 Remarks

List<T> and String classes implement their iterator types using the **RandomAccessIter<T, R, P>** class.

1.1.32.2 Type Definitions

Name	Type	Description
PointerType	P	A pointer to an element.
ReferenceType	R	A reference to an element.
ValueType	T	The type of an element.

1.1.32.3 Member Functions

Member Function	Description
RandomAccessIter()	Constructor. Default constructs a random access iterator.
RandomAccessIter(PointerType)	Constructor. Constructs a random access iterator with a pointer to elements.
operator[](int) const	Returns a reference to an element with a given index.
operator–()	Backs the random access iterator to point to the preceding element.
operator++()	Advances the random access iterator to point to the succeeding element.
operator->() const	Returns a pointer to the element currently pointed to.

operator*() const	Returns a reference to the element currently pointed to.
GetPtr() const	Returns the contained pointer.

1.1.32.3.1 RandomAccessIter() Member Function

Constructor. Default constructs a random access iterator.

Syntax

```
public inline nothrow RandomAccessIter();
```

1.1.32.3.2 RandomAccessIter(PointerType) Member Function

Constructor. Constructs a random access iterator with a pointer to elements.

Syntax

```
public explicit inline nothrow RandomAccessIter(PointerType ptr_);
```

Parameters

Name	Type	Description
ptr_	PointerType	A pointer to elements.

1.1.32.3.3 operator[](int) const Member Function

Returns a reference to an element with a given index.

Syntax

```
public inline nothrow ReferenceType operator[](int index) const;
```

Parameters

Name	Type	Description
index	int	An index.

Returns

ReferenceType

Returns *ptr*[*index*] where *ptr* is the contained pointer to elements.

1.1.32.3.4 operator–() Member Function

Backs the random access iterator to point to the preceding element.

Syntax

```
public inline nothrow RandomAccessIter<T, R, P>& operator--();
```

Returns

RandomAccessIter<T, R, P>&

Returns a reference to the random access iterator.

1.1.32.3.5 operator++() Member Function

Advances the random access iterator to point to the succeeding element.

Syntax

```
public inline nothrow RandomAccessIter<T, R, P>& operator++();
```

Returns

RandomAccessIter<T, R, P>&

Returns a reference to the random access iterator.

1.1.32.3.6 operator->() const Member Function

Returns a pointer to the element currently pointed to.

Syntax

```
public inline nothrow PointerType operator->() const;
```

Returns

PointerType

Returns *ptr*, where *ptr* is the contained pointer.

1.1.32.3.7 operator*() const Member Function

Returns a reference to the element currently pointed to.

Syntax

```
public inline nothrow ReferenceType operator*() const;
```

Returns

ReferenceType

Returns $*ptr$, where *ptr* is the contained pointer.

1.1.32.3.8 GetPtr() const Member Function

Returns the contained pointer.

Syntax

```
public inline nothrow PointerType GetPtr() const;
```

Returns

PointerType

Returns *ptr*, where *ptr* is the contained pointer.

1.1.32.4 Nonmember Functions

Function	Description
operator==<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIter<T, R, P>&)	Compares two random access iterators for equality.
operator<<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIter<T, R, P>&)	Compares two random access iterators for less than relationship.
operator-<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIter<T, R, P>&)	Returns the distance between two random access iterators.
operator-<T, R, P>(const RandomAccessIter<T, R, P>&, int)	Returns the difference of a random access iterator and an integer.
operator+<T, R, P>(const RandomAccessIter<T, R, P>&, int)	Returns a random access iterator advanced by the given integer offset.
operator+<T, R, P>(int, const RandomAccessIter<T, R, P>&)	Returns a random access iterator advanced by the given integer offset.

1.1.32.4.1 operator==<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIter<T, R, P>&) Function

Compares two random access iterators for equality.

Syntax

```
public inline nothrow bool operator==<T, R, P>(const RandomAccessIter<T, R, P>&
left, const RandomAccessIter<T, R, P>& right);
```

Parameters

Name	Type	Description
left	const RandomAccessIter<T, R, P>&	The first random access iterator.
right	const RandomAccessIter<T, R, P>&	The second random access iterator.

Returns

bool

Returns true if *left* points to the same element as *right* or both are end iterators, false otherwise.

1.1.32.4.2 operator<<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIte R, P>&) Function

Compares two random access iterators for less than relationship.

Syntax

```
public inline nothrow bool operator<<T, R, P>(const RandomAccessIter<T, R, P>&
left, const RandomAccessIter<T, R, P>& right);
```

Parameters

Name	Type	Description
left	const RandomAccessIter<T, R, P>&	The first random access iterator.
right	const RandomAccessIter<T, R, P>&	The second random access iterator.

Returns

bool

Returns true if the element pointed by *left* comes before the element pointed by *right* in the sequence, false otherwise.

Remarks

Both iterators must point to an element of the same sequence or both must be end iterators.

1.1.32.4.3 operator-<T, R, P>(const RandomAccessIter<T, R, P>&, const RandomAccessIter R, P>&) Function

Returns the distance between two random access iterators.

Syntax

```
public inline nothrow int operator-<T, R, P>(const RandomAccessIter<T, R, P>&
left, const RandomAccessIter<T, R, P>& right);
```

Parameters

Name	Type	Description
left	const RandomAccessIter<T, R, P>&	The first random access iterator.
right	const RandomAccessIter<T, R, P>&	The second random access iterator.

Returns

int

Returns the difference of the pointer contained by *left* and the pointer contained by *right*.

Remarks

Both iterators must point to an element in the same sequence or both must be end iterators.

1.1.32.4.4 operator-<T, R, P>(const RandomAccessIter<T, R, P>&, int) Function

Returns the difference of a random access iterator and an integer.

Syntax

```
public inline nothrow RandomAccessIter<T, R, P> operator-<T, R, P>(const RandomAccessIte
R, P>& it, int offset);
```

Parameters

Name	Type	Description
it	const RandomAccessIter<T, R, P>&	A random access iterator.
offset	int	An integer.

Returns

RandomAccessIter<T, R, P>

Returns a random access iterator that comes *offset* elements before the *it*.

1.1.32.4.5 operator+<T, R, P>(const RandomAccessIter<T, R, P>&, int) Function

Returns a random access iterator advanced by the given integer offset.

Syntax

```
public inline nothrow RandomAccessIter<T, R, P> operator+<T, R, P>(const RandomAccessIte
R, P>& it, int offset);
```

Parameters

Name	Type	Description
it	const RandomAccessIter<T, R, P>&	A random access iterator.
offset	int	An integer offset.

Returns

RandomAccessIter<T, R, P>

Returns a random access iterator advanced by the given integer offset.

1.1.32.4.6 operator+<T, R, P>(int, const RandomAccessIter<T, R, P>&) Function

Returns a random access iterator advanced by the given integer offset.

Syntax

```
public inline nothrow RandomAccessIter<T, R, P> operator+<T, R, P>(int offset,
const RandomAccessIter<T, R, P>& it);
```

Parameters

Name	Type	Description
offset	int	An integer offset.
it	const RandomAccessIter<T, R, P>&	A random access iterator.

Returns

RandomAccessIter<T, R, P>

Returns a random access iterator advanced by the given integer offset.

1.1.33 Rel<Argument> Class

A base class for relation function objects.

Syntax

```
public class Rel<Argument>;
```

Constraint

where Argument is Semiregular;

Model of

Relation<T>

Base Class

BinaryPred<Argument, Argument>

1.1.33.1 Remarks

EqualTo<T>, NotEqualTo<T>, Less<T>, Greater<T>, LessOrEqualTo<T> and GreaterOrEqualTo<T> classes derive from the **Rel<Argument>** class.

1.1.33.2 Type Definitions

Name	Type	Description
Domain	Argument	The domain i.e. the type of the argument of the relation.

1.1.34 Remainder<T> Class

A remainder function object.

Syntax

```
public class Remainder<T>;
```

Constraint

where T is EuclideanSemiring;

Model of

BinaryFunction<T>

Base Class

BinaryFun<T, T, T>

1.1.34.1 Member Functions

Member Function	Description
operator()(const T&, const T&) const	Returns the remainder of division of the first argument and the second argument.

1.1.34.1.1 operator()(const T&, const T&) const Member Function

Returns the remainder of division of the first argument and the second argument.

Syntax

```
public inline nothrow T operator()(const T& a, const T& b) const;
```

Parameters

Name	Type	Description
a	const T&	The first argument.
b	const T&	The second argument.

Returns

T

Returns a % b.

1.1.35 SelectFirst<T, U> Class

A function object for returning the first component of a pair.

Syntax

```
public class SelectFirst<T, U>;
```

Constraint

where T is Semiregular and U is Semiregular;

Base Class

UnaryFun<Pair<T, U>, T>

1.1.35.1 Member Functions

Member Function	Description
operator()(const ArgumentType&) const	Returns the first component of the given pair.

1.1.35.1.1 operator()(const ArgumentType&) const Member Function

Returns the first component of the given pair.

Syntax

```
public nothrow const ResultType& operator()(const ArgumentType& p) const;
```

Parameters

Name	Type	Description
p	const ArgumentType&	A pair of values.

Returns

const ResultType&

Returns the first component of *p*.

1.1.36 SelectSecond<T, U> Class

A function object for returning the second component of a pair.

Syntax

```
public class SelectSecond<T, U>;
```

Constraint

where T is Semiregular and U is Semiregular;

Base Class

UnaryFun<Pair<T, U>, U>

1.1.36.1 Member Functions

Member Function	Description
operator()(const ArgumentType&) const	Returns the second component of the given pair.

1.1.36.1.1 operator()(const ArgumentType&) const Member Function

Returns the second component of the given pair.

Syntax

```
public nothrow const ResultType& operator()(const ArgumentType& p) const;
```

Parameters

Name	Type	Description
p	const ArgumentType&	A pair of values.

Returns

const ResultType&

Returns the second component of *p*.

1.1.37 ShareableFromThis<T> Class

A class that implements the "shared from this" idiom.

Syntax

```
public class ShareableFromThis<T>;
```

1.1.37.1 Remarks

By deriving a class from **ShareableFromThis<T>**
with itself as the template argument, you can obtain a shared pointer to the class in its member functions (other that constructors) provided that there is a "living" SharedPtr<T> to the object.

1.1.37.2 Example

```
using System;

public class C: ShareableFromThis<C>
{
    public SharedPtr<C> mf()
    {
        return GetSharedFromThis();
    }
}

void main()
{
    SharedPtr<C> c(new C());
//  ...
    C* rawPtr = c.GetPtr();
//  ...
    SharedPtr<C> p = rawPtr->mf();
}
```

1.1.37.3 Member Functions

Member Function	Description
GetSharedFromThis() const	Returns a shared pointer to the class object.
GetWeakThis()	Returns the contained weak pointer to the class object.

1.1.37.3.1 GetSharedFromThis() const Member Function

Returns a shared pointer to the class object.

Syntax

```
public nothrow SharedPtr<T> GetSharedFromThis() const;
```

Returns

SharedPtr<T>

Returns a shared pointer to the class object.

1.1.37.3.2 GetWeakThis() Member Function

Returns the contained weak pointer to the class object.

Syntax

```
public nothrow WeakPtr<T>& GetWeakThis();
```

Returns

WeakPtr<T>&

Returns the contained weak pointer to the class object.

1.1.38 SharedCount<T> Class

A handle to a Counter<T> that maintains the use count portion of the counter.

Syntax

```
public class SharedCount<T>;
```

1.1.38.1 Member Functions

Member Function	Description
SharedCount()	Constructor. Initializes an empty shared count.
SharedCount(T*)	Constructor. Initializes the counter to contain a pointer to a counted object.
SharedCount(const SharedCount<T>&)	Constructor. Implementation detail.
SharedCount(Counter<T>*)	Copy constructor. Increments the use count.
SharedCount(const WeakCount<T>&)	Constructor. Gets the counter from a WeakCount<T>.
operator=(const SharedCount<T>&)	Copy assignment. Decrements the use count of the old counter and increments the use count of the copied counter.
~SharedCount()	Destructor. Decrements the use count.
GetCounter() const	Returns the contained pointer to a counter.
GetUseCount() const	Returns the use count of the counter.
IsUnique() const	Returns true if there is exactly one SharedPtr<T> to an object.
Swap(SharedCount<T>&)	Exchanges the contents with another shared count.

1.1.38.1.1 SharedCount() Member Function

Constructor. Initializes an empty shared count.

Syntax

```
public nothrow SharedCount();
```

1.1.38.1.2 SharedCount(T*) Member Function

Constructor. Initializes the counter to contain a pointer to a counted object.

Syntax

```
public nothrow SharedCount(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to a counted object.

1.1.38.1.3 SharedCount(const SharedCount<T>&) Member Function

Constructor. Implementation detail.

Syntax

```
public nothrow SharedCount(const SharedCount<T>& that);
```

Parameters

Name	Type	Description
that	const SharedCount<T>&	Pointer to the counter.

1.1.38.1.4 SharedCount(Counter<T>*) Member Function

Copy constructor. Increments the use count.

Syntax

```
public nothrow SharedCount(Counter<T>* counter_);
```

Parameters

Name	Type	Description
counter_	Counter<T>*	A shared count to copy.

1.1.38.1.5 SharedCount(const WeakCount<T>&) Member Function

Constructor. Gets the counter from a WeakCount<T>.

Syntax

```
public nothrow SharedCount(const WeakCount<T>& that);
```

Parameters

Name	Type	Description
that	const WeakCount<T>&	A weak count.

1.1.38.1.6 operator=(const SharedCount<T>&) Member Function

Copy assignment. Decrements the use count of the old counter and increments the use count of the copied counter.

Syntax

```
public nothrow void operator=(const SharedCount<T>& that);
```

Parameters

Name	Type	Description
that	const SharedCount<T>&	A shared count to assign.

1.1.38.1.7 ~SharedCount() Member Function

Destructor. Decrements the use count.

Syntax

```
public nothrow ~SharedCount();
```

1.1.38.1.8 GetCounter() const Member Function

Returns the contained pointer to a counter.

Syntax

```
public nothrow Counter<T>* GetCounter() const;
```

Returns

Counter<T>*

Returns the contained pointer to a counter.

1.1.38.1.9 GetUseCount() const Member Function

Returns the use count of the counter.

Syntax

```
public nothrow int GetUseCount() const;
```

Returns

int

Returns the use count of the counter.

1.1.38.1.10 IsUnique() const Member Function

Returns true if there is exactly one SharedPtr<T> to an object.

Syntax

```
public nothrow bool IsUnique() const;
```

Returns

bool

Returns true if there is exactly one SharedPtr<T> to an object.

1.1.38.1.11 Swap(SharedCount<T>&) Member Function

Exchanges the contents with another shared count.

Syntax

```
public nothrow void Swap(SharedCount<T>& that);
```

Parameters

Name	Type	Description
that	SharedCount<T>&	A shared count to exchange contents with.

1.1.38.2 Nonmember Functions

Function	Description
operator==<T>(const SharedCount<T>&, const SharedCount<T>&)	Compares to shared count objects for equality.
operator<<T>(const SharedCount<T>&, const SharedCount<T>&)	Compares two shared count objects for less than relationship.

1.1.38.2.1 operator==<T>(const SharedCount<T>&, const SharedCount<T>&) Function

Compares to shared count objects for equality.

Syntax

```
public nothrow bool operator==<T>(const SharedCount<T>& left, const SharedCount<T>&
right);
```

Parameters

Name	Type	Description
left	const SharedCount<T>&	The first shared count.
right	const SharedCount<T>&	The second shared count.

Returns

bool

Returns true if *left* contains the same counter as *right* or both are empty, false otherwise.

1.1.38.2.2 operator<<T>(const SharedCount<T>&, const SharedCount<T>&) Function

Compares two shared count objects for less than relationship.

Syntax

```
public nothrow bool operator<<T>(const SharedCount<T>& left, const SharedCount<T>&
right);
```

Parameters

Name	Type	Description
left	const SharedCount<T>&	The first shared count.
right	const SharedCount<T>&	The second shared count.

Returns

bool

Returns true if the memory address the counter contained by the *left* count is less than the memory address of the counter contained by the *right* count, false otherwise.

1.1.39 SharedPtr<T> Class

A shared pointer to an object.

Syntax

```
public class SharedPtr<T>;
```

Model of

TotallyOrdered<T>

1.1.39.1 Example

```
using System;
using System.Collections;
using Animals;

//  Writes:
//  Rose says meow
//  Rudolf says woof

namespace Animals
{
    public abstract class Animal
    {
        public Animal(const string& name_): name(name_)
        {
        }
        public virtual ~Animal()
        {
        }
        public const string& Name() const
        {
            return name;
        }
        public abstract void Talk();
        private string name;
    }

    public typedef SharedPtr<Animal> AnimalPtr;

    public class Dog: Animal
    {
        public Dog(const string& name_): base(name_)
        {
        }
        public override void Talk()
        {
            Console.Out() << Name() << " says woof" << endl();
        }
    }
```

```
    public class Cat: Animal
    {
        public Cat(const string& name_): base(name_)
        {
        }
        public override void Talk()
        {
            Console.Out() << Name() << " says meow" << endl();
        }
    }
}

void main()
{
    List<AnimalPtr> animals;
    animals.Add(AnimalPtr(new Cat("Rose")));
    animals.Add(AnimalPtr(new Dog("Rudolf")));
//  ...
    for (AnimalPtr animal : animals)
    {
        animal->Talk();
    }
}
```

1.1.39.2 Member Functions

Member Function	Description
SharedPtr()	Constructor. Constructs a null shared pointer.
SharedPtr(const SharedPtr<T>&)	Copy constructor. Increments the use count.
SharedPtr(const WeakPtr<T>&)	Constructor. Constructs a shared pointer from a weak pointer.
SharedPtr(T*)	Constructor. Constructs a shared pointer to the given object.
SharedPtr(T*, const SharedCount<T>&)	Constructor. Implementation detail to support PtrCast<U, T>(const SharedPtr<T>&) function.
operator=(const SharedPtr<T>&)	Copy assignment. Assigns another shared pointer.

~SharedPtr()	Destructor. Decrements the use count.
operator->() const	Returns the contained pointer to the counted object.
operator*() const	Returns a reference to the counted object.
GetCount() const	Returns the contained SharedCount<T>.
GetPtr() const	Returns the contained pointer to the counted object.
GetUseCount() const	Returns the use count.
IsNull() const	Returns true if the contained pointer is null, false otherwise.
IsUnique() const	Returns true if there is exactly one **SharedPtr<T>** to an object.
Reset()	Resets the shared pointer to null.
Reset(T*)	Resets the shared pointer to contain a pointer to another counted object.
Swap(SharedPtr<T>&)	Exchanges the contents with another shared pointer.

1.1.39.2.1 SharedPtr() Member Function

Constructor. Constructs a null shared pointer.

Syntax

```
public nothrow SharedPtr();
```

1.1.39.2.2 SharedPtr(const SharedPtr<T>&) Member Function

Copy constructor. Increments the use count.

Syntax

```
public nothrow SharedPtr(const SharedPtr<T>& that);
```

Parameters

Name	Type	Description
that	const SharedPtr<T>&	A shared pointer to copy.

1.1.39.2.3 SharedPtr(const WeakPtr<T>&) Member Function

Constructor. Constructs a shared pointer from a weak pointer.

Syntax

```
public nothrow SharedPtr(const WeakPtr<T>& that);
```

Parameters

Name	Type	Description
that	const WeakPtr<T>&	A weak pointer to an object.

1.1.39.2.4 SharedPtr(T*) Member Function

Constructor. Constructs a shared pointer to the given object.

Syntax

```
public explicit nothrow SharedPtr(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to an object.

1.1.39.2.5 SharedPtr(T*, const SharedCount<T>&) Member Function

Constructor. Implementation detail to support PtrCast<U, T>(const SharedPtr<T>&) function.

Syntax

```
public nothrow SharedPtr(T* ptr_, const SharedCount<T>& count_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to counted object.
count_	const SharedCount<T>&	A shared count.

1.1.39.2.6 operator=(const SharedPtr<T>&) Member Function

Copy assignment. Assigns another shared pointer.

Syntax

```
public nothrow void operator=(const SharedPtr<T>& that);
```

Parameters

Name	Type	Description
that	const SharedPtr<T>&	A shared pointer to assign.

1.1.39.2.7 ~SharedPtr() Member Function

Destructor. Decrements the use count.

Syntax

```
public default nothrow ~SharedPtr();
```

Remarks

If the use count has gone to zero, destroyes the counted object.

1.1.39.2.8 operator->() const Member Function

Returns the contained pointer to the counted object.

Syntax

```
public inline nothrow T* operator->() const;
```

Returns

T*

Returns the contained pointer to the counted object.

1.1.39.2.9 operator*() const Member Function

Returns a reference to the counted object.

Syntax

```
public inline nothrow T& operator*() const;
```

Returns

T&

Returns a reference to the counted object.

1.1.39.2.10 GetCount() const Member Function

Returns the contained SharedCount<T>.

Syntax

```
public inline nothrow const SharedCount<T>& GetCount() const;
```

Returns

const SharedCount<T>&

Returns the contained shared count.

1.1.39.2.11 GetPtr() const Member Function

Returns the contained pointer to the counted object.

Syntax

```
public inline nothrow T* GetPtr() const;
```

Returns

T*

Returns the contained pointer to the counted object.

1.1.39.2.12 GetUseCount() const Member Function

Returns the use count.

Syntax

```
public nothrow int GetUseCount() const;
```

Returns

int

Returns the use count.

1.1.39.2.13 IsNull() const Member Function

Returns true if the contained pointer is null, false otherwise.

Syntax

```
public inline nothrow bool IsNull() const;
```

Returns

bool

Returns true if the contained pointer is null, false otherwise.

1.1.39.2.14 IsUnique() const Member Function

Returns true if there is exactly one SharedPtr<T> to an object.

Syntax

```
public nothrow bool IsUnique() const;
```

Returns

bool

Returns true if there is exactly one SharedPtr<T> to an object.

1.1.39.2.15 Reset() Member Function

Resets the shared pointer to null.

Syntax

```
public nothrow void Reset();
```

1.1.39.2.16 Reset(T*) Member Function

Resets the shared pointer to contain a pointer to another counted object.

Syntax

```
public nothrow void Reset(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to another counted object.

1.1.39.2.17 Swap(SharedPtr<T>&) Member Function

Exchanges the contents with another shared pointer.

Syntax

```
public nothrow void Swap(SharedPtr<T>& that);
```

Parameters

Name	Type	Description
that	SharedPtr<T>&	A shared pointer to exchange contents with.

1.1.39.3 Nonmember Functions

Function	Description
operator==<T>(const SharedPtr<T>&, const SharedPtr<T>&)	Compares two shared pointers for equality.

operator<<T>(const SharedPtr<T>&, const SharedPtr<T>&)	Compares two shared pointers for less than relationship.

1.1.39.3.1 operator==<T>(const SharedPtr<T>&, const SharedPtr<T>&) Function

Compares two shared pointers for equality.

Syntax

```
public nothrow bool operator==<T>(const SharedPtr<T>& left, const SharedPtr<T>&
right);
```

Parameters

Name	Type	Description
left	const SharedPtr<T>&	The first shared pointer.
right	const SharedPtr<T>&	The second shared pointer.

Returns

bool

Returns true if *left* points to the same object as *right* or both are null, false otherwise.

1.1.39.3.2 operator<<T>(const SharedPtr<T>&, const SharedPtr<T>&) Function

Compares two shared pointers for less than relationship.

Syntax

```
public nothrow bool operator<<T>(const SharedPtr<T>& left, const SharedPtr<T>&
right);
```

Parameters

Name	Type	Description
left	const SharedPtr<T>&	The first shared pointer.
right	const SharedPtr<T>&	The second shared pointer.

Returns

bool

Returns true if the memory address of the counted object pointed by *left* is less than the memory address of the counted object pointed by *right*, false otherwise.

1.1.40 String Class

A string of ASCII characters.

Syntax

```
public class String;
```

1.1.40.1 Type Definitions

Name	Type	Description
ConstIterator	RandomAccessIter<char, const char&, const char*>	A constant iterator type.
Iterator	RandomAccessIter<char, char&, char*>	An iterator type.

1.1.40.2 Member Functions

Member Function	Description
String()	Constructor. Constructs an empty string.
String(const char*)	Constructor. Constructs a string from a C-style string.
String(string&&)	Move constructor.
String(const string&)	Copy constructor.
String(const char*, int)	Constructor. Constructs a string from given number of characters of a C-style string.
String(char)	Constructor. Constructs a string that has the given character.
String(char, int)	Constructor. Constructs a string that has given number of the specified character.
operator=(const string&)	Copy assignment.
operator=(string&&)	Move assignment.
~String()	Destructor. Releases the memory occupied by the string.

operator==(const string&) const	Compares a string for equality with another string.
operator[](int)	Returns a reference to the character with the specified index.
operator[](int) const	Returns a character with the specified index.
operator<(const string&) const	Compares a string for less than relationship with another string.
Append(const char*)	Appends the specified string to the end of this string.
Append(const string&)	Appends the specified C-style string to the end of this string.
Append(const char*, int)	Appends the given number of characters from a C-style string to the end of this string.
Append(char)	Appends the given chacter to the end of this string.
Begin()	Returns an iterator to the beginning of the string.
Begin() const	Returns a constant iterator to the beginning of the string.
CBegin() const	Returns a constant iterator to the beginning of the string.
CEnd() const	Returns a constant iterator one past the end of the string.
Capacity() const	Returns the number of characters that the string can hold without allocating more memory for it.

Chars() const	Returns the string as a C-syle string.
Clear()	Makes the string empty.
End()	Returns an iterator to one past the end of the string.
End() const	Returns a constant iterator to one past the end of the string.
EndsWith(const string&) const	Returns true, if the string ends with the specified substring, false otherwise.
Find(const string&) const	Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.
Find(const string&, int) const	Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.
Find(char) const	Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string.
Find(char, int) const	Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.
IsEmpty() const	Returns true if the string is empty, false otherwise.
Length() const	Returns the length of the string.
RFind(const string&) const	Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.

RFind(const string&, int) const	Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.
RFind(char) const	Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string.
RFind(char, int) const	Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.
Replace(char, char)	Replaces every occurrence of the given character with another character.
Reserve(int)	Reserves room for a string with the given number of characters.
Split(char)	Returns a list of substrings separated by the given character.
StartsWith(const string&) const	Returns true if the string starts with the given substring, false otherwise.
Substring(int) const	Returns a substring starting from the given index.
Substring(int, int) const	Returns a substring starting from the given index whose length is at most given number of characters.
Swap(string&)	Exchanges the contents a the string with another string.

1.1.40.2.1 String() Member Function

Constructor. Constructs an empty string.

Syntax

```
public nothrow String();
```

1.1.40.2.2 String(const char*) Member Function

Constructor. Constructs a string from a C-style string.

Syntax

```
public nothrow String(const char* chars_);
```

Parameters

Name	Type	Description
chars_	const char*	A C-style string.

1.1.40.2.3 String(string&&) Member Function

Move constructor.

Syntax

```
public nothrow String(string&& that);
```

Parameters

Name	Type	Description
that	string&&	A string to move from.

1.1.40.2.4 String(const string&) Member Function

Copy constructor.

Syntax

```
public nothrow String(const string& that);
```

Parameters

Name	Type	Description
that	const string&	A string to copy.

1.1.40.2.5 String(const char*, int) Member Function

Constructor. Constructs a string from given number of characters of a C-style string.

Syntax

```
public nothrow String(const char* chars_, int length_);
```

Parameters

Name	Type	Description
chars_	const char*	A C-style string.
length_	int	Maximum number of characters to copy.

Example

```
using System;

//  Writes:
//  A bird

void main()
{
    string proverb("A bird in the hand is worth two in the bush.", 6);
    Console.Out() << proverb << endl();
}
```

1.1.40.2.6 String(char) Member Function

Constructor. Constructs a string that has the given character.

Syntax

`public nothrow String(char c);`

Parameters

Name	Type	Description
c	char	A character.

1.1.40.2.7 String(char, int) Member Function

Constructor. Constructs a string that has given number of the specified character.

Syntax

`public nothrow String(char c, int n);`

Parameters

Name	Type	Description
c	char	A character.
n	int	Number of characters.

Example

```
using System;

//  Writes:
//  aaaaaa

void main()
{
    string s('a', 6);
    Console.Out() << s << endl();
}
```

1.1.40.2.8 operator=(const string&) Member Function

Copy assignment.

Syntax

`public nothrow void operator=(const string& that);`

Parameters

Name	Type	Description
that	const string&	A string to assign.

Example

```
using System;

//  Writes:
//  Parempi pyy pivossa kuin kymmenen oksalla.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    proverb = "Parempi pyy pivossa kuin kymmenen oksalla.";
    Console.Out() << proverb << endl();
}
```

1.1.40.2.9 operator=(string&&) Member Function

Move assignment.

Syntax

`public nothrow void operator=(string&& that);`

Parameters

Name	Type	Description
that	string&&	A string to move from.

1.1.40.2.10 ∼String() Member Function

Destructor. Releases the memory occupied by the string.

Syntax

```
public nothrow ~String();
```

1.1.40.2.11 operator==(const string&) const Member Function

Compares a string for equality with another string.

Syntax

```
public nothrow bool operator==(const string& that) const;
```

Parameters

Name	Type	Description
that	const string&	A string to compare with.

Returns

bool

Returns true if this string has the same number of characters than the given string, and the characters are pairwise equal, false otherwise.

Remarks

The comparison is case sensitive.

1.1.40.2.12 operator[](int) Member Function

Returns a reference to the character with the specified index.

Syntax

```
public nothrow char& operator[](int index);
```

Parameters

Name	Type	Description
index	int	An index.

Returns

char&

Returns a reference to the character with the specified index.

Example

```
using System;

//  Writes:
//  A bird in the band is worth two in the bush.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    proverb[14] = 'b';
    Console.Out() << proverb << endl();
}
```

1.1.40.2.13 operator[](int) const Member Function

Returns a character with the specified index.

Syntax

`public nothrow char operator[](int index) const;`

Parameters

Name	Type	Description
index	int	An index.

Returns

char

Returns a character with the specified index.

1.1.40.2.14 operator<(const string&) const Member Function

Compares a string for less than relationship with another string.

Syntax

`public nothrow bool operator<(const string& that) const;`

Parameters

Name	Type	Description
that	const string&	A string to compare with.

Returns

bool

Returns true if this string comes lexicographically before the specified string, false otherwise.

Remarks

The comparison is case sensitive and done with the ASCII code values of the characters.

1.1.40.2.15 Append(const char*) Member Function

Appends the specified string to the end of this string.

Syntax

`public nothrow void Append(const char* that);`

Parameters

Name	Type	Description
that	const char*	A string to append.

Example

```
using System;

//  Writes:
//  A bird in the hand is worth two in the bush.

void main()
{
    string proverb("A bird in the hand ");
    string e("is worth two in the bush.");
    proverb.Append(e);
    Console.Out() << proverb << endl();
}
```

1.1.40.2.16 Append(const string&) Member Function

Appends the specified C-style string to the end of this string.

Syntax

`public nothrow void Append(const string& that);`

Parameters

Name	Type	Description
that	const string&	A C-style string to append.

1.1.40.2.17 Append(const char*, int) Member Function

Appends the given number of characters from a C-style string to the end of this string.

Syntax

```
public nothrow void Append(const char* that, int count);
```

Parameters

Name	Type	Description
that	const char*	A C-style string.
count	int	The maximum number of characters to append.

1.1.40.2.18 Append(char) Member Function

Appends the given chacter to the end of this string.

Syntax

```
public nothrow void Append(char c);
```

Parameters

Name	Type	Description
c	char	A character to append.

1.1.40.2.19 Begin() Member Function

Returns an iterator to the beginning of the string.

Syntax

```
public nothrow Iterator Begin();
```

Returns

Iterator

Returns an iterator to the beginning of the string.

1.1.40.2.20 Begin() const Member Function

Returns a constant iterator to the beginning of the string.

Syntax

```
public nothrow ConstIterator Begin() const;
```

Returns

ConstIterator

Returns a constant iterator to the beginning of the string.

1.1.40.2.21 CBegin() const Member Function

Returns a constant iterator to the beginning of the string.

Syntax

```
public nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant iterator to the beginning of the string.

1.1.40.2.22 CEnd() const Member Function

Returns a constant iterator one past the end of the string.

Syntax

```
public nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant iterator one past the end of the string.

1.1.40.2.23 Capacity() const Member Function

Returns the number of characters that the string can hold without allocating more memory for it.

Syntax

```
public inline nothrow int Capacity() const;
```

Returns

int

Returns the number of characters that the string can hold without allocating more memory for it.

1.1.40.2.24 Chars() const Member Function

Returns the string as a C-syle string.

Syntax

```
public nothrow const char* Chars() const;
```

Returns

const char*

Returns the string as a C-syle string.

1.1.40.2.25 Clear() Member Function

Makes the string empty.

Syntax

```
public nothrow void Clear();
```

1.1.40.2.26 End() Member Function

Returns an iterator to one past the end of the string.

Syntax

```
public nothrow Iterator End();
```

Returns

Iterator

Returns an iterator to one past the end of the string.

1.1.40.2.27 End() const Member Function

Returns a constant iterator to one past the end of the string.

Syntax

```
public nothrow ConstIterator End() const;
```

Returns

ConstIterator

Returns a constant iterator to one past the end of the string.

1.1.40.2.28 EndsWith(const string&) const Member Function

Returns true, if the string ends with the specified substring, false otherwise.

Syntax

```
public nothrow bool EndsWith(const string& suffix) const;
```

Parameters

Name	Type	Description
suffix	const string&	A suffix to test.

Returns

bool

Returns true, if the string ends with the specified substring, false otherwise.

Remarks

The comparison is case sensitive.

Example

```
using System;

//  Writes:
//  true

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.EndsWith("bush.") << endl();
}
```

1.1.40.2.29 Find(const string&) const Member Function

Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.

Syntax

`public nothrow int Find(const string& s) const;`

Parameters

Name	Type	Description
s	const string&	A substring to search.

Returns

int

Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.

Example

```
using System;

//  Writes:
//  7
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.Find("in") << endl();
    Console.Out() << proverb.Find("foo") << endl();
}
```

1.1.40.2.30 Find(const string&, int) const Member Function

Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.

Syntax

`public nothrow int Find(const string& s, int start) const;`

Parameters

Name	Type	Description
s	const string&	A substring to search.
start	int	A search start index.

Returns

int

Returns the starting index of the first occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.

Example

```
using System;

//  Writes:
//  32
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
```

```
    Console.Out() << proverb.Find("in", 8) << endl();
    Console.Out() << proverb.Find("foo", 8) << endl();
}
```

1.1.40.2.31 Find(char) const Member Function

Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string.

Syntax

`public nothrow int Find(char x) const;`

Parameters

Name	Type	Description
x	char	A character to search.

Returns

int

Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string.

Example

```
using System;

//  Writes:
//  10
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.Find('t') << endl();
    Console.Out() << proverb.Find('x') << endl();
}
```

1.1.40.2.32 Find(char, int) const Member Function

Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.

Syntax

`public nothrow int Find(char x, int start) const;`

Parameters

Name	Type	Description
x	char	A character to search.
start	int	A search start index.

Returns

int

Returns the index of the first occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.

Example

```
using System;

//  Writes:
//  25
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.Find('t', 11) << endl();
    Console.Out() << proverb.Find('x', 11) << endl();
}
```

1.1.40.2.33 IsEmpty() const Member Function

Returns true if the string is empty, false otherwise.

Syntax

`public inline nothrow bool IsEmpty() const;`

Returns

bool

Returns true if the string is empty, false otherwise.

1.1.40.2.34 Length() const Member Function

Returns the length of the string.

Syntax

`public inline nothrow int Length() const;`

Returns

int

Returns the length of the string.

1.1.40.2.35 RFind(const string&) const Member Function

Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.

Syntax

```
public nothrow int RFind(const string& s) const;
```

Parameters

Name	Type	Description
s	const string&	A substring to search.

Returns

int

Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string.

Example

```
using System;

//  Writes:
//  32
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.RFind("in") << endl();
    Console.Out() << proverb.RFind("foo") << endl();
}
```

1.1.40.2.36 RFind(const string&, int) const Member Function

Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.

Syntax

```
public nothrow int RFind(const string& s, int start) const;
```

Parameters

Name	Type	Description
s	const string&	A substring to search.
start	int	A search start index.

Returns

int

Returns the starting index of the last occurrence of the specified substring within this string, or -1 is the specified string does not occur within this string. The search starts with the specified index.

Example

```
using System;

//  Writes:
//  7
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.RFind("in", 31) << endl();
    Console.Out() << proverb.RFind("foo", 31) << endl();
}
```

1.1.40.2.37 RFind(char) const Member Function

Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string.

Syntax

`public nothrow int RFind(char x) const;`

Parameters

Name	Type	Description
x	char	A character to search.

Returns

int

Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string.

Example

```
using System;

//  Writes:
//  35
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.RFind('t') << endl();
    Console.Out() << proverb.RFind('x') << endl();
}
```

1.1.40.2.38 RFind(char, int) const Member Function

Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.

Syntax

`public nothrow int RFind(char x, int start) const;`

Parameters

Name	Type	Description
x	char	A character to search.
start	int	A search start index.

Returns

int

Returns the index of the last occurrence of the given character within this string, or -1 if the specified character does not occur in this string. The search starts with the specified index.

Example

```
using System;

//  Writes:
//  28
//  -1

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.RFind('t', 34) << endl();
    Console.Out() << proverb.RFind('x', 34) << endl();
```

```
}
```

1.1.40.2.39 Replace(char, char) Member Function

Replaces every occurrence of the given character with another character.

Syntax

`public nothrow void Replace(char oldChar, char newChar);`

Parameters

Name	Type	Description
oldChar	char	A character to replace.
newChar	char	The replacement character.

Example

```
using System;

//  Writes:
//  A bord on the hand os worth two on the bush.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    proverb.Replace('i', 'o');
    Console.Out() << proverb << endl();
}
```

1.1.40.2.40 Reserve(int) Member Function

Reserves room for a string with the given number of characters.

Syntax

`public nothrow void Reserve(int minLen);`

Parameters

Name	Type	Description
minLen	int	The minimum number of characters that the string can hold without a memory allocation.

1.1.40.2.41 Split(char) Member Function

Returns a list of substrings separated by the given character.

Syntax

`public List<String> Split(char c);`

Parameters

Name	Type	Description
c	char	A separator character.

Returns

List<String>

Returns a list of substrings separated by the given character.

Example

```
using System;
using System.Collections;

//  Writes:
//  A
//  bird
//  in
//  the
//  hand
//  is
//  worth
//  two
//  in
//  the
//  bush.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    List<string> words = proverb.Split(' ');
    for (const string& word : words)
    {
        Console.Out() << word << endl();
    }
}
```

1.1.40.2.42 StartsWith(const string&) const Member Function

Returns true if the string starts with the given substring, false otherwise.

Syntax

`public nothrow bool StartsWith(const string& prefix) const;`

Parameters

Name	Type	Description
prefix	const string&	A prefix to test.

Returns

bool

Returns true if the string starts with the given substring, false otherwise.

1.1.40.2.43 Substring(int) const Member Function

Returns a substring starting from the given index.

Syntax

```
public nothrow string Substring(int start) const;
```

Parameters

Name	Type	Description
start	int	A starting index of the substring.

Returns

string

Returns a substring starting from the given index.

Example

```
using System;

//  Writes:
//  worth two in the bush.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.Substring(22) << endl();
}
```

1.1.40.2.44 Substring(int, int) const Member Function

Returns a substring starting from the given index whose length is at most given number of characters.

Syntax

```
public nothrow string Substring(int start, int length) const;
```

Parameters

Name	Type	Description
start	int	A starting index of the substring.
length	int	The maximum number of characters in the substring.

Returns

string

Returns a substring starting from the given index whose length is at most given number of characters.

Example

```
using System;

//  Writes:
//  worth
//  bush.

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    Console.Out() << proverb.Substring(22,5) << endl();
    Console.Out() << proverb.Substring(39,10) << endl();
}
```

1.1.40.2.45 Swap(string&) Member Function

Exchanges the contents a the string with another string.

Syntax

`public nothrow void Swap(string& that);`

Parameters

Name	Type	Description
that	string&	A string to exchange contents with.

Example

```
using System;

//  Writes:
//  A bird in the hand is worth two in the bush.
//  Parempi pyy pivossa kuin kymmenen oksalla.
//  Parempi pyy pivossa kuin kymmenen oksalla.
//  A bird in the hand is worth two in the bush.
```

```
void main()
{
    string proverb1("A bird in the hand is worth two in the bush.");
    string proverb2("Parempi pyy pivossa kuin kymmenen oksalla.");
    Console.Out() << proverb1 << endl();
    Console.Out() << proverb2 << endl();
    proverb1.Swap(proverb2);
    Console.Out() << proverb1 << endl();
    Console.Out() << proverb2 << endl();
}
```

1.1.40.3 Nonmember Functions

Function	Description
operator+(const string&, const string&)	Concatenates two strings.
operator+(const string&, const char*)	Concatenates a string and a C-style string.
operator+(const char*, const string&)	Concatenates a C-style string and a string.

1.1.40.3.1 operator+(const string&, const string&) Function

Concatenates two strings.

Syntax

`public nothrow string operator+(const string& first, const string& second);`

Parameters

Name	Type	Description
first	const string&	The first string.
second	const string&	The second string.

Returns

string

Returns a string that result when *first* and *second* is concatenated.

Example

```
using System;

//  Writes:
//  A bird in the hand is worth two in the bush.
```

```
void main()
{
    string start("A bird in the hand is worth ");
    string end("two in the bush.");
    string proverb = start + end;
    Console.Out() << proverb << endl();
}
```

1.1.40.3.2 operator+(const string&, const char*) Function

Concatenates a string and a C-style string.

Syntax

`public nothrow string operator+(const string& first, const char* second);`

Parameters

Name	Type	Description
first	const string&	A string.
second	const char*	A C-style string.

Returns

string

Returns a string that result when *first* and *second* is concatenated.

Example

```
using System;

//  Writes:
//  A bird in the hand is worth two in the bush.

void main()
{
    const char* start = "A bird in the hand is worth ";
    string end("two in the bush.");
    string proverb = start + end;
    Console.Out() << proverb << endl();
}
```

1.1.40.3.3 operator+(const char*, const string&) Function

Concatenates a C-style string and a string.

Syntax

`public nothrow string operator+(const char* first, const string& second);`

Parameters

Name	Type	Description
first	const char*	A C-style string.
second	const string&	A string.

Returns

string

Returns a string that result when *first* and *second* is concatenated.

Example

```
using System;

// Writes:
// A bird in the hand is worth two in the bush.

void main()
{
    string start("A bird in the hand is worth ");
    const char* end = "two in the bush.";
    string proverb = start + end;
    Console.Out() << proverb << endl();
}
```

1.1.41 TimeError Class

An exception thrown when a time function fails.

Syntax

```
public class TimeError;
```

Base Class

Exception

1.1.41.1 Member Functions

Member Function	Description
TimeError(TimeError&&)	Move constructor.
TimeError(const TimeError&)	Copy constructor.
TimeError(const string&, const string&)	Constructor. Initializes the time error with specified operation description and failure reason.
operator=(const TimeError&)	Copy assignment.
operator=(TimeError&&)	Move assignment.
~TimeError()	Destructor.

1.1.41.1.1 TimeError(TimeError&&) Member Function

Move constructor.

Syntax

```
public nothrow TimeError(TimeError&& that);
```

Parameters

Name	Type	Description
that	TimeError&&	A time error to move from.

1.1.41.1.2 TimeError(const TimeError&) Member Function

Copy constructor.

Syntax

```
public nothrow TimeError(const TimeError& that);
```

Parameters

Name	Type	Description
that	const TimeError&	A time error to copy from.

1.1.41.1.3 TimeError(const string&, const string&) Member Function

Constructor. Initializes the time error with specified operation description and failure reason.

Syntax

```
public TimeError(const string& operation, const string& reason);
```

Parameters

Name	Type	Description
operation	const string&	Description of operation.
reason	const string&	Failure reason.

1.1.41.1.4 operator=(const TimeError&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const TimeError& that);
```

Parameters

Name	Type	Description
that	const TimeError&	A time error to assign.

1.1.41.1.5 operator=(TimeError&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(TimeError&& that);
```

Parameters

Name	Type	Description
that	TimeError&&	A time error to move from.

1.1.41.1.6 ~TimeError() Member Function

Destructor.

Syntax

```
public override nothrow ~TimeError();
```

1.1.42 TimePoint Class

Represents a point in time as specified nanoseconds elapsed since epoch.

Syntax

```
public class TimePoint;
```

1.1.42.1 Remarks

Epoch is midnight 1.1.1970.

1.1.42.2 Member Functions

Member Function	Description
TimePoint()	Constructor. Initializes the time point to zero nanoseconds elapsed since epoch.
TimePoint(const TimePoint&)	Copy constructor.
TimePoint(TimePoint&&)	Move constructor.
TimePoint(uhuge)	Constructor. Initializes the time point to specified number of nanoseconds elapsed since epoch.
operator=(const TimePoint&)	Copy assignment.
operator=(TimePoint&&)	Move assignment.
Rep() const	Returns the number of nanoseconds elapsed since epoch.

1.1.42.2.1 TimePoint() Member Function

Constructor. Initializes the time point to zero nanoseconds elapsed since epoch.

Syntax

```
public nothrow TimePoint();
```

1.1.42.2.2 TimePoint(const TimePoint&) Member Function

Copy constructor.

Syntax

```
public nothrow TimePoint(const TimePoint& that);
```

Parameters

Name	Type	Description
that	const TimePoint&	A time point to copy.

1.1.42.2.3 TimePoint(TimePoint&&) Member Function

Move constructor.

Syntax

```
public nothrow TimePoint(TimePoint&& that);
```

Parameters

Name	Type	Description
that	TimePoint&&	A time point to move from.

1.1.42.2.4 TimePoint(uhuge) Member Function

Constructor. Initializes the time point to specified number of nanoseconds elapsed since epoch.

Syntax

```
public explicit nothrow TimePoint(uhuge nanosecs_);
```

Parameters

Name	Type	Description
nanosecs_	uhuge	Nanoseconds.

1.1.42.2.5 operator=(const TimePoint&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const TimePoint& that);
```

Parameters

Name	Type	Description
that	const TimePoint&	A time point to assign from.

1.1.42.2.6 operator=(TimePoint&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(TimePoint&& that);
```

Parameters

Name	Type	Description
that	TimePoint&&	A time point to move from.

1.1.42.2.7 Rep() const Member Function

Returns the number of nanoseconds elapsed since epoch.

Syntax

```
public inline nothrow uhuge Rep() const;
```

Returns

uhuge

Returns the number of nanoseconds elapsed since epoch.

1.1.42.3 Nonmember Functions

Function	Description
operator==(TimePoint, TimePoint)	Compares two time points for equality.
operator<(TimePoint, TimePoint)	Compares two time points for less than relationship.
operator-(TimePoint, TimePoint)	Subtracts a time point from a time point and returns a duration.
operator-(TimePoint, Duration)	Subtracts a duration from a time point and returns a time point.
operator+(TimePoint, Duration)	Computes the sum of a time point and a duration and returns a time point.

1.1.42.3.1 operator==(TimePoint, TimePoint) Function

Compares two time points for equality.

Syntax

```
public inline nothrow bool operator==(TimePoint left, TimePoint right);
```

Parameters

Name	Type	Description
left	TimePoint	The first time point.
right	TimePoint	The second time point.

Returns

bool

Returns true, if the first time point is equal to the second time point, false otherwise.

1.1.42.3.2 operator<(TimePoint, TimePoint) Function

Compares two time points for less than relationship.

Syntax

```
public inline nothrow bool operator<(TimePoint left, TimePoint right);
```

Parameters

Name	Type	Description
left	TimePoint	The first time point.
right	TimePoint	The second time point.

Returns

bool

Returns true, if the first time point is less than the second time point, false otherwise.

1.1.42.3.3 operator-(TimePoint, TimePoint) Function

Subtracts a time point from a time point and returns a duration.

Syntax

```
public inline nothrow Duration operator-(TimePoint left, TimePoint right);
```

Parameters

Name	Type	Description
left	TimePoint	The first time point.
right	TimePoint	The second time point.

Returns

Duration

Returns $Duration(left - right)$.

1.1.42.3.4 operator-(TimePoint, Duration) Function

Subtracts a duration from a time point and returns a time point.

Syntax

```
public inline nothrow TimePoint operator-(TimePoint tp, Duration d);
```

Parameters

Name	Type	Description
tp	TimePoint	A time point.
d	Duration	A duration.

Returns

TimePoint

Returns $TimePoint(left - right$.

1.1.42.3.5 operator+(TimePoint, Duration) Function

Computes the sum of a time point and a duration and returns a time point.

Syntax

```
public inline nothrow TimePoint operator+(TimePoint tp, Duration d);
```

Parameters

Name	Type	Description
tp	TimePoint	A time point.
d	Duration	A duration.

Returns

TimePoint

Returns tp+d.

1.1.43 Tracer Class

A utility class for tracing entry and exit of some operation.

Syntax

```
public class Tracer;
```

1.1.43.1 Member Functions

Member Function	Description
Tracer(const string&)	Constructor. Writes the specified string to standard error stream.
~Tracer()	Destructor. Writes ~ and a string specified in constructor to standard error stream.

1.1.43.1.1 Tracer(const string&) Member Function

Constructor. Writes the specified string to standard error stream.

Syntax

```
public nothrow Tracer(const string& s_);
```

Parameters

Name	Type	Description
s_	const string&	String to write.

1.1.43.1.2 ~Tracer() Member Function

Destructor. Writes ~ and a string specified in constructor to standard error stream.

Syntax

```
public nothrow ~Tracer();
```

1.1.44 UnaryFun<Argument, Result> Class

A base class for unary function objects.

Syntax

```
public class UnaryFun<Argument, Result>;
```

Constraint

where Argument is Semiregular;

1.1.44.1 Remarks

A derived unary function inherits the type definitions of this base class and provides an implementation for the *operator()(ArgumentType)* function.

1.1.44.2 Type Definitions

Name	Type	Description
ArgumentType	Argument	The type of the argument of the unary function.
ResultType	Result	The type of the result of the unary function.

1.1.45 UnaryPred<Argument> Class

A base class for unary predicates.

Syntax

```
public class UnaryPred<Argument>;
```

Constraint

where Argument is Semiregular;

Model of

UnaryPredicate<T>

Base Class

UnaryFun<Argument, bool>

1.1.45.1 Remarks

A unary predicate is an unary function whose application operator returns a truth value.

1.1.46 UniquePtr<T> Class

A unique pointer to an object.

Syntax

`public class UniquePtr<T>;`

Model of

DefaultConstructible<T>

Movable<T>

1.1.46.1 Remarks

The unique pointer destroys the object it owns in its destructor. The copy constructor and copy assignment operator are suppressed, but unique pointer has move constructor and move assignment operator, so it can be moved to containers.

1.1.46.2 Example

```
using System;
using System.Collections;

//  Writes:
//  foo
//  bar

void main()
{
    List<UniquePtr<string>> list;
    UniquePtr<string> foo(new string("foo"));
    list.Add(Rvalue(foo));
    list.Add(UniquePtr<string>(new string("bar")));
    for (const UniquePtr<string>& s : list)
    {
        Console.Out() << *s << endl();
    }
}
```

1.1.46.3 Member Functions

Member Function	Description
UniquePtr()	Constructor. Constructs a null unique pointer.
UniquePtr(UniquePtr<T>&&)	Move constructor.

UniquePtr(T*)	Constructor. Constructs a unique pointer to the given object.
operator=(T*)	Assigns a new object to the unique pointer.
operator=(UniquePtr<T>&&)	Move assignment.
~UniquePtr()	Destructor. Destroys the owned object.
operator->() const	Returns the contained pointer to the pointed object.
operator*() const	Returns a reference to the pointed object.
GetPtr() const	Returns the contained pointer to the owned object.
IsNull() const	Returns true if the contained pointer is null, false otherwise.
Release()	Releases the ownership of the owned object and sets the unique pointer to null.
Reset()	Resets the unique ptr to null.
Reset(T*)	Resets the contained pointer to point to a new object.
Swap(UniquePtr<T>&)	Exchanges the contents of the unique pointer with another unique pointer.

1.1.46.3.1 UniquePtr() Member Function

Constructor. Constructs a null unique pointer.

Syntax

```
public nothrow UniquePtr();
```

1.1.46.3.2 UniquePtr(UniquePtr<T>&&) Member Function

Move constructor.

Syntax

```
public nothrow UniquePtr(UniquePtr<T>&& that);
```

Parameters

Name	Type	Description
that	UniquePtr<T>&&	A unique pointer to move from.

1.1.46.3.3 UniquePtr(T*) Member Function

Constructor. Constructs a unique pointer to the given object.

Syntax

```
public nothrow UniquePtr(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to an object.

1.1.46.3.4 operator=(T*) Member Function

Assigns a new object to the unique pointer.

Syntax

```
public nothrow void operator=(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to an object.

Remarks

If the unique pointer is not null before the assignment, destroys the old object before the assignment. Then acquires the ownership of the new object.

1.1.46.3.5 operator=(UniquePtr<T>&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(UniquePtr<T>&& that);
```

Parameters

Name	Type	Description
that	UniquePtr<T>&&	A unique pointer to move from.

1.1.46.3.6 ~UniquePtr() Member Function

Destructor. Destroys the owned object.

Syntax

```
public nothrow ~UniquePtr();
```

1.1.46.3.7 operator->() const Member Function

Returns the contained pointer to the pointed object.

Syntax

```
public inline nothrow T* operator->() const;
```

Returns

T*

Returns the contained pointer to the pointed object.

1.1.46.3.8 operator*() const Member Function

Returns a reference to the pointed object.

Syntax

```
public inline nothrow T& operator*() const;
```

Returns

T&

Returns a reference to the pointed object.

1.1.46.3.9 GetPtr() const Member Function

Returns the contained pointer to the owned object.

Syntax

```
public inline nothrow T* GetPtr() const;
```

Returns

T*

Returns the contained pointer to the owned object.

1.1.46.3.10 IsNull() const Member Function

Returns true if the contained pointer is null, false otherwise.

Syntax

```
public inline nothrow bool IsNull() const;
```

Returns

bool

Returns true if the contained pointer is null, false otherwise.

1.1.46.3.11 Release() Member Function

Releases the ownership of the owned object and sets the unique pointer to null.

Syntax

```
public inline nothrow T* Release();
```

Returns

T*

Returns the contained pointer.

1.1.46.3.12 Reset() Member Function

Resets the unique ptr to null.

Syntax

```
public nothrow void Reset();
```

Remarks

Destroys the owned object.

1.1.46.3.13 Reset(T*) Member Function

Resets the contained pointer to point to a new object.

Syntax

```
public nothrow void Reset(T* ptr_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to an object.

Remarks

If the unique pointer owns an object before the operation, destroys the owned object before the operation. Then acquires the ownership of the new object.

1.1.46.3.14 Swap(UniquePtr<T>&) Member Function

Exchanges the contents of the unique pointer with another unique pointer.

Syntax

```
public nothrow void Swap(UniquePtr<T>& that);
```

Parameters

Name	Type	Description
that	UniquePtr<T>&	A unique pointer to exchange contents with.

1.1.47 WeakCount<T> Class

A handle to a Counter<T> that maintains the weak count portion of the counter.

Syntax

```
public class WeakCount<T>;
```

1.1.47.1 Member Functions

Member Function	Description
WeakCount()	Constructor. Initializes an empty weak count.
WeakCount(const WeakCount<T>&)	Copy constructor. Increments weak count.
WeakCount(const SharedCount<T>&)	Constructor. Constructs a weak count from a shared count.
operator=(const SharedCount<T>&)	Assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.
operator=(const WeakCount<T>&)	Copy assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.
~WeakCount()	Destructor. Decrements the weak count.
GetCounter() const	Returns the contained pointer to a counter.
GetUseCount() const	Returns the use count.
Swap(WeakCount<T>&)	Exchanges the contents of the weak count with another weak count.

1.1.47.1.1 WeakCount() Member Function

Constructor. Initializes an empty weak count.

Syntax

```
public nothrow WeakCount();
```

1.1.47.1.2 WeakCount(const WeakCount<T>&) Member Function

Copy constructor. Increments weak count.

Syntax

```
public nothrow WeakCount(const WeakCount<T>& that);
```

Parameters

Name	Type	Description
that	const WeakCount<T>&	A weak count to copy.

1.1.47.1.3 WeakCount(const SharedCount<T>&) Member Function

Constructor. Constructs a weak count from a shared count.

Syntax

```
public nothrow WeakCount(const SharedCount<T>& that);
```

Parameters

Name	Type	Description
that	const SharedCount<T>&	A shared count.

1.1.47.1.4 operator=(const SharedCount<T>&) Member Function

Assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.

Syntax

```
public nothrow void operator=(const SharedCount<T>& that);
```

Parameters

Name	Type	Description
that	const SharedCount<T>&	A shared count to assign.

1.1.47.1.5 operator=(const WeakCount<T>&) Member Function

Copy assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.

Syntax

```
public nothrow void operator=(const WeakCount<T>& that);
```

Parameters

Name	Type	Description
that	const WeakCount<T>&	A weak count to assign.

1.1.47.1.6 ∼WeakCount() Member Function

Destructor. Decrements the weak count.

Syntax

```
public nothrow ~WeakCount();
```

1.1.47.1.7 GetCounter() const Member Function

Returns the contained pointer to a counter.

Syntax

```
public nothrow Counter<T>* GetCounter() const;
```

Returns

Counter<T>*

Returns the contained pointer to a counter.

1.1.47.1.8 GetUseCount() const Member Function

Returns the use count.

Syntax

```
public nothrow int GetUseCount() const;
```

Returns

int

Returns the use count.

1.1.47.1.9 Swap(WeakCount<T>&) Member Function

Exchanges the contents of the weak count with another weak count.

Syntax

```
public nothrow void Swap(WeakCount<T>& that);
```

Parameters

Name	Type	Description
that	WeakCount<T>&	A weak count to exchange contents with.

1.1.47.2 Nonmember Functions

Function	Description
operator==<T>(const WeakCount<T>&, const WeakCount<T>&)	Compares two weak counts for equality.
operator<<T>(const WeakCount<T>&, const WeakCount<T>&)	Compares two weak counts for less than relationship.

1.1.47.2.1 operator==<T>(const WeakCount<T>&, const WeakCount<T>&) Function

Compares two weak counts for equality.

Syntax

```
public nothrow bool operator==<T>(const WeakCount<T>& left, const WeakCount<T>&
right);
```

Parameters

Name	Type	Description
left	const WeakCount<T>&	The first weak count.
right	const WeakCount<T>&	The second weak count.

Returns

bool

Returns true if *left* contains the same counter as *right* or both are empty, false otherwise.

1.1.47.2.2 operator<<T>(const WeakCount<T>&, const WeakCount<T>&) Function

Compares two weak counts for less than relationship.

Syntax

```
public nothrow bool operator<<T>(const WeakCount<T>& left, const WeakCount<T>&
right);
```

Parameters

Name	Type	Description
left	const WeakCount<T>&	The first weak count.
right	const WeakCount<T>&	The second weak count.

Returns

bool

Returns true if the memory address of the counter contained by the *left* count is less than the memory addess of the counter contained by the *right* count, false otherwise.

1.1.48 WeakPtr<T> Class

Used to break cycles in shared ownership.

Syntax

```
public class WeakPtr<T>;
```

1.1.48.1 Remarks

If objects contain shared pointers to each other thus forming a cycle, the use count will never go to zero, and the destructors of the objects will not be called. Replacing one of the shared pointers with a weak pointer breaks the cycle and the objects will be released.

1.1.48.2 Example

```
using System;
using System.Collections;

//  Writes:
//  a.next == b
//  b.next == c
//  c.next == a
//  a destroyed
//  b destroyed
//  c destroyed

public typedef SharedPtr<Base> BasePtr;
public typedef WeakPtr<Base> WeakBasePtr;

public abstract class Base
{
    public Base(const string& name_): name(name_)
    {
    }
    public virtual ~Base()
    {
        Console.WriteLine(name + " destroyed");
    }
    public const string& Name() const
    {
        return name;
    }
    public abstract BasePtr GetNext() const;
    public abstract void SetNext(BasePtr next_);
    public void PrintNext()
    {
        BasePtr next = GetNext();
        if (!next.IsNull())
        {
            Console.WriteLine(name + ".next == " + next->Name());
```

```
        }
    }
    private string name;
}

public class A: Base
{
    public A(const string& name_): base(name_)
    {
    }
    public override BasePtr GetNext() const
    {
        return next;
    }
    public override void SetNext(BasePtr next_)
    {
        next = next_;
    }
    private BasePtr next;
}

public class B: Base
{
    public B(const string& name_): base(name_)
    {
    }
    public override BasePtr GetNext() const
    {
        return next;
    }
    public override void SetNext(BasePtr next_)
    {
        next = next_;
    }
    private BasePtr next;
}

public class C: Base
{
    public C(const string& name_): base(name_)
    {
    }
    public override BasePtr GetNext() const
    {
        return next.Lock();
    }
    public override void SetNext(BasePtr next_)
    {
        next = next_;
    }
    private WeakBasePtr next;
}
```

```
void main()
{
    List<BasePtr> objects;
    BasePtr a(new A("a"));
    BasePtr b(new B("b"));
    BasePtr c(new C("c"));
    a->SetNext(b);
    b->SetNext(c);
    c->SetNext(a);
    objects.Add(a);
    objects.Add(b);
    objects.Add(c);
    for (BasePtr o : objects)
    {
        o->PrintNext();
    }
}
```

1.1.48.3 Member Functions

Member Function	Description
WeakPtr()	Constructor. Constructs null weak pointer.
WeakPtr(const WeakPtr<T>&)	Copy constructor. Increments weak count.
WeakPtr(const SharedPtr<T>&)	Constructor. Constructs a weak pointer from a shared pointer.
operator=(const WeakPtr<T>&)	Copy assignment.
operator=(const SharedPtr<T>&)	Assignment. Assigns a weak pointer from a shared pointer.
∼WeakPtr()	Destructor. Decrements weak count.
Assign(T*, const SharedCount<T>&)	Implementation detail to support EnableSharedFromThis<T, U>(ShareableFromThis<T>*, U*, const SharedCount<U>&) function.
GetCount() const	Returns the weak count.
GetPtr() const	Returns a pointer to the counted object.

GetUseCount() const	Returns the use count of the counted object.
IsExpired() const	Returns true if the use count has gone to zero and the counted object has been destroyed.
Lock() const	If the weak pointer has not been expired, returns a shared pointer to the counted object; otherwise returns null shared pointer.
Reset()	Resets the weak pointer to null.
Swap(WeakPtr<T>&)	Exchanges the contents of this weak pointer with another.

1.1.48.3.1 WeakPtr() Member Function

Constructor. Constructs null weak pointer.

Syntax

```
public nothrow WeakPtr();
```

1.1.48.3.2 WeakPtr(const WeakPtr<T>&) Member Function

Copy constructor. Increments weak count.

Syntax

```
public nothrow WeakPtr(const WeakPtr<T>& that);
```

Parameters

Name	Type	Description
that	const WeakPtr<T>&	A weak pointer to copy.

1.1.48.3.3 WeakPtr(const SharedPtr<T>&) Member Function

Constructor. Constructs a weak pointer from a shared pointer.

Syntax

```
public nothrow WeakPtr(const SharedPtr<T>& that);
```

Parameters

Name	Type	Description
that	const SharedPtr<T>&	A shared pointer.

1.1.48.3.4 operator=(const WeakPtr<T>&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const WeakPtr<T>& that);
```

Parameters

Name	Type	Description
that	const WeakPtr<T>&	A weak pointer to assign.

1.1.48.3.5 operator=(const SharedPtr<T>&) Member Function

Assignment. Assigns a weak pointer from a shared pointer.

Syntax

```
public nothrow void operator=(const SharedPtr<T>& that);
```

Parameters

Name	Type	Description
that	const SharedPtr<T>&	A shared pointer to assign.

1.1.48.3.6 ~WeakPtr() Member Function

Destructor. Decrements weak count.

Syntax

```
public default nothrow ~WeakPtr();
```

1.1.48.3.7 Assign(T*, const SharedCount<T>&) Member Function

Implementation detail to support EnableSharedFromThis<T, U>(ShareableFromThis<T>*, U*, const SharedCount<U>&) function.

Syntax

```
public nothrow void Assign(T* ptr_, const SharedCount<T>& count_);
```

Parameters

Name	Type	Description
ptr_	T*	A pointer to counted object.

count_	const SharedCount<T>&	A shared count.

1.1.48.3.8 GetCount() const Member Function

Returns the weak count.

Syntax

```
public inline nothrow const WeakCount<T>& GetCount() const;
```

Returns

const WeakCount<T>&

Returns the weak count.

1.1.48.3.9 GetPtr() const Member Function

Returns a pointer to the counted object.

Syntax

```
public inline nothrow T* GetPtr() const;
```

Returns

T*

Returns a pointer to the counted object.

1.1.48.3.10 GetUseCount() const Member Function

Returns the use count of the counted object.

Syntax

```
public nothrow int GetUseCount() const;
```

Returns

int

Returns the use count of the counted object.

1.1.48.3.11 IsExpired() const Member Function

Returns true if the use count has gone to zero and the counted object has been destroyed.

Syntax

```
public nothrow bool IsExpired() const;
```

Returns

bool

Returns true if the use count has gone to zero and the counted object has been destroyed.

1.1.48.3.12 Lock() const Member Function

If the weak pointer has not been expired, returns a shared pointer to the counted object; otherwise returns null shared pointer.

Syntax

```
public nothrow SharedPtr<T> Lock() const;
```

Returns

SharedPtr<T>

If the weak pointer has not been expired, returns a shared pointer to the counted object; otherwise returns null shared pointer.

1.1.48.3.13 Reset() Member Function

Resets the weak pointer to null.

Syntax

```
public nothrow void Reset();
```

1.1.48.3.14 Swap(WeakPtr<T>&) Member Function

Exchanges the contents of this weak pointer with another.

Syntax

```
public nothrow void Swap(WeakPtr<T>& that);
```

Parameters

Name	Type	Description
that	WeakPtr<T>&	A weak pointer to exchange contents with.

1.1.49 uhuge Class

128-bit unsigned integer type.

Syntax

```
public class uhuge;
```

1.1.49.1 Member Functions

Member Function	Description
uhuge()	Constructor. Initializes the value to zero.
uhuge(const uhuge&)	Copy constructor.
uhuge(uhuge&&)	Move constrcutor.
uhuge(ulong)	Constructor. Initializes the value to specified 64-bit value.
uhuge(ulong, ulong)	Constructor. Initializes the value to $2^{64}h\_ + l\_$.
operator=(const uhuge&)	Copy assignment.
operator=(uhuge&&)	Move assignment.
operator–()	Decrements the value by one.
operator++()	Increments the value by one.

1.1.49.1.1 uhuge() Member Function

Constructor. Initializes the value to zero.

Syntax

```
public nothrow uhuge();
```

1.1.49.1.2 uhuge(const uhuge&) Member Function

Copy constructor.

Syntax

```
public nothrow uhuge(const uhuge& that);
```

Parameters

Name	Type	Description
that	const uhuge&	An value to copy from.

1.1.49.1.3 uhuge(uhuge&&) Member Function

Move constrcutor.

Syntax

```
public nothrow uhuge(uhuge&& that);
```

Parameters

Name	Type	Description
that	uhuge&&	A value to move from.

1.1.49.1.4 uhuge(ulong) Member Function

Constructor. Initializes the value to specified 64-bit value.

Syntax

```
public nothrow uhuge(ulong l_);
```

Parameters

Name	Type	Description
l_	ulong	An unsigned 64-bit value.

1.1.49.1.5 uhuge(ulong, ulong) Member Function

Constructor. Initializes the value to $2^{64}h\_ + l\_$.

Syntax

```
public nothrow uhuge(ulong h_, ulong l_);
```

Parameters

Name	Type	Description
h_	ulong	High part.
l_	ulong	Low part.

1.1.49.1.6 operator=(const uhuge&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const uhuge& that);
```

Parameters

Name	Type	Description
that	const uhuge&	A value to assign from.

1.1.49.1.7 operator=(uhuge&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(uhuge&& that);
```

Parameters

Name	Type	Description
that	uhuge&&	A value to move from.

1.1.49.1.8 operator–() Member Function

Decrements the value by one.

Syntax

```
public nothrow uhuge& operator--();
```

Returns

uhuge&

Returns the value.

1.1.49.1.9 operator++() Member Function

Increments the value by one.

Syntax

```
public nothrow uhuge& operator++();
```

Returns

uhuge&

Returns the value.

1.1.49.2 Nonmember Functions

Function	Description
operator&(uhuge, uhuge)	Bitwise-AND operation of two 128-bit values.

operator/(uhuge, uhuge)	Division of two 128-bit values.
operator==(uhuge, uhuge)	Compares two 128-bit values for equality.
operator<(uhuge, uhuge)	Compares two 128-bit values for less than relationship.
operator-(uhuge, uhuge)	Subtracts a 128-bit value from 128-bit value.
operator%(uhuge, uhuge)	Computes the remainder of division of two 128-bit values.
operator∼(uhuge)	Computes bitwise complement of a 128-bit value.
operator—(uhuge, uhuge)	Computes bitwise-OR of two 128-bit values.
operator+(uhuge, uhuge)	Computes the sum of two 128-bit values.
operator<<(uhuge, uhuge)	Shifts a 128-bit value left.
operator>>(uhuge, uhuge)	Shifts a 128-bit value right.
operator*(uhuge, uhuge)	Computes the product of two 128-bit values.
operatorˆ(uhuge, uhuge)	Computes the bitwise-XOR operation of two 128-bit values.
divmod(uhuge, uhuge)	Computes quotient and remainder of division of two 128-bit values.
divmod(uhuge, uint)	Computes quotient and remainder of division of a 128-bit value divided by a 32-bit value.
mul(uhuge, uhuge)	Computes the 256-bit product of two 128-bit values.

1.1.49.2.1 operator&(uhuge, uhuge) Function

Bitwise-AND operation of two 128-bit values.

Syntax

```
public inline nothrow uhuge operator&(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns bitwise AND of *left* and *right*.

1.1.49.2.2 operator/(uhuge, uhuge) Function

Division of two 128-bit values.

Syntax

```
public uhuge operator/(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	Dividend.
right	uhuge	Divisor.

Returns

uhuge

Returns *left* divided by *right*.

1.1.49.2.3 operator==(uhuge, uhuge) Function

Compares two 128-bit values for equality.

Syntax

```
public inline nothrow bool operator==(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

bool

Returns true, if the first 128-bit value is equal to the second 128-bit value, false otherwise.

1.1.49.2.4 operator<(uhuge, uhuge) Function

Compares two 128-bit values for less than relationship.

Syntax

```
public inline nothrow bool operator<(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

bool

Returns true, if the first 128-bit value is less than the second 128-bit value, false otherwise.

1.1.49.2.5 operator-(uhuge, uhuge) Function

Subtracts a 128-bit value from 128-bit value.

Syntax

```
public inline nothrow uhuge operator-(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns $left - right$.

1.1.49.2.6 operator%(uhuge, uhuge) Function

Computes the remainder of division of two 128-bit values.

Syntax

```
public uhuge operator%(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	Dividend.
right	uhuge	Divisor.

Returns

uhuge

Returns $left\%right$.

1.1.49.2.7 operator∼(uhuge) Function

Computes bitwise complement of a 128-bit value.

Syntax

```
public inline nothrow uhuge operator~(uhuge x);
```

Parameters

Name	Type	Description
x	uhuge	A 128-bit value.

Returns

uhuge

Returns bitwise complement of x.

1.1.49.2.8 operator—(uhuge, uhuge) Function

Computes bitwise-OR of two 128-bit values.

Syntax

```
public inline nothrow uhuge operator|(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns bitwise-OR of $left$ and $right$.

1.1.49.2.9 operator+(uhuge, uhuge) Function

Computes the sum of two 128-bit values.

Syntax

```
public inline nothrow uhuge operator+(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns $left + right$.

1.1.49.2.10 operator<<(uhuge, uhuge) Function

Shifts a 128-bit value left.

Syntax

```
public nothrow uhuge operator<<(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	A 128-bit value to shift.
right	uhuge	Number of bits to shift.

Returns

uhuge

Returns $left$ shifted $right$ bits left.

1.1.49.2.11 operator>>(uhuge, uhuge) Function

Shifts a 128-bit value right.

Syntax

```
public nothrow uhuge operator>>(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	A 128-bit value to shift.
right	uhuge	Number of bits to shift.

Returns

uhuge

Returns *left* shifted *right* bits right.

1.1.49.2.12 operator*(uhuge, uhuge) Function

Computes the product of two 128-bit values.

Syntax

```
public uhuge operator*(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	Ths first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns $left \times right$.

1.1.49.2.13 operator^(uhuge, uhuge) Function

Computes the bitwise-XOR operation of two 128-bit values.

Syntax

```
public inline nothrow uhuge operator^(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

uhuge

Returns *left* XOR *right*.

1.1.49.2.14 divmod(uhuge, uhuge) Function

Computes quotient and remainder of division of two 128-bit values.

Syntax

```
public Pair<uhuge, uhuge> divmod(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	Dividend.
right	uhuge	Divisor.

Returns

Pair<uhuge, uhuge>

Returns a pair in which the first value is the quotient and the second value is the remainder of $left/right$.

1.1.49.2.15 divmod(uhuge, uint) Function

Computes quotient and remainder of division of a 128-bit value divided by a 32-bit value.

Syntax

```
public Pair<uhuge, uint> divmod(uhuge left, uint right);
```

Parameters

Name	Type	Description
left	uhuge	Dividend.
right	uint	Divisor.

Returns

Pair<uhuge, uint>

Returns a pair in which the first value is the quotient and the second value is the remainder of $left/right$.

1.1.49.2.16 mul(uhuge, uhuge) Function

Computes the 256-bit product of two 128-bit values.

Syntax

```
public Pair<uhuge, uhuge> mul(uhuge left, uhuge right);
```

Parameters

Name	Type	Description
left	uhuge	The first 128-bit value.
right	uhuge	The second 128-bit value.

Returns

Pair<uhuge, uhuge>

Returns a pair in which the first value is the high part and the second value is the low part of of $left \times right$.

1.1.50 Date Class

Syntax

```
public class Date;
```

1.1.50.1 Member Functions

Member Function	Description
Date()	
Date(const Date&)	
Date(Date&&)	
Date(ushort, byte, byte)	
operator=(const Date&)	
operator=(Date&&)	
Day() const	
Month() const	
Year() const	

1.1.50.1.1 Date() Member Function

Syntax

```
public Date();
```

1.1.50.1.2 Date(const Date&) Member Function

Syntax

```
public nothrow Date(const Date& that);
```

1.1.50.1.3 Date(Date&&) Member Function

Syntax

```
public nothrow Date(Date&& that);
```

1.1.50.1.4 Date(ushort, byte, byte) Member Function

Syntax

```
public Date(ushort year_, byte month_, byte day_);
```

1.1.50.1.5 operator=(const Date&) Member Function

Syntax

```
public nothrow void operator=(const Date& that);
```

1.1.50.1.6 operator=(Date&&) Member Function

Syntax

```
public nothrow void operator=(Date&& that);
```

1.1.50.1.7 Day() const Member Function

Syntax

```
public inline nothrow byte Day() const;
```

1.1.50.1.8 Month() const Member Function

Syntax

```
public inline nothrow byte Month() const;
```

1.1.50.1.9 Year() const Member Function

Syntax

```
public inline nothrow ushort Year() const;
```

1.2 Type Definitions

Name	Type	Description
string	String	An alias for String class.

1.3 Functions

Function	Description
Abs<T>(const T&)	Returns the absolute value of the argument.
Accumulate<I, T, Op>(I, I, T, Op)	Accumulates a sequence with respect to a binary operation.
BackInserter<C>(C&)	Returns a BackInsertIterator<C> for a back insertion sequence.
Copy<I, O>(I, I, O)	Copies a sequence.
CopyBackward<I, O>(I, I, O)	Copies a source sequence to a target sequence starting from the end of the source sequence.
Count<I, P>(I, I, P)	Counts the number of elements in a sequence that satisfy a predicate.
Count<I, T>(I, I, const T&)	Counts the number of elements in a sequence that are equal to the given value.
Distance<I>(I, I)	Returns the distance between two forward iterators.
Distance<I>(I, I)	Returns the distance between two random access iterators.
EnableSharedFromThis<T, U>(ShareableFromThis<T>*, U*, const SharedCount<U>&)	A function that enables the *shared from this* idiom. Implementation detail.
EnableSharedFromThis<T>(void*, void*, const SharedCount<T>&)	An empty function that catches classes that are not shareable from this. Implementation detail.
Equal<I1, I2>(I1, I1, I2, I2)	Compares two sequences for equality.
Equal<I1, I2, R>(I1, I1, I2, I2, R)	Compares two sequences for equality using the given equality relation.
EqualRange<I, T>(I, I, const T&)	Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence.

EqualRange<I, T, R>(I, I, const T&, R)	Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence. Uses the given ordering relation to infer equality.
Factorial<U>(U)	Returns a factorial of the argument.
Find<I, P>(I, I, P)	Searches the first occurrence of a value from a sequence that matches a predicate.
Find<I, T>(I, I, const T&)	Searches a value from a sequence.
ForEach<I, F>(I, I, F)	Applies a function object for each element of a sequence.
FrontInserter<C>(C&)	Returns a FrontInsertIterator<C> for a front insert sequence.
Gcd<T>(T, T)	Returns the greatest common divisor of two values.
HexChar(byte)	Returns hexadecimal character representation of a four-bit value.
IdentityElement<T>(Plus<T>)	Returns the identity element of addition, that is: T(0).
IdentityElement<T>(Multiplies<T>)	Returns the identity element of multiplication, that is T(1).
Inserter<C, I>(C&, I)	Returns an InsertIterator<C> for an insertion sequence and its iterator.
InsertionSort<I>(I, I)	Sorts a sequence of values using insertion sort algorithm.
InsertionSort<I, R>(I, I, R)	Sorts a sequence of values using insertion sort algorithm and given ordering relation.
IsAlpha(char)	Returns true if the given character is an alphabetic character, false otherwise.

IsAlphanumeric(char)	Returns true if the given character is an alphanumeric character, false otherwise.
IsControl(char)	Returns true if the given character is a control character, false otherwise.
IsDigit(char)	Returns true if the given character is a decimal digit, false otherwise.
IsGraphic(char)	Returns true if the given character is a graphical character, false otherwise.
IsHexDigit(char)	Returns true if the given character is a hexadecimal digit, false otherwise.
IsLower(char)	Returns true if the given character is a lower case letter, false otherwise.
IsPrintable(char)	Returns true if the given character is a printable character, false otherwise.
IsPunctuation(char)	Returns true if the given character is a punctuation character, false otherwise.
IsSpace(char)	Returns true if the given character is a space character, false otherwise.
IsUpper(char)	Returns true if the given character is an upper case letter, false otherwise.
LexicographicalCompare<I1, I2>(I1, I1, I2, I2)	Returns true if the first sequence comes lexicographically before the second sequence, false otherwise.
LexicographicalCompare<I1, I2, R>(I1, I1, I2, I2, R)	Returns true if the first sequence comes lexicographically before the second sequence according to the given ordering relation, false otherwise.

LowerBound<I, T>(I, I, const T&)	Finds a position of the first element in a sorted sequence that is greater than or equal to the given value.
LowerBound<I, T, R>(I, I, const T&, R)	Finds a position of the first element in a sorted sequence that is greater than or equal to the given value according to the given ordering relation.
MakePair<T, U>(const T&, const U&)	Returns a pair composed of the given values.
Max<T>(const T&, const T&)	Returns the maximum of two values.
MaxElement<I>(I, I)	Returns the position of the first occurrence of the largest element in a sequence of elements.
MaxElement<I, R>(I, I, R)	Returns the position of the first occurrence of the largest element according to the given ordering relation in a sequence of elements.
MaxValue<I>()	Returns the largest value of an integer type.
MaxValue(byte)	Returns the maximum value of **byte**: 255.
MaxValue(int)	Returns the maximum value of **int**: 2147483647.
MaxValue(long)	Returns the maximum value of **long**: 9223372036854775807.
MaxValue(sbyte)	Returns the maximum value of **sbyte**: 127.
MaxValue(short)	Returns the maximum value of **short**: 32767.
MaxValue(uint)	Returns the maximum value of **uint**: 4294967295.
MaxValue(ulong)	Returns the maximum value of **ulong**: 18446744073709551615.

MaxValue(ushort)	Returns the maximum value of **ushort**: 65535.
Median<T, R>(const T&, const T&, const T&, R)	Returns the median of three values according to the given ordering relation.
Median<T>(const T&, const T&, const T&)	Returns the median of three values.
Min<T>(const T&, const T&)	Returns the minimum of two values.
MinElement<I>(I, I)	Returns the position of the first occurrence of the smallest element in a sequence of elements.
MinElement<I, R>(I, I, R)	Returns the position of the first occurrence of the smallest element according to the given ordering relation in a sequence of elements.
MinValue<I>()	Returns the smallest value of an integer type.
MinValue(byte)	Returns the minimum value of **byte**: 0.
MinValue(int)	Returns the minimum value of **int**: -2147483648.
MinValue(long)	Returns the minimum value of **long**: -9223372036854775808.
MinValue(sbyte)	Returns the minimum value of **sbyte**: -128.
MinValue(short)	Returns the minimum value of **short**: -32768.
MinValue(uint)	Returns the minimum value of **uint**: 0.
MinValue(ulong)	Returns the minimum value of **ulong**: 0.
MinValue(ushort)	Returns the minimum value of **ushort**: 0.
Move<I, O>(I, I, O)	Moves a sequence.
MoveBackward<I, O>(I, I, O)	Moves a source sequence to a target sequence starting from the end of the source sequence.

Next<I>(I, int)	Returns a forward iterator advanced the specified number of steps.
Next<I>(I, int)	Returns a random access iterator advanced the specified offset.
NextPermutation<I>(I, I)	Computes the lexicographically next permutation of a sequence of elements.
NextPermutation<I, R>(I, I, R)	Computes the lexicographically next permutation of a sequence of elements according to the given ordering relation.
Now()	Returns current time point value from computer's real time clock.
ParseBool(const string&)	Parses a Boolean value “true” or “false” from the given string and returns it.
ParseBool(const string&, bool&)	Parses a Boolean value “true” or “false” from the given string and returns true if the parsing was successful, false if not.
ParseDouble(const string&)	Parses a **double** value from the given string and returns it.
ParseDouble(const string&, double&)	Parses a **double** value from the given string and returns true if the parsing was successful, false if not.
ParseHex(const string&)	Parses a hexadecimal value from a string.
ParseHex(const string&, ulong&)	Parses a hexadecimal value from a string and returns true if the parsing was successful.
ParseHex(const string&, uhuge&)	Parses a hexadecimal 128-bit value from a string and returns true, if the parsing was successful.

ParseHexUHuge(const string&)	Parses a 128-bit decimal value from a string.
ParseInt(const string&)	Parses an **int** from the given string and returns it.
ParseInt(const string&, int&)	Parses an **int** value from the given string and returns true if the parsing was successful, false if not.
ParseUHuge(const string&)	Parses a decimal 128-bit value from a string.
ParseUHuge(const string&, uhuge&)	Parses a decimal 128-bit value from a string and returns true, if the parsing was successful.
ParseUInt(const string&)	Parses an **uint** from the given string and returns it.
ParseUInt(const string&, uint&)	Parses an **uint** value from the given string and returns true if the parsing was successful, false if not.
ParseULong(const string&)	Parses an **ulong** from the given string and returns it.
ParseULong(const string&, ulong&)	Parses an **ulong** value from the given string and returns true if the parsing was successful, false if not.
PrevPermutation<I>(I, I)	Computes the lexicographically previous permutation of a sequence of elements.
PrevPermutation<I, R>(I, I, R)	Computes the lexicographically previous permutation according to the given ordering relation of a sequence of elements.
PtrCast<U, T>(const SharedPtr<T>&)	Casts a shared pointer.
Reverse<I>(I, I)	Reverses a sequence.
Reverse<I>(I, I)	Reverses a sequence.

Rvalue<T>(T&&)	Converts an argument to an rvalue so that it can be moved.
Select_0_2<T, R>(const T&, const T&, R)	Returns the smaller of two values according to the given ordering relation.
Select_0_3<T, R>(const T&, const T&, const T&, R)	Returns the smallest of tree values according to the given ordering relation.
Select_1_2<T, R>(const T&, const T&, R)	Returns the larger of two values according to the given ordering relation.
Select_1_3<T, R>(const T&, const T&, const T&, R)	Returns the median of three values according to the given ordering relation.
Select_1_3_ab<T, R>(const T&, const T&, const T&, R)	Returns the median of three values when the first two are in increasing order according to the given ordering relation.
Select_2_3<T, R>(const T&, const T&, const T&, R)	Returns the largest of three values according to the given ordering relation.
Sort<C>(C&)	Sorts the elements of a forward container to increasing order.
Sort<C>(C&)	Sorts the elements of a random access container to increasing order.
Sort<C, R>(C&, R)	Sorts the elements of a forward container to order according to the given ordering relation.
Sort<C, R>(C&, R)	Sorts the elements of a random access container to order according to the given ordering relation.
Sort<I>(I, I)	Sorts the elements of a sequence to increasing order.
Sort<I, R>(I, I, R)	Sorts the elements of a sequence to order according to the given ordering relation.

Swap<T>(T&, T&)	Exchanges two values.
ToHexString<U>(U)	Converts an unsigned integer value to hexadecimal string representation.
ToHexString(byte)	Converts a **byte** to hexadecimal string representation.
ToHexString(uhuge)	Returns a 128-bit value converted to hexadecimal representation.
ToHexString(uint)	Converts an **uint** to hexadecimal string representation.
ToHexString(ulong)	Converts an **ulong** to hexadecimal string representation.
ToHexString(ushort)	Converts an **ushort** to hexadecimal string representation.
ToLower(const string&)	Converts a string to lower case.
ToString<I, U>(I)	Converts a signed integer value to string representation.
ToString<U>(U)	Converts an unsigned integer value to string representation.
ToString(bool)	Converts a Boolean value to string representation.
ToString(byte)	Converts a **byte** to string representation.
ToString(char)	Converts a character to string representation.
ToString(double)	Converts a **double** to string representation.
ToString(double, int)	Converts a **double** to string representation using the given maximum number of decimal places.

ToString(int)	Converts an **int** to string representation.
ToString(long)	Converts a **long** to string representation.
ToString(uhuge)	Converts a 128-bit value to string representation.
ToString(sbyte)	Converts an **sbyte** to string representation.
ToString(short)	Converts a **short** to string representation.
ToString(uint)	Converts an **uint** to string representation.
ToString(ulong)	Converts an **ulong** to string representation.
ToString(ushort)	Converts an **ushort** to string representation.
ToUpper(const string&)	Converts a string to upper case.
Transform<I, O, F>(I, I, O, F)	Transforms an input sequence to an output sequence using a unary function.
Transform<I1, I2, O, F>(I1, I1, I2, O, F)	Transforms two input sequences to an ouput sequence using a binary function.
UpperBound<I, T>(I, I, const T&)	Finds a position of the first element in a sorted sequence that is greater than the given value.
UpperBound<I, T, R>(I, I, const T&, R)	Finds a position of the first element in a sorted sequence that is greater than the given value according to the given ordering relation.
endl()	Returns EndLine object that represents an end of line character.

operator==(Date, Date)
operator<(Date, Date)
CurrentDate()
ParseDate(const string&)
ToString(Date)
ToUtf8(uint)

1.3.51 Abs<T>(const T&) Function

Returns the absolute value of the argument.

Syntax

```
public inline nothrow T Abs<T>(const T& x);
```

Constraint

where T is OrderedAdditiveGroup;

Parameters

Name	Type	Description
x	const T&	A value.

Returns

T

if $x < T(0)$ returns $-x$, else returns x.

Example

```
using System;

//  Writes:
//  0
//  5
//  10

void main()
{
    int zero = 0;
    int minusFive = -5;
    int ten = 10;
//  ...
    Console.Out() << Abs(zero) << endl();
    Console.Out() << Abs(minusFive) << endl();
    Console.Out() << Abs(ten) << endl();
}
```

1.3.52 Accumulate<I, T, Op>(I, I, T, Op) Function

Accumulates a sequence with respect to a binary operation.

Syntax

```
public nothrow T Accumulate<I, T, Op>(I begin, I end, T init, Op op);
```

Constraint

where I is InputIterator and T is Semiregular and Op is BinaryOperation and Op.FirstArgumentType is T and Op.SecondArgumentType is I.ValueType;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
init	T	Initial value.
op	Op	A binary operation.

Returns

T

Returns accumulated result.

Remarks

When the binary operation is Plus<T> and *init* is zero calculates the sum of a sequence. When the binary operation is Multiplies<T> and *init* is one calculates the product of a sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  6

void main()
{
    List<int> ints;
    ints.Add(1);
    ints.Add(2);
    ints.Add(3);
    int init = 0;
    int sum = Accumulate(ints.CBegin(), ints.CEnd(), init, Plus<int>());
    Console.WriteLine(sum);
}
```

Implementation

algorithm.cm, page 8

1.3.53 BackInserter<C>(C&) Function

Returns a BackInsertIterator<C> for a back insertion sequence.

Syntax

```
public nothrow BackInsertIterator<C> BackInserter<C>(C& c);
```

Constraint

where C is BackInsertionSequence;

Parameters

Name	Type	Description
c	C&	A back insertion sequence.

Returns

BackInsertIterator<C>

Returns a BackInsertIterator<C> for a back insertion sequence.

Remarks

A BackInsertIterator<C> is an output iterator that inserts elements to the end of a back insertion sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  1, 2, 3

void main()
{
    Set<int> s;
    s.Insert(2);
    s.Insert(3);
    s.Insert(1);
//  Set is a sorted container, so it contains now 1, 2, 3...

    List<int> list; // list is a model of a back insertion sequence
```

```
//  Copy ints from s to the end of list using BackInsertIterator...
    Copy(s.CBegin(), s.CEnd(), BackInserter(list));

//  ForwardContainers containing ints can be put to OutputStream...
    Console.Out() << list << endl();
}
```

1.3.54 Copy<I, O>(I, I, O) Function

Copies a sequence.

Syntax

`public O Copy<I, O>(I begin, I end, O to);`

Constraint

where I is InputIterator and O is OutputIterator and CopyAssignable<O.ValueType, I.ValueType>;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a the source sequence.
end	I	An input iterator pointing one past the end of a source sequence.
to	O	An output iterator pointing to the beginning of the target sequence.

Returns

O

Returns an output iterator pointing one past the end of the copied sequence.

Remarks

The source and target sequences may overlap, but then the iterator pointing to the beginning of the target sequence must point to an object coming before the object pointed by the beginning iterator of the source sequence.

Example

```
using System;
using System.Collections;
using System.IO;

//  Writes:
//  0, 1, 2, 3, 4, 5, 6, 7, 8, 9
//  5, 6, 7, 8, 9, 0, 1, 2, 3, 4

void main()
{
    List<int> source;
    int n = 10;
    for (int i = 0; i < n; ++i)
    {
        source.Add(i);
    }
    Console.Out() << source << endl();
    List<int> target;
    Copy(source.CBegin() + n / 2, source.CEnd(), BackInserter(target));
    Copy(source.CBegin(), source.CBegin() + n / 2, BackInserter(target));
    Console.Out() << target << endl();
}
```

Implementation

algorithm.cm, page 2

1.3.55 CopyBackward<I, O>(I, I, O) Function

Copies a source sequence to a target sequence starting from the end of the source sequence.

Syntax

```
public O CopyBackward<I, O>(I begin, I end, O to);
```

Constraint

where I is BidirectionalIterator and O is BidirectionalIterator and CopyAssignable<O.ValueType, I.ValueType>;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of the source sequence.

end	I	A bidirectional iterator pointing one past the end of the source sequence.
to	O	A bidirectional iterator pointing one past the end of the target sequence.

Returns

O

Returns a birectional iterator pointing to the beginning of the target sequence.

Remarks

The source and target sequences may overlap, but then the end iterator of the target sequence must point to an element coming after the element pointed by end iterator of the source sequence.

Example

```
using System;
using System.Collections;
using System.IO;

//  Writes:
//  0, 1, 2, 3

void main()
{
    List<int> list;
    list.Add(0);
    list.Add(2);
    list.Add(3);
    list.Add(3);
    CopyBackward(list.CBegin() + 1, list.CEnd() - 1, list.End());
    list[1] = 1;
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 2

1.3.56 Count<I, P>(I, I, P) Function

Counts the number of elements in a sequence that satisfy a predicate.

Syntax

`public nothrow int Count<I, P>(I begin, I end, P p);`

Constraint

where I is InputIterator and P is UnaryPredicate and P.ArgumentType is I.ValueType;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
p	P	A unary predicate.

Returns

int

Returns the number of elements that satisfy *p*.

Example

```
using System;
using System.Concepts;
using System.Collections;

//  Writes:
//  3

public class Even<I>: UnaryPred<I> where I is SignedInteger
{
    public inline nothrow bool operator()(I n) const
    {
        return n % I(2) == I(0);
    }
}

void main()
{
    List<int> list;
    list.Add(0);
    list.Add(2);
    list.Add(3);
    list.Add(4);
    list.Add(5);
```

```
    Console.Out() << Count(list.CBegin(), list.CEnd(), Even<int>()) <<
        endl();
}
```

Implementation

algorithm.cm, page 7

1.3.57 Count<I, T>(I, I, const T&) Function

Counts the number of elements in a sequence that are equal to the given value.

Syntax

`public nothrow int Count<I, T>(I begin, I end, const T& value);`

Constraint

where I is InputIterator and T is Semiregular and EqualityComparable<T, I.ValueType>;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
value	const T&	A value.

Returns

int

Returns the number of elements equal to *value*.

Example

```
using System;
using System.Collections;

//  Writes:
//  3

void main()
{
```

```
    List<string> animals;
    animals.Add("cat");
    animals.Add("cat");
    animals.Add("lion");
    animals.Add("dog");
    animals.Add("mouse");
    animals.Add("lion");
    animals.Add("moose");
    animals.Add("lion");
    animals.Add("bear");
    animals.Add("dog");
    string lion("lion");
    Console.Out() << Count(animals.CBegin(), animals.CEnd(), lion) <<
        endl();
}
```

Implementation

algorithm.cm, page 7

1.3.58 Distance<I>(I, I) Function

Returns the distance between two forward iterators.

Syntax

`public nothrow int Distance<I>(I first, I last);`

Constraint

where I is ForwardIterator;

Parameters

Name	Type	Description
first	I	The first forward iterator.
last	I	The second forward iterator reachable from the first forward iterator.

Returns

int

Returns the number of steps that the *first* iterator must be incremented to reach the *last* iterator.

Remarks

The *last* iterator must be reachable from the *first* iterator.

Implementation

algorithm.cm, page 3

1.3.59 Distance<I>(I, I) Function

Returns the distance between two random access iterators.

Syntax

```
public inline nothrow int Distance<I>(I first, I last);
```

Constraint

where I is RandomAccessIterator;

Parameters

Name	Type	Description
first	I	The first random access iterator.
last	I	The second random access iterator.

Returns

int

Returns $last - first$.

Example

```
using System;
using System.Collections;

//  Writes:
//  4

void main()
{
    List<int> list;
    for (int i = 0; i < 10; ++i)
    {
        list.Add(i + 3);
    }
    List<int>.ConstIterator it = Find(list.CBegin(), list.CEnd(), 7);
    int indexOfSeven = Distance(list.CBegin(), it);
    Console.Out() << indexOfSeven << endl();
}
```

Implementation

algorithm.cm, page 3

1.3.60 EnableSharedFromThis<T, U>(ShareableFromThis<T>*, U*, const SharedCount<U>&) Function

A function that enables the *shared from this* idiom. Implementation detail.

Syntax

```
public nothrow void EnableSharedFromThis<T, U>(ShareableFromThis<T>* left, U*
right, const SharedCount<U>& count);
```

Parameters

Name	Type	Description
left	ShareableFromThis<T>*	A pointer to a class derived from ShareableFromThis<T>.
right	U*	A pointer to class derived from ShareableFromThis<T>.
count	const SharedCount<U>&	A shared count.

1.3.61 EnableSharedFromThis<T>(void*, void*, const SharedCount<T> Function

An empty function that catches classes that are not shareable from this. Implementation detail.

Syntax

```
public inline nothrow void EnableSharedFromThis<T>(void* __parameter0, void*
__parameter1, const SharedCount<T>& __parameter2);
```

Parameters

Name	Type	Description
__parameter0	void*	Pointer to any object.
__parameter1	void*	Pointer to any object.
__parameter2	const SharedCount<T>&	A shared count.

1.3.62 Equal<I1, I2>(I1, I1, I2, I2) Function

Compares two sequences for equality.

Syntax

```
public inline nothrow bool Equal<I1, I2>(I1 first1, I1 last1, I2 first2, I2 last2);
```

Constraint

where I1 is InputIterator and I2 is InputIterator and EqualityComparable<I1.ValueType, I2.ValueType>;

Parameters

Name	Type	Description
first1	I1	An input iterator pointing to the beginning of the first sequence.
last1	I1	An input iterator pointer one past the end of the first sequence.
first2	I2	An input iterator pointing to the beginning of the second sequence.
last2	I2	An input iterator pointing one past the end of the second sequence.

Returns

bool

Returns true if the first sequence is equal to the second sequence, false otherwise.

Remarks

Two sequences are equal if they contain the same number of elements that compare pairwise equal. Uses the EqualTo2<T, U> binary predicate to compare the elements of the sequences.

Example

```
using System;
using System.Collections;

//  Writes:
//  set: 41, 6334, 11478, 15724, 18467, 19169, 24464, 26500, 26962, 29358
```

```
// list: 41, 18467, 6334, 26500, 19169, 15724, 11478, 29358, 26962,
   24464
// sorted list: 41, 6334, 11478, 15724, 18467, 19169, 24464, 26500,
   26962, 29358
// this was expected

void main()
{
    Set<int> set;
    List<int> list;
    int n = 10;
    for (int i = 0; i < n; ++i)
    {
        int r = rand();
        set.Insert(r);
        list.Add(r);
    }
    Console.Out() << "set: " << set << endl();
    Console.Out() << "list: " << list << endl();
    Sort(list);
    Console.Out() << "sorted list: " << list << endl();
    if (Equal(list.CBegin(), list.CEnd(), set.CBegin(), set.CEnd()))
    {
        Console.Out() << "this was expected" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

algorithm.cm, page 13

1.3.63 Equal<I1, I2, R>(I1, I1, I2, I2, R) Function

Compares two sequences for equality using the given equality relation.

Syntax

```
public nothrow bool Equal<I1, I2, R>(I1 first1, I1 last1, I2 first2, I2 last2,
R r);
```

Constraint

where I1 is InputIterator and I2 is InputIterator and Relation<R, I1.ValueType, I2.ValueType>;

Parameters

Name	Type	Description
first1	I1	An input iterator pointing to the beginning of the first sequence.
last1	I1	An input iterator pointing one past the end of the first sequence.
first2	I2	An input iterator pointing to the beginning of the second sequence.
last2	I2	An input iterator pointing one past the end of the second sequence.
r	R	An equality relation.

Returns

bool

Returns true if the first sequence is equal to the second sequence according to the given equality relation.

Example

```
using System;
using System.Collections;

//  Writes:
//  true

public class A
{
    public A(): id()
    {
    }
    public A(const string& id_): id(id_)
    {
    }
    public nothrow const string& Id() const
    {
        return id;
    }
    private string id;
}

public class ALess: Rel<A>
{
    public nothrow inline bool operator()(const A& left, const A& right)
        const
```

```
    {
        return left.Id() < right.Id();
    }
}

public class AEq: Rel<A>
{
    public nothrow inline bool operator()(const A& left, const A& right)
        const
    {
        return left.Id() == right.Id();
    }
}

void main()
{
    List<A> list;
    list.Add(A("bar"));
    list.Add(A("baz"));
    list.Add(A("foo"));
    Set<A, ALess> set;
    set.Insert(A("foo"));
    set.Insert(A("bar"));
    set.Insert(A("baz"));
    Console.Out() << Equal(list.CBegin(), list.CEnd(), set.CBegin(), set.
        CEnd(), AEq()) << endl();
}
```

Implementation

algorithm.cm, page 12

1.3.64 EqualRange<I, T>(I, I, const T&) Function

Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence.

Syntax

```
public Pair<I, I> EqualRange<I, T>(I first, I last, const T& value);
```

Constraint

where I is ForwardIterator and TotallyOrdered<T, I.ValueType>;

Parameters

Name	Type	Description
first	I	A forward iterator that points to the beginning of a sorted sequence.
last	I	A forward iterator that points to one past the end of a sorted sequence.
value	const T&	A value.

Returns

Pair<I, I>

Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence.

Remarks

If the value is not found in the sorted sequence, returns a pair of iterators that form an empty range (that is: a range with two equal iterators.) The iterators point to the position where the given value would be if it were in the sorted sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  number 1 occurs 0 times
//  number 2 occurs 1 times
//  number 3 occurs 2 times
//  number 4 occurs 3 times

void main()
{
    List<int> list;
    list.Add(4);
    list.Add(2);
    list.Add(3);
    list.Add(4);
    list.Add(3);
    list.Add(4);
    Sort(list); // EqualRange needs a sorted sequence
    Pair<List<int>.ConstIterator, List<int>.ConstIterator> p1 =
        EqualRange(list.CBegin(), list.CEnd(), 1);
    Console.Out() << "number " << 1 << " occurs " << Distance(p1.first,
        p1.second) << " times" << endl();
```

```
    Pair<List<int>.ConstIterator, List<int>.ConstIterator> p2 =
        EqualRange(list.CBegin(), list.CEnd(), 2);
    Console.Out() << "number " << 2 << " occurs " << Distance(p2.first,
        p2.second) << " times" << endl();
    Pair<List<int>.ConstIterator, List<int>.ConstIterator> p3 =
        EqualRange(list.CBegin(), list.CEnd(), 3);
    Console.Out() << "number " << 3 << " occurs " << Distance(p3.first,
        p3.second) << " times" << endl();
    Pair<List<int>.ConstIterator, List<int>.ConstIterator> p4 =
        EqualRange(list.CBegin(), list.CEnd(), 4);
    Console.Out() << "number " << 4 << " occurs " << Distance(p4.first,
        p4.second) << " times" << endl();
}
```

Example

```
using System;

//  Writes:
//  sentence 'A bird in the hand is worth two in the bush.' contains 2 'a
    ' letters

void main()
{
    string proverb("A bird in the hand is worth two in the bush.");
    string letters = ToLower(proverb);
    Sort(letters.Begin(), letters.End());
    Pair<string.ConstIterator, string.ConstIterator> a = EqualRange(
        letters.CBegin(), letters.CEnd(), 'a');
    Console.Out() << "sentence '" << proverb << "' contains " << Distance
        (a.first, a.second) << " 'a' letters" << endl();
}
```

Implementation

algorithm.cm, page 5

1.3.65 EqualRange<I, T, R>(I, I, const T&, R) Function

Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence. Uses the given ordering relation to infer equality.

Syntax

`public Pair<I, I> EqualRange<I, T, R>(I first, I last, const T& value, R r);`

Constraint

where I is ForwardIterator and T is I.ValueType and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
first	I	A forward iterator that points to the beginning of a sorted sequence.
last	I	A forward iterator that points to one past the end of a sorted sequence.
value	const T&	A value.
r	R	An ordering relation.

Returns

Pair<I, I>

Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence.

Remarks

If the value is not found in the sorted sequence, returns a pair of iterators that form an empty range (that is: a range with two equal iterators.) The iterators point to the position where the given value would be if it were in the sorted sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  A(1) occurs 0 times
//  A(2) occurs 1 times
//  A(3) occurs 2 times
//  A(4) occurs 3 times

public class A
{
    public A(): id(0)
    {
    }
    public A(int id_): id(id_)
    {
```

```
    }
    public inline nothrow int Id() const
    {
        return id;
    }
    private int id;
}

public class ALess: Rel<A>
{
    public inline nothrow bool operator()(const A& left, const A& right)
    {
        return left.Id() < right.Id();
    }
}

void main()
{
    List<A> list;
    list.Add(A(4));
    list.Add(A(2));
    list.Add(A(3));
    list.Add(A(4));
    list.Add(A(3));
    list.Add(A(4));
    Sort(list, ALess()); // EqualRange needs a sorted sequence
    Pair<List<A>.ConstIterator, List<A>.ConstIterator> p1 = EqualRange(
        list.CBegin(), list.CEnd(), A(1), ALess());
    Console.Out() << "A(" << 1 << ") occurs " << Distance(p1.first, p1.
        second) << " times" << endl();
    Pair<List<A>.ConstIterator, List<A>.ConstIterator> p2 = EqualRange(
        list.CBegin(), list.CEnd(), A(2), ALess());
    Console.Out() << "A(" << 2 << ") occurs " << Distance(p2.first, p2.
        second) << " times" << endl();
    Pair<List<A>.ConstIterator, List<A>.ConstIterator> p3 = EqualRange(
        list.CBegin(), list.CEnd(), A(3), ALess());
    Console.Out() << "A(" << 3 << ") occurs " << Distance(p3.first, p3.
        second) << " times" << endl();
    Pair<List<A>.ConstIterator, List<A>.ConstIterator> p4 = EqualRange(
        list.CBegin(), list.CEnd(), A(4), ALess());
    Console.Out() << "A(" << 4 << ") occurs " << Distance(p4.first, p4.
        second) << " times" << endl();
}
```

Implementation

algorithm.cm, page 6

1.3.66 Factorial<U>(U) Function

Returns a factorial of the argument.

Syntax

`public nothrow U Factorial<U>(U n);`

Constraint

where U is UnsignedInteger;

Parameters

Name	Type	Description
n	U	An unsigned integer value.

Returns

U

Returns $n!$.

Example

```
using System;

//  Writes:
//  120

void main()
{
    uint x = 5u;
    uint f = Factorial(x);
    Console.Out() << f << endl();
}
```

Implementation

algorithm.cm, page 15

1.3.67 Find<I, P>(I, I, P) Function

Searches the first occurrence of a value from a sequence that matches a predicate.

Syntax

`public nothrow I Find<I, P>(I begin, I end, P p);`

Constraint

where I is InputIterator and P is UnaryPredicate and P.ArgumentType is I.ValueType;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
p	P	A unary predicate.

Returns

I

Returns an iterator pointing to the found value, or *end* if no value satisfied *p*.

Example

```
using System;
using System.Collections;

// Writes:
// index of first odd int is 3

public class Odd<I>: UnaryPred<I> where I is SignedInteger
{
    public nothrow inline bool operator()(I n)
    {
        return n % I(2) == I(1);
    }
}

void main()
{
    List<int> list;
    list.Add(2);
    list.Add(4);
    list.Add(6);
    list.Add(3);
    list.Add(8);
    list.Add(9);
    List<int>.ConstIterator p = Find(list.CBegin(), list.CEnd(), Odd<int
        >());
    if (p != list.CEnd())
    {
        Console.Out() << "index of first odd int is " << Distance(list.
            CBegin(), p) << endl();
    }
    else
    {
        Console.Out() << "no odd int found" << endl();
```

```
    }
}
```

Implementation

algorithm.cm, page 7

1.3.68 Find<I, T>(I, I, const T&) Function

Searches a value from a sequence.

Syntax

```
public nothrow I Find<I, T>(I begin, I end, const T& value);
```

Constraint

where I is InputIterator and T is Semiregular and EqualityComparable<T, I.ValueType>;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
value	const T&	A value to search.

Returns

I

Returns an iterator pointing to the found value, or *end* if no equal value found.

Example

```
using System;
using System.Collections;

//  Writes:
//  A(2) found in index 3

public class A
{
    public A(): id(0)
```

```
    {
    }
    public A(int id_): id(id_)
    {
    }
    public nothrow inline int Id() const
    {
        return id;
    }
    private int id;
}

public nothrow inline bool operator==(const A& left, const A& right)
{
    return left.Id() == right.Id();
}

void main()
{
    List<A> list;
    list.Add(A(3));
    list.Add(A(1));
    list.Add(A(4));
    list.Add(A(2));
    list.Add(A(3));
    list.Add(A(5));
    list.Add(A(2));
    List<A>.ConstIterator i = Find(list.CBegin(), list.CEnd(), A(2));
    if (i != list.CEnd())
    {
        Console.Out() << "A(2) found in index " << Distance(list.CBegin()
            , i) << endl();
    }
    else
    {
        Console.Out() << "A(2) not found" << endl();
    }
}
```

Implementation

algorithm.cm, page 6

1.3.69 ForEach<I, F>(I, I, F) Function

Applies a function object for each element of a sequence.

Syntax

```
public F ForEach<I, F>(I begin, I end, F f);
```

Constraint

where I is InputIterator and F is UnaryFunction and F.ArgumentType is I.ValueType;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a sequence.
end	I	An input iterator pointing one past the end of a sequence.
f	F	A unary function object.

Returns

F

Returns the function object.

Example

```
using System;
using System.Collections;

//  Writes:
//  Triangle.Draw()
//  Circle.Draw()
//  Rectangle.Draw()
//  ~Triangle()
//  ~Circle()
//  ~Rectangle()

public abstract class Figure
{
    public virtual ~Figure()
    {
    }
    public abstract void Draw();
}

public typedef SharedPtr<Figure> FigurePtr;

public class Circle: Figure
{
    public override ~Circle()
    {
        Console.WriteLine("~Circle()");
    }
```

```
    public override void Draw()
    {
        Console.Out() << "Circle.Draw()" << endl();
    }
}

public class Rectangle: Figure
{
    public override ~Rectangle()
    {
        Console.WriteLine("~Rectangle()");
    }
    public override void Draw()
    {
        Console.Out() << "Rectangle.Draw()" << endl();
    }
}

public class Triangle: Figure
{
    public override ~Triangle()
    {
        Console.WriteLine("~Triangle()");
    }
    public override void Draw()
    {
        Console.Out() << "Triangle.Draw()" << endl();
    }
}

public class Draw: UnaryFun<FigurePtr, void>
{
    public void operator()(FigurePtr figure) const
    {
        figure->Draw();
    }
}

public void main()
{
    List<FigurePtr> figures;
    figures.Add(FigurePtr(new Triangle()));
    figures.Add(FigurePtr(new Circle()));
    figures.Add(FigurePtr(new Rectangle()));
    ForEach(figures.CBegin(), figures.CEnd(), Draw());
}
```

Implementation

algorithm.cm, page 8

1.3.70 FrontInserter<C>(C&) Function

Returns a FrontInsertIterator<C> for a front insert sequence.

Syntax

```
public nothrow FrontInsertIterator<C> FrontInserter<C>(C& c);
```

Constraint

where C is FrontInsertionSequence;

Parameters

Name	Type	Description
c	C&	A front insertion sequence.

Returns

FrontInsertIterator<C>

Returns a FrontInsertIterator<C> for a front insertion sequence.

Remarks

A FrontInsertIterator<C> is an output iterator that inserts elements to the front of a front insertion sequence.

Example

```
using System;
using System.Collections;

// Writes:
// list: 3, 2, 1

void main()
{
    Set<int> s;
    s.Insert(2);
    s.Insert(1);
    s.Insert(3);
//  Set is a sorted container, so it contains now 1, 2, 3..

    List<int> list; // list is a model of a front insertion sequence

//  Copy each element in the set to the front of list using
    FrontInsertIterator...
    Copy(s.CBegin(), s.CEnd(), FrontInserter(list));

//  ForwardContainers containing ints can be put to OutputStream...
```

```
    Console.Out() << "list: " << list << endl();
}
```

1.3.71 Gcd<T>(T, T) Function

Returns the greatest common divisor of two values.

Syntax

`public nothrow T Gcd<T>(T a, T b);`

Constraint

where T is EuclideanSemiring;

Parameters

Name	Type	Description
a	T	The first value.
b	T	The second value.

Returns

T

Returns $gcd(a, b)$.

Example

```
using System;

//  Writes:
//  4

void main()
{
    Console.WriteLine(Gcd(12, 8));
}
```

Implementation

algorithm.cm, page 15

1.3.72 HexChar(byte) Function

Returns hexadecimal character representation of a four-bit value.

Syntax

```
public inline nothrow char HexChar(byte nibble);
```

Parameters

Name	Type	Description
nibble	byte	A four bit value.

Returns

char

Returns hexadecimal character representation of a four-bit value.

1.3.73 IdentityElement<T>(Plus<T>) Function

Returns the identity element of addition, that is: T(0).

Syntax

```
public inline nothrow T IdentityElement<T>(Plus<T> __parameter0);
```

Constraint

where T is AdditiveMonoid;

Parameters

Name	Type	Description
__parameter0	Plus<T>	

Returns

T

Returns $T(0)$.

1.3.74 IdentityElement<T>(Multiplies<T>) Function

Returns the identity element of multiplication, that is T(1).

Syntax

```
public inline nothrow T IdentityElement<T>(Multiplies<T> __parameter0);
```

Constraint

where T is MultiplicativeMonoid;

Parameters

Name	Type	Description
__parameter0	Multiplies<T>	

Returns

T

Returns $T(1)$.

1.3.75 Inserter<C, I>(C&, I) Function

Returns an InsertIterator<C> for an insertion sequence and its iterator.

Syntax

```
public nothrow InsertIterator<C> Inserter<C, I>(C& c, I i);
```

Constraint

where C is InsertionSequence and I is C.Iterator;

Parameters

Name	Type	Description
c	C&	An insertion sequence.
i	I	An iterator pointing a to position to insert elements.

Returns

InsertIterator<C>

Returns an InsertIterator<C> for an insertion sequence and iterator.

Remarks

An InsertIterator<C> is an output iterator that inserts elements to some position of an insertion sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  0, 1, 2, 3, 4

void main()
{
    Set<int> s;
    s.Insert(3);
```

```
    s.Insert(2);
    s.Insert(1);
//  Set is a sorted container, so it contains now 1, 2, 3...

    List<int> list;
    list.Add(0);
    list.Add(4);

//  Copy the set in the middle of the list using InsertIterator...
    Copy(s.CBegin(), s.CEnd(), Inserter(list, list.Begin() + 1));

//  ForwardContainers containing ints can be put to OutputStream...
    Console.Out() << list << endl();
}
```

1.3.76 InsertionSort<I>(I, I) Function

Sorts a sequence of values using insertion sort algorithm.

Syntax

```
public inline void InsertionSort<I>(I begin, I end);
```

Constraint

where I is RandomAccessIterator and I.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
begin	I	A random access iterator pointing to the beginning of a sequence.
end	I	A random access iterator pointing one past the end of a sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  -1, 0, 4, 4, 6, 7

void main()
{
    List<int> list;
    list.Add(7);
```

```
    list.Add(-1);
    list.Add(4);
    list.Add(6);
    list.Add(0);
    list.Add(4);
    InsertionSort(list.Begin(), list.End());
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 11

1.3.77 InsertionSort<I, R>(I, I, R) Function

Sorts a sequence of values using insertion sort algorithm and given ordering relation.

Syntax

```
public void InsertionSort<I, R>(I begin, I end, R r);
```

Constraint

where I is RandomAccessIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
begin	I	A random access iterator pointing to the beginning of a sequence.
end	I	A random access iterator pointing one past the end of a sequence.
r	R	An ordering relation.

Example

```
using System;
using System.Collections;

//  Writes:
//  7, 6, 4, 4, 0, -1

void main()
{
    List<int> list;
```

```
    list.Add(7);
    list.Add(-1);
    list.Add(4);
    list.Add(6);
    list.Add(0);
    list.Add(4);
    InsertionSort(list.Begin(), list.End(), Greater<int>());
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 11

1.3.78 IsAlpha(char) Function

Returns true if the given character is an alphabetic character, false otherwise.

Syntax

`public inline nothrow bool IsAlpha(char c);`

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is an alphabetic character, false otherwise.

Remarks

Alphabetic characters are the lower case and upper case letters 'a' .. 'z' and 'A' ..'Z'.

1.3.79 IsAlphanumeric(char) Function

Returns true if the given character is an alphanumeric character, false otherwise.

Syntax

`public inline nothrow bool IsAlphanumeric(char c);`

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is an alphanumeric character, false otherwise.

Remarks

Alphanumeric characters are the lower case and upper case letters and decimal digits 'a'..'z', 'A'..'Z' and '0'..'9'.

1.3.80 IsControl(char) Function

Returns true if the given character is a control character, false otherwise.

Syntax

```
public inline nothrow bool IsControl(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a control character, false otherwise.

Remarks

The control characters are characters whose ASCII codes are 0 .. 31 and 127.

1.3.81 IsDigit(char) Function

Returns true if the given character is a decimal digit, false otherwise.

Syntax

```
public inline nothrow bool IsDigit(char c);
```

Parameters

Name	Type	Description

c	char	A character to test.

Returns

bool

Returns true if the given character is a decimal digit, false otherwise.

Remarks

Decimal digits are '0' .. '9'.

1.3.82 IsGraphic(char) Function

Returns true if the given character is a graphical character, false otherwise.

Syntax

```
public inline nothrow bool IsGraphic(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a graphical character, false otherwise.

Remarks

The graphical characters are characters whose ASCII codes are in the range 33 .. 126.

1.3.83 IsHexDigit(char) Function

Returns true if the given character is a hexadecimal digit, false otherwise.

Syntax

```
public inline nothrow bool IsHexDigit(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a hexadecimal digit, false otherwise.

Remarks

Hexadecimal digits are '0' .. '9', 'a' .. 'f' and 'A' ..'F'.

1.3.84 IsLower(char) Function

Returns true if the given character is a lower case letter, false otherwise.

Syntax

```
public inline nothrow bool IsLower(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a lower case letter, false otherwise.

Remarks

Lower case letters are 'a' .. 'z'.

1.3.85 IsPrintable(char) Function

Returns true if the given character is a printable character, false otherwise.

Syntax

```
public inline nothrow bool IsPrintable(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a printable character, false otherwise.

Remarks

Printable characters are characters whose ASCII codes are in the range 32 .. 126.

1.3.86 IsPunctuation(char) Function

Returns true if the given character is a punctuation character, false otherwise.

Syntax

```
public inline nothrow bool IsPunctuation(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a punctuation character, false otherwise.

Remarks

Punctuation characters are characters whose ASCII codes are in ranges 33 .. 47, 58 .. 64, 91 .. 96 and 123 .. 126.

1.3.87 IsSpace(char) Function

Returns true if the given character is a space character, false otherwise.

Syntax

```
public inline nothrow bool IsSpace(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is a space character, false otherwise.

Remarks

The space characters are characters whose ASCII codes are 9 .. 13 and 32.

1.3.88 IsUpper(char) Function

Returns true if the given character is an upper case letter, false otherwise.

Syntax

```
public inline nothrow bool IsUpper(char c);
```

Parameters

Name	Type	Description
c	char	A character to test.

Returns

bool

Returns true if the given character is an upper case letter, false otherwise.

Remarks

Upper case letters are 'A' .. 'Z'.

1.3.89 LexicographicalCompare<I1, I2>(I1, I1, I2, I2) Function

Returns true if the first sequence comes lexicographically before the second sequence, false otherwise.

Syntax

```
public inline nothrow bool LexicographicalCompare<I1, I2>(I1 first1, I1 last1,
I2 first2, I2 last2);
```

Constraint

where I1 is InputIterator and I2 is InputIterator and LessThanComparable<I1.ValueType, I2.ValueType>;

Parameters

Name	Type	Description
first1	I1	An input iterator pointing to the beginning of the first sequence.
last1	I1	An input iterator pointing one past the end of the first sequence.

first2	I2	An input iterator pointing to the beginning of the second sequence.
last2	I2	An input iterator pointing one past the end of the second sequence.

Returns

bool

Returns true if the first sequence comes lexicographically before the second sequence, false otherwise.

Remarks

Compares the sequences element by element, until (1) an element in the first sequence is less than the corresponding element in the second sequence. In that case returns true. (2) an element in the second sequence is less than the corresponding element in the first sequence. In that case returns false. (3) The corresponding elements of the sequences are equal, but the first sequence is shorter than the second sequence. In that case returns true. (4) Otherwise returns false.

Example

```
using System;
using System.Collections;

//  Writes:
//  I thought so

void main()
{
    List<string> fruits1;
    fruits1.Add("apple");
    fruits1.Add("banana");
    fruits1.Add("grape");

    List<string> fruits2;
    fruits2.Add("apple");
    fruits2.Add("banana");
    fruits2.Add("grape");
    fruits2.Add("orange");

//  fruits1 comes lexicographically before fruits2, so...

    if (LexicographicalCompare(fruits1.CBegin(), fruits1.CEnd(), fruits2.
        CBegin(), fruits2.CEnd()))
    {
        Console.Out() << "I thought so" << endl();
    }
```

```
    if (LexicographicalCompare(fruits2.CBegin(), fruits2.CEnd(), fruits1.
        CBegin(), fruits1.CEnd()))
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

algorithm.cm, page 13

1.3.90 LexicographicalCompare<I1, I2, R>(I1, I1, I2, I2, R) Function

Returns true if the first sequence comes lexicographically before the second sequence according to the given ordering relation, false otherwise.

Syntax

`public nothrow bool LexicographicalCompare<I1, I2, R>(I1 first1, I1 last1, I2 first2, I2 last2, R r);`

Constraint

where I1 is InputIterator and I2 is InputIterator and Same<I1.ValueType, I2.ValueType> and Relation<R, I1.ValueType, I2.ValueType> and Relation<R, I2.ValueType, I1.ValueType>;

Parameters

Name	Type	Description
first1	I1	An input iterator pointing to the beginning of the first sequence.
last1	I1	An input iterator pointing one past the end of the first sequence.
first2	I2	An input iterator pointing to the beginning of the second sequence.
last2	I2	An input iterator pointing one past the end of the second sequence.

r	R	An ordering relation.

Returns

bool

Returns true if the first sequence comes lexicographically before the second sequence according to the given ordering relation, false otherwise.

Remarks

Compares the sequences element by element, until (1) an element in the first sequence is less than the corresponding element in the second sequence according to the given ordering relation. In that case returns true. (2) an element in the second sequence is less than the corresponding element in the first sequence according to the given ordering relation. In that case returns false. (3) The corresponding elements of the sequences are equal, but the first sequence is shorter than the second sequence. In that case returns true. (4) Otherwise returns false.

Example

```
using System;
using System.Collections;

//  Writes:
//  list comes lexicographically before set

public class A
{
    public A(): id(0)
    {
    }
    public A(int id_): id(id_)
    {
    }
    public nothrow inline int Id() const
    {
        return id;
    }
    private int id;
}

public class ALess: Rel<A>
{
    public nothrow inline bool operator()(const A& left, const A& right)
        const
    {
        return left.Id() < right.Id();
    }
}
```

```
void main()
{
    List<A> list;
    list.Add(A(3));
    list.Add(A(4));
    list.Add(A(5));

    Set<A, ALess> set;
    set.Insert(A(3));
    set.Insert(A(4));
    set.Insert(A(6));

    if (LexicographicalCompare(list.CBegin(), list.CEnd(), set.CBegin(),
        set.CEnd(), ALess()))
    {
        Console.Out() << "list comes lexicographically before set" <<
            endl();
    }
    else
    {
        Console.Out() << "bug" << endl();
    }
}
```

Implementation

algorithm.cm, page 13

1.3.91 LowerBound<I, T>(I, I, const T&) Function

Finds a position of the first element in a sorted sequence that is greater than or equal to the given value.

Syntax

```
public nothrow I LowerBound<I, T>(I first, I last, const T& value);
```

Constraint

where I is ForwardIterator and TotallyOrdered<T, I.ValueType>;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sorted sequence.

last	I	A forward iterator pointing one past the end of a sorted sequence.
value	const T&	A value to search.

Returns

I

Returns an iterator pointing to the first element in a sorted sequence that is greater than or equal to the given value, if there is one, otherwise returns *last*.

Remarks

Uses a binary search algorithm to search the position.

Example

```
using System;
using System.Collections;

//  Writes:
//  position of "Stroustrup" is 3

void main()
{
    List<string> persons;
    persons.Add("Stroustrup, Bjarne");
    persons.Add("Stepanov, Alexander");
    persons.Add("Knuth, Donald E.");
    persons.Add("Dijkstra, Edsger W.");
    persons.Add("Turing, Alan");
    Sort(persons);
    List<string>.ConstIterator s = LowerBound(persons.CBegin(), persons.
        CEnd(), string("Stroustrup"));
    if (s != persons.CEnd())
    {
        Console.Out() << "position of \"Stroustrup\" is " << Distance(
            persons.CBegin(), s) << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

algorithm.cm, page 4

1.3.92 LowerBound<I, T, R>(I, I, const T&, R) Function

Finds a position of the first element in a sorted sequence that is greater than or equal to the given value according to the given ordering relation.

Syntax

```
public nothrow I LowerBound<I, T, R>(I first, I last, const T& value, R r);
```

Constraint

where I is ForwardIterator and T is I.ValueType and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sorted sequence.
last	I	A forward iterator pointing one past the end of a sorted sequence.
value	const T&	A value to search.
r	R	An ordering relation.

Returns

I

Returns an iterator pointing to the first element in a sorted sequence that is greater than or equal to the given value according to the given ordering relation, if there is one, otherwise returns *last*.

Remarks

Uses a binary search algorithm to search the position.

Implementation

algorithm.cm, page 4

1.3.93 MakePair<T, U>(const T&, const U&) Function

Returns a pair composed of the given values.

Syntax

```
public ArgumentType MakePair<T, U>(const T& first, const U& second);
```

Constraint

where T is Semiregular and U is Semiregular;

Parameters

Name	Type	Description
first	const T&	The first value.
second	const U&	The second value.

Returns

ArgumentType

Returns a pair composed of the given values.

1.3.94 Max<T>(const T&, const T&) Function

Returns the maximum of two values.

Syntax

```
public inline nothrow const T& Max<T>(const T& left, const T& right);
```

Constraint

where T is LessThanComparable;

Parameters

Name	Type	Description
left	const T&	The first value.
right	const T&	The second value.

Returns

const T&

If $right \geq left$ returns $right$, else returns $left$.

1.3.95 MaxElement<I>(I, I) Function

Returns the position of the first occurrence of the largest element in a sequence of elements.

Syntax

`public nothrow I MaxElement<I>(I first, I last);`

Constraint

where I is ForwardIterator and I.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of sequence.
last	I	A forward iterator pointing one past the end of a sequence.

Returns

I

Returns the position of the first occurrence of the largest element in a sequence of elements.

Example

```
using System;
using System.Collections;

//   Writes:
//  maximum element is 2 at position 3

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(0);
    list.Add(1);
    list.Add(2);
    list.Add(0);
    list.Add(2);
    List<int>.ConstIterator maxPos = MaxElement(list.CBegin(), list.CEnd
        ());
    if (maxPos != list.CEnd())
    {
        Console.Out() << "maximum element is " << *maxPos << " at
            position " << Distance(list.CBegin(), maxPos) << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
```

```
    }
}
```

Implementation

algorithm.cm, page 14

1.3.96 MaxElement<I, R>(I, I, R) Function

Returns the position of the first occurrence of the largest element according to the given ordering relation in a sequence of elements.

Syntax

```
public nothrow I MaxElement<I, R>(I first, I last, R r);
```

Constraint

where I is ForwardIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of sequence.
last	I	A forward iterator pointing one past the end of a sequence.
r	R	An ordering relation.

Returns

I

Returns the position of the first occurrence of the largest element according to the given ordering relation in a sequence of elements.

Implementation

algorithm.cm, page 15

1.3.97 MaxValue<I>() Function

Returns the largest value of an integer type.

Syntax

```
public inline nothrow I MaxValue<I>();
```

Returns

I

Returns the largest value of an integer type.

1.3.98 MaxValue(byte) Function

Returns the maximum value of **byte**: 255.

Syntax

```
public inline nothrow byte MaxValue(byte __parameter0);
```

Parameters

Name	Type	Description
__parameter0	byte	

Returns

byte

Returns the maximum value of **byte**: 255.

1.3.99 MaxValue(int) Function

Returns the maximum value of **int**: 2147483647.

Syntax

```
public inline nothrow int MaxValue(int __parameter0);
```

Parameters

Name	Type	Description
__parameter0	int	

Returns

int

Returns the maximum value of **int**: 2147483647.

1.3.100 MaxValue(long) Function

Returns the maximum value of **long**: 9223372036854775807.

Syntax

```
public inline nothrow long MaxValue(long __parameter0);
```

Parameters

Name	Type	Description
__parameter0	long	

Returns

long

Returns the maximum value of **long**: 9223372036854775807.

1.3.101 MaxValue(sbyte) Function

Returns the maximum value of **sbyte**: 127.

Syntax

```
public inline nothrow sbyte MaxValue(sbyte __parameter0);
```

Parameters

Name	Type	Description
__parameter0	sbyte	

Returns

sbyte

Returns the maximum value of **sbyte**: 127.

1.3.102 MaxValue(short) Function

Returns the maximum value of **short**: 32767.

Syntax

```
public inline nothrow short MaxValue(short __parameter0);
```

Parameters

Name	Type	Description
__parameter0	short	

Returns

short

Returns the maximum value of **short**: 32767.

1.3.103 MaxValue(uint) Function

Returns the maximum value of **uint**: 4294967295.

Syntax

```
public inline nothrow uint MaxValue(uint __parameter0);
```

Parameters

Name	Type	Description
__parameter0	uint	

Returns

uint

Returns the maximum value of **uint**: 4294967295.

1.3.104 MaxValue(ulong) Function

Returns the maximum value of **ulong**: 18446744073709551615.

Syntax

```
public inline nothrow ulong MaxValue(ulong __parameter0);
```

Parameters

Name	Type	Description
__parameter0	ulong	

Returns

ulong

Returns the maximum value of **ulong**: 18446744073709551615.

1.3.105 MaxValue(ushort) Function

Returns the maximum value of **ushort**: 65535.

Syntax

```
public inline nothrow ushort MaxValue(ushort __parameter0);
```

Parameters

Name	Type	Description
__parameter0	ushort	

Returns

ushort

Returns the maximum value of **ushort**: 65535.

1.3.106 Median<T, R>(const T&, const T&, const T&, R) Function

Returns the median of three values according to the given ordering relation.

Syntax

```
public nothrow const T& Median<T, R>(const T& a, const T& b, const T& c, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.
r	R	An ordering relation.

Returns

const T&

Returns the median of three values according to the given ordering relation.

1.3.107 Median<T>(const T&, const T&, const T&) Function

Returns the median of three values.

Syntax

```
public nothrow const T& Median<T>(const T& a, const T& b, const T& c);
```

Constraint

where T is TotallyOrdered;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.

Returns

const T&

Returns the median of three values.

1.3.108 Min<T>(const T&, const T&) Function

Returns the minimum of two values.

Syntax

```
public inline nothrow const T& Min<T>(const T& left, const T& right);
```

Constraint

where T is LessThanComparable;

Parameters

Name	Type	Description
left	const T&	The first value.
right	const T&	The second value.

Returns

const T&

If $left \leq right$ returns $left$, else returns $right$.

1.3.109 MinElement<I>(I, I) Function

Returns the position of the first occurrence of the smallest element in a sequence of elements.

Syntax

`public nothrow I MinElement<I>(I first, I last);`

Constraint

where I is ForwardIterator and I.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sequence.
last	I	A forward iterator pointing one past the end of a sequence.

Returns

I

Returns the position of the first occurrence of the smallest element in a sequence of elements.

Example

```
using System;
using System.Collections;

//  Writes:
//  minimum element is 0 at position 1

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(0);
    list.Add(1);
    list.Add(2);
    list.Add(0);
    list.Add(2);
    List<int>.ConstIterator minPos = MinElement(list.CBegin(), list.CEnd
        ());
    if (minPos != list.CEnd())
    {
        Console.Out() << "minimum element is " << *minPos << " at
            position " << Distance(list.CBegin(), minPos) << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
```

```
    }
}
```

Implementation

algorithm.cm, page 13

1.3.110 MinElement<I, R>(I, I, R) Function

Returns the position of the first occurrence of the smallest element according to the given ordering relation in a sequence of elements.

Syntax

```
public nothrow I MinElement<I, R>(I first, I last, R r);
```

Constraint

where I is ForwardIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sequence.
last	I	A forward iteraor pointing one past the end of a sequence.
r	R	An ordering relation.

Returns

I

Returns the position of the first occurrence of the smallest element according to the given ordering relation in a sequence of elements.

Implementation

algorithm.cm, page 14

1.3.111 MinValue<I>() Function

Returns the smallest value of an integer type.

Syntax

```
public inline nothrow I MinValue<I>();
```

Returns

I

Returns the smallest value of an integer type.

1.3.112 MinValue(byte) Function

Returns the minimum value of **byte**: 0.

Syntax

```
public inline nothrow byte MinValue(byte __parameter0);
```

Parameters

Name	Type	Description
__parameter0	byte	

Returns

byte

Returns the minimum value of **byte**: 0.

1.3.113 MinValue(int) Function

Returns the minimum value of **int**: -2147483648.

Syntax

```
public inline nothrow int MinValue(int __parameter0);
```

Parameters

Name	Type	Description
__parameter0	int	

Returns

int

Returns the minimum value of **int**: -2147483648.

1.3.114 MinValue(long) Function

Returns the minimum value of **long**: -9223372036854775808.

Syntax

```
public inline nothrow long MinValue(long __parameter0);
```

Parameters

Name	Type	Description
__parameter0	long	

Returns

long

Returns the minimum value of **long**: -9223372036854775808.

1.3.115 MinValue(sbyte) Function

Returns the minimum value of **sbyte**: -128.

Syntax

```
public inline nothrow sbyte MinValue(sbyte __parameter0);
```

Parameters

Name	Type	Description
__parameter0	sbyte	

Returns

sbyte

Returns the minimum value of **sbyte**: -128.

1.3.116 MinValue(short) Function

Returns the minimum value of **short**: -32768.

Syntax

```
public inline nothrow short MinValue(short __parameter0);
```

Parameters

Name	Type	Description
__parameter0	short	

Returns

short

Returns the minimum value of **short**: -32768.

1.3.117 MinValue(uint) Function

Returns the minimum value of **uint**: 0.

Syntax

```
public inline nothrow uint MinValue(uint __parameter0);
```

Parameters

Name	Type	Description
__parameter0	uint	

Returns

uint

Returns the minimum value of **uint**: 0.

1.3.118 MinValue(ulong) Function

Returns the minimum value of **ulong**: 0.

Syntax

```
public inline nothrow ulong MinValue(ulong __parameter0);
```

Parameters

Name	Type	Description
__parameter0	ulong	

Returns

ulong

Returns the minimum value of **ulong**: 0.

1.3.119 MinValue(ushort) Function

Returns the minimum value of **ushort**: 0.

Syntax

```
public inline nothrow ushort MinValue(ushort __parameter0);
```

Parameters

Name	Type	Description
__parameter0	ushort	

Returns

ushort

Returns the minimum value of **ushort**: 0.

1.3.120 Move<I, O>(I, I, O) Function

Moves a sequence.

Syntax

```
public O Move<I, O>(I begin, I end, O to);
```

Constraint

where I is InputIterator and O is OutputIterator and O.ValueType is I.ValueType and I.ValueType is MoveAssignable;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of a source sequence.
end	I	An input iterator pointing one past the end of a source sequence.
to	O	An output iterator pointing to the beginning of the target sequence.

Returns

O

Returns an output iterator pointing one past the end of the target sequence.

Implementation

algorithm.cm, page 2

1.3.121 MoveBackward<I, O>(I, I, O) Function

Moves a source sequence to a target sequence starting from the end of the source sequence.

Syntax

```
public O MoveBackward<I, O>(I begin, I end, O to);
```

Constraint

where I is BidirectionalIterator and O is BidirectionalIterator and O.ValueType is I.ValueType and I.ValueType is MoveAssignable;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of the source sequence.
end	I	A bidirectional iterator pointing one pasth the end of the source sequence.
to	O	A bidirectional iterator pointing to the beginning of the target sequence.

Returns

O

Returns a bidirectional iterator pointing to the beginning of the target sequence.

Implementation

algorithm.cm, page 3

1.3.122 Next<I>(I, int) Function

Returns a forward iterator advanced the specified number of steps.

Syntax

```
public nothrow I Next<I>(I i, int n);
```

Constraint

where I is ForwardIterator;

Parameters

Name	Type	Description
i	I	A forward iterator.
n	int	A non-negative number of steps to advance.

Returns

I

Returns a forward iterator advanced the specified number of steps.

Implementation

algorithm.cm, page 3

1.3.123 Next<I>(I, int) Function

Returns a random access iterator advanced the specified offset.

Syntax

```
public inline nothrow I Next<I>(I i, int n);
```

Constraint

where I is RandomAccessIterator;

Parameters

Name	Type	Description
i	I	A random access iterator.
n	int	An offset to advance.

Returns

I

Returns $i + n$.

Implementation

algorithm.cm, page 4

1.3.124 NextPermutation<I>(I, I) Function

Computes the lexicographically next permutation of a sequence of elements.

Syntax

```
public nothrow bool NextPermutation<I>(I begin, I end);
```

Constraint

where I is BidirectionalIterator and I.ValueType is LessThanComparable;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of a sequence.
end	I	A bidirectional iterator pointing one past the end of a sequence.

Returns

bool

Returns true if the permutation was not last permutation, false otherwise. If the permutation was last, the permutation returned is the lexicographically first permutation of the sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  1, 2, 3
//  1, 3, 2
//  2, 1, 3
//  2, 3, 1
//  3, 1, 2
//  3, 2, 1

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(2);
    list.Add(3);
    Console.Out() << list << endl();
```

```
    while (NextPermutation(list.Begin(), list.End()))
    {
        Console.Out() << list << endl();
    }
}
```

Implementation

algorithm.cm, page 16

1.3.125 NextPermutation<I, R>(I, I, R) Function

Computes the lexicographically next permutation of a sequence of elements according to the given ordering relation.

Syntax

```
public nothrow bool NextPermutation<I, R>(I begin, I end, R r);
```

Constraint

where I is BidirectionalIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of a sequence.
end	I	A bidirectional iterator pointing one past the end of a sequence.
r	R	An ordering relation.

Returns

bool

Returns true if the permutation was not last permutation, false otherwise. If the permutation was last, the permutation returned is the lexicographically first permutation of the sequence.

Implementation

algorithm.cm, page 16

1.3.126 Now() Function

Returns current time point value from computer's real time clock.

Syntax

```
public TimePoint Now();
```

Returns

TimePoint

Returns number of nanoseconds elapsed since 1.1.1970 as a time point value.

1.3.127 ParseBool(const string&) Function

Parses a Boolean value "true" or "false" from the given string and returns it.

Syntax

```
public bool ParseBool(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

bool

Returns true if *s* contains "true", false if *s* contains "false". Otherwise throws ConversionException.

1.3.128 ParseBool(const string&, bool&) Function

Parses a Boolean value "true" or "false" from the given string and returns true if the parsing was successful, false if not.

Syntax

```
public bool ParseBool(const string& s, bool& b);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

b	bool&	If *s* contains "true" b is set to true, if *s* contains "false" b is set to false.

Returns

bool

Returns true if the parsing was successful, false if not.

1.3.129 ParseDouble(const string&) Function

Parses a **double** value from the given string and returns it.

Syntax

```
public double ParseDouble(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

double

Returns the parsed **double** value if the parsing was successful. Otherwise throws ConversionException.

1.3.130 ParseDouble(const string&, double&) Function

Parses a **double** value from the given string and returns true if the parsing was successful, false if not.

Syntax

```
public bool ParseDouble(const string& s, double& x);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
x	double&	A **double** parsed from *s*.

Returns

bool

Returns true if the parsing was successful, false if not.

1.3.131 ParseHex(const string&) Function

Parses a hexadecimal value from a string.

Syntax

```
public ulong ParseHex(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

ulong

Returns the parsed hexadecimal value if the parsing was successful, otherwise throws ConversionException.

1.3.132 ParseHex(const string&, ulong&) Function

Parses a hexadecimal value from a string and returns true if the parsing was successful.

Syntax

```
public bool ParseHex(const string& s, ulong& hex);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
hex	ulong&	Parsed value.

Returns

bool

Returns true, if the parsing was successful, false otherwise.

1.3.133 ParseHex(const string&, uhuge&) Function

Parses a hexadecimal 128-bit value from a string and returns true, if the parsing was successful.

Syntax

```
public bool ParseHex(const string& s, uhuge& hex);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
hex	uhuge&	Parsed 128-bit value.

Returns

bool

Returns true, if the parsing was successful, false otherwise.

1.3.134 ParseHexUHuge(const string&) Function

Parses a 128-bit decimal value from a string.

Syntax

```
public uhuge ParseHexUHuge(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

uhuge

Returns the parsed 128-bit decimal value if the parsing was successful, otherwise throws ConversionException.

1.3.135 ParseInt(const string&) Function

Parses an **int** from the given string and returns it.

Syntax

```
public int ParseInt(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

int

Returns the parsed **int** value if the parsing was successful. Otherwise throws ConversionException.

1.3.136 ParseInt(const string&, int&) Function

Parses an **int** value from the given string and returns true if the parsing was successful, false if not.

Syntax

```
public bool ParseInt(const string& s, int& x);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
x	int&	An **int** parsed from *s*.

Returns

bool

Returns true if the parsing was successful, false if not.

1.3.137 ParseUHuge(const string&) Function

Parses a decimal 128-bit value from a string.

Syntax

```
public uhuge ParseUHuge(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

uhuge

Returns parsed 128-bit decimal value if parsing was successful, otherwise throws ConversionException.

1.3.138 ParseUHuge(const string&, uhuge&) Function

Parses a decimal 128-bit value from a string and returns true, if the parsing was successful.

Syntax

```
public bool ParseUHuge(const string& s, uhuge& x);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
x	uhuge&	Parsed 128-bit value.

Returns

bool

Returns true, if the parsing was successful, false otherwise.

1.3.139 ParseUInt(const string&) Function

Parses an **uint** from the given string and returns it.

Syntax

```
public uint ParseUInt(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

uint

Returns the parsed **uint** value if the parsing was successful. Otherwise throws ConversionException.

1.3.140 ParseUInt(const string&, uint&) Function

Parses an **uint** value from the given string and returns true if the parsing was successful, false if not.

Syntax

```
public bool ParseUInt(const string& s, uint& x);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
x	uint&	An **uint** parsed from *s*.

Returns

bool

Returns true if the parsing was successful, false if not.

1.3.141 ParseULong(const string&) Function

Parses an **ulong** from the given string and returns it.

Syntax

```
public ulong ParseULong(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.

Returns

ulong

Returns the parsed **ulong** value if the parsing was successful. Otherwise throws ConversionException.

1.3.142 ParseULong(const string&, ulong&) Function

Parses an **ulong** value from the given string and returns true if the parsing was successful, false if not.

Syntax

```
public bool ParseULong(const string& s, ulong& x);
```

Parameters

Name	Type	Description
s	const string&	A string to parse.
x	ulong&	An **ulong** parsed from *s*.

Returns

bool

Returns true if the parsing was successful, false if not.

1.3.143 PrevPermutation<I>(I, I) Function

Computes the lexicographically previous permutation of a sequence of elements.

Syntax

```
public nothrow bool PrevPermutation<I>(I begin, I end);
```

Constraint

where I is BidirectionalIterator and I.ValueType is LessThanComparable;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of a sequence.
end	I	A bidirectional iterator pointing one past the end of a sequence.

Returns

bool

Returns true if the permutation was not first permutation, false otherwise. If the permutation was first, the permutation returned is the lexicographically last permutation of the sequence.

1.3.144 PrevPermutation<I, R>(I, I, R) Function

Computes the lexicographically previous permutation according to the given ordering relation of a sequence of elements.

Syntax

```
public nothrow bool PrevPermutation<I, R>(I begin, I end, R r);
```

Constraint

where I is BidirectionalIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of a sequence.
end	I	A bidirectional iterator pointing one past the end of a sequence.
r	R	An ordering relation.

Returns

bool

Returns true if the permutation was not first permutation, false otherwise. If the permutation was first, the permutation returned is the lexicographically last permutation of the sequence.

1.3.145 PtrCast<U, T>(const SharedPtr<T>&) Function

Casts a shared pointer.

Syntax

```
public nothrow SharedPtr<U> PtrCast<U, T>(const SharedPtr<T>& from);
```

Parameters

Name	Type	Description
from	const SharedPtr<T>&	A shared pointer to cast from.

Returns

SharedPtr<U>

Returns the shared pointer casted to $SharedPtr < U >$.

Example

```
using System;

public class Base
{
    public virtual ~Base()
    {
    }
}

public class Derived: Base
{
}

void main()
{
    SharedPtr<Base> ptr(new Derived());
    SharedPtr<Derived> derived = PtrCast<Derived>(ptr);
}
```

1.3.146 Reverse<I>(I, I) Function

Reverses a sequence.

Syntax

```
public nothrow void Reverse<I>(I begin, I end);
```

Constraint

where I is BidirectionalIterator;

Parameters

Name	Type	Description
begin	I	A bidirectional iterator pointing to the beginning of a sequence.
end	I	A bidirectional iterator pointing one past the end of a sequence.

Implementation

algorithm.cm, page 2

1.3.147 Reverse<I>(I, I) Function

Reverses a sequence.

Syntax

```
public nothrow void Reverse<I>(I begin, I end);
```

Constraint

where I is RandomAccessIterator;

Parameters

Name	Type	Description
begin	I	A random access iterator pointing to the beginning of a sequence.
end	I	A random access iterator pointing one past the end of a sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  3, 2, 1

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(2);
    list.Add(3);
    Reverse(list.Begin(), list.End());
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 1

1.3.148 Rvalue<T>(T&&) Function

Converts an argument to an rvalue so that it can be moved.

Syntax

`public inline nothrow T&& Rvalue<T>(T&& x);`

Parameters

Name	Type	Description
x	T&&	A argument to convert.

Returns

T&&

Returns an rvalue reference of the argument.

Example

```
//  Writes:
//  10

using System;
```

```
using System.Collections;
using System.IO;

public class A
{
    public A(): ptr(null) {}
    public A(int x): ptr(new int(x)) {}
    public ~A()
    {
        delete ptr;
    }
    suppress A(const A&);
    suppress void operator=(const A&);
    public A(A&& that): ptr(that.ptr)
    {
        that.ptr = null;
    }
    public void operator=(A&& that)
    {
        Swap(ptr, that.ptr);
    }
    public int Value() const
    {
        #assert(ptr != null);
        return *ptr;
    }
    private int* ptr;
}

OutputStream& operator<<(OutputStream& s, const A& a)
{
    return s << a.Value() << endl();
}

void main()
{
    List<A> alist;
    A a(10);
    alist.Add(Rvalue(a));
    for (const A& a : alist)
    {
        Console.Out() << a << endl();
    }
}
```

1.3.149 Select_0_2<T, R>(const T&, const T&, R) Function

Returns the smaller of two values according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_0_2<T, R>(const T& a, const T& b, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
r	R	An ordering relation.

Returns

const T&

Returns the smaller of two values according to the given ordering relation.

1.3.150 Select_0_3<T, R>(const T&, const T&, const T&, R) Function

Returns the smallest of tree values according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_0_3<T, R>(const T& a, const T& b, const
T& c, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.
r	R	An ordering relation.

Returns

const T&

Returns the smallest of tree values according to the given ordering relation.

1.3.151 Select_1_2<T, R>(const T&, const T&, R) Function

Returns the larger of two values according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_1_2<T, R>(const T& a, const T& b, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
r	R	An ordering relation.

Returns

const T&

Returns the larger of two values according to the given ordering relation.

1.3.152 Select_1_3<T, R>(const T&, const T&, const T&, R) Function

Returns the median of three values according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_1_3<T, R>(const T& a, const T& b, const
T& c, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.
r	R	An ordering relation.

Returns

const T&

Returns the median of three values according to the given ordering relation.

1.3.153 Select_1_3_ab<T, R>(const T&, const T&, const T&, R) Function

Returns the median of three values when the first two are in increasing order according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_1_3_ab<T, R>(const T& a, const T& b, const
T& c, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.
r	R	An ordering relation.

Returns

const T&

Returns the median of three values when the first two are in increasing order according to the given ordering relation.

1.3.154 Select_2_3<T, R>(const T&, const T&, const T&, R) Function

Returns the largest of three values according to the given ordering relation.

Syntax

```
public inline nothrow const T& Select_2_3<T, R>(const T& a, const T& b, const
T& c, R r);
```

Constraint

where T is Semiregular and R is Relation and R.Domain is T;

Parameters

Name	Type	Description
a	const T&	The first value.
b	const T&	The second value.
c	const T&	The third value.
r	R	An ordering relation.

Returns

const T&

Returns the largest of three values according to the given ordering relation.

1.3.155 Sort<C>(C&) Function

Sorts the elements of a forward container to increasing order.

Syntax

```
public void Sort<C>(C& c);
```

Constraint

where C is ForwardContainer and C.Iterator.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
c	C&	A forward container,

Remarks

First the elements from the forward container are copied to a List<T>, then the List<T> is sorted, and finally the elements from the List<T> are copied back to the forward container.

Implementation

algorithm.cm, page 12

1.3.156 Sort<C>(C&) Function

Sorts the elements of a random access container to increasing order.

Syntax

```
public inline void Sort<C>(C& c);
```

Constraint

where C is RandomAccessContainer and C.Iterator.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
c	C&	A random access container.

Example

```
using System;
using System.Collections;

// Writes:
// 41, 6334, 11478, 15724, 18467, 19169, 24464, 26500, 26962, 29358

void main()
{
    List<int> list;
    for (int i = 0; i < 10; ++i)
    {
        list.Add(rand());
    }
    Sort(list);
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 12

1.3.157 Sort<C, R>(C&, R) Function

Sorts the elements of a forward container to order according to the given ordering relation.

Syntax

```
public void Sort<C, R>(C& c, R r);
```

Constraint

where C is ForwardContainer and R is Relation and R.Domain is C.Iterator.ValueType;

Parameters

Name	Type	Description
c	C&	A forward container.
r	R	An ordering relation.

Remarks

First the elements from the forward container are copied to a List<T>, then the List<T> is sorted, and finally the elements from the List<T> are copied back to the forward container.

1.3.158 Sort<C, R>(C&, R) Function

Sorts the elements of a random access container to order according to the given ordering relation.

Syntax

```
public inline void Sort<C, R>(C& c, R r);
```

Constraint

where C is RandomAccessContainer and R is Relation and R.Domain is C.Iterator.ValueType;

Parameters

Name	Type	Description
c	C&	A random access container.
r	R	An ordering relation.

Example

```
using System;
using System.Collections;

//  Writes:
//  29358, 26962, 26500, 24464, 19169, 18467, 15724, 11478, 6334, 41

void main()
{
    List<int> list;
    for (int i = 0; i < 10; ++i)
    {
        list.Add(rand());
    }
    Sort(list, Greater<int>());
    Console.Out() << list << endl();
}
```

Implementation

algorithm.cm, page 11

1.3.159 Sort<I>(I, I) Function

Sorts the elements of a sequence to increasing order.

Syntax

```
public inline void Sort<I>(I begin, I end);
```

Constraint

where I is RandomAccessIterator and I.ValueType is TotallyOrdered;

Parameters

Name	Type	Description
begin	I	A random access iterator pointing to the beginning of a sequence.
end	I	A random access iterator pointing one past the end of a sequence.

Implementation

algorithm.cm, page 12

1.3.160 Sort<I, R>(I, I, R) Function

Sorts the elements of a sequence to order according to the given ordering relation.

Syntax

```
public void Sort<I, R>(I begin, I end, R r);
```

Constraint

where I is RandomAccessIterator and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
begin	I	A random access iterator pointing to the beginning of a sequence.
end	I	A random access iterator pointing one past the end of a sequence.
r	R	An ordering relation.

Implementation

algorithm.cm, page 11

1.3.161 Swap<T>(T&, T&) Function

Exchanges two values.

Syntax

```
public inline nothrow void Swap<T>(T& left, T& right);
```

Constraint

where T is MoveConstructible and T is MoveAssignable and T is Destructible;

Parameters

Name	Type	Description
left	T&	The first value.
right	T&	The second value.

Remarks

The values are converted to rvalues using Rvalue<T>(T&&) function and exchanged by moving them.

Implementation

algorithm.cm, page 1

1.3.162 ToHexString<U>(U) Function

Converts an unsigned integer value to hexadecimal string representation.

Syntax

```
public nothrow string ToHexString<U>(U x);
```

Constraint

where U is UnsignedInteger and ExplicitlyConvertible<U, byte>;

Parameters

Name	Type	Description
x	U	An unsigned integer value.

Returns

string

Returns x converted to hexadecimal string representation.

Implementation

convert.cm, page 4

1.3.163 ToHexString(byte) Function

Converts a **byte** to hexadecimal string representation.

Syntax

```
public nothrow string ToHexString(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte**.

Returns

string

Returns *b* converted to hexadecimal string representation.

Implementation

convert.cm, page 4

1.3.164 ToHexString(uhuge) Function

Returns a 128-bit value converted to hexadecimal representation.

Syntax

```
public nothrow string ToHexString(uhuge x);
```

Parameters

Name	Type	Description
x	uhuge	A 128-bit value to convert.

Returns

string

A hexadecimal string.

1.3.165 ToHexString(uint) Function

Converts an **uint** to hexadecimal string representation.

Syntax

```
public nothrow string ToHexString(uint u);
```

Parameters

Name	Type	Description
u	uint	An **uint**.

Returns

string

Returns u converted to hexadecimal string representation.

Implementation

convert.cm, page 4

1.3.166 ToHexString(ulong) Function

Converts an **ulong** to hexadecimal string representation.

Syntax

```
public nothrow string ToHexString(ulong u);
```

Parameters

Name	Type	Description
u	ulong	An **ulong**.

Returns

string

Returns u converted to hexadecimal string representation.

Implementation

convert.cm, page 4

1.3.167 ToHexString(ushort) Function

Converts an **ushort** to hexadecimal string representation.

Syntax

```
public nothrow string ToHexString(ushort u);
```

Parameters

Name	Type	Description
u	ushort	An **ushort**.

Returns

string

Returns *u* converted to hexadecimal string representation.

Implementation

convert.cm, page 4

1.3.168 ToLower(const string&) Function

Converts a string to lower case.

Syntax

```
public nothrow string ToLower(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string.

Returns

string

Returns *s* converted to lower case.

Implementation

string.cm, page 9

1.3.169 ToString<I, U>(I) Function

Converts a signed integer value to string representation.

Syntax

```
public nothrow string ToString<I, U>(I x);
```

Constraint

where I is SignedInteger and U is UnsignedInteger and ExplicitlyConvertible<I, U> and ExplicitlyConvertible<U, byte>;

Parameters

Name	Type	Description
x	I	A signed integer value.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 1

1.3.170 ToString<U>(U) Function

Converts an unsigned integer value to string representation.

Syntax

```
public nothrow string ToString<U>(U x);
```

Constraint

where U is UnsignedInteger and ExplicitlyConvertible<U, byte>;

Parameters

Name	Type	Description
x	U	An unsigned integer value.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.171 ToString(bool) Function

Converts a Boolean value to string representation.

Syntax

```
public nothrow string ToString(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value.

Returns

string

If *b* returns "true" else returns "false".

Implementation

convert.cm, page 3

1.3.172 ToString(byte) Function

Converts a **byte** to string representation.

Syntax

```
public nothrow string ToString(byte x);
```

Parameters

Name	Type	Description
x	byte	A **byte**.

Returns

string

Returns a **byte** converted to string representation.

Implementation

convert.cm, page 2

1.3.173 ToString(char) Function

Converts a character to string representation.

Syntax

```
public nothrow string ToString(char c);
```

Parameters

Name	Type	Description
c	char	A character.

Returns

string

Returns *c* converted to string representation.

Implementation

convert.cm, page 3

1.3.174 ToString(double) Function

Converts a **double** to string representation.

Syntax

```
public nothrow string ToString(double x);
```

Parameters

Name	Type	Description
x	double	A **double**.

Returns

string

Returns x converted to string representation.

Remarks

The maximum number of decimal places in the conversion is 15.

Implementation

convert.cm, page 3

1.3.175 ToString(double, int) Function

Converts a **double** to string representation using the given maximum number of decimal places.

Syntax

```
public nothrow string ToString(double x, int maxNumDecimals);
```

Parameters

Name	Type	Description
x	double	A **double**.
maxNumDecimals	int	Maximum number of decimal digits.

Returns

string

Returns x converted to string representation using the given maximum number of decimal places.

Implementation

convert.cm, page 3

1.3.176 ToString(int) Function

Converts an **int** to string representation.

Syntax

```
public nothrow string ToString(int x);
```

Parameters

Name	Type	Description
x	int	An **int**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.177 ToString(long) Function

Converts a **long** to string representation.

Syntax

```
public nothrow string ToString(long x);
```

Parameters

Name	Type	Description
x	long	A **long**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.178 ToString(uhuge) Function

Converts a 128-bit value to string representation.

Syntax

```
public string ToString(uhuge x);
```

Parameters

Name	Type	Description
x	uhuge	A 128-bit value.

Returns

string

Returns x converted to string.

1.3.179 ToString(sbyte) Function

Converts an **sbyte** to string representation.

Syntax

```
public nothrow string ToString(sbyte x);
```

Parameters

Name	Type	Description
x	sbyte	An **sbyte**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.180 ToString(short) Function

Converts a **short** to string representation.

Syntax

```
public nothrow string ToString(short x);
```

Parameters

Name	Type	Description
x	short	An **short**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.181 ToString(uint) Function

Converts an **uint** to string representation.

Syntax

```
public nothrow string ToString(uint x);
```

Parameters

Name	Type	Description
x	uint	An **uint**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.182 ToString(ulong) Function

Converts an **ulong** to string representation.

Syntax

```
public nothrow string ToString(ulong x);
```

Parameters

Name	Type	Description
x	ulong	An **ulong**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.183 ToString(ushort) Function

Converts an **ushort** to string representation.

Syntax

```
public nothrow string ToString(ushort x);
```

Parameters

Name	Type	Description
x	ushort	An **ushort**.

Returns

string

Returns x converted to string representation.

Implementation

convert.cm, page 2

1.3.184 ToUpper(const string&) Function

Converts a string to upper case.

Syntax

```
public nothrow string ToUpper(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string.

Returns

string

Returns *s* converted to upper case.

Implementation

string.cm, page 10

1.3.185 Transform<I, O, F>(I, I, O, F) Function

Transforms an input sequence to an output sequence using a unary function.

Syntax

```
public O Transform<I, O, F>(I begin, I end, O to, F fun);
```

Constraint

where I is InputIterator and O is OutputIterator and F is UnaryFunction and F.ArgumentType is I.ValueType and CopyAssignable<O.ValueType, F.ResultType>;

Parameters

Name	Type	Description
begin	I	An input iterator pointing to the beginning of an input sequence.
end	I	An input iterator pointing one past the end of an input sequence.
to	O	An output iterator pointing to the beginning of an output sequence.
fun	F	A unary function object.

Returns

O

Returns an output iterator pointing one past the end of the output sequence.

Example

```
using System;
using System.Collections;

// Writes:
// 2, 4, 6

public class Double<A>: UnaryFun<A, A> where A is AdditiveSemigroup
{
    public A operator()(const A& x) const
    {
        return x + x;
    }
}

void main()
{
    List<int> list;
    list.Add(1);
    list.Add(2);
    list.Add(3);
    List<int> doubledList;
    Transform(list.CBegin(), list.CEnd(), BackInserter(doubledList),
        Double<int>());
    Console.Out() << doubledList << endl();
}
```

Implementation

algorithm.cm, page 8

1.3.186 Transform<I1, I2, O, F>(I1, I1, I2, O, F) Function

Transforms two input sequences to an ouput sequence using a binary function.

Syntax

```
public O Transform<I1, I2, O, F>(I1 begin1, I1 end1, I2 begin2, O to, F fun);
```

Constraint

where I1 is InputIterator and I2 is InputIterator and O is OutputIterator and F is BinaryFunction and F.FirstArgumentType is I1.ValueType and F.SecondArgumentType is I2.ValueType and CopyAssignable<O.ValueType, F.ResultType>;

Parameters

Name	Type	Description
begin1	I1	An input iterator pointing to the beginning of the first input sequence.
end1	I1	An input iterator pointing one past the end of the first input sequence.
begin2	I2	An input iterator pointing to the beginning of the second input sequence.
to	O	An output iterator pointing to the beginning of the output sequence.
fun	F	A binary function object.

Returns

O

Returns an output iterator pointing one past the end of the output sequence.

Example

```
using System;
using System.Collections;

//  Writes:
//  4, 6, 8

void main()
{
    List<int> list1;
    list1.Add(1);
    list1.Add(2);
    list1.Add(3);
    List<int> list2;
    list2.Add(3);
    list2.Add(4);
    list2.Add(5);
    List<int> sum;
```

```
    Transform(list1.CBegin(), list1.CEnd(), list2.CBegin(), BackInserter(
        sum), Plus<int>());
    Console.Out() << sum << endl();
}
```

Implementation

algorithm.cm, page 8

1.3.187 UpperBound<I, T>(I, I, const T&) Function

Finds a position of the first element in a sorted sequence that is greater than the given value.

Syntax

`public nothrow I UpperBound<I, T>(I first, I last, const T& value);`

Constraint

where I is ForwardIterator and TotallyOrdered<T, I.ValueType>;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sorted sequence.
last	I	A forward iterator pointing one past the end of a sorted sequence.
value	const T&	A value to search.

Returns

I

Returns an iterator pointing to the first element in a sorted sequence that is greater than the given value, if there is one, otherwise returns *last*.

Example

```
using System;
using System.Collections;

//  Writes:
//  1, 1, 2, 2, 3, 3
//  upper bound of 2 is 3 at position 4
```

```
void main()
{
    List<int> list;
    list.Add(1);
    list.Add(2);
    list.Add(3);
    list.Add(2);
    list.Add(3);
    list.Add(1);
    Sort(list);
    Console.Out() << list << endl();
    List<int>.ConstIterator ub = UpperBound(list.CBegin(), list.CEnd(),
        2);
    if (ub != list.CEnd())
    {
        Console.Out() << "upper bound of 2 is " << *ub << " at position "
            << Distance(list.CBegin(), ub) << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

algorithm.cm, page 4

1.3.188 UpperBound<I, T, R>(I, I, const T&, R) Function

Finds a position of the first element in a sorted sequence that is greater than the given value according to the given ordering relation.

Syntax

```
public nothrow I UpperBound<I, T, R>(I first, I last, const T& value, R r);
```

Constraint

where I is ForwardIterator and T is I.ValueType and R is Relation and R.Domain is I.ValueType;

Parameters

Name	Type	Description
first	I	A forward iterator pointing to the beginning of a sorted sequence.

last	I	A forward iterator pointing one past the end of a sorted sequence.
value	const T&	A value to search.
r	R	An ordering relation.

Returns

I

Returns an iterator pointing to the first element in a sorted sequence that is greater than the given value according to the given ordering relation, if there is one, otherwise returns *last*.

Implementation

algorithm.cm, page 5

1.3.189 endl() Function

Returns EndLine object that represents an end of line character.

Syntax

```
public inline nothrow EndLine endl();
```

Returns

EndLine

Returns EndLine object that represents and end of line character.

1.3.190 operator==(Date, Date) Function

Syntax

```
public nothrow bool operator==(Date left, Date right);
```

1.3.191 operator<(Date, Date) Function

Syntax

```
public nothrow bool operator<(Date left, Date right);
```

1.3.192 CurrentDate() Function

Syntax

```
public Date CurrentDate();
```

1.3.193 ParseDate(const string&) Function

Syntax

```
public Date ParseDate(const string& s);
```

1.3.194 ToString(Date) Function

Syntax

```
public nothrow string ToString(Date date);
```

1.3.195 ToUtf8(uint) Function

Syntax

```
public string ToUtf8(uint c);
```

1.4 Constants

Constant	Type	Value	Description
EXIT_CHAR_CLASS_TABLE_ALLOCATE	int	249	Program exit status when the character class table could not be allocated.

2 System.Collections Namespace

Contains collection classes and functions that operate on collections.

Figure 2.1 contains the classes in this namespace.

Figure 2.1: Class Diagram: Collection Classes

System.Collections.BitSet

System.Collections.ForwardList<T>

System.Collections.ForwardListNodeIterator<T, R, P>

System.Collections.List<T>

System.Collections.Map<Key, Value, KeyCompare>

System.Collections.Queue<T>

System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>

System.Collections.RedBlackTreeNodeIterator<T, R, P>

System.Collections.Set<T, C>

System.Collections.Stack<T>

2.5 Classes

Class	Description
BitSet	A compact sequence of bits.
ForwardList<T>	A singly linked list of elements.
ForwardListNodeIterator<T, R, P>	A forward iterator that iterates throught a ForwardList<T>.
List<T>	A container of elements in which the contained elements are in consecutive locations in memory.
Map<Key, Value, KeyCompare>	An associative container of key-value pairs organized in a red-black tree. The keys need to be ordered.
Queue<T>	A first-in first-out data structure.
RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>	A self-balancing binary search tree of unique elements used to implement Set<T, C> and Map<Key, Value, KeyCompare>. The keys of the elements in the tree need to be ordered.
RedBlackTreeNodeIterator<T, R, P>	A bidirectional iterator that iterates through the elements in a reb-black tree.
Set<T, C>	A container that contains a set of unique elements organized in a red-black tree. The elements need to be ordered.
Stack<T>	A last-in first-out data structure.

2.5.1 BitSet Class

A compact sequence of bits.

Syntax

```
public class BitSet;
```

2.5.1.1 Example

```
using System;
using System.Collections;

//   Writes:
//   11111111
//   all bits are 1
//   10101010
//   s[0] == 1
//   01010101
//   now s[0] == 0

void main()
{
    int n = 8;
    BitSet s(n);
    s.Set();
    Console.Out() << s.ToString() << endl();
    if (s.All())
    {
        Console.Out() << "all bits are 1" << endl();
    }
    for (int i = 0; i < n; ++i)
    {
        if (i % 2 == 1)
        {
            s.Reset(i);
        }
    }
    Console.Out() << s.ToString() << endl();
    if (s[0])
    {
        Console.Out() << "s[0] == 1" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    s.Flip();
    Console.Out() << s.ToString() << endl();
    if (!s[0])
    {
        Console.Out() << "now s[0] == 0" << endl();
```

```
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

2.5.1.2 Member Functions

Member Function	Description
BitSet()	Constructor. Constructs an empty bit set.
BitSet(const string&)	Constructor. Constructs a bit set from a string of bits.
BitSet(BitSet&&)	Move constructor.
BitSet(const BitSet&)	Copy constructor.
BitSet(int)	Constructor. Constructs a bit set capable of holding given number of bits.
operator=(const BitSet&)	Copy assignment.
operator=(BitSet&&)	Move assignment.
~BitSet()	Destructor.
operator==(const BitSet&) const	Compares this bit set and given bit set for equality.
operator[](int) const	Returns true if the bit with the given index is 1 and false if it is 0.
All() const	Returns true if all the bits are 1, false otherwise.
Any() const	Returns true if any bit is 1, false otherwise.
Clear()	Makes the bit set empty.
Count() const	Returns the number of bits the bit set contains.

Flip()	Toggles the values of the bits that the bit set contains.
Flip(int)	Toggles the value of the bit with the given index.
None() const	Returns true if all the bits are 0, false otherwise.
Reset()	Resets all the bits to 0.
Reset(int)	Resets the bit with the given index to 0.
Resize(int)	Resizes the bit set to contain the given number of bits.
Set()	Sets all the bits to 1.
Set(int)	Sets the bit with the given index to 1.
Set(int, bool)	Sets the bit with the given index to the given value.
Test(int) const	Returns true if a bit with the given index is 1, false otherwise.
ToString() const	Returns the string representation of the bit set.

2.5.1.2.1 BitSet() Member Function

Constructor. Constructs an empty bit set.

Syntax

```
public nothrow BitSet();
```

Implementation

bitset.cm, page 1

2.5.1.2.2 BitSet(const string&) Member Function

Constructor. Constructs a bit set from a string of bits.

Syntax

```
public BitSet(const string& bits_);
```

Parameters

Name	Type	Description
bits_	const string&	A string of bits.

Implementation

bitset.cm, page 1

2.5.1.2.3 BitSet(BitSet&&) Member Function

Move constructor.

Syntax

```
public nothrow BitSet(BitSet&& that);
```

Parameters

Name	Type	Description
that	BitSet&&	A bit set to move from.

Implementation

bitset.cm, page 2

2.5.1.2.4 BitSet(const BitSet&) Member Function

Copy constructor.

Syntax

```
public BitSet(const BitSet& that);
```

Parameters

Name	Type	Description
that	const BitSet&	A bit set to copy.

Implementation

bitset.cm, page 2

2.5.1.2.5 BitSet(int) Member Function

Constructor. Constructs a bit set capable of holding given number of bits.

Syntax

```
public BitSet(int numBits_);
```

Parameters

Name	Type	Description
numBits_	int	Number of bits the bit set will contain.

Implementation

bitset.cm, page 1

2.5.1.2.6 operator=(const BitSet&) Member Function

Copy assignment.

Syntax

```
public void operator=(const BitSet& that);
```

Parameters

Name	Type	Description
that	const BitSet&	A bit set to assign.

Implementation

bitset.cm, page 2

2.5.1.2.7 operator=(BitSet&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(BitSet&& that);
```

Parameters

Name	Type	Description
that	BitSet&&	A bit set to move from.

Implementation

bitset.cm, page 2

2.5.1.2.8 ~BitSet() Member Function

Destructor.

Syntax

```
public nothrow ~BitSet();
```

Implementation

bitset.cm, page 2

2.5.1.2.9 operator==(const BitSet&) const Member Function

Compares this bit set and given bit set for equality.

Syntax

```
public bool operator==(const BitSet& that) const;
```

Parameters

Name	Type	Description
that	const BitSet&	A bit set to compare with.

Returns

bool

Returns true if both bitsets contain the same number of equal bits, false otherwise.

Implementation

bitset.cm, page 4

2.5.1.2.10 operator[](int) const Member Function

Returns true if the bit with the given index is 1 and false if it is 0.

Syntax

```
public bool operator[](int index) const;
```

Parameters

Name	Type	Description
index	int	Index of the bit to test.

Returns

bool

Returns true if the bit with the given index is 1 and false if it is 0.

Implementation

bitset.cm, page 3

2.5.1.2.11 All() const Member Function

Returns true if all the bits are 1, false otherwise.

Syntax

```
public bool All() const;
```

Returns

bool

Returns true if all the bits are 1, false otherwise.

Implementation

bitset.cm, page 3

2.5.1.2.12 Any() const Member Function

Returns true if any bit is 1, false otherwise.

Syntax

```
public bool Any() const;
```

Returns

bool

Returns true if any bit is 1, false otherwise.

Implementation

bitset.cm, page 4

2.5.1.2.13 Clear() Member Function

Makes the bit set empty.

Syntax

```
public nothrow void Clear();
```

Implementation

bitset.cm, page 2

2.5.1.2.14 Count() const Member Function

Returns the number of bits the bit set contains.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of bits the bit set contains.

Implementation

bitset.cm, page 2

2.5.1.2.15 Flip() Member Function

Toggles the values of the bits that the bit set contains.

Syntax

```
public void Flip();
```

Implementation

bitset.cm, page 3

2.5.1.2.16 Flip(int) Member Function

Toggles the value of the bit with the given index.

Syntax

```
public void Flip(int pos);
```

Parameters

Name	Type	Description
pos	int	Index of the bit to toggle.

Implementation

bitset.cm, page 3

2.5.1.2.17 None() const Member Function

Returns true if all the bits are 0, false otherwise.

Syntax

```
public bool None() const;
```

Returns

bool

Returns true if all the bits are 0, false otherwise.

Implementation

bitset.cm, page 4

2.5.1.2.18 Reset() Member Function

Resets all the bits to 0.

Syntax

```
public void Reset();
```

Implementation

bitset.cm, page 2

2.5.1.2.19 Reset(int) Member Function

Resets the bit with the given index to 0.

Syntax

```
public void Reset(int pos);
```

Parameters

Name	Type	Description
pos	int	Index of the bit to reset.

Implementation

bitset.cm, page 2

2.5.1.2.20 Resize(int) Member Function

Resizes the bit set to contain the given number of bits.

Syntax

```
public void Resize(int numBits_);
```

Parameters

Name	Type	Description

numBits_	int	Number of bits the bit set will contain.

Implementation

bitset.cm, page 2

2.5.1.2.21 Set() Member Function

Sets all the bits to 1.

Syntax

```
public void Set();
```

Implementation

bitset.cm, page 2

2.5.1.2.22 Set(int) Member Function

Sets the bit with the given index to 1.

Syntax

```
public void Set(int pos);
```

Parameters

Name	Type	Description
pos	int	Index of the bit to set.

Implementation

bitset.cm, page 2

2.5.1.2.23 Set(int, bool) Member Function

Sets the bit with the given index to the given value.

Syntax

```
public void Set(int pos, bool bit);
```

Parameters

Name	Type	Description
pos	int	Index of the bit to set.

bit	bool	If true, the bit will be set to 1, otherwise it will be reset to 0.

Implementation

bitset.cm, page 3

2.5.1.2.24 Test(int) const Member Function

Returns true if a bit with the given index is 1, false otherwise.

Syntax

```
public bool Test(int pos) const;
```

Parameters

Name	Type	Description
pos	int	Index of the bit to test.

Returns

bool

Returns true if a bit with the given index is 1, false otherwise.

Implementation

bitset.cm, page 3

2.5.1.2.25 ToString() const Member Function

Returns the string representation of the bit set.

Syntax

```
public string ToString() const;
```

Returns

string

Returns the string representation of the bit set.

Implementation

bitset.cm, page 5

2.5.2 ForwardList<T> Class

A singly linked list of elements.

Syntax

```
public class ForwardList<T>;
```

Constraint

where T is Regular;

Model of

ForwardContainer<T>

2.5.2.1 Remarks

ForwardList<**T**> is suitable only for very short sequences of items. Often a List<T> is more appropriate.

2.5.2.2 Type Definitions

Name	Type	Description
ConstIterator	ForwardListNodeIterator<T, const T&, const T*>	A constant iterator type.
Iterator	ForwardListNodeIterator<T, T&, T*>	An iterator type.
ValueType	T	The type of the contained element.

2.5.2.3 Member Functions

Member Function	Description
ForwardList()	Constructor. Constructs an empty forward list.
ForwardList(const ForwardList<T>&)	Copy constructor.
ForwardList(ForwardList<T>&&)	Move constructor.
operator=(ForwardList<T>&&)	Move assignment.
operator=(const ForwardList<T>&)	Copy assignment.

~ForwardList()	Destructor.
Begin()	Returns an iterator pointing to the first element of the forward list, or End() if the forward list is empty.
Begin() const	Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.
CBegin() const	Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.
CEnd() const	Returns a constant iterator pointing to one past the end of the forward list.
Clear()	Makes the forward list empty.
Count() const	Counts the number of elements in the forward list.
End()	Returns an iterator pointing to one past the end of the forward list.
End() const	Returns a constant iterator pointing to one past the end of the forward list.
Front() const	Returns a constant reference to the first element in the forward list.
InsertAfter(Iterator, const ValueType&)	Inserts an element after the position pointed by the given iterator, or to the front of the forward list if the forward list is empty and the iterator is an End() iterator.
InsertFront(const ValueType&)	Inserts an element to the front of the forward list.
IsEmpty() const	Returns true if the forward list is empty, false otherwise.

Remove(const ValueType&)	Removes all occurrences of the given value from the forward list.
RemoveAfter(Iterator)	Removes an element after the position pointed by the given iterator.
RemoveFront()	Removes an element from the front of the forward list.
Swap(ForwardList<T>&)	Exchanges the contents with another forward list of the same type.

2.5.2.3.1 ForwardList() Member Function

Constructor. Constructs an empty forward list.

Syntax

```
public nothrow ForwardList();
```

Implementation

fwdlist.cm, page 3

2.5.2.3.2 ForwardList(const ForwardList<T>&) Member Function

Copy constructor.

Syntax

```
public ForwardList(const ForwardList<T>& that);
```

Parameters

Name	Type	Description
that	const ForwardList<T>&	A forward list to copy.

Implementation

fwdlist.cm, page 3

2.5.2.3.3 ForwardList(ForwardList<T>&&) Member Function

Move constructor.

Syntax

```
public nothrow ForwardList(ForwardList<T>&& that);
```

Parameters

Name	Type	Description
that	ForwardList<T>&&	A forward list to move from.

Implementation

fwdlist.cm, page 3

2.5.2.3.4 operator=(ForwardList<T>&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(ForwardList<T>&& that);
```

Parameters

Name	Type	Description
that	ForwardList<T>&&	A forward list to move from.

Implementation

fwdlist.cm, page 3

2.5.2.3.5 operator=(const ForwardList<T>&) Member Function

Copy assignment.

Syntax

```
public void operator=(const ForwardList<T>& that);
```

Parameters

Name	Type	Description
that	const ForwardList<T>&	A forward list to assign.

Implementation

fwdlist.cm, page 3

2.5.2.3.6 ~ForwardList() Member Function

Destructor.

Syntax

```
public nothrow ~ForwardList();
```

Implementation

fwdlist.cm, page 3

2.5.2.3.7 Begin() Member Function

Returns an iterator pointing to the first element of the forward list, or End() if the forward list is empty.

Syntax

```
public nothrow Iterator Begin();
```

Returns

Iterator

Returns an iterator pointing to the first element of the forward list, or End() if the forward list is empty.

Implementation

fwdlist.cm, page 4

2.5.2.3.8 Begin() const Member Function

Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.

Syntax

```
public nothrow ConstIterator Begin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.

Implementation

fwdlist.cm, page 4

2.5.2.3.9 CBegin() const Member Function

Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.

Syntax

```
public nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the first element of the forward list, or CEnd() const if the forward list is empty.

Implementation

fwdlist.cm, page 4

2.5.2.3.10 CEnd() const Member Function

Returns a constant iterator pointing to one past the end of the forward list.

Syntax

```
public nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to one past the end of the forward list.

Implementation

fwdlist.cm, page 4

2.5.2.3.11 Clear() Member Function

Makes the forward list empty.

Syntax

```
public nothrow void Clear();
```

Implementation

fwdlist.cm, page 3

2.5.2.3.12 Count() const Member Function

Counts the number of elements in the forward list.

Syntax

```
public nothrow int Count() const;
```

Returns

int

Returns the number of elements in the forward list.

Remarks

This is a *O(n)* operation where n is the number of elements in the forward list.

Implementation

fwdlist.cm, page 3

2.5.2.3.13 End() Member Function

Returns an iterator pointing to one past the end of the forward list.

Syntax

```
public nothrow Iterator End();
```

Returns

Iterator

Returns an iterator pointing to one past the end of the forward list.

Implementation

fwdlist.cm, page 4

2.5.2.3.14 End() const Member Function

Returns a constant iterator pointing to one past the end of the forward list.

Syntax

```
public nothrow ConstIterator End() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to one past the end of the forward list.

Implementation

fwdlist.cm, page 4

2.5.2.3.15 Front() const Member Function

Returns a constant reference to the first element in the forward list.

Syntax

```
public const ValueType& Front() const;
```

Returns

const ValueType&

Returns a constant reference to the first element in the forward list.

Implementation

fwdlist.cm, page 4

2.5.2.3.16 InsertAfter(Iterator, const ValueType&) Member Function

Inserts an element after the position pointed by the given iterator, or to the front of the forward list if the forward list is empty and the iterator is an End() iterator.

Syntax

```
public Iterator InsertAfter(Iterator pos, const ValueType& value);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator.
value	const ValueType&	A value to insert.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Implementation

fwdlist.cm, page 4

2.5.2.3.17 InsertFront(const ValueType&) Member Function

Inserts an element to the front of the forward list.

Syntax

```
public Iterator InsertFront(const ValueType& value);
```

Parameters

Name	Type	Description

value	const ValueType&	A value to insert.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Implementation

fwdlist.cm, page 4

2.5.2.3.18 IsEmpty() const Member Function

Returns true if the forward list is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the forward list is empty, false otherwise.

Implementation

fwdlist.cm, page 3

2.5.2.3.19 Remove(const ValueType&) Member Function

Removes all occurrences of the given value from the forward list.

Syntax

```
public void Remove(const ValueType& value);
```

Parameters

Name	Type	Description
value	const ValueType&	A value to remove.

Implementation

fwdlist.cm, page 5

2.5.2.3.20 RemoveAfter(Iterator) Member Function

Removes an element after the position pointed by the given iterator.

Syntax

```
public void RemoveAfter(Iterator pos);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator.

Implementation

fwdlist.cm, page 5

2.5.2.3.21 RemoveFront() Member Function

Removes an element from the front of the forward list.

Syntax

```
public void RemoveFront();
```

Implementation

fwdlist.cm, page 4

2.5.2.3.22 Swap(ForwardList<T>&) Member Function

Exchanges the contents with another forward list of the same type.

Syntax

```
public nothrow void Swap(ForwardList<T>& that);
```

Parameters

Name	Type	Description
that	ForwardList<T>&	A forward list to exchange contents with.

Implementation

fwdlist.cm, page 5

2.5.2.4 Nonmember Functions

Function	Description
operator==<T>(const ForwardList<T>&, const ForwardList<T>&)	Compares two forward lists for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.
operator<<T>(const ForwardList<T>&, const ForwardList<T>&)	Compares two forward lists for less than relationship and returns true if the first forward list comes lexicographically before the second forward list, false otherwise.

2.5.2.4.1 operator==<T>(const ForwardList<T>&, const ForwardList<T>&) Function

Compares two forward lists for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.

Syntax

```
public bool operator==<T>(const ForwardList<T>& left, const ForwardList<T>&
right);
```

Constraint

where T is Regular;

Parameters

Name	Type	Description
left	const ForwardList<T>&	The first forward list.
right	const ForwardList<T>&	The second forward list.

Returns

bool

returns true if both contain the same number of pairwise equal elements, false otherwise.

Implementation

fwdlist.cm, page 6

2.5.2.4.2 operator<<T>(const ForwardList<T>&, const ForwardList<T>&) Function

Compares two forward lists for less than relationship and returns true if the first forward list comes lexicographically before the second forward list, false otherwise.

Syntax

```
public bool operator<<T>(const ForwardList<T>& left, const ForwardList<T>&
right);
```

Constraint

where T is TotallyOrdered;

Parameters

Name	Type	Description
left	const ForwardList<T>&	The first forward list.
right	const ForwardList<T>&	The second forward list.

Returns

bool

Returns true if the first forward list comes lexicographically before the second forward list, false otherwise.

Implementation

fwdlist.cm, page 6

2.5.3 ForwardListNodeIterator<T, R, P> Class

A forward iterator that iterates throught a ForwardList<T>.

Syntax

```
public class ForwardListNodeIterator<T, R, P>;
```

Model of

ForwardIterator<T>

2.5.3.1 Type Definitions

Name	Type	Description
PointerType	P	The type of a pointer to an element.
ReferenceType	R	The type of a reference to an element.
ValueType	T	The type of an element.

2.5.3.2 Member Functions

Member Function	Description
ForwardListNodeIterator()	Constructor. Default constructs a forward list node iterator.
ForwardListNodeIterator(ForwardListNode<T>*)	Constructor. Constructs a forward list node iterator pointing to a forward list node.
operator++()	Advances the iterator pointing to the next element in the forward list.
operator->() const	Returns a pointer to an element.
operator*() const	Returns a reference to an element.
GetNode() const	Returns the contained pointer to an element.

2.5.3.2.1 ForwardListNodeIterator() Member Function

Constructor. Default constructs a forward list node iterator.

Syntax

```
public nothrow ForwardListNodeIterator();
```

Implementation

fwdlist.cm, page 2

2.5.3.2.2 ForwardListNodeIterator(ForwardListNode<T>*) Member Function

Constructor. Constructs a forward list node iterator pointing to a forward list node.

Syntax

```
public nothrow ForwardListNodeIterator(ForwardListNode<T>* node_);
```

Parameters

Name	Type	Description
node_	ForwardListNode<T>*	A pointer to a forward list node.

Implementation

fwdlist.cm, page 2

2.5.3.2.3 operator++() Member Function

Advances the iterator pointing to the next element in the forward list.

Syntax

```
public inline nothrow ForwardListNodeIterator<T, R, P>& operator++();
```

Returns

ForwardListNodeIterator<T, R, P>&

Returns a reference to the iterator.

Implementation

fwdlist.cm, page 2

2.5.3.2.4 operator->() const Member Function

Returns a pointer to an element.

Syntax

```
public nothrow PointerType operator->() const;
```

Returns

PointerType

Returns a pointer to an element.

Implementation

fwdlist.cm, page 2

2.5.3.2.5 operator*() const Member Function

Returns a reference to an element.

Syntax

```
public nothrow ReferenceType operator*() const;
```

Returns

ReferenceType

Returns a reference to an element.

Implementation

fwdlist.cm, page 2

2.5.3.2.6 GetNode() const Member Function

Returns the contained pointer to an element.

Syntax

```
public inline nothrow ForwardListNode<T>* GetNode() const;
```

Returns

ForwardListNode<T>*

Returns the contained pointer to an element.

Implementation

fwdlist.cm, page 2

2.5.3.3 Nonmember Functions

Function	Description
operator==<T, R, P>(ForwardListNodeIterator<T, R, P>, ForwardListNodeIterator<T, R, P>)	Compares two forward list node iterators for equality.

2.5.3.3.1 operator==<T, R, P>(ForwardListNodeIterator<T, R, P>, ForwardListNodeIterat R, P>) Function

Compares two forward list node iterators for equality.

Syntax

```
public nothrow bool operator==<T, R, P>(ForwardListNodeIterator<T, R, P> left,
ForwardListNodeIterator<T, R, P> right);
```

Parameters

Name	Type	Description
left	ForwardListNodeIterator<T, R, P>	The first forward list node iterator.
right	ForwardListNodeIterator<T, R, P>	The second forward list node iterator.

Returns

bool

Returns true if both iterators point to same forward list node, or both are End() iterators, false otherwise.

Implementation

fwdlist.cm, page 6

2.5.4 List<T> Class

A container of elements in which the contained elements are in consecutive locations in memory.

Syntax

```
public class List<T>;
```

Constraint

where T is Semiregular;

Model of

BackInsertionSequence<T>

InsertionSequence<T>

FrontInsertionSequence<T>

RandomAccessContainer<T>

2.5.4.1 Example

```
using System;
using System.Collections;

//  Writes:
//  0, 1, 2
//  1
//  0, 2
//  0, 3

void main()
{
    List<int> intList;
    intList.Add(0);
    intList.Add(2);
    intList.Insert(intList.Begin() + 1, 1);
    Console.Out() << intList << endl();
    int first = intList.RemoveFirst();
    #assert(first == 0);
    int last = intList.RemoveLast();
    #assert(last == 2);
    #assert(intList.Count() == 1);
    Console.Out() << intList << endl();
    intList.InsertFront(0);
    intList.Add(2);
    int one = intList.Remove(intList.Begin() + 1);
    #assert(one == 1);
```

```
    Console.Out() << intList << endl();
    int zero = intList[0];
    #assert(zero == 0);
    intList[1] = 3;
    Console.Out() << intList << endl();
    intList.Clear();
    #assert(intList.IsEmpty());
}
```

2.5.4.2 Type Definitions

Name	Type	Description
ConstIterator	RandomAccessIter<T, const T&, const T*>	A constant iterator type.
Iterator	RandomAccessIter<T, T&, T*>	An iterator type.
ValueType	T	The type of the contained element.

2.5.4.3 Member Functions

Member Function	Description
List()	Constructor. Constructs an empty list.
List(List<T>&&)	Move constructor.
List(const List<T>&)	Copy constructor.
List(int, const ValueType&)	Constructor. Constructs a list with the given number of the given element.
operator=(List<T>&&)	Move assignment.
operator=(const List<T>&)	Copy assignment.
~List()	Destructor.
operator[](int)	Returns a reference to the element with the given index.
operator[](int) const	Returns a constant reference to the element with the given index.

Add(ValueType&&)	Moves an element to the end of the list.
Add(const ValueType&)	Adds an element to the end of the list.
Back()	Returns a reference to the last element in the list.
Back() const	Returns a constant reference to the last element in the list.
Begin()	Returns an iterator pointing to the first element of the list, or End() if the list is empty.
Begin() const	Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.
CBegin() const	Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.
CEnd() const	Returns a constant iterator pointing one past the end of the list.
Capacity() const	Returns the number of elements that the list can contain without a memory allocation.
Clear()	Makes the list empty.
Count() const	Returns the number of elements in the list.
End()	Returns an iterator pointing one past the end of the list.
End() const	Returns a constant iterator pointing one past the end of the list.
Front()	Returns a reference to the first element in the list.

Front() const	Returns a constant reference to the first element in the list.
Insert(Iterator, ValueType&&)	Moves an element into the list before the given position.
Insert(Iterator, const ValueType&)	Inserts the given element before the given position in the list.
InsertFront(ValueType&&)	Moves an element to the front of the list.
InsertFront(const ValueType&)	Inserts an element to the front of the list.
IsEmpty() const	Returns true if the list is empty, false otherwise.
Remove(Iterator)	Removes an element pointed by the given iterator from the list and returns the removed element.
RemoveFirst()	Removes the first element from the list.
RemoveLast()	Removes the last element from the list.
Reserve(int)	Makes the capacity of the list at least the given number of elements.
Resize(int)	Makes the size of the list equal to the given number of elements.
Swap(List<T>&)	Exchanges the contents of the list with another list.

2.5.4.3.1 List() Member Function

Constructor. Constructs an empty list.

Syntax

```
public nothrow List();
```

Implementation

list.cm, page 1

2.5.4.3.2 List(List<T>&&) Member Function

Move constructor.

Syntax

```
public nothrow List(List<T>&& that);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
that	List<T>&&	A list to move from.

Implementation

list.cm, page 1

2.5.4.3.3 List(const List<T>&) Member Function

Copy constructor.

Syntax

```
public List(const List<T>& that);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
that	const List<T>&	A list to copy.

Implementation

list.cm, page 1

2.5.4.3.4 List(int, const ValueType&) Member Function

Constructor. Constructs a list with the given number of the given element.

Syntax

```
public List(int n, const ValueType& value);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
n	int	A number of elements.
value	const ValueType&	An element.

Implementation

list.cm, page 2

2.5.4.3.5 operator=(List<T>&&) Member Function

Move assignment.

Syntax

```
public default nothrow void operator=(List<T>&& that);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
that	List<T>&&	A list to move from.

Implementation

list.cm, page 2

2.5.4.3.6 operator=(const List<T>&) Member Function

Copy assignment.

Syntax

```
public void operator=(const List<T>& that);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
that	const List<T>&	A list to assign.

Implementation

list.cm, page 2

2.5.4.3.7 ∼List() Member Function

Destructor.

Syntax

```
public nothrow ~List();
```

2.5.4.3.8 operator[](int) Member Function

Returns a reference to the element with the given index.

Syntax

```
public nothrow ValueType& operator[](int index);
```

Parameters

Name	Type	Description
index	int	An index.

Returns

ValueType&

Returns a reference to the element with the given index.

Implementation

list.cm, page 5

2.5.4.3.9 operator[](int) const Member Function

Returns a constant reference to the element with the given index.

Syntax

```
public nothrow const ValueType& operator[](int index) const;
```

Parameters

Name	Type	Description
index	int	An index.

Returns

const ValueType&

Returns a constant reference to the element with the given index.

Implementation

list.cm, page 5

2.5.4.3.10 Add(ValueType&&) Member Function

Moves an element to the end of the list.

Syntax

```
public void Add(ValueType&& item);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
item	ValueType&&	An element to move.

Implementation

list.cm, page 3

2.5.4.3.11 Add(const ValueType&) Member Function

Adds an element to the end of the list.

Syntax

```
public void Add(const ValueType& item);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
item	const ValueType&	An element to add.

Implementation

list.cm, page 3

2.5.4.3.12 Back() Member Function

Returns a reference to the last element in the list.

Syntax

```
public nothrow ValueType& Back();
```

Returns

ValueType&

Returns a reference to the last element in the list.

Implementation

list.cm, page 6

2.5.4.3.13 Back() const Member Function

Returns a constant reference to the last element in the list.

Syntax

```
public nothrow const ValueType& Back() const;
```

Returns

const ValueType&

Returns a constant reference to the last element in the list.

Implementation

list.cm, page 6

2.5.4.3.14 Begin() Member Function

Returns an iterator pointing to the first element of the list, or End() if the list is empty.

Syntax

```
public nothrow Iterator Begin();
```

Returns

Iterator

Returns an iterator pointing to the first element of the list, or End() if the list is empty.

Implementation

list.cm, page 5

2.5.4.3.15 Begin() const Member Function

Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.

Syntax

```
public nothrow ConstIterator Begin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.

Implementation

list.cm, page 5

2.5.4.3.16 CBegin() const Member Function

Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.

Syntax

```
public nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the first element of the list, or CEnd() const if the list is empty.

Implementation

list.cm, page 5

2.5.4.3.17 CEnd() const Member Function

Returns a constant iterator pointing one past the end of the list.

Syntax

```
public nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant iterator pointing one past the end of the list.

Implementation

list.cm, page 5

2.5.4.3.18 Capacity() const Member Function

Returns the number of elements that the list can contain without a memory allocation.

Syntax

```
public inline nothrow int Capacity() const;
```

Returns

int

Returns the number of elements that the list can contain without a memory allocation.

Implementation

list.cm, page 2

2.5.4.3.19 Clear() Member Function

Makes the list empty.

Syntax

```
public nothrow void Clear();
```

Implementation

list.cm, page 3

2.5.4.3.20 Count() const Member Function

Returns the number of elements in the list.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of elements in the list.

Implementation

list.cm, page 3

2.5.4.3.21 End() Member Function

Returns an iterator pointing one past the end of the list.

Syntax

```
public nothrow Iterator End();
```

Returns

Iterator

Returns an iterator pointing one past the end of the list.

Implementation

list.cm, page 5

2.5.4.3.22 End() const Member Function

Returns a constant iterator pointing one past the end of the list.

Syntax

```
public nothrow ConstIterator End() const;
```

Returns

ConstIterator

Returns a constant iterator pointing one past the end of the list.

Implementation

list.cm, page 5

2.5.4.3.23 Front() Member Function

Returns a reference to the first element in the list.

Syntax

```
public nothrow ValueType& Front();
```

Returns

ValueType&

Returns a reference to the first element int the list.

Implementation

list.cm, page 5

2.5.4.3.24 Front() const Member Function

Returns a constant reference to the first element in the list.

Syntax

```
public nothrow const ValueType& Front() const;
```

Returns

const ValueType&

Returns a constant reference to the first element in the list.

Implementation

list.cm, page 5

2.5.4.3.25 Insert(Iterator, ValueType&&) Member Function

Moves an element into the list before the given position.

Syntax

```
public Iterator Insert(Iterator pos, ValueType&& item);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
pos	Iterator	An itertor pointing to the position to insert.
item	ValueType&&	An element to move.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Remarks

The list may grow when an item is inserted so the iterator returned may not be the same as the supplied iterator.

Implementation

list.cm, page 4

2.5.4.3.26 Insert(Iterator, const ValueType&) Member Function

Inserts the given element before the given position in the list.

Syntax

```
public Iterator Insert(Iterator pos, const ValueType& item);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
pos	Iterator	An itertor pointing to the position to insert.
item	const ValueType&	An element to insert.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Remarks

The list may grow when an item is inserted so the iterator returned may not be the same as the supplied iterator.

Implementation

list.cm, page 3

2.5.4.3.27 InsertFront(ValueType&&) Member Function

Moves an element to the front of the list.

Syntax

```
public Iterator InsertFront(ValueType&& item);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
item	ValueType&&	An element to move.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Remarks

The list may grow when an item is inserted so the iterator returned may not be the same as the supplied iterator.

Implementation

list.cm, page 4

2.5.4.3.28 InsertFront(const ValueType&) Member Function

Inserts an element to the front of the list.

Syntax

```
public Iterator InsertFront(const ValueType& item);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
item	const ValueType&	An element to insert.

Returns

Iterator

Returns an iterator pointing to the inserted element.

Remarks

The list may grow when an item is inserted so the iterator returned may not be the same as the supplied iterator.

Implementation

list.cm, page 4

2.5.4.3.29 IsEmpty() const Member Function

Returns true if the list is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the list is empty, false otherwise.

Implementation

list.cm, page 3

2.5.4.3.30 Remove(Iterator) Member Function

Removes an element pointed by the given iterator from the list and returns the removed element.

Syntax

```
public ValueType Remove(Iterator pos);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator pointing to the element to remove.

Returns

ValueType

Returns the removed element.

Implementation

list.cm, page 4

2.5.4.3.31 RemoveFirst() Member Function

Removes the first element from the list.

Syntax

```
public ValueType RemoveFirst();
```

Returns

ValueType

Returns the removed element.

Implementation

list.cm, page 4

2.5.4.3.32 RemoveLast() Member Function

Removes the last element from the list.

Syntax

```
public ValueType RemoveLast();
```

Returns

ValueType

Returns the removed element,.

Implementation

list.cm, page 4

2.5.4.3.33 Reserve(int) Member Function

Makes the capacity of the list at least the given number of elements.

Syntax

```
public void Reserve(int minRes);
```

Parameters

Name	Type	Description
minRes	int	The minimum number of elements the list can hold without a memory allocation.

Implementation

list.cm, page 2

2.5.4.3.34 Resize(int) Member Function

Makes the size of the list equal to the given number of elements.

Syntax

```
public void Resize(int newCount);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
newCount	int	The number of elements that the list will hold.

Remarks

If the given number of elements is less than the current number of elements, the list is shrinked. If the given number of elements if greater than the current number of elements, additional default constructed elements are inserted to the end of the list.

Implementation

list.cm, page 2

2.5.4.3.35 Swap(List<T>&) Member Function

Exchanges the contents of the list with another list.

Syntax

```
public nothrow void Swap(List<T>& that);
```

Parameters

Name	Type	Description
that	List<T>&	A list to exchanges contents with.

Implementation

list.cm, page 6

2.5.4.4 Nonmember Functions

Function	Description
operator==<T>(const List<T>&, const List<T>&)	Compares two lists for equality and returns true if both lists contain the same number of pairwise equal elements, false otherwise.

operator<<T>(const List<T>&, const List<T>&)	Compares two lists for less than relationship and returns true if the first list comes lexicographically before the second list, false otherwise.

2.5.4.4.1 operator==<T>(const List<T>&, const List<T>&) Function

Compares two lists for equality and returns true if both lists contain the same number of pairwise equal elements, false otherwise.

Syntax

```
public nothrow bool operator==<T>(const List<T>& left, const List<T>& right);
```

Constraint

where T is Regular;

Parameters

Name	Type	Description
left	const List<T>&	The first list.
right	const List<T>&	The second list.

Returns

bool

Returns true if both lists contain the same number of pairwise equal elements, false otherwise.

Implementation

list.cm, page 7

2.5.4.4.2 operator<<T>(const List<T>&, const List<T>&) Function

Compares two lists for less than relationship and returns true if the first list comes lexicographically before the second list, false otherwise.

Syntax

```
public nothrow bool operator<<T>(const List<T>& left, const List<T>& right);
```

Constraint

where T is TotallyOrdered;

Parameters

Name	Type	Description
left	const List<T>&	The first list.
right	const List<T>&	The second list.

Returns

bool

Returns true if the first list comes lexicographically before the second list, false otherwise.

Implementation

list.cm, page 7

2.5.5 Map<Key, Value, KeyCompare> Class

An associative container of key-value pairs organized in a red-black tree. The keys need to be ordered.

Syntax

```
public class Map<Key, Value, KeyCompare>;
```

Constraint

where Key is Semiregular and Value is Semiregular and KeyCompare is Relation and KeyCompare.Domain is Key;

Model of

BidirectionalContainer<T>

Default Template Arguments

KeyCompare = Less<Key>

2.5.5.1 Example

```
using System;
using System.Collections;

//  Writes:
//  the phone number of Stepanov, Alexander is 765432
//  Knuth already inserted
//  Dijkstra, Edsger W. : 111222
//  Knuth, Donald E. : 999888
//  Stepanov, Alexander : 765432
//  Stroustrup, Bjarne : 123456
//  Turing, Alan : 555444

void main()
{
    Map<string, int> phoneBook;
    phoneBook["Stroustrup, Bjarne"] = 123456;
    phoneBook["Stepanov, Alexander"] = 765432;
    phoneBook["Knuth, Donald E."] = 999888;
    phoneBook["Dijkstra, Edsger W."] = 111222;
    phoneBook.Insert(MakePair(string("Turing, Alan"), 555444));

    Map<string, int>.Iterator s = phoneBook.Find("Stepanov, Alexander");
    if (s != phoneBook.End())
    {
        Console.Out() << "the phone number of " << s->first << " is " <<
            s->second << endl();
    }
```

```
    else
    {
        Console.Error() << "phone number not found" << endl();
    }

    if (!phoneBook.Insert(MakePair(string("Knuth, Donald E."), 999888)).
        second)
    {
        Console.Out() << "Knuth already inserted" << endl();
    }

    for (const Pair<string, int>& p : phoneBook)
    {
        Console.Out() << p.first << " : " << p.second << endl();
    }
}
```

2.5.5.2 Type Definitions

Name	Type	Description
Compare	KeyCompare	The type of a relation used to compare keys.
ConstIterator	RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType*>	A constant iterator type.
Iterator	RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>	An iterator type.
KeyType	Key	The type of key.
MappedType	Value	The type associated with the key.
ValueType	Pair<Key, Value>	A pair composed of key type and mapped type.

2.5.5.3 Member Functions

Member Function	Description
Map()	Default constructor. Constructs an empty map.
Map(const Map<Key, Value, KeyCompare>&)	Copy constructor.
Map(Map<Key, Value, KeyCompare>&&)	Move constructor.

operator=(const Map<Key, Value, KeyCompare>&)	Copy assignment.
operator=(Map<Key, Value, KeyCompare>&&)	Move assignment.
~Map()	Destructor.
operator[](const KeyType&)	Returns a reference to the value associated with the given key. If there are currently no value associated with the given key, a default constructed value is created and inserted in the map.
Begin()	Returns an iterator pointing to the beginning of the map.
CBegin() const	Returns a constant iterator pointing to the beginning of the map.
CEnd() const	Returns a constant iterator pointing to one past the end of the map.
Clear()	Makes the map empty.
Count() const	Returns the number of key-value pairs in the map.
End()	Returns an iterator pointing to one past the end of the map.
Find(const KeyType&)	Searches the given key in the map and returns an iterator pointing to it, if found, or an End() iterator otherwise.
Insert(const ValueType&)	Inserts a key-value pair to the map if the map does not already contain the key. In that case returns a pair consisting of an iterator pointing to the inserted element and **true**. Otherwise does not insert an element, but returns a pair consisting of an iterator pointing to the previously inserted element and **false**.

Insert(ValueType&&)	Moves a key-value pair to the map if the map does not already contain the key. I that case returns a pair consisting of an iterator pointing to the inserted element and **true**. Otherwise does not insert an element, but returns a pair consisting of an iterator pointing to the previously inserted element and **false**.
IsEmpty() const	Returns true if the map is empty, false otherwise.
Remove(Iterator)	Removes a key-value pair pointed by the given iterator from the map.
Remove(const KeyType&)	Removes a key-value pair associated with the given key from the map.
Swap(Map<Key, Value, KeyCompare>&)	Exchanges the contents with another map of the same type.

2.5.5.3.1 Map() Member Function

Default constructor. Constructs an empty map.

Syntax

```
public Map();
```

Implementation

map.cm, page 1

2.5.5.3.2 Map(const Map<Key, Value, KeyCompare>&) Member Function

Copy constructor.

Syntax

```
public default Map(const Map<Key, Value, KeyCompare>& that);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description

that	const Map<Key, Value, KeyCompare>&	A map to copy.

Implementation

map.cm, page 1

2.5.5.3.3 Map(Map<Key, Value, KeyCompare>&&) Member Function

Move constructor.

Syntax

```
public default nothrow Map(Map<Key, Value, KeyCompare>&& that);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
that	Map<Key, Value, KeyCompare>&&	A map to move from.

Implementation

map.cm, page 1

2.5.5.3.4 operator=(const Map<Key, Value, KeyCompare>&) Member Function

Copy assignment.

Syntax

```
public default void operator=(const Map<Key, Value, KeyCompare>& that);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description
that	const Map<Key, Value, KeyCompare>&	A map to assign.

Implementation

map.cm, page 1

2.5.5.3.5 operator=(Map<Key, Value, KeyCompare>&&) Member Function

Move assignment.

Syntax

```
public default nothrow void operator=(Map<Key, Value, KeyCompare>&& that);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
that	Map<Key, Value, KeyCompare>&&	A map to move from.

Implementation

map.cm, page 1

2.5.5.3.6 ~Map() Member Function

Destructor.

Syntax

```
public default nothrow ~Map();
```

Implementation

map.cm, page 1

2.5.5.3.7 operator[](const KeyType&) Member Function

Returns a reference to the value associated with the given key. If there are currently no value associated with the given key, a default constructed value is created and inserted in the map.

Syntax

```
public MappedType& operator[](const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	A key.

Returns

MappedType&

A value associated with the key.

Implementation

map.cm, page 2

2.5.5.3.8 Begin() Member Function

Returns an iterator pointing to the beginning of the map.

Syntax

```
public inline nothrow Iterator Begin();
```

Returns

Iterator

Returns an iterator pointing to the beginning of the map.

Implementation

map.cm, page 1

2.5.5.3.9 CBegin() const Member Function

Returns a constant iterator pointing to the beginning of the map.

Syntax

```
public inline nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the beginning of the map.

Implementation

map.cm, page 2

2.5.5.3.10 CEnd() const Member Function

Returns a constant iterator pointing to one past the end of the map.

Syntax

```
public inline nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to one past the end of the map.

Implementation

map.cm, page 2

2.5.5.3.11 Clear() Member Function

Makes the map empty.

Syntax

```
public inline nothrow void Clear();
```

Implementation

map.cm, page 2

2.5.5.3.12 Count() const Member Function

Returns the number of key-value pairs in the map.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of key-value pairs in the map.

Implementation

map.cm, page 2

2.5.5.3.13 End() Member Function

Returns an iterator pointing to one past the end of the map.

Syntax

```
public inline nothrow Iterator End();
```

Returns

Iterator

Returns an iterator pointing to one past the end of the map.

Implementation

map.cm, page 2

2.5.5.3.14 Find(const KeyType&) Member Function

Searches the given key in the map and returns an iterator pointing to it, if found, or an End() iterator otherwise.

Syntax

```
public inline nothrow Iterator Find(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	A key to search.

Returns

Iterator

Returns an iterator pointing to the found key-value pair, if the search was successful, or End() iterator otherwise.

Implementation

map.cm, page 2

2.5.5.3.15 Insert(const ValueType&) Member Function

Inserts a key-value pair to the map if the map does not already contain the key. In that case returns a pair consisting of an iterator pointing to the inserted element and **true**. Otherwise does not insert an element, but returns a pair consisting of an iterator pointing to the previously inserted element and **false**.

Syntax

```
public inline Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>,
bool> Insert(const ValueType& value);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description
value	const ValueType&	A key-value pair to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting an iterator pointing to the key-value pair in the map, and a Boolean value indicating whether the element was inserted in the map.

Implementation

map.cm, page 2

2.5.5.3.16 Insert(ValueType&&) Member Function

Moves a key-value pair to the map if the map does not already contain the key. I that case returns a pair consisting of an iterator pointing to the inserted element and **true**. Otherwise does not insert an element, but returns a pair consisting of an iterator pointing to the previously inserted element and **false**.

Syntax

```
public inline Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>,
bool> Insert(ValueType&& value);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
value	ValueType&&	A key-value pair to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting an iterator pointing to the key-value pair in the map, and a Boolean value indicating whether the element was inserted in the map.

Implementation

map.cm, page 2

2.5.5.3.17 IsEmpty() const Member Function

Returns true if the map is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the map is empty, false otherwise.

Implementation

map.cm, page 2

2.5.5.3.18 Remove(Iterator) Member Function

Removes a key-value pair pointed by the given iterator from the map.

Syntax

```
public inline nothrow void Remove(Iterator pos);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator pointing to a key-value pair in the map.

Implementation

map.cm, page 2

2.5.5.3.19 Remove(const KeyType&) Member Function

Removes a key-value pair associated with the given key from the map.

Syntax

```
public inline nothrow bool Remove(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	A key.

Returns

bool

Returns true if there was a key-value pair for the given key in the map.

Implementation

map.cm, page 2

2.5.5.3.20 Swap(Map<Key, Value, KeyCompare>&) Member Function

Exchanges the contents with another map of the same type.

Syntax

```
public inline nothrow void Swap(Map<Key, Value, KeyCompare>& that);
```

Parameters

Name	Type	Description
that	Map<Key, Value, KeyCompare>&	A map to exchange contents with.

Implementation

map.cm, page 3

2.5.5.4 Nonmember Functions

Function	Description
operator==<Key, Value, KeyCompare>(const Map<Key, Value, KeyCompare>&, const Map<Key, Value, KeyCompare>&)	Compares two maps for equality and returns true if both maps contain the same number of pairwise equal elements, false otherwise.
operator<<Key, Value, KeyCompare>(const Map<Key, Value, KeyCompare>&, const Map<Key, Value, KeyCompare>&)	Compares two maps for less than relationship and returns true if the first map comes lexicographically before the second map, false otherwise.

2.5.5.4.1 operator==<Key, Value, KeyCompare>(const Map<Key, Value, KeyCompare>&, const Map<Key, Value, KeyCompare>&) Function

Compares two maps for equality and returns true if both maps contain the same number of pairwise equal elements, false otherwise.

Syntax

```
public nothrow bool operator==<Key, Value, KeyCompare>(const Map<Key, Value,
KeyCompare>& left, const Map<Key, Value, KeyCompare>& right);
```

Constraint

where Key is Regular and Value is Regular and KeyCompare is Relation and KeyCompare.Domain is Key;

Parameters

Name	Type	Description
left	const Map<Key, Value, KeyCompare>&	The first map to compare.
right	const Map<Key, Value, KeyCompare>&	The second map to compare.

Returns

bool

Returns true if both maps contain the same number of pairwise equal elements, false otherwise.

Example

```
using System;
using System.Collections;

//  Writes:
//  m1 == m2
//  m1 != m3
//  m1 != m4

void main()
{
    Map<int, string> m1;
    m1[0] = "foo";
    m1[1] = "bar";
    m1[2] = "baz";
    Map<int, string> m2;
    m2[0] = "foo";
    m2[1] = "bar";
    m2[2] = "baz";
    if (m1 == m2)    // same number of pairwise equal elements
    {
        Console.Out() << "m1 == m2" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    Map<int, string> m3;
    m3[0] = "foo";
    m3[1] = "bar";
```

```
    m3[2] = "fluffy";
    if (m1 != m3) //    third element differ
    {
        Console.Out() << "m1 != m3" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    Map<int, string> m4;
    m4[0] = "foo";
    m4[1] = "bar";
    if (m1 != m4)   // different number of elements
    {
        Console.Out() << "m1 != m4" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

map.cm, page 3

2.5.5.4.2 operator<<Key, Value, KeyCompare>(const Map<Key, Value, KeyCompare>&, const Map<Key, Value, KeyCompare>&) Function

Compares two maps for less than relationship and returns true if the first map comes lexicographically before the second map, false otherwise.

Syntax

```
public nothrow bool operator<<Key, Value, KeyCompare>(const Map<Key, Value, KeyCompare>&
left, const Map<Key, Value, KeyCompare>& right);
```

Constraint

where Key is Semiregular and Value is TotallyOrdered and KeyCompare is Relation and KeyCompare.Domain is Key;

Parameters

Name	Type	Description
left	const Map<Key, Value, KeyCompare>&	The first map.
right	const Map<Key, Value, KeyCompare>&	The second map.

Returns

bool

Returns true if the first map comes lexicographically before the second map, false otherwise.

Example

```
using System;
using System.Collections;

//  Writes:
//  !(m1 < m2) && !(m2 < m1) => m1 == m2
//  m1 < m3
//  m4 < m1

void main()
{
    Map<int, string> m1;
    m1[0] = "foo";
    m1[1] = "bar";
    m1[2] = "baz";
    Map<int, string> m2;
    m2[0] = "foo";
    m2[1] = "bar";
    m2[2] = "baz";
    if (m1 < m2)
    {
        Console.Error() << "bug" << endl();
    }
    else if (m2 < m1)
    {
        Console.Error() << "bug" << endl();
    }
    else if (m1 != m2)
    {
        Console.Error() << "bug" << endl();
    }
    else
    {
        Console.Out() << "!(m1 < m2) && !(m2 < m1) => m1 == m2" << endl()
            ;
    }
    Map<int, string> m3;
    m3[0] = "foo";
    m3[1] = "bar";
    m3[2] = "fluffy";
    if (m1 < m3) //    third element of m1 is less than third element of
        m3
    {
```

```
        Console.Out() << "m1 < m3" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    Map<int, string> m4;
    m4[0] = "foo";
    m4[1] = "bar";
    if (m1 < m4)
    {
        Console.Error() << "bug" << endl();
    }
    else if (m4 < m1)    // m4[0] == m1[0] && m4[1] == m1[1], but m4 has
        fewer elements
    {
        Console.Out() << "m4 < m1" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

map.cm, page 3

2.5.6 Queue<T> Class

A first-in first-out data structure.

Syntax

```
public class Queue<T>;
```

Constraint

where T is Semiregular;

2.5.6.1 Example

```
using System;
using System.Collections;
using Simulation;

// Writes:
// clock: 2: customer 1 arrives
// clock: 7: customer 1 leaves
// clock: 15: customer 2 arrives
// clock: 20: customer 2 leaves
// clock: 25: customer 3 arrives
// clock: 30: customer 3 leaves
// clock: 31: customer 4 arrives
// clock: 36: customer 4 leaves
// clock: 46: customer 5 arrives
// clock: 51: customer 5 leaves
// clock: 56: customer 6 arrives
// clock: 61: customer 6 leaves
// clock: 70: customer 7 arrives
// clock: 75: customer 7 leaves
// clock: 84: customer 8 arrives
// clock: 89: customer 8 leaves
// clock: 92: customer 9 arrives
// clock: 97: customer 9 leaves
// clock: 102: customer 10 arrives
// clock: 107: customer 10 leaves
// clock: 107: end of simulation.

namespace Simulation
{
    public class CustomerEvent
    {
        public CustomerEvent(): elapsed(0), customerNumber(0)
        {
        }
        public CustomerEvent(int elapsed_, int customerNumber_): elapsed(
            elapsed_), customerNumber(customerNumber_)
        {
        }
```

```
        public int Elapsed() const
        {
            return elapsed;
        }
        public int CustomerNumber() const
        {
            return customerNumber;
        }
        private int elapsed;
        private int customerNumber;
    }
    public typedef Queue<CustomerEvent> CustomerEventQueue;
}

public const int serviceTime = 5;

void main()
{
    CustomerEventQueue queue;
    int customerNumber = 1;
    int n = 10;
    for (int i = 0; i < n; ++i)
    {
        queue.Put(CustomerEvent(rand() % 10 + 1, customerNumber++));
    }
    int clock = 0;
    while (!queue.IsEmpty())
    {
        CustomerEvent event = queue.Get();
        clock = clock + event.Elapsed();
        Console.Out() << "clock: " << clock << ": customer " << event.
           CustomerNumber() << " arrives" << endl();
        clock  = clock + serviceTime;
        Console.Out() << "clock: " << clock << ": customer " << event.
           CustomerNumber() << " leaves" << endl();
    }
    Console.Out() << "clock: " << clock << ": end of simulation." << endl
       ();
}
```

2.5.6.2 Type Definitions

Name	Type	Description
ValueType	T	The type of the queue element.

2.5.6.3 Member Functions

Member Function	Description
Queue()	Default constructor. Constructs an empty queue.

Queue(Queue<T>&&)	Move constructor.
operator=(Queue<T>&&)	Move assignment.
Clear()	Makes the queue empty.
Count() const	Returns the number of elements in the queue.
Front() const	Returns a constant reference to the first element in the queue.
Get()	Removes the first element from the queue and returns it.
IsEmpty() const	Returns true if the queue is empty, false otherwise.
Put(ValueType&&)	Moves an element to the back of the queue.
Put(const ValueType&)	Puts an element to the back of the queue.

2.5.6.3.1 Queue() Member Function

Default constructor. Constructs an empty queue.

Syntax

```
public default Queue();
```

Implementation

queue.cm, page 1

2.5.6.3.2 Queue(Queue<T>&&) Member Function

Move constructor.

Syntax

```
public default Queue(Queue<T>&& __parameter1);
```

Parameters

Name	Type	Description
__parameter1	Queue<T>&&	A queue to move from.

Implementation

queue.cm, page 1

2.5.6.3.3 operator=(Queue<T>&&) Member Function

Move assignment.

Syntax

```
public default void operator=(Queue<T>&& __parameter1);
```

Parameters

Name	Type	Description
__parameter1	Queue<T>&&	A queue to move from.

Implementation

queue.cm, page 1

2.5.6.3.4 Clear() Member Function

Makes the queue empty.

Syntax

```
public nothrow void Clear();
```

Implementation

queue.cm, page 2

2.5.6.3.5 Count() const Member Function

Returns the number of elements in the queue.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of elements in the queue.

Implementation

queue.cm, page 1

2.5.6.3.6 Front() const Member Function

Returns a constant reference to the first element in the queue.

Syntax

```
public inline nothrow const ValueType& Front() const;
```

Returns

const ValueType&

Returns a constant reference to the first element in the queue.

Implementation

queue.cm, page 2

2.5.6.3.7 Get() Member Function

Removes the first element from the queue and returns it.

Syntax

```
public inline ValueType Get();
```

Returns

ValueType

Returns the removed first element of the queue.

Implementation

queue.cm, page 1

2.5.6.3.8 IsEmpty() const Member Function

Returns true if the queue is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the queue is empty, false otherwise.

Implementation

queue.cm, page 1

2.5.6.3.9 Put(ValueType&&) Member Function

Moves an element to the back of the queue.

Syntax

```
public inline void Put(ValueType&& item);
```

Parameters

Name	Type	Description
item	ValueType&&	An element to insert.

Implementation

queue.cm, page 1

2.5.6.3.10 Put(const ValueType&) Member Function

Puts an element to the back of the queue.

Syntax

```
public inline void Put(const ValueType& item);
```

Parameters

Name	Type	Description
item	const ValueType&	An element to put.

Implementation

queue.cm, page 1

2.5.7 RedBlackTree<KeyType, ValueType, KeyOfValue, Compare> Class

A self-balancing binary search tree of unique elements used to implement Set<T, C> and Map<Key, Value, KeyCompare>. The keys of the elements in the tree need to be ordered.

Syntax

```
public class RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>;
```

Constraint

where KeyType is Semiregular and ValueType is Semiregular and KeySelectionFunction<KeyOfValue, KeyType, ValueType> and Compare is Relation and Compare.Domain is KeyType;

2.5.7.1 Type Definitions

Name	Type	Description
ConstIterator	RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType*>	A constant iterator type.
Iterator	RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>	An iterator type.

2.5.7.2 Member Functions

Member Function	Description
RedBlackTree()	Default constructor. Constructs an empty red-black tree.
RedBlackTree(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&)	Move constructor.
RedBlackTree(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&)	Copy constructor.
operator=(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&)	Copy assignment.
operator=(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&)	Move assignment.
~RedBlackTree()	Destructor.
Begin()	Returns a bidirectional iterator pointing to the beginning of the red-black tree.

Begin() const	Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.
CBegin() const	Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.
CEnd() const	Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.
Clear()	Makes the red-black tree empty.
Count() const	Returns the number of elements in the red-black tree.
End()	Returns a bidirectional iterator pointing to one past the end of the red-black tree.
End() const	Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.
Find(const KeyType&)	Finds an element with the given key in the red-black tree and returns an iterator pointing to it if found, or End() iterator otherwise.
Find(const KeyType&) const	Finds an element with the given key in the red-black tree and returns a constant iterator pointing to it if found, or CEnd() const iterator otherwise.
Insert(ValueType&&)	Moves the given element to the red-black tree, if an element with an equal key is not found in the tree.
Insert(const ValueType&)	Inserts an element into the red-black tree, if an element with an equal key is not found in the tree.

IsEmpty() const	Returns true if the red-black tree is empty, false otherwise.
Remove(Iterator)	Removes an element pointed by the given iterator from the red-black tree.
Remove(const KeyType&)	Removes an element with the given key from the red-black tree. If an element with the given is not found, does nothing.
Swap(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&)	Exchanges the contents with another red-black tree of the same type.

2.5.7.2.1 RedBlackTree() Member Function

Default constructor. Constructs an empty red-black tree.

Syntax

```
public RedBlackTree();
```

Implementation

rbtree.cm, page 12

2.5.7.2.2 RedBlackTree(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&) Member Function

Move constructor.

Syntax

```
public default nothrow RedBlackTree(RedBlackTree<KeyType, ValueType, KeyOfValue,
Compare>&& that);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
that	RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&	A red-black tree to move from.

Implementation

rbtree.cm, page 12

2.5.7.2.3 RedBlackTree(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&) Member Function

Copy constructor.

Syntax

```
public RedBlackTree(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&
that);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description
that	const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&	A red-black tree to copy.

Implementation

rbtree.cm, page 12

2.5.7.2.4 operator=(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&) Member Function

Copy assignment.

Syntax

```
public void operator=(const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&
that);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description
that	const RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&	A red-black tree to assign.

Implementation

rbtree.cm, page 12

2.5.7.2.5 operator=(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&) Member Function

Move assignment.

Syntax

```
public default nothrow void operator=(RedBlackTree<KeyType, ValueType, KeyOfValue,
Compare>&& that);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
that	RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&	A red-black tree to move from.

Implementation

rbtree.cm, page 12

2.5.7.2.6 ~RedBlackTree() Member Function

Destructor.

Syntax

```
public nothrow ~RedBlackTree();
```

Implementation

rbtree.cm, page 12

2.5.7.2.7 Begin() Member Function

Returns a bidirectional iterator pointing to the beginning of the red-black tree.

Syntax

```
public nothrow Iterator Begin();
```

Returns

Iterator

Returns a bidirectional iterator pointing to the beginning of the red-black tree.

Implementation

rbtree.cm, page 12

2.5.7.2.8 Begin() const Member Function

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

Syntax

```
public nothrow ConstIterator Begin() const;
```

Returns

ConstIterator

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

Implementation

rbtree.cm, page 12

2.5.7.2.9 CBegin() const Member Function

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

Syntax

```
public nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

Implementation

rbtree.cm, page 12

2.5.7.2.10 CEnd() const Member Function

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

Syntax

```
public nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

Implementation

rbtree.cm, page 13

2.5.7.2.11 Clear() Member Function

Makes the red-black tree empty.

Syntax

```
public nothrow void Clear();
```

Implementation

rbtree.cm, page 13

2.5.7.2.12 Count() const Member Function

Returns the number of elements in the red-black tree.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of elements in the red-black tree.

Implementation

rbtree.cm, page 13

2.5.7.2.13 End() Member Function

Returns a bidirectional iterator pointing to one past the end of the red-black tree.

Syntax

```
public nothrow Iterator End();
```

Returns

Iterator

Returns a bidirectional iterator pointing to one past the end of the red-black tree.

Implementation

rbtree.cm, page 13

2.5.7.2.14 End() const Member Function

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

Syntax

```
public nothrow ConstIterator End() const;
```

Returns

ConstIterator

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

Implementation

rbtree.cm, page 12

2.5.7.2.15 Find(const KeyType&) Member Function

Finds an element with the given key in the red-black tree and returns an iterator pointing to it if found, or End() iterator otherwise.

Syntax

```
public nothrow Iterator Find(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	A key to search.

Returns

Iterator

Returns an iterator pointing to the found element if search is successful, or End() iterator otherwise.

Implementation

rbtree.cm, page 13

2.5.7.2.16 Find(const KeyType&) const Member Function

Finds an element with the given key in the red-black tree and returns a constant iterator pointing to it if found, or CEnd() const iterator otherwise.

Syntax

```
public nothrow ConstIterator Find(const KeyType& key) const;
```

Parameters

Name	Type	Description
key	const KeyType&	A key to search.

Returns

ConstIterator

Returns a constant iterator pointing to the found element if search is successful, or CEnd() const iterator otherwise.

Implementation

rbtree.cm, page 14

2.5.7.2.17 Insert(ValueType&&) Member Function

Moves the given element to the red-black tree, if an element with an equal key is not found in the tree.

Syntax

```
public Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>
Insert(ValueType&& value);
```

Constraint

where ValueType is Movable;

Parameters

Name	Type	Description
value	ValueType&&	An element to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting of an iterator pointing to the inserted element and **true** if an element was inserted, or a pair consisting of an iterator to an existing element and **false** otherwise.

Implementation

rbtree.cm, page 15

2.5.7.2.18 Insert(const ValueType&) Member Function

Inserts an element into the red-black tree, if an element with an equal key is not found in the tree.

Syntax

```
public Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>
Insert(const ValueType& value);
```

Constraint

where ValueType is Copyable;

Parameters

Name	Type	Description
value	const ValueType&	An element to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting of an iterator pointing to the inserted element and **true** if an element was inserted, or a pair consisting of an iterator to an existing element and **false** otherwise.

Implementation

rbtree.cm, page 14

2.5.7.2.19 IsEmpty() const Member Function

Returns true if the red-black tree is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the red-black tree is empty, false otherwise.

Implementation

rbtree.cm, page 13

2.5.7.2.20 Remove(Iterator) Member Function

Removes an element pointed by the given iterator from the red-black tree.

Syntax

```
public nothrow void Remove(Iterator pos);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator pointing to the element to be removed.

Implementation

rbtree.cm, page 17

2.5.7.2.21 Remove(const KeyType&) Member Function

Removes an element with the given key from the red-black tree. If an element with the given is not found, does nothing.

Syntax

```
public nothrow bool Remove(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	A key of an element to remove.

Returns

bool

Returns true if an element was removed, false otherwise.

Implementation

rbtree.cm, page 16

2.5.7.2.22 Swap(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&) Member Function

Exchanges the contents with another red-black tree of the same type.

Syntax

```
public inline nothrow void Swap(RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&
that);
```

Parameters

Name	Type	Description
that	RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&	A red-black tree to exchange contents with.

Implementation

rbtree.cm, page 17

2.5.8 RedBlackTreeNodeIterator<T, R, P> Class

A bidirectional iterator that iterates through the elements in a reb-black tree.

Syntax

```
public class RedBlackTreeNodeIterator<T, R, P>;
```

Model of

BidirectionalIterator<T>

2.5.8.1 Type Definitions

Name	Type	Description
PointerType	P	The type of a pointer to an element.
ReferenceType	R	The type of a reference to an element.
ValueType	T	The type of the element.

2.5.8.2 Member Functions

Member Function	Description
RedBlackTreeNodeIterator()	Constructor. Default constructs a red-black tree node iterator.
RedBlackTreeNodeIterator(RedBlackTreeNode<T>*)	Constructor. Constructs an iterator pointing to a red-black tree node.
operator–()	Backs the iterator pointing to the previous element in the red-black tree.
operator++()	Advances the iterator pointing to the next element in the red-black tree.
operator->() const	Returns a pointer to an element.
operator*() const	Returns a reference to an element.
GetNode() const	Returns a pointer to a red-black tree node.

2.5.8.2.1 RedBlackTreeNodeIterator() Member Function

Constructor. Default constructs a red-black tree node iterator.

Syntax

```
public nothrow RedBlackTreeNodeIterator();
```

Implementation

rbtree.cm, page 11

2.5.8.2.2 RedBlackTreeNodeIterator(RedBlackTreeNode<T>*) Member Function

Constructor. Constructs an iterator pointing to a red-black tree node.

Syntax

```
public nothrow RedBlackTreeNodeIterator(RedBlackTreeNode<T>* node_);
```

Parameters

Name	Type	Description
node_	RedBlackTreeNode<T>*	A pointer to a red-black tree node.

Implementation

rbtree.cm, page 11

2.5.8.2.3 operator–() Member Function

Backs the iterator pointing to the previous element in the red-black tree.

Syntax

```
public nothrow RedBlackTreeNodeIterator<T, R, P>& operator--();
```

Returns

RedBlackTreeNodeIterator<T, R, P>&

Returns a reference to the iterator.

Implementation

rbtree.cm, page 11

2.5.8.2.4 operator++() Member Function

Advances the iterator pointing to the next element in the red-black tree.

Syntax

```
public nothrow RedBlackTreeNodeIterator<T, R, P>& operator++();
```

Returns

RedBlackTreeNodeIterator<T, R, P>&

Returns a reference to the iterator.

Implementation

rbtree.cm, page 11

2.5.8.2.5 operator->() const Member Function

Returns a pointer to an element.

Syntax

```
public nothrow PointerType operator->() const;
```

Returns

PointerType

Returns a pointer to an element.

Implementation

rbtree.cm, page 11

2.5.8.2.6 operator*() const Member Function

Returns a reference to an element.

Syntax

```
public nothrow ReferenceType operator*() const;
```

Returns

ReferenceType

Returns a reference to an element.

Implementation

rbtree.cm, page 11

2.5.8.2.7 GetNode() const Member Function

Returns a pointer to a red-black tree node.

Syntax

```
public inline nothrow RedBlackTreeNode<T>* GetNode() const;
```

Returns

RedBlackTreeNode<T>*

Returns a pointer to a red-black tree node.

Implementation

rbtree.cm, page 11

2.5.8.3 Nonmember Functions

Function	Description
operator==<T, R, P>(const RedBlackTreeNodeIterator<T, R, P>&, const RedBlackTreeNodeIterator<T, R, P>&)	Compares two red-black tree node iterators for equality.

2.5.8.3.1 operator==<T, R, P>(const RedBlackTreeNodeIterator<T, R, P>&, const RedBlackTreeNodeIterator<T, R, P>&) Function

Compares two red-black tree node iterators for equality.

Syntax

```
public inline nothrow bool operator==<T, R, P>(const RedBlackTreeNodeIterator<T,
R, P>& left, const RedBlackTreeNodeIterator<T, R, P>& right);
```

Parameters

Name	Type	Description
left	const RedBlackTreeNodeIterator<T, R, P>&	The first red-black tree node iterator.
right	const RedBlackTreeNodeIterator<T, R, P>&	The second red-black tree node iterator.

Returns

bool

Returns true if both iterators point to same red-black tree node, or both are End() iterators, false otherwise.

Implementation

rbtree.cm, page 11

2.5.9 Set<T, C> Class

A container that contains a set of unique elements organized in a red-black tree. The elements need to be ordered.

Syntax

```
public class Set<T, C>;
```

Constraint

where T is Semiregular and C is Relation and C.Domain is T;

Model of

BidirectionalContainer<T>

Default Template Arguments

C = Less<T>

2.5.9.1 Example

```
using System;
using System.Collections;

//  Writes:
//  10, 43, 112
//  112 already exists
//  i1 points to item number 0
//  15 not found
//  43 removed
//  112

void main()
{
    Set<int> set;
    set.Insert(43);
    set.Insert(10);
    set.Insert(112);
    Console.Out() << set << endl();
    if (!set.Insert(112).second)
    {
        Console.Out() << 112 << " already exists" << endl();
    }
    Set<int>.Iterator i1 = set.Find(10);
    if (i1 != set.End())
    {
        Console.Out() << "i1 points to item number " << Distance(set.
            Begin(), i1) << endl();
    }
```

```
    else
    {
        Console.Error() << "bug" << endl();
    }
    Set<int>.Iterator i2 = set.Find(15);
    if (i2 == set.End())
    {
        Console.Out() << 15 << " not found" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    bool removed = set.Remove(43);
    if (removed)
    {
        Console.Out() << 43 << " removed" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    set.Remove(i1);
    Console.Out() << set << endl();
}
```

2.5.9.2 Type Definitions

Name	Type	Description
Compare	C	A relation used to order elements in the set.
ConstIterator	RedBlackTreeNodeIterator<T, const T&, const T*>	A constant iterator type.
Iterator	RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>	An iterator type.
KeyType	T	The key type is equal to the ValueType for the red-black tree.
ValueType	T	The type of the element in the set.

2.5.9.3 Member Functions

Member Function	Description

Set()	Default constructor. Constructs an empty set.
Set(const Set<T, C>&)	Copy constructor.
Set(Set<T, C>&&)	Move constructor.
operator=(const Set<T, C>&)	Copy assignment.
operator=(Set<T, C>&&)	Move assignment.
~Set()	Destructor.
Begin()	Returns an iterator pointing to the beginning of the set.
CBegin() const	Returns a constant iterator pointing to the beginning of the set.
CEnd() const	Returns a constant iterator pointing one past the end of the set.
Clear()	Makes the set empty.
Count() const	Returns the number of elements in the set.
End()	Returns an iterator pointing one past the end of the set.
Find(const KeyType&)	Searches an element in the set and returns an iterator pointing to it if found, or End() iterator otherwise.
Insert(const KeyType&)	Inserts an element into the set, if it is not already there.
Insert(KeyType&&)	Moves an element into the set, if it is not already there.
IsEmpty() const	Returns true if the set is empty, false otherwise.

Remove(Iterator)	Removes an element pointed by the specified iterator from the set.
Remove(const KeyType&)	Removes an element from the set. If the element was not found, does nothing.
Swap(Set<T, C>&)	Exchanges the contents with another set of the same type.

2.5.9.3.1 Set() Member Function

Default constructor. Constructs an empty set.

Syntax

```
public Set();
```

Implementation

set.cm, page 1

2.5.9.3.2 Set(const Set<T, C>&) Member Function

Copy constructor.

Syntax

```
public default Set(const Set<T, C>& that);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
that	const Set<T, C>&	A set to copy.

Implementation

set.cm, page 1

2.5.9.3.3 Set(Set<T, C>&&) Member Function

Move constructor.

Syntax

```
public default nothrow Set(Set<T, C>&& that);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
that	Set<T, C>&&	A set to move from.

Implementation

set.cm, page 1

2.5.9.3.4 operator=(const Set<T, C>&) Member Function

Copy assignment.

Syntax

```
public default void operator=(const Set<T, C>& that);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
that	const Set<T, C>&	A set to assign.

Implementation

set.cm, page 1

2.5.9.3.5 operator=(Set<T, C>&&) Member Function

Move assignment.

Syntax

```
public default nothrow void operator=(Set<T, C>&& that);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
that	Set<T, C>&&	A set to move from.

Implementation

set.cm, page 1

2.5.9.3.6 ~Set() Member Function

Destructor.

Syntax

```
public default nothrow ~Set();
```

Implementation

set.cm, page 1

2.5.9.3.7 Begin() Member Function

Returns an iterator pointing to the beginning of the set.

Syntax

```
public inline nothrow Iterator Begin();
```

Returns

Iterator

Returns an iterator pointing to the beginning of the set.

Implementation

set.cm, page 1

2.5.9.3.8 CBegin() const Member Function

Returns a constant iterator pointing to the beginning of the set.

Syntax

```
public inline nothrow ConstIterator CBegin() const;
```

Returns

ConstIterator

Returns a constant iterator pointing to the beginning of the set.

Implementation

set.cm, page 1

2.5.9.3.9 CEnd() const Member Function

Returns a constant iterator pointing one past the end of the set.

Syntax

```
public inline nothrow ConstIterator CEnd() const;
```

Returns

ConstIterator

Returns a constant iterator pointing one past the end of the set.

Implementation

set.cm, page 2

2.5.9.3.10 Clear() Member Function

Makes the set empty.

Syntax

```
public nothrow void Clear();
```

Implementation

set.cm, page 2

2.5.9.3.11 Count() const Member Function

Returns the number of elements in the set.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of elements in the set.

Implementation

set.cm, page 2

2.5.9.3.12 End() Member Function

Returns an iterator pointing one past the end of the set.

Syntax

```
public inline nothrow Iterator End();
```

Returns

Iterator

Returns an iterator pointing one past the end of the set.

Implementation

set.cm, page 2

2.5.9.3.13 Find(const KeyType&) Member Function

Searches an element in the set and returns an iterator pointing to it if found, or End() iterator otherwise.

Syntax

```
public inline nothrow Iterator Find(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	An element to seach.

Returns

Iterator

Returns an iterator pointing to the found element if the search was successful, or End() iterator otherwise.

Implementation

set.cm, page 2

2.5.9.3.14 Insert(const KeyType&) Member Function

Inserts an element into the set, if it is not already there.

Syntax

```
public inline Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>,
bool> Insert(const KeyType& value);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
value	const KeyType&	An element to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting of an iterator pointing to the inserted element and **true** if the element was inserted, or a pair consisting an iterator pointing to an existing element and **false** otherwise.

Implementation

set.cm, page 2

2.5.9.3.15 Insert(KeyType&&) Member Function

Moves an element into the set, if it is not already there.

Syntax

```
public inline Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>,
bool> Insert(KeyType&& value);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
value	KeyType&&	An element to insert.

Returns

Pair<RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>, bool>

Returns a pair consisting of an iterator pointing to the inserted element and **true** if the element was inserted, or a pair consisting an iterator pointing to an existing element and **false** otherwise.

Implementation

set.cm, page 2

2.5.9.3.16 IsEmpty() const Member Function

Returns true if the set is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the set is empty, false otherwise.

Implementation

set.cm, page 2

2.5.9.3.17 Remove(Iterator) Member Function

Removes an element pointed by the specified iterator from the set.

Syntax

```
public inline nothrow void Remove(Iterator pos);
```

Parameters

Name	Type	Description
pos	Iterator	An iterator pointing to an element to be removed.

Implementation

set.cm, page 2

2.5.9.3.18 Remove(const KeyType&) Member Function

Removes an element from the set. If the element was not found, does nothing.

Syntax

```
public inline nothrow bool Remove(const KeyType& key);
```

Parameters

Name	Type	Description
key	const KeyType&	An element to remove.

Returns

bool

Returns true if element was removed, false otherwise.

Implementation

set.cm, page 2

2.5.9.3.19 Swap(Set<T, C>&) Member Function

Exchanges the contents with another set of the same type.

Syntax

```
public inline nothrow void Swap(Set<T, C>& that);
```

Parameters

Name	Type	Description
that	Set<T, C>&	A set to exchange contents with.

Implementation

set.cm, page 2

2.5.9.4 Nonmember Functions

Function	Description
operator==<T, C>(const Set<T, C>&, const Set<T, C>&)	Compares two sets for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.
operator<<T, C>(const Set<T, C>&, const Set<T, C>&)	Compares two sets for less than relationship and returns true if the first set comes lexicographically before the second set, false otherwise.

2.5.9.4.1 operator==<T, C>(const Set<T, C>&, const Set<T, C>&) Function

Compares two sets for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.

Syntax

```
public inline nothrow bool operator==<T, C>(const Set<T, C>& left, const Set<T,
C>& right);
```

Constraint

where T is Regular and C is Relation and C.Domain is T;

Parameters

Name	Type	Description
left	const Set<T, C>&	The first set.
right	const Set<T, C>&	The second set.

Returns

bool

Returns true if both contain the same number of pairwise equal elements, false otherwise.

Example

```
using System;
using System.Collections;

//  Writes:
//  s1 == s2
//  s1 != s3
//  s1 != s4

void main()
{
    Set<string> s1;
    s1.Insert("foo");
    s1.Insert("bar");
    s1.Insert("baz");
    Set<string> s2;
    s2.Insert("bar");
    s2.Insert("foo");
    s2.Insert("baz");
    if (s1 == s2)   // same number of pairwise equal elements
    {
        Console.Out() << "s1 == s2" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    Set<string> s3;
    s3.Insert("foo");
    s3.Insert("bar");
    s3.Insert("fluffy");
    if (s1 != s3) //    third element differ
    {
        Console.Out() << "s1 != s3" << endl();
    }
```

```
    else
    {
        Console.Error() << "bug" << endl();
    }
    Set<string> s4;
    s4.Insert("foo");
    s4.Insert("bar");
    if (s1 != s4)   // different number of elements
    {
        Console.Out() << "s1 != s4" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

set.cm, page 2

2.5.9.4.2 operator<<T, C>(const Set<T, C>&, const Set<T, C>&) Function

Compares two sets for less than relationship and returns true if the first set comes lexicographically before the second set, false otherwise.

Syntax

```
public inline nothrow bool operator<<T, C>(const Set<T, C>& left, const Set<T, C>& right);
```

Constraint

where T is Semiregular and C is Relation and C.Domain is T;

Parameters

Name	Type	Description
left	const Set<T, C>&	The first set.
right	const Set<T, C>&	The second set.

Returns

bool

Returns true if the first set comes lexicographically before the second set, false otherwise.

Example

```
using System;
using System.Collections;

// Writes:
// !(s1 < s2) && !(s2 < s1) => s1 == s2
// s1 < s3
// s4 < s1

void main()
{
    Set<string> s1;
    s1.Insert("foo");
    s1.Insert("bar");
    s1.Insert("baz");
    Set<string> s2;
    s2.Insert("foo");
    s2.Insert("bar");
    s2.Insert("baz");
    if (s1 < s2)
    {
        Console.Error() << "bug" << endl();
    }
    else if (s2 < s1)
    {
        Console.Error() << "bug" << endl();
    }
    else if (s1 != s2)
    {
        Console.Error() << "bug" << endl();
    }
    else
    {
        Console.Out() << "!(s1 < s2) && !(s2 < s1) => s1 == s2" << endl()
            ;
    }
    Set<string> s3;
    s3.Insert("foo");
    s3.Insert("bar");
    s3.Insert("fluffy");
    if (s1 < s3) //    third element of s1 is less than third element of
        s3
    {
        Console.Out() << "s1 < s3" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
    Set<string> s4;
    s4.Insert("bar");
    s4.Insert("baz");
```

```
    if (s1 < s4)
    {
        Console.Error() << "bug" << endl();
    }
    else if (s4 < s1)    // s4[0] == s1[0] && s4[1] == s1[1], but s4 has
        fewer elements
    {
        Console.Out() << "s4 < s1" << endl();
    }
    else
    {
        Console.Error() << "bug" << endl();
    }
}
```

Implementation

set.cm, page 3

2.5.10 Stack<T> Class

A last-in first-out data structure.

Syntax

```
public class Stack<T>;
```

Constraint

where T is Semiregular;

2.5.10.1 Example

```
using System;
using System.Collections;

//  Writes:
//  foo at the top
//  bar at the top
//  baz at the top
//  baz popped, 2 items in the stack
//  bar popped, 1 items in the stack
//  foo popped, 0 items in the stack

void main()
{
    Stack<string> stack;
    stack.Push("foo");
    Console.Out() << stack.Top() << " at the top" << endl();
    stack.Push("bar");
    Console.Out() << stack.Top() << " at the top" << endl();
    stack.Push("baz");
    Console.Out() << stack.Top() << " at the top" << endl();
    while (!stack.IsEmpty())
    {
        string popped = stack.Pop();
        Console.Out() << popped << " popped, " << stack.Count() << "
            items in the stack" << endl();
    }
}
```

2.5.10.2 Type Definitions

Name	Type	Description
ValueType	T	The type of the stack element.

2.5.10.3 Member Functions

Member Function	Description
Stack()	Default constructor. Constructs an empty stack.
Stack(Stack<T>&&)	Move constructor.
operator=(Stack<T>&&)	Move assignment.
Clear()	Makes the stack empty.
Count() const	Returns the number of elements in the stack.
IsEmpty() const	Returns true if the stack is empty, false otherwise.
Pop()	Removes an element from the top of the stack and returns it.
Push(ValueType&&)	Moves an element to the top of the stack.
Push(const ValueType&)	Adds an element to the top of the stack.
Top()	Return a reference to the element at the top of the stack.
Top() const	Returns a constant reference to the element at the top of the stack.

2.5.10.3.1 Stack() Member Function

Default constructor. Constructs an empty stack.

Syntax

```
public default Stack();
```

Implementation

stack.cm, page 1

2.5.10.3.2 Stack(Stack<T>&&) Member Function

Move constructor.

Syntax

```
public default Stack(Stack<T>&& __parameter1);
```

Parameters

Name	Type	Description
__parameter1	Stack<T>&&	A stack to move from.

Implementation

stack.cm, page 1

2.5.10.3.3 operator=(Stack<T>&&) Member Function

Move assignment.

Syntax

```
public default void operator=(Stack<T>&& __parameter1);
```

Parameters

Name	Type	Description
__parameter1	Stack<T>&&	A stack to move from.

Implementation

stack.cm, page 1

2.5.10.3.4 Clear() Member Function

Makes the stack empty.

Syntax

```
public nothrow void Clear();
```

Implementation

stack.cm, page 2

2.5.10.3.5 Count() const Member Function

Returns the number of elements in the stack.

Syntax

```
public inline nothrow int Count() const;
```

Returns

int

Returns the number of elements in the stack.

Implementation

stack.cm, page 1

2.5.10.3.6 IsEmpty() const Member Function

Returns true if the stack is empty, false otherwise.

Syntax

```
public inline nothrow bool IsEmpty() const;
```

Returns

bool

Returns true if the stack is empty, false otherwise.

Implementation

stack.cm, page 1

2.5.10.3.7 Pop() Member Function

Removes an element from the top of the stack and returns it.

Syntax

```
public inline ValueType Pop();
```

Constraint

where T is Movable;

Returns

ValueType

Returns the element removed from the top of the stack.

Implementation

stack.cm, page 2

2.5.10.3.8 Push(ValueType&&) Member Function

Moves an element to the top of the stack.

Syntax

```
public inline void Push(ValueType&& item);
```

Constraint

where T is Movable;

Parameters

Name	Type	Description
item	ValueType&&	An element to move.

Implementation

stack.cm, page 1

2.5.10.3.9 Push(const ValueType&) Member Function

Adds an element to the top of the stack.

Syntax

```
public inline void Push(const ValueType& item);
```

Constraint

where T is Copyable;

Parameters

Name	Type	Description
item	const ValueType&	An element to add.

Implementation

stack.cm, page 1

2.5.10.3.10 Top() Member Function

Return a reference to the element at the top of the stack.

Syntax

```
public inline nothrow ValueType& Top();
```

Returns

ValueType&

Return a reference to the element at the top of the stack.

Implementation

stack.cm, page 2

2.5.10.3.11 Top() const Member Function

Returns a constant reference to the element at the top of the stack.

Syntax

```
public inline nothrow const ValueType& Top() const;
```

Returns

const ValueType&

Returns a constant reference to the element at the top of the stack.

Implementation

stack.cm, page 2

2.6 Functions

Function	Description
ConstructiveCopy<ValueType>(ValueType*, ValueType*, int)	Copies a sequence of values by constructing them into raw memory.
ConstructiveMove<ValueType>(ValueType*, ValueType*, int)	Moves a sequence of values by moving them into raw memory.
Destroy<ValueType>(ValueType*, int)	Destroys a sequence of values but does not release the memory allocated for them.

2.6.11 ConstructiveCopy<ValueType>(ValueType*, ValueType*, int) Function

Copies a sequence of values by constructing them into raw memory.

Syntax

```
public void ConstructiveCopy<ValueType>(ValueType* to, ValueType* from, int count);
```

Constraint

where ValueType is CopyConstructible;

Parameters

Name	Type	Description
to	ValueType*	A pointer to beginning of raw memory to copy the elements to.
from	ValueType*	A pointer to elements to copy.
count	int	The number of elements to copy.

Implementation

list.cm, page 7

2.6.12 ConstructiveMove<ValueType>(ValueType*, ValueType*, int) Function

Moves a sequence of values by moving them into raw memory.

Syntax

```
public void ConstructiveMove<ValueType>(ValueType* to, ValueType* from, int count);
```

Constraint

where ValueType is MoveConstructible;

Parameters

Name	Type	Description
to	ValueType*	A pointer to beginning of raw memory to move the elements to.

from	ValueType*	A pointer to elements to move.
count	int	The number of elements to move.

Implementation

list.cm, page 7

2.6.13 Destroy<ValueType>(ValueType*, int) Function

Destroys a sequence of values but does not release the memory allocated for them.

Syntax

```
public nothrow void Destroy<ValueType>(ValueType* items, int count);
```

Constraint

where ValueType is Destructible;

Parameters

Name	Type	Description
items	ValueType*	A pointer to elements to destroy.
count	int	The number of elements to destroy.

Implementation

list.cm, page 7

3 System.Concepts Namespace

Contains system library concepts.

Figures 3.1, 3.2, 3.3, 3.4 and 3.5 contain the concepts in this namespace.

Figure 3.1: Concept Diagram 1: Basic Concepts

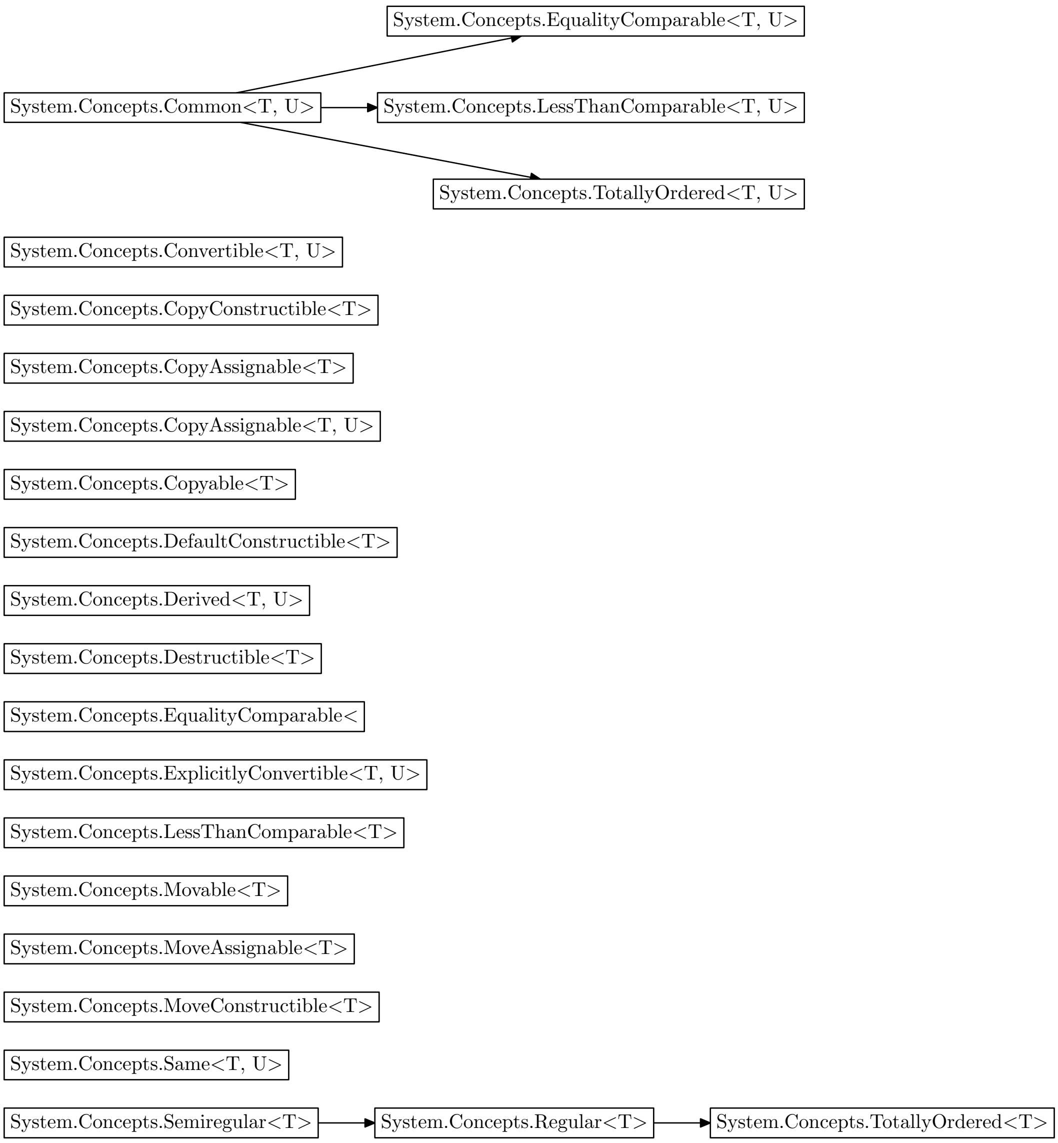


Figure 3.2: Concept Diagram 2: Iterator Concepts

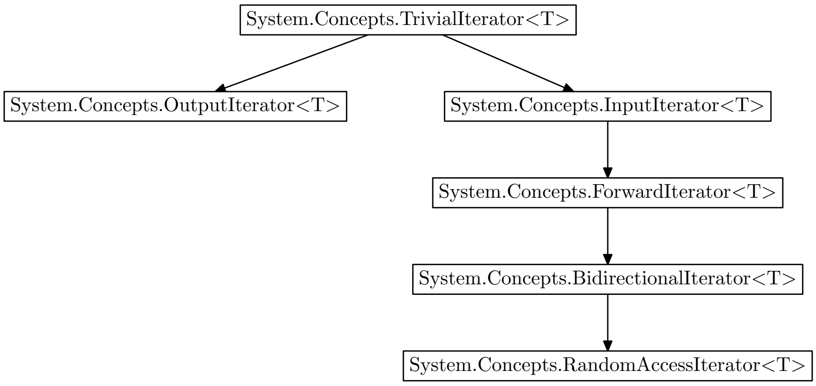


Figure 3.3: Concept Diagram 3: Container Concepts

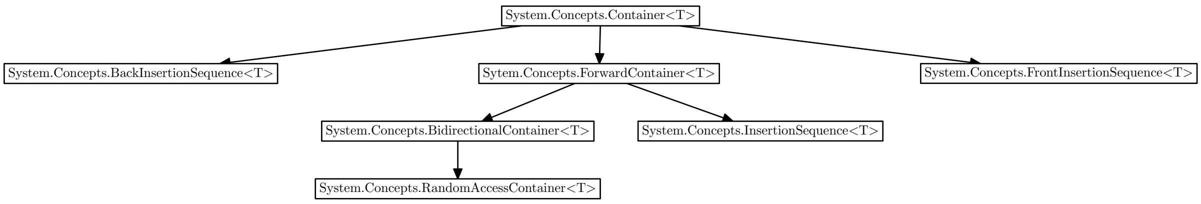


Figure 3.4: Concept Diagram 4: Functional Concepts

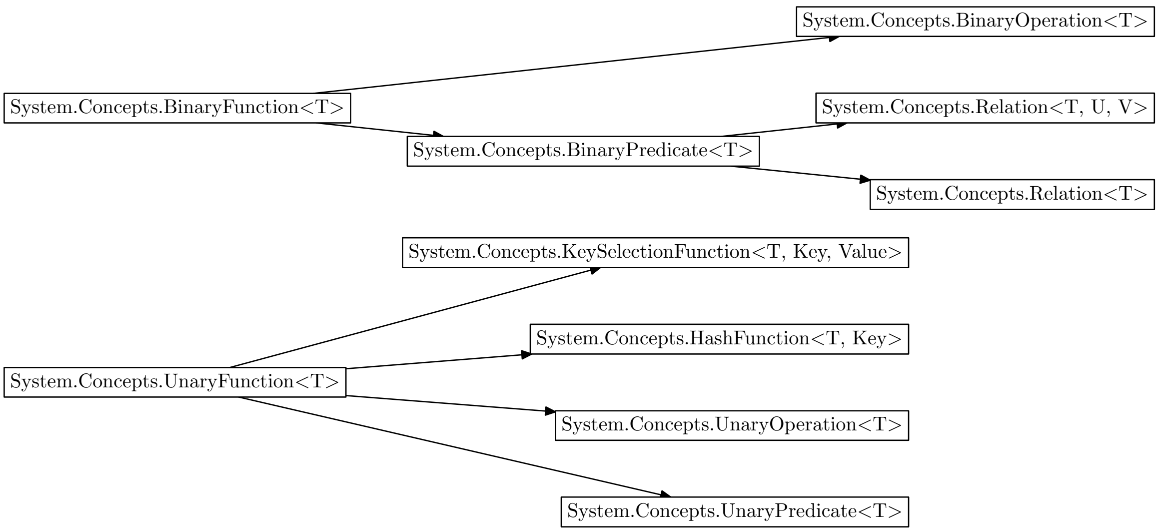
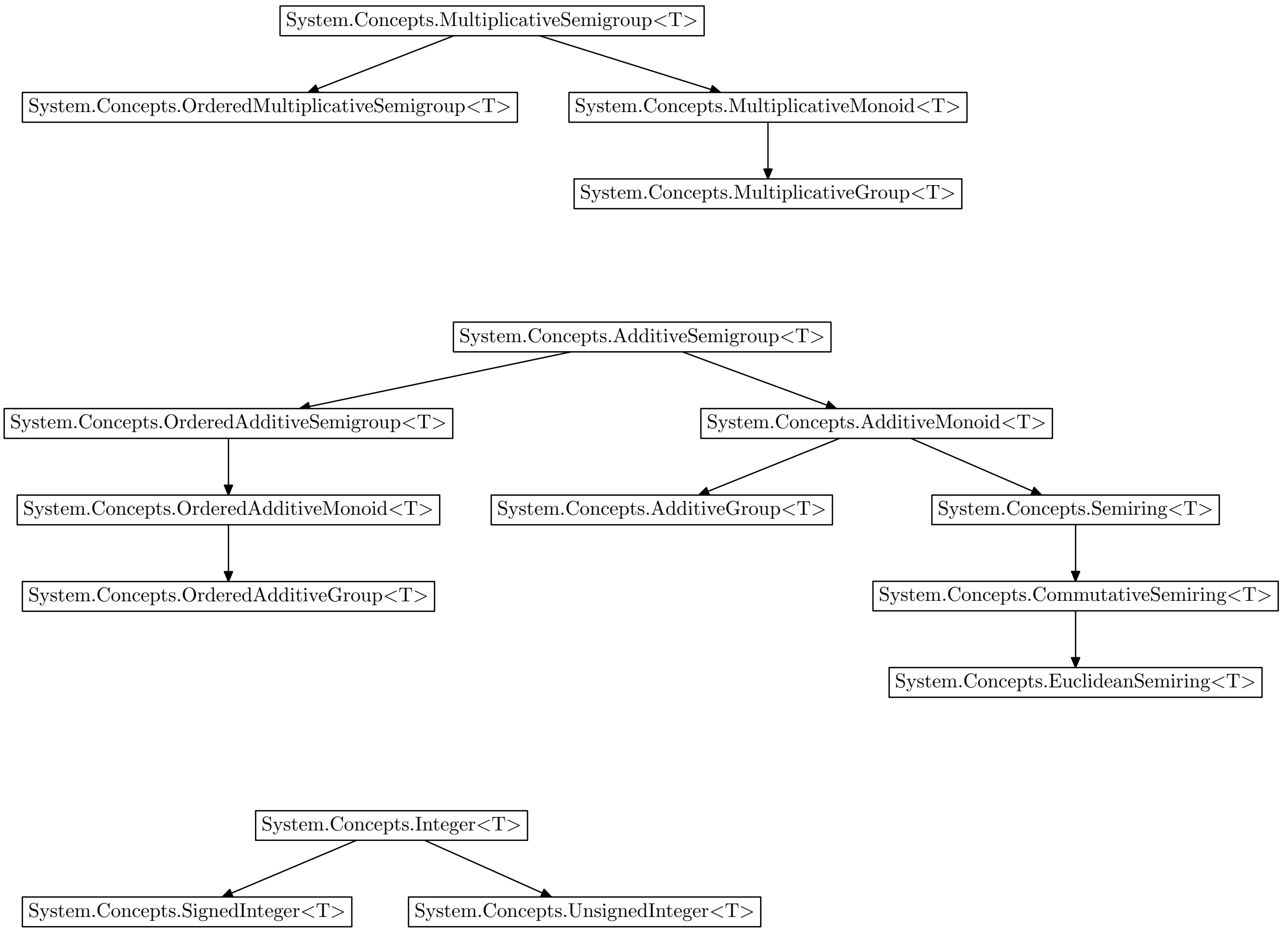


Figure 3.5: Concept Diagram 5: Algebraic Concepts



3.7 Concepts

Concept	Description
AdditiveGroup<T>	An additive group is an additive monoid that has an inverse of addition operation.
AdditiveMonoid<T>	An additive semigroup that has an identitity element 0 is called an additive monoid.
AdditiveSemigroup<T>	A set with an associative and commutative addition operation + is called an additive semigroup.
BackInsertionSequence<T>	A container with an **Add** member function, that adds elements to the end of the container is called a back insertion sequence.
BidirectionalContainer<T>	A container whose iterators are bidirectional iterators is a bidirectional container.
BidirectionalIterator<T>	An iterator that can be incremented and decremented is a bidirectional iterator.
BinaryFunction<T>	A function object that implements an **operator()** member function that accepts two parameters is called a binary function.
BinaryOperation<T>	A binary function whose result type is same as its first argument type is called a binary operation.
BinaryPredicate<T>	A binary function whose result type is **bool** is called a binary predicate.
Common<T, U>	A built-in concept that sets a requirement that types T and U have a common type. That common type is exposed as CommonType type definition.

CommutativeSemiring<T>	A semiring with commutative multiplication operation is called a commutative semiring.
Container<T>	A class that contains other objects is called a container. A container must expose the type of its contained object as an associated type named ValueType, and provide iterator types for iterating through the container, for example.
Convertible<T, U>	Convertible is a built-in concept that sets a requirement that type T is implicitly convertible to type U.
CopyAssignable<T>	Copy assignable type can be target of an copy assignment operation.
CopyAssignable<T, U>	Types T and U satisfy the CopyAssignable<T, U> concept if type T has an copy assignment operator taking a parameter of type U.
CopyConstructible<T>	A copy constructible type can be copied.
Copyable<T>	A copy constructible and copy assignable type is copyable.
DefaultConstructible<T>	A default constructible type has a constructor that takes no parameters.
Derived<T, U>	Derived is a built-in concept that sets a requirement that T and U are class types and T is derived from U.
Destructible<T>	A destructible type has a user defined or compiler generated destructor, or is trivially destructible.
EqualityComparable<T>	An equality comparable type can be compared for equality and inequality.

EqualityComparable<T, U>	Types T and U satisfy the cross-type equality comparable concept, if T is equality comparable, U is equality comparable and they have a common type that is equality comparable.
EuclideanSemiring<T>	Euclidean semiring is a commutative semiring that has division and remainder operations.
ExplicitlyConvertible<T, U>	Explicitly convertible is a built-in concept that sets a requirement that type T is excplicity convertible (i.e. using a **cast**) to type U.
ForwardContainer<T>	A container whose iterators are forward iterators is a forward container.
ForwardIterator<T>	A multipass iterator that can be incremented is a forward iterator.
FrontInsertionSequence<T>	A container with an **InsertFront** member function is called a front insertion sequence.
HashFunction<T, Key>	A unary function that computes a hash index from a key is a hash function.
InputIterator<T>	A one-pass iterator that can be incremented and compared for equality is an input iterator.
InsertionSequence<T>	A container with an **Insert** member function is called an insertion sequence.
Integer<I>	An integer type supports the usual integer operations.
KeySelectionFunction<T, Key, Value>	A unary function that extracts a key from a value type is called a key selection function.
LessThanComparable<T>	An less than comparable type can be compared for less than, greater than, less than or equal to and greater than or equal to releations.

LessThanComparable<T, U>	Types T and U satisfy the cross-type less than comparable concept, if T is less than comparable, U is less than comparable and they have a common type that is less than comparable.
Movable<T>	A move constructible and move assignable type is movable.
MoveAssignable<T>	A type equipped with move assignment operator is called move assignable.
MoveConstructible<T>	A type equipped with move constructor is called move constructible.
MultiplicativeGroup<T>	A multiplicative monoid with a division operation is called a multiplicative group.
MultiplicativeMonoid<T>	A multiplicative semigroup with an identity element is called a multiplicative monoid.
MultiplicativeSemigroup<T>	A set with an associative multiplication operation is called a multiplicative semigroup.
OrderedAdditiveGroup<T>	An ordered additive monoid that forms also an additive group is called an ordered additive group.
OrderedAdditiveMonoid<T>	An ordered additive semigroup that forms also an additive monoid is called an ordered additive monoid.
OrderedAdditiveSemigroup<T>	An additive semigroup with a total ordering relation on its elements is called an ordered additive semigroup.
OrderedMultiplicativeSemigroup<T>	A multiplicative semigroup with a total ordering relation on its elements is called an ordered multiplicative semigroup.

OutputIterator<T>	A writable iterator that is incrementable is an output iterator.
RandomAccessContainer<T>	A container whose iterators are random access iterators is a random access container.
RandomAccessIterator<T>	An iterator that supports incrementing, decrementing, subscripting, adding or subtracting an integer offset, and computing the difference of two iterators is a random access iterator.
Regular<T>	A regular type behaves like a built-in type: its objects can be default initialized, either copied, or moved (or both) and compared for equality and inequality.
Relation<T>	A binary predicate whose argument types are same is called a relation.
Relation<T, U, V>	A binary predicate with two not necessarily same argument types satisfy a multiparameter relation concept.
Same<T, U>	Same is a built-in concept that sets a requirement that T and U are exactly the same type.
Semiregular<T>	A semiregular type behaves in many ways like a built-in type: its objects can be default initialized, either copied or moved (or both), but not necessarily compared for equality and inequality.
Semiring<T>	A set with addition and multiplication operations that are connected with given axioms is called a semiring.
SignedInteger<I>	An integer type with a conversion from **sbyte** is called a signed integer.

TotallyOrdered<T>	A totally ordered type is a regular type that can be also compared for less than, greater than, less than or equal to, and greater than or equal to relationships.
TotallyOrdered<T, U>	Types T and U satisfy the cross-type totally ordered concept if T is totally ordered, U is totally ordered and they have a common type that is totally ordered.
TrivialIterator<T>	A trivial iterator concept collects together requirements common to all iterator types.
UnaryFunction<T>	A function object that implements an **operator()** that accepts one parameter is called a unary function.
UnaryOperation<T>	A unary function whose result type and argument type are same is called a unary operation.
UnaryPredicate<T>	A unary function whose result type is **bool** is called a unary predicate.
UnsignedInteger<U>	An integer type with a conversion from **byte** is called an unsigned integer.

3.7.1 AdditiveGroup<T> Concept

An additive group is an additive monoid that has an inverse of addition operation.

Syntax

```
public concept AdditiveGroup<T>;
```

Refines

AdditiveMonoid<T>

Constraints

```
T operator-(T);

T operator-(T, T);
```

Axioms

```
unaryMinusIsInverseOp(T a)
{
    a + (-a) == 0 && (-a) + a == 0;
}
subtract(T a, T b)
{
    a - b == a + (-b);
}
```

Models

Integer and floating-point types with + and - are partial models of an additive group.

3.7.2 AdditiveMonoid<T> Concept

An additive semigroup that has an identity element 0 is called an additive monoid.

Syntax

```
public concept AdditiveMonoid<T>;
```

Refines

AdditiveSemigroup<T>

Constraints

```
T(sbyte);
```

Axioms

```
zeroIsIdentityElement(T a)
{
    a + 0 == a && 0 + a == a;
}
```

Models

Integer and floating-point types with + are partial models of an additive monoid.

3.7.3 AdditiveSemigroup<T> Concept

A set with an associative and commutative addition operation + is called an additive semigroup.

Syntax

```
public concept AdditiveSemigroup<T>;
```

Constraints

where T is Regular;

```
T operator+(T, T);
```

Axioms

```
additionIsAssociative(T a, T b, T c)
{
    (a + b) + c == a + (b + c);
}
additionIsCommutative(T a, T b)
{
    a + b == b + a;
}
```

Models

Integer and floating-point types with + are partial models of an additive semigroup.

3.7.4 BackInsertionSequence<T> Concept

A container with an **Add** member function, that adds elements to the end of the container is called a back insertion sequence.

Syntax

```
public concept BackInsertionSequence<T>;
```

Refines

Container<T>

Constraints

```
void T.Add(ValueType);
```

Models

List<T> is a back insertion sequence.

3.7.5 BidirectionalContainer<T> Concept

A container whose iterators are bidirectional iterators is a bidirectional container.

Syntax

```
public concept BidirectionalContainer<T>;
```

Refines

ForwardContainer<T>

Constraints

where Iterator is BidirectionalIterator and ConstIterator is BidirectionalIterator;

Models

List<T>, Set<T, C> and Map<Key, Value, KeyCompare> are bidirectional containers.

3.7.6 BidirectionalIterator<T> Concept

An iterator that can be incremented and decremented is a bidirectional iterator.

Syntax

```
public concept BidirectionalIterator<T>;
```

Refines

ForwardIterator<T>

Constraints

```
T& operator--();
```

Models

RedBlackTreeNodeIterator<T, R, P> is a bidirectional iterator.

3.7.7 BinaryFunction<T> Concept

A function object that implements an **operator()** member function that accepts two parameters is called a binary function.

Syntax

```
public concept BinaryFunction<T>;
```

Constraints

where T is Semiregular;
typename FirstArgumentType;
typename SecondArgumentType;
typename ResultType;
where FirstArgumentType is Semiregular and SecondArgumentType is Semiregular;

```
ResultType operator()(FirstArgumentType, SecondArgumentType);
```

Models

Plus<T>, Minus<T>, Multiplies<T>, Divides<T> and Remainder<T> are binary functions.

3.7.8 BinaryOperation<T> Concept

A binary function whose result type is same as its first argument type is called a binary operation.

Syntax

```
public concept BinaryOperation<T>;
```

Refines

BinaryFunction<T>

Constraints

where ResultType is FirstArgumentType;

Models

Plus<T>, Minus<T>, Multiplies<T>, Divides<T> and Remainder<T> are binary operations.

3.7.9 BinaryPredicate<T> Concept

A binary function whose result type is **bool** is called a binary predicate.

Syntax

```
public concept BinaryPredicate<T>;
```

Refines

BinaryFunction<T>

Constraints

where ResultType is bool;

Models

EqualTo<T>, EqualTo2<T, U>, NotEqualTo<T>, NotEqualTo2<T, U>, Less<T>, Less2<T, U>, Greater<T>, Greater2<T, U>, LessOrEqualTo<T>, LessOrEqualTo2<T, U>, GreaterOrEqualTo<T>, GreaterOrEqualTo2<T, U> are binary predicates.

3.7.10 Common<T, U> Concept

A built-in concept that sets a requirement that types T and U have a common type. That common type is exposed as CommonType type definition.

Syntax

```
public concept Common<T, U>;
```

Models

int and **double** satisfy the common type requirement and their common type is **double**.

3.7.11 CommutativeSemiring<T> Concept

A semiring with commutative multiplication operation is called a commutative semiring.

Syntax

```
public concept CommutativeSemiring<T>;
```

Refines

Semiring<T>

Axioms

```
multiplicationIsCommutative(T a, T b)
{
    a * b == b * a;
}
```

Models

Integer types with + and * are partial models of a commutative semiring.

3.7.12 Container<T> Concept

A class that contains other objects is called a container. A container must expose the type of its contained object as an associated type named ValueType, and provide iterator types for iterating through the container, for example.

Syntax

```
public concept Container<T>;
```

Constraints

where T is Semiregular;
typename ValueType;
typename Iterator;
typename ConstIterator;
where Iterator is TrivialIterator and ConstIterator is TrivialIterator and ValueType is Iterator.ValueType;

```
Iterator T.Begin();

ConstIterator T.CBegin();

Iterator T.End();

ConstIterator T.CEnd();

int T.Count();

bool T.IsEmpty();

void T.Swap(T&);
```

Models

List<T>, Set<T, C> and Map<Key, Value, KeyCompare> and ForwardList<T> are containers.

3.7.13 Convertible<T, U> Concept

Convertible is a built-in concept that sets a requirement that type T is implicitly convertible to type U.

Syntax

```
public concept Convertible<T, U>;
```

Models

For example, Convertible<int, double> is true, but Convertible<double, int> is false.

3.7.14 CopyAssignable<T> Concept

Copy assignable type can be target of an copy assignment operation.

Syntax

```
public concept CopyAssignable<T>;
```

Constraints

```
void operator=(const T&);
```

Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are copy assignable. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are copy assignable. String, Exception, Pair<T, U> and SharedPtr<T>, for example, are also copy assignable, but UniquePtr<T> is not copy assignable, because its copy assignment operator is suppressed.

3.7.14.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.CopyAssignable.T.cm at line 53:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.CopyAssignable' because
//  there is no member function with signature 'void D.operator=(const D
    &)':
//  Tester<D> td;    // error
//  ^

// A has compiler generated copy assignment operation:

class A
{
}

// B has compiler generated copy assignment operation:

class B
{
    public default void operator=(const B&);
}

// C has user defined copy assignment operation:

class C
{
```

```
    public void operator=(const C& that) {}
}

//  But D is not copy assignable
//  because user defined destructor
//  suppresses the compiler from generating
//  a copy assignment operation:

class D
{
    public ~D() {}
}

class Tester<T> where T is CopyAssignable
{
}

void main()
{
    A a;
    A a2;
    a = a2;          // calls copy assignment operation
    Tester<A> ta;    // ok
    Tester<B> tb;    // ok
    Tester<C> tc;    // ok
    Tester<D> td;    // error
}
```

3.7.15 CopyAssignable<T, U> Concept

Types T and U satisfy the **CopyAssignable<T, U>** concept if type T has an copy assignment operator taking a parameter of type U.

Syntax

```
public concept CopyAssignable<T, U>;
```

Constraints

```
void operator=(const U&);
```

3.7.15.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.CopyAssignable.T.U.cm at line 38:
//  types (C, A) do not satisfy the requirements of concept 'System.
    Concepts.CopyAssignable' because
//  there is no member function with signature 'void C.operator=(const A
    &)':
//  Tester<C, A> tca;    // error
//  ^

class A
{
}

//  CopyAssignable<B, A> is true...

class B
{
    public void operator=(const A& a) {}
}

//  But CopyAssignable<C, A> is false...

class C
{
}

class Tester<T, U> where CopyAssignable<T, U>
{
}

void main()
{
```

```
    A a;
    B b;
    b = a;                // calls copy assignment operation of B
    Tester<B, A> tba;     // ok
    Tester<C, A> tca;     // error
}
```

3.7.16 CopyConstructible<T> Concept

A copy constructible type can be copied.

Syntax

```
public concept CopyConstructible<T>;
```

Constraints

```
T(const T&);
```

Axioms

```
copyIsEqual(T a)
{
    eq(T(a), a);
}
```

Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are copy constructible. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are copy constructible if their value type is copy constructible. String, Exception, Pair<T, U> and SharedPtr<T>, for example, are also copy constructible, but UniquePtr<T> is not copy constructible, because its copy constructor is suppressed.

3.7.16.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.CopyConstructible.cm at line 53:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.CopyConstructible' because
//  there is no constructor with signature 'D.D(const D&)':
//  Tester<D> td;    // error
//  ^

// A has compiler generated copy constructor:

class A
{
}

// B has compiler generated copy constructor:
```

```
class B
{
    public default B(const B&);
}

// C has user defined copy constructor:

class C
{
    public C(const C& that) {}
}

//  But D is not copy constructible
//  because user defined destructor
//  suppresses the compiler from generating
//  a copy constructor:

class D
{
    public ~D() {}
}

class Tester<T> where T is CopyConstructible
{
}

void main()
{
    A a;
    A a2(a);        // a2 is copy constructed
    A a3 = a;       // a3 is copy constructed
    Tester<A> ta;   // ok
    Tester<B> tb;   // ok
    Tester<C> tc;   // ok
    Tester<D> td;   // error
}
```

3.7.17 Copyable<T> Concept

A copy constructible and copy assignable type is copyable.

Syntax

```
public concept Copyable<T>;
```

Constraints

where T is CopyConstructible and T is CopyAssignable;

3.7.18 DefaultConstructible<T> Concept

A default constructible type has a constructor that takes no parameters.

Syntax

```
public concept DefaultConstructible<T>;
```

Constraints

```
T();
```

Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are default constructible. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are default constructible. String, Exception, Pair<T, U>, UniquePtr<T> and SharedPtr<T>, for example, are also default constructible.

Remarks

The default constructor initializes a value of its type to the natural default value: that is 0 for a numeric type, **false** for a Boolean type, empty string for a string type, empty container for a container type, etc...

3.7.18.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.DefaultConstructible.cm at line 55:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.DefaultConstructible' because
//  there is no constructor with signature 'D.D()':
//  Tester<D> td;    // error
//  ^

// A has compiler generated default constructor:

class A
{
}

// B has compiler generated default constructor:

class B
{
    public default B();
}
```

```
// C has user defined default constructor:

class C
{
    public C() {}
}

//  But D is not default constructible
//  because user defined constructor
//  suppresses the compiler from generating
//  a default constructor:

class D
{
    public D(int x) {}
}

class Tester<T> where T is DefaultConstructible
{
}

void main()
{
    int x;           // x is default constructed
    Tester<int> ti; // ok
    A a;             // a is default constructed
    Tester<A> ta;    // ok
    B b;             // b is default constructed
    Tester<B> tb;    // ok
    C c;             // c is default constructed
    Tester<C> tc;    // ok
    Tester<D> td;    // error
    //D d;           // would generate error: "overload resolution failed:
        '@constructor(D*)' not found..."
}
```

3.7.19 Derived<T, U> Concept

Derived is a built-in concept that sets a requirement that T and U are class types and T is derived from U.

Syntax

```
public concept Derived<T, U>;
```

3.7.20 Destructible<T> Concept

A destructible type has a user defined or compiler generated destructor, or is trivially destructible.

Syntax

```
public concept Destructible<T>;
```

Constraints

```
~T();
```

Models

All types in Cmajor are destructible.

3.7.20.1 Example

```
using System;
using System.Concepts;

// A has trivial destructor:

class A
{
}

// B has compiler generated destructor:

class B
{
    public default ~B();
}

// C has user defined destructor:

class C
{
    public ~C() {}
}

class Tester<T> where T is Destructible
{
}

void main()
{
    {
        A a;
//      ...
```

```
//        <-- a is destroyed here
    }
    Tester<A> ta;    // ok
    Tester<B> tb;    // ok
    Tester<C> tc;    // ok
}
```

3.7.21 EqualityComparable<T> Concept

An equality comparable type can be compared for equality and inequality.

Syntax

```
public concept EqualityComparable<T>;
```

Constraints

```
bool operator==(T, T);
```

Axioms

```
equal(T a, T b)
{
    a == b <=> eq(a, b);
}
reflexive(T a)
{
    a == a;
}
symmetric(T a, T b)
{
    a == b => b == a;
}
transitive(T a, T b, T c)
{
    a == b && b == c => a == c;
}
notEqualTo(T a, T b)
{
    a != b <=> !(a == b);
}
```

3.7.22 EqualityComparable<T, U> Concept

Types T and U satisfy the cross-type equality comparable concept, if T is equality comparable, U is equality comparable and they have a common type that is equality comparable.

Syntax

```
public concept EqualityComparable<T, U>;
```

Refines

Common<T, U>

Constraints

where T is EqualityComparable and U is EqualityComparable and CommonType is EqualityComparable;

3.7.23 EuclideanSemiring<T> Concept

Euclidean semiring is a commutative semiring that has division and remainder operations.

Syntax

```
public concept EuclideanSemiring<T>;
```

Refines

CommutativeSemiring<T>

Constraints

```
T operator%(T, T);

T operator/(T, T);
```

Axioms

```
quotientAndRemainder(T a, T b)
{
    b != 0 => a == a / b * b + a % b;
}
```

3.7.24 ExplicitlyConvertible<T, U> Concept

Explicitly convertible is a built-in concept that sets a requirement that type T is excplicity convertible (i.e. using a **cast**) to type U.

Syntax

```
public concept ExplicitlyConvertible<T, U>;
```

3.7.25 ForwardContainer<T> Concept

A container whose iterators are forward iterators is a forward container.

Syntax

```
public concept ForwardContainer<T>;
```

Refines

Container<T>

Constraints

where Iterator is ForwardIterator and ConstIterator is ForwardIterator;

Models

ForwardList<T> is a forward container.

3.7.26 ForwardIterator<T> Concept

A multipass iterator that can be incremented is a forward iterator.

Syntax

```
public concept ForwardIterator<T>;
```

Refines

InputIterator<T>

Constraints

where T is OutputIterator;

Models

ForwardListNodeIterator<T, R, P> is a forward iterator.

3.7.27 FrontInsertionSequence<T> Concept

A container with an **InsertFront** member function is called a front insertion sequence.

Syntax

```
public concept FrontInsertionSequence<T>;
```

Refines

Container<T>

Constraints

```
Iterator T.InsertFront(ValueType);
```

Models

List<T> and ForwardList<T> are front insertion sequences.

3.7.28 HashFunction<T, Key> Concept

A unary function that computes a hash index from a key is a hash function.

Syntax

```
public concept HashFunction<T, Key>;
```

Refines

UnaryFunction<T>

Constraints

where ArgumentType is Key and ResultType is int;

3.7.29 InputIterator<T> Concept

A one-pass iterator that can be incremented and compared for equality is an input iterator.

Syntax

```
public concept InputIterator<T>;
```

Refines

TrivialIterator<T>

Constraints

`T& operator++();`
where T is Regular;

3.7.30 InsertionSequence<T> Concept

A container with an **Insert** member function is called an insertion sequence.

Syntax

```
public concept InsertionSequence<T>;
```

Refines

ForwardContainer<T>

Constraints

```
Iterator T.Insert(Iterator, ValueType);
```

Models

List<T> is an insertion sequence.

3.7.31 Integer<I> Concept

An integer type supports the usual integer operations.

Syntax

```
public concept Integer<I>;
```

Constraints

where I is TotallyOrdered;

```
I operator-(I);

I operator~(I);

I& operator++(I&);

I& operator--(I&);

I operator+(I, I);

I operator-(I, I);

I operator*(I, I);

I operator/(I, I);

I operator%(I, I);

I operator<<(I, I);

I operator>>(I, I);

I operator&(I, I);

I operator|(I, I);

I operator^(I, I);
```

Models

All Cmajor integer types.

3.7.32 KeySelectionFunction<T, Key, Value> Concept

A unary function that extracts a key from a value type is called a key selection function.

Syntax

```
public concept KeySelectionFunction<T, Key, Value>;
```

Refines

UnaryFunction<T>

Constraints

where ArgumentType is Value and ResultType is Key;

3.7.33 LessThanComparable<T> Concept

An less than comparable type can be compared for less than, greater than, less than or equal to and greater than or equal to releations.

Syntax

```
public concept LessThanComparable<T>;
```

Constraints

```
bool operator<(T, T);
```

Axioms

```
irreflexive(T a)
{
    !(a < a);
}
antisymmetric(T a, T b)
{
    a < b => !(b < a);
}
transitive(T a, T b, T c)
{
    a < b && b < c => a < c;
}
total(T a, T b)
{
    a < b || a == b || a > b;
}
greaterThan(T a, T b)
{
    a > b <=> b < a;
}
greaterThanOrEqualTo(T a, T b)
{
    a >= b <=> !(a < b);
}
lessThanOrEqualTo(T a, T b)
{
    a <= b <=> !(b < a);
}
```

3.7.34 LessThanComparable<T, U> Concept

Types T and U satisfy the cross-type less than comparable concept, if T is less than comparable, U is less than comparable and they have a common type that is less than comparable.

Syntax

```
public concept LessThanComparable<T, U>;
```

Refines

Common<T, U>

Constraints

where T is LessThanComparable and U is LessThanComparable and CommonType is LessThanComparable;

3.7.35 Movable<T> Concept

A move constructible and move assignable type is movable.

Syntax

```
public concept Movable<T>;
```

Constraints

where T is MoveConstructible and T is MoveAssignable;

3.7.36 MoveAssignable<T> Concept

A type equipped with move assignment operator is called move assignable.

Syntax

```
public concept MoveAssignable<T>;
```

Constraints

```
void operator=(T&&);
```

3.7.37 MoveConstructible<T> Concept

A type equipped with move constructor is called move constructible.

Syntax

```
public concept MoveConstructible<T>;
```

Constraints

```
T(T&&);
```

3.7.38 MultiplicativeGroup<T> Concept

A multiplicative monoid with a division operation is called a multiplicative group.

Syntax

```
public concept MultiplicativeGroup<T>;
```

Refines

MultiplicativeMonoid<T>

Constraints

```
T operator/(T, T);
```

Axioms

```
multiplicativeInverseIsInverseOp(T a)
{
    a * (1/a) == 1 && (1/a) * a == 1;
}
division(T a, T b)
{
    a / b == a * (1/b);
}
```

3.7.39 MultiplicativeMonoid<T> Concept

A multiplicative semigroup with an identity element is called a multiplicative monoid.

Syntax

```
public concept MultiplicativeMonoid<T>;
```

Refines

MultiplicativeSemigroup<T>

Constraints

```
T(sbyte);
```

Axioms

```
oneIsIdentityElement(T a)
{
    a * 1 == a && 1 * a == a;
}
```

3.7.40 MultiplicativeSemigroup<T> Concept

A set with an associative multiplication operation is called a multiplicative semigroup.

Syntax

```
public concept MultiplicativeSemigroup<T>;
```

Constraints

where T is Regular;

```
T operator*(T, T);
```

Axioms

```
multiplicationIsAssociative(T a, T b, T c)
{
    (a * b) * c == a * (b * c);
}
```

3.7.41 OrderedAdditiveGroup<T> Concept

An ordered additive monoid that forms also an additive group is called an ordered additive group.

Syntax

```
public concept OrderedAdditiveGroup<T>;
```

Refines

OrderedAdditiveMonoid<T>

Constraints

where T is AdditiveGroup;

3.7.42 OrderedAdditiveMonoid<T> Concept

An ordered additive semigroup that forms also an additive monoid is called an ordered additive monoid.

Syntax

```
public concept OrderedAdditiveMonoid<T>;
```

Refines

OrderedAdditiveSemigroup<T>

Constraints

where T is AdditiveMonoid;

3.7.43 OrderedAdditiveSemigroup<T> Concept

An additive semigroup with a total ordering relation on its elements is called an ordered additive semigroup.

Syntax

```
public concept OrderedAdditiveSemigroup<T>;
```

Refines

AdditiveSemigroup<T>

Constraints

where T is TotallyOrdered;

Axioms

```
additionPreservesOrder(T a, T b, T c)
{
    a < b => a + c < b + c;
}
```

3.7.44 OrderedMultiplicativeSemigroup<T> Concept

A multiplicative semigroup with a total ordering relation on its elements is called an ordered multiplicative semigroup.

Syntax

```
public concept OrderedMultiplicativeSemigroup<T>;
```

Refines

MultiplicativeSemigroup<T>

Constraints

where T is TotallyOrdered;

3.7.45 OutputIterator<T> Concept

A writable iterator that is incrementable is an output iterator.

Syntax

```
public concept OutputIterator<T>;
```

Refines

TrivialIterator<T>

Constraints

```
T& operator++();
```

3.7.46 RandomAccessContainer<T> Concept

A container whose iterators are random access iterators is a random access container.

Syntax

```
public concept RandomAccessContainer<T>;
```

Refines

BidirectionalContainer<T>

Constraints

where Iterator is RandomAccessIterator and ConstIterator is RandomAccessIterator;

3.7.47 RandomAccessIterator<T> Concept

An iterator that supports incrementing, decrementing, subscripting, adding or subtracting an integer offset, and computing the difference of two iterators is a random access iterator.

Syntax

```
public concept RandomAccessIterator<T>;
```

Refines

BidirectionalIterator<T>

Constraints

```
ReferenceType operator[](int);
```

```
T operator+(T, int);
```

```
T operator+(int, T);
```

```
T operator-(T, int);
```

```
int operator-(T, T);
```

where T is LessThanComparable;

3.7.48 Regular<T> Concept

A regular type behaves like a built-in type: its objects can be default initialized, either copied, or moved (or both) and compared for equality and inequality.

Syntax

```
public concept Regular<T>;
```

Refines

Semiregular<T>

Constraints

where T is EqualityComparable;

Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are regular. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are regular, if their value type is regular.

Remarks

If a type implements the == operator, the compiler implements the != operator automatically.

3.7.48.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.Regular.cm at line 98:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.Regular'
//  because type 'D' does not satisfy the requirements of concept 'System
    .Concepts.EqualityComparable' because
//  there is no member function with signature 'bool D.operator==(D)' and
     no function with signature 'bool operator==(D, D)':
//  Tester<D> td;    // error
//  ^

// A has user-defined default constructor, compiler generated copy
    constructor, copy assignment,
// move constructor and move assignment,
// it is trivially destructible and its objects can be compare for
    equality, so it is regular:

class A
```

```
{
    public A(): id(0)
    {
    }
    public A(int id_): id(id_)
    {
    }
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const A& left, const A& right)
{
    return left.Id() == right.Id();
}

// B has user-defined default constructor, compiler generated copy
    constructor, copy assignment,
// move constructor, move assignment and destructor, and its objects can
    be compared for equality, so it is regular:

class B
{
    public B(): id(0)
    {
    }
    public B(int id_): id(id_)
    {
    }
    public default B(const B&);
    public default void operator=(const B&);
    public default B(B&&);
    public default void operator=(B&&);
    public default ~B();
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const B& left, const B& right)
{
    return left.Id() == right.Id();
}

// C has user defined default constructor, copy constructor, copy
    assignment,
// move constructior, move assignment and destructor,
// its objects can be compared for equality, so it is regular:
```

```
class C
{
    public C(): id(0) {}
    public C(const C& that): id(that.id) {}
    public void operator=(const C& that) { id = that.id; }
    public C(C&& that): id(that.id)
    {
        that.id = 0;
    }
    public void operator=(C&& that)
    {
        Swap(id, that.id);
    }
    public ~C() {}
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const C& left, const C& right)
{
    return left.Id() == right.Id();
}

//  But D is not regular,
//  because its objects cannot be compared for equality:

class D
{
}

class Tester<T> where T is Regular
{
}

void main()
{
    A a;
    A a2(a);        // a2 is copy constructed
    A a3;
    a3 = a2;        // a3 is copy assigned
    Tester<A> ta;   // ok
    Tester<B> tb;   // ok
    Tester<C> tc;   // ok
    Tester<D> td;   // error
}
```

3.7.49 Relation<T> Concept

A binary predicate whose argument types are same is called a relation.

Syntax

```
public concept Relation<T>;
```

Refines

BinaryPredicate<T>

Constraints

typename Domain;
where Same<Domain, FirstArgumentType> and Same<SecondArgumentType, Domain>;

3.7.50 Relation<T, U, V> Concept

A binary predicate with two not necessarily same argument types satisfy a multiparameter relation concept.

Syntax

```
public concept Relation<T, U, V>;
```

Refines

BinaryPredicate<T>

Constraints

where FirstArgumentType is U and SecondArgumentType is V;

3.7.51 Same<T, U> Concept

Same is a built-in concept that sets a requirement that T and U are exactly the same type.

Syntax

```
public concept Same<T, U>;
```

3.7.52 Semiregular<T> Concept

A semiregular type behaves in many ways like a built-in type: its objects can be default initialized, either copied or moved (or both), but not necessarily compared for equality and inequality.

Syntax

```
public concept Semiregular<T>;
```

Constraints

where T is DefaultConstructible and (T is Copyable or T is Movable) and T is Destructible;

Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are semiregular. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are semiregular. String, Exception, Pair<T, U> and SharedPtr<T>, for example, are also semiregular. Although UniquePtr<T> is not copyable because its copy constructor and assignment operator are suppressed, it is movable because it implements move constructor and move assignment operator, so it is semiregular.

Remarks

Many containers require that the type of the contained object is semiregular.

3.7.52.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.Semiregular.cm at line 65:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.Semiregular' because
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.CopyConstructible' because
//  there is no constructor with signature 'D.D(const D&)':
//  Tester<D> td;    // error
//  ^

// A has compiler generated default constructor, copy constructor and
    copy assignment,
// move constructor and move assignment and it is trivially destructible,
     so it is semiregular:

class A
{
```

```
}

// B has compiler generated default constructor, copy constructor, copy
    assignment,
// move constructor, move assignment and destructor, so it is semiregular
    :

class B
{
    public default B();
    public default B(const B&);
    public default void operator=(const B&);
    public default B(B&&);
    public default void operator=(B&&);
    public default ~B();
}

// C has user defined default constructor, copy constructor, copy
    assignment,
// move constructor, move assignment and destructor, so it is semiregular
    :

class C
{
    public C() {}
    public C(const C& that) {}
    public void operator=(const C& that) {}
    public C(C&& that) {}
    public void operator=(C&& that) {}
    public ~C() {}
}

//  But D is not semiregular,
//  because the copy constructor and the
//  copy assignment operator are suppressed.

class D
{
    public D() {}
    suppress D(const D&);
    suppress void operator=(const D&);
}

class Tester<T> where T is Semiregular
{
}

void main()
{
    A a;
    A a2(a);          // a2 is copy constructed
    A a3;
    a3 = a2;          // a3 is copy assigned
```

```
    A a4;
    a4 = Rvalue(a3);// a4 is move assigned
    Tester<A> ta;    // ok
    Tester<B> tb;    // ok
    Tester<C> tc;    // ok
    Tester<D> td;    // error
}
```

3.7.53 Semiring<T> Concept

A set with addition and multiplication operations that are connected with given axioms is called a semiring.

Syntax

```
public concept Semiring<T>;
```

Refines

AdditiveMonoid<T>

Constraints

where T is MultiplicativeMonoid;

Axioms

```
zeroIsNotOne
{
    0 != 1;
}
multiplyingByZeroYieldsZero(T a)
{
    0 * a == 0 && a * 0 == 0;
}
distributivity(T a, T b, T c)
{
    a * (b + c) == a * b + a * c && (b + c) * a == b * a + c * a;
}
```

3.7.54 SignedInteger<I> Concept

An integer type with a conversion from **sbyte** is called a signed integer.

Syntax

```
public concept SignedInteger<I>;
```

Constraints

where I is Integer;

```
I(sbyte);
```

Models

sbyte, **short**, **int** and **long** are signed integer types.

3.7.55 TotallyOrdered<T> Concept

A totally ordered type is a regular type that can be also compared for less than, greater than, less than or equal to, and greater than or equal to relationships.

Syntax

```
public concept TotallyOrdered<T>;
```

Refines

Regular<T>

Constraints

where T is LessThanComparable;

Models

Arithmetic and character types (**int**, **double**, **char**, etc...) are totally ordered. All types in the System.Collections namespace (List<T>, Set<T, C>, etc...) are totally ordered if their value type is totally ordered.

Remarks

If a type implements the < operator, the compiler implements the >, <= and >= operators automatically.

3.7.55.1 Example

```
using System;
using System.Concepts;

//  Writes:
//  Error T138 in file C:/Programming/cmajor++/doc/lib/examples/System.
    Concepts.TotallyOrdered.cm at line 127:
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.TotallyOrdered' because
//  type 'D' does not satisfy the requirements of concept 'System.
    Concepts.LessThanComparable' because
//  there is no member function with signature 'bool D.operator<(D)' and
    no function with signature 'bool operator<(D, D)':
//  Tester<D> td;   // error
//  ^

// A has user-defined default constructor, compiler generated copy
    constructor and copy assignment,
// it is trivially destructible and its objects can be compare for
    equality and less than relationship,
// so it is totally ordered:
```

```
class A
{
    public A(): id(0)
    {
    }
    public A(int id_): id(id_)
    {
    }
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const A& left, const A& right)
{
    return left.Id() == right.Id();
}

public bool operator<(const A& left, const A& right)
{
    return left.Id() < right.Id();
}

// B has user-defined default constructor, compiler generated copy
    constructor, copy assignment and destructor, and
// its objects can be compared for equality and less than relationship,
    so it is totally ordered:

class B
{
    public B(): id(0)
    {
    }
    public B(int id_): id(id_)
    {
    }
    public default B(const B&);
    public default void operator=(const B&);
    public default ~B();
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const B& left, const B& right)
{
    return left.Id() == right.Id();
}
```

```
public bool operator<(const B& left, const B& right)
{
    return left.Id() < right.Id();
}

// C has user defined default constructor, copy constructor, copy
   assignment
// and destructor, its objects can be compared for equality and less than
    relationship,
// so it is totally ordered:

class C
{
    public C(): id(0) {}
    public C(const C& that): id(that.id) {}
    public void operator=(const C& that) { id = that.id; }
    public ~C() {}
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const C& left, const C& right)
{
    return left.Id() == right.Id();
}

public bool operator<(const C& left, const C& right)
{
    return left.Id() < right.Id();
}

//  But D is not totally ordered,
//  because its objects cannot be compared for less than relationship:

class D
{
    public D(): id(0) {}
    public D(const D& that): id(that.id) {}
    public void operator=(const D& that) { id = that.id; }
    public ~D() {}
    public int Id() const
    {
        return id;
    }
    private int id;
}

public bool operator==(const D& left, const D& right)
{
    return left.Id() == right.Id();
```

```
}

class Tester<T> where T is TotallyOrdered
{
}

void main()
{
    A a;
    A a2(a);        // a2 is copy constructed
    A a3;
    a3 = a2;        // a3 is copy assigned
    Tester<A> ta;   // ok
    Tester<B> tb;   // ok
    Tester<C> tc;   // ok
    Tester<D> td;   // error
}
```

3.7.56 TotallyOrdered<T, U> Concept

Types T and U satisfy the cross-type totally ordered concept if T is totally ordered, U is totally ordered and they have a common type that is totally ordered.

Syntax

```
public concept TotallyOrdered<T, U>;
```

Refines

Common<T, U>

Constraints

where T is TotallyOrdered and U is TotallyOrdered and CommonType is TotallyOrdered;

3.7.57 TrivialIterator<T> Concept

A trivial iterator concept collects together requirements common to all iterator types.

Syntax

```
public concept TrivialIterator<T>;
```

Constraints

where T is Semiregular;
typename ValueType;
where ValueType is Semiregular;
typename ReferenceType;
where ReferenceType is ValueType&;

```
ReferenceType operator*();
```

typename PointerType;
where PointerType is ValueType*;

```
PointerType operator->();
```

3.7.58 UnaryFunction<T> Concept

A function object that implements an **operator()** that accepts one parameter is called a unary function.

Syntax

```
public concept UnaryFunction<T>;
```

Constraints

where T is Semiregular;
typename ArgumentType;
typename ResultType;
where ArgumentType is Semiregular;

```
ResultType operator()(ArgumentType);
```

Models

Negate<T> and Identity<T> are unary functions.

3.7.59 UnaryOperation<T> Concept

A unary function whose result type and argument type are same is called a unary operation.

Syntax

```
public concept UnaryOperation<T>;
```

Refines

UnaryFunction<T>

Constraints

where ResultType is ArgumentType;

Models

Negate<T> and Identity<T> are unary operations.

3.7.60 UnaryPredicate<T> Concept

A unary function whose result type is **bool** is called a unary predicate.

Syntax

```
public concept UnaryPredicate<T>;
```

Refines

UnaryFunction<T>

Constraints

where ResultType is bool;

3.7.61 UnsignedInteger<U> Concept

An integer type with a conversion from **byte** is called an unsigned integer.

Syntax

```
public concept UnsignedInteger<U>;
```

Constraints

where U is Integer;

```
U(byte);
```

Models

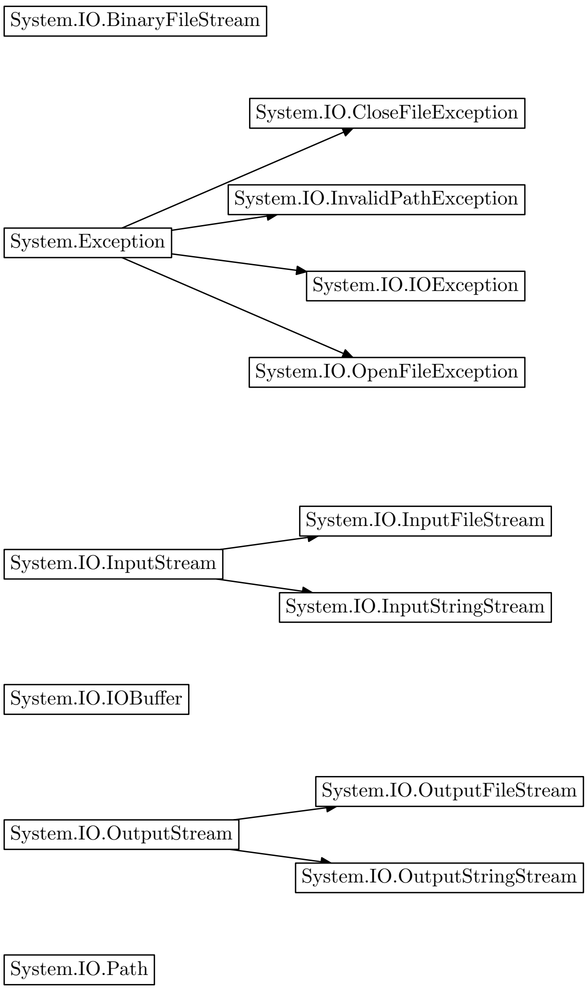
byte, **ushort**, **uint** and **ulong** are unsigned integer types.

4 System.IO Namespace

Contains classes and functions for doing input and output.

Figure 4.1 contains the classes in this namespace.

Figure 4.1: Class Diagram: I/O Classes



4.8 Classes

Class	Description
BinaryFileStream	A stream of bytes connected to a file.
CloseFileException	An exception thrown when closing a file fails.
IOBuffer	A handle to dynamically allocated memory.
IOException	An exception thrown when an I/O operation fails.
InputFileStream	A stream of characters connected to an input file.
InputStream	An abstract base class for input stream classes.
InputStringStream	A class for reading from a string.
InvalidPathException	An exception class that is thrown if a path contains too many .. components.
OpenFileException	An exception class thrown if opening a file fails.
OutputFileStream	A stream of characters connected to an output file.
OutputStream	An abstract base class for output stream classes.
OutputStringStream	A class for writing to a string.
Path	A static class for manipulating paths.

4.8.1 BinaryFileStream Class

A stream of bytes connected to a file.

Syntax

```
public class BinaryFileStream;
```

4.8.1.1 Member Functions

Member Function	Description
BinaryFileStream(BinaryFileStream&&)	Move constructor.
BinaryFileStream(const string&, OpenMode)	Opens a binary file stream with the specified open mode and file name.
BinaryFileStream(const string&, OpenMode, int)	Opens a binary file stream with the specified open mode, file name and permission mode.
operator=(BinaryFileStream&&)	Move assignment.
~BinaryFileStream()	If the binary file stream is connected to an open file, closes the file.
Close()	If the binary file stream is connected to an open file, closes the file, otherwise throws CloseFileException.
GetFileSize()	Returns the size of the file binary file stream is connected to.
Open(const string&, OpenMode, int)	Opens a file with the specified open mode, file name and permission mode and connects it with the binary file stream.
Read(void*, int)	Reads at most given number of bytes from the connected file into the specified buffer.
ReadBool()	Reads a Boolean value from the binary file stream and returns it.
ReadByte()	Reads a **byte** from the binary file stream and returns it.

ReadChar()	Reads a character from the binary file stream and returns it.
ReadDouble()	Reads a **double** from the binary file stream and returns it.
ReadFloat()	Reads a **float** from the binary file stream and returns it.
ReadInt()	Reads an **int** from the binary file stream and returns it.
ReadLong()	Reads an **long** from the binary file stream and returns it.
ReadSByte()	Reads an **sbyte** from the binary file stream and returns it.
ReadShort()	Reads a **short** from the binary file stream and returns it.
ReadSize(void*, int)	Reads exactly the given number of bytes from the connected file into the specified buffer. If could not read that much, throws an IOException.
ReadString()	Reads the length of a string (an **int**) followed by the contents of the string from the binary file stream and returns the string.
ReadUInt()	Reads a **uint** from the binary file stream and returns it.
ReadULong()	Reads a **ulong** from the binary file stream and returns it.
ReadUShort()	Reads a **ushort** from the binary file stream and returns it.
Seek(long, int)	Sets the current file position.

Tell()	Returns the current file position.
Write(const char*)	Writes a length (an **int**) of the given C-style string followed by the contents of the C-style string to the binary file stream.
Write(const string&)	Writes a length (an **int**) of the given string followed by the contents of the string to the binary file stream.
Write(void*, int)	Writes the contents of the given buffer to the binary file stream.
Write(bool)	Writes a Boolean value to the binary file stream.
Write(byte)	Writes a **byte** to the binary file stream.
Write(char)	Writes a character to the binary file stream.
Write(double)	Writes a **double** to the binary file stream.
Write(float)	Writes a **float** to the binary file stream.
Write(int)	Writes an **int** to the binary file stream.
Write(long)	Writes a **long** to the binary file stream.
Write(sbyte)	Writes an **sbyte** to the binary file stream.
Write(short)	Writes a **short** to the binary file stream.
Write(uint)	Writes a **uint** to the binary file stream.
Write(ulong)	Writes a **ulong** to the binary file stream.
Write(ushort)	Writes a **ushort** to the binary file stream.

4.8.1.1.1 BinaryFileStream(BinaryFileStream&&) Member Function

Move constructor.

Syntax

```
public nothrow BinaryFileStream(BinaryFileStream&& that);
```

Parameters

Name	Type	Description
that	BinaryFileStream&&	A binary file stream to move from.

4.8.1.1.2 BinaryFileStream(const string&, OpenMode) Member Function

Opens a binary file stream with the specified open mode and file name.

Syntax

```
public BinaryFileStream(const string& fileName_, OpenMode mode_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of file to open.
mode_	OpenMode	An open mode.

4.8.1.1.3 BinaryFileStream(const string&, OpenMode, int) Member Function

Opens a binary file stream with the specified open mode, file name and permission mode.

Syntax

```
public BinaryFileStream(const string& fileName_, OpenMode mode_, int pmode);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of file to open.
mode_	OpenMode	An open mode.
pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).

4.8.1.1.4 operator=(BinaryFileStream&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(BinaryFileStream&& that);
```

Parameters

Name	Type	Description
that	BinaryFileStream&&	A binary file stream to move from.

4.8.1.1.5 ~BinaryFileStream() Member Function

If the binary file stream is connected to an open file, closes the file.

Syntax

```
public nothrow ~BinaryFileStream();
```

4.8.1.1.6 Close() Member Function

If the binary file stream is connected to an open file, closes the file, otherwise throws CloseFileException.

Syntax

```
public void Close();
```

4.8.1.1.7 GetFileSize() Member Function

Returns the size of the file binary file stream is connected to.

Syntax

```
public long GetFileSize();
```

Returns

long

Returns the size of the file binary file stream is connected to.

4.8.1.1.8 Open(const string&, OpenMode, int) Member Function

Opens a file with the specified open mode, file name and permission mode and connects it with the binary file stream.

Syntax

```
public void Open(const string& fileName_, OpenMode mode_, int pmode);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the file to open.
mode_	OpenMode	An open mode.
pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).

Remarks

If the binary file stream was connected to a file before the call, that file is closed first.

4.8.1.1.9 Read(void*, int) Member Function

Reads at most given number of bytes from the connected file into the specified buffer.

Syntax

```
public int Read(void* buffer, int size);
```

Parameters

Name	Type	Description
buffer	void*	A buffer to read to.
size	int	The maximum number of bytes to read.

Returns

int

Returns the number of bytes read.

4.8.1.1.10 ReadBool() Member Function

Reads a Boolean value from the binary file stream and returns it.

Syntax

```
public bool ReadBool();
```

Returns

bool

Returns the read Boolean value.

4.8.1.1.11 ReadByte() Member Function

Reads a **byte** from the binary file stream and returns it.

Syntax

```
public byte ReadByte();
```

Returns

byte

Returns the read **byte**.

4.8.1.1.12 ReadChar() Member Function

Reads a character from the binary file stream and returns it.

Syntax

```
public char ReadChar();
```

Returns

char

Returns the read character.

4.8.1.1.13 ReadDouble() Member Function

Reads a **double** from the binary file stream and returns it.

Syntax

```
public double ReadDouble();
```

Returns

double

Returns the read **double**.

4.8.1.1.14 ReadFloat() Member Function

Reads a **float** from the binary file stream and returns it.

Syntax

```
public float ReadFloat();
```

Returns

float

Returns the read **float**.

4.8.1.1.15 ReadInt() Member Function

Reads an **int** from the binary file stream and returns it.

Syntax

```
public int ReadInt();
```

Returns

int

Returns the read **int**.

4.8.1.1.16 ReadLong() Member Function

Reads an **long** from the binary file stream and returns it.

Syntax

```
public long ReadLong();
```

Returns

long

Returns the read **long**.

4.8.1.1.17 ReadSByte() Member Function

Reads an **sbyte** from the binary file stream and returns it.

Syntax

```
public sbyte ReadSByte();
```

Returns

sbyte

Returns the read **sbyte**.

4.8.1.1.18 ReadShort() Member Function

Reads a **short** from the binary file stream and returns it.

Syntax

```
public short ReadShort();
```

Returns

short

Returns the read **short**.

4.8.1.1.19 ReadSize(void*, int) Member Function

Reads exactly the given number of bytes from the connected file into the specified buffer. If could not read that much, throws an IOException.

Syntax

```
public void ReadSize(void* buffer, int size);
```

Parameters

Name	Type	Description
buffer	void*	A buffer to read to.
size	int	The number of bytes to read.

4.8.1.1.20 ReadString() Member Function

Reads the length of a string (an **int**) followed by the contents of the string from the binary file stream and returns the string.

Syntax

```
public string ReadString();
```

Returns

string

Returns the read string.

4.8.1.1.21 ReadUInt() Member Function

Reads a **uint** from the binary file stream and returns it.

Syntax

```
public uint ReadUInt();
```

Returns

uint

Returns the read **uint**.

4.8.1.1.22 ReadULong() Member Function

Reads a **ulong** from the binary file stream and returns it.

Syntax

```
public ulong ReadULong();
```

Returns

ulong

Returns the read **ulong**.

4.8.1.1.23 ReadUShort() Member Function

Reads a **ushort** from the binary file stream and returns it.

Syntax

```
public ushort ReadUShort();
```

Returns

ushort

Returns the read **ushort**.

4.8.1.1.24 Seek(long, int) Member Function

Sets the current file position.

Syntax

```
public long Seek(long offset, int origin);
```

Parameters

Name	Type	Description
offset	long	An offset.
origin	int	One of the constants SEEK_SET, SEEK_CUR, and SEEK_END.

Returns

long

Returns the current file position.

4.8.1.1.25 Tell() Member Function

Returns the current file position.

Syntax

```
public long Tell();
```

Returns

long

Returns the current file position.

4.8.1.1.26 Write(const char*) Member Function

Writes a length (an **int**) of the given C-style string followed by the contents of the C-style string to the binary file stream.

Syntax

```
public void Write(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.1.1.27 Write(const string&) Member Function

Writes a length (an **int**) of the given string followed by the contents of the string to the binary file stream.

Syntax

```
public void Write(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.1.1.28 Write(void*, int) Member Function

Writes the contents of the given buffer to the binary file stream.

Syntax

```
public void Write(void* buffer, int size);
```

Parameters

Name	Type	Description
buffer	void*	A buffer of data to write.
size	int	The size of the buffer.

4.8.1.1.29 Write(bool) Member Function

Writes a Boolean value to the binary file stream.

Syntax

```
public void Write(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

4.8.1.1.30 Write(byte) Member Function

Writes a **byte** to the binary file stream.

Syntax

```
public void Write(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.1.1.31 Write(char) Member Function

Writes a character to the binary file stream.

Syntax

```
public void Write(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.1.1.32 Write(double) Member Function

Writes a **double** to the binary file stream.

Syntax

```
public void Write(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.1.1.33 Write(float) Member Function

Writes a **float** to the binary file stream.

Syntax

```
public void Write(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.1.1.34 Write(int) Member Function

Writes an **int** to the binary file stream.

Syntax

```
public void Write(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.1.1.35 Write(long) Member Function

Writes a **long** to the binary file stream.

Syntax

```
public void Write(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.1.1.36 Write(sbyte) Member Function

Writes an **sbyte** to the binary file stream.

Syntax

```
public void Write(sbyte s);
```

Parameters

Name	Type	Description
s	sbyte	An **sbyte** to write.

4.8.1.1.37 Write(short) Member Function

Writes a **short** to the binary file stream.

Syntax

```
public void Write(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.1.1.38 Write(uint) Member Function

Writes a **uint** to the binary file stream.

Syntax

```
public void Write(uint u);
```

Parameters

Name	Type	Description
u	uint	A **uint** to write.

4.8.1.1.39 Write(ulong) Member Function

Writes a **ulong** to the binary file stream.

Syntax

```
public void Write(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.1.1.40 Write(ushort) Member Function

Writes a **ushort** to the binary file stream.

Syntax

```
public void Write(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.2 CloseFileException Class

An exception thrown when closing a file fails.

Syntax

```
public class CloseFileException;
```

Base Class

Exception

4.8.2.1 Member Functions

Member Function	Description
CloseFileException(CloseFileException&&)	Move constructor.
CloseFileException(const string&)	Constructor. Initializes the close file exception with the given error message.
CloseFileException(const CloseFileException&)	Copy constructor.
operator=(const CloseFileException&)	Copy assignment.
operator=(CloseFileException&&)	Move assignment.
~CloseFileException()	Destructor.

4.8.2.1.1 CloseFileException(CloseFileException&&) Member Function

Move constructor.

Syntax

```
public nothrow CloseFileException(CloseFileException&& that);
```

Parameters

Name	Type	Description
that	CloseFileException&&	A close file exception to move from.

4.8.2.1.2 CloseFileException(const string&) Member Function

Constructor. Initializes the close file exception with the given error message.

Syntax

```
public CloseFileException(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	An error message.

4.8.2.1.3 CloseFileException(const CloseFileException&) Member Function

Copy constructor.

Syntax

```
public nothrow CloseFileException(const CloseFileException& that);
```

Parameters

Name	Type	Description
that	const CloseFileException&	A close file exception to copy.

4.8.2.1.4 operator=(const CloseFileException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const CloseFileException& that);
```

Parameters

Name	Type	Description
that	const CloseFileException&	A close file exception to assign.

4.8.2.1.5 operator=(CloseFileException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(CloseFileException&& that);
```

Parameters

Name	Type	Description
that	CloseFileException&&	A close file exception to move from.

4.8.2.1.6 ∼CloseFileException() Member Function

Destructor.

Syntax

```
public override nothrow ~CloseFileException();
```

4.8.3 IOBuffer Class

A handle to dynamically allocated memory.

Syntax

```
public class IOBuffer;
```

4.8.3.1 Member Functions

Member Function	Description
IOBuffer(IOBuffer&&)	Move constructor.
IOBuffer(uint)	Constructor. Allocates given number of bytes for the I/O buffer.
operator=(IOBuffer&&)	Move assignment.
~IOBuffer()	Destructor. Releases the allocated memory back to system.
Mem() const	Returns a generic pointer to the allocated memory.
Size() const	Returns the size of the I/O buffer.

4.8.3.1.1 IOBuffer(IOBuffer&&) Member Function

Move constructor.

Syntax

```
public nothrow IOBuffer(IOBuffer&& that);
```

Parameters

Name	Type	Description
that	IOBuffer&&	An I/O buffer to move from.

4.8.3.1.2 IOBuffer(uint) Member Function

Constructor. Allocates given number of bytes for the I/O buffer.

Syntax

```
public nothrow IOBuffer(uint size_);
```

Parameters

Name	Type	Description
size_	uint	The number of bytes to allocate.

4.8.3.1.3 operator=(IOBuffer&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(IOBuffer&& that);
```

Parameters

Name	Type	Description
that	IOBuffer&&	An I/O buffer to move from.

4.8.3.1.4 ~IOBuffer() Member Function

Destructor. Releases the allocated memory back to system.

Syntax

```
public nothrow ~IOBuffer();
```

4.8.3.1.5 Mem() const Member Function

Returns a generic pointer to the allocated memory.

Syntax

```
public inline nothrow void* Mem() const;
```

Returns

void*

Returns a generic pointer to the allocated memory.

4.8.3.1.6 Size() const Member Function

Returns the size of the I/O buffer.

Syntax

```
public inline nothrow uint Size() const;
```

Returns

uint

Returns the size of the I/O buffer.

4.8.4 IOException Class

An exception thrown when an I/O operation fails.

Syntax

```
public class IOException;
```

Base Class

Exception

4.8.4.1 Member Functions

Member Function	Description
IOException(const string&)	Constructor. Initializes the exception with the given error message.
IOException(const IOException&)	Copy constructor.
IOException(IOException&&)	Move constructor.
operator=(const IOException&)	Copy assignment.
operator=(IOException&&)	Move assignment.
~IOException()	Destructor.

4.8.4.1.1 IOException(const string&) Member Function

Constructor. Initializes the exception with the given error message.

Syntax

```
public IOException(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	An error message.

4.8.4.1.2 IOException(const IOException&) Member Function

Copy constructor.

Syntax

```
public nothrow IOException(const IOException& that);
```

Parameters

Name	Type	Description
that	const IOException&	An I/O exception to copy.

4.8.4.1.3 IOException(IOException&&) Member Function

Move constructor.

Syntax

```
public nothrow IOException(IOException&& that);
```

Parameters

Name	Type	Description
that	IOException&&	An I/O exception to move from.

4.8.4.1.4 operator=(const IOException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const IOException& that);
```

Parameters

Name	Type	Description
that	const IOException&	An I/O exception to assign.

4.8.4.1.5 operator=(IOException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(IOException&& that);
```

Parameters

Name	Type	Description
that	IOException&&	An I/O exception to move from.

4.8.4.1.6 ~IOException() Member Function

Destructor.

Syntax

```
public override nothrow ~IOException();
```

4.8.5 InputFileStream Class

A stream of characters connected to an input file.

Syntax

```
public class InputFileStream;
```

Base Class

InputStream

4.8.5.1 Member Functions

Member Function	Description
InputFileStream()	Default constructor. Initializes the input file stream and connects it to standard input stream.
InputFileStream(const string&)	Constructor. Initializes the input file stream and connects it to a file with the given name.
InputFileStream(InputFileStream&&)	Move constructor.
InputFileStream(const string&, uint)	Constructor. Initializes the input file stream with the given buffer size and connects it to a file with the given name.
InputFileStream(int, uint)	Constructor. Initializes the input file stream with the given buffer size and connects it to the file with the given file handle.
operator=(InputFileStream&&)	Move assignment.
~InputFileStream()	Destructor. If the input file stream is connected to an open file, closes the file.
Close()	If the input file stream is connected to an open file, closes the file, otherwise throws CloseFileException;
EndOfStream() const	Returns true if the end of the stream is encountered, false otherwise.

FileName() const	Returns the name of the file input file stream is connected to.
Handle() const	Returns the file handler the input file stream is connected to.
Open(const string&)	Opens an input file and connects it to the input file stream.
ReadLine()	Reads a line of text from the input file stream and returns it.
ReadToEnd()	Reads the content of the input file into a string and returns it.

4.8.5.1.1 InputFileStream() Member Function

Default constructor. Initializes the input file stream and connects it to standard input stream.

Syntax

```
public nothrow InputFileStream();
```

4.8.5.1.2 InputFileStream(const string&) Member Function

Constructor. Initializes the input file stream and connects it to a file with the given name.

Syntax

```
public InputFileStream(const string& fileName_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the file to open.

4.8.5.1.3 InputFileStream(InputFileStream&&) Member Function

Move constructor.

Syntax

```
public nothrow InputFileStream(InputFileStream&& that);
```

Parameters

Name	Type	Description
that	InputFileStream&&	An input file stream to move from.

4.8.5.1.4 InputFileStream(const string&, uint) Member Function

Constructor. Initializes the input file stream with the given buffer size and connects it to a file with the given name.

Syntax

```
public InputFileStream(const string& fileName_, uint bufferSize_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the file to open.
bufferSize_	uint	The size of the input buffer.

4.8.5.1.5 InputFileStream(int, uint) Member Function

Constructor. Initializes the input file stream with the given buffer size and connects it to the file with the given file handle.

Syntax

```
public nothrow InputFileStream(int handle_, uint bufferSize_);
```

Parameters

Name	Type	Description
handle_	int	A file handle.
bufferSize_	uint	The size of the input buffer.

4.8.5.1.6 operator=(InputFileStream&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(InputFileStream&& that);
```

Parameters

Name	Type	Description
that	InputFileStream&&	An input file stream to move from.

4.8.5.1.7 ~InputFileStream() Member Function

Destructor. If the input file stream is connected to an open file, closes the file.

Syntax

```
public override nothrow ~InputFileStream();
```

4.8.5.1.8 Close() Member Function

If the input file stream is connected to an open file, closes the file, otherwise throws CloseFileException;

Syntax

```
public void Close();
```

4.8.5.1.9 EndOfStream() const Member Function

Returns true if the end of the stream is encountered, false otherwise.

Syntax

```
public override bool EndOfStream() const;
```

Returns

bool

Returns true if the end of the stream is encountered, false otherwise.

4.8.5.1.10 FileName() const Member Function

Returns the name of the file input file stream is connected to.

Syntax

```
public const string& FileName() const;
```

Returns

const string&

Returns the name of the file input file stream is connected to.

4.8.5.1.11 Handle() const Member Function

Returns the file handler the input file stream is connected to.

Syntax

```
public int Handle() const;
```

Returns

int

Returns the file handler the input file stream is connected to.

4.8.5.1.12 Open(const string&) Member Function

Opens an input file and connects it to the input file stream.

Syntax

```
public void Open(const string& fileName_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the file to open.

Remarks

If the input file stream was connected to a file before the call, that file is closed first.

4.8.5.1.13 ReadLine() Member Function

Reads a line of text from the input file stream and returns it.

Syntax

```
public override string ReadLine();
```

Returns

string

Returns the line of text read.

4.8.5.1.14 ReadToEnd() Member Function

Reads the content of the input file into a string and returns it.

Syntax

```
public override string ReadToEnd();
```

Returns

string

Returns the contents of the file read.

4.8.6 InputStream Class

An abstract base class for input stream classes.

Syntax

```
public abstract class InputStream;
```

4.8.6.1 Member Functions

Member Function	Description
InputStream()	Default constructor.
~InputStream()	Destructor.
EndOfStream() const	Abstract member function for returning the end-of-file status.
ReadLine()	Abstract member function for reading a line of text from the input stream.
ReadToEnd()	Abstract member function for reading the contents of the stream into a string.

4.8.6.1.1 InputStream() Member Function

Default constructor.

Syntax

```
public nothrow InputStream();
```

4.8.6.1.2 ~InputStream() Member Function

Destructor.

Syntax

```
public virtual nothrow ~InputStream();
```

4.8.6.1.3 EndOfStream() const Member Function

Abstract member function for returning the end-of-file status.

Syntax

```
public abstract bool EndOfStream() const;
```

Returns

bool

Returns true if the end of the stream is encountered, false otherwise.

4.8.6.1.4 ReadLine() Member Function

Abstract member function for reading a line of text from the input stream.

Syntax

```
public abstract string ReadLine();
```

Returns

string

Returns the line read.

4.8.6.1.5 ReadToEnd() Member Function

Abstract member function for reading the contents of the stream into a string.

Syntax

```
public abstract string ReadToEnd();
```

Returns

string

Returns the contents of the stream.

4.8.7 InputStringStream Class

A class for reading from a string.

Syntax

```
public class InputStringStream;
```

Base Class

InputStream

4.8.7.1 Member Functions

Member Function	Description
InputStringStream()	Default constructor.
InputStringStream(const string&)	Constructor. Initializes the input string stream with the given string.
InputStringStream(InputStringStream&&)	Move constructor.
operator=(InputStringStream&&)	Move assignment.
~InputStringStream()	Destructor.
EndOfStream() const	Returns true if the end of the string has been encountered, false otherwise.
GetStr() const	Returns the contained string.
ReadLine()	Reads a line of text from the string and returns it.
ReadToEnd()	Returns the contents of the rest of the string.
SetStr(const string&)	Sets the contained string.

4.8.7.1.1 InputStringStream() Member Function

Default constructor.

Syntax

```
public nothrow InputStringStream();
```

4.8.7.1.2 InputStringStream(const string&) Member Function

Constructor. Initializes the input string stream with the given string.

Syntax

```
public nothrow InputStringStream(const string& str_);
```

Parameters

Name	Type	Description
str_	const string&	A string to read from.

4.8.7.1.3 InputStringStream(InputStringStream&&) Member Function

Move constructor.

Syntax

```
public nothrow InputStringStream(InputStringStream&& that);
```

Parameters

Name	Type	Description
that	InputStringStream&&	An input string stream to move from.

4.8.7.1.4 operator=(InputStringStream&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(InputStringStream&& that);
```

Parameters

Name	Type	Description
that	InputStringStream&&	An input string stream to move from.

4.8.7.1.5 ~InputStringStream() Member Function

Destructor.

Syntax

```
public override nothrow ~InputStringStream();
```

4.8.7.1.6 EndOfStream() const Member Function

Returns true if the end of the string has been encountered, false otherwise.

Syntax

```
public override nothrow bool EndOfStream() const;
```

Returns

bool

Returns true if the end of the string has been encountered, false otherwise.

4.8.7.1.7 GetStr() const Member Function

Returns the contained string.

Syntax

```
public nothrow const string& GetStr() const;
```

Returns

const string&

Returns the contained string.

4.8.7.1.8 ReadLine() Member Function

Reads a line of text from the string and returns it.

Syntax

```
public override nothrow string ReadLine();
```

Returns

string

Returns the line read.

4.8.7.1.9 ReadToEnd() Member Function

Returns the contents of the rest of the string.

Syntax

```
public override nothrow string ReadToEnd();
```

Returns

string

Returns the contents of the rest of the string.

4.8.7.1.10 SetStr(const string&) Member Function

Sets the contained string.

Syntax

```
public nothrow void SetStr(const string& str_);
```

Parameters

Name	Type	Description
str_	const string&	A string.

4.8.8 InvalidPathException Class

An exception class that is thrown if a path contains too many .. components.

Syntax

```
public class InvalidPathException;
```

Base Class

Exception

4.8.8.1 Member Functions

Member Function	Description
InvalidPathException(const string&)	Constructor. Initializes the exception with the given error message.
InvalidPathException(const InvalidPathException&)	Copy constructor.
InvalidPathException(InvalidPathException&&)	Move constructor.
operator=(const InvalidPathException&)	Copy assignment.
operator=(InvalidPathException&&)	Move assignment.
~InvalidPathException()	Destructor.

4.8.8.1.1 InvalidPathException(const string&) Member Function

Constructor. Initializes the exception with the given error message.

Syntax

```
public InvalidPathException(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	An error message.

4.8.8.1.2 InvalidPathException(const InvalidPathException&) Member Function

Copy constructor.

Syntax

```
public nothrow InvalidPathException(const InvalidPathException& that);
```

Parameters

Name	Type	Description
that	const InvalidPathException&	An exception to copy.

4.8.8.1.3 InvalidPathException(InvalidPathException&&) Member Function

Move constructor.

Syntax

```
public nothrow InvalidPathException(InvalidPathException&& that);
```

Parameters

Name	Type	Description
that	InvalidPathException&&	An exception to move from.

4.8.8.1.4 operator=(const InvalidPathException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const InvalidPathException& that);
```

Parameters

Name	Type	Description
that	const InvalidPathException&	An exception to assign.

4.8.8.1.5 operator=(InvalidPathException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(InvalidPathException&& that);
```

Parameters

Name	Type	Description
that	InvalidPathException&&	An exception to move from.

4.8.8.1.6 ~InvalidPathException() Member Function

Destructor.

Syntax

```
public override nothrow ~InvalidPathException();
```

4.8.9 OpenFileException Class

An exception class thrown if opening a file fails.

Syntax

```
public class OpenFileException;
```

Base Class

Exception

4.8.9.1 Member Functions

Member Function	Description
OpenFileException(const string&)	Constructor. Initializes the exception with the given error message.
OpenFileException(const OpenFileException&)	Copy constructor.
OpenFileException(OpenFileException&&)	Move constructor.
operator=(const OpenFileException&)	Copy assignment.
operator=(OpenFileException&&)	Move assignment.
~OpenFileException()	Destructor.

4.8.9.1.1 OpenFileException(const string&) Member Function

Constructor. Initializes the exception with the given error message.

Syntax

```
public OpenFileException(const string& message_);
```

Parameters

Name	Type	Description
message_	const string&	An error message.

4.8.9.1.2 OpenFileException(const OpenFileException&) Member Function

Copy constructor.

Syntax

```
public nothrow OpenFileException(const OpenFileException& that);
```

Parameters

Name	Type	Description
that	const OpenFileException&	An exception to copy.

4.8.9.1.3 OpenFileException(OpenFileException&&) Member Function

Move constructor.

Syntax

```
public nothrow OpenFileException(OpenFileException&& that);
```

Parameters

Name	Type	Description
that	OpenFileException&&	An exception to mvoe from.

4.8.9.1.4 operator=(const OpenFileException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const OpenFileException& that);
```

Parameters

Name	Type	Description
that	const OpenFileException&	An exception to assign.

4.8.9.1.5 operator=(OpenFileException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(OpenFileException&& that);
```

Parameters

Name	Type	Description
that	OpenFileException&&	An exception to move from.

4.8.9.1.6 ∼OpenFileException() Member Function

Destructor.

Syntax

```
public override nothrow ~OpenFileException();
```

4.8.10 OutputFileStream Class

A stream of characters connected to an output file.

Syntax

```
public class OutputFileStream;
```

Base Class

OutputStream

4.8.10.1 Member Functions

Member Function	Description
OutputFileStream()	Default constructor. Connects the output file stream with standard output stream.
OutputFileStream(const string&)	Constructor. Connects the output file stream with the output file of the given name.
OutputFileStream(OutputFileStream&&)	Move constructor.
OutputFileStream(const string&, bool)	Constructor. Connects the output file stream to an output file of the given name.
OutputFileStream(const string&, int)	Constructor. Connects the output file stream to an output file with the given name using given permissions.
OutputFileStream(const string&, int, bool)	Constructor. Connects the output file stream to an output file with the given name using given permissions.
OutputFileStream(int)	Constructor. Connects the output file stream to an output file with the given file handle.
operator=(OutputFileStream&&)	Move assignment.
~OutputFileStream()	Destructor. If the output file stream is connected to an open file, the file is closed.
Close()	If the output file stream is connected to an open file, the file is closed, otherwise throws CloseFileException.

FileName() const	Returns the name of the output file the output file stream is connected to.
Handle() const	Returns the file handle of the file the output file stream is connected to.
Open(const string&)	Opens the given output file and connects it to the output file stream.
Open(const string&, bool)	Opens the given output file and connects it to the output file stream.
Open(const string&, int)	Opens the given output file and connects it to the output file stream using given permissions.
Open(const string&, int, bool)	Opens the given output file and connects it to the output file stream using given permissions.
Write(const string&)	Writes the given string to the output file stream.
Write(const char*)	Writes the given C-style string to the output file stream.
Write(bool)	Writes the given Boolean value to the output file stream.
Write(byte)	Writes the given **byte** to the output file stream.
Write(char)	Writes the given character to the output file stream.
Write(double)	Writes the given **double** value to the output file stream.
Write(float)	Writes the given **float** value to the output file stream.

Write(int)	Writes the given **int** value to the output file stream.
Write(long)	Writes the given **long** value to the output file stream.
Write(sbyte)	Writes the given **sbyte** value to the output file stream.
Write(short)	Writes the given **short** value to the output file stream.
Write(uint)	Writes the given **uint** value to the output file stream.
Write(ulong)	Writes the given **ulong** value to the output file stream.
Write(ushort)	Writes the given **ushort** value to the output file stream.
WriteLine()	Writes a new line to the output file stream.
WriteLine(const char*)	Writes the given C-style string followed by a new line to the output file stream.
WriteLine(const string&)	Writes the given string followed by a new line to the output file stream.
WriteLine(bool)	Writes the given Boolean value followed by a new line to the output file stream.
WriteLine(byte)	Writes the given **byte** value followed by a new line to the output file stream.
WriteLine(char)	Writes the given character followed by a new line to the output file stream.
WriteLine(double)	Writes the given **double** value followed by a new line to the output file stream.

WriteLine(float)	Writes the given **float** value followed by a new line to the output file stream.
WriteLine(int)	Writes the given **int** value followed by a new line to the output file stream.
WriteLine(long)	Writes the given **long** value followed by a new line to the output file stream.
WriteLine(sbyte)	Writes the given **sbyte** value followed by a new line to the output file stream.
WriteLine(short)	Writes the given **short** value followed by a new line to the output file stream.
WriteLine(uint)	Writes the given **uint** value followed by a new line to the output file stream.
WriteLine(ulong)	Writes the given **ulong** value followed by a new line to the output file stream.
WriteLine(ushort)	Writes the given **ushort** value followed by a new line to the output file stream.

4.8.10.1.1 OutputFileStream() Member Function

Default constructor. Connects the output file stream with standard output stream.

Syntax

```
public nothrow OutputFileStream();
```

4.8.10.1.2 OutputFileStream(const string&) Member Function

Constructor. Connects the output file stream with the output file of the given name.

Syntax

```
public OutputFileStream(const string& fileName_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.

Remarks

If the file does not exist, it is created. If the file exists, it is truncated to zero length.

4.8.10.1.3 OutputFileStream(OutputFileStream&&) Member Function

Move constructor.

Syntax

```
public nothrow OutputFileStream(OutputFileStream&& that);
```

Parameters

Name	Type	Description
that	OutputFileStream&&	An output file stream to move from.

4.8.10.1.4 OutputFileStream(const string&, bool) Member Function

Constructor. Connects the output file stream to an output file of the given name.

Syntax

```
public OutputFileStream(const string& fileName_, bool append);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.
append	bool	If true, the file is opened for appending.

Remarks

If the append parameter is true, opens the file for appending.

4.8.10.1.5 OutputFileStream(const string&, int) Member Function

Constructor. Connects the output file stream to an output file with the given name using given permissions.

Syntax

```
public OutputFileStream(const string& fileName_, int pmode);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.

pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).

4.8.10.1.6 OutputFileStream(const string&, int, bool) Member Function

Constructor. Connects the output file stream to an output file with the given name using given permissions.

Syntax

```
public OutputFileStream(const string& fileName_, int pmode, bool append);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.
pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).
append	bool	If true, the file is opened for appending.

Remarks

If the append parameter is true, opens the file for appending.

4.8.10.1.7 OutputFileStream(int) Member Function

Constructor. Connects the output file stream to an output file with the given file handle.

Syntax

```
public nothrow OutputFileStream(int handle_);
```

Parameters

Name	Type	Description
handle_	int	A file handle.

4.8.10.1.8 operator=(OutputFileStream&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(OutputFileStream&& that);
```

Parameters

Name	Type	Description
that	OutputFileStream&&	An output file stream to move from.

4.8.10.1.9 ~OutputFileStream() Member Function

Destructor. If the output file stream is connected to an open file, the file is closed.

Syntax

```
public override nothrow ~OutputFileStream();
```

4.8.10.1.10 Close() Member Function

If the output file stream is connected to an open file, the file is closed, otherwise throws CloseFileException.

Syntax

```
public void Close();
```

4.8.10.1.11 FileName() const Member Function

Returns the name of the output file the output file stream is connected to.

Syntax

```
public const string& FileName() const;
```

Returns

const string&

Returns the name of the output file.

4.8.10.1.12 Handle() const Member Function

Returns the file handle of the file the output file stream is connected to.

Syntax

```
public int Handle() const;
```

Returns

int

Returns the file handle.

4.8.10.1.13 Open(const string&) Member Function

Opens the given output file and connects it to the output file stream.

Syntax

```
public void Open(const string& fileName_);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.

Remarks

If the output file stream was connected to an open file before the operation, that file is closed first. If the given file does not exist, it is created. If the given file exists, it is truncated to zero length.

4.8.10.1.14 Open(const string&, bool) Member Function

Opens the given output file and connects it to the output file stream.

Syntax

```
public void Open(const string& fileName_, bool append);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.
append	bool	If true, the file is opened for appending.

Remarks

If the append parameter is true, opens the file for appending.

4.8.10.1.15 Open(const string&, int) Member Function

Opens the given output file and connects it to the output file stream using given permissions.

Syntax

```
public void Open(const string& fileName_, int pmode);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.
pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).

4.8.10.1.16 Open(const string&, int, bool) Member Function

Opens the given output file and connects it to the output file stream using given permissions.

Syntax

```
public void Open(const string& fileName_, int pmode, bool append);
```

Parameters

Name	Type	Description
fileName_	const string&	The name of the output file.
pmode	int	A permission mode that is formed by ORing together following constants: S_IREAD, S_IWRITE (Windows); S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH (Unix).

append	bool	If true, the file is opened for appending.

Remarks

If the append parameter is true, opens the file for appending.

4.8.10.1.17 Write(const string&) Member Function

Writes the given string to the output file stream.

Syntax

```
public override void Write(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.10.1.18 Write(const char*) Member Function

Writes the given C-style string to the output file stream.

Syntax

```
public override void Write(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.10.1.19 Write(bool) Member Function

Writes the given Boolean value to the output file stream.

Syntax

```
public override void Write(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

Remarks

If the value is true, writes “true”, otherwise writes “false”.

4.8.10.1.20 Write(byte) Member Function

Writes the given **byte** to the output file stream.

Syntax

```
public override void Write(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.10.1.21 Write(char) Member Function

Writes the given character to the output file stream.

Syntax

```
public override void Write(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.10.1.22 Write(double) Member Function

Writes the given **double** value to the output file stream.

Syntax

```
public override void Write(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.10.1.23 Write(float) Member Function

Writes the given **float** value to the output file stream.

Syntax

```
public override void Write(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.10.1.24 Write(int) Member Function

Writes the given **int** value to the output file stream.

Syntax

```
public override void Write(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.10.1.25 Write(long) Member Function

Writes the given **long** value to the output file stream.

Syntax

```
public override void Write(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.10.1.26 Write(sbyte) Member Function

Writes the given **sbyte** value to the output file stream.

Syntax

```
public override void Write(sbyte s);
```

Parameters

Name	Type	Description
s	sbyte	An **sbyte** to write.

4.8.10.1.27 Write(short) Member Function

Writes the given **short** value to the output file stream.

Syntax

```
public override void Write(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.10.1.28 Write(uint) Member Function

Writes the given **uint** value to the output file stream.

Syntax

```
public override void Write(uint i);
```

Parameters

Name	Type	Description
i	uint	A **uint** to write.

4.8.10.1.29 Write(ulong) Member Function

Writes the given **ulong** value to the output file stream.

Syntax

```
public override void Write(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.10.1.30 Write(ushort) Member Function

Writes the given **ushort** value to the output file stream.

Syntax

```
public override void Write(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.10.1.31 WriteLine() Member Function

Writes a new line to the output file stream.

Syntax

```
public override void WriteLine();
```

4.8.10.1.32 WriteLine(const char*) Member Function

Writes the given C-style string followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.10.1.33 WriteLine(const string&) Member Function

Writes the given string followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.10.1.34 WriteLine(bool) Member Function

Writes the given Boolean value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

4.8.10.1.35 WriteLine(byte) Member Function

Writes the given **byte** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.10.1.36 WriteLine(char) Member Function

Writes the given character followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.10.1.37 WriteLine(double) Member Function

Writes the given **double** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.10.1.38 WriteLine(float) Member Function

Writes the given **float** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.10.1.39 WriteLine(int) Member Function

Writes the given **int** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.10.1.40 WriteLine(long) Member Function

Writes the given **long** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.10.1.41 WriteLine(sbyte) Member Function

Writes the given **sbyte** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(sbyte s);
```

Parameters

Name	Type	Description
s	sbyte	An **sbyte** to write.

4.8.10.1.42 WriteLine(short) Member Function

Writes the given **short** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.10.1.43 WriteLine(uint) Member Function

Writes the given **uint** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(uint u);
```

Parameters

Name	Type	Description
u	uint	A **uint** to write.

4.8.10.1.44 WriteLine(ulong) Member Function

Writes the given **ulong** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.10.1.45 WriteLine(ushort) Member Function

Writes the given **ushort** value followed by a new line to the output file stream.

Syntax

```
public override void WriteLine(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.11 OutputStream Class

An abstract base class for output stream classes.

Syntax

```
public abstract class OutputStream;
```

4.8.11.1 Member Functions

Member Function	Description
OutputStream()	Default constructor.
~OutputStream()	Destructor.
Write(const string&)	Writes a string to the output stream.
Write(const char*)	Writes a C-style string to the output steram.
Write(bool)	Writes a Boolean value to output stream.
Write(byte)	Writes a **byte** to the output stream.
Write(char)	Writes a character to the output stream.
Write(double)	Writes a **double** to the output stream.
Write(float)	Writes a **float** to the output stream.
Write(int)	Writes an **int** to the output stream.
Write(long)	Writes a **long** to the output stream.
Write(sbyte)	Writes an **sbyte** to the output stream.
Write(short)	Writes a **short** to the output stream.
Write(uint)	Writes a **uint** to the output stream.
Write(ulong)	Writes a **ulong** to the output stream.
Write(ushort)	Writes a **ushort** to the output stream.
WriteLine()	Writes a new line to the output stream.

WriteLine(const string&)	Writes a string followed by a new line to the output stream.
WriteLine(const char*)	Writes a C-style string followed by a new line to the output stream.
WriteLine(bool)	Writes a Boolean followed by a new line to the output stream.
WriteLine(byte)	Writes a **byte** followed by a new line to the output stream.
WriteLine(char)	Writes a character followed by a new line to the output stream.
WriteLine(double)	Writes a **double** followed by a new line to the output stream.
WriteLine(float)	Writes a **float** followed by a new line to the output stream.
WriteLine(int)	Writes an **int** followed by a new line to the output stream.
WriteLine(long)	Writes a **long** followed by a new line to the output stream.
WriteLine(sbyte)	Writes an **sbyte** followed by a new line to the output stream.
WriteLine(short)	Writes a **short** followed by a new line to the output stream.
WriteLine(uint)	Writes a **uint** followed by a new line to the output stream.
WriteLine(ulong)	Writes a **ulong** followed by a new line to the output stream.
WriteLine(ushort)	Writes a **ushort** followed by a new line to the output stream.

4.8.11.1.1 OutputStream() Member Function

Default constructor.

Syntax

```
public nothrow OutputStream();
```

4.8.11.1.2 ~OutputStream() Member Function

Destructor.

Syntax

```
public virtual nothrow ~OutputStream();
```

4.8.11.1.3 Write(const string&) Member Function

Writes a string to the output stream.

Syntax

```
public abstract void Write(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.11.1.4 Write(const char*) Member Function

Writes a C-style string to the output steram.

Syntax

```
public abstract void Write(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.11.1.5 Write(bool) Member Function

Writes a Boolean value to output stream.

Syntax

```
public abstract void Write(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean to write.

4.8.11.1.6 Write(byte) Member Function

Writes a **byte** to the output stream.

Syntax

```
public abstract void Write(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.11.1.7 Write(char) Member Function

Writes a character to the output stream.

Syntax

```
public abstract void Write(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.11.1.8 Write(double) Member Function

Writes a **double** to the output stream.

Syntax

```
public abstract void Write(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.11.1.9 Write(float) Member Function

Writes a **float** to the output stream.

Syntax

```
public abstract void Write(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.11.1.10 Write(int) Member Function

Writes an **int** to the output stream.

Syntax

```
public abstract void Write(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.11.1.11 Write(long) Member Function

Writes a **long** to the output stream.

Syntax

```
public abstract void Write(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.11.1.12 Write(sbyte) Member Function

Writes an **sbyte** to the output stream.

Syntax

```
public abstract void Write(sbyte b);
```

Parameters

Name	Type	Description
b	sbyte	An **sbyte** to write.

4.8.11.1.13 Write(short) Member Function

Writes a **short** to the output stream.

Syntax

```
public abstract void Write(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.11.1.14 Write(uint) Member Function

Writes a **uint** to the output stream.

Syntax

```
public abstract void Write(uint i);
```

Parameters

Name	Type	Description
i	uint	A **uint** to write.

4.8.11.1.15 Write(ulong) Member Function

Writes a **ulong** to the output stream.

Syntax

```
public abstract void Write(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.11.1.16 Write(ushort) Member Function

Writes a **ushort** to the output stream.

Syntax

```
public abstract void Write(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.11.1.17 WriteLine() Member Function

Writes a new line to the output stream.

Syntax

```
public abstract void WriteLine();
```

4.8.11.1.18 WriteLine(const string&) Member Function

Writes a string followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.11.1.19 WriteLine(const char*) Member Function

Writes a C-style string followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.11.1.20 WriteLine(bool) Member Function

Writes a Boolean followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean to write.

4.8.11.1.21 WriteLine(byte) Member Function

Writes a **byte** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.11.1.22 WriteLine(char) Member Function

Writes a character followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.11.1.23 WriteLine(double) Member Function

Writes a **double** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.11.1.24 WriteLine(float) Member Function

Writes a **float** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.11.1.25 WriteLine(int) Member Function

Writes an **int** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.11.1.26 WriteLine(long) Member Function

Writes a **long** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.11.1.27 WriteLine(sbyte) Member Function

Writes an **sbyte** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(sbyte b);
```

Parameters

Name	Type	Description
b	sbyte	An **sbyte** to write.

4.8.11.1.28 WriteLine(short) Member Function

Writes a **short** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.11.1.29 WriteLine(uint) Member Function

Writes a **uint** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(uint i);
```

Parameters

Name	Type	Description
i	uint	A **uint** to write.

4.8.11.1.30 WriteLine(ulong) Member Function

Writes a **ulong** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.11.1.31 WriteLine(ushort) Member Function

Writes a **ushort** followed by a new line to the output stream.

Syntax

```
public abstract void WriteLine(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.11.2 Nonmember Functions

Function	Description
operator<<<C>(OutputStream&, const C&)	Puts the values of a forward container containing integers to the given stream.
operator<<(OutputStream&, bool)	Puts a Boolean value to an output stream.
operator<<(OutputStream&, byte)	Puts a **byte** to an output stream.
operator<<(OutputStream&, char)	Puts a character to an output stream.
operator<<(OutputStream&, double)	Puts a **double** to an output stream.
operator<<(OutputStream&, float)	Puts a **float** to an output stream.
operator<<(OutputStream&, int)	Puts an **int** to an output stream.

operator<<(OutputStream&, long)	Puts a **long** to an output stream.
operator<<(OutputStream&, const string&)	Puts a String to an output stream.
operator<<(OutputStream&, const char*)	Puts a C-style string to an output stream.
operator<<(OutputStream&, EndLine)	Puts an end-of-line character to an output stream.
operator<<(OutputStream&, sbyte)	Puts an **sbyte** to an output stream.
operator<<(OutputStream&, short)	Puts a **short** to an output stream.
operator<<(OutputStream&, uint)	Puts a **uint** to an output stream.
operator<<(OutputStream&, ulong)	Puts a **ulong** to an output stream.
operator<<(OutputStream&, ushort)	Puts a **ushort** to an output stream.

4.8.11.2.1 operator<<<C>(OutputStream&, const C&) Function

Puts the values of a forward container containing integers to the given stream.

Syntax

```
public OutputStream& operator<<<C>(OutputStream& s, const C& c);
```

Constraint

where C is ForwardContainer and C.ValueType is int;

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
c	const C&	A forward container.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.2 operator<<(OutputStream&, bool) Function

Puts a Boolean value to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, bool b);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
b	bool	A Boolean value.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.3 operator<<(OutputStream&, byte) Function

Puts a **byte** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, byte b);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
b	byte	A **byte**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.4 operator<<(OutputStream&, char) Function

Puts a character to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, char c);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
c	char	A character.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.5 operator<<(OutputStream&, double) Function

Puts a **double** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, double d);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
d	double	A **double**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.6 operator<<(OutputStream&, float) Function

Puts a **float** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, float f);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
f	float	A **float**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.7 operator<<(OutputStream&, int) Function

Puts an **int** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, int i);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
i	int	An **int**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.8 operator<<(OutputStream&, long) Function

Puts a **long** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, long l);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
l	long	A **long**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.9 operator<<(OutputStream&, const string&) Function

Puts a String to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, const string& str);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
str	const string&	A String.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.10 operator<<(OutputStream&, const char*) Function

Puts a C-style string to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, const char* str);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
str	const char*	A C-style string.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.11 operator<<(OutputStream&, EndLine) Function

Puts an end-of-line character to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, EndLine __parameter1);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
__parameter1	EndLine	End-of-line.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.12 operator<<(OutputStream&, sbyte) Function

Puts an **sbyte** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, sbyte b);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
b	sbyte	An **sbyte**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.13 operator<<(OutputStream&, short) Function

Puts a **short** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, short x);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
x	short	A **short**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.14 operator<<(OutputStream&, uint) Function

Puts a **uint** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, uint u);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
u	uint	A **uint**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.15 operator<<(OutputStream&, ulong) Function

Puts a **ulong** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, ulong u);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
u	ulong	A **ulong**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.11.2.16 operator<<(OutputStream&, ushort) Function

Puts a **ushort** to an output stream.

Syntax

```
public OutputStream& operator<<(OutputStream& s, ushort u);
```

Parameters

Name	Type	Description
s	OutputStream&	An output stream.
u	ushort	A **ushort**.

Returns

OutputStream&

Returns a reference to the output stream.

4.8.12 OutputStringStream Class

A class for writing to a string.

Syntax

```
public class OutputStringStream;
```

Base Class

OutputStream

4.8.12.1 Member Functions

Member Function	Description
OutputStringStream()	Default constructor. Write to an empty string.
OutputStringStream(OutputStringStream&&)	Move constructor.
OutputStringStream(const string&)	Constructor. Write to the end of the given string.
operator=(OutputStringStream&&)	Move assignment.
∼OutputStringStream()	Destructor.
GetStr() const	Returns the contained string.
SetStr(const string&)	Sets the contained string.
Write(const char*)	Writes a C-style string to the end of the contained string.
Write(const string&)	Writes a string to the end of the contained string.
Write(bool)	Writes a Boolean value to the end of the contained string.
Write(byte)	Writes a **byte** to the end of the contained string.

Write(char)	Writes a character to the endf of the contained string.
Write(double)	Writes a **double** to the end of the contained string.
Write(float)	Writes a **float** to the end of the contained string.
Write(int)	Writes an **int** to the end of the contained string.
Write(long)	Writes a **long** to the end of the contained string.
Write(sbyte)	Writes an **sbyte** to the end of the contained string.
Write(short)	Writes a **short** to the end of the contained string.
Write(uint)	Writes a **uint** to the end of the contained string.
Write(ulong)	Writes a **ulong** to the end of the contained string.
Write(ushort)	Writes a **ushort** to the end of the contained string.
WriteLine()	Writes a new line to the end of the contained string.
WriteLine(const char*)	Writes a C-style string followed by a new line to the end of the contained string.
WriteLine(const string&)	Writes a string followed by a new line to the end of the contained string.
WriteLine(bool)	Writes a Boolean followed by a new line to the end of the contained string.

WriteLine(byte)	Writes a **byte** followed by a new line to the end of the contained string.
WriteLine(char)	Writes a character followed by a new line to the end of the contained string.
WriteLine(double)	Writes a **double** followed by a new line to the end of the contained string.
WriteLine(float)	Writes a **float** followed by a new line to the end of the contained string.
WriteLine(int)	Writes an **int** followed by a new line to the end of the contained string.
WriteLine(long)	Writes a **long** followed by a new line to the end of the contained string.
WriteLine(sbyte)	Writes an **sbyte** followed by a new line to the end of the contained string.
WriteLine(short)	Writes a **short** followed by a new line to the end of the contained string.
WriteLine(uint)	Writes a **uint** followed by a new line to the end of the contained string.
WriteLine(ulong)	Writes a **ulong** followed by a new line to the end of the contained string.
WriteLine(ushort)	Writes a **ushort** followed by a new line to the end of the contained string.

4.8.12.1.1 OutputStringStream() Member Function

Default constructor. Write to an empty string.

Syntax

```
public OutputStringStream();
```

4.8.12.1.2 OutputStringStream(OutputStringStream&&) Member Function

Move constructor.

Syntax

```
public nothrow OutputStringStream(OutputStringStream&& that);
```

Parameters

Name	Type	Description
that	OutputStringStream&&	An output string stream to move from.

4.8.12.1.3 OutputStringStream(const string&) Member Function

Constructor. Write to the end of the given string.

Syntax

```
public OutputStringStream(const string& str_);
```

Parameters

Name	Type	Description
str_	const string&	A string to write to.

4.8.12.1.4 operator=(OutputStringStream&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(OutputStringStream&& that);
```

Parameters

Name	Type	Description
that	OutputStringStream&&	An output string stream to move from.

4.8.12.1.5 ~OutputStringStream() Member Function

Destructor.

Syntax

```
public override nothrow ~OutputStringStream();
```

4.8.12.1.6 GetStr() const Member Function

Returns the contained string.

Syntax

```
public const string& GetStr() const;
```

Returns

const string&

Returns the contained string.

4.8.12.1.7 SetStr(const string&) Member Function

Sets the contained string.

Syntax

```
public void SetStr(const string& str_);
```

Parameters

Name	Type	Description
str_	const string&	A string to write to.

4.8.12.1.8 Write(const char*) Member Function

Writes a C-style string to the end of the contained string.

Syntax

```
public override void Write(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.12.1.9 Write(const string&) Member Function

Writes a string to the end of the contained string.

Syntax

```
public override void Write(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.12.1.10 Write(bool) Member Function

Writes a Boolean value to the end of the contained string.

Syntax

```
public override void Write(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean value to write.

4.8.12.1.11 Write(byte) Member Function

Writes a **byte** to the end of the contained string.

Syntax

```
public override void Write(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.12.1.12 Write(char) Member Function

Writes a character to the endf of the contained string.

Syntax

```
public override void Write(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.12.1.13 Write(double) Member Function

Writes a **double** to the end of the contained string.

Syntax

```
public override void Write(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.12.1.14 Write(float) Member Function

Writes a **float** to the end of the contained string.

Syntax

```
public override void Write(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.12.1.15 Write(int) Member Function

Writes an **int** to the end of the contained string.

Syntax

```
public override void Write(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.12.1.16 Write(long) Member Function

Writes a **long** to the end of the contained string.

Syntax

```
public override void Write(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.12.1.17 Write(sbyte) Member Function

Writes an **sbyte** to the end of the contained string.

Syntax

```
public override void Write(sbyte b);
```

Parameters

Name	Type	Description
b	sbyte	An **sbyte** to write.

4.8.12.1.18 Write(short) Member Function

Writes a **short** to the end of the contained string.

Syntax

```
public override void Write(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.12.1.19 Write(uint) Member Function

Writes a **uint** to the end of the contained string.

Syntax

```
public override void Write(uint i);
```

Parameters

Name	Type	Description
i	uint	A **uint** to write.

4.8.12.1.20 Write(ulong) Member Function

Writes a **ulong** to the end of the contained string.

Syntax

```
public override void Write(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.12.1.21 Write(ushort) Member Function

Writes a **ushort** to the end of the contained string.

Syntax

```
public override void Write(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.12.1.22 WriteLine() Member Function

Writes a new line to the end of the contained string.

Syntax

```
public override void WriteLine();
```

4.8.12.1.23 WriteLine(const char*) Member Function

Writes a C-style string followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(const char* s);
```

Parameters

Name	Type	Description
s	const char*	A C-style string to write.

4.8.12.1.24 WriteLine(const string&) Member Function

Writes a string followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to write.

4.8.12.1.25 WriteLine(bool) Member Function

Writes a Boolean followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(bool b);
```

Parameters

Name	Type	Description
b	bool	A Boolean to write.

4.8.12.1.26 WriteLine(byte) Member Function

Writes a **byte** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(byte b);
```

Parameters

Name	Type	Description
b	byte	A **byte** to write.

4.8.12.1.27 WriteLine(char) Member Function

Writes a character followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(char c);
```

Parameters

Name	Type	Description
c	char	A character to write.

4.8.12.1.28 WriteLine(double) Member Function

Writes a **double** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(double d);
```

Parameters

Name	Type	Description
d	double	A **double** to write.

4.8.12.1.29 WriteLine(float) Member Function

Writes a **float** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(float f);
```

Parameters

Name	Type	Description
f	float	A **float** to write.

4.8.12.1.30 WriteLine(int) Member Function

Writes an **int** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(int i);
```

Parameters

Name	Type	Description
i	int	An **int** to write.

4.8.12.1.31 WriteLine(long) Member Function

Writes a **long** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(long l);
```

Parameters

Name	Type	Description
l	long	A **long** to write.

4.8.12.1.32 WriteLine(sbyte) Member Function

Writes an **sbyte** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(sbyte b);
```

Parameters

Name	Type	Description
b	sbyte	An **sbyte** to write.

4.8.12.1.33 WriteLine(short) Member Function

Writes a **short** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(short s);
```

Parameters

Name	Type	Description
s	short	A **short** to write.

4.8.12.1.34 WriteLine(uint) Member Function

Writes a **uint** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(uint i);
```

Parameters

Name	Type	Description
i	uint	A **uint** to write.

4.8.12.1.35 WriteLine(ulong) Member Function

Writes a **ulong** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(ulong u);
```

Parameters

Name	Type	Description
u	ulong	A **ulong** to write.

4.8.12.1.36 WriteLine(ushort) Member Function

Writes a **ushort** followed by a new line to the end of the contained string.

Syntax

```
public override void WriteLine(ushort u);
```

Parameters

Name	Type	Description
u	ushort	A **ushort** to write.

4.8.13 Path Class

A static class for manipulating paths.

Syntax

```
public static class Path;
```

4.8.13.1 Member Functions

Member Function	Description
ChangeExtension(const string&, const string&)	Changes an extension of a path.
Combine(const string&, const string&)	Combines to paths.
GetDirectoryName(const string&)	Returns the directory name part of a path.
GetExtension(const string&)	Returns the extension of a path.
GetFileName(const string&)	Returns the file name part of a path.
GetFileNameWithoutExtension(const string&)	Returns the file name without an extension part of a path.
HasExtension(const string&)	Returns true, if the given path has an extension, false otherwise.
IsAbsolute(const string&)	Returns true if the given path is absolute, false otherwise.
IsRelative(const string&)	Returns true if the given path is relative, false otherwise.
MakeCanonical(const string&)	Returns a canonical representation of a path.

4.8.13.1.1 ChangeExtension(const string&, const string&) Member Function

Changes an extension of a path.

Syntax

```
public static nothrow string ChangeExtension(const string& path, const string&
extension);
```

Parameters

Name	Type	Description
path	const string&	A path.
extension	const string&	A new extension.

Returns

string

A path with a new extension.

Remarks

If new extension is empty, returns a path with extension removed. Otherwise, if path has no extension, returns a path with the new extension appended. Otherwise, returns a path with the new extension replaced. New extension can have '.' in it or not.

4.8.13.1.2 Combine(const string&, const string&) Member Function

Combines to paths.

Syntax

```
public static string Combine(const string& path1, const string& path2);
```

Parameters

Name	Type	Description
path1	const string&	The first path.
path2	const string&	The second path.

Returns

string

Returns the first path combined with the second path.

Remarks

If the first path is empty, returns the second path. Otherwise, if the second path is empty, returns the first path. Otherwise, if the second path is absolute, returns the second path. Otherwise returns the first path and the second path separated by the '/' character.

4.8.13.1.3 GetDirectoryName(const string&) Member Function

Returns the directory name part of a path.

Syntax

```
public static string GetDirectoryName(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

A path with last component removed.

Remarks

If the path parameter is empty, returns empty string. Otherwise, if the path consists of alphabetical letter, a colon and a slash, returns empty string. Otherwise, if the path has a '/' character, returns a path with the last '/' character and characters following it removed. Otherwise returns an empty string.

4.8.13.1.4 GetExtension(const string&) Member Function

Returns the extension of a path.

Syntax

```
public static string GetExtension(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

Returns the extension of the path.

Remarks

If the path contains a '.' character but it also contains '/' character after the last '.' character, an empty string is returned. Otherwise, if the path contains a '.' character, returns a substring containing the last '.' character and characters following it. Otherwise returns an empty string.

4.8.13.1.5 GetFileName(const string&) Member Function

Returns the file name part of a path.

Syntax

```
public static string GetFileName(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

Returns a path with extension removed.

Remarks

If the given path is empty, or the last character of it is '/' or ':', returns empty string. Otherwise, if the path contains a '/' character, returns the characters following the last '/' character. Otherwise returns the path.

4.8.13.1.6 GetFileNameWithoutExtension(const string&) Member Function

Returns the file name without an extension part of a path.

Syntax

```
public static string GetFileNameWithoutExtension(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

Returns a file name without an extension.

Remarks

First gets the file name part of the path. If the file name has an extension, returns the file name with extension removed. Otherwise returns the file name.

4.8.13.1.7 HasExtension(const string&) Member Function

Returns true, if the given path has an extension, false otherwise.

Syntax

```
public static bool HasExtension(const string& path);
```

Parameters

Name	Type	Description

path	const string&	A path.

Returns

bool

Returns true, if the given path has an extension, false otherwise.

Remarks

If the path has no '.' character or the path ends with a '.' character, returns false. Otherwise, if the path has a '/' or ':' character after the last '.' character, returns false. Otherwise returns true.

4.8.13.1.8 IsAbsolute(const string&) Member Function

Returns true if the given path is absolute, false otherwise.

Syntax

```
public static bool IsAbsolute(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

bool

Returns true if the given path is absolute, false otherwise.

Remarks

If the given path is empty, returns false. Otherwise, if the given path begins with the '/' character, returns true. Otherwise, if the given path begins with an alphabetical letter followed by a ':' character followed by a '/' character, returns true. Otherwise returns false.

4.8.13.1.9 IsRelative(const string&) Member Function

Returns true if the given path is relative, false otherwise.

Syntax

```
public static bool IsRelative(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

bool

Returns true if the given path is relative, false otherwise.

Remarks

Returns the complement of the value returned by the IsAbsolute(const string&) function for the given path.

4.8.13.1.10 MakeCanonical(const string&) Member Function

Returns a canonical representation of a path.

Syntax

```
public static string MakeCanonical(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

Returns a canonical representation of the given path.

Remarks

First replaces each '\' character of the path with the '/' character. Then, if the path consists of an alphabetical letter followed by the ':' character followed by the '/' character returns the path. Otherwise, if the path consists of a '/' character, returns the path. Otherwise, if the path ends with a '/' character, returns the path with last '/' character removed. Otherwise returns the path.

4.9 Functions

Function	Description
DirectoryExists(const string&)	Returns true if a directory with a given path name exists, false otherwise.
FileExists(const string&)	Returns true if a file with a given path name exists, false otherwise.
GetCurrentWorkingDirectory()	Returns a path to the current working directory.
GetFullPath(const string&)	Returns an absolute path corresponding to the given absolute or relative path.
PathExists(const string&)	Returns true if the given path exists, false otherwise.
ReadFile(const string&)	Reads a file with the given name into a string and returns it.
operator<<(OutputStream&, Date)	

4.9.14 DirectoryExists(const string&) Function

Returns true if a directory with a given path name exists, false otherwise.

Syntax

```
public nothrow bool DirectoryExists(const string& directoryPath);
```

Parameters

Name	Type	Description
directoryPath	const string&	A path to test.

Returns

bool

Returns true if a directory with a given path name exists, false otherwise.

4.9.15 FileExists(const string&) Function

Returns true if a file with a given path name exists, false otherwise.

Syntax

```
public nothrow bool FileExists(const string& filePath);
```

Parameters

Name	Type	Description
filePath	const string&	A path to test.

Returns

bool

Returns true if a file with a given path name exists, false otherwise.

4.9.16 GetCurrentWorkingDirectory() Function

Returns a path to the current working directory.

Syntax

```
public string GetCurrentWorkingDirectory();
```

Returns

string

Returns a path to the current working directory.

4.9.17 GetFullPath(const string&) Function

Returns an absolute path corresponding to the given absolute or relative path.

Syntax

```
public string GetFullPath(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

string

Returns an absolute path corresponding to the given absolute or relative path.

Remarks

If the given path is relative, prefixes it with the path to current working directory.

4.9.18 PathExists(const string&) Function

Returns true if the given path exists, false otherwise.

Syntax

```
public nothrow bool PathExists(const string& path);
```

Parameters

Name	Type	Description
path	const string&	A path.

Returns

bool

Returns true if the given path exists, false otherwise.

4.9.19 ReadFile(const string&) Function

Reads a file with the given name into a string and returns it.

Syntax

```
public string ReadFile(const string& fileName);
```

Parameters

Name	Type	Description
fileName	const string&	The name of the file to read.

Returns

string

Returns the contents of the given file.

4.9.20 operator<<(OutputStream&, Date) Function

Syntax

```
public OutputStream& operator<<(OutputStream& s, Date date);
```

4.10 Enumerations

Enumeration	Description
OpenMode	An open mode for a binary file stream.

4.10.20.1 OpenMode Enumeration

An open mode for a binary file stream.

Enumeration Constants

Constant	Value	Description
readOnly	0	Open the file with read only access.
writeOnly	1	Open the file with write only access. Truncates the file if it exists.
readWrite	2	Open the file with read and write access.

5 System.Text Namespace

Contains classes and functions for manipulating text.

5.11 Classes

Class	Description
CodeFormatter	A class for generating indented text.

5.11.1 CodeFormatter Class

A class for generating indented text.

Syntax

```
public class CodeFormatter;
```

5.11.1.1 Member Functions

Member Function	Description
CodeFormatter(OutputStream&)	Constructor. Initalizes the the code formatter with the given output stream.
CurrentIndent() const	Returns the current indent in characters.
DecIndent()	Decreases the indent.
IncIndent()	Increases the indent.
Indent() const	Returns the indent level.
IndentSize() const	Returns the number of characters to indent.
SetIndentSize(int)	Sets the number of characters to indent.
Write(const string&)	If at the beginning of a line, writes a string indented with the current indent. Otherwise writes the given string.
WriteLine()	Writes end-of-line character.
WriteLine(const string&)	Writes given string using Write(const string&) function and then writes end-of-line character.

5.11.1.1.1 CodeFormatter(OutputStream&) Member Function

Constructor. Initalizes the the code formatter with the given output stream.

Syntax

```
public CodeFormatter(OutputStream& stream_);
```

Parameters

Name	Type	Description
stream_	OutputStream&	An output stream.

5.11.1.1.2 CurrentIndent() const Member Function

Returns the current indent in characters.

Syntax

```
public inline nothrow int CurrentIndent() const;
```

Returns

int

Returns the current indent in characters.

5.11.1.1.3 DecIndent() Member Function

Decreases the indent.

Syntax

```
public inline nothrow void DecIndent();
```

5.11.1.1.4 IncIndent() Member Function

Increases the indent.

Syntax

```
public inline nothrow void IncIndent();
```

5.11.1.1.5 Indent() const Member Function

Returns the indent level.

Syntax

```
public inline nothrow int Indent() const;
```

Returns

int

Returns the indent level.

5.11.1.1.6 IndentSize() const Member Function

Returns the number of characters to indent.

Syntax

```
public inline nothrow int IndentSize() const;
```

Returns

int

Returns the number of characters to indent.

5.11.1.1.7 SetIndentSize(int) Member Function

Sets the number of characters to indent.

Syntax

```
public inline nothrow void SetIndentSize(int indentSize_);
```

Parameters

Name	Type	Description
indentSize_	int	The number of characters to indent.

5.11.1.1.8 Write(const string&) Member Function

If at the beginning of a line, writes a string indented with the current indent. Otherwise writes the given string.

Syntax

```
public void Write(const string& text);
```

Parameters

Name	Type	Description
text	const string&	A string to write.

5.11.1.1.9 WriteLine() Member Function

Writes end-of-line character.

Syntax

```
public void WriteLine();
```

5.11.1.1.10 WriteLine(const string&) Member Function

Writes given string using Write(const string&) function and then writes end-of-line character.

Syntax

```
public void WriteLine(const string& text);
```

Parameters

Name	Type	Description
text	const string&	A string to write

5.12 Functions

Function	Description
CharStr(char)	Returns a string representation of a character.
HexEscape(char)	Returns a hexadecimal representation of the given character.
MakeCharLiteral(char)	Returns a character literal representation of a character.
MakeStringLiteral(const string&)	Returns a string literal representation of a string.

5.12.2 CharStr(char) Function

Returns a string representation of a character.

Syntax

```
public string CharStr(char c);
```

Parameters

Name	Type	Description
c	char	A character to convert.

Returns

string

Returns the string representation of the given character.

Remarks

If the character is one of the C escape characters, returns a string containing the character prefixed with the backslash. Otherwise if the character is printable, returns a string containing the character. Otherwise returns a string containing the hexadecimal escape of the character.

5.12.3 HexEscape(char) Function

Returns a hexadecimal representation of the given character.

Syntax

```
public nothrow string HexEscape(char c);
```

Parameters

Name	Type	Description
c	char	A character.

Returns

string

Returns a hexadecimal representation of the given character.

5.12.4 MakeCharLiteral(char) Function

Returns a character literal representation of a character.

Syntax

```
public string MakeCharLiteral(char c);
```

Parameters

Name	Type	Description
c	char	A character to convert.

Returns

string

Returns a character literal representation of a character.

Remarks

Returns the character enclosed in apostrophes and escaped if necessary.

5.12.5 MakeStringLiteral(const string&) Function

Returns a string literal representation of a string.

Syntax

```
public string MakeStringLiteral(const string& s);
```

Parameters

Name	Type	Description
s	const string&	A string to convert.

Returns

string

Returns a string literal representation of a string.

Remarks

Returns the string enclosed in quotes and characters escaped if necessary.

6 System.Threading Namespace

Contains classes and functions for controlling multiple threads of execution.

6.13 Concepts

Concept	Description
Lockable<M>	Lockable class contains Lock() and Unlock() member functions.

6.13.1 Lockable<M> Concept

Lockable class contains Lock() and Unlock() member functions.

Syntax

```
public concept Lockable<M>;
```

Constraints

```
void M.Lock();
```

```
void M.Unlock();
```

Models

Mutex and RecursiveMutex are models of lockable.

6.14 Classes

Class	Description
ConditionVariable	Condition variables can be used as a communication mechanism among threads.
LockGuard<M>	Helper class for locking and unlocking lockable object (mutex or recursive mutex).
Mutex	Mutexes are a synchronization mechanism for controlling threads' access to some data.
RecursiveMutex	Mutexes are a synchronization mechanism for controlling threads' access to some data. A recursive mutex allows the calling thread lock the mutex many times recursively.
Thread	Represents thread of execution.
ThreadingException	Exception class thrown when some thread operation fails.

6.14.2 ConditionVariable Class

Condition variables can be used as a communication mechanism among threads.

Syntax

```
public class ConditionVariable;
```

6.14.2.1 Member Functions

Member Function	Description
ConditionVariable()	Constructor. Initializes the condition variable.
~ConditionVariable()	Destructor. Destroys the condition variable.
NotifyAll()	Unblock all threads waiting on this condition variable.
NotifyOne()	Unblock one thread waiting on this condition variable.
Wait(Mutex&)	Wait on a condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.
WaitFor(Mutex&, Duration)	Wait for specified duration that the condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.
WaitUntil(Mutex&, TimePoint)	Wait until specified time point that the condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.

6.14.2.1.1 ConditionVariable() Member Function

Constructor. Initializes the condition variable.

Syntax

```
public ConditionVariable();
```

6.14.2.1.2 ~ConditionVariable() Member Function

Destructor. Destroys the condition variable.

Syntax

```
public nothrow ~ConditionVariable();
```

6.14.2.1.3 NotifyAll() Member Function

Unblock all threads waiting on this condition variable.

Syntax

```
public void NotifyAll();
```

6.14.2.1.4 NotifyOne() Member Function

Unblock one thread waiting on this condition variable.

Syntax

```
public void NotifyOne();
```

6.14.2.1.5 Wait(Mutex&) Member Function

Wait on a condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.

Syntax

```
public void Wait(Mutex& m);
```

Parameters

Name	Type	Description
m	Mutex&	Mutex associated with this condition variable.

6.14.2.1.6 WaitFor(Mutex&, Duration) Member Function

Wait for specified duration that the condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.

Syntax

```
public bool WaitFor(Mutex& m, Duration d);
```

Parameters

Name	Type	Description
m	Mutex&	Mutex associated with this condition variable.
d	Duration	Duration to wait.

Returns

bool

Returns true, if specified duration has elapsed without the condition variable to become signaled, false otherwise.

6.14.2.1.7 WaitUntil(Mutex&, TimePoint) Member Function

Wait until specified time point that the condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.

Syntax

```
public bool WaitUntil(Mutex& m, TimePoint tp);
```

Parameters

Name	Type	Description
m	Mutex&	Mutex associated with this condition variable.
tp	TimePoint	Due time.

Returns

bool

Returns true, if specified time point has been reached without the condition variable to become signaled, false otherwise.

6.14.3 LockGuard<M> Class

Helper class for locking and unlocking lockable object (mutex or recursive mutex).

Syntax

```
public class LockGuard<M>;
```

Constraint

where M is Lockable;

6.14.3.1 Member Functions

Member Function	Description
LockGuard(M&)	Locks the lockable object.
~LockGuard()	Unlocks the lockable object.
GetLock()	Returns the lockable object.

6.14.3.1.1 LockGuard(M&) Member Function

Locks the lockable object.

Syntax

```
public LockGuard(M& m_);
```

Parameters

Name	Type	Description
m_	M&	A lockable object.

6.14.3.1.2 ~LockGuard() Member Function

Unlocks the lockable object.

Syntax

```
public nothrow ~LockGuard();
```

6.14.3.1.3 GetLock() Member Function

Returns the lockable object.

Syntax

```
public nothrow M& GetLock();
```

Returns

M&

Returns the lockable object.

6.14.4 Mutex Class

Mutexes are a synchronization mechanism for controlling threads' access to some data.

Syntax

```
public class Mutex;
```

6.14.4.1 Remarks

Basic mutex cannot be locked recursively many times by same thread.

6.14.4.2 Member Functions

Member Function	Description
Mutex()	Constructor. Initializes the mutex.
~Mutex()	Destructor. Destroys the mutex.
Handle() const	Returns pointer to the mutex handle.
Lock()	Locks the mutex.
TryLock()	If the mutex is currently unlocked, locks the mutex and returns true. Otherwise returns false.
Unlock()	Unlocks the mutex.

6.14.4.2.1 Mutex() Member Function

Constructor. Initializes the mutex.

Syntax

```
public Mutex();
```

6.14.4.2.2 ~Mutex() Member Function

Destructor. Destroys the mutex.

Syntax

```
public nothrow ~Mutex();
```

6.14.4.2.3 Handle() const Member Function

Returns pointer to the mutex handle.

Syntax

```
public void** Handle() const;
```

Returns

void**

Returns pointer to the mutex handle.

6.14.4.2.4 Lock() Member Function

Locks the mutex.

Syntax

```
public void Lock();
```

6.14.4.2.5 TryLock() Member Function

If the mutex is currently unlocked, locks the mutex and returns true. Otherwise returns false.

Syntax

```
public bool TryLock();
```

Returns

bool

Returns true, if the mutex is currently unlocked, false otherwise.

6.14.4.2.6 Unlock() Member Function

Unlocks the mutex.

Syntax

```
public void Unlock();
```

6.14.5 RecursiveMutex Class

Mutexes are a synchronization mechanism for controlling threads' access to some data. A recursive mutex allows the calling thread lock the mutex many times recursively.

Syntax

```
public class RecursiveMutex;
```

Base Class

Mutex

6.14.5.1 Member Functions

Member Function	Description
RecursiveMutex()	Constructor. Initializes the recursive mutex.
~RecursiveMutex()	Destructor. Destroys the recursive mutex.

6.14.5.1.1 RecursiveMutex() Member Function

Constructor. Initializes the recursive mutex.

Syntax

```
public RecursiveMutex();
```

6.14.5.1.2 ~RecursiveMutex() Member Function

Destructor. Destroys the recursive mutex.

Syntax

```
public nothrow ~RecursiveMutex();
```

6.14.6 Thread Class

Represents thread of execution.

Syntax

```
public class Thread;
```

6.14.6.1 Member Functions

Member Function	Description
Thread()	Default constructor. Initializes a thread that is not associated with operating system thread.
Thread(Thread&&)	Move constructor.
Thread(ThreadFun, void*)	Constructor. Associates the thread with an operating system thread and starts executing the specified start function asynchronously.
operator=(Thread&&)	Move assignment.
∼Thread()	If the thread is associated with an operating system thread and it is not joined or detached, calls exit with EXIT_THREADS_NOT_JOINED exit status.
Detach()	Detaches the thread. Detached thread cannot be joined.
Handle() const	Returns thread handle.
Join()	Waits for the thread to terminate.
Joinable() const	Returns true, if the thread can be joined, false otherwise.

6.14.6.1.1 Thread() Member Function

Default constructor. Initializes a thread that is not associated with operating system thread.

Syntax

```
public Thread();
```

6.14.6.1.2 Thread(Thread&&) Member Function

Move constructor.

Syntax

```
public Thread(Thread&& that);
```

Parameters

Name	Type	Description
that	Thread&&	A thread to move from.

6.14.6.1.3 Thread(ThreadFun, void*) Member Function

Constructor. Associates the thread with an operating system thread and starts executing the specified start function asynchronously.

Syntax

```
public Thread(ThreadFun start, void* arg);
```

Parameters

Name	Type	Description
start	ThreadFun	A thread start function.
arg	void*	Argument to the thread start function.

6.14.6.1.4 operator=(Thread&&) Member Function

Move assignment.

Syntax

```
public void operator=(Thread&& that);
```

Parameters

Name	Type	Description
that	Thread&&	A thread to assign from.

6.14.6.1.5 ~Thread() Member Function

If the thread is associated with an operating system thread and it is not joined or detached, calls exit with EXIT_THREADS_NOT_JOINED exit status.

Syntax

```
public nothrow ~Thread();
```

6.14.6.1.6 Detach() Member Function

Detaches the thread. Detached thread cannot be joined.

Syntax

```
public void Detach();
```

6.14.6.1.7 Handle() const Member Function

Returns thread handle.

Syntax

```
public ulong Handle() const;
```

Returns

ulong

Returns thread handle.

6.14.6.1.8 Join() Member Function

Waits for the thread to terminate.

Syntax

```
public void Join();
```

6.14.6.1.9 Joinable() const Member Function

Returns true, if the thread can be joined, false otherwise.

Syntax

```
public bool Joinable() const;
```

Returns

bool

Returns true, if the thread can be joined, false otherwise.

6.14.7 ThreadingException Class

Exception class thrown when some thread operation fails.

Syntax

```
public class ThreadingException;
```

Base Class

Exception

6.14.7.1 Member Functions

Member Function	Description
ThreadingException(const ThreadingException&)	Copy constructor.
ThreadingException(ThreadingException&&)	Move constructor.
ThreadingException(const string&, const string&)	Constructor. Initializes the threading exception with the specified operation description and failure reason.
operator=(const ThreadingException&)	Copy assignment.
operator=(ThreadingException&&)	Move assignment.
~ThreadingException()	Destructor.

6.14.7.1.1 ThreadingException(const ThreadingException&) Member Function

Copy constructor.

Syntax

```
public nothrow ThreadingException(const ThreadingException& that);
```

Parameters

Name	Type	Description
that	const ThreadingException&	A threading exception to copy.

6.14.7.1.2 ThreadingException(ThreadingException&&) Member Function

Move constructor.

Syntax

```
public nothrow ThreadingException(ThreadingException&& that);
```

Parameters

Name	Type	Description
that	ThreadingException&&	A threading exception to move from.

6.14.7.1.3 ThreadingException(const string&, const string&) Member Function

Constructor. Initializes the threading exception with the specified operation description and failure reason.

Syntax

```
public ThreadingException(const string& operation, const string& reason);
```

Parameters

Name	Type	Description
operation	const string&	Description of the thread operation.
reason	const string&	Reason for failure.

6.14.7.1.4 operator=(const ThreadingException&) Member Function

Copy assignment.

Syntax

```
public nothrow void operator=(const ThreadingException& that);
```

Parameters

Name	Type	Description
that	const ThreadingException&	A threading exception to assign from.

6.14.7.1.5 operator=(ThreadingException&&) Member Function

Move assignment.

Syntax

```
public nothrow void operator=(ThreadingException&& that);
```

Parameters

Name	Type	Description
that	ThreadingException&&	A threading exception to move from.

6.14.7.1.6 ~ThreadingException() Member Function

Destructor.

Syntax

```
public override nothrow ~ThreadingException();
```

6.15 Functions

Function	Description
operator==(const Thread&, const Thread&)	Returns true, if the specified threads are associated with the same operation system thread, false otherwise.
SleepFor(Duration)	Puts the calling thread to sleep for specified duration.
SleepUntil(TimePoint)	Puts the calling thread to sleep until specified time point has been reached.
ThreadStart(void*)	Implementation detail.

6.15.8 operator==(const Thread&, const Thread&) Function

Returns true, if the specified threads are associated with the same operation system thread, false otherwise.

Syntax

```
public bool operator==(const Thread& t1, const Thread& t2);
```

Parameters

Name	Type	Description
t1	const Thread&	The first thread.
t2	const Thread&	The second thread.

Returns

bool

Returns true, if the specified threads are associated with the same operation system thread, false otherwise.

6.15.9 SleepFor(Duration) Function

Puts the calling thread to sleep for specified duration.

Syntax

```
public void SleepFor(Duration d);
```

Parameters

Name	Type	Description
d	Duration	Duration to sleep.

6.15.10 SleepUntil(TimePoint) Function

Puts the calling thread to sleep until specified time point has been reached.

Syntax

```
public void SleepUntil(TimePoint tp);
```

Parameters

Name	Type	Description
tp	TimePoint	Time point to sleep until.

6.15.11 ThreadStart(void*) Function

Implementation detail.

Syntax

```
public nothrow void ThreadStart(void* arg);
```

Parameters

Name	Type	Description
arg	void*	

6.16 Delegates

Delegate	Description
ThreadFun	A thread start function delegate.

6.16.12 ThreadFun Delegate

A thread start function delegate.

Syntax

`ThreadFun`

Parameters

Name	Type	Description
arg	void*	Argument to thread start function.

6.17 Constants

Constant	Type	Value	Description
EXIT_THREADS_NOT_JOINED	int	250	Program exit status when all threads are not joined or detached.