

# SYSTEM LIBRARY REFERENCE

January 5, 2016

# Contents

<b>Description</b>	<b>xiv</b>
<b>Copyrights</b>	<b>xv</b>
<b>Namespaces</b>	<b>xviii</b>
<b>1 Global Namespace</b>	<b>1</b>
1.1 Concepts . . . . .	2
1.1.1 Common<T, U> Concept . . . . .	3
1.1.2 Convertible<T, U> Concept . . . . .	4
1.1.3 Derived<T, U> Concept . . . . .	5
1.1.4 ExplicitlyConvertible<T, U> Concept . . . . .	6
1.1.5 NonReferenceType<T> Concept . . . . .	7
1.1.6 Same<T, U> Concept . . . . .	8
<b>2 System Namespace</b>	<b>9</b>
2.2 Classes . . . . .	12
2.2.1 BackInsertIterator<C> Class . . . . .	15
2.2.1.1 Remarks . . . . .	15
2.2.1.2 Example . . . . .	15
2.2.1.3 Type Definitions . . . . .	16
2.2.1.4 Member Functions . . . . .	16
2.2.2 BackInsertProxy<C> Class . . . . .	26
2.2.2.1 Member Functions . . . . .	26
2.2.3 BinaryFun<Argument1, Argument2, Result> Class . . . . .	29
2.2.3.1 Remarks . . . . .	29
2.2.3.2 Type Definitions . . . . .	30
2.2.3.3 Member Functions . . . . .	30
2.2.4 BinaryPred<Argument1, Argument2> Class . . . . .	36
2.2.4.1 Remarks . . . . .	36
2.2.4.2 Member Functions . . . . .	36
2.2.5 Console Class . . . . .	42
2.2.5.1 Remarks . . . . .	42
2.2.5.2 Example . . . . .	42
2.2.5.3 Member Functions . . . . .	42
2.2.6 ConversionException Class . . . . .	82
2.2.6.1 Remarks . . . . .	82
2.2.6.2 Example . . . . .	82
2.2.6.3 Member Functions . . . . .	82
2.2.7 Counter<T> Class . . . . .	90
2.2.7.1 Remarks . . . . .	90
2.2.7.2 Member Functions . . . . .	90

2.2.8	CounterBase Class . . . . .	94
2.2.8.1	Member Functions . . . . .	94
2.2.9	Date Class . . . . .	104
2.2.9.1	Remarks . . . . .	104
2.2.9.2	Member Functions . . . . .	104
2.2.9.3	Nonmember Functions . . . . .	113
2.2.10	Divides<T> Class . . . . .	116
2.2.10.1	Example . . . . .	116
2.2.10.2	Member Functions . . . . .	116
2.2.11	Duration Class . . . . .	124
2.2.11.1	Member Functions . . . . .	124
2.2.11.2	Nonmember Functions . . . . .	144
2.2.12	EndLine Class . . . . .	153
2.2.12.1	Remarks . . . . .	153
2.2.12.2	Member Functions . . . . .	153
2.2.13	EqualTo2<T, U> Class . . . . .	157
2.2.13.1	Remarks . . . . .	157
2.2.13.2	Member Functions . . . . .	157
2.2.14	EqualTo<T> Class . . . . .	164
2.2.14.1	Member Functions . . . . .	164
2.2.15	Exception Class . . . . .	171
2.2.15.1	Remarks . . . . .	171
2.2.15.2	Example . . . . .	171
2.2.15.3	Member Functions . . . . .	171
2.2.16	FrontInsertIterator<C> Class . . . . .	189
2.2.16.1	Remarks . . . . .	189
2.2.16.2	Example . . . . .	189
2.2.16.3	Type Definitions . . . . .	190
2.2.16.4	Member Functions . . . . .	190
2.2.17	FrontInsertProxy<C> Class . . . . .	200
2.2.17.1	Member Functions . . . . .	200
2.2.18	Greater2<T, U> Class . . . . .	203
2.2.18.1	Remarks . . . . .	203
2.2.18.2	Member Functions . . . . .	203
2.2.19	Greater<T> Class . . . . .	210
2.2.19.1	Example . . . . .	210
2.2.19.2	Example . . . . .	210
2.2.19.3	Member Functions . . . . .	211
2.2.20	GreaterOrEqualTo2<T, U> Class . . . . .	219
2.2.20.1	Remarks . . . . .	219
2.2.20.2	Member Functions . . . . .	219
2.2.21	GreaterOrEqualTo<T> Class . . . . .	226
2.2.21.1	Member Functions . . . . .	226
2.2.22	Identity<T> Class . . . . .	233
2.2.22.1	Member Functions . . . . .	233
2.2.23	InsertIterator<C> Class . . . . .	240
2.2.23.1	Remarks . . . . .	240
2.2.23.2	Example . . . . .	240
2.2.23.3	Type Definitions . . . . .	241
2.2.23.4	Member Functions . . . . .	241
2.2.24	InsertProxy<C> Class . . . . .	250
2.2.24.1	Member Functions . . . . .	250
2.2.25	Less2<T, U> Class . . . . .	253
2.2.25.1	Remarks . . . . .	253

2.2.25.2 Member Functions . . . . .	253
2.2.26 Less<T> Class . . . . .	260
2.2.26.1 Example . . . . .	260
2.2.26.2 Example . . . . .	260
2.2.26.3 Member Functions . . . . .	261
2.2.27 LessOrEqualTo2<T, U> Class . . . . .	269
2.2.27.1 Remarks . . . . .	269
2.2.27.2 Member Functions . . . . .	269
2.2.28 LessOrEqualTo<T> Class . . . . .	276
2.2.28.1 Member Functions . . . . .	276
2.2.29 MT Class . . . . .	283
2.2.29.1 Remarks . . . . .	283
2.2.29.2 Member Functions . . . . .	283
2.2.30 Minus<T> Class . . . . .	287
2.2.30.1 Member Functions . . . . .	287
2.2.31 Multiplies<T> Class . . . . .	294
2.2.31.1 Example . . . . .	294
2.2.31.2 Member Functions . . . . .	294
2.2.32 Negate<T> Class . . . . .	302
2.2.32.1 Member Functions . . . . .	302
2.2.33 NotEqualTo2<T, U> Class . . . . .	309
2.2.33.1 Remarks . . . . .	309
2.2.33.2 Member Functions . . . . .	309
2.2.34 NotEqualTo<T> Class . . . . .	316
2.2.34.1 Member Functions . . . . .	316
2.2.35 Pair<T, U> Class . . . . .	323
2.2.35.1 Remarks . . . . .	323
2.2.35.2 Member Functions . . . . .	323
2.2.35.3 Nonmember Functions . . . . .	329
2.2.36 Plus<T> Class . . . . .	332
2.2.36.1 Example . . . . .	332
2.2.36.2 Member Functions . . . . .	332
2.2.37 RandomAccessIter<T, R, P> Class . . . . .	340
2.2.37.1 Remarks . . . . .	340
2.2.37.2 Type Definitions . . . . .	341
2.2.37.3 Member Functions . . . . .	341
2.2.37.4 Nonmember Functions . . . . .	353
2.2.38 Rel<Argument> Class . . . . .	360
2.2.38.1 Remarks . . . . .	360
2.2.38.2 Type Definitions . . . . .	361
2.2.38.3 Member Functions . . . . .	361
2.2.39 Remainder<T> Class . . . . .	367
2.2.39.1 Member Functions . . . . .	367
2.2.40 SelectFirst<T, U> Class . . . . .	374
2.2.40.1 Member Functions . . . . .	374
2.2.41 SelectSecond<T, U> Class . . . . .	381
2.2.41.1 Member Functions . . . . .	381
2.2.42 ShareableFromThis<T> Class . . . . .	388
2.2.42.1 Remarks . . . . .	388
2.2.42.2 Example . . . . .	388
2.2.42.3 Member Functions . . . . .	388
2.2.43 SharedCount<T> Class . . . . .	397
2.2.43.1 Member Functions . . . . .	397
2.2.43.2 Nonmember Functions . . . . .	410

2.2.44	SharedPtr<T> Class . . . . .	413
2.2.44.1	Example . . . . .	413
2.2.44.2	Member Functions . . . . .	414
2.2.44.3	Nonmember Functions . . . . .	433
2.2.45	String Class . . . . .	436
2.2.45.1	Type Definitions . . . . .	437
2.2.45.2	Member Functions . . . . .	437
2.2.45.3	Example . . . . .	443
2.2.45.4	Nonmember Functions . . . . .	484
2.2.46	TimeError Class . . . . .	488
2.2.46.1	Member Functions . . . . .	488
2.2.47	TimePoint Class . . . . .	496
2.2.47.1	Remarks . . . . .	496
2.2.47.2	Member Functions . . . . .	496
2.2.47.3	Nonmember Functions . . . . .	503
2.2.48	TracedFun Class . . . . .	509
2.2.48.1	Member Functions . . . . .	509
2.2.49	Tracer Class . . . . .	513
2.2.49.1	Member Functions . . . . .	513
2.2.50	UnaryFun<Argument, Result> Class . . . . .	517
2.2.50.1	Remarks . . . . .	517
2.2.50.2	Type Definitions . . . . .	518
2.2.50.3	Member Functions . . . . .	518
2.2.51	UnaryPred<Argument> Class . . . . .	524
2.2.51.1	Remarks . . . . .	524
2.2.51.2	Member Functions . . . . .	524
2.2.52	UniquePtr<T> Class . . . . .	530
2.2.52.1	Remarks . . . . .	530
2.2.52.2	Example . . . . .	530
2.2.52.3	Member Functions . . . . .	530
2.2.52.4	Nonmember Functions . . . . .	545
2.2.53	WeakCount<T> Class . . . . .	548
2.2.53.1	Member Functions . . . . .	548
2.2.53.2	Nonmember Functions . . . . .	559
2.2.54	WeakPtr<T> Class . . . . .	562
2.2.54.1	Remarks . . . . .	562
2.2.54.2	Example . . . . .	562
2.2.54.3	Member Functions . . . . .	564
2.2.55	uhuge Class . . . . .	582
2.2.55.1	Member Functions . . . . .	582
2.2.55.2	Nonmember Functions . . . . .	591
2.3	Type Definitions . . . . .	609
2.4	Functions . . . . .	610
2.4.56	Abs(const T&) Function . . . . .	619
2.4.56.1	Example . . . . .	619
2.4.57	Accumulate(I, I, T, Op) Function . . . . .	620
2.4.57.1	Example . . . . .	620
2.4.58	BackInserter(C&) Function . . . . .	622
2.4.58.1	Example . . . . .	622
2.4.59	Copy(I, I, O) Function . . . . .	623
2.4.59.1	Example . . . . .	623
2.4.60	CopyBackward(I, I, O) Function . . . . .	625
2.4.60.1	Example . . . . .	625
2.4.61	Count(I, I, P) Function . . . . .	627

2.4.61.1 Example . . . . .	627
2.4.62 Count(I, I, const T&) Function . . . . .	629
2.4.62.1 Example . . . . .	629
2.4.63 CurrentDate() Function . . . . .	631
2.4.64 Distance(I, I) Function . . . . .	632
2.4.64.1 Example . . . . .	632
2.4.65 Distance(I, I) Function . . . . .	633
2.4.66 EnableSharedFromThis(System.ShareableFromThis<T>*, U*, const System.SharedCount<U>& Function . . . . .	634
2.4.67 EnableSharedFromThis(void*, void*, const System.SharedCount<T>&) Function . . . . .	635
2.4.68 Equal(I1, I1, I2, I2) Function . . . . .	636
2.4.68.1 Example . . . . .	636
2.4.69 Equal(I1, I1, I2, I2, R) Function . . . . .	638
2.4.69.1 Example . . . . .	638
2.4.70 EqualRange(I, I, const T&) Function . . . . .	640
2.4.70.1 Example . . . . .	640
2.4.70.2 Example . . . . .	641
2.4.71 EqualRange(I, I, const T&, R) Function . . . . .	642
2.4.71.1 Example . . . . .	642
2.4.72 Factorial(U) Function . . . . .	644
2.4.72.1 Example . . . . .	644
2.4.73 Find(I, I, P) Function . . . . .	645
2.4.73.1 Example . . . . .	645
2.4.74 Find(I, I, const T&) Function . . . . .	647
2.4.74.1 Example . . . . .	647
2.4.75 ForEach(I, I, F) Function . . . . .	649
2.4.75.1 Example . . . . .	649
2.4.76 FrontInserter(C&) Function . . . . .	651
2.4.76.1 Example . . . . .	651
2.4.77 Gcd(T, T) Function . . . . .	652
2.4.77.1 Example . . . . .	652
2.4.78 HexChar(byte) Function . . . . .	653
2.4.79 IdentityElement(System.Multiplies<T>) Function . . . . .	654
2.4.80 IdentityElement(System.Plus<T>) Function . . . . .	655
2.4.81 Inserter(C&, I) Function . . . . .	656
2.4.81.1 Example . . . . .	656
2.4.82 InsertionSort(I, I) Function . . . . .	657
2.4.82.1 Example . . . . .	657
2.4.83 InsertionSort(I, I, R) Function . . . . .	658
2.4.83.1 Example . . . . .	658
2.4.84 IsAlpha(char) Function . . . . .	659
2.4.85 IsAlphanumeric(char) Function . . . . .	660
2.4.86 IsControl(char) Function . . . . .	661
2.4.87 IsDigit(char) Function . . . . .	662
2.4.88 IsGraphic(char) Function . . . . .	663
2.4.89 IsHexDigit(char) Function . . . . .	664
2.4.90 IsLower(char) Function . . . . .	665
2.4.91 IsPrintable(char) Function . . . . .	666
2.4.92 IsPunctuation(char) Function . . . . .	667
2.4.93 IsSpace(char) Function . . . . .	668
2.4.94 IsUpper(char) Function . . . . .	669
2.4.95 LastComponentsEqual(const System.String&, const System.String&, char Function . . . . .	670

2.4.96 LexicographicalCompare(I1, I1, I2, I2) Function . . . . .	671
2.4.96.1 Example . . . . .	671
2.4.97 LexicographicalCompare(I1, I1, I2, I2, R) Function . . . . .	673
2.4.97.1 Example . . . . .	673
2.4.98 LowerBound(I, I, const T&) Function . . . . .	675
2.4.98.1 Example . . . . .	675
2.4.99 LowerBound(I, I, const T&, R) Function . . . . .	677
2.4.100 MakePair(const T&, const U&) Function . . . . .	678
2.4.101 Max(const T&, const T&) Function . . . . .	679
2.4.102 MaxElement(I, I) Function . . . . .	680
2.4.102.1 Example . . . . .	680
2.4.103 MaxElement(I, I, R) Function . . . . .	682
2.4.104 MaxValue() Function . . . . .	683
2.4.105 MaxValue(byte) Function . . . . .	684
2.4.106 MaxValue(int) Function . . . . .	685
2.4.107 MaxValue(long) Function . . . . .	686
2.4.108 MaxValue(sbyte) Function . . . . .	687
2.4.109 MaxValue(short) Function . . . . .	688
2.4.110 MaxValue(uint) Function . . . . .	689
2.4.111 MaxValue(ulong) Function . . . . .	690
2.4.112 MaxValue(ushort) Function . . . . .	691
2.4.113 Median(const T&, const T&, const T&) Function . . . . .	692
2.4.114 Median(const T&, const T&, const T&, R) Function . . . . .	693
2.4.115 Min(const T&, const T&) Function . . . . .	694
2.4.116 MinElement(I, I) Function . . . . .	695
2.4.116.1 Example . . . . .	695
2.4.117 MinElement(I, I, R) Function . . . . .	697
2.4.118 MinValue() Function . . . . .	698
2.4.119 MinValue(byte) Function . . . . .	699
2.4.120 MinValue(int) Function . . . . .	700
2.4.121 MinValue(long) Function . . . . .	701
2.4.122 MinValue(sbyte) Function . . . . .	702
2.4.123 MinValue(short) Function . . . . .	703
2.4.124 MinValue(uint) Function . . . . .	704
2.4.125 MinValue(ulong) Function . . . . .	705
2.4.126 MinValue(ushort) Function . . . . .	706
2.4.127 Move(I, I, O) Function . . . . .	707
2.4.128 MoveBackward(I, I, O) Function . . . . .	708
2.4.129 Next(I, int) Function . . . . .	709
2.4.130 Next(I, int) Function . . . . .	710
2.4.131 NextPermutation(I, I) Function . . . . .	711
2.4.131.1 Example . . . . .	711
2.4.132 NextPermutation(I, I, R) Function . . . . .	713
2.4.133 Now() Function . . . . .	714
2.4.134 ParseBool(const System.String&) Function . . . . .	715
2.4.135 ParseBool(const System.String&, bool&) Function . . . . .	716
2.4.136 ParseDate(const System.String&) Function . . . . .	717
2.4.137 ParseDouble(const System.String&) Function . . . . .	718
2.4.138 ParseDouble(const System.String&, double&) Function . . . . .	719
2.4.139 ParseHex(const System.String&) Function . . . . .	720
2.4.140 ParseHex(const System.String&, System.uhuge&) Function . . . . .	721
2.4.141 ParseHex(const System.String&, ulong&) Function . . . . .	722
2.4.142 ParseHexUHuge(const System.String&) Function . . . . .	723
2.4.143 ParseInt(const System.String&) Function . . . . .	724

2.4.144 ParseInt(const System.String&, int&) Function . . . . .	725
2.4.145 ParseUHuge(const System.String&) Function . . . . .	726
2.4.146 ParseUHuge(const System.String&, System.uhuge&) Function . . . . .	727
2.4.147 ParseUInt(const System.String&) Function . . . . .	728
2.4.148 ParseUInt(const System.String&, uint&) Function . . . . .	729
2.4.149 ParseULong(const System.String&) Function . . . . .	730
2.4.150 ParseULong(const System.String&, ulong&) Function . . . . .	731
2.4.151 PrevPermutation(I, I) Function . . . . .	732
2.4.152 PrevPermutation(I, I, R) Function . . . . .	733
2.4.153 PtrCast(const System.SharedPtr<T>&) Function . . . . .	734
2.4.153.1 Example . . . . .	734
2.4.154 Rand() Function . . . . .	735
2.4.155 RandomNumber(uint) Function . . . . .	736
2.4.156 RandomShuffle(I, I) Function . . . . .	737
2.4.157 Reverse(I, I) Function . . . . .	738
2.4.158 Reverse(I, I) Function . . . . .	739
2.4.158.1 Example . . . . .	739
2.4.159 ReverseUntil(I, I, I) Function . . . . .	740
2.4.160 Rotate(I, I, I) Function . . . . .	741
2.4.161 Rvalue(T&&) Function . . . . .	742
2.4.161.1 Example . . . . .	742
2.4.162 Select_0_2(const T&, const T&, R) Function . . . . .	744
2.4.163 Select_0_3(const T&, const T&, const T&, R) Function . . . . .	745
2.4.164 Select_1_2(const T&, const T&, R) Function . . . . .	746
2.4.165 Select_1_3(const T&, const T&, const T&, R) Function . . . . .	747
2.4.166 Select_1_3_ab(const T&, const T&, const T&, R) Function . . . . .	748
2.4.167 Select_2_3(const T&, const T&, const T&, R) Function . . . . .	749
2.4.168 Sort(C&) Function . . . . .	750
2.4.169 Sort(C&) Function . . . . .	751
2.4.169.1 Example . . . . .	751
2.4.170 Sort(C&, R) Function . . . . .	752
2.4.170.1 Example . . . . .	752
2.4.171 Sort(C&, R) Function . . . . .	753
2.4.172 Sort(I, I) Function . . . . .	754
2.4.173 Sort(I, I, R) Function . . . . .	755
2.4.174 Swap(T&, T&) Function . . . . .	756
2.4.175 ToHexString(System.uhuge) Function . . . . .	757
2.4.176 ToHexString(U) Function . . . . .	758
2.4.177 ToHexString(byte) Function . . . . .	759
2.4.178 ToHexString(uint) Function . . . . .	760
2.4.179 ToHexString(ulong) Function . . . . .	761
2.4.180 ToHexString(ushort) Function . . . . .	762
2.4.181 ToLower(const System.String&) Function . . . . .	763
2.4.182 ToString(I) Function . . . . .	764
2.4.183 ToString(System.Date) Function . . . . .	765
2.4.184 ToString(System.uhuge) Function . . . . .	766
2.4.185 ToString(U) Function . . . . .	767
2.4.186 ToString(bool) Function . . . . .	768
2.4.187 ToString(byte) Function . . . . .	769
2.4.188 ToString(char) Function . . . . .	770
2.4.189 ToString(double) Function . . . . .	771
2.4.190 ToString(double, int) Function . . . . .	772
2.4.191 ToString(int) Function . . . . .	773
2.4.192 ToString(long) Function . . . . .	774

2.4.193 <code>ToString(sbyte)</code> Function . . . . .	775
2.4.194 <code>ToString(short)</code> Function . . . . .	776
2.4.195 <code>ToString(uint)</code> Function . . . . .	777
2.4.196 <code>ToString(ulong)</code> Function . . . . .	778
2.4.197 <code>ToString(ushort)</code> Function . . . . .	779
2.4.198 <code>ToUpper(const System.String&amp;)</code> Function . . . . .	780
2.4.199 <code>ToUtf8(uint)</code> Function . . . . .	781
2.4.200 <code>Transform(I, I, O, F)</code> Function . . . . .	782
2.4.200.1 Example . . . . .	782
2.4.201 <code>Transform(I1, I1, I2, O, F)</code> Function . . . . .	784
2.4.201.1 Example . . . . .	784
2.4.202 <code>UpperBound(I, I, const T&amp;)</code> Function . . . . .	786
2.4.202.1 Example . . . . .	786
2.4.203 <code>UpperBound(I, I, const T&amp;, R)</code> Function . . . . .	788
2.4.204 <code>endl()</code> Function . . . . .	789
2.5 Enumerations . . . . .	790
2.5.204.1 <code>CharClass</code> Enumeration . . . . .	791
2.6 Constants . . . . .	792
<b>3 System.Collections Namespace</b> . . . . .	<b>793</b>
3.7 Classes . . . . .	794
3.7.1 <code>BitSet</code> Class . . . . .	796
3.7.1.1 Example . . . . .	796
3.7.1.2 Member Functions . . . . .	797
3.7.2 <code>Bucket&lt;T&gt;</code> Class . . . . .	824
3.7.2.1 Type Definitions . . . . .	825
3.7.2.2 Member Functions . . . . .	825
3.7.3 <code>ForwardList&lt;T&gt;</code> Class . . . . .	835
3.7.3.1 Remarks . . . . .	835
3.7.3.2 Type Definitions . . . . .	836
3.7.3.3 Member Functions . . . . .	836
3.7.3.4 Nonmember Functions . . . . .	858
3.7.4 <code>ForwardListNode&lt;T&gt;</code> Class . . . . .	861
3.7.4.1 Member Functions . . . . .	861
3.7.5 <code>ForwardListNodeIterator&lt;T, R, P&gt;</code> Class . . . . .	872
3.7.5.1 Type Definitions . . . . .	873
3.7.5.2 Member Functions . . . . .	873
3.7.5.3 Nonmember Functions . . . . .	883
3.7.6 <code>HashMap&lt;K, T, H, C&gt;</code> Class . . . . .	885
3.7.6.1 Type Definitions . . . . .	886
3.7.6.2 Member Functions . . . . .	886
3.7.6.3 Nonmember Functions . . . . .	909
3.7.7 <code>HashSet&lt;T, H, C&gt;</code> Class . . . . .	911
3.7.7.1 Type Definitions . . . . .	912
3.7.7.2 Member Functions . . . . .	912
3.7.7.3 Nonmember Functions . . . . .	933
3.7.8 <code>Hasher&lt;T&gt;</code> Class . . . . .	935
3.7.8.1 Member Functions . . . . .	935
3.7.9 <code>Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;</code> Class . . . . .	942
3.7.9.1 Type Definitions . . . . .	943
3.7.9.2 Member Functions . . . . .	943
3.7.10 <code>HashtableBase&lt;T&gt;</code> Class . . . . .	970
3.7.10.1 Type Definitions . . . . .	971
3.7.10.2 Member Functions . . . . .	971

3.7.11	HashtableIterator<T, R, P> Class . . . . .	972
3.7.11.1	Type Definitions . . . . .	973
3.7.11.2	Member Functions . . . . .	973
3.7.11.3	Nonmember Functions . . . . .	982
3.7.12	LinkedList<T> Class . . . . .	984
3.7.12.1	Type Definitions . . . . .	985
3.7.12.2	Member Functions . . . . .	985
3.7.12.3	Nonmember Functions . . . . .	1010
3.7.13	LinkedListBase Class . . . . .	1013
3.7.13.1	Member Functions . . . . .	1013
3.7.14	LinkedListNode<T> Class . . . . .	1014
3.7.14.1	Type Definitions . . . . .	1015
3.7.14.2	Member Functions . . . . .	1015
3.7.15	LinkedListNodeBase Class . . . . .	1024
3.7.15.1	Member Functions . . . . .	1024
3.7.16	LinkedListNodeIterator<T, R, P> Class . . . . .	1026
3.7.16.1	Type Definitions . . . . .	1027
3.7.16.2	Member Functions . . . . .	1027
3.7.16.3	Nonmember Functions . . . . .	1038
3.7.17	List<T> Class . . . . .	1040
3.7.17.1	Example . . . . .	1040
3.7.17.2	Type Definitions . . . . .	1041
3.7.17.3	Member Functions . . . . .	1041
3.7.17.4	Nonmember Functions . . . . .	1077
3.7.18	Map<Key, Value, KeyCompare> Class . . . . .	1080
3.7.18.1	Example . . . . .	1080
3.7.18.2	Type Definitions . . . . .	1082
3.7.18.3	Member Functions . . . . .	1082
3.7.18.4	Nonmember Functions . . . . .	1104
3.7.19	Queue<T> Class . . . . .	1109
3.7.19.1	Example . . . . .	1109
3.7.19.2	Type Definitions . . . . .	1111
3.7.19.3	Member Functions . . . . .	1111
3.7.20	RedBlackTree<KeyType, ValueType, KeyOfValue, Compare> Class . . . . .	1124
3.7.20.1	Type Definitions . . . . .	1125
3.7.20.2	Member Functions . . . . .	1125
3.7.21	RedBlackTreeNodeIterator<T, R, P> Class . . . . .	1142
3.7.21.1	Type Definitions . . . . .	1143
3.7.21.2	Member Functions . . . . .	1143
3.7.21.3	Nonmember Functions . . . . .	1154
3.7.22	Set<T, C> Class . . . . .	1156
3.7.22.1	Example . . . . .	1156
3.7.22.2	Type Definitions . . . . .	1158
3.7.22.3	Member Functions . . . . .	1158
3.7.22.4	Nonmember Functions . . . . .	1178
3.7.23	Stack<T> Class . . . . .	1183
3.7.23.1	Example . . . . .	1183
3.7.23.2	Type Definitions . . . . .	1184
3.7.23.3	Member Functions . . . . .	1184
3.8	Functions . . . . .	1198
3.8.24	ConstructiveCopy(ValueType*, ValueType*, int) Function . . . . .	1199
3.8.25	ConstructiveMove(ValueType*, ValueType*, int) Function . . . . .	1200
3.8.26	Destroy(ValueType*, int) Function . . . . .	1201
3.8.27	GetHashCode(char) Function . . . . .	1202

3.8.28	GetHashCode(const System.String&) Function . . . . .	1203
3.8.29	GetHashCode(long) Function . . . . .	1204
3.8.30	GetHashCode(ulong) Function . . . . .	1205
3.8.31	GetHashCode(void*) Function . . . . .	1206
<b>4</b>	<b>System.Concepts Namespace</b>	<b>1207</b>
4.9	Concepts . . . . .	1210
4.9.1	AdditiveGroup<T> Concept . . . . .	1215
4.9.2	AdditiveMonoid<T> Concept . . . . .	1216
4.9.3	AdditiveSemigroup<T> Concept . . . . .	1217
4.9.4	BackInsertionSequence<T> Concept . . . . .	1218
4.9.5	BidirectionalContainer<T> Concept . . . . .	1219
4.9.6	BidirectionalIterator<T> Concept . . . . .	1220
4.9.7	BinaryFunction<T> Concept . . . . .	1221
4.9.8	BinaryOperation<T> Concept . . . . .	1222
4.9.9	BinaryPredicate<T> Concept . . . . .	1223
4.9.10	CommutativeSemiring<T> Concept . . . . .	1224
4.9.11	Container<T> Concept . . . . .	1225
4.9.12	CopyAssignable<T, U> Concept . . . . .	1226
4.9.12.1	Example . . . . .	1226
4.9.13	CopyAssignable<T> Concept . . . . .	1227
4.9.13.1	Example . . . . .	1227
4.9.14	CopyConstructible<T> Concept . . . . .	1229
4.9.14.1	Example . . . . .	1229
4.9.15	Copyable<T> Concept . . . . .	1231
4.9.16	DefaultConstructible<T> Concept . . . . .	1232
4.9.16.1	Example . . . . .	1232
4.9.17	Destructible<T> Concept . . . . .	1234
4.9.17.1	Example . . . . .	1234
4.9.18	EqualityComparable<T, U> Concept . . . . .	1236
4.9.19	EqualityComparable<T> Concept . . . . .	1237
4.9.20	EuclideanSemiring<T> Concept . . . . .	1238
4.9.21	ForwardContainer<T> Concept . . . . .	1239
4.9.22	ForwardIterator<T> Concept . . . . .	1240
4.9.23	FrontInsertionSequence<T> Concept . . . . .	1241
4.9.24	HashFunction<T, Key> Concept . . . . .	1242
4.9.25	InputIterator<T> Concept . . . . .	1243
4.9.26	InsertionSequence<T> Concept . . . . .	1244
4.9.27	Integer<I> Concept . . . . .	1245
4.9.28	KeySelectionFunction<T, Key, Value> Concept . . . . .	1246
4.9.29	LessThanComparable<T, U> Concept . . . . .	1247
4.9.30	LessThanComparable<T> Concept . . . . .	1248
4.9.31	Movable<T> Concept . . . . .	1249
4.9.32	MoveAssignable<T> Concept . . . . .	1250
4.9.33	MoveConstructible<T> Concept . . . . .	1251
4.9.34	MultiplicativeGroup<T> Concept . . . . .	1252
4.9.35	MultiplicativeMonoid<T> Concept . . . . .	1253
4.9.36	MultiplicativeSemigroup<T> Concept . . . . .	1254
4.9.37	OrderedAdditiveGroup<T> Concept . . . . .	1255
4.9.38	OrderedAdditiveMonoid<T> Concept . . . . .	1256
4.9.39	OrderedAdditiveSemigroup<T> Concept . . . . .	1257
4.9.40	OrderedMultiplicativeSemigroup<T> Concept . . . . .	1258
4.9.41	OutputIterator<T> Concept . . . . .	1259
4.9.42	RandomAccessContainer<T> Concept . . . . .	1260

4.9.43	RandomAccessIterator<T> Concept . . . . .	1261
4.9.44	Regular<T> Concept . . . . .	1262
4.9.44.1	Example . . . . .	1262
4.9.45	Relation<T, U, V> Concept . . . . .	1265
4.9.46	Relation<T> Concept . . . . .	1266
4.9.47	Semiregular<T> Concept . . . . .	1267
4.9.47.1	Example . . . . .	1267
4.9.48	Semiring<T> Concept . . . . .	1269
4.9.49	SignedInteger<I> Concept . . . . .	1270
4.9.50	TotallyOrdered<T, U> Concept . . . . .	1271
4.9.51	TotallyOrdered<T> Concept . . . . .	1272
4.9.51.1	Example . . . . .	1272
4.9.52	TrivialIterator<T> Concept . . . . .	1276
4.9.53	UnaryFunction<T> Concept . . . . .	1277
4.9.54	UnaryOperation<T> Concept . . . . .	1278
4.9.55	UnaryPredicate<T> Concept . . . . .	1279
4.9.56	UnsignedInteger<U> Concept . . . . .	1280
<b>5</b>	<b>System.IO Namespace</b> . . . . .	<b>1281</b>
5.10	Classes . . . . .	1282
5.10.1	BinaryFileStream Class . . . . .	1283
5.10.1.1	Member Functions . . . . .	1283
5.10.2	CloseFileException Class . . . . .	1326
5.10.2.1	Member Functions . . . . .	1326
5.10.3	IOBuffer Class . . . . .	1334
5.10.3.1	Member Functions . . . . .	1334
5.10.4	IOException Class . . . . .	1341
5.10.4.1	Member Functions . . . . .	1341
5.10.5	InputStream Class . . . . .	1349
5.10.5.1	Member Functions . . . . .	1349
5.10.6	InputStream Class . . . . .	1365
5.10.6.1	Member Functions . . . . .	1365
5.10.7	InputStreamString Class . . . . .	1373
5.10.7.1	Member Functions . . . . .	1373
5.10.8	InvalidPathException Class . . . . .	1384
5.10.8.1	Member Functions . . . . .	1384
5.10.9	OpenFileException Class . . . . .	1392
5.10.9.1	Member Functions . . . . .	1392
5.10.10	OutputStream Class . . . . .	1400
5.10.10.1	Member Functions . . . . .	1400
5.10.11	OutputStream Class . . . . .	1449
5.10.11.1	Member Functions . . . . .	1449
5.10.11.2	Nonmember Functions . . . . .	1483
5.10.12	OutputStreamString Class . . . . .	1501
5.10.12.1	Member Functions . . . . .	1501
5.10.13	Path Class . . . . .	1540
5.10.13.1	Member Functions . . . . .	1540
5.11	Functions . . . . .	1552
5.11.14	CreateDirectories(const System.String&) Function . . . . .	1553
5.11.15	DirectoryExists(const System.String&) Function . . . . .	1554
5.11.16	FileContentsEqual(const System.String&, const System.String&) Function . . . . .	1555
5.11.17	FileExists(const System.String&) Function . . . . .	1556
5.11.18	GetCurrentWorkingDirectory() Function . . . . .	1557
5.11.19	GetFullPath(const System.String&) Function . . . . .	1558

5.11.20 PathExists(const System.String&) Function . . . . .	1559
5.11.21 ReadFile(const System.String&) Function . . . . .	1560
5.12 Enumerations . . . . .	1561
5.12.21.1 OpenMode Enumeration . . . . .	1562
<b>6 System.Security Namespace</b>	<b>1563</b>
6.13 Classes . . . . .	1564
6.13.1 Sha1 Class . . . . .	1565
6.13.1.1 Member Functions . . . . .	1565
6.13.2 Sha256 Class . . . . .	1576
6.13.2.1 Member Functions . . . . .	1576
6.13.3 Sha512 Class . . . . .	1587
6.13.3.1 Member Functions . . . . .	1587
6.14 Functions . . . . .	1598
6.14.4 GetSha1FileDigest(const System.String&) Function . . . . .	1599
6.14.5 GetSha1MessageDigest(const System.String&) Function . . . . .	1600
6.14.6 GetSha256FileDigest(const System.String&) Function . . . . .	1601
6.14.7 GetSha256MessageDigest(const System.String&) Function . . . . .	1602
6.14.8 GetSha512FileDigest(const System.String&) Function . . . . .	1603
6.14.9 GetSha512MessageDigest(const System.String&) Function . . . . .	1604
6.14.10 LeftRotate(uint, uint) Function . . . . .	1605
6.14.11 RightRotate(uint, uint) Function . . . . .	1606
6.14.12 RightRotate(ulong, ulong) Function . . . . .	1607
<b>7 System.Text Namespace</b>	<b>1608</b>
7.15 Classes . . . . .	1609
7.15.1 CodeFormatter Class . . . . .	1610
7.15.1.1 Member Functions . . . . .	1610
7.16 Functions . . . . .	1624
7.16.2 CharStr(char) Function . . . . .	1625
7.16.3 HexEscape(char) Function . . . . .	1626
7.16.4 MakeCharLiteral(char) Function . . . . .	1627
7.16.5 MakeStringLiteral(const System.String&) Function . . . . .	1628
7.16.6 StringStr(const System.String&) Function . . . . .	1629
7.16.7 Trim(const System.String&) Function . . . . .	1630
7.16.8 TrimAll(const System.String&) Function . . . . .	1631
<b>8 System.Threading Namespace</b>	<b>1632</b>
8.17 Concepts . . . . .	1633
8.17.1 Lockable<M> Concept . . . . .	1634
8.18 Classes . . . . .	1635
8.18.2 ConditionVariable Class . . . . .	1636
8.18.2.1 Member Functions . . . . .	1636
8.18.3 LockGuard<M> Class . . . . .	1644
8.18.3.1 Member Functions . . . . .	1644
8.18.4 Mutex Class . . . . .	1649
8.18.4.1 Remarks . . . . .	1649
8.18.4.2 Member Functions . . . . .	1649
8.18.5 RecursiveMutex Class . . . . .	1656
8.18.5.1 Member Functions . . . . .	1656
8.18.6 Thread Class . . . . .	1659
8.18.6.1 Member Functions . . . . .	1659
8.18.6.2 Nonmember Functions . . . . .	1668
8.18.7 ThreadingException Class . . . . .	1670

8.18.7.1 Member Functions . . . . .	1670
8.19 Functions . . . . .	1678
8.19.8 SleepFor(System.Duration) Function . . . . .	1679
8.19.9 SleepUntil(System.TimePoint) Function . . . . .	1680
8.19.10 ThreadStart(void*) Function . . . . .	1681
8.20 Delegates . . . . .	1682
8.20.11 ThreadFun Delegate . . . . .	1683
8.21 Constants . . . . .	1684

# 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

system
support
os
C run-time

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

# Copyrights

=====

Copyright (c) 2012-2016 Seppo Laakko  
<http://sourceforge.net/projects/cmajor/>

Distributed under the GNU General Public License, version 3 (GPLv3).  
(See accompanying LICENSE.txt or <http://www.gnu.org/licenses/gpl.html>)

=====

```
* Copyright (c) 1994
* Hewlett-Packard Company
*
* Permission to use, copy, modify, distribute and sell this software
* and its documentation for any purpose is hereby granted without fee,
* provided that the above copyright notice appear in all copies and
* that both that copyright notice and this permission notice appear
* in supporting documentation. Hewlett-Packard Company makes no
* representations about the suitability of this software for any
* purpose. It is provided "as is" without express or implied warranty.
*
*
* Copyright (c) 1996,1997
* Silicon Graphics Computer Systems, Inc.
*
* Permission to use, copy, modify, distribute and sell this software
* and its documentation for any purpose is hereby granted without fee,
* provided that the above copyright notice appear in all copies and
* that both that copyright notice and this permission notice appear
* in supporting documentation. Silicon Graphics makes no
* representations about the suitability of this software for any
* purpose. It is provided "as is" without express or implied warranty.
*
```

Copyright (c) 2009 Alexander Stepanov and Paul McJones

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. The authors make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

# Namespaces

Namespace	Description
Global	The global namespace contains intrinsic concepts.
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.Security	Contains classes and functions for computing cryptographic hash functions.
System.Text	Contains classes and functions for manipulating text.
System.Threading	Contains classes and functions for controlling multiple threads of execution.

# 1 Global Namespace

The global namespace contains intrinsic concepts.

## 1.1 Concepts

Concept	Description
<code>Common&lt;T, U&gt;</code>	Sets a requirement that types T and U have a common type. The common type is exposed as type parameter CommonType.
<code>Convertible&lt;T, U&gt;</code>	Sets a requirement that type T is implicitly convertible to type U.
<code>Derived&lt;T, U&gt;</code>	Sets a requirement that types T and U are class types and class T is derived from type class U.
<code>ExplicitlyConvertible&lt;T, U&gt;</code>	Sets a requirement that type T is explicitly convertible to type U.
<code>NonReferenceType&lt;T&gt;</code>	Sets a requirement that type T is not a reference type.
<code>Same&lt;T, U&gt;</code>	Sets a requirement that type T is exactly the same type as U.

### 1.1.1 Common<T, U> Concept

Sets a requirement that types T and U have a common type. The common type is exposed as type parameter CommonType.

#### Syntax

```
public concept Common<T, U>;
```

#### Remarks

For example common type for **int** and **double** is **double** because **int** can be implicitly converted to **double**.

### 1.1.2 Convertible<T, U> Concept

Sets a requirement that type T is implicitly convertible to type U.

#### Syntax

```
public concept Convertible<T, U>;
```

#### Remarks

For example **int** is implicitly convertible to **double** but not vice versa. More generally: conversions that do not lose information are implicit in Cmajor.

### 1.1.3 Derived<T, U> Concept

Sets a requirement that types T type and U are class types and class T is derived from type class U.

#### Syntax

```
public concept Derived<T, U>;
```

### 1.1.4 `ExplicitlyConvertible<T, U>` Concept

Sets a requirement that type T is explicitly convertible to type U.

#### Syntax

```
public concept ExplicitlyConvertible<T, U>;
```

#### Remarks

For example `double` is explicitly convertible to `int`. Explicit conversions need a `cast`.

### 1.1.5 NonReferenceType<T> Concept

Sets a requirement that type T is not a reference type.

#### Syntax

```
public concept NonReferenceType<T>;
```

### 1.1.6 Same<T, U> Concept

Sets a requirement that type T is exatly the same type as U.

#### Syntax

```
public concept Same<T, U>;
```

## 2 System Namespace

Contains fundamental classes, functions and type definitions.

Figures 2.1 and 2.2 contain the classes in this namespace.

Figure 2.1: Class Diagram 1: Basic Classes

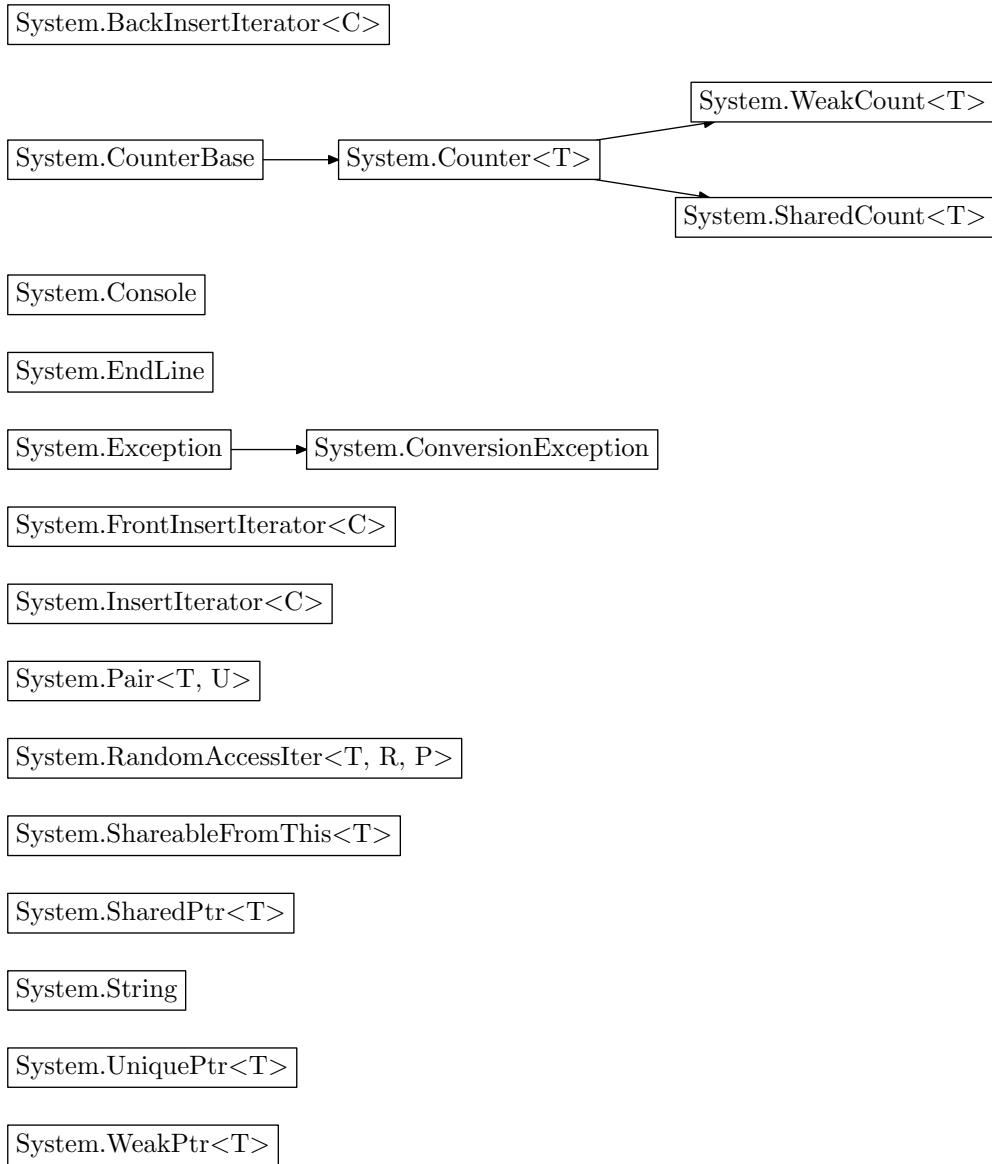


Figure 2.2: Class Diagram 2: Function Objects



## 2.2 Classes

Class	Description
<code>BackInsertIterator&lt;C&gt;</code>	An output iterator that inserts elements to the end of a back insertion sequence.
<code>BackInsertProxy&lt;C&gt;</code>	Implementation detail.
<code>BinaryFun&lt;Argument1, Argument2, Result&gt;</code>	A base class for binary function objects.
<code>BinaryPred&lt;Argument1, Argument2&gt;</code>	A base class for binary predicates.
<code>BitAnd&lt;T&gt;</code>	Bitwise AND binary function object.
<code>BitNot&lt;T&gt;</code>	Bitwise complement unary function object.
<code>BitOr&lt;T&gt;</code>	Bitwise inclusive OR binary function object.
<code>BitXor&lt;T&gt;</code>	Bitwise exclusive OR binary function object.
<code>Console</code>	A static class that contains console input and output functions.
<code>ConversionException</code>	An exception thrown when a conversion function fails.
<code>Counter&lt;T&gt;</code>	A counter class that stores a pointer to the counted object.
<code>CounterBase</code>	An abstract base class for <code>SharedCount&lt;T&gt;</code> and <code>WeakCount&lt;T&gt;</code> that keeps track of use count and weak count.
<code>Date</code>	A class for representing a date.
<code>Divides&lt;T&gt;</code>	Division binary function object.
<code>Duration</code>	Represents a duration in nanoseconds.
<code>EndLine</code>	Represents an end of line character.
<code>EqualTo2&lt;T, U&gt;</code>	An <i>equal to</i> binary predicate.
<code>EqualTo&lt;T&gt;</code>	An <i>equal to</i> relation.
<code>Exception</code>	A base class for all exception classes.
<code>ExceptionPtr</code>	The <code>ExceptionPtr</code> class can be used to hold an exception with full type information.

<code>FrontInsertIterator&lt;C&gt;</code>	An output iterator that inserts elements to the front of a container.
<code>FrontInsertProxy&lt;C&gt;</code>	Implementation detail.
<code>Greater2&lt;T, U&gt;</code>	A <i>greater than</i> binary predicate.
<code>Greater&lt;T&gt;</code>	A <i>greater than</i> relation.
<code>GreaterOrEqualTo2&lt;T, U&gt;</code>	A <i>greater than or equal to</i> binary predicate.
<code>GreaterOrEqualTo&lt;T&gt;</code>	A <i>greater than or equal to</i> relation.
<code>Identity&lt;T&gt;</code>	An identity unary function object.
<code>InsertIterator&lt;C&gt;</code>	An output iterator that inserts elements to given position of a container.
<code>InsertProxy&lt;C&gt;</code>	Implementation detail.
<code>Less2&lt;T, U&gt;</code>	A <i>less than</i> binary predicate.
<code>Less&lt;T&gt;</code>	A <i>less than</i> relation.
<code>LessOrEqualTo2&lt;T, U&gt;</code>	A <i>less than or equal to</i> binary predicate.
<code>LessOrEqualTo&lt;T&gt;</code>	A <i>less than or equal to</i> relation.
<code>LogicalAnd&lt;T&gt;</code>	A <i>logical AND</i> binary predicate.
<code>LogicalNot&lt;T&gt;</code>	A <i>logical NOT</i> unary predicate.
<code>LogicalOr&lt;T&gt;</code>	A <i>logical OR</i> binary predicate.
<code>MT</code>	Mersenne Twister pseudorandom number generator.
<code>Minus&lt;T&gt;</code>	A subtraction binary function object.
<code>Multiplies&lt;T&gt;</code>	A multiplication binary function object.
<code>Negate&lt;T&gt;</code>	A negation unary operation.
<code>NotEqualTo2&lt;T, U&gt;</code>	A <i>not equal to</i> binary predicate.
<code>NotEqualTo&lt;T&gt;</code>	An <i>not equal to</i> relation.
<code>Pair&lt;T, U&gt;</code>	A pair of values.

<code>Plus&lt;T&gt;</code>	An addition binary function object.
<code>RandomAccessIter&lt;T, R, P&gt;</code>	A random access iterator that contains a pointer to elements.
<code>Rel&lt;Argument&gt;</code>	A base class for relation function objects.
<code>Remainder&lt;T&gt;</code>	A remainder function object.
<code>SelectFirst&lt;T, U&gt;</code>	A function object for returning the first component of a pair.
<code>SelectSecond&lt;T, U&gt;</code>	A function object for returning the second component of a pair.
<code>ShareableFromThis&lt;T&gt;</code>	A class that implements the “shared from this” idiom.
<code>SharedCount&lt;T&gt;</code>	A handle to a <code>Counter&lt;T&gt;</code> that maintains the use count portion of the counter.
<code>SharedPtr&lt;T&gt;</code>	A shared pointer to an object.
<code>ShiftLeft&lt;T&gt;</code>	Shift left binary function.
<code>ShiftRight&lt;T&gt;</code>	Shift right binary function.
<code>String</code>	A string of ASCII characters.
<code>TimeError</code>	An exception thrown when a time function fails.
<code>TimePoint</code>	Represents a point in time as specified nanoseconds elapsed since epoch.
<code>TracedFun</code>	Function tracer.
<code>Tracer</code>	A utility class for tracing entry and exit of some operation.
<code>UnaryFun&lt;Argument, Result&gt;</code>	A base class for unary function objects.
<code>UnaryPred&lt;Argument&gt;</code>	A base class for unary predicates.
<code>UniquePtr&lt;T&gt;</code>	A unique pointer to an object.
<code>WeakCount&lt;T&gt;</code>	A handle to a <code>Counter&lt;T&gt;</code> that maintains the weak count portion of the counter.

<code>WeakPtr&lt;T&gt;</code>	Used to break cycles in shared ownership.
<code>uhuge</code>	128-bit unsigned integer type.

### 2.2.1 BackInsertIterator<C> Class

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

#### Syntax

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

#### Constraint

C is [BackInsertionSequence](#)

#### Model of

[OutputIterator<T>](#)

#### 2.2.1.1 Remarks

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

#### 2.2.1.2 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();
}
```

### 2.2.1.3 Type Definitions

Name	Type	Description
PointerType	<code>System.BackInsertProxy&lt;C&gt;*</code>	Pointer to implementation defined proxy type.
ReferenceType	<code>System.BackInsertProxy&lt;C&gt;&amp;</code>	Reference to implementation defined proxy type.
ValueType	<code>System.BackInsertProxy&lt;C&gt;</code>	Implementation defined proxy type.

### 2.2.1.4 Member Functions

Member Function	Description	
<code>BackInsertIterator&lt;C&gt;()</code>		Constructor. Default constructs a back insert iterator.
<code>BackInsertIterator&lt;C&gt;(const BackInsertIterator&lt;C&gt;&amp;)</code>	<code>System.-</code>	Copy constructor.
<code>operator=(const System.BackInsertIterator&lt;C&gt;- &amp;)</code>		Copy assignment.
<code>BackInsertIterator&lt;C&gt;(System.- BackInsertIterator&lt;C&gt;&amp;&amp;)</code>		Move constructor.
<code>operator=(System.BackInsertIterator&lt;C&gt;&amp;&amp;)</code>		Move assignment.
<code>BackInsertIterator&lt;C&gt;(C&amp;)</code>		Constructor. Constructs a back insert iterator with a container.
<code>operator*()</code>		Returns a reference to a proxy object that inserts values to the end of a container.
<code>operator++()</code>		Advances the iterator to the next element.
<code>operator-&gt;()</code>		Returns a pointer to a proxy object that inserts values to the end of a container.

**BackInsertIterator<C>() Member Function**

Constructor. Default constructs a back insert iterator.

**Syntax**

```
public BackInsertIterator<C>();
```

**BackInsertIterator<C>(const System.BackInsertIterator<C>&) Member Function**

Copy constructor.

**Syntax**

```
public BackInsertIterator<C>(const System.BackInsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.BackInsertIterator<C>&	Argument to copy.

**operator=(const System.BackInsertIterator<C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BackInsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.BackInsertIterator<C>&	Argument to assign.

**BackInsertIterator<C>(System.BackInsertIterator<C>&&) Member Function**

Move constructor.

**Syntax**

```
public BackInsertIterator<C>(System.BackInsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.BackInsertIterator<C>&&	Argument to move from.

**operator=(System.BackInsertIterator<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BackInsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.BackInsertIterator<C>&&	Argument to assign from.

**BackInsertIterator<C>(C&) Member Function**

Constructor. Constructs a back insert iterator with a container.

**Syntax**

```
public BackInsertIterator<C>(C& c);
```

**Parameters**

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

**operator\*() Member Function**

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

**Syntax**

```
public System.BackInsertProxy<C>& operator*();
```

**Returns**

[System.BackInsertProxy<C>&](#)

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

**operator++() Member Function**

Advances the iterator to the next element.

**Syntax**

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

**Returns**

[System.BackInsertIterator<C>&](#)

Returns a reference to the iterator.

**operator->() Member Function**

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

**Syntax**

```
public System.BackInsertProxy<C>* operator->();
```

**Returns**

[System.BackInsertProxy<C>\\*](#)

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

## 2.2.2 BackInsertProxy<C> Class

Implementation detail.

### Syntax

```
public class BackInsertProxy<C>;
```

### Constraint

C is [BackInsertionSequence](#)

#### 2.2.2.1 Member Functions

Member Function	Description
<code>BackInsertProxy&lt;C&gt;(System.-</code>	Move constructor.
<code>BackInsertProxy&lt;C&gt;&amp;&amp;)</code>	
<code>operator=(System.BackInsertProxy&lt;C&gt;&amp;&amp;)</code>	Move assignment.

**BackInsertProxy<C>(System.BackInsertProxy<C>&&) Member Function**

Move constructor.

**Syntax**

```
public BackInsertProxy<C>(System.BackInsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.BackInsertProxy<C>&&	Argument to move from.

**operator=(System.BackInsertProxy<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BackInsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.BackInsertProxy<C>&&	Argument to assign from.

### 2.2.3 BinaryFun<Argument1, Argument2, Result> Class

A base class for binary function objects.

#### Syntax

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

#### Constraint

Argument1 is [Semiregular](#) and Argument2 is [Semiregular](#)

#### 2.2.3.1 Remarks

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

### 2.2.3.2 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.

### 2.2.3.3 Member Functions

Member Function	Description
<code>BinaryFun&lt;Argument1, Argument2, Result&gt;()</code>	Default constructor.
<code>BinaryFun&lt;Argument1, Argument2, Result&gt;::</code> <code>(const System.BinaryFun&lt;Argument1, Argument2, Result&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BinaryFun&lt;Argument1, Argument2, Result&gt;&amp;)</code>	Copy assignment.
<code>BinaryFun&lt;Argument1, Argument2, Result&gt;::</code> <code>(System.BinaryFun&lt;Argument1, Argument2, Result&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BinaryFun&lt;Argument1, Argument2, Result&gt;&amp;&amp;)</code>	Move assignment.

**BinaryFun<Argument1, Argument2, Result>() Member Function**

Default constructor.

**Syntax**

```
public BinaryFun<Argument1, Argument2, Result>();
```

**BinaryFun<Argument1, Argument2, Result>(const System.BinaryFun<Argument1, Argument2, Result>&) Member Function**

Copy constructor.

#### Syntax

```
public BinaryFun<Argument1, Argument2, Result>(const System.BinaryFun<Argument1, Argument2, Result>& that);
```

#### Parameters

Name	Type	Description
that	const System.BinaryFun<Argument1, Argument2, Result>&	Argument to copy.

**operator=(const System.BinaryFun<Argument1, Argument2, Result>&) Member Function**

Copy assignment.

#### Syntax

```
public void operator=(const System.BinaryFun<Argument1, Argument2, Result>& that);
```

#### Parameters

Name	Type	Description
that	const System.BinaryFun<Argument1, Argument2, Result>&	Argument to assign.

**BinaryFun<Argument1, Argument2, Result>(System.BinaryFun<Argument1, Argument2, Result>&&) Member Function**

Move constructor.

**Syntax**

```
public BinaryFun<Argument1, Argument2, Result>(System.BinaryFun<Argument1, Argument2, Result>&& that);
```

**Parameters**

Name	Type	Description
that	System.BinaryFun<Argument1, Argument2, Result>-&&	Argument to move from.

**operator=(System.BinaryFun<Argument1, Argument2, Result>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BinaryFun<Argument1, Argument2, Result>&& that);
```

**Parameters**

Name	Type	Description
that	System.BinaryFun<Argument1, Argument2, Result>-&&	Argument to assign from.

## 2.2.4 BinaryPred<Argument1, Argument2> Class

A base class for binary predicates.

### Syntax

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

### Constraint

Argument1 is [Semiregular](#) and Argument2 is [Semiregular](#)

### Model of

[BinaryPredicate<T>](#)

### Base Class

[System.BinaryFun<Argument1, Argument2, bool>](#)

#### 2.2.4.1 Remarks

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

#### 2.2.4.2 Member Functions

Member Function	Description
<code>BinaryPred&lt;Argument1, Argument2&gt;()</code>	Default constructor.
<code>BinaryPred&lt;Argument1, Argument2&gt;(const System.BinaryPred&lt;Argument1, Argument2&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BinaryPred&lt;Argument1, Argument2&gt;&amp;)</code>	Copy assignment.
<code>BinaryPred&lt;Argument1, Argument2&gt;(System.- BinaryPred&lt;Argument1, Argument2&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BinaryPred&lt;Argument1, Argument2&gt;&amp;&amp;)</code>	Move assignment.

**BinaryPred<Argument1, Argument2>() Member Function**

Default constructor.

**Syntax**

```
public BinaryPred<Argument1, Argument2>();
```

**BinaryPred<Argument1, Argument2>(const System.BinaryPred<Argument1, Argument2>&)**  
**Member Function**

Copy constructor.

#### Syntax

```
public BinaryPred<Argument1, Argument2>(const System.BinaryPred<Argument1, Argument2>& that);
```

#### Parameters

Name	Type	Description
that	const System.BinaryPred<Argument1, Argument2>&	Argument to copy.

**operator=(const System.BinaryPred<Argument1, Argument2>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BinaryPred<Argument1, Argument2>& that);
```

**Parameters**

Name	Type	Description
that	const System.BinaryPred<Argument1, Argument2>&	Argument to assign.

**BinaryPred<Argument1, Argument2>(System.BinaryPred<Argument1, Argument2>&&) Member Function**

Move constructor.

**Syntax**

```
public BinaryPred<Argument1, Argument2>(System.BinaryPred<Argument1, Argument2>&&  
that);
```

**Parameters**

Name	Type	Description
that	System.BinaryPred<Argument1, Argument2>&&	Argument to move from.

**operator=(System.BinaryPred<Argument1, Argument2>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BinaryPred<Argument1, Argument2>&& that);
```

**Parameters**

Name	Type	Description
that	System.BinaryPred<Argument1, Argument2>&&	Argument to assign from.

## 2.2.5 BitAnd<T> Class

Bitwise AND binary function object.

### Syntax

```
public class BitAnd<T>;
```

### Constraint

T is [Semiregular](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.5.1 Member Functions

Member Function	Description
<code>BitAnd&lt;T&gt;()</code>	Default constructor.
<code>BitAnd&lt;T&gt;(const System.BitAnd&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BitAnd&lt;T&gt;&amp;)</code>	Copy assignment.
<code>BitAnd&lt;T&gt;(System.BitAnd&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BitAnd&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns bitwise AND of <code>left</code> and <code>right</code> .

**BitAnd<T>() Member Function**

Default constructor.

**Syntax**

```
public BitAnd<T>();
```

**BitAnd<T>(const System.BitAnd<T>&) Member Function**

Copy constructor.

**Syntax**

```
public BitAnd<T>(const System.BitAnd<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitAnd<T>&	Argument to copy.

**operator=(const System.BitAnd<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BitAnd<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitAnd<T>&	Argument to assign.

**BitAnd<T>(System.BitAnd<T>&&) Member Function**

Move constructor.

**Syntax**

```
public BitAnd<T>(System.BitAnd<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitAnd<T>&&	Argument to move from.

**operator=(System.BitAnd<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BitAnd<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitAnd<T>&&	Argument to assign from.

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

Returns bitwise AND of [left](#) and [right](#).

**Syntax**

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

**Parameters**

Name	Type	Description
left	const T&	Left argument.
right	const T&	Right argument.

**Returns**

T

Returns bitwise AND of [left](#) and [right](#).

## 2.2.6 BitNot<T> Class

Bitwise complement unary function object.

### Syntax

```
public class BitNot<T>;
```

### Constraint

T is [Semiregular](#)

### Base Class

[System.UnaryFun<T, T>](#)

### 2.2.6.1 Member Functions

Member Function	Description
<code>BitNot&lt;T&gt;()</code>	Default constructor.
<code>BitNot&lt;T&gt;(const System.BitNot&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BitNot&lt;T&gt;&amp;)</code>	Copy assignment.
<code>BitNot&lt;T&gt;(System.BitNot&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BitNot&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;) const</code>	Returns bitwise complement of <a href="#">operand</a> .

**BitNot<T>() Member Function**

Default constructor.

**Syntax**

```
public BitNot<T>();
```

**BitNot<T>(const System.BitNot<T>&) Member Function**

Copy constructor.

**Syntax**

```
public BitNot<T>(const System.BitNot<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitNot<T>&	Argument to copy.

**operator=(const System.BitNot<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BitNot<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitNot<T>&	Argument to assign.

**BitNot<T>(System.BitNot<T>&&) Member Function**

Move constructor.

**Syntax**

```
public BitNot<T>(System.BitNot<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitNot<T>&&	Argument to move from.

**operator=(System.BitNot<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BitNot<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitNot<T>&&	Argument to assign from.

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

Returns bitwise complement of [operand](#).

**Syntax**

```
public T operator()(const T& operand) const;
```

**Parameters**

Name	Type	Description
operand	const T&	Argument.

**Returns**

T

Returns bitwise complement of [operand](#).

### 2.2.7 BitOr<T> Class

Bitwise inclusive OR binary function object.

#### Syntax

```
public class BitOr<T>;
```

#### Constraint

T is [Semiregular](#)

#### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.7.1 Member Functions

Member Function	Description
<code>BitOr&lt;T&gt;()</code>	Default constructor.
<code>BitOr&lt;T&gt;(const System.BitOr&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BitOr&lt;T&gt;&amp;)</code>	Copy assignment.
<code>BitOr&lt;T&gt;(System.BitOr&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BitOr&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns bitwise inclusive OR of <code>left</code> and <code>right</code> .

**BitOr<T>() Member Function**

Default constructor.

**Syntax**

```
public BitOr<T>();
```

**BitOr<T>(const System.BitOr<T>&) Member Function**

Copy constructor.

**Syntax**

```
public BitOr<T>(const System.BitOr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitOr<T>&	Argument to copy.

**operator=(const System.BitOr<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BitOr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitOr<T>&	Argument to assign.

**BitOr<T>(System.BitOr<T>&&) Member Function**

Move constructor.

**Syntax**

```
public BitOr<T>(System.BitOr<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitOr<T>&&	Argument to move from.

**operator=(System.BitOr<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BitOr<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitOr<T>&&	Argument to assign from.

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

Returns bitwise inclusive OR of [left](#) and [right](#).

**Syntax**

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

**Parameters**

Name	Type	Description
left	const T&	Left argument.
right	const T&	Right argument.

**Returns**

T

Returns bitwise inclusive OR of [left](#) and [right](#).

## 2.2.8 BitXor<T> Class

Bitwise exclusive OR binary function object.

### Syntax

```
public class BitXor<T>;
```

### Constraint

T is [Semiregular](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

### 2.2.8.1 Member Functions

Member Function	Description
<code>BitXor&lt;T&gt;()</code>	Default constructor.
<code>BitXor&lt;T&gt;(const System.BitXor&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.BitXor&lt;T&gt;&amp;)</code>	Copy assignment.
<code>BitXor&lt;T&gt;(System.BitXor&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.BitXor&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns bitwise exclusive OR of <code>left</code> and <code>right</code> .

**BitXor<T>() Member Function**

Default constructor.

**Syntax**

```
public BitXor<T>();
```

**BitXor<T>(const System.BitXor<T>&) Member Function**

Copy constructor.

**Syntax**

```
public BitXor<T>(const System.BitXor<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitXor<T>&	Argument to copy.

**operator=(const System.BitXor<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.BitXor<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.BitXor<T>&	Argument to assign.

**BitXor<T>(System.BitXor<T>&&) Member Function**

Move constructor.

**Syntax**

```
public BitXor<T>(System.BitXor<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitXor<T>&&	Argument to move from.

**operator=(System.BitXor<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.BitXor<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.BitXor<T>&&	Argument to assign from.

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

Returns bitwise exclusive OR of [left](#) and [right](#).

**Syntax**

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

**Parameters**

Name	Type	Description
left	const T&	Left argument.
right	const T&	Right argument.

**Returns**

T

Returns bitwise exclusive OR of [left](#) and [right](#).

## 2.2.9 Console Class

A static class that contains console input and output functions.

### Syntax

```
public static class Console;
```

#### 2.2.9.1 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\(System.UniquePtr<System.IO.InputStream>&&\)](#), [SetOut\(System.UniquePtr<System.IO.OutputStream>&&\)](#) and [SetError\(System.UniquePtr<System.IO.OutputStream>&&\)](#) functions.

#### 2.2.9.2 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();
}
```

#### 2.2.9.3 Member Functions

Member Function	Description
<a href="#">Error()</a>	Returns a reference to the console error stream.
<a href="#">In()</a>	Returns a reference to the console input stream.
<a href="#">Out()</a>	Returns a reference to the console output stream.
<a href="#">ReadLine()</a>	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(System.UniquePtr<System.IO.- OutputStream>&&)	Sets the error stream the <b>Console</b> class uses.
SetIn(System.UniquePtr<System.IO.- InputStream>&&)	Sets the input stream the <b>Console</b> class uses.
SetOut(System.UniquePtr<System.IO.- OutputStream>&&)	Sets the output stream the <b>Console</b> class uses.
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(const System.String&) const	Writes a string to the console output stream.
Write(const char*) const	Writes a C-style string 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(bool)	Writes a Boolean value followed by a newline to the console output stream.

<code>WriteLine(byte)</code>	Writes a byte followed by a newline to the console output stream.
<code>WriteLine(char)</code>	Writes a character followed by a newline to the console output stream.
<code>WriteLine(const System.String&amp;)</code> const	Writes a string followed by a newline to the console output stream.
<code>WriteLine(const char*)</code> const	Writes a C-style string followed by a newline to the console output stream.
<code>WriteLine(double)</code>	Writes a double value followed by a newline to the console output stream.
<code>WriteLine(float)</code>	Writes a float value followed by a newline to the console output stream.
<code>WriteLine(int)</code>	Writes an int value followed by a newline to the console output stream.
<code>WriteLine(long)</code>	Writes a long value followed by a newline to the console output stream.
<code>WriteLine(sbyte)</code>	Writes a signed byte followed by a newline to the console output stream.
<code>WriteLine(short)</code>	Writes a short value followed by a newline to the console output stream.
<code>WriteLine(uint)</code>	Writes a unsigned int followed by a newline to the console output stream.
<code>WriteLine(ulong)</code>	Writes an unsigned long followed by a newline to the console output stream.
<code>WriteLine(ushort)</code>	Writes an unsigned short followed by a newline to the console output stream.

**Error() Member Function**

Returns a reference to the console error stream.

**Syntax**

```
public static System.IO.OutputStream& Error();
```

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the console error stream.

**In() Member Function**

Returns a reference to the console input stream.

**Syntax**

```
public static System.IO.InputStream& In();
```

**Returns**

[System.IO.InputStream&](#)

Returns a reference to the console input stream.

**Out() Member Function**

Returns a reference to the console output stream.

**Syntax**

```
public static System.IO.OutputStream& Out();
```

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the console output stream.

**ReadLine() Member Function**

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

**Syntax**

```
public static System.String ReadLine();
```

**Returns**

[System.String](#)

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

**ReadToEnd() Member Function**

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

**Syntax**

```
public static System.String ReadToEnd();
```

**Returns**

[System.String](#)

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

**SetError(System.UniquePtr<System.IO.OutputStream>&&) Member Function**

Sets the error stream the [Console](#) class uses.

**Syntax**

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

**Parameters**

Name	Type	Description
err_	<a href="#">System.UniquePtr&lt;System.IO.OutputStream&gt;&amp;&amp;</a>	A unique pointer to an output stream.

**SetIn(System.UniquePtr<System.IO.InputStream>&&)** Member Function

Sets the input stream the [Console](#) class uses.

**Syntax**

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

**Parameters**

Name	Type	Description
in_	<a href="#">System.UniquePtr&lt;System.IO.InputStream&gt;&amp;&amp;</a>	A unique pointer to an input stream.

**SetOut(System.UniquePtr<System.IO.OutputStream>&&)** Member Function

Sets the output stream the [Console](#) class uses.

**Syntax**

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

**Parameters**

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

**Example**

```
using System;
using System.IO;

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

**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.

**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.

**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.

**Write(const System.String&) const Member Function**

Writes a string to the console output stream.

**Syntax**

```
public static void Write(const System.String& s) const;
```

**Parameters**

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

**Write(const char\*) const Member Function**

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

**Syntax**

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

**Parameters**

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

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**WriteLine() Member Function**

Writes a newline to the console output stream.

**Syntax**

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

**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.

**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.

**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.

**WriteLine(const System.String&) const Member Function**

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

**Syntax**

```
public static void WriteLine(const System.String& s) const;
```

**Parameters**

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

**WriteLine(const char\*) const Member Function**

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

**Syntax**

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

**Parameters**

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

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

## 2.2.10 ConversionException Class

An exception thrown when a conversion function fails.

### Syntax

```
public class ConversionException;
```

### Base Class

[System.Exception](#)

#### 2.2.10.1 Remarks

Conversion functions that throw **ConversionException** are: [ParseBool\(const System.String&\)](#), [ParseDate\(const System.String&\)](#), [ParseDouble\(const System.String&\)](#) [ParseHex\(const System.String&\)](#), [ParseHexUHuge\(const System.String&\)](#), [ParseInt\(const System.String&\)](#), [ParseUInt\(const System.String&\)](#), [ParseUHuge\(const System.String&\)](#) and [ParseULong\(const System.String&\)](#).

#### 2.2.10.2 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());
    }
}
```

#### 2.2.10.3 Member Functions

Member Function	Description
<a href="#">ConversionException()</a>	Default constructor.
<a href="#">ConversionException(const System.-ConversionException&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.ConversionException&amp;)</a>	Copy assignment.
<a href="#">ConversionException(System.-ConversionException&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.ConversionException&amp;&amp;)</a>	Move assignment.

ConversionException(const System.String&) Copy constructor.

**ConversionException() Member Function**

Default constructor.

**Syntax**

```
public ConversionException();
```

**ConversionException(const System.ConversionException&) Member Function**

Copy constructor.

**Syntax**

```
public ConversionException(const System.ConversionException& that);
```

**Parameters**

Name	Type	Description
that	const System.ConversionException&	Argument to copy.

**operator=(const System.ConversionException&)** Member Function

Copy assignment.

**Syntax**

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

**Parameters**

Name	Type	Description
that	const System.ConversionException&	Argument to assign.

**ConversionException(System.ConversionException&&) Member Function**

Move constructor.

**Syntax**

```
public ConversionException(System.ConversionException&& that);
```

**Parameters**

Name	Type	Description
that	System.ConversionException&&	Argument to move from.

**operator=(System.ConversionException&&)** Member Function

Move assignment.

**Syntax**

```
public void operator=(System.ConversionException&& that);
```

**Parameters**

Name	Type	Description
that	System.ConversionException&&	Argument to assign from.

**ConversionException(const System.String&) Member Function**

Copy constructor.

**Syntax**

```
public ConversionException(const System.String& message_);
```

**Parameters**

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

### 2.2.11 Counter<T> Class

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

#### Syntax

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

#### Base Class

[System.CounterBase](#)

#### 2.2.11.1 Remarks

Base class for [SharedCount<T>](#) and [WeakCount<T>](#).

#### 2.2.11.2 Member Functions

Member Function	Description
<a href="#">Counter&lt;T&gt;()</a>	Default constructor.
<a href="#">Counter&lt;T&gt;(T*)</a>	Constructor. Stores a pointer to counted object.
<a href="#">Dispose()</a>	Overridden. Deletes the counted object.

**Counter<T>() Member Function**

Default constructor.

**Syntax**

```
public Counter<T>();
```

**Counter<T>(T\*) Member Function**

Constructor. Stores a pointer to counted object.

**Syntax**

```
public Counter<T>(T* ptr_);
```

**Parameters**

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

**Dispose() Member Function**

Overridden. Deletes the counted object.

**Syntax**

```
public void Dispose();
```

## 2.2.12 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;
```

#### 2.2.12.1 Member Functions

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

**CounterBase() Member Function**

Constructor. Sets use count and weak count to one.

**Syntax**

```
public CounterBase();
```

**AddReference() Member Function**

Increments use count and weak count.

**Syntax**

```
public void AddReference();
```

**Destruct() Member Function**

Deletes this counter.

**Syntax**

```
public void Destruct();
```

**Dispose() Member Function**

Abstract disposing function overridden in [Counter<T>](#) class.

**Syntax**

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

**GetUseCount() const Member Function**

Returns the use count.

**Syntax**

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

**Returns**

int

Returns the use count.

**Release() Member Function**

Decrements use count and weak count.

**Syntax**

```
public void Release();
```

**Remarks**

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

**WeakAddReference() Member Function**

Increments weak count.

**Syntax**

```
public void WeakAddReference();
```

**WeakRelease() Member Function**

Decrements weak count.

**Syntax**

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

**Remarks**

If weak count has gone to zero, calls System.CounterBase.Destruct to delete this counter object.

**~CounterBase() Member Function**

Destructor.

**Syntax**

```
public ~CounterBase();
```

### 2.2.13 Date Class

A class for representing a date.

#### Syntax

```
public class Date;
```

#### 2.2.13.1 Remarks

[CurrentDate\(\)](#) function returns current date.

#### 2.2.13.2 Member Functions

Member Function	Description
<a href="#">Date()</a>	Default constructor. Constructs a date 1.1.1.
<a href="#">Date(const System.Date&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Date&amp;)</a>	Copy assignment.
<a href="#">Date(System.Date&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Date&amp;&amp;)</a>	Move assignment.
<a href="#">Date(ushort, byte, byte)</a>	Constructor. Initializes a date with given year, month and day.
<a href="#">Day() const</a>	Returns the day part of the date.
<a href="#">Month() const</a>	Returns the month part of the date.
<a href="#">Year() const</a>	Returns the year part of the date.

**Date() Member Function**

Default constructor. Constructs a date 1.1.1.

**Syntax**

```
public Date();
```

**Date(const System.Date&) Member Function**

Copy constructor.

**Syntax**

```
public Date(const System.Date& that);
```

**Parameters**

Name	Type	Description
that	const System.Date&	Argument to copy.

**operator=(const System.Date&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Date& that);
```

**Parameters**

Name	Type	Description
that	const System.Date&	Argument to assign.

**Date(System.Date&&) Member Function**

Move constructor.

**Syntax**

```
public Date(System.Date&& that);
```

**Parameters**

Name	Type	Description
that	System.Date&&	Argument to move from.

**operator=(System.Date&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Date&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Date&amp;&amp;</a>	Argument to assign from.

**Date(ushort, byte, byte) Member Function**

Constructor. Initializes a date with given year, month and day.

**Syntax**

```
public Date(ushort year_, byte month_, byte day_);
```

**Parameters**

Name	Type	Description
year_	ushort	Year.
month_	byte	Month.
day_	byte	Day.

**Day() const Member Function**

Returns the day part of the date.

**Syntax**

```
public byte Day() const;
```

**Returns**

byte

Returns the day part of the date.

**Month() const Member Function**

Returns the month part of the date.

**Syntax**

```
public byte Month() const;
```

**Returns**

byte

Returns the month part of the date.

**Year() const Member Function**

Returns the year part of the date.

**Syntax**

```
public ushort Year() const;
```

**Returns**

ushort

Returns the year part of the date.

**2.2.13.3 Nonmember Functions**

Function	Description
<a href="#">operator&lt;(System.Date, System.Date)</a>	Compares two dates for less than relationship.
<a href="#">operator==(System.Date, System.Date)</a>	Compares two dates for equality.

**operator<(System.Date, System.Date) Function**

Compares two dates for less than relationship.

**Syntax**

```
public bool operator<(System.Date left, System.Date right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.Date</a>	The first date.
right	<a href="#">System.Date</a>	The second date.

**Returns**

bool

Returns true if the first date comes before the second date, false otherwise.

**operator==(System.Date, System.Date) Function**

Compares two dates for equality.

**Syntax**

```
public bool operator==(System.Date left, System.Date right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.Date</a>	The first date.
right	<a href="#">System.Date</a>	The second date.

**Returns**

bool

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

## 2.2.14 Divides<T> Class

Division binary function object.

### Syntax

```
public class Divides<T>;
```

### Constraint

T is [MultiplicativeGroup](#)

### Model of

[BinaryOperation<T>](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.14.1 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);
}
```

#### 2.2.14.2 Member Functions

Member Function	Description
<a href="#">Divides&lt;T&gt;()</a>	Default constructor.
<a href="#">Divides&lt;T&gt;(const System.Divides&lt;T&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Divides&lt;T&gt;&amp;)</a>	Copy assignment.
<a href="#">Divides&lt;T&gt;(System.Divides&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Divides&lt;T&gt;&amp;&amp;)</a>	Move assignment.

`operator()(const T&, const T&) const`

Returns the first argument divided by the second argument.

**Divides<T>() Member Function**

Default constructor.

**Syntax**

```
public Divides<T>();
```

**Divides<T>(const System.Divides<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Divides<T>(const System.Divides<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Divides<T>&	Argument to copy.

**operator=(const System.Divides<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Divides<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Divides<T>&	Argument to assign.

**Divides<T>(System.Divides<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Divides<T>(System.Divides<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Divides<T>&&	Argument to move from.

**operator=(System.Divides<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Divides<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Divides<T>&&	Argument to assign from.

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

Returns the first argument divided by the second argument.

**Syntax**

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

**Parameters**

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

**Returns**

T

Returns  $a/b$ .

## 2.2.15 Duration Class

Represents a duration in nanoseconds.

### Syntax

```
public class Duration;
```

#### 2.2.15.1 Member Functions

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

Rep() const	Returns total nanoseconds elapsed.
Seconds() const	Returns total seconds elapsed.

**Duration() Member Function**

Default constructor. Initializes the duration to zero nanoseconds.

**Syntax**

```
public Duration();
```

**Duration(const System.Duration&) Member Function**

Copy constructor.

**Syntax**

```
public Duration(const System.Duration& that);
```

**Parameters**

Name	Type	Description
that	const System.Duration&	Argument to copy.

**operator=(const System.Duration&)** Member Function

Copy assignment.

**Syntax**

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

**Parameters**

Name	Type	Description
that	const System.Duration&	Argument to assign.

**Duration(System.Duration&&) Member Function**

Move constructor.

**Syntax**

```
public Duration(System.Duration&& that);
```

**Parameters**

Name	Type	Description
that	System.Duration&&	Argument to move from.

**operator=(System.Duration&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Duration&& that);
```

**Parameters**

Name	Type	Description
that	System.Duration&&	Argument to assign from.

**Duration(long) Member Function**

Constructor. Initializes the duration to specified number of nanoseconds.

**Syntax**

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

**Parameters**

Name	Type	Description
nanosecs_	long	Nanoseconds.

**FromHours(long) Member Function**

Returns a duration of specified number of hours.

**Syntax**

```
public static System.Duration FromHours(long hours);
```

**Parameters**

Name	Type	Description
hours	long	Hours.

**Returns**

[System.Duration](#)

Returns a duration of specified number of hours.

**FromMicroseconds(long) Member Function**

Returns duration of specified number of microseconds.

**Syntax**

```
public static System.Duration FromMicroseconds(long microseconds);
```

**Parameters**

Name	Type	Description
microseconds	long	Microseconds.

**Returns**

[System.Duration](#)

Returns duration of specified number of microseconds.

**FromMilliseconds(long) Member Function**

Returns duration of specified number of milliseconds.

**Syntax**

```
public static System.Duration FromMilliseconds(long milliseconds);
```

**Parameters**

Name	Type	Description
milliseconds	long	Milliseconds.

**Returns**

[System.Duration](#)

Returns duration of specified number of milliseconds.

**FromMinutes(long) Member Function**

Returns duration of specified number of minutes.

**Syntax**

```
public static System.Duration FromMinutes(long minutes);
```

**Parameters**

Name	Type	Description
minutes	long	Minutes.

**Returns**

[System.Duration](#)

Returns duration of specified number of minutes.

**FromNanoseconds(long) Member Function**

Returns duration of specified number of nanoseconds.

**Syntax**

```
public static System.Duration FromNanoseconds(long nanoseconds);
```

**Parameters**

Name	Type	Description
nanoseconds	long	Nanoseconds.

**Returns**

[System.Duration](#)

Returns duration of specified number of nanoseconds.

**FromSeconds(long) Member Function**

Returns duration of specified number of seconds.

**Syntax**

```
public static System.Duration FromSeconds(long seconds);
```

**Parameters**

Name	Type	Description
seconds	long	Seconds.

**Returns**

[System.Duration](#)

Returns duration of specified number of seconds.

**Hours() const Member Function**

Returns total hours elapsed.

**Syntax**

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

**Returns**

long

Returns total hours elapsed.

**Microseconds() const Member Function**

Returns total microseconds elapsed.

**Syntax**

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

**Returns**

long

Returns total microseconds elapsed.

**Milliseconds() const Member Function**

Returns total milliseconds elapsed.

**Syntax**

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

**Returns**

long

Returns total milliseconds elapsed.

**Minutes() const Member Function**

Returns total minutes elapsed.

**Syntax**

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

**Returns**

long

Returns total minutes elapsed.

**Nanoseconds() const Member Function**

Returns total nanoseconds elapsed.

**Syntax**

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

**Returns**

long

Returns total nanoseconds elapsed.

**Rep() const Member Function**

Returns total nanoseconds elapsed.

**Syntax**

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

**Returns**

long

Returns total nanoseconds elapsed.

**Seconds() const Member Function**

Returns total seconds elapsed.

**Syntax**

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

**Returns**

long

Returns total seconds elapsed.

**2.2.15.2 Nonmember Functions**

<b>Function</b>	<b>Description</b>
<code>operator%(System.Duration, System.Duration)</code>	Computes the remainder of division of two durations.
<code>operator*(System.Duration, System.Duration)</code>	Computes the product of two durations.
<code>operator+(System.Duration, System.Duration)</code>	Computes the sum of two durations.
<code>operator+(System.Duration, System.TimePoint)</code>	Computes the sum of a duration and a time point and returns a time point.
<code>operator-(System.Duration, System.Duration)</code>	Subtracts a duration from a duration.
<code>operator/(System.Duration, System.Duration)</code>	Division of two durations.
<code>operator&lt;(System.Duration, System.Duration)</code>	Compares two durations for less than relationship.
<code>operator==(System.Duration, System.Duration)</code>	Compares two durations for equality.

**operator%(System.Duration, System.Duration) Function**

Computes the remainder of division of two durations.

**Syntax**

```
public System.Duration operator%(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

[System.Duration](#)

Returns *left*%*right*.

**operator\*(System.Duration, System.Duration) Function**

Computes the product of two durations.

**Syntax**

```
public System.Duration operator*(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

[System.Duration](#)

Returns  $left \times right$ .

**operator+(System.Duration, System.Duration) Function**

Computes the sum of two durations.

**Syntax**

```
public System.Duration operator+(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

[System.Duration](#)

Returns *left* + *right*.

**operator+(System.Duration, System.TimePoint) Function**

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

**Syntax**

```
public System.TimePoint operator+(System.Duration d, System.TimePoint tp);
```

**Parameters**

Name	Type	Description
d	<a href="#">System.Duration</a>	A duration.
tp	<a href="#">System.TimePoint</a>	A time point.

**Returns**

[System.TimePoint](#)

Returns  $d + tp$ .

**operator-(System.Duration, System.Duration) Function**

Subtracts a duration from a duration.

**Syntax**

```
public System.Duration operator-(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

[System.Duration](#)

Returns  $left - right$ .

**operator/(System.Duration, System.Duration) Function**

Division of two durations.

**Syntax**

```
public System.Duration operator/(System.Duration left, System.Duration right);
```

**Parameters**

Name	Type	Description
left	System.Duration	Dividend.
right	System.Duration	Divisor.

**Returns**

[System.Duration](#)

Returns *left* divided by *right*.

**operator<(System.Duration, System.Duration) Function**

Compares two durations for less than relationship.

**Syntax**

```
public bool operator<(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

bool

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

**operator==(System.Duration, System.Duration) Function**

Compares two durations for equality.

**Syntax**

```
public bool operator==(System.Duration left, System.Duration right);
```

**Parameters**

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

**Returns**

bool

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

## 2.2.16 EndLine Class

Represents an end of line character.

### Syntax

```
public class EndLine;
```

#### 2.2.16.1 Remarks

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

#### 2.2.16.2 Member Functions

Member Function	Description
<code>EndLine()</code>	Default constructor.
<code>EndLine(const System.EndLine&amp;)</code>	Copy constructor.
<code>EndLine(const System.EndLine&amp;)</code>	Copy constructor.

**EndLine() Member Function**

Default constructor.

**Syntax**

```
public EndLine();
```

**EndLine(const System.EndLine&)** Member Function

Copy constructor.

**Syntax**

```
public EndLine(const System.EndLine& that);
```

**Parameters**

Name	Type	Description
that	const System.EndLine&	

**EndLine(const System.EndLine&)** Member Function

Copy constructor.

**Syntax**

```
public EndLine(const System.EndLine& that);
```

**Parameters**

Name	Type	Description
that	const System.EndLine&	Argument to copy from.

### 2.2.17 EqualTo2<T, U> Class

An *equal to* binary predicate.

#### Syntax

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

#### Constraint

`EqualityComparable<T, U>`

#### Model of

`Relation<T, U, V>`

#### Base Class

`System.BinaryPred<T, U>`

#### 2.2.17.1 Remarks

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

#### 2.2.17.2 Member Functions

Member Function	Description
<code>EqualTo2&lt;T, U&gt;()</code>	Default constructor.
<code>EqualTo2&lt;T, U&gt;(const System.EqualTo2&lt;T, U&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.EqualTo2&lt;T, U&gt;&amp;)</code>	Copy assignment.
<code>EqualTo2&lt;T, U&gt;(System.EqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.EqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const U&amp;) const</code>	Returns true if the first argument is equal to the second argument, false otherwise.

**EqualTo2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public EqualTo2<T, U>();
```

**EqualTo2<T, U>(const System.EqualTo2<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public EqualTo2<T, U>(const System.EqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.EqualTo2<T, U>&	Argument to copy.

**operator=(const System.EqualTo2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.EqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.EqualTo2<T, U>&	Argument to assign.

**EqualTo2<T, U>(System.EqualTo2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public EqualTo2<T, U>(System.EqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.EqualTo2<T, U>&&	Argument to move from.

**operator=(System.EqualTo2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.EqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.EqualTo2<T, U>&&	Argument to assign from.

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

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

**Syntax**

```
public 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*.

## 2.2.18 EqualTo<T> Class

An *equal to* relation.

### Syntax

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

### Constraint

T is [Regular](#)

### Model of

[Relation<T>](#)

### Base Class

[System.Rel<T>](#)

## 2.2.18.1 Member Functions

Member Function	Description
<a href="#">EqualTo&lt;T&gt;()</a>	Default constructor.
<a href="#">EqualTo&lt;T&gt;(const System.EqualTo&lt;T&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.EqualTo&lt;T&gt;&amp;)</a>	Copy assignment.
<a href="#">EqualTo&lt;T&gt;(System.EqualTo&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.EqualTo&lt;T&gt;&amp;&amp;)</a>	Move assignment.
<a href="#">operator()(const T&amp;, const T&amp;) const</a>	Returns true if the first argument is equal to the second argument, false otherwise.

**EqualTo<T>() Member Function**

Default constructor.

**Syntax**

```
public EqualTo<T>();
```

**EqualTo<T>(const System.EqualTo<T>&) Member Function**

Copy constructor.

**Syntax**

```
public EqualTo<T>(const System.EqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.EqualTo<T>&	Argument to copy.

**operator=(const System.EqualTo<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.EqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.EqualTo<T>&	Argument to assign.

**EqualTo<T>(System.EqualTo<T>&&) Member Function**

Move constructor.

**Syntax**

```
public EqualTo<T>(System.EqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.EqualTo<T>&&	Argument to move from.

**operator=(System.EqualTo<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.EqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.EqualTo<T>&&	Argument to assign from.

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

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

**Syntax**

```
public 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*.

## 2.2.19 Exception Class

A base class for all exception classes.

### Syntax

```
public class Exception;
```

#### 2.2.19.1 Remarks

All possible Cmajor exceptions can be caught by catching **Exception** class.

#### 2.2.19.2 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();
    }
}
```

#### 2.2.19.3 Member Functions

Member Function	Description
<a href="#">Exception()</a>	Default constructor.
<a href="#">Exception(const System.Exception&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Exception&amp;)</a>	Copy assignment.
<a href="#">Exception(System.Exception&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Exception&amp;&amp;)</a>	Move assignment.
<a href="#">Exception(const System.String&amp;)</a>	Constructor. Constructs an exception with an error message.

<code>ExceptionType() const</code>	Returns the dynamic class name of the exception.
<code>File() const</code>	Returns the path to the source file where the exception was thrown.
<code>Line() const</code>	Returns the source line number where the exception was thrown.
<code>Message() const</code>	Returns the error message.
<code>SetCallStack(const System.String&amp;)</code>	Sets the call stack string.
<code>SetExceptionType(const System.String&amp;)</code>	Sets the dynamic class name.
<code>SetFile(const System.String&amp;)</code>	Sets the source file.
<code>SetLine(int)</code>	Sets the source line number.
<code>ToString() const</code>	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.
<code>~Exception()</code>	Destructor.

**Exception() Member Function**

Default constructor.

**Syntax**

```
public Exception();
```

**Exception(const System.Exception&)** Member Function

Copy constructor.

**Syntax**

```
public Exception(const System.Exception& that);
```

**Parameters**

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

**operator=(const System.Exception&)** Member Function

Copy assignment.

**Syntax**

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

**Parameters**

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

**Exception(System.Exception&&) Member Function**

Move constructor.

**Syntax**

```
public Exception(System.Exception&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.Exception&&	An exception to move from.

**operator=(System.Exception&&)** Member Function

Move assignment.

**Syntax**

```
public void operator=(System.Exception&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.Exception&&	An exception to move from.

**Exception(const System.String&) Member Function**

Constructor. Constructs an exception with an error message.

**Syntax**

```
public Exception(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message.

**ExceptionType() const Member Function**

Returns the dynamic class name of the exception.

**Syntax**

```
public const System.String& ExceptionType() const;
```

**Returns**

`const System.String&`

Returns the dynamic class name of the exception.

**File() const Member Function**

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

**Syntax**

```
public const System.String& File() const;
```

**Returns**

`const System.String&`

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

**Line() const Member Function**

Returns the source line number where the exception was thrown.

**Syntax**

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

**Returns**

int

Returns the source line number where the exception was thrown.

**Message() const Member Function**

Returns the error message.

**Syntax**

```
public const System.String& Message() const;
```

**Returns**

`const System.String&`

Returns the error message.

**SetCallStack(const System.String&)** Member Function

Sets the call stack string.

**Syntax**

```
public void SetCallStack(const System.String& callStack_);
```

**Parameters**

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

**SetExceptionType(const System.String&)** Member Function

Sets the dynamic class name.

**Syntax**

```
public void SetExceptionType(const System.String& exceptionType_);
```

**Parameters**

Name	Type	Description
exceptionType_	const System.String&	A dynamic class name.

**SetFile(const System.String&)** Member Function

Sets the source file.

**Syntax**

```
public void SetFile(const System.String& file_);
```

**Parameters**

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

**SetLine(int) Member Function**

Sets the source line number.

**Syntax**

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

**Parameters**

Name	Type	Description
line_	int	A source line number.

**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 System.String ToString() const;
```

**Returns**

[System.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.

**~Exception() Member Function**

Destructor.

**Syntax**

```
public ~Exception();
```

## 2.2.20 ExceptionPtr Class

The **ExceptionPtr** class can be used to hold an exception with full type information.

### Syntax

```
public class ExceptionPtr;
```

#### 2.2.20.1 Remarks

The **ExceptionPtr** is used with functions [CaptureCurrentException\(\)](#) and [RethrowException\(System.ExceptionPtr&\)](#).

#### 2.2.20.2 Example

```
using System;

// This program writes the following message to the standard error stream:
// FooException at 'C:/Temp/exptr/System.ExceptionPtr.cm' line 32:
// exception from thread
// call stack:
// 1> function 'foo()' file C:/Temp/exptr/System.ExceptionPtr.cm line 32
// 0> function 'ThreadFunction(void*)' file C:/Temp/exptr/System.
//    ExceptionPtr.cm line 40

public class FooException : Exception
{
    public FooException(const string& message_) : base(message_)
    {
    }
}

public class ThreadData
{
    public void SetException(ExceptionPtr&& exPtr_)
    {
        exPtr = exPtr_;
    }
    public ExceptionPtr GetException() const
    {
        return Rvalue(exPtr);
    }
    private ExceptionPtr exPtr;
}

void foo()
{
    throw FooException("exception from thread");
}

void ThreadFunction(void* data)
{
    ThreadData* threadData = cast<ThreadData*>(data);
    try
    {
        foo();
    }
}
```

```

catch (const Exception& ex)
{
    ExceptionPtr exPtr = CaptureCurrentException();
    threadData->SetException(Rvalue(exPtr));
}
}

void main()
{
    try
    {
        ThreadData threadData;
        System.Threading.Thread thread(System.Threading.ThreadFun(
            ThreadFunction), &threadData);
        thread.Join();
        ExceptionPtr exPtr = threadData.GetException();
        if (exPtr.HasException())
        {
            RethrowException(exPtr);
        }
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.ToString() << endl();
    }
}
}

```

### 2.2.20.3 Member Functions

Member Function	Description
ExceptionPtr()	Default constructor. Initializes the <b>ExceptionPtr</b> empty.
ExceptionPtr(System.ExceptionPtr&&)	Move constructor.
operator=(System.ExceptionPtr&&)	Move assignment.
ExceptionId() const	Returns identifier of the captured exception, or 0 if the <b>ExceptionPtr</b> does not hold an exception.
ExceptionPtr(int, System.Exception*)	Constructor. Initializes the <b>ExceptionPtr</b> to hold an exception with the given identifier and exception.
HasException() const	Returns true if the <b>ExceptionPtr</b> holds an exception, false otherwise.
Release() const	Releases the captured exception and sets the <b>ExceptionPtr</b> empty.

[`~ExceptionPtr\(\)`](#)

Destroys the exception if the **ExceptionPtr** has one.

**ExceptionPtr() Member Function**

Default constructor. Initializes the [ExceptionPtr](#) empty.

**Syntax**

```
public ExceptionPtr();
```

**ExceptionPtr(System.ExceptionPtr&&) Member Function**

Move constructor.

**Syntax**

```
public ExceptionPtr(System.ExceptionPtr&& that);
```

**Parameters**

Name	Type	Description
that	System.ExceptionPtr&&	A <a href="#">ExceptionPtr</a> to move from.

**operator=(System.ExceptionPtr&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.ExceptionPtr&& that);
```

**Parameters**

Name	Type	Description
that	System.ExceptionPtr&&	A <a href="#">ExceptionPtr</a> to assign.

**ExceptionId() const Member Function**

Returns identifier of the captured exception, or 0 if the [ExceptionPtr](#) does not hold an exception.

**Syntax**

```
public int ExceptionId() const;
```

**Returns**

int

Returns identifier of the captured exception, or 0 if the [ExceptionPtr](#) does not hold an exception.

**ExceptionPtr(int, System.Exception\*) Member Function**

Constructor. Initializes the [ExceptionPtr](#) to hold an exception with the given identifier and exception.

**Syntax**

```
public ExceptionPtr(int exceptionId_, System.Exception* exception_);
```

**Parameters**

Name	Type	Description
exceptionId_	int	Identifier of the captured exception.
exception_	<a href="#">System.Exception*</a>	Captured exception.

**HasException() const Member Function**

Returns true if the [ExceptionPtr](#) holds an exception, false otherwise.

**Syntax**

```
public bool HasException() const;
```

**Returns**

bool

Returns true if the [ExceptionPtr](#) holds an exception, false otherwise.

**Release() const Member Function**

Releases the captured exception and sets the [ExceptionPtr](#) empty.

**Syntax**

```
public System.Exception* Release() const;
```

**Returns**

[System.Exception\\*](#)

Returns captured exception.

**~ExceptionPtr() Member Function**

Destroys the exception if the [ExceptionPtr](#) has one.

**Syntax**

```
public ~ExceptionPtr();
```

## 2.2.21 FrontInsertIterator<C> Class

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

### Syntax

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

### Constraint

C is [FrontInsertionSequence](#)

### Model of

[OutputIterator<T>](#)

#### 2.2.21.1 Remarks

System.FrontInserter.C.C.is.FrontInsertionSequence.C.ref is a helper function that returns a **FrontInsertIterator<C>** for a container.

#### 2.2.21.2 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();
}
```

### 2.2.21.3 Type Definitions

Name	Type	Description
PointerType	<code>System.FrontInsertProxy&lt;C&gt;*</code>	Pointer to implementation defined proxy type.
ReferenceType	<code>System.FrontInsertProxy&lt;C&gt;&amp;</code>	Reference to implementation defined proxy type.
ValueType	<code>System.FrontInsertProxy&lt;C&gt;</code>	Implementation defined proxy type.

### 2.2.21.4 Member Functions

Member Function	Description	
<code>FrontInsertIterator&lt;C&gt;()</code>	Constructor.	Default constructs a front insert iterator.
<code>FrontInsertIterator&lt;C&gt;(const FrontInsertIterator&lt;C&gt;&amp;)</code>	System.-	Copy constructor.
<code>operator=(const FrontInsertIterator&lt;C&gt;&amp;)</code>	System.FrontInsertIterator<->	Copy assignment.
<code>FrontInsertIterator&lt;C&gt;(System.FrontInsertIterator&lt;C&gt;&amp;&amp;)</code>	-	Move constructor.
<code>operator=(System.FrontInsertIterator&lt;C&gt;&amp;&amp;)</code>	&&	Move assignment.
<code>FrontInsertIterator&lt;C&gt;(C&amp;)</code>		Copy constructor.
<code>operator*()</code>		Returns a reference to a proxy object that inserts values at the beginning of a container.
<code>operator++()</code>		Advances the iterator to the next element.
<code>operator-&gt;()</code>		Returns a pointer to a proxy object that inserts values at the beginning of a container.

**FrontInsertIterator<C>() Member Function**

Constructor. Default constructs a front insert iterator.

**Syntax**

```
public FrontInsertIterator<C>();
```

**FrontInsertIterator<C>(const System.FrontInsertIterator<C>&) Member Function**

Copy constructor.

**Syntax**

```
public FrontInsertIterator<C>(const System.FrontInsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.FrontInsertIterator<C>&	Argument to copy.

**operator=(const System.FrontInsertIterator<C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.FrontInsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.FrontInsertIterator<C>&	Argument to assign.

**FrontInsertIterator<C>(System.FrontInsertIterator<C>&&) Member Function**

Move constructor.

**Syntax**

```
public FrontInsertIterator<C>(System.FrontInsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.FrontInsertIterator<C>&&	Argument to move from.

**operator=(System.FrontInsertIterator<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.FrontInsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.FrontInsertIterator<C>&&	Argument to assign from.

**FrontInsertIterator<C>(C&) Member Function**

Copy constructor.

**Syntax**

```
public FrontInsertIterator<C>(C& c);
```

**Parameters**

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

**operator\*() Member Function**

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

**Syntax**

```
public System.FrontInsertProxy<C>& operator*();
```

**Returns**

[System.FrontInsertProxy<C>&](#)

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

**operator++() Member Function**

Advances the iterator to the next element.

**Syntax**

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

**Returns**

[System.FrontInsertIterator<C>&](#)

Returns a reference to the iterator.

**operator->() Member Function**

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

**Syntax**

```
public System.FrontInsertProxy<C>* operator->();
```

**Returns**

[System.FrontInsertProxy<C>\\*](#)

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

## 2.2.22 FrontInsertProxy<C> Class

Implementation detail.

### Syntax

```
public class FrontInsertProxy<C>;
```

### Constraint

C is [FrontInsertionSequence](#)

#### 2.2.22.1 Member Functions

Member Function	Description
<a href="#">FrontInsertProxy&lt;C&gt;(System.-FrontInsertProxy&lt;C&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.FrontInsertProxy&lt;C&gt;&amp;&amp;)</a>	Move assignment.

**FrontInsertProxy<C>(System.FrontInsertProxy<C>&&) Member Function**

Move constructor.

**Syntax**

```
public FrontInsertProxy<C>(System.FrontInsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.FrontInsertProxy<C>&&	Argument to move from.

**operator=(System.FrontInsertProxy<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.FrontInsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.FrontInsertProxy<C>&&	Argument to assign from.

## 2.2.23 Greater2<T, U> Class

A *greater than* binary predicate.

### Syntax

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

### Constraint

`LessThanComparable<T, U>`

### Model of

`Relation<T, U, V>`

### Base Class

`System.BinaryPred<T, U>`

#### 2.2.23.1 Remarks

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

#### 2.2.23.2 Member Functions

Member Function	Description
<code>Greater2&lt;T, U&gt;()</code>	Default constructor.
<code>Greater2&lt;T, U&gt;(const System.Greater2&lt;T, U&gt; &amp;)</code>	Copy constructor.
<code>operator=(const System.Greater2&lt;T, U&gt;&amp;)</code>	Copy assignment.
<code>Greater2&lt;T, U&gt;(System.Greater2&lt;T, U&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Greater2&lt;T, U&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const U&amp;) const</code>	Returns true if the first argument is greater than the second argument, false otherwise.

**Greater2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public Greater2<T, U>();
```

**Greater2<T, U>(const System.Greater2<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public Greater2<T, U>(const System.Greater2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Greater2<T, U>&	Argument to copy.

**operator=(const System.Greater2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Greater2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Greater2<T, U>&	Argument to assign.

**Greater2<T, U>(System.Greater2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public Greater2<T, U>(System.Greater2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Greater2&lt;T, U&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.Greater2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Greater2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.Greater2<T, U>&&	Argument to assign from.

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

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

**Syntax**

```
public 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*.

## 2.2.24 Greater<T> Class

A *greater than* relation.

### Syntax

```
public class Greater<T>;
```

### Constraint

T is [LessThanComparable](#)

### Model of

[Relation<T>](#)

### Base Class

[System.Rel<T>](#)

#### 2.2.24.1 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();
}
```

#### 2.2.24.2 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();
}

```

### 2.2.24.3 Member Functions

Member Function	Description
<a href="#">Greater&lt;T&gt;()</a>	Default constructor.
<a href="#">Greater&lt;T&gt;(const System.Greater&lt;T&gt;&amp;)</a>	Copy constructor.

operator=(const System.Greater<T>&)	Copy assignment.
Greater<T>(System.Greater<T>&&)	Move constructor.
operator=(System.Greater<T>&&)	Move assignment.
operator()(const T&, const T&) const	Returns true if the first argument is greater than the second argument, false otherwise.

**Greater<T>() Member Function**

Default constructor.

**Syntax**

```
public Greater<T>();
```

**Greater<T>(const System.Greater<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Greater<T>(const System.Greater<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Greater<T>&	Argument to copy.

**operator=(const System.Greater<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Greater<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Greater<T>&	Argument to assign.

**Greater<T>(System.Greater<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Greater<T>(System.Greater<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Greater<T>&&	Argument to move from.

**operator=(System.Greater<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Greater<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Greater<T>&&	Argument to assign from.

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

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

**Syntax**

```
public 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*.

## 2.2.25 GreaterOrEqualTo2<T, U> Class

A *greater than or equal to* binary predicate.

### Syntax

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

### Constraint

`LessThanComparable<T, U>`

### Model of

`Relation<T, U, V>`

### Base Class

`System.BinaryPred<T, U>`

#### 2.2.25.1 Remarks

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

#### 2.2.25.2 Member Functions

Member Function	Description
<code>GreaterOrEqualTo2&lt;T, U&gt;()</code>	Default constructor.
<code>GreaterOrEqualTo2&lt;T, U&gt;(const System.-&gt;GreaterOrEqualTo2&lt;T, U&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.GreaterOrEqualTo2&lt;T, U&gt;&amp;)</code>	Copy assignment.
<code>GreaterOrEqualTo2&lt;T, U&gt;(System.-&gt;GreaterOrEqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.GreaterOrEqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const U&amp;) const</code>	Returns true if the first argument is greater than or equal to the second argument, false otherwise.

**GreaterOrEqualTo2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public GreaterOrEqualTo2<T, U>();
```

`GreaterOrEqualTo2<T, U>(const System.GreaterOrEqualTo2<T, U>&)` Member Function

Copy constructor.

#### Syntax

```
public GreaterOrEqualTo2<T, U>(const System.GreaterOrEqualTo2<T, U>& that);
```

#### Parameters

Name	Type	Description
that	const System.GreaterOrEqualTo2<T, U>&	Argument to copy.

**operator=(const System.GreaterOrEqualTo2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.GreaterOrEqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.GreaterOrEqualTo2<T, U>&	Argument to assign.

**GreaterOrEqualTo2<T, U>(System.GreaterOrEqualTo2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public GreaterOrEqualTo2<T, U>(System.GreaterOrEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.GreaterOrEqualTo2<T, U>&&	Argument to move from.

**operator=(System.GreaterOrEqualTo2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.GreaterOrEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.GreaterOrEqualTo2<T, U>&&	Argument to assign from.

**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 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  $left \geq right$ .

## 2.2.26 GreaterOrEqualTo<T> Class

A *greater than or equal to* relation.

### Syntax

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

### Constraint

T is [LessThanComparable](#)

### Model of

[Relation<T>](#)

### Base Class

[System.Rel<T>](#)

### 2.2.26.1 Member Functions

Member Function	Description
<code>GreaterOrEqualTo&lt;T&gt;()</code>	Default constructor.
<code>GreaterOrEqualTo&lt;T&gt;(const System.- GreaterOrEqualTo&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.GreaterOrEqualTo&lt;T&gt;&amp;)</code>	Copy assignment.
<code>GreaterOrEqualTo&lt;T&gt;(System.- GreaterOrEqualTo&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.GreaterOrEqualTo&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns true if the first argument is greater than or equal to the second argument, false otherwise.

**GreaterOrEqualTo<T>() Member Function**

Default constructor.

**Syntax**

```
public GreaterOrEqualTo<T>();
```

**GreaterOrEqualTo<T>(const System.GreaterOrEqualTo<T>&) Member Function**

Copy constructor.

**Syntax**

```
public GreaterOrEqualTo<T>(const System.GreaterOrEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.GreaterOrEqualTo<T>&	Argument to copy.

**operator=(const System.GreaterOrEqualTo<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.GreaterOrEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.GreaterOrEqualTo<T>&	Argument to assign.

**GreaterOrEqualTo<T>(System.GreaterOrEqualTo<T>&&) Member Function**

Move constructor.

**Syntax**

```
public GreaterOrEqualTo<T>(System.GreaterOrEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.GreaterOrEqualTo&lt;T&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.GreaterOrEqualTo<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.GreaterOrEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.GreaterOrEqualTo<T>&&	Argument to assign from.

**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 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  $left \geq right$ .

### 2.2.27 Identity<T> Class

An identity unary function object.

#### Syntax

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

#### Constraint

T is [Semiregular](#)

#### Model of

[UnaryOperation<T>](#)

#### Base Class

[System.UnaryFun<T, T>](#)

#### 2.2.27.1 Member Functions

Member Function	Description
<code>Identity&lt;T&gt;()</code>	Default constructor.
<code>Identity&lt;T&gt;(const System.Identity&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Identity&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Identity&lt;T&gt;(System.Identity&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Identity&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;) const</code>	Returns the argument.

**Identity<T>() Member Function**

Default constructor.

**Syntax**

```
public Identity<T>();
```

**Identity<T>(const System.Identity<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Identity<T>(const System.Identity<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Identity<T>&	Argument to copy.

**operator=(const System.Identity<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Identity<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Identity<T>&	Argument to assign.

**Identity<T>(System.Identity<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Identity<T>(System.Identity<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Identity<T>&&	Argument to move from.

**operator=(System.Identity<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Identity<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Identity<T>&&	Argument to assign from.

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

Returns the argument.

**Syntax**

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

**Parameters**

Name	Type	Description
x	const T&	An argument.

**Returns**

const T&

Returns *x*.

## 2.2.28 InsertIterator<C> Class

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

### Syntax

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

### Constraint

C is [InsertionSequence](#)

### Model of

[OutputIterator<T>](#)

#### 2.2.28.1 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.

#### 2.2.28.2 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();
}
```

### 2.2.28.3 Type Definitions

Name	Type	Description
PointerType	<code>System.InsertProxy&lt;C&gt;*</code>	Pointer to an implementation defined proxy type.
ReferenceType	<code>System.InsertProxy&lt;C&gt;&amp;</code>	Reference to an implementation defined proxy type.
ValueType	<code>System.InsertProxy&lt;C&gt;</code>	An implementation defined proxy type.

### 2.2.28.4 Member Functions

Member Function	Description
<code>InsertIterator&lt;C&gt;()</code>	Constructor. Default constructs an insert iterator.
<code>InsertIterator&lt;C&gt;(const System.InsertIterator&lt;C&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.InsertIterator&lt;C&gt;&amp;)</code>	Copy assignment.
<code>InsertIterator&lt;C&gt;(System.InsertIterator&lt;C&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.InsertIterator&lt;C&gt;&amp;&amp;)</code>	Move assignment.
<code>operator*()</code>	Returns a reference to a proxy type that inserts values to the container.
<code>operator++()</code>	Advances the insert iterator to the next element.
<code>operator-&gt;()</code>	Returns a pointer to a proxy type that inserts values to the container.

**InsertIterator<C>() Member Function**

Constructor. Default constructs an insert iterator.

**Syntax**

```
public InsertIterator<C>();
```

**InsertIterator<C>(const System.InsertIterator<C>&) Member Function**

Copy constructor.

**Syntax**

```
public InsertIterator<C>(const System.InsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.InsertIterator<C>&	Argument to copy.

**operator=(const System.InsertIterator<C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.InsertIterator<C>& that);
```

**Parameters**

Name	Type	Description
that	const System.InsertIterator<C>&	Argument to assign.

**InsertIterator<C>(System.InsertIterator<C>&&) Member Function**

Move constructor.

**Syntax**

```
public InsertIterator<C>(System.InsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.InsertIterator&lt;C&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.InsertIterator<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.InsertIterator<C>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.InsertIterator&lt;C&gt;&amp;&amp;</a>	Argument to assign from.

**operator\*() Member Function**

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

**Syntax**

```
public System.InsertProxy<C>& operator*();
```

**Returns**

[System.InsertProxy<C>&](#)

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

**operator++() Member Function**

Advances the insert iterator to the next element.

**Syntax**

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

**Returns**

[System.InsertIterator<C>&](#)

Returns a reference to the iterator.

**operator->() Member Function**

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

**Syntax**

```
public System.InsertProxy<C>* operator->();
```

**Returns**

[System.InsertProxy<C>\\*](#)

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

## 2.2.29 InsertProxy<C> Class

Implementation detail.

### Syntax

```
public class InsertProxy<C>;
```

### Constraint

C is [InsertionSequence](#)

#### 2.2.29.1 Member Functions

Member Function	Description
<code>InsertProxy&lt;C&gt;(System.InsertProxy&lt;C&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.InsertProxy&lt;C&gt;&amp;&amp;)</code>	Move assignment.

**InsertProxy<C>(System.InsertProxy<C>&&) Member Function**

Move constructor.

**Syntax**

```
public InsertProxy<C>(System.InsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.InsertProxy<C>&&	Argument to move from.

**operator=(System.InsertProxy<C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.InsertProxy<C>&& that);
```

**Parameters**

Name	Type	Description
that	System.InsertProxy<C>&&	Argument to assign from.

## 2.2.30 Less2<T, U> Class

A *less than* binary predicate.

### Syntax

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

### Constraint

[LessThanComparable<T, U>](#)

### Model of

[Relation<T, U, V>](#)

### Base Class

[System.BinaryPred<T, U>](#)

#### 2.2.30.1 Remarks

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

#### 2.2.30.2 Member Functions

Member Function	Description
<a href="#">Less2&lt;T, U&gt;()</a>	Default constructor.
<a href="#">Less2&lt;T, U&gt;(const System.Less2&lt;T, U&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Less2&lt;T, U&gt;&amp;)</a>	Copy assignment.
<a href="#">Less2&lt;T, U&gt;(System.Less2&lt;T, U&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Less2&lt;T, U&gt;&amp;&amp;)</a>	Move assignment.
<a href="#">operator()(const T&amp;, const U&amp;) const</a>	Returns true if the first argument is less than the second argument.

**Less2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public Less2<T, U>();
```

**Less2<T, U>(const System.Less2<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public Less2<T, U>(const System.Less2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Less2<T, U>&	Argument to copy.

**operator=(const System.Less2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Less2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Less2<T, U>&	Argument to assign.

**Less2<T, U>(System.Less2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public Less2<T, U>(System.Less2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.Less2<T, U>&&	Argument to move from.

**operator=(System.Less2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Less2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.Less2<T, U>&&	Argument to assign from.

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

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

**Syntax**

```
public 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  $left < right$ .

### 2.2.31 Less<T> Class

A *less than* relation.

#### Syntax

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

#### Constraint

T is [LessThanComparable](#)

#### Model of

[Relation<T>](#)

#### Base Class

[System.Rel<T>](#)

#### 2.2.31.1 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();
}
```

#### 2.2.31.2 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 noexcept inline const string& Id() const
    {
        return id;
    }
    private string id;
}

public noexcept 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();
}
}

```

### 2.2.31.3 Member Functions

Member Function	Description
Less<T>()	Default constructor.
Less<T>(const System.Less<T>&)	Copy constructor.
operator=(const System.Less<T>&)	Copy assignment.
Less<T>(System.Less<T>&&)	Move constructor.

<code>operator=(System.Less&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns true if the first argument is less than the second argument, false otherwise.

**Less<T>() Member Function**

Default constructor.

**Syntax**

```
public Less<T>();
```

**Less<T>(const System.Less<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Less<T>(const System.Less<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Less<T>&	Argument to copy.

**operator=(const System.Less<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Less<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Less<T>&	Argument to assign.

**Less<T>(System.Less<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Less<T>(System.Less<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Less<T>&&	Argument to move from.

**operator=(System.Less<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Less<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Less<T>&&	Argument to assign from.

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

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

**Syntax**

```
public 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  $left < right$ .

## 2.2.32 LessOrEqualTo2<T, U> Class

A *less than or equal to* binary predicate.

### Syntax

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

### Constraint

`LessThanComparable<T, U>`

### Model of

`Relation<T, U, V>`

### Base Class

`System.BinaryPred<T, U>`

#### 2.2.32.1 Remarks

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

#### 2.2.32.2 Member Functions

Member Function	Description
<code>LessOrEqualTo2&lt;T, U&gt;()</code>	Default constructor.
<code>LessOrEqualTo2&lt;T, U&gt;(const System.-&gt;LessOrEqualTo2&lt;T, U&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.LessOrEqualTo2&lt;T, U&gt;&amp;)</code>	Copy assignment.
<code>LessOrEqualTo2&lt;T, U&gt;(System.-&gt;LessOrEqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.LessOrEqualTo2&lt;T, U&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const U&amp;) const</code>	Returns true if the first argument is less than or equal to the second argument, false otherwise.

**LessOrEqualTo2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public LessOrEqualTo2<T, U>();
```

**LessOrEqualTo2<T, U>(const System.LessOrEqualTo2<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public LessOrEqualTo2<T, U>(const System.LessOrEqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.LessOrEqualTo2<T, U>&	Argument to copy.

**operator=(const System.LessOrEqualTo2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.LessOrEqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.LessOrEqualTo2<T, U>&	Argument to assign.

**LessOrEqualTo2<T, U>(System.LessOrEqualTo2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public LessOrEqualTo2<T, U>(System.LessOrEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.LessOrEqualTo2<T, U>&&	Argument to move from.

**operator=(System.LessOrEqualTo2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.LessOrEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.LessOrEqualTo2<T, U>&&	Argument to assign from.

**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 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*  $\leq$  *right*.

### 2.2.33 LessOrEqualTo<T> Class

A *less than or equal to* relation.

#### Syntax

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

#### Constraint

T is [LessThanComparable](#)

#### Model of

[Relation<T>](#)

#### Base Class

[System.Rel<T>](#)

#### 2.2.33.1 Member Functions

Member Function	Description
<code>LessOrEqualTo&lt;T&gt;()</code>	Default constructor.
<code>LessOrEqualTo&lt;T&gt;(const System.-&gt;LessOrEqualTo&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.LessOrEqualTo&lt;T&gt;&amp;)</code>	Copy assignment.
<code>LessOrEqualTo&lt;T&gt;(System.LessOrEqualTo&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.LessOrEqualTo&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns true if the first argument is less than or equal to the second argument, false otherwise.

**LessOrEqualTo<T>() Member Function**

Default constructor.

**Syntax**

```
public LessOrEqualTo<T>();
```

**LessOrEqualTo<T>(const System.LessOrEqualTo<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LessOrEqualTo<T>(const System.LessOrEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LessOrEqualTo<T>&	Argument to copy.

**operator=(const System.LessOrEqualTo<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.LessOrEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LessOrEqualTo<T>&	Argument to assign.

**LessOrEqualTo<T>(System.LessOrEqualTo<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LessOrEqualTo<T>(System.LessOrEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LessOrEqualTo<T>&&	Argument to move from.

**operator=(System.LessOrEqualTo<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.LessOrEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LessOrEqualTo<T>&&	Argument to assign from.

**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 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*  $\leq$  *right*.

## 2.2.34 LogicalAnd<T> Class

A *logical AND* binary predicate.

### Syntax

```
public class LogicalAnd<T>;
```

### Default Template Arguments

T = bool

### Base Class

[System.BinaryPred<bool, bool>](#)

### 2.2.34.1 Member Functions

Member Function	Description
<code>LogicalAnd&lt;T&gt;()</code>	Default constructor.
<code>LogicalAnd&lt;T&gt;(const System.LogicalAnd&lt;T&gt;&amp; &amp;z)</code>	Copy constructor.
<code>operator=(const System.LogicalAnd&lt;T&gt;&amp;)</code>	Copy assignment.
<code>LogicalAnd&lt;T&gt;(System.LogicalAnd&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.LogicalAnd&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(bool, bool) const</code>	Returns true if both <code>left</code> and <code>right</code> arguments are true, false otherwise.

**LogicalAnd<T>() Member Function**

Default constructor.

**Syntax**

```
public LogicalAnd<T>();
```

**LogicalAnd<T>(const System.LogicalAnd<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LogicalAnd<T>(const System.LogicalAnd<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalAnd<T>&	Argument to copy.

**operator=(const System.LogicalAnd<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.LogicalAnd<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalAnd<T>&	Argument to assign.

**LogicalAnd<T>(System.LogicalAnd<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LogicalAnd<T>(System.LogicalAnd<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalAnd<T>&&	Argument to move from.

**operator=(System.LogicalAnd<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.LogicalAnd<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalAnd<T>&&	Argument to assign from.

**operator()(bool, bool) const Member Function**

Returns true if both `left` and `right` arguments are true, false otherwise.

**Syntax**

```
public bool operator()(bool left, bool right) const;
```

**Parameters**

Name	Type	Description
left	bool	Left argument.
right	bool	Right argument.

**Returns**

bool

Returns true if both `left` and `right` arguments are true, false otherwise.

## 2.2.35 LogicalNot<T> Class

A *logical NOT* unary predicate.

### Syntax

```
public class LogicalNot<T>;
```

### Default Template Arguments

T = bool

### Base Class

[System.UnaryPred<bool>](#)

### 2.2.35.1 Member Functions

Member Function	Description
<code>LogicalNot&lt;T&gt;()</code>	Default constructor.
<code>LogicalNot&lt;T&gt;(const System.LogicalNot&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.LogicalNot&lt;T&gt;&amp;)</code>	Copy assignment.
<code>LogicalNot&lt;T&gt;(System.LogicalNot&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.LogicalNot&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(bool) const</code>	Returns true if <code>operand</code> is false, false otherwise.

**LogicalNot<T>() Member Function**

Default constructor.

**Syntax**

```
public LogicalNot<T>();
```

**LogicalNot<T>(const System.LogicalNot<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LogicalNot<T>(const System.LogicalNot<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalNot<T>&	Argument to copy.

**operator=(const System.LogicalNot<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.LogicalNot<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalNot<T>&	Argument to assign.

**LogicalNot<T>(System.LogicalNot<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LogicalNot<T>(System.LogicalNot<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalNot<T>&&	Argument to move from.

**operator=(System.LogicalNot<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.LogicalNot<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalNot<T>&&	Argument to assign from.

**operator()(bool) const Member Function**

Returns true if [operand](#) is false, false otherwise.

**Syntax**

```
public bool operator()(bool operand) const;
```

**Parameters**

Name	Type	Description
operand	bool	Operand.

**Returns**

bool

Returns true if [operand](#) is false, false otherwise.

## 2.2.36 LogicalOr<T> Class

A *logical OR* binary predicate.

### Syntax

```
public class LogicalOr<T>;
```

### Default Template Arguments

T = bool

### Base Class

[System.BinaryPred<bool, bool>](#)

### 2.2.36.1 Member Functions

Member Function	Description
<code>LogicalOr&lt;T&gt;()</code>	Default constructor.
<code>LogicalOr&lt;T&gt;(const System.LogicalOr&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.LogicalOr&lt;T&gt;&amp;)</code>	Copy assignment.
<code>LogicalOr&lt;T&gt;(System.LogicalOr&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.LogicalOr&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(bool, bool) const</code>	Returns true if <code>left</code> is true, or <code>right</code> is true, or both are true, false otherwise.

**LogicalOr<T>() Member Function**

Default constructor.

**Syntax**

```
public LogicalOr<T>();
```

**LogicalOr<T>(const System.LogicalOr<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LogicalOr<T>(const System.LogicalOr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalOr<T>&	Argument to copy.

**operator=(const System.LogicalOr<T>&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.LogicalOr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.LogicalOr<T>&	Argument to assign.

**LogicalOr<T>(System.LogicalOr<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LogicalOr<T>(System.LogicalOr<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalOr<T>&&	Argument to move from.

**operator=(System.LogicalOr<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.LogicalOr<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.LogicalOr<T>&&	Argument to assign from.

**operator()(bool, bool) const Member Function**

Returns true if `left` is true, or `right` is true, or both are true, false otherwise.

**Syntax**

```
public bool operator()(bool left, bool right) const;
```

**Parameters**

Name	Type	Description
left	bool	Left argument.
right	bool	Right argument.

**Returns**

bool

Returns true if `left` is true, or `right` is true, or both are true, false otherwise.

## 2.2.37 MT Class

Mersenne Twister pseudorandom number generator.

### Syntax

```
public static class MT;
```

#### 2.2.37.1 Remarks

[Rand\(\)](#) function returns a pseudorandom number in range 0..[MaxValue\(uint\)](#) inclusive.

#### 2.2.37.2 Member Functions

Member Function	Description
<a href="#">GenRand()</a>	Generates a pseudorandom number in range 0.. <a href="#">MaxValue(uint)</a> and returns it.
<a href="#">Init(uint)</a>	Initializes the pseudorandom number generator with the given seed.
<a href="#">InitWithRandomSeed()</a>	Initializes the pseudorandom number generator with a random seed returned by the system.

**GenRand() Member Function**

Generates a pseudorandom number in range 0..[MaxValue\(uint\)](#) and returns it.

**Syntax**

```
public static uint GenRand();
```

**Returns**

uint

Generates a pseudorandom number in range 0..[MaxValue\(uint\)](#) and returns it.

**Init(uint) Member Function**

Initializes the pseudorandom number generator with the given seed.

**Syntax**

```
public static void Init(uint seed);
```

**Parameters**

Name	Type	Description
seed	uint	A seed value.

**Remarks**

In general the **Init(uint)** function need not be called because the static constructor initializes the generator with random seed returned by the system.

**InitWithRandomSeed() Member Function**

Initializes the pseudorandom number generator with a random seed returned by the system.

**Syntax**

```
public static void InitWithRandomSeed();
```

### 2.2.38 Minus<T> Class

A subtraction binary function object.

#### Syntax

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

#### Constraint

T is [AdditiveGroup](#)

#### Model of

[BinaryFunction<T>](#)

#### Base Class

[System.BinaryFun<T, T, T>](#)

### 2.2.38.1 Member Functions

Member Function	Description
<code>Minus&lt;T&gt;()</code>	Default constructor.
<code>Minus&lt;T&gt;(const System.Minus&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Minus&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Minus&lt;T&gt;(System.Minus&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Minus&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns the difference of the the first and second argument.

**Minus<T>() Member Function**

Default constructor.

**Syntax**

```
public Minus<T>();
```

**Minus<T>(const System.Minus<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Minus<T>(const System.Minus<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Minus<T>&	Argument to copy.

**operator=(const System.Minus<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Minus<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Minus<T>&	Argument to assign.

**Minus<T>(System.Minus<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Minus<T>(System.Minus<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Minus<T>&&	Argument to move from.

**operator=(System.Minus<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Minus<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Minus<T>&&	Argument to assign from.

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

Returns the difference of the the first and second argument.

**Syntax**

```
public 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  $a - b$ .

## 2.2.39 Multiplies<T> Class

A multiplication binary function object.

### Syntax

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

### Constraint

T is [MultiplicativeSemigroup](#)

### Model of

[BinaryOperation<T>](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.39.1 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);
}
```

#### 2.2.39.2 Member Functions

Member Function	Description
<a href="#">Multiplies&lt;T&gt;()</a>	Default constructor.
<a href="#">Multiplies&lt;T&gt;(const System.Multiplies&lt;T&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Multiplies&lt;T&gt;&amp;)</a>	Copy assignment.
<a href="#">Multiplies&lt;T&gt;(System.Multiplies&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Multiplies&lt;T&gt;&amp;&amp;)</a>	Move assignment.

`operator()(const T&, const T&) const`

Returns the first argument multiplied with the second argument.

**Multiplies<T>() Member Function**

Default constructor.

**Syntax**

```
public Multiplies<T>();
```

**Multiplies<T>(const System.Multiplies<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Multiplies<T>(const System.Multiplies<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Multiplies<T>&	Argument to copy.

**operator=(const System.Multiplies<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Multiplies<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Multiplies<T>&	Argument to assign.

**Multiplies<T>(System.Multiplies<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Multiplies<T>(System.Multiplies<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Multiplies<T>&&	Argument to move from.

**operator=(System.Multiplies<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Multiplies<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Multiplies<T>&&	Argument to assign from.

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

Returns the first argument multiplied with the second argument.

**Syntax**

```
public 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  $a * b$ .

## 2.2.40 Negate<T> Class

A negation unary operation.

### Syntax

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

### Constraint

T is [AdditiveGroup](#)

### Model of

[UnaryOperation<T>](#)

### Base Class

[System.UnaryFun<T, T>](#)

## 2.2.40.1 Member Functions

Member Function	Description
<code>Negate&lt;T&gt;()</code>	Default constructor.
<code>Negate&lt;T&gt;(const System.Negate&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Negate&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Negate&lt;T&gt;(System.Negate&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Negate&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;) const</code>	Returns the negation of its argument.

**Negate<T>() Member Function**

Default constructor.

**Syntax**

```
public Negate<T>();
```

**Negate<T>(const System.Negate<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Negate<T>(const System.Negate<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Negate<T>&	Argument to copy.

**operator=(const System.Negate<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Negate<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Negate<T>&	Argument to assign.

**Negate<T>(System.Negate<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Negate<T>(System.Negate<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Negate<T>&&	Argument to move from.

**operator=(System.Negate<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Negate<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Negate<T>&&	Argument to assign from.

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

Returns the negation of its argument.

**Syntax**

```
public T operator()(const T& a) const;
```

**Parameters**

Name	Type	Description
a	const T&	The argument.

**Returns**

T

Returns  $-a$ .

### 2.2.41 NotEqualTo2<T, U> Class

A *not equal to* binary predicate.

#### Syntax

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

#### Constraint

[EqualityComparable<T, U>](#)

#### Model of

[Relation<T, U, V>](#)

#### Base Class

[System.BinaryPred<T, U>](#)

#### 2.2.41.1 Remarks

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

#### 2.2.41.2 Member Functions

Member Function	Description
<a href="#">NotEqualTo2&lt;T, U&gt;()</a>	Default constructor.
<a href="#">NotEqualTo2&lt;T, U&gt;(const System.-NotEqualTo2&lt;T, U&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.NotEqualTo2&lt;T, U&gt;&amp;)</a>	Copy assignment.
<a href="#">NotEqualTo2&lt;T, U&gt;(System.NotEqualTo2&lt;T, U&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.NotEqualTo2&lt;T, U&gt;&amp;&amp;)</a>	Move assignment.
<a href="#">operator()(const T&amp;, const U&amp;) const</a>	Returns true if the first argument is not equal to the second argument, false otherwise.

**NotEqualTo2<T, U>() Member Function**

Default constructor.

**Syntax**

```
public NotEqualTo2<T, U>();
```

**NotEqualTo2<T, U>(const System.NotEqualTo2<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public NotEqualTo2<T, U>(const System.NotEqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.NotEqualTo2<T, U>&	Argument to copy.

**operator=(const System.NotEqualTo2<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.NotEqualTo2<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.NotEqualTo2<T, U>&	Argument to assign.

**NotEqualTo2<T, U>(System.NotEqualTo2<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public NotEqualTo2<T, U>(System.NotEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.NotEqualTo2<T, U>&&	Argument to move from.

**operator=(System.NotEqualTo2<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.NotEqualTo2<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.NotEqualTo2<T, U>&&	Argument to assign from.

**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 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*.

### 2.2.42 NotEqualTo<T> Class

An *not equal to* relation.

#### Syntax

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

#### Constraint

T is [Regular](#)

#### Model of

[Relation<T>](#)

#### Base Class

[System.Rel<T>](#)

#### 2.2.42.1 Member Functions

Member Function	Description
<code>NotEqualTo&lt;T&gt;()</code>	Default constructor.
<code>NotEqualTo&lt;T&gt;(const System.NotEqualTo&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.NotEqualTo&lt;T&gt;&amp;)</code>	Copy assignment.
<code>NotEqualTo&lt;T&gt;(System.NotEqualTo&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.NotEqualTo&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns true if the first argument is not equal to the second argument, false otherwise.

**NotEqualTo<T>() Member Function**

Default constructor.

**Syntax**

```
public NotEqualTo<T>();
```

**NotEqualTo<T>(const System.NotEqualTo<T>&) Member Function**

Copy constructor.

**Syntax**

```
public NotEqualTo<T>(const System.NotEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.NotEqualTo<T>&	Argument to copy.

**operator=(const System.NotEqualTo<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.NotEqualTo<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.NotEqualTo<T>&	Argument to assign.

**NotEqualTo<T>(System.NotEqualTo<T>&&) Member Function**

Move constructor.

**Syntax**

```
public NotEqualTo<T>(System.NotEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.NotEqualTo<T>&&	Argument to move from.

**operator=(System.NotEqualTo<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.NotEqualTo<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.NotEqualTo<T>&&	Argument to assign from.

**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 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*.

## 2.2.43 Pair<T, U> Class

A pair of values.

### Syntax

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

### Constraint

T is [Semiregular](#) and U is [Semiregular](#)

#### 2.2.43.1 Remarks

The **Pair<T, U>** class is used for example in the system library algorithm `System.EqualRange.I.T.I.is.ForwardIterator.and.TotallyOrdered.T.I.ValueType.I.I.T.const.ref` 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.

#### 2.2.43.2 Member Functions

Member Function	Description
<code>Pair&lt;T, U&gt;()</code>	Constructor. Constructs a pair with default values.
<code>Pair&lt;T, U&gt;(const System.Pair&lt;T, U&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Pair&lt;T, U&gt;&amp;)</code>	Copy assignment.
<code>Pair&lt;T, U&gt;(System.Pair&lt;T, U&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Pair&lt;T, U&gt;&amp;&amp;)</code>	Move assignment.
<code>Pair&lt;T, U&gt;(const T&amp;, const U&amp;)</code>	Constructor. Constructs a pair with specified values.

**Pair<T, U>() Member Function**

Constructor. Constructs a pair with default values.

**Syntax**

```
public Pair<T, U>();
```

**Pair<T, U>(const System.Pair<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public Pair<T, U>(const System.Pair<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Pair<T, U>&	Argument to copy.

**operator=(const System.Pair<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Pair<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.Pair<T, U>&	Argument to assign.

**Pair<T, U>(System.Pair<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public Pair<T, U>(System.Pair<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.Pair<T, U>&&	Argument to move from.

**operator=(System.Pair<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Pair<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.Pair<T, U>&&	Argument to assign from.

**Pair<T, U>(const T&, const U&) Member Function**

Constructor. Constructs a pair with specified values.

**Syntax**

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

**Parameters**

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

**2.2.43.3 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.Pair&lt;T, U&gt;&amp;, const System.Pair&lt;T, U&gt;&amp;)</code>	Compares two pairs for less than relationship.
<code>operator==(const System.Pair&lt;T, U&gt;&amp;, const System.Pair&lt;T, U&gt;&amp;)</code>	Compares two pairs for equality.

**operator<(const System.Pair<T, U>&, const System.Pair<T, U>&) Function**

Compares two pairs for less than relationship.

**Syntax**

```
public bool operator<(const System.Pair<T, U>& left, const System.Pair<T, U>& right);
```

**Constraint**

T is [TotallyOrdered](#) and U is [TotallyOrdered](#)

**Parameters**

Name	Type	Description
left	const System.Pair<T, U>&	The first pair.
right	const System.Pair<T, U>&	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.

**operator==(const System.Pair<T, U>&, const System.Pair<T, U>&) Function**

Compares two pairs for equality.

**Syntax**

```
public bool operator==(const System.Pair<T, U>& left, const System.Pair<T, U>& right);
```

**Constraint**

T is [Regular](#) and U is [Regular](#)

**Parameters**

Name	Type	Description
left	const System.Pair<T, U>&	The first pair.
right	const System.Pair<T, U>&	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.

## 2.2.44 Plus<T> Class

An addition binary function object.

### Syntax

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

### Constraint

T is [AdditiveSemigroup](#)

### Model of

[BinaryOperation<T>](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.44.1 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);
}
```

#### 2.2.44.2 Member Functions

Member Function	Description
<a href="#">Plus&lt;T&gt;()</a>	Default constructor.
<a href="#">Plus&lt;T&gt;(const System.Plus&lt;T&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.Plus&lt;T&gt;&amp;)</a>	Copy assignment.
<a href="#">Plus&lt;T&gt;(System.Plus&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Plus&lt;T&gt;&amp;&amp;)</a>	Move assignment.

`operator()(const T&, const T&) const`

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

**Plus<T>() Member Function**

Default constructor.

**Syntax**

```
public Plus<T>();
```

**Plus<T>(const System.Plus<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Plus<T>(const System.Plus<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Plus<T>&	Argument to copy.

**operator=(const System.Plus<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Plus<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Plus<T>&	Argument to assign.

**Plus<T>(System.Plus<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Plus<T>(System.Plus<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Plus<T>&&	Argument to move from.

**operator=(System.Plus<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Plus<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Plus<T>&&	Argument to assign from.

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

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

**Syntax**

```
public 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  $a + b$ .

## 2.2.45 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>](#)

#### 2.2.45.1 Remarks

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

### 2.2.45.2 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.

### 2.2.45.3 Member Functions

Member Function	Description
<code>RandomAccessIter&lt;T, R, P&gt;()</code>	Constructor. Default constructs a random access iterator.
<code>RandomAccessIter&lt;T, R, P&gt;(const System.-&gt;RandomAccessIter&lt;T, R, P&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.RandomAccessIter&lt;T, R, P&gt;&amp;)</code>	Copy assignment.
<code>RandomAccessIter&lt;T, R, P&gt;(System.-&gt;RandomAccessIter&lt;T, R, P&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.RandomAccessIter&lt;T, R, P&gt;&amp;&amp;)</code>	Move assignment.
<code>GetPtr() const</code>	Returns the contained pointer.
<code>RandomAccessIter&lt;T, R, P&gt;(P)</code>	Constructor. Constructs a random access iterator with a pointer to elements.
<code>operator*() const</code>	Returns a reference to the element currently pointed to.
<code>operator++()</code>	Advances the random access iterator to point to the succeeding element.
<code>operator--()</code>	Backs the random access iterator to point to the preceding element.
<code>operator-&gt;() const</code>	Returns a pointer to the element currently pointed to.
<code>operator[](int) const</code>	Returns a reference to an element with a given index.

**RandomAccessIter<T, R, P>() Member Function**

Constructor. Default constructs a random access iterator.

**Syntax**

```
public RandomAccessIter<T, R, P>();
```

**RandomAccessIter<T, R, P>(const System.RandomAccessIter<T, R, P>&) Member Function**

Copy constructor.

**Syntax**

```
public RandomAccessIter<T, R, P>(const System.RandomAccessIter<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.RandomAccessIter<T, R, P>&	Argument to copy.

**operator=(const System.RandomAccessIter<T, R, P>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.RandomAccessIter<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.RandomAccessIter<T, R, P>&	Argument to assign.

**RandomAccessIter<T, R, P>(System.RandomAccessIter<T, R, P>&&) Member Function**

Move constructor.

**Syntax**

```
public RandomAccessIter<T, R, P>(System.RandomAccessIter<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.RandomAccessIter<T, R, P>&&	Argument to move from.

**operator=(System.RandomAccessIter<T, R, P>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.RandomAccessIter<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.RandomAccessIter<T, R, P>&&	Argument to assign from.

**GetPtr() const Member Function**

Returns the contained pointer.

**Syntax**

```
public P GetPtr() const;
```

**Returns**

P

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

**RandomAccessIter<T, R, P>(P) Member Function**

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

**Syntax**

```
public RandomAccessIter<T, R, P>(P ptr_);
```

**Parameters**

Name	Type	Description
ptr_	P	A pointer to elements.

**operator\*() const Member Function**

Returns a reference to the element currently pointed to.

**Syntax**

```
public R operator*() const;
```

**Returns**

R

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

**operator++() Member Function**

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

**Syntax**

```
public System.RandomAccessIter<T, R, P>& operator++();
```

**Returns**

[System.RandomAccessIter<T, R, P>&](#)

Returns a reference to the random access iterator.

**operator–() Member Function**

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

**Syntax**

```
public System.RandomAccessIter<T, R, P>& operator--();
```

**Returns**

[System.RandomAccessIter<T, R, P>&](#)

Returns a reference to the random access iterator.

**operator->() const Member Function**

Returns a pointer to the element currently pointed to.

**Syntax**

```
public P operator->() const;
```

**Returns**

P

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

**operator[](int) const Member Function**

Returns a reference to an element with a given index.

**Syntax**

```
public R operator[](int index) const;
```

**Parameters**

Name	Type	Description
index	int	An index.

**Returns**

R

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

**2.2.45.4 Nonmember Functions**

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

**operator+(const System.RandomAccessIter<T, R, P>&, int) Function**

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

**Syntax**

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

**Parameters**

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

**Returns**

`System.RandomAccessIter<T, R, P>`

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

**operator+(int, const System.RandomAccessIter<T, R, P>&) Function**

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

**Syntax**

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

**Parameters**

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

**Returns**

System.RandomAccessIter<T, R, P>

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

**operator-(const System.RandomAccessIter<T, R, P>&, const System.RandomAccessIter<T, R, P>&) Function**

Returns the distance between two random access iterators.

**Syntax**

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

**Parameters**

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

**operator-(const System.RandomAccessIter<T, R, P>&, int) Function**

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

**Syntax**

```
public System.RandomAccessIter<T, R, P> operator-(const System.RandomAccessIter<T, R, P>& it, int offset);
```

**Parameters**

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

**Returns**

System.RandomAccessIter<T, R, P>

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

**operator<(const System.RandomAccessIter<T, R, P>&, const System.RandomAccessIter<T, R, P>&) Function**

Compares two random access iterators for less than relationship.

**Syntax**

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

**Parameters**

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

**operator==(const System.RandomAccessIter<T, R, P>&, const System.RandomAccessIter<T, R, P>&) Function**

Compares two random access iterators for equality.

**Syntax**

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

**Parameters**

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

## 2.2.46 Rel<Argument> Class

A base class for relation function objects.

### Syntax

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

### Constraint

Argument is [Semiregular](#)

### Model of

[Relation<T>](#)

### Base Class

[System.BinaryPred<Argument, Argument>](#)

#### 2.2.46.1 Remarks

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

### 2.2.46.2 Type Definitions

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

### 2.2.46.3 Member Functions

Member Function	Description
<code>Rel&lt;Argument&gt;()</code>	Default constructor.
<code>Rel&lt;Argument&gt;(const System.Rel&lt;Argument&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Rel&lt;Argument&gt;&amp;)</code>	Copy assignment.
<code>Rel&lt;Argument&gt;(System.Rel&lt;Argument&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Rel&lt;Argument&gt;&amp;&amp;)</code>	Move assignment.

**Rel<Argument>() Member Function**

Default constructor.

**Syntax**

```
public Rel<Argument>();
```

**Rel<Argument>(const System.Rel<Argument>&) Member Function**

Copy constructor.

**Syntax**

```
public Rel<Argument>(const System.Rel<Argument>& that);
```

**Parameters**

Name	Type	Description
that	const System.Rel<Argument>&	Argument to copy.

**operator=(const System.Rel<Argument>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Rel<Argument>& that);
```

**Parameters**

Name	Type	Description
that	const System.Rel<Argument>&	Argument to assign.

**Rel<Argument>(System.Rel<Argument>&&) Member Function**

Move constructor.

**Syntax**

```
public Rel<Argument>(System.Rel<Argument>&& that);
```

**Parameters**

Name	Type	Description
that	System.Rel<Argument>&&	Argument to move from.

**operator=(System.Rel<Argument>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Rel<Argument>&& that);
```

**Parameters**

Name	Type	Description
that	System.Rel<Argument>&&	Argument to assign from.

### 2.2.47 Remainder<T> Class

A remainder function object.

#### Syntax

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

#### Constraint

T is [EuclideanSemiring](#)

#### Model of

[BinaryFunction<T>](#)

#### Base Class

[System.BinaryFun<T, T, T>](#)

### 2.2.47.1 Member Functions

Member Function	Description
<code>Remainder&lt;T&gt;()</code>	Default constructor.
<code>Remainder&lt;T&gt;(const System.Remainder&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Remainder&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Remainder&lt;T&gt;(System.Remainder&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Remainder&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns the remainder of division of the first argument and the second argument.

**Remainder<T>() Member Function**

Default constructor.

**Syntax**

```
public Remainder<T>();
```

**Remainder<T>(const System.Remainder<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Remainder<T>(const System.Remainder<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Remainder<T>&	Argument to copy.

**operator=(const System.Remainder<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Remainder<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Remainder<T>&	Argument to assign.

**Remainder<T>(System.Remainder<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Remainder<T>(System.Remainder<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Remainder<T>&&	Argument to move from.

**operator=(System.Remainder<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Remainder<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Remainder<T>&&	Argument to assign from.

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

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

**Syntax**

```
public 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  $a \% b$ .

## 2.2.48 SelectFirst<T, U> Class

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

### Syntax

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

### Constraint

T is [Semiregular](#) and U is [Semiregular](#)

### Base Class

[System.UnaryFun<System.Pair<T, U>, T>](#)

#### 2.2.48.1 Member Functions

Member Function	Description
SelectFirst<T, U>()	Default constructor.
SelectFirst<T, U>(const System.SelectFirst<T, U>&)	Copy constructor.
operator=(const System.SelectFirst<T, U>&)	Copy assignment.
SelectFirst<T, U>(System.SelectFirst<T, U>& &)	Move constructor.
operator=(System.SelectFirst<T, U>&&)	Move assignment.
operator()(const System.Pair<T, U>&) const	Returns the first component of the given pair.

**SelectFirst<T, U>() Member Function**

Default constructor.

**Syntax**

```
public SelectFirst<T, U>();
```

**SelectFirst<T, U>(const System.SelectFirst<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public SelectFirst<T, U>(const System.SelectFirst<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.SelectFirst<T, U>&	Argument to copy.

**operator=(const System.SelectFirst<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.SelectFirst<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.SelectFirst<T, U>&	Argument to assign.

**SelectFirst<T, U>(System.SelectFirst<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public SelectFirst<T, U>(System.SelectFirst<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.SelectFirst&lt;T, U&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.SelectFirst<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.SelectFirst<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.SelectFirst<T, U>&&	Argument to assign from.

**operator()(const System.Pair<T, U>&) const Member Function**

Returns the first component of the given pair.

**Syntax**

```
public const T& operator()(const System.Pair<T, U>& p) const;
```

**Parameters**

Name	Type	Description
p	const System.Pair<T, U>&	A pair of values.

**Returns**

const T&

Returns the first component of *p*.

## 2.2.49 SelectSecond<T, U> Class

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

### Syntax

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

### Constraint

T is [Semiregular](#) and U is [Semiregular](#)

### Base Class

[System.UnaryFun<System.Pair<T, U>, U>](#)

### 2.2.49.1 Member Functions

Member Function	Description
SelectSecond<T, U>()	Default constructor.
SelectSecond<T, U>(const System. <a href="#">SelectSecond&lt;T, U&gt; &amp;</a> )	Copy constructor.
operator=(const System.SelectSecond<T, U>&)	Copy assignment.
SelectSecond<T, U>(System.SelectSecond<T, U>&&)	Move constructor.
operator=(System.SelectSecond<T, U>&&)	Move assignment.
operator()(const System.Pair<T, U>&) const	Returns the second component of the given pair.

**SelectSecond<T, U>() Member Function**

Default constructor.

**Syntax**

```
public SelectSecond<T, U>();
```

**SelectSecond<T, U>(const System.SelectSecond<T, U>&) Member Function**

Copy constructor.

**Syntax**

```
public SelectSecond<T, U>(const System.SelectSecond<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.SelectSecond<T, U>&	Argument to copy.

**operator=(const System.SelectSecond<T, U>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.SelectSecond<T, U>& that);
```

**Parameters**

Name	Type	Description
that	const System.SelectSecond<T, U>&	Argument to assign.

**SelectSecond<T, U>(System.SelectSecond<T, U>&&) Member Function**

Move constructor.

**Syntax**

```
public SelectSecond<T, U>(System.SelectSecond<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.SelectSecond<T, U>&&	Argument to move from.

**operator=(System.SelectSecond<T, U>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.SelectSecond<T, U>&& that);
```

**Parameters**

Name	Type	Description
that	System.SelectSecond<T, U>&&	Argument to assign from.

**operator()(const System.Pair<T, U>&) const Member Function**

Returns the second component of the given pair.

**Syntax**

```
public const U& operator()(const System.Pair<T, U>& p) const;
```

**Parameters**

Name	Type	Description
p	const System.Pair<T, U>&	A pair of values.

**Returns**

const U&

Returns the second component of *p*.

## 2.2.50 ShareableFromThis<T> Class

A class that implements the “shared from this” idiom.

### Syntax

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

#### 2.2.50.1 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 than constructors) provided that there is a “living” **SharedPtr<T>** to the object.

#### 2.2.50.2 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();
}
```

#### 2.2.50.3 Member Functions

Member Function	Description
<a href="#">ShareableFromThis&lt;T&gt;()</a>	Default constructor.
<a href="#">ShareableFromThis&lt;T&gt;(const ShareableFromThis&lt;T&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const ShareableFromThis&lt;T&gt;&amp;)</a>	Copy assignment.
<a href="#">ShareableFromThis&lt;T&gt;(System.. ShareableFromThis&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.ShareableFromThis&lt;T&gt;&amp;&amp;)</a>	Move assignment.
<a href="#">GetSharedFromThis() const</a>	Returns a shared pointer to the class object.
<a href="#">GetWeakThis()</a>	Returns the contained weak pointer to the class object.



**ShareableFromThis<T>() Member Function**

Default constructor.

**Syntax**

```
public ShareableFromThis<T>();
```

**ShareableFromThis<T>(const System.ShareableFromThis<T>&) Member Function**

Copy constructor.

**Syntax**

```
public ShareableFromThis<T>(const System.ShareableFromThis<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShareableFromThis<T>&	Argument to copy.

**operator=(const System.ShareableFromThis<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.ShareableFromThis<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShareableFromThis<T>&	Argument to assign.

**ShareableFromThis<T>(System.ShareableFromThis<T>&&) Member Function**

Move constructor.

**Syntax**

```
public ShareableFromThis<T>(System.ShareableFromThis<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.ShareableFromThis<T>&&	Argument to move from.

**operator=(System.ShareableFromThis<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.ShareableFromThis<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.ShareableFromThis<T>&&	Argument to assign from.

**GetSharedFromThis() const Member Function**

Returns a shared pointer to the class object.

**Syntax**

```
public SystemSharedPtr<T> GetSharedFromThis() const;
```

**Returns**

[SystemSharedPtr<T>](#)

Returns a shared pointer to the class object.

**GetWeakThis() Member Function**

Returns the contained weak pointer to the class object.

**Syntax**

```
public System.WeakPtr<T>& GetWeakThis();
```

**Returns**

[System.WeakPtr<T>&](#)

Returns the contained weak pointer to the class object.

## 2.2.51 SharedCount<T> Class

A handle to a [Counter<T>](#) that maintains the use count portion of the counter.

### Syntax

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

#### 2.2.51.1 Member Functions

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

**SharedCount<T>() Member Function**

Constructor. Initializes an empty shared count.

**Syntax**

```
public SharedCount<T>();
```

**SharedCount<T>(const System.SharedCount<T>&) Member Function**

Constructor. Implementation detail.

**Syntax**

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

**Parameters**

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

**SharedCount<T>(const System.SharedCount<T>&) Member Function**

Constructor. Implementation detail.

**Syntax**

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

**Parameters**

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

**operator=(const System.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 void operator=(const System.SharedCount<T>& that);
```

**Parameters**

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

**operator=(const System.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 void operator=(const System.SharedCount<T>& that);
```

**Parameters**

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

**GetCounter() const Member Function**

Returns the contained pointer to a counter.

**Syntax**

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

**Returns**

[System.Counter<T>\\*](#)

Returns the contained pointer to a counter.

**GetUseCount() const Member Function**

Returns the use count of the counter.

**Syntax**

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

**Returns**

int

Returns the use count of the counter.

**IsUnique() const Member Function**

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

**Syntax**

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

**Returns**

`bool`

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

**SharedCount<T>(System.Counter<T>\*) Member Function**

Copy constructor. Increments the use count.

**Syntax**

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

**Parameters**

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

**SharedCount<T>(T\*) Member Function**

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

**Syntax**

```
public SharedCount<T>(T* ptr_);
```

**Parameters**

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

**SharedCount<T>(const System.WeakCount<T>&) Member Function**

Constructor. Gets the counter from a [WeakCount<T>](#).

**Syntax**

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

**Parameters**

Name	Type	Description
that	<a href="#">const System.WeakCount&lt;T&gt;&amp;</a>	A weak count.

**Swap(System.SharedCount<T>&) Member Function**

Exchanges the contents with another shared count.

**Syntax**

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

**Parameters**

Name	Type	Description
that	<a href="#">System.SharedCount&lt;T&gt;&amp;</a>	A shared count to exchange contents with.

**~SharedCount<T>() Member Function**

Destructor. Decrements the use count.

**Syntax**

```
public ~SharedCount<T>();
```

**2.2.51.2 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.SharedCount&lt;T&gt;&amp;, const System.SharedCount&lt;T&gt;&amp;)</code>	Compares two shared count objects for less than relationship.
<code>operator==(const System.SharedCount&lt;T&gt;&amp;, const System.SharedCount&lt;T&gt;&amp;)</code>	Compares two shared count objects for equality.

**operator<(const System.SharedCount<T>&, const System.SharedCount<T>&) Function**

Compares two shared count objects for less than relationship.

**Syntax**

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

**Parameters**

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

**operator==(const System.SharedCount<T>&, const System.SharedCount<T>&) Function**

Compares to shared count objects for equality.

**Syntax**

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

**Parameters**

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

**Returns**

bool

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

## 2.2.52 SharedPtr<T> Class

A shared pointer to an object.

### Syntax

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

### Model of

[TotallyOrdered<T>](#)

#### 2.2.52.1 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();
    }
}

```

### 2.2.52.2 Member Functions

Member Function	Description
<code>SharedPtr&lt;T&gt;()</code>	Constructor. Constructs a null shared pointer.
<code>SharedPtr&lt;T&gt;(const SystemSharedPtr&lt;T&gt;&amp;)</code>	Copy constructor. Increments the use count.
<code>SharedPtr&lt;T&gt;(const SystemSharedPtr&lt;T&gt;&amp;)</code>	Copy constructor. Increments the use count.
<code>operator=(const SystemSharedPtr&lt;T&gt;&amp;)</code>	Copy assignment. Assigns another shared pointer.
<code>operator=(const SystemSharedPtr&lt;T&gt;&amp;)</code>	Copy assignment. Assigns another shared pointer.
<code>GetCount() const</code>	Returns the contained <code>SharedCount&lt;T&gt;</code> .
<code>GetPtr() const</code>	Returns the contained pointer to the counted object.
<code>GetUseCount() const</code>	Returns the use count.
<code>IsNull() const</code>	Returns true if the contained pointer is null, false otherwise.
<code>IsUnique() const</code>	Returns true if there is exactly one <code>SharedPtr&lt;T&gt;</code> to an object.
<code>Reset()</code>	Resets the shared pointer to null.
<code>Reset(T*)</code>	Resets the shared pointer to contain a pointer to another counted object.

<code>SharedPtr&lt;T&gt;(T*)</code>		Constructor. Constructs a shared pointer to the given object.
<code>SharedPtr&lt;T&gt;(T*, const System.- SharedCount&lt;T&gt;&amp;)</code>	<code>System.-</code>	Constructor. Implementation detail to support System.PtrCast.U.T.SystemSharedPtr.T.-const.ref function.
<code>SharedPtr&lt;T&gt;(const System.WeakPtr&lt;T&gt;&amp;)</code>		Constructor. Constructs a shared pointer from a weak pointer.
<code>Swap(SystemSharedPtr&lt;T&gt;&amp;)</code>		Exchanges the contents with another shared pointer.
<code>operator*() const</code>		Returns a reference to the counted object.
<code>operator-&gt;() const</code>		Returns the contained pointer to the counted object.

**SharedPtr<T>() Member Function**

Constructor. Constructs a null shared pointer.

**Syntax**

```
public SharedPtr<T>();
```

**SharedPtr<T>(const SystemSharedPtr<T>&) Member Function**

Copy constructor. Increments the use count.

**Syntax**

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

**Parameters**

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

**SharedPtr<T>(const SystemSharedPtr<T>&) Member Function**

Copy constructor. Increments the use count.

**Syntax**

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

**Parameters**

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

**operator=(const SystemSharedPtr<T>&)** Member Function

Copy assignment. Assigns another shared pointer.

**Syntax**

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

**Parameters**

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

**operator=(const SystemSharedPtr<T>&)** Member Function

Copy assignment. Assigns another shared pointer.

**Syntax**

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

**Parameters**

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

**GetCount() const Member Function**

Returns the contained [SharedCount<T>](#).

**Syntax**

```
public const System.SharedCount<T>& GetCount() const;
```

**Returns**

```
const System.SharedCount<T>&
```

Returns the contained shared count.

**GetPtr() const Member Function**

Returns the contained pointer to the counted object.

**Syntax**

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

**Returns**

T\*

Returns the contained pointer to the counted object.

**GetUseCount() const Member Function**

Returns the use count.

**Syntax**

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

**Returns**

int

Returns the use count.

**IsNull() const Member Function**

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

**Syntax**

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

**Returns**

bool

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

**IsUnique() const Member Function**

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

**Syntax**

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

**Returns**

`bool`

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

**Reset() Member Function**

Resets the shared pointer to null.

**Syntax**

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

**Reset(T\*) Member Function**

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

**Syntax**

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

**Parameters**

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

**SharedPtr<T>(T\*) Member Function**

Constructor. Constructs a shared pointer to the given object.

**Syntax**

```
public SharedPtr<T>(T* ptr_);
```

**Parameters**

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

**SharedPtr<T>(T\*, const System.SharedCount<T>&) Member Function**

Constructor. Implementation detail to support  
System.PtrCast.U.T.SystemSharedPtr.T.const.ref function.

**Syntax**

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

**Parameters**

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

**SharedPtr<T>(const System::WeakPtr<T>&) Member Function**

Constructor. Constructs a shared pointer from a weak pointer.

**Syntax**

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

**Parameters**

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

**Swap(SystemSharedPtr<T>&) Member Function**

Exchanges the contents with another shared pointer.

**Syntax**

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

**Parameters**

Name	Type	Description
that	<a href="#">SystemSharedPtr&lt;T&gt;&amp;</a>	A shared pointer to exchange contents with.

**operator\*() const Member Function**

Returns a reference to the counted object.

**Syntax**

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

**Returns**

T&

Returns a reference to the counted object.

**operator->() const Member Function**

Returns the contained pointer to the counted object.

**Syntax**

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

**Returns**

T\*

Returns the contained pointer to the counted object.

**2.2.52.3 Nonmember Functions**

Function	Description
<code>operator&lt;(const SystemSharedPtr&lt;T&gt;&amp;, const SystemSharedPtr&lt;T&gt;&amp;)</code>	Compares two shared pointers for less than relationship.
<code>operator==(const SystemSharedPtr&lt;T&gt;&amp;, const SystemSharedPtr&lt;T&gt;&amp;)</code>	Compares two shared pointers for equality.

**operator<(const SystemSharedPtr<T>&, const SystemSharedPtr<T>&) Function**

Compares two shared pointers for less than relationship.

**Syntax**

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

**Parameters**

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

**operator==(const SystemSharedPtr<T>&, const SystemSharedPtr<T>&) Function**

Compares two shared pointers for equality.

**Syntax**

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

**Parameters**

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

**Returns**

bool

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

### 2.2.53 ShiftLeft<T> Class

Shift left binary function.

#### Syntax

```
public class ShiftLeft<T>;
```

#### Constraint

T is [Semiregular](#)

#### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.53.1 Member Functions

Member Function	Description
<code>ShiftLeft&lt;T&gt;()</code>	Default constructor.
<code>ShiftLeft&lt;T&gt;(const System.ShiftLeft&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.ShiftLeft&lt;T&gt;&amp;)</code>	Copy assignment.
<code>ShiftLeft&lt;T&gt;(System.ShiftLeft&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.ShiftLeft&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns <code>left</code> shifted left by <code>right</code> bit positions.

**ShiftLeft<T>() Member Function**

Default constructor.

**Syntax**

```
public ShiftLeft<T>();
```

**ShiftLeft<T>(const System.ShiftLeft<T>&) Member Function**

Copy constructor.

**Syntax**

```
public ShiftLeft<T>(const System.ShiftLeft<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShiftLeft<T>&	Argument to copy.

**operator=(const System.ShiftLeft<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.ShiftLeft<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShiftLeft<T>&	Argument to assign.

**ShiftLeft<T>(System.ShiftLeft<T>&&) Member Function**

Move constructor.

**Syntax**

```
public ShiftLeft<T>(System.ShiftLeft<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.ShiftLeft&lt;T&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.ShiftLeft<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.ShiftLeft<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.ShiftLeft&lt;T&gt;&amp;&amp;</a>	Argument to assign from.

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

Returns `left` shifted left by `right` bit positions.

**Syntax**

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

**Parameters**

Name	Type	Description
left	const T&	Left argument.
right	const T&	Right argument.

**Returns**

T

Returns `left` shifted left by `right` bit positions.

## 2.2.54 ShiftRight<T> Class

Shift right binary function.

### Syntax

```
public class ShiftRight<T>;
```

### Constraint

T is [Semiregular](#)

### Base Class

[System.BinaryFun<T, T, T>](#)

#### 2.2.54.1 Member Functions

Member Function	Description
<code>ShiftRight&lt;T&gt;()</code>	Default constructor.
<code>ShiftRight&lt;T&gt;(const System.ShiftRight&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.ShiftRight&lt;T&gt;&amp;)</code>	Copy assignment.
<code>ShiftRight&lt;T&gt;(System.ShiftRight&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.ShiftRight&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;, const T&amp;) const</code>	Returns <code>left</code> shifted right by <code>right</code> bit positions.

**ShiftRight<T>() Member Function**

Default constructor.

**Syntax**

```
public ShiftRight<T>();
```

**ShiftRight<T>(const System.ShiftRight<T>&) Member Function**

Copy constructor.

**Syntax**

```
public ShiftRight<T>(const System.ShiftRight<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShiftRight<T>&	Argument to copy.

**operator=(const System.ShiftRight<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.ShiftRight<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.ShiftRight<T>&	Argument to assign.

**ShiftRight<T>(System.ShiftRight<T>&&) Member Function**

Move constructor.

**Syntax**

```
public ShiftRight<T>(System.ShiftRight<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.ShiftRight<T>&&	Argument to move from.

**operator=(System.ShiftRight<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.ShiftRight<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.ShiftRight&lt;T&gt;&amp;&amp;</a>	Argument to assign from.

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

Returns `left` shifted right by `right` bit positions.

**Syntax**

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

**Parameters**

Name	Type	Description
left	const T&	Left argument.
right	const T&	Right argument.

**Returns**

T

Returns `left` shifted right by `right` bit positions.

## 2.2.55 String Class

A string of ASCII characters.

### Syntax

```
public class String;
```

### 2.2.55.1 Type Definitions

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

### 2.2.55.2 Member Functions

Member Function	Description
<code>String()</code>	Constructor. Constructs an empty string.
<code>String(const System.String&amp;)</code>	Copy constructor.
<code>operator=(const System.String&amp;)</code>	Copy assignment.
<code>String(System.String&amp;&amp;)</code>	Move constructor.
<code>operator=(System.String&amp;&amp;)</code>	Move assignment.
<code>Append(char)</code>	Appends the given character to the end of this string.
<code>Append(char, int)</code>	Appends given number of occurrences of given character to the end of this string.
<code>Append(const System.String&amp;)</code>	Appends the specified C-style string to the end of this string.
<code>Append(const char*)</code>	Appends the specified string to the end of this string.
<code>Append(const char*, int)</code>	Appends the given number of characters from a C-style string to the end of this string.
<code>Begin()</code>	Returns an iterator to the beginning of the string.
<code>Begin() const</code>	Returns a constant iterator to the beginning of the string.
<code>CBegin() const</code>	Returns a constant iterator to the beginning of the string.
<code>CEnd() const</code>	Returns a constant iterator one past the end of the string.
<code>Capacity() const</code>	Returns the number of characters that the string can hold without allocating more memory for it.

Chars() const	Returns the string as a C-style 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 System.String&) const	Returns true, if the string ends with the specified substring, false otherwise.
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.
Find(const System.String&) const	Returns the starting index of the first occurrence of the specified substring within this string, or -1 if the specified string does not occur within this string.
Find(const System.String&, int) const	Returns the starting index of the first occurrence of the specified substring within this string, or -1 if the specified string does not occur within 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(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.

<code>RFind(const System.String&amp;) const</code>	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.
<code>RFind(const System.String&amp;, int) const</code>	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.
<code>Replace(char, char)</code>	Replaces every occurrence of the given character with another character.
<code>Reserve(int)</code>	Reserves room for a string with the given number of characters.
<code>Split(char)</code>	Returns a list of substrings separated by the given character.
<code>StartsWith(const System.String&amp;) const</code>	Returns true if the string starts with the given substring, false otherwise.
<code>String(char)</code>	Constructor. Constructs a string that has the given character.
<code>String(char, int)</code>	Constructor. Constructs a string that has given number of the specified character.
<code>String(const char*)</code>	Constructor. Constructs a string from a C-style string.
<code>String(const char*, const char*)</code>	Constructor. Constructs a string of characters included in range <code>begin</code> and <code>end</code> .
<code>String(const char*, int)</code>	Constructor. Constructs a string from given number of characters of a C-style string.
<code>Substring(int) const</code>	Returns a substring starting from the given index.
<code>Substring(int, int) const</code>	Returns a substring starting from the given index whose length is at most given number of characters.
<code>operator&lt;(const System.String&amp;) const</code>	Compares a string for less than relationship with another string.

<code>operator==(const System.String&amp;) const</code>	Compares a string for equality with another string.
<code>operator[](int)</code>	Returns a reference to the character with the specified index.
<code>operator[](int) const</code>	Returns a character with the specified index.
<code>~String()</code>	Destructor. Releases the memory occupied by the string.

**String() Member Function**

Constructor. Constructs an empty string.

**Syntax**

```
public String();
```

**String(const System.String&) Member Function**

Copy constructor.

**Syntax**

```
public String(const System.String& that);
```

**Parameters**

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

**operator=(const System.String&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.String& that);
```

**Parameters**

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

**2.2.55.3 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();
}
```

**String(System.String&&) Member Function**

Move constructor.

**Syntax**

```
public String(System.String&& that);
```

**Parameters**

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

**operator=(System.String&&)** Member Function

Move assignment.

**Syntax**

```
public void operator=(System.String&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.String&amp;&amp;</a>	A string to move from.

**Append(char) Member Function**

Appends the given character to the end of this string.

**Syntax**

```
public System.String& Append(char c);
```

**Parameters**

Name	Type	Description
c	char	A character to append.

**Returns**

[System.String&](#)

Returns a reference to this string.

**Append(char, int) Member Function**

Appends given number of occurrences of given character to the end of this string.

**Syntax**

```
public System.String& Append(char c, int count);
```

**Parameters**

Name	Type	Description
c	char	A character to append.
count	int	Number of times to append.

**Returns**

[System.String&](#)

Returns a reference to this string.

**Append(const System.String&) Member Function**

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

**Syntax**

```
public System.String& Append(const System.String& that);
```

**Parameters**

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

**Returns**

[System.String&](#)

Returns a reference to the this string.

**Append(const char\*) Member Function**

Appends the specified string to the end of this string.

**Syntax**

```
public System.String& Append(const char* that);
```

**Parameters**

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

**Returns**

[System.String&](#)

Returns a reference to the this string.

**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();
}
```

**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 System.String& 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.

**Returns**

[System.String&](#)

Returns a reference to the this string.

**Begin() Member Function**

Returns an iterator to the beginning of the string.

**Syntax**

```
public System.RandomAccessIter<char, char&, char*> Begin();
```

**Returns**

[System.RandomAccessIter<char, char&, char\\*>](#)

Returns an iterator to the beginning of the string.

**Begin() const Member Function**

Returns a constant iterator to the beginning of the string.

**Syntax**

```
public System.RandomAccessIter<char, const char&, const char*> Begin() const;
```

**Returns**

[System.RandomAccessIter<char, const char&, const char\\*>](#)

Returns a constant iterator to the beginning of the string.

**CBegin() const Member Function**

Returns a constant iterator to the beginning of the string.

**Syntax**

```
public System.RandomAccessIter<char, const char&, const char*> CBegin() const;
```

**Returns**

[System.RandomAccessIter<char, const char&, const char\\*>](#)

Returns a constant iterator to the beginning of the string.

**CEnd() const Member Function**

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

**Syntax**

```
public System.RandomAccessIter<char, const char&, const char*> CEnd() const;
```

**Returns**

[System.RandomAccessIter<char, const char&, const char\\*>](#)

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

**Capacity() const Member Function**

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

**Syntax**

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

**Returns**

int

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

**Chars() const Member Function**

Returns the string as a C-style string.

**Syntax**

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

**Returns**

const char\*

Returns the string as a C-style string.

**Clear() Member Function**

Makes the string empty.

**Syntax**

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

**End() Member Function**

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

**Syntax**

```
public System.RandomAccessIter<char, char&, char*> End();
```

**Returns**

[System.RandomAccessIter<char, char&, char\\*>](#)

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

**End() const Member Function**

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

**Syntax**

```
public System.RandomAccessIter<char, const char&, const char*> End() const;
```

**Returns**

[System.RandomAccessIter<char, const char&, const char\\*>](#)

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

**EndsWith(const System.String&) const Member Function**

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

**Syntax**

```
public bool EndsWith(const System.String& suffix) const;
```

**Parameters**

Name	Type	Description
suffix	const System.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();
}
```

**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 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();
}
```

**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 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();
}
```

**Find(const System.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 int Find(const System.String& s) const;
```

**Parameters**

Name	Type	Description
s	const System.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();
}
```

**Find(const System.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 int Find(const System.String& s, int start) const;
```

**Parameters**

Name	Type	Description
s	const System.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();
}
```

**IsEmpty() const Member Function**

Returns true if the string is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the string is empty, false otherwise.

**Length() const Member Function**

Returns the length of the string.

**Syntax**

```
public int Length() const;
```

**Returns**

int

Returns the length of the string.

**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 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();
}
```

**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 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();
}
```

**RFind(const System.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 int RFind(const System.String& s) const;
```

**Parameters**

Name	Type	Description
s	const System.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();
}
```

**RFind(const System.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 int RFind(const System.String& s, int start) const;
```

**Parameters**

Name	Type	Description
s	const System.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();
}
```

**Replace(char, char) Member Function**

Replaces every occurrence of the given character with another character.

**Syntax**

```
public 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();
}
```

**Reserve(int) Member Function**

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

**Syntax**

```
public void Reserve(int minLen);
```

**Parameters**

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

**Split(char) Member Function**

Returns a list of substrings separated by the given character.

**Syntax**

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

**Parameters**

Name	Type	Description
c	char	A separator character.

**Returns**

[System.Collections.List<System.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();
    }
}
```

**StartsWith(const System.String&) const Member Function**

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

**Syntax**

```
public bool StartsWith(const System.String& prefix) const;
```

**Parameters**

Name	Type	Description
prefix	const System.String&	A prefix to test.

**Returns**

bool

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

**String(char) Member Function**

Constructor. Constructs a string that has the given character.

**Syntax**

```
public String(char c);
```

**Parameters**

Name	Type	Description
c	char	A character.

**String(char, int) Member Function**

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

**Syntax**

```
public 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();
}
```

**String(const char\*) Member Function**

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

**Syntax**

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

**Parameters**

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

**String(const char\*, const char\*) Member Function**

Constructor. Constructs a string of characters included in range `begin` and `end`.

**Syntax**

```
public String(const char* begin, const char* end);
```

**Parameters**

Name	Type	Description
begin	const char*	Start of the character range.
end	const char*	One past the end of the character range.

**String(const char\*, int) Member Function**

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

**Syntax**

```
public 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();
}
```

**Substring(int) const Member Function**

Returns a substring starting from the given index.

**Syntax**

```
public System.String Substring(int start) const;
```

**Parameters**

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

**Returns**

[System.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();
}
```

**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 System.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**

[System.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();
}
```

**operator<(const System.String&) const Member Function**

Compares a string for less than relationship with another string.

**Syntax**

```
public bool operator<(const System.String& that) const;
```

**Parameters**

Name	Type	Description
that	const System.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.

**operator==(const System.String&) const Member Function**

Compares a string for equality with another string.

**Syntax**

```
public bool operator==(const System.String& that) const;
```

**Parameters**

Name	Type	Description
that	const System.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.

**operator[](int) Member Function**

Returns a reference to the character with the specified index.

**Syntax**

```
public 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 hand 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();
}
```

**operator[](int) const Member Function**

Returns a character with the specified index.

**Syntax**

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

**Parameters**

Name	Type	Description
index	int	An index.

**Returns**

char

Returns a character with the specified index.

**~String() Member Function**

Destructor. Releases the memory occupied by the string.

**Syntax**

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

**2.2.55.4 Nonmember Functions**

Function	Description
<code>operator+(const System.String&amp;, const System.String&amp;)</code>	Concatenates two strings.
<code>operator+(const System.String&amp;, const char*)</code>	Concatenates a string and a C-style string.
<code>operator+(const char*, const System.String&amp;)</code>	Concatenates a C-style string and a string.

**operator+(const System.String&, const System.String&) Function**

Concatenates two strings.

**Syntax**

```
public System.String operator+(const System.String& first, const System.String& second);
```

**Parameters**

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

**Returns**

[System.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();
}
```

**operator+(const System.String&, const char\*) Function**

Concatenates a string and a C-style string.

**Syntax**

```
public System.String operator+(const System.String& first, const char* second);
```

**Parameters**

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

**Returns**

[System.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();
}
```

**operator+(const char\*, const System.String&)** Function

Concatenates a C-style string and a string.

**Syntax**

```
public System.String operator+(const char* first, const System.String& second);
```

**Parameters**

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

**Returns**

[System.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();
}
```

## 2.2.56 TimeError Class

An exception thrown when a time function fails.

### Syntax

```
public class TimeError;
```

### Base Class

[System.Exception](#)

### 2.2.56.1 Member Functions

Member Function	Description
<code>TimeError()</code>	Default constructor.
<code>TimeError(const System.TimeError&amp;)</code>	Copy constructor.
<code>operator=(const System.TimeError&amp;)</code>	Copy assignment.
<code>TimeError(System.TimeError&amp;&amp;)</code>	Move constructor.
<code>operator=(System.TimeError&amp;&amp;)</code>	Move assignment.
<code>TimeError(const System.String&amp;, const System.String&amp;)</code>	Constructor. Initializes the time error with specified operation description and failure reason.
<code>~TimeError()</code>	Destructor.

**TimeError() Member Function**

Default constructor.

**Syntax**

```
public TimeError();
```

**TimeError(const System.TimeError&) Member Function**

Copy constructor.

**Syntax**

```
public TimeError(const System.TimeError& that);
```

**Parameters**

Name	Type	Description
that	const System.TimeError&	Argument to copy.

**operator=(const System.TimeError&)** Member Function

Copy assignment.

**Syntax**

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

**Parameters**

Name	Type	Description
that	const System.TimeError&	Argument to assign.

**TimeError(System.TimeError&&) Member Function**

Move constructor.

**Syntax**

```
public TimeError(System.TimeError&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.TimeError&amp;&amp;</a>	Argument to move from.

**operator=(System.TimeError&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.TimeError&& that);
```

**Parameters**

Name	Type	Description
that	System.TimeError&&	Argument to assign from.

**TimeError(const System.String&, const System.String&) Member Function**

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

**Syntax**

```
public TimeError(const System.String& operation, const System.String& reason);
```

**Parameters**

Name	Type	Description
operation	const System.String&	Description of operation.
reason	const System.String&	Failure reason.

**~TimeError() Member Function**

Destructor.

**Syntax**

```
public ~TimeError();
```

## 2.2.57 TimePoint Class

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

### Syntax

```
public class TimePoint;
```

#### 2.2.57.1 Remarks

Epoch is midnight 1.1.1970.

#### 2.2.57.2 Member Functions

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

**TimePoint() Member Function**

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

**Syntax**

```
public TimePoint();
```

**TimePoint(const System.TimePoint&) Member Function**

Copy constructor.

**Syntax**

```
public TimePoint(const System.TimePoint& that);
```

**Parameters**

Name	Type	Description
that	const System.TimePoint&	Argument to copy.

**operator=(const System.TimePoint&)** Member Function

Copy assignment.

**Syntax**

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

**Parameters**

Name	Type	Description
that	const System.TimePoint&	Argument to assign.

**TimePoint(System.TimePoint&&) Member Function**

Move constructor.

**Syntax**

```
public TimePoint(System.TimePoint&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.TimePoint&amp;&amp;</a>	Argument to move from.

**operator=(System.TimePoint&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.TimePoint&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.TimePoint&amp;&amp;</a>	Argument to assign from.

**Rep() const Member Function**

Returns the number of nanoseconds elapsed since epoch.

**Syntax**

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

**Returns**

long

Returns the number of nanoseconds elapsed since epoch.

**TimePoint(long) Member Function**

Constructor. Initializes the time point with the given number of nanoseconds elapsed since epoch.

**Syntax**

```
public TimePoint(long nanosecs_);
```

**Parameters**

Name	Type	Description
nanosecs_	long	Number of nanoseconds.

**2.2.57.3 Nonmember Functions**

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

**operator+(System.TimePoint, System.Duration) Function**

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

**Syntax**

```
public System.TimePoint operator+(System.TimePoint tp, System.Duration d);
```

**Parameters**

Name	Type	Description
tp	<a href="#">System.TimePoint</a>	A time point.
d	<a href="#">System.Duration</a>	A duration.

**Returns**

[System.TimePoint](#)

Returns tp+d.

**operator-(System.TimePoint, System.Duration) Function**

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

**Syntax**

```
public System.TimePoint operator-(System.TimePoint tp, System.Duration d);
```

**Parameters**

Name	Type	Description
tp	<a href="#">System.TimePoint</a>	A time point.
d	<a href="#">System.Duration</a>	A duration.

**Returns**

[System.TimePoint](#)

Returns *TimePoint*(*tp* – *d*).

**operator-(System.TimePoint, System.TimePoint) Function**

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

**Syntax**

```
public System.Duration operator-(System.TimePoint left, System.TimePoint right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.TimePoint</a>	The first time point.
right	<a href="#">System.TimePoint</a>	The second time point.

**Returns**

[System.Duration](#)

Returns  $Duration(left - right)$ .

**operator<(System.TimePoint, System.TimePoint) Function**

Compares two time points for less than relationship.

**Syntax**

```
public bool operator<(System.TimePoint left, System.TimePoint right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.TimePoint</a>	The first time point.
right	<a href="#">System.TimePoint</a>	The second time point.

**Returns**

bool

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

**operator==(System.TimePoint, System.TimePoint) Function**

Compares two time points for equality.

**Syntax**

```
public bool operator==(System.TimePoint left, System.TimePoint right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.TimePoint</a>	The first time point.
right	<a href="#">System.TimePoint</a>	The second time point.

**Returns**

bool

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

## 2.2.58 TracedFun Class

Function tracer.

### Syntax

```
public class TracedFun;
```

#### 2.2.58.1 Member Functions

Member Function	Description
<a href="#">TracedFun()</a>	Default constructor.
<a href="#">TracedFun(const char*, const char*, int)</a>	Writes > followed by the given function name, file and line to standard error stream.
<a href="#">~TracedFun()</a>	Writes < followed by the function name, file and line to standard error stream.

**TracedFun() Member Function**

Default constructor.

**Syntax**

```
public TracedFun();
```

**TracedFun(const char\*, const char\*, int) Member Function**

Writes > followed by the given function name, file and line to standard error stream.

**Syntax**

```
public TracedFun(const char* fun_, const char* file_, int line_);
```

**Parameters**

Name	Type	Description
fun_	const char*	A name of a function.
file_	const char*	Source file name.
line_	int	Source line number.

**~TracedFun() Member Function**

Writes < followed by the function name, file and line to standard error stream.

**Syntax**

```
public ~TracedFun();
```

## 2.2.59 Tracer Class

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

### Syntax

```
public class Tracer;
```

#### 2.2.59.1 Member Functions

Member Function	Description
<a href="#">Tracer()</a>	Default constructor.
<a href="#">Tracer(const System.String&amp;)</a>	Constructor. Writes the specified string to standard error stream.
<a href="#">~Tracer()</a>	Destructor. Writes ~ and a string specified in constructor to standard error stream.

**Tracer() Member Function**

Default constructor.

**Syntax**

```
public Tracer();
```

**Tracer(const System.String&) Member Function**

Constructor. Writes the specified string to standard error stream.

**Syntax**

```
public Tracer(const System.String& s_);
```

**Parameters**

Name	Type	Description
s_	const System.String&	String to write.

**~Tracer() Member Function**

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

**Syntax**

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

## 2.2.60 UnaryFun<Argument, Result> Class

A base class for unary function objects.

### Syntax

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

### Constraint

Argument is [Semiregular](#)

#### 2.2.60.1 Remarks

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

### 2.2.60.2 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.

### 2.2.60.3 Member Functions

Member Function	Description
<code>UnaryFun&lt;Argument, Result&gt;()</code>	Default constructor.
<code>UnaryFun&lt;Argument, Result&gt;(const System.-&gt;UnaryFun&lt;Argument, Result&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.UnaryFun&lt;Argument, Result&gt;&amp;)</code>	Copy assignment.
<code>UnaryFun&lt;Argument, Result&gt;(System.-&gt;UnaryFun&lt;Argument, Result&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.UnaryFun&lt;Argument, Result&gt;&amp;&amp;)</code>	Move assignment.

**UnaryFun<Argument, Result>() Member Function**

Default constructor.

**Syntax**

```
public UnaryFun<Argument, Result>();
```

**UnaryFun<Argument, Result>(const System.UnaryFun<Argument, Result>&) Member Function**

Copy constructor.

**Syntax**

```
public UnaryFun<Argument, Result>(const System.UnaryFun<Argument, Result>& that);
```

**Parameters**

Name	Type	Description
that	const System.UnaryFun<Argument, Result>&	Argument to copy.

**operator=(const System.UnaryFun<Argument, Result>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.UnaryFun<Argument, Result>& that);
```

**Parameters**

Name	Type	Description
that	const System.UnaryFun<Argument, Result>&	Argument to assign.

**UnaryFun<Argument, Result>(System.UnaryFun<Argument, Result>&&) Member Function**

Move constructor.

**Syntax**

```
public UnaryFun<Argument, Result>(System.UnaryFun<Argument, Result>&& that);
```

**Parameters**

Name	Type	Description
that	System.UnaryFun<Argument, Result>&&	Argument to move from.

**operator=(System.UnaryFun<Argument, Result>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.UnaryFun<Argument, Result>&& that);
```

**Parameters**

Name	Type	Description
that	System.UnaryFun<Argument, Result>&&	Argument to assign from.

## 2.2.61 UnaryPred<Argument> Class

A base class for unary predicates.

### Syntax

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

### Constraint

Argument is [Semiregular](#)

### Model of

[UnaryPredicate<T>](#)

### Base Class

[System.UnaryFun<Argument, bool>](#)

#### 2.2.61.1 Remarks

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

#### 2.2.61.2 Member Functions

Member Function	Description
<a href="#">UnaryPred&lt;Argument&gt;()</a>	Default constructor.
<a href="#">UnaryPred&lt;Argument&gt;(const System.-&gt;UnaryPred&lt;Argument&gt;&amp;)</a>	Copy constructor.
<a href="#">operator=(const System.UnaryPred&lt;-&gt;Argument&gt;&amp;)</a>	Copy assignment.
<a href="#">UnaryPred&lt;Argument&gt;(System.UnaryPred&lt;-&gt;Argument&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.UnaryPred&lt;Argument&gt;&amp;&amp;)</a>	Move assignment.

**UnaryPred<Argument>() Member Function**

Default constructor.

**Syntax**

```
public UnaryPred<Argument>();
```

**UnaryPred<Argument>(const System.UnaryPred<Argument>&) Member Function**

Copy constructor.

**Syntax**

```
public UnaryPred<Argument>(const System.UnaryPred<Argument>& that);
```

**Parameters**

Name	Type	Description
that	const System.UnaryPred<Argument>&	Argument to copy.

**operator=(const System.UnaryPred<Argument>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.UnaryPred<Argument>& that);
```

**Parameters**

Name	Type	Description
that	const System.UnaryPred<Argument>&	Argument to assign.

**UnaryPred<Argument>(System.UnaryPred<Argument>&&) Member Function**

Move constructor.

**Syntax**

```
public UnaryPred<Argument>(System.UnaryPred<Argument>&& that);
```

**Parameters**

Name	Type	Description
that	System.UnaryPred<Argument>&&	Argument to move from.

**operator=(System.UnaryPred<Argument>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.UnaryPred<Argument>&& that);
```

**Parameters**

Name	Type	Description
that	System.UnaryPred<Argument>&&	Argument to assign from.

## 2.2.62 UniquePtr<T> Class

A unique pointer to an object.

### Syntax

```
public class UniquePtr<T>;
```

### Model of

[DefaultConstructible<T>](#)

[Movable<T>](#)

### 2.2.62.1 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.

### 2.2.62.2 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();
    }
}
```

### 2.2.62.3 Member Functions

Member Function	Description
<a href="#">UniquePtr&lt;T&gt;()</a>	Constructor. Constructs a null unique pointer.
<a href="#">UniquePtr&lt;T&gt;(System.UniquePtr&lt;T&gt;&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.UniquePtr&lt;T&gt;&amp;&amp;)</a>	Move assignment.
<a href="#">GetPtr() const</a>	Returns the contained pointer to the owned object.

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

**UniquePtr<T>() Member Function**

Constructor. Constructs a null unique pointer.

**Syntax**

```
public UniquePtr<T>();
```

**UniquePtr<T>(System.UniquePtr<T>&&) Member Function**

Move constructor.

**Syntax**

```
public UniquePtr<T>(System.UniquePtr<T>&& that);
```

**Parameters**

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

**operator=(System.UniquePtr<T>&&) Member Function**

Move assignment.

**Syntax**

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

**Parameters**

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

**GetPtr() const Member Function**

Returns the contained pointer to the owned object.

**Syntax**

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

**Returns**

T\*

Returns the contained pointer to the owned object.

**IsNull() const Member Function**

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

**Syntax**

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

**Returns**

bool

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

**Release() Member Function**

Releases the ownership of the owned object and sets the unique pointer to null.

**Syntax**

```
public T* Release();
```

**Returns**

T\*

Returns the contained pointer.

**Reset() Member Function**

Resets the unique ptr to null.

**Syntax**

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

**Remarks**

Destroys the owned object.

**Reset(T\*) Member Function**

Resets the contained pointer to point to a new object.

**Syntax**

```
public 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.

**Swap(System.UniquePtr<T>&) Member Function**

Exchanges the contents of the unique pointer with another unique pointer.

**Syntax**

```
public void Swap(System.UniquePtr<T>& that);
```

**Parameters**

Name	Type	Description
that	System.UniquePtr<T>&	A unique pointer to exchange contents with.

**UniquePtr<T>(T\*) Member Function**

Constructor. Constructs a unique pointer to the given object.

**Syntax**

```
public UniquePtr<T>(T* ptr_);
```

**Parameters**

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

**operator\*() const Member Function**

Returns a reference to the pointed object.

**Syntax**

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

**Returns**

T&

Returns a reference to the pointed object.

**operator->() const Member Function**

Returns the contained pointer to the pointed object.

**Syntax**

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

**Returns**

T\*

Returns the contained pointer to the pointed object.

**operator=(T\*) Member Function**

Assigns a new object to the unique pointer.

**Syntax**

```
public 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.

**~UniquePtr<T>() Member Function**

Destructor. Destroys the owned object.

**Syntax**

```
public ~UniquePtr<T>();
```

**2.2.62.4 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.UniquePtr&lt;T&gt;&amp;, const System.UniquePtr&lt;T&gt;&amp;)</code>	Compares two unique pointers for less than relationship.
<code>operator==(const System.UniquePtr&lt;T&gt;&amp;, const System.UniquePtr&lt;T&gt;&amp;)</code>	Compares two unique pointers for equality.

**operator<(const System.UniquePtr<T>&, const System.UniquePtr<T>&) Function**

Compares two unique pointers for less than relationship.

**Syntax**

```
public bool operator<(const System.UniquePtr<T>& left, const System.UniquePtr<T>& right);
```

**Parameters**

Name	Type	Description
left	const System.UniquePtr<T>&	The first unique pointer.
right	const System.UniquePtr<T>&	The second unique pointer.

**Returns**

bool

Returns true if the pointer contained by the first unique pointer is less than the pointer contained by the second unique pointer, false otherwise.

**operator==(const System.UniquePtr<T>&, const System.UniquePtr<T>&) Function**

Compares two unique pointers for equality.

**Syntax**

```
public bool operator==(const System.UniquePtr<T>& left, const System.UniquePtr<T>& right);
```

**Parameters**

Name	Type	Description
left	const System.UniquePtr<T>&	The first unique pointer.
right	const System.UniquePtr<T>&	The second unique pointer.

**Returns**

bool

Returns true if the pointer contained by the first unique pointer is equal to the pointer contained by the second unique pointer.

## 2.2.63 WeakCount<T> Class

A handle to a [Counter<T>](#) that maintains the weak count portion of the counter.

### Syntax

```
public class WeakCount<T>;
```

#### 2.2.63.1 Member Functions

Member Function	Description
<code>WeakCount&lt;T&gt;()</code>	Constructor. Initializes an empty weak count.
<code>WeakCount&lt;T&gt;(const System.WeakCount&lt;T&gt;&amp;)</code>	Copy constructor. Increments weak count.
<code>WeakCount&lt;T&gt;(const System.WeakCount&lt;T&gt;&amp;)</code>	Copy constructor. Increments weak count.
<code>operator=(const System.WeakCount&lt;T&gt;&amp;)</code>	Copy assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.
<code>operator=(const System.WeakCount&lt;T&gt;&amp;)</code>	Copy assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.
<code>GetCounter() const</code>	Returns the contained pointer to a counter.
<code>GetUseCount() const</code>	Returns the use count.
<code>Swap(System.WeakCount&lt;T&gt;&amp;)</code>	Exchanges the contents of the weak count with another weak count.
<code>WeakCount&lt;T&gt;(const System.SharedCount&lt;T&gt;&amp;)</code>	Constructor. Constructs a weak count from a shared count.
<code>operator=(const System.SharedCount&lt;T&gt;&amp;)</code>	Assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.
<code>~WeakCount&lt;T&gt;()</code>	Destructor. Decrements the weak count.

**WeakCount<T>() Member Function**

Constructor. Initializes an empty weak count.

**Syntax**

```
public WeakCount<T>();
```

**WeakCount<T>(const System.WeakCount<T>&) Member Function**

Copy constructor. Increments weak count.

**Syntax**

```
public WeakCount<T>(const System.WeakCount<T>& that);
```

**Parameters**

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

**WeakCount<T>(const System.WeakCount<T>&) Member Function**

Copy constructor. Increments weak count.

**Syntax**

```
public WeakCount<T>(const System.WeakCount<T>& that);
```

**Parameters**

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

**operator=(const System.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 void operator=(const System.WeakCount<T>& that);
```

**Parameters**

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

**operator=(const System.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 void operator=(const System.WeakCount<T>& that);
```

**Parameters**

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

**GetCounter() const Member Function**

Returns the contained pointer to a counter.

**Syntax**

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

**Returns**

[System.Counter<T>\\*](#)

Returns the contained pointer to a counter.

**GetUseCount() const Member Function**

Returns the use count.

**Syntax**

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

**Returns**

int

Returns the use count.

**Swap(System.WeakCount<T>&) Member Function**

Exchanges the contents of the weak count with another weak count.

**Syntax**

```
public void Swap(System.WeakCount<T>& that);
```

**Parameters**

Name	Type	Description
that	System.WeakCount<T>&	A weak count to exchange contents with.

**WeakCount<T>(const System.SharedCount<T>&) Member Function**

Constructor. Constructs a weak count from a shared count.

**Syntax**

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

**Parameters**

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

**operator=(const System.SharedCount<T>&) Member Function**

Assignment. Decrements the weak count of the old counter and increments the weak count of the copied counter.

**Syntax**

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

**Parameters**

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

**~WeakCount<T>() Member Function**

Destructor. Decrements the weak count.

**Syntax**

```
public ~WeakCount<T>();
```

**2.2.63.2 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.WeakCount&lt;T&gt;&amp;, const System.WeakCount&lt;T&gt;&amp;)</code>	Compares two weak counts for less than relationship.
<code>operator==(const System.WeakCount&lt;T&gt;&amp;, const System.WeakCount&lt;T&gt;&amp;)</code>	Compares two weak counts for equality.

**operator<(const System.WeakCount<T>&, const System.WeakCount<T>&) Function**

Compares two weak counts for less than relationship.

**Syntax**

```
public bool operator<(const System.WeakCount<T>& left, const System.WeakCount<T>& right);
```

**Parameters**

Name	Type	Description
left	const System.WeakCount<T>&	The first weak count.
right	const System.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 address of the counter contained by the *right* count, false otherwise.

**operator==(const System.WeakCount<T>&, const System.WeakCount<T>&) Function**

Compares two weak counts for equality.

**Syntax**

```
public bool operator==(const System.WeakCount<T>& left, const System.WeakCount<T>& right);
```

**Parameters**

Name	Type	Description
left	const System.WeakCount<T>&	The first weak count.
right	const System.WeakCount<T>&	The second weak count.

**Returns**

bool

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

## 2.2.64 WeakPtr<T> Class

Used to break cycles in shared ownership.

### Syntax

```
public class WeakPtr<T>;
```

#### 2.2.64.1 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.

#### 2.2.64.2 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();
}
}

```

### 2.2.64.3 Member Functions

Member Function	Description
<code>WeakPtr&lt;T&gt;()</code>	Constructor. Constructs null weak pointer.
<code>WeakPtr&lt;T&gt;(const System.WeakPtr&lt;T&gt;&amp;)</code>	Copy constructor. Increments weak count.
<code>WeakPtr&lt;T&gt;(const System.WeakPtr&lt;T&gt;&amp;)</code>	Copy constructor. Increments weak count.
<code>operator=(const System.WeakPtr&lt;T&gt;&amp;)</code>	Copy assignment.
<code>operator=(const System.WeakPtr&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Assign(T*, const System.SharedCount&lt;T&gt;&amp;)</code>	Implementation detail to support System.-EnableSharedFromThis.T.U.System.-ShareableFromThis.T.ptr.U.ptr.System.-SharedCount.U.const.ref function.
<code>GetCount() const</code>	Returns the weak count.
<code>GetPtr() const</code>	Returns a pointer to the counted object.
<code>GetUseCount() const</code>	Returns the use count of the counted object.
<code>IsExpired() const</code>	Returns true if the use count has gone to zero and the counted object has been destroyed.
<code>Lock() const</code>	If the weak pointer has not been expired, returns a shared pointer to the counted object; otherwise returns null shared pointer.
<code>Reset()</code>	Resets the weak pointer to null.
<code>Swap(System.WeakPtr&lt;T&gt;&amp;)</code>	Exchanges the contents of this weak pointer with another.
<code>WeakPtr&lt;T&gt;(const System.SharedPtr&lt;T&gt;&amp;)</code>	Constructor. Constructs a weak pointer from a shared pointer.
<code>operator=(const System.SharedPtr&lt;T&gt;&amp;)</code>	Assignment. Assigns a weak pointer from a shared pointer.

`~WeakPtr<T>()`

Destructor. Decrements weak count.

**WeakPtr<T>() Member Function**

Constructor. Constructs null weak pointer.

**Syntax**

```
public WeakPtr<T>();
```

**WeakPtr<T>(const System::WeakPtr<T>&) Member Function**

Copy constructor. Increments weak count.

**Syntax**

```
public WeakPtr<T>(const System::WeakPtr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System::WeakPtr<T>&	A weak pointer to copy.

**WeakPtr<T>(const System::WeakPtr<T>&) Member Function**

Copy constructor. Increments weak count.

**Syntax**

```
public WeakPtr<T>(const System::WeakPtr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System::WeakPtr<T>&	A weak pointer to copy.

**operator=(const System.WeakPtr<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.WeakPtr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.WeakPtr<T>&	A weak pointer to assign.

**operator=(const System.WeakPtr<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.WeakPtr<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.WeakPtr<T>&	A weak pointer to assign.

**Assign(T\*, const System.SharedCount<T>&) Member Function**

Implementation detail to support System.EnableSharedFromThis.T.U.System.-ShareableFromThis.T.ptr.U.ptr.System.SharedCount.U.const.ref function.

**Syntax**

```
public void Assign(T* ptr_, const System.SharedCount<T>& count_);
```

**Parameters**

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

**GetCount() const Member Function**

Returns the weak count.

**Syntax**

```
public const System.WeakCount<T>& GetCount() const;
```

**Returns**

```
const System.WeakCount<T>&
```

Returns the weak count.

**GetPtr() const Member Function**

Returns a pointer to the counted object.

**Syntax**

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

**Returns**

T\*

Returns a pointer to the counted object.

**GetUseCount() const Member Function**

Returns the use count of the counted object.

**Syntax**

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

**Returns**

int

Returns the use count of the counted object.

**IsExpired() const Member Function**

Returns true if the use count has gone to zero and the counted object has been destroyed.

**Syntax**

```
public bool IsExpired() const;
```

**Returns**

bool

Returns true if the use count has gone to zero and the counted object has been destroyed.

**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 SystemSharedPtr<T> Lock() const;
```

**Returns**

[SystemSharedPtr<T>](#)

If the weak pointer has not been expired, returns a shared pointer to the counted object; otherwise returns null shared pointer.

**Reset() Member Function**

Resets the weak pointer to null.

**Syntax**

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

**Swap(System.WeakPtr<T>&) Member Function**

Exchanges the contents of this weak pointer with another.

**Syntax**

```
public void Swap(System.WeakPtr<T>& that);
```

**Parameters**

Name	Type	Description
that	System.WeakPtr<T>&	A weak pointer to exchange contents with.

**WeakPtr<T>(const SystemSharedPtr<T>&) Member Function**

Constructor. Constructs a weak pointer from a shared pointer.

**Syntax**

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

**Parameters**

Name	Type	Description
that	const SystemSharedPtr<T>&	A shared pointer.

**operator=(const SystemSharedPtr<T>&)** Member Function

Assignment. Assigns a weak pointer from a shared pointer.

**Syntax**

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

**Parameters**

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

**`~WeakPtr<T>()` Member Function**

Destructor. Decrements weak count.

**Syntax**

```
public ~WeakPtr<T>();
```

## 2.2.65 uhuge Class

128-bit unsigned integer type.

### Syntax

```
public class uhuge;
```

#### 2.2.65.1 Member Functions

Member Function	Description
<code>uhuge()</code>	Constructor. Initializes the value to zero.
<code>uhuge(const System.uhuge&amp;)</code>	Copy constructor.
<code>operator=(const System.uhuge&amp;)</code>	Copy assignment.
<code>uhuge(System.uhuge&amp;&amp;)</code>	Move constructor.
<code>operator=(System.uhuge&amp;&amp;)</code>	Move assignment.
<code>operator++()</code>	Increments the value by one.
<code>operator--()</code>	Decrements the value by one.
<code>uhuge(ulong)</code>	Constructor. Initializes the value to specified 64-bit value.
<code>uhuge(ulong, ulong)</code>	Constructor. Initializes the value to $2^{64}h\_ + l\_$ .

**uhuge() Member Function**

Constructor. Initializes the value to zero.

**Syntax**

```
public uhuge();
```

**uhuge(const System.uhuge&) Member Function**

Copy constructor.

**Syntax**

```
public uhuge(const System.uhuge& that);
```

**Parameters**

Name	Type	Description
that	const System.uhuge&	Argument to copy.

**operator=(const System.uhuge&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.uhuge& that);
```

**Parameters**

Name	Type	Description
that	const System.uhuge&	Argument to assign.

**uhuge(System.uhuge&&) Member Function**

Move constructor.

**Syntax**

```
public uhuge(System.uhuge&& that);
```

**Parameters**

Name	Type	Description
that	System.uhuge&&	Argument to move from.

**operator=(System.uhuge&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.uhuge&& that);
```

**Parameters**

Name	Type	Description
that	System.uhuge&&	Argument to assign from.

**operator++() Member Function**

Increments the value by one.

**Syntax**

```
public System.uhuge& operator++();
```

**Returns**

[System.uhuge&](#)

Returns the value.

**operator-() Member Function**

Decrements the value by one.

**Syntax**

```
public System.uhuge& operator--();
```

**Returns**

[System.uhuge&](#)

Returns the value.

**uhuge(ulong) Member Function**

Constructor. Initializes the value to specified 64-bit value.

**Syntax**

```
public uhuge(ulong l_);
```

**Parameters**

Name	Type	Description
l_	ulong	An unsigned 64-bit value.

**uhuge(ulong, ulong) Member Function**

Constructor. Initializes the value to  $2^{64}h\_ + l\_.$

**Syntax**

```
public uhuge(ulong h_, ulong l_);
```

**Parameters**

Name	Type	Description
h_	ulong	High part.
l_	ulong	Low part.

**2.2.65.2 Nonmember Functions**

Function	Description
<a href="#">divmod(System.uhuge, System.uhuge)</a>	Computes quotient and remainder of division of two 128-bit values.
<a href="#">divmod(System.uhuge, uint)</a>	Computes quotient and remainder of division of a 128-bit value divided by a 32-bit value.
<a href="#">mul(System.uhuge, System.uhuge)</a>	Computes the 256-bit product of two 128-bit values.
<a href="#">operator%(System.uhuge, System.uhuge)</a>	Computes the remainder of division of two 128-bit values.
<a href="#">operator&amp;(System.uhuge, System.uhuge)</a>	Bitwise-AND operation of two 128-bit values.
<a href="#">operator*(System.uhuge, System.uhuge)</a>	Computes the product of two 128-bit values.
<a href="#">operator+(System.uhuge, System.uhuge)</a>	Computes the sum of two 128-bit values.
<a href="#">operator-(System.uhuge, System.uhuge)</a>	Subtracts a 128-bit value from 128-bit value.
<a href="#">operator/(System.uhuge, System.uhuge)</a>	Division of two 128-bit values.
<a href="#">operator&lt;(System.uhuge, System.uhuge)</a>	Compares two 128-bit values for less than relationship.
<a href="#">operator&lt;&lt;(System.uhuge, System.uhuge)</a>	Shifts the given <b>uhuge</b> value to the given number of bit positions left and returns it.
<a href="#">operator==(System.uhuge, System.uhuge)</a>	Compares two 128-bit values for equality.
<a href="#">operator&gt;&gt;(System.uhuge, System.uhuge)</a>	Shifts the given <b>uhuge</b> value to given bit positions right and returns it.

`operator^(System.uhuge, System.uhuge)`

Computes the bitwise-XOR operation of two 128-bit values.

`operator—(System.uhuge, System.uhuge)`

Computes bitwise-OR of two 128-bit values.

`operator~(System.uhuge)`

Computes bitwise complement of a 128-bit value.

**divmod(System.uhuge, System.uhuge) Function**

Computes quotient and remainder of division of two 128-bit values.

**Syntax**

```
public System.Pair<System.uhuge, System.uhuge> divmod(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	Dividend.
right	<a href="#">System.uhuge</a>	Divisor.

**Returns**

[System.Pair<System.uhuge, System.uhuge>](#)

Returns a pair in which the first value is the quotient and the second value is the remainder of *left/right*.

**divmod(System.uhuge, uint) Function**

Computes quotient and remainder of division of a 128-bit value divided by a 32-bit value.

**Syntax**

```
public System.Pair<System.uhuge, uint> divmod(System.uhuge left, uint right);
```

**Parameters**

Name	Type	Description
left	System.uhuge	Dividend.
right	uint	Divisor.

**Returns**

[System.Pair<System.uhuge, uint>](#)

Returns a pair in which the first value is the quotient and the second value is the remainder of *left/right*.

**mul(System.uhuge, System.uhuge) Function**

Computes the 256-bit product of two 128-bit values.

**Syntax**

```
public System.Pair<System.uhuge, System.uhuge> mul(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.Pair<System.uhuge, System.uhuge>](#)

Returns a pair in which the first value is the high part and the second value is the low part of of  $left \times right$ .

**operator%(System.uhuge, System.uhuge) Function**

Computes the remainder of division of two 128-bit values.

**Syntax**

```
public System.uhuge operator%(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	Dividend.
right	<a href="#">System.uhuge</a>	Divisor.

**Returns**

[System.uhuge](#)

Returns  $left \% right$ .

**operator&(System.uhuge, System.uhuge) Function**

Bitwise-AND operation of two 128-bit values.

**Syntax**

```
public System.uhuge operator&(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	System.uhuge	The first 128-bit value.
right	System.uhuge	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns bitwise AND of *left* and *right*.

**operator\*(System.uhuge, System.uhuge) Function**

Computes the product of two 128-bit values.

**Syntax**

```
public System.uhuge operator*(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	Ths first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns *left*  $\times$  *right*.

**operator+(System.uhuge, System.uhuge) Function**

Computes the sum of two 128-bit values.

**Syntax**

```
public System.uhuge operator+(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns *left* + *right*.

**operator-(System.uhuge, System.uhuge) Function**

Subtracts a 128-bit value from 128-bit value.

**Syntax**

```
public System.uhuge operator-(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns  $left - right$ .

**operator/(System.uhuge, System.uhuge) Function**

Division of two 128-bit values.

**Syntax**

```
public System.uhuge operator/(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	System.uhuge	Dividend.
right	System.uhuge	Divisor.

**Returns**

[System.uhuge](#)

Returns *left* divided by *right*.

**operator<(System.uhuge, System.uhuge) Function**

Compares two 128-bit values for less than relationship.

**Syntax**

```
public bool operator<(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	System.uhuge	The first 128-bit value.
right	System.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.

**operator<<(System.uhuge, System.uhuge) Function**

Shifts the given [uhuge](#) value to the given number of bit positions left and returns it.

**Syntax**

```
public System.uhuge operator<<(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The uhuge value to shift.
right	<a href="#">System.uhuge</a>	The number of bit positions to shift.

**Returns**

[System.uhuge](#)

Returns an uhuge value shifted given number of bit positions left.

**operator==(System.uhuge, System.uhuge) Function**

Compares two 128-bit values for equality.

**Syntax**

```
public bool operator==(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	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.

**operator>>(System.uhuge, System.uhuge) Function**

Shifts the given [uhuge](#) value to given bit positions right and returns it.

**Syntax**

```
public System.uhuge operator>>(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	An uhuge value to shift.
right	<a href="#">System.uhuge</a>	Number of bit positions to shift.

**Returns**

[System.uhuge](#)

Returns the given uhuge value shifted given number of bit positions right.

**operator^(System.uhuge, System.uhuge) Function**

Computes the bitwise-XOR operation of two 128-bit values.

**Syntax**

```
public System.uhuge operator^(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns *left* XOR *right*.

**operator|(System.uhuge, System.uhuge) Function**

Computes bitwise-OR of two 128-bit values.

**Syntax**

```
public System.uhuge operator|(System.uhuge left, System.uhuge right);
```

**Parameters**

Name	Type	Description
left	<a href="#">System.uhuge</a>	The first 128-bit value.
right	<a href="#">System.uhuge</a>	The second 128-bit value.

**Returns**

[System.uhuge](#)

Returns bitwise-OR of *left* and *right*.

**operator~(System.uhuge) Function**

Computes bitwise complement of a 128-bit value.

**Syntax**

```
public System.uhuge operator~(System.uhuge x);
```

**Parameters**

Name	Type	Description
x	System.uhuge	A 128-bit value.

**Returns**

[System.uhuge](#)

Returns bitwise complement of *x*.

## 2.3 Type Definitions

Name	Type	Description
string	System.String	An alias for <a href="#">String</a> class.

## 2.4 Functions

Function	Description
<code>Abs(const T&amp;)</code>	Returns the absolute value of the argument.
<code>Accumulate(I, I, T, Op)</code>	Accumulates a sequence with respect to a binary operation.
<code>BackInserter(C&amp;)</code>	Returns a <code>BackInsertIterator&lt;C&gt;</code> for a back insertion sequence.
<code>CaptureCurrentException()</code>	This function can be used in exception handlers (catch clauses) to capture the current exception with full type information to an <code>ExceptionPtr</code> . This is useful for example in multithreaded programs where exception thrown inside a thread function can be transferred to the main thread and rethrown using <code>RethrowException(System::ExceptionPtr&amp;)</code> function from there.
<code>Copy(I, I, O)</code>	Copies a sequence.
<code>CopyBackward(I, I, O)</code>	Copies a source sequence to a target sequence starting from the end of the source sequence.
<code>Count(I, I, P)</code>	Counts the number of elements in a sequence that satisfy a predicate.
<code>Count(I, I, const T&amp;)</code>	Counts the number of elements in a sequence that are equal to the given value.
<code>GetCurrentDate()</code>	Returns current date as <code>Date</code> value.
<code>Distance(I, I)</code>	Returns the distance between two random access iterators.
<code>Distance(I, I)</code>	Returns the distance between two forward iterators.
<code>EnableSharedFromThis(System::ShareableFromThis&lt;T&gt;*, U*, const System::SharedCount&lt;U&gt;&amp;)</code>	A function that enables the <i>shared from this</i> idiom. Implementation detail.
<code>EnableSharedFromThis(void*, void*, const System::SharedCount&lt;T&gt;&amp;)</code>	An empty function that catches classes that are not shareable from this. Implementation detail.
<code>Equal(I1, I1, I2, I2)</code>	Compares two sequences for equality.
<code>Equal(I1, I1, I2, I2, R)</code>	Compares two sequences for equality using the given equality relation.

<code>EqualRange(I, I, const T&amp;)</code>	Returns a pair of iterators that form a range of values equal to the given value in a sorted sequence.
<code>EqualRange(I, I, const T&amp;, R)</code>	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.
<code>Factorial(U)</code>	Returns a factorial of the argument.
<code>Fill(I, I, const T&amp;)</code>	Fills a sequence with the given value.
<code>Find(I, I, P)</code>	Searches the first occurrence of a value from a sequence that matches a predicate.
<code>Find(I, I, const T&amp;)</code>	Searches a value from a sequence.
<code>ForEach(I, I, F)</code>	Applies a function object for each element of a sequence.
<code>FrontInserter(C&amp;)</code>	Returns a <code>FrontInsertIterator&lt;C&gt;</code> for a front insert sequence.
<code>Gcd(T, T)</code>	Returns the greatest common divisor of two values.
<code>HexChar(byte)</code>	Returns hexadecimal character representation of a four-bit value.
<code>IdentityElement(System.Multiplies&lt;T&gt;)</code>	Returns the identity element of multiplication, that is $T(1)$ .
<code>IdentityElement(System.Plus&lt;T&gt;)</code>	Returns the identity element of addition, that is: $T(0)$ .
<code>Inserter(C&amp;, I)</code>	Returns an <code>InsertIterator&lt;C&gt;</code> for an insertion sequence and its iterator.
<code>InsertionSort(I, I)</code>	Sorts a sequence of values using insertion sort algorithm.
<code>InsertionSort(I, I, R)</code>	Sorts a sequence of values using insertion sort algorithm and given ordering relation.
<code>IsAlpha(char)</code>	Returns true if the given character is an alphabetic character, false otherwise.

<code>IsAlphanumeric(char)</code>	Returns true if the given character is an alphanumeric character, false otherwise.
<code>IsControl(char)</code>	Returns true if the given character is a control character, false otherwise.
<code>IsDigit(char)</code>	Returns true if the given character is a decimal digit, false otherwise.
<code>IsGraphic(char)</code>	Returns true if the given character is a graphical character, false otherwise.
<code>IsHexDigit(char)</code>	Returns true if the given character is a hexadecimal digit, false otherwise.
<code>IsLower(char)</code>	Returns true if the given character is a lower case letter, false otherwise.
<code>IsPrintable(char)</code>	Returns true if the given character is a printable character, false otherwise.
<code>IsPunctuation(char)</code>	Returns true if the given character is a punctuation character, false otherwise.
<code>IsSpace(char)</code>	Returns true if the given character is a space character, false otherwise.
<code>IsUpper(char)</code>	Returns true if the given character is an upper case letter, false otherwise.
<code>LastComponentsEqual(const System.String&amp;, const System.String&amp;, char)</code>	Returns true, if last components of two strings are equal, false otherwise.
<code>LexicographicalCompare(I1, I1, I2, I2)</code>	Returns true if the first sequence comes lexicographically before the second sequence, false otherwise.
<code>LexicographicalCompare(I1, I1, I2, I2, R)</code>	Returns true if the first sequence comes lexicographically before the second sequence according to the given ordering relation, false otherwise.
<code>LowerBound(I, I, const T&amp;)</code>	Finds a position of the first element in a sorted sequence that is greater than or equal to the given value.

<code>LowerBound(I, I, const T&amp;, R)</code>	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.
<code>MakePair(const T&amp;, const U&amp;)</code>	Returns a pair composed of the given values.
<code>Max(const T&amp;, const T&amp;)</code>	Returns the maximum of two values.
<code>MaxElement(I, I)</code>	Returns the position of the first occurrence of the largest element in a sequence of elements.
<code>MaxElement(I, I, R)</code>	Returns the position of the first occurrence of the largest element according to the given ordering relation in a sequence of elements.
<code>.MaxValue()</code>	Returns the largest value of an integer type.
<code>.MaxValue(byte)</code>	Returns the maximum value of <b>byte</b> : 255.
<code>.MaxValue(int)</code>	Returns the maximum value of <b>int</b> : 2147483647.
<code>.MaxValue(long)</code>	Returns the maximum value of <b>long</b> : 9223372036854775807.
<code>.MaxValue(sbyte)</code>	Returns the maximum value of <b>sbyte</b> : 127.
<code>.MaxValue(short)</code>	Returns the maximum value of <b>short</b> : 32767.
<code>.MaxValue(uint)</code>	Returns the maximum value of <b>uint</b> : 4294967295.
<code>.MaxValue(ulong)</code>	Returns the maximum value of <b>ulong</b> : 18446744073709551615.
<code>.MaxValue(ushort)</code>	Returns the maximum value of <b>ushort</b> : 65535.
<code>Median(const T&amp;, const T&amp;, const T&amp;)</code>	Returns the median of three values.
<code>Median(const T&amp;, const T&amp;, const T&amp;, R)</code>	Returns the median of three values according to the given ordering relation.
<code>Min(const T&amp;, const T&amp;)</code>	Returns the minimum of two values.
<code>MinElement(I, I)</code>	Returns the position of the first occurrence of the smallest element in a sequence of elements.
<code>MinElement(I, I, R)</code>	Returns the position of the first occurrence of the smallest element according to the given ordering relation in a sequence of elements.

<code>MinValue()</code>	Returns the smallest value of an integer type.
<code>MinValue(byte)</code>	Returns the minimum value of <code>byte</code> : 0.
<code>MinValue(int)</code>	Returns the minimum value of <code>int</code> : -2147483648.
<code>MinValue(long)</code>	Returns the minimum value of <code>long</code> : -9223372036854775808.
<code>MinValue(sbyte)</code>	Returns the minimum value of <code>sbyte</code> : -128.
<code>MinValue(short)</code>	Returns the minimum value of <code>short</code> : -32768.
<code>MinValue(uint)</code>	Returns the minimum value of <code>uint</code> : 0.
<code>MinValue(ulong)</code>	Returns the minimum value of <code>ulong</code> : 0.
<code>MinValue(ushort)</code>	Returns the minimum value of <code>ushort</code> : 0.
<code>Move(I, I, O)</code>	Moves a sequence.
<code>MoveBackward(I, I, O)</code>	Moves a source sequence to a target sequence starting from the end of the source sequence.
<code>Next(I, int)</code>	Returns a forward iterator advanced the specified number of steps.
<code>Next(I, int)</code>	Returns a random access iterator advanced the specified offset.
<code>NextPermutation(I, I)</code>	Computes the lexicographically next permutation of a sequence of elements.
<code>NextPermutation(I, I, R)</code>	Computes the lexicographically next permutation of a sequence of elements according to the given ordering relation.
<code>Now()</code>	Returns current time point value from computer's real time clock.
<code>ParseBool(const System.String&amp;)</code>	Parses a Boolean value "true" or "false" from the given string and returns it.
<code>ParseBool(const System.String&amp;, bool&amp;)</code>	Parses a Boolean value "true" or "false" from the given string and returns true if the parsing was successful, false if not.

<code>ParseDate(const System.String&amp;)</code>	Parses a date from the given string and returns it.
<code>ParseDouble(const System.String&amp;)</code>	Parses a <b>double</b> value from the given string and returns it.
<code>ParseDouble(const System.String&amp;, double&amp;)</code>	Parses a <b>double</b> value from the given string and returns true if the parsing was successful, false if not.
<code>ParseHex(const System.String&amp;)</code>	Parses a hexadecimal value from a string.
<code>ParseHex(const System.String&amp;, System.uhuge&amp;)</code>	Parses a hexadecimal 128-bit value from a string and returns true, if the parsing was successful.
<code>ParseHex(const System.String&amp;, ulong&amp;)</code>	Parses a hexadecimal value from a string and returns true if the parsing was successful.
<code>ParseHexUHuge(const System.String&amp;)</code>	Parses a 128-bit decimal value from a string.
<code>ParseInt(const System.String&amp;)</code>	Parses an <b>int</b> from the given string and returns it.
<code>ParseInt(const System.String&amp;, int&amp;)</code>	Parses an <b>int</b> value from the given string and returns true if the parsing was successful, false if not.
<code>ParseUHuge(const System.String&amp;)</code>	Parses a decimal 128-bit value from a string.
<code>ParseUHuge(const System.String&amp;, System.uhuge&amp;)</code>	Parses a decimal 128-bit value from a string and returns true, if the parsing was successful.
<code>ParseUInt(const System.String&amp;)</code>	Parses an <b>uint</b> from the given string and returns it.
<code>ParseUInt(const System.String&amp;, uint&amp;)</code>	Parses an <b>uint</b> value from the given string and returns true if the parsing was successful, false if not.
<code>ParseULong(const System.String&amp;)</code>	Parses an <b>ulong</b> from the given string and returns it.
<code>ParseULong(const System.String&amp;, ulong&amp;)</code>	Parses an <b>ulong</b> value from the given string and returns true if the parsing was successful, false if not.
<code>PrevPermutation(I, I)</code>	Computes the lexicographically previous permutation of a sequence of elements.

<code>PrevPermutation(I, I, R)</code>	Computes the lexicographically previous permutation according to the given ordering relation of a sequence of elements.
<code>PtrCast(const SystemSharedPtr&lt;T&gt;&amp;)</code>	Casts a shared pointer.
<code>Rand()</code>	Returns a pseudorandom number generated by the Mersenne Twister pseudorandom number generator <code>MT</code> .
<code>RandomNumber(uint)</code>	Returns a pseudorandom number in range <code>0..n - 1</code> inclusive.
<code>RandomShuffle(I, I)</code>	Computes a random permutation of a random access sequence.
<code>Remove(I, I, P)</code>	Removes values satisfying a given predicate from a sequence by moving them to the end of the sequence. Returns an iterator pointing to the beginning of removed values.
<code>Remove(I, I, const T&amp;)</code>	Removes the values equal to the given value from the given sequence of values by moving them to the end of the sequence. Returns an iterator pointing to the beginning of removed values.
<code>RemoveCopy(I, I, O, P)</code>	Copies elements from given range to a destination range excluding elements satisfying a given predicate. Returns an iterator pointing one past the end of the destination range.
<code>RemoveCopy(I, I, O, const T&amp;)</code>	Copies elements from given range to a destination range excluding elements that are equal to given value. Returns an iterator pointing one past the end of the destination range.
<code>RethrowException(System.ExceptionPtr&amp;)</code>	Rethrows an exception captured to an <code>ExceptionPtr</code> .
<code>Reverse(I, I)</code>	Reverses a sequence.
<code>Reverse(I, I)</code>	Reverses a sequence.
<code>ReverseUntil(I, I, I)</code>	Reverses a sequence until the given <code>middle</code> iterator is hit.

<code>Rotate(I, I, I)</code>	Rotates a sequence with respect to a <code>middle</code> iterator and returns an iterator pointing to the new middle.
<code>Rvalue(T&amp;&amp;)</code>	Converts an argument to an rvalue so that it can be moved.
<code>Select_0_2(const T&amp;, const T&amp;, R)</code>	Returns the smaller of two values according to the given ordering relation.
<code>Select_0_3(const T&amp;, const T&amp;, const T&amp;, R)</code>	Returns the smallest of tree values according to the given ordering relation.
<code>Select_1_2(const T&amp;, const T&amp;, R)</code>	Returns the larger of two values according to the given ordering relation.
<code>Select_1_3(const T&amp;, const T&amp;, const T&amp;, R)</code>	Returns the median of three values according to the given ordering relation.
<code>Select_1_3_ab(const T&amp;, const T&amp;, const T&amp;, R)</code>	Returns the median of three values when the first two are in increasing order according to the given ordering relation.
<code>Select_2_3(const T&amp;, const T&amp;, const T&amp;, R)</code>	Returns the largest of three values according to the given ordering relation.
<code>Sort(C&amp;)</code>	Sorts the elements of a forward container to increasing order.
<code>Sort(C&amp;)</code>	Sorts the elements of a random access container to increasing order.
<code>Sort(C&amp;, R)</code>	Sorts the elements of a random access container to order according to the given ordering relation.
<code>Sort(C&amp;, R)</code>	Sorts the elements of a forward container to order according to the given ordering relation.
<code>Sort(I, I)</code>	Sorts the elements of a sequence to increasing order.
<code>Sort(I, I, R)</code>	Sorts the elements of a sequence to order according to the given ordering relation.
<code>Swap(T&amp;, T&amp;)</code>	Exchanges two values.
<code>ToHexString(System.uhuge)</code>	Returns a 128-bit value converted to hexadecimal representation.

ToHexString(U)	Converts an unsigned integer value to hexadecimal string representation.
ToHexString(byte)	Converts a <b>byte</b> to hexadecimal string representation.
ToHexString(uint)	Converts an <b>uint</b> to hexadecimal string representation.
ToHexString(ulong)	Converts an <b>ulong</b> to hexadecimal string representation.
ToHexString(ushort)	Converts an <b>ushort</b> to hexadecimal string representation.
ToLower(const System.String&)	Converts a string to lower case.
ToString(I)	Converts a signed integer value to string representation.
ToString(System.Date)	Converts a <b>Date</b> to a string representation.
ToString(System.uhuge)	Converts a 128-bit value to string representation.
ToString(U)	Converts an unsigned integer value to string representation.
ToString(bool)	Converts a Boolean value to string representation.
ToString(byte)	Converts a <b>byte</b> to string representation.
ToString(char)	Converts a character to string representation.
ToString(double)	Converts a <b>double</b> to string representation.
ToString(double, int)	Converts a <b>double</b> to string representation using the given maximum number of decimal places.
ToString(int)	Converts an <b>int</b> to string representation.
ToString(long)	Converts a <b>long</b> to string representation.
ToString(sbyte)	Converts an <b>sbyte</b> to string representation.
ToString(short)	Converts a <b>short</b> to string representation.
ToString(uint)	Converts an <b>uint</b> to string representation.

<code>ToString(ulong)</code>	Converts an <b>ulong</b> to string representation.
<code>ToString(ushort)</code>	Converts an <b>ushort</b> to string representation.
<code>ToUpper(const System.String&amp;)</code>	Converts a string to upper case.
<code>ToUtf8(uint)</code>	Converts the given Unicode code point to UTF-8 representation.
<code>Transform(I, I, O, F)</code>	Transforms an input sequence to an output sequence using a unary function.
<code>Transform(I1, I1, I2, O, F)</code>	Transforms two input sequences to an ouput sequence using a binary function.
<code>UpperBound(I, I, const T&amp;)</code>	Finds a position of the first element in a sorted sequence that is greater than the given value.
<code>UpperBound(I, I, const T&amp;, R)</code>	Finds a position of the first element in a sorted sequence that is greater than the given value according to the given ordering relation.
<code>endl()</code>	Returns <code>EndLine</code> object that represents an end of line character.

## 2.4.66 Abs(const T&) Function

Returns the absolute value of the argument.

### Syntax

```
public T Abs(const T& x);
```

### Constraint

T is [OrderedAdditiveGroup](#)

### Parameters

Name	Type	Description
x	const T&	A value.

### Returns

T

if  $x < T(0)$  returns  $-x$ , else returns  $x$ .

### 2.4.66.1 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();
}
```

## 2.4.67 Accumulate(I, I, T, Op) Function

Accumulates a sequence with respect to a binary operation.

### Syntax

```
public T Accumulate(I begin, I end, T init, Op op);
```

### Constraint

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.

### 2.4.67.1 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);
}
```

**Implementation**

[algorithm.cm](#), page 8

## 2.4.68 BackInserter(C&) Function

Returns a [BackInsertIterator<C>](#) for a back insertion sequence.

### Syntax

```
public System.BackInsertIterator<C> BackInserter(C& c);
```

### Constraint

C is [BackInsertionSequence](#)

### Parameters

Name	Type	Description
c	C&	A back insertion sequence.

### Returns

[System.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.

### 2.4.68.1 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();
}
```

## 2.4.69 CaptureCurrentException() Function

This function can be used in exception handlers (catch clauses) to capture the current exception with full type information to an [ExceptionPtr](#). This is useful for example in multithreaded programs where exception thrown inside a thread function can be transferred to the main thread and rethrown using [RethrowException\(System.ExceptionPtr&\)](#) function from there.

### Syntax

```
public System.ExceptionPtr CaptureCurrentException();
```

### Returns

[System.ExceptionPtr](#)

Returns [ExceptionPtr](#)

### 2.4.69.1 Example

```
using System;

// This program writes the following message to the standard error stream:
// FooException at 'C:/Temp/exptr/System.ExceptionPtr.cm' line 32:
// exception from thread
// call stack:
// 1> function 'foo()' file C:/Temp/exptr/System.ExceptionPtr.cm line 32
// 0> function 'ThreadFunction(void*)' file C:/Temp/exptr/System.
// ExceptionPtr.cm line 40

public class FooException : Exception
{
    public FooException(const string& message_) : base(message_)
    {
    }
}

public class ThreadData
{
    public void SetException(ExceptionPtr&& exPtr_)
    {
        exPtr = exPtr_;
    }
    public ExceptionPtr GetException() const
    {
        return Rvalue(exPtr);
    }
    private ExceptionPtr exPtr;
}

void foo()
{
    throw FooException("exception from thread");
}

void ThreadFunction(void* data)
{
    ThreadData* threadData = cast<ThreadData*>(data);
```

```
try
{
    foo();
}
catch (const Exception& ex)
{
    ExceptionPtr exPtr = CaptureCurrentException();
    threadData->SetException(Rvalue(exPtr));
}

void main()
{
    try
    {
        ThreadData threadData;
        System.Threading.Thread thread(System.Threading.ThreadFun(
            ThreadFunction), &threadData);
        thread.Join();
        ExceptionPtr exPtr = threadData.GetException();
        if (exPtr.HasException())
        {
            RethrowException(exPtr);
        }
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.ToString() << endl();
    }
}
```

## 2.4.70 Copy(I, I, O) Function

Copies a sequence.

### Syntax

```
public O Copy(I begin, I end, O to);
```

### Constraint

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.

#### 2.4.70.1 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

## 2.4.71 CopyBackward(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 begin, I end, O to);
```

### Constraint

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.

#### 2.4.71.1 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);
    list.Add(3);
    CopyBackward(list.CBegin() + 1, list.CEnd() - 1, list.End());
}
```

```
list [1] = 1;  
Console.Out() << list << endl();  
}
```

## Implementation

[algorithm.cm](#), page 2

## 2.4.72 Count(I, I, P) Function

Counts the number of elements in a sequence that satisfy a predicate.

### Syntax

```
public int Count(I begin, I end, P p);
```

### Constraint

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*.

#### 2.4.72.1 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

## 2.4.73 Count(I, I, const T&) Function

Counts the number of elements in a sequence that are equal to the given value.

### Syntax

```
public int Count(I begin, I end, const T& value);
```

### Constraint

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*.

### 2.4.73.1 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

## 2.4.74 CurrentDate() Function

Returns current date as [Date](#) value.

### Syntax

```
public System.Date CurrentDate();
```

### Returns

[System.Date](#)

Returns current date as [Date](#) value.

## 2.4.75 Distance(I, I) Function

Returns the distance between two random access iterators.

### Syntax

```
public int Distance(I first, I last);
```

### Constraint

I is [RandomAccessIterator](#)

### Parameters

Name	Type	Description
first	I	The first random access iterator.
last	I	The second random access iterator.

### Returns

int

Returns  $last - first$ .

### 2.4.75.1 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

## 2.4.76 Distance(I, I) Function

Returns the distance between two forward iterators.

### Syntax

```
public int Distance(I first, I last);
```

### Constraint

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

### 2.4.77 `EnableSharedFromThis(System.ShareableFromThis<T>*, U*, const System.SharedCount<U>&)` Function

A function that enables the *shared from this* idiom. Implementation detail.

#### Syntax

```
public void EnableSharedFromThis(System.ShareableFromThis<T>* left, U* right, const System.SharedCount<U>& count);
```

#### Parameters

Name	Type	Description
left	<code>System.ShareableFromThis&lt;T&gt;*</code>	A pointer to a class derived from <code>ShareableFromThis&lt;T&gt;</code> .
right	<code>U*</code>	A pointer to class derived from <code>ShareableFromThis&lt;T&gt;</code> .
count	<code>const System.SharedCount&lt;U&gt;&amp;</code>	A shared count.

## 2.4.78 EnableSharedFromThis(void\*, void\*, const System.SharedCount<T>&)

An empty function that catches classes that are not shareable from this. Implementation detail.

### Syntax

```
public void EnableSharedFromThis(void* __parameter0, void* __parameter1, const System.SharedCount<T>& __parameter2);
```

### Parameters

Name	Type	Description
__parameter0	void*	Pointer to any object.
__parameter1	void*	Pointer to any object.
__parameter2	const System.SharedCount<T>&	A shared count.

## 2.4.79 Equal(I1, I1, I2, I2) Function

Compares two sequences for equality.

### Syntax

```
public bool Equal(I1 first1, I1 last1, I2 first2, I2 last2);
```

### Constraint

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.

#### 2.4.79.1 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

## 2.4.80 Equal(I1, I1, I2, I2, R) Function

Compares two sequences for equality using the given equality relation.

### Syntax

```
public bool Equal(I1 first1, I1 last1, I2 first2, I2 last2, R r);
```

### Constraint

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.

### 2.4.80.1 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](#)

## 2.4.81 EqualRange(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 System.Pair<I, I> EqualRange(I first, I last, const T& value);
```

### Constraint

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

[System.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.

### 2.4.81.1 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();
}

```

### 2.4.81.2 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

## 2.4.82 EqualRange(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 System.Pair<I, I> EqualRange(I first, I last, const T& value, R r);
```

### Constraint

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

[System.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.

### 2.4.82.1 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

[algorith.cm, page 6](#)

## 2.4.83 Factorial(U) Function

Returns a factorial of the argument.

### Syntax

```
public U Factorial(U n);
```

### Constraint

U is [UnsignedInteger](#)

### Parameters

Name	Type	Description
n	U	An unsigned integer value.

### Returns

U

Returns  $n!$ .

### 2.4.83.1 Example

```
using System;

// Writes:
// 120

void main()
{
    uint x = 5u;
    uint f = Factorial(x);
    Console.Out() << f << endl();
}
```

### Implementation

[algorithm.cm, page 15](#)

## 2.4.84 `Fill(I, I, const T&)` Function

Fills a sequence with the given value.

### Syntax

```
public void Fill(I begin, I end, const T& value);
```

### Constraint

T is [Semiregular](#) and I is [ForwardIterator](#) and I.ValueType is T

### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of a sequence.
end	I	A iterator pointing one past the end of a sequence.
value	const T&	A value.

## 2.4.85 Find(I, I, P) Function

Searches the first occurrence of a value from a sequence that matches a predicate.

### Syntax

```
public I Find(I begin, I end, P p);
```

### Constraint

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*.

### 2.4.85.1 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 noexcept 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

## 2.4.86 Find(I, I, const T&) Function

Searches a value from a sequence.

### Syntax

```
public I Find(I begin, I end, const T& value);
```

### Constraint

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.

### 2.4.86.1 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 noexcept inline int Id() const
    {
        return id;
    }
    private int id;
}

public noexcept 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

## 2.4.87 ForEach(I, I, F) Function

Applies a function object for each element of a sequence.

### Syntax

```
public F ForEach(I begin, I end, F f);
```

### Constraint

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.

### 2.4.87.1 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](#)

## 2.4.88 FrontInserter(C&) Function

Returns a [FrontInsertIterator<C>](#) for a front insert sequence.

### Syntax

```
public System.FrontInsertIterator<C> FrontInserter(C& c);
```

### Constraint

C is [FrontInsertionSequence](#)

### Parameters

Name	Type	Description
c	C&	A front insertion sequence.

### Returns

[System.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.

### 2.4.88.1 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();
}
```

## 2.4.89 Gcd(T, T) Function

Returns the greatest common divisor of two values.

### Syntax

```
public T Gcd(T a, T b);
```

### Constraint

T is [EuclideanSemiring](#)

### Parameters

Name	Type	Description
a	T	The first value.
b	T	The second value.

### Returns

T

Returns  $gcd(a, b)$ .

### 2.4.89.1 Example

```
using System;

// Writes:
// 4

void main()
{
    Console.WriteLine(Gcd(12, 8));
}
```

### Implementation

[algorithm.cm, page 15](#)

## 2.4.90 HexChar(byte) Function

Returns hexadecimal character representation of a four-bit value.

### Syntax

```
public 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.

## 2.4.91 IdentityElement(System.Multiples<T>) Function

Returns the identity element of multiplication, that is  $T(1)$ .

### Syntax

```
public T IdentityElement(System.Multiples<T> __parameter0);
```

### Constraint

$T$  is [MultiplicativeMonoid](#)

### Parameters

Name	Type	Description
<code>__parameter0</code>	<a href="#">System.Multiples&lt;T&gt;</a>	

### Returns

$T$

Returns  $T(1)$ .

## 2.4.92 IdentityElement(System.Plus<T>) Function

Returns the identity element of addition, that is:  $T(0)$ .

### Syntax

```
public T IdentityElement(System.Plus<T> __parameter0);
```

### Constraint

$T$  is [AdditiveMonoid](#)

### Parameters

Name	Type	Description
<code>__parameter0</code>	<a href="#">System.Plus&lt;T&gt;</a>	

### Returns

$T$

Returns  $T(0)$ .

## 2.4.93 Inserter(C&, I) Function

Returns an [InsertIterator<C>](#) for an insertion sequence and its iterator.

### Syntax

```
public System.InsertIterator<C> Inserter(C& c, I i);
```

### Constraint

C is [InsertionSequence](#) and I is C.Iterator

### Parameters

Name	Type	Description
c	C&	An insertion sequence.
i	I	An iterator pointing to position to insert elements.

### Returns

[System.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.

### 2.4.93.1 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();
}
```

## 2.4.94 InsertionSort(I, I) Function

Sorts a sequence of values using insertion sort algorithm.

### Syntax

```
public void InsertionSort(I begin, I end);
```

### Constraint

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.

### 2.4.94.1 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

## 2.4.95 InsertionSort(I, I, R) Function

Sorts a sequence of values using insertion sort algorithm and given ordering relation.

### Syntax

```
public void InsertionSort(I begin, I end, R r);
```

### Constraint

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.

### 2.4.95.1 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(4);
    list.Add(6);
    list.Add(0);
    list.Add(4);
    InsertionSort(list.Begin(), list.End(), Greater<int>());
    Console.Out() << list << endl();
}
```

### Implementation

[algorithcm.cm](#), page 11

## 2.4.96 IsAlpha(char) Function

Returns true if the given character is an alphabetic character, false otherwise.

### Syntax

```
public 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'.

## 2.4.97 IsAlphanumeric(char) Function

Returns true if the given character is an alphanumeric character, false otherwise.

### Syntax

```
public 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'.

## 2.4.98 IsControl(char) Function

Returns true if the given character is a control character, false otherwise.

### Syntax

```
public 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.

## 2.4.99 IsDigit(char) Function

Returns true if the given character is a decimal digit, false otherwise.

### Syntax

```
public 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'.

### 2.4.100 IsGraphic(char) Function

Returns true if the given character is a graphical character, false otherwise.

#### Syntax

```
public 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.

### 2.4.101 IsHexDigit(char) Function

Returns true if the given character is a hexadecimal digit, false otherwise.

#### Syntax

```
public 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'.

### 2.4.102 IsLower(char) Function

Returns true if the given character is a lower case letter, false otherwise.

#### Syntax

```
public 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'.

### 2.4.103 IsPrintable(char) Function

Returns true if the given character is a printable character, false otherwise.

#### Syntax

```
public 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.

### 2.4.104 IsPunctuation(char) Function

Returns true if the given character is a punctuation character, false otherwise.

#### Syntax

```
public 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.

### 2.4.105 IsSpace(char) Function

Returns true if the given character is a space character, false otherwise.

#### Syntax

```
public 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.

### 2.4.106 IsUpper(char) Function

Returns true if the given character is an upper case letter, false otherwise.

#### Syntax

```
public 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'.

### 2.4.107 LastComponentsEqual(const System.String&, const System.String&, char) Function

Returns true, if last components of two strings are equal, false otherwise.

#### Syntax

```
public bool LastComponentsEqual(const System.String& s0, const System.String& s1,  
char componentSeparator);
```

#### Parameters

Name	Type	Description
s0	const System.String&	First string.
s1	const System.String&	Second string.
componentSeparator	char	Component separator character.

#### Returns

bool

Returns true, if last components of two strings are equal, false otherwise.

#### Remarks

First splits `s0` and `s1` to components with respect to component separator character `componentSeparator`. Then computes  $n$ , the minimum number of components of `s0` and `s1`. Then returns true, if last  $n$  components of `s0` and `s1` are equal, false otherwise.

## 2.4.108 LexicographicalCompare(I1, I1, I2, I2) Function

Returns true if the first sequence comes lexicographically before the second sequence, false otherwise.

### Syntax

```
public bool LexicographicalCompare(I1 first1, I1 last1, I2 first2, I2 last2);
```

### Constraint

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.

### 2.4.108.1 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

## 2.4.109 LexicographicalCompare(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 bool LexicographicalCompare(I1 first1, I1 last1, I2 first2, I2 last2, R r);
```

### Constraint

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.

### 2.4.109.1 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](#)

## 2.4.110 LowerBound(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 I LowerBound(I first, I last, const T& value);
```

### Constraint

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.

#### 2.4.110.1 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](#)

### 2.4.111 LowerBound(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 I LowerBound(I first, I last, const T& value, R r);
```

#### Constraint

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

### 2.4.112 MakePair(const T&, const U&) Function

Returns a pair composed of the given values.

#### Syntax

```
public System.Pair<T, U> MakePair(const T& first, const U& second);
```

#### Constraint

T is [Semiregular](#) and U is [Semiregular](#)

#### Parameters

Name	Type	Description
first	const T&	The first value.
second	const U&	The second value.

#### Returns

[System.Pair<T, U>](#)

Returns a pair composed of the given values.

### 2.4.113 Max(const T&, const T&) Function

Returns the maximum of two values.

#### Syntax

```
public const T& Max(const T& left, const T& right);
```

#### Constraint

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  $left$ .

## 2.4.114 MaxElement(I, I) Function

Returns the position of the first occurrence of the largest element in a sequence of elements.

### Syntax

```
public I MaxElement(I first, I last);
```

### Constraint

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.

### 2.4.114.1 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](#)

### 2.4.115 MaxElement(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 I MaxElement(I first, I last, R r);
```

#### Constraint

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

### 2.4.116 MaxValue() Function

Returns the largest value of an integer type.

#### Syntax

```
public I MaxValue();
```

#### Returns

I

Returns the largest value of an integer type.

### 2.4.117 MaxValue(byte) Function

Returns the maximum value of **byte**: 255.

#### Syntax

```
public byte MaxValue(byte __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	byte	

#### Returns

byte

Returns the maximum value of **byte**: 255.

### 2.4.118 MaxValue(int) Function

Returns the maximum value of **int**: 2147483647.

#### Syntax

```
public int MaxValue(int __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	int	

#### Returns

int

Returns the maximum value of **int**: 2147483647.

### 2.4.119 MaxValue(long) Function

Returns the maximum value of **long**: 9223372036854775807.

#### Syntax

```
public long MaxValue(long __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	long	

#### Returns

long

Returns the maximum value of **long**: 9223372036854775807.

## 2.4.120 MaxValue(sbyte) Function

Returns the maximum value of **sbyte**: 127.

### Syntax

```
public sbyte MaxValue(sbyte __parameter0);
```

### Parameters

Name	Type	Description
__parameter0	sbyte	

### Returns

sbyte

Returns the maximum value of **sbyte**: 127.

### 2.4.121 MaxValue(short) Function

Returns the maximum value of **short**: 32767.

#### Syntax

```
public short MaxValue(short __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	short	

#### Returns

short

Returns the maximum value of **short**: 32767.

### 2.4.122 MaxValue(uint) Function

Returns the maximum value of **uint**: 4294967295.

#### Syntax

```
public uint MaxValue(uint __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	uint	

#### Returns

uint

Returns the maximum value of **uint**: 4294967295.

### 2.4.123 MaxValue(ulong) Function

Returns the maximum value of **ulong**: 18446744073709551615.

#### Syntax

```
public ulong MaxValue(ulong __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	ulong	

#### Returns

ulong

Returns the maximum value of **ulong**: 18446744073709551615.

### 2.4.124 MaxValue(ushort) Function

Returns the maximum value of **ushort**: 65535.

#### Syntax

```
public ushort MaxValue(ushort __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	ushort	

#### Returns

ushort

Returns the maximum value of **ushort**: 65535.

## 2.4.125 Median(const T&, const T&, const T&) Function

Returns the median of three values.

### Syntax

```
public const T& Median(const T& a, const T& b, const T& c);
```

### Constraint

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.

## 2.4.126 Median(const T&, const T&, const T&, R) Function

Returns the median of three values according to the given ordering relation.

### Syntax

```
public const T& Median(const T& a, const T& b, const T& c, R r);
```

### Constraint

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.

### 2.4.127 Min(const T&, const T&) Function

Returns the minimum of two values.

#### Syntax

```
public const T& Min(const T& left, const T& right);
```

#### Constraint

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  $right$ .

## 2.4.128 MinElement(I, I) Function

Returns the position of the first occurrence of the smallest element in a sequence of elements.

### Syntax

```
public I MinElement(I first, I last);
```

### Constraint

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.

### 2.4.128.1 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

### 2.4.129 MinElement(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 I MinElement(I first, I last, R r);
```

#### Constraint

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 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 smallest element according to the given ordering relation in a sequence of elements.

#### Implementation

[algorithm.cm](#), page 14

### 2.4.130 MinValue() Function

Returns the smallest value of an integer type.

#### Syntax

```
public I MinValue();
```

#### Returns

I

Returns the smallest value of an integer type.

### 2.4.131 MinValue(byte) Function

Returns the minimum value of **byte**: 0.

#### Syntax

```
public byte MinValue(byte __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	byte	

#### Returns

byte

Returns the minimum value of **byte**: 0.

### 2.4.132 MinValue(int) Function

Returns the minimum value of **int**: -2147483648.

#### Syntax

```
public int MinValue(int __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	int	

#### Returns

int

Returns the minimum value of **int**: -2147483648.

### 2.4.133 MinValue(long) Function

Returns the minimum value of **long**: -9223372036854775808.

#### Syntax

```
public long MinValue(long __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	long	

#### Returns

long

Returns the minimum value of **long**: -9223372036854775808.

### 2.4.134 MinValue(sbyte) Function

Returns the minimum value of **sbyte**: -128.

#### Syntax

```
public sbyte MinValue(sbyte __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	sbyte	

#### Returns

sbyte

Returns the minimum value of **sbyte**: -128.

### 2.4.135 MinValue(short) Function

Returns the minimum value of **short**: -32768.

#### Syntax

```
public short MinValue(short __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	short	

#### Returns

short

Returns the minimum value of **short**: -32768.

### 2.4.136 MinValue(uint) Function

Returns the minimum value of **uint**: 0.

#### Syntax

```
public uint MinValue(uint __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	uint	

#### Returns

uint

Returns the minimum value of **uint**: 0.

### 2.4.137 MinValue(ulong) Function

Returns the minimum value of **ulong**: 0.

#### Syntax

```
public ulong MinValue(ulong __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	ulong	

#### Returns

ulong

Returns the minimum value of **ulong**: 0.

### 2.4.138 MinValue(ushort) Function

Returns the minimum value of **ushort**: 0.

#### Syntax

```
public ushort MinValue(ushort __parameter0);
```

#### Parameters

Name	Type	Description
__parameter0	ushort	

#### Returns

ushort

Returns the minimum value of **ushort**: 0.

### 2.4.139 Move(I, I, O) Function

Moves a sequence.

#### Syntax

```
public O Move(I begin, I end, O to);
```

#### Constraint

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

### 2.4.140 MoveBackward(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 begin, I end, O to);
```

#### Constraint

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 past 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

### 2.4.141 Next(I, int) Function

Returns a forward iterator advanced the specified number of steps.

#### Syntax

```
public I Next(I i, int n);
```

#### Constraint

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

### 2.4.142 Next(I, int) Function

Returns a random access iterator advanced the specified offset.

#### Syntax

```
public I Next(I i, int n);
```

#### Constraint

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](#)

### 2.4.143 NextPermutation(I, I) Function

Computes the lexicographically next permutation of a sequence of elements.

#### Syntax

```
public bool NextPermutation(I begin, I end);
```

#### Constraint

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.

#### 2.4.143.1 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

### 2.4.144 NextPermutation(I, I, R) Function

Computes the lexicographically next permutation of a sequence of elements according to the given ordering relation.

#### Syntax

```
public bool NextPermutation(I begin, I end, R r);
```

#### Constraint

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

### 2.4.145 Now() Function

Returns current time point value from computer's real time clock.

#### Syntax

```
public System.TimePoint Now();
```

#### Returns

[System.TimePoint](#)

Returns number of nanoseconds elapsed since 1.1.1970 as a time point value.

### 2.4.146 ParseBool(const System.String&) Function

Parses a Boolean value “true” or “false” from the given string and returns it.

#### Syntax

```
public bool ParseBool(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

bool

Returns true if *s* contains “true”, false if *s* contains “false”. Otherwise throws [ConversionException](#).

### 2.4.147 ParseBool(const System.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 System.String& s, bool& b);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
b	bool&	If <i>s</i> contains “true” <i>b</i> is set to true, if <i>s</i> contains “false” <i>b</i> is set to false.

#### Returns

bool

Returns true if the parsing was successful, false if not.

### 2.4.148 ParseDate(const System.String&) Function

Parses a date from the given string and returns it.

#### Syntax

```
public System.Date ParseDate(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

[System.Date](#)

Returns the parsed date value if the parsing was successful. Otherwise throws [ConversionException](#).

### 2.4.149 ParseDouble(const System.String&) Function

Parses a **double** value from the given string and returns it.

#### Syntax

```
public double ParseDouble(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

double

Returns the parsed **double** value if the parsing was successful. Otherwise throws [ConversionException](#).

## 2.4.150 ParseDouble(const System.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 System.String& s, double& x);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
x	double&	A <b>double</b> parsed from <i>s</i> .

### Returns

bool

Returns true if the parsing was successful, false if not.

### 2.4.151 ParseHex(const System.String&) Function

Parses a hexadecimal value from a string.

#### Syntax

```
public ulong ParseHex(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

ulong

Returns the parsed hexadecimal value if the parsing was successful, otherwise throws [ConversionException](#).

## 2.4.152 ParseHex(const System.String&, System.uhuge&) Function

Parses a hexadecimal 128-bit value from a string and returns true, if the parsing was successful.

### Syntax

```
public bool ParseHex(const System.String& s, System.uhuge& hex);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
hex	System.uhuge&	Parsed 128-bit value.

### Returns

bool

Returns true, if the parsing was successful, false otherwise.

### 2.4.153 ParseHex(const System.String&, ulong&) Function

Parses a hexadecimal value from a string and returns true if the parsing was successful.

#### Syntax

```
public bool ParseHex(const System.String& s, ulong& hex);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
hex	ulong&	Parsed value.

#### Returns

bool

Returns true, if the parsing was successful, false otherwise.

## 2.4.154 ParseHexUHuge(const System.String&) Function

Parses a 128-bit decimal value from a string.

### Syntax

```
public System.uhuge ParseHexUHuge(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

### Returns

[System.uhuge](#)

Returns the parsed 128-bit decimal value if the parsing was successful, otherwise throws [ConversionException](#).

### 2.4.155 ParseInt(const System.String&) Function

Parses an **int** from the given string and returns it.

#### Syntax

```
public int ParseInt(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

int

Returns the parsed **int** value if the parsing was successful. Otherwise throws [ConversionException](#).

### 2.4.156 ParseInt(const System.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 System.String& s, int& x);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
x	int&	An <b>int</b> parsed from <i>s</i> .

#### Returns

bool

Returns true if the parsing was successful, false if not.

### 2.4.157 ParseUHuge(const System.String&) Function

Parses a decimal 128-bit value from a string.

#### Syntax

```
public System.uhuge ParseUHuge(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

[System.uhuge](#)

Returns parsed 128-bit decimal value if parsing was successful, otherwise throws [ConversionException](#).

### 2.4.158 ParseUHuge(const System.String&, System.uhuge&) Function

Parses a decimal 128-bit value from a string and returns true, if the parsing was successful.

#### Syntax

```
public bool ParseUHuge(const System.String& s, System.uhuge& x);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
x	System.uhuge&	Parsed 128-bit value.

#### Returns

bool

Returns true, if the parsing was successful, false otherwise.

## 2.4.159 ParseUInt(const System.String&) Function

Parses an **uint** from the given string and returns it.

### Syntax

```
public uint ParseUInt(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

### Returns

uint

Returns the parsed **uint** value if the parsing was successful. Otherwise throws [ConversionException](#).

## 2.4.160 ParseUInt(const System.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 System.String& s, uint& x);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
x	uint&	An <b>uint</b> parsed from <i>s</i> .

### Returns

bool

Returns true if the parsing was successful, false if not.

### 2.4.161 ParseULong(const System.String&) Function

Parses an **ulong** from the given string and returns it.

#### Syntax

```
public ulong ParseULong(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.

#### Returns

ulong

Returns the parsed **ulong** value if the parsing was successful. Otherwise throws [ConversionException](#).

### 2.4.162 ParseULong(const System.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 System.String& s, ulong& x);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to parse.
x	ulong&	An <b>ulong</b> parsed from <i>s</i> .

#### Returns

bool

Returns true if the parsing was successful, false if not.

### 2.4.163 PrevPermutation(I, I) Function

Computes the lexicographically previous permutation of a sequence of elements.

#### Syntax

```
public bool PrevPermutation(I begin, I end);
```

#### Constraint

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.

### 2.4.164 PrevPermutation(I, I, R) Function

Computes the lexicographically previous permutation according to the given ordering relation of a sequence of elements.

#### Syntax

```
public bool PrevPermutation(I begin, I end, R r);
```

#### Constraint

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.

## 2.4.165 PtrCast(const SystemSharedPtr<T>&) Function

Casts a shared pointer.

### Syntax

```
public SystemSharedPtr<U> PtrCast(const SystemSharedPtr<T>& from);
```

### Parameters

Name	Type	Description
from	const SystemSharedPtr<T>&	A shared pointer to cast from.

### Returns

[SystemSharedPtr<U>](#)

Returns the shared pointer casted to *SharedPtr < U >*.

### 2.4.165.1 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);
}
```

## 2.4.166 Rand() Function

Returns a pseudorandom number generated by the Mersenne Twister pseudorandom number generator [MT](#).

### Syntax

```
public uint Rand();
```

### Returns

uint

Returns a pseudorandom number generated by the Mersenne Twister pseudorandom number generator [MT](#).

### Remarks

The returned pseudorandom number is in range 0..[MaxValue\(uint\)](#) inclusive.

## 2.4.167 RandomNumber(uint) Function

Returns a pseudorandom number in range 0..[n](#) - 1 inclusive.

### Syntax

```
public uint RandomNumber(uint n);
```

### Parameters

Name	Type	Description
n	uint	Upper bound for the generated pseudorandom number.

### Returns

uint

Returns a pseudorandom number in range 0..[n](#) - 1 inclusive.

### Remarks

Uses [Rand\(\)](#) function to generate a pseudorandom number and returns that number modulo [n](#).

### 2.4.168 RandomShuffle(I, I) Function

Computes a random permutation of a random access sequence.

#### Syntax

```
public void RandomShuffle(I begin, I end);
```

#### Constraint

I is [RandomAccessIterator](#)

#### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of the sequence.
end	I	An iterator pointing one past the end of the sequence.

### 2.4.169 Remove(I, I, P) Function

Removes values satisfying a given predicate from a sequence by moving them to the end of the sequence. Returns an iterator pointing to the beginning of removed values.

#### Syntax

```
public I Remove(I begin, I end, P p);
```

#### Constraint

I is [ForwardIterator](#) and P is [UnaryPredicate](#) and P.ArgumentType is I.ValueType

#### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of a sequence.
end	I	An iterator pointing to one past the end of a sequence.
p	P	A unary predicate.

#### Returns

I

Returns an iterator pointing to the beginning of removed values.

### 2.4.170 Remove(I, I, const T&) Function

Removes the values equal to the given value from the given sequence of values by moving them to the end of the sequence. Returns an iterator pointing to the beginning of removed values.

#### Syntax

```
public I Remove(I begin, I end, const T& value);
```

#### Constraint

I is [ForwardIterator](#) and T is [Semiregular](#) and [EqualityComparable<T, I.ValueType>](#)

#### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of a sequence.
end	I	An iterator pointing one past the end of a sequence.
value	const T&	A value to remove.

#### Returns

I

Returns an iterator pointing to the beginning of removed values.

### 2.4.171 RemoveCopy(I, I, O, P) Function

Copies elements from given range to a destination range excluding elements satisfying a given predicate. Returns an iterator pointing one past the end of the destination range.

#### Syntax

```
public O RemoveCopy(I begin, I end, O result, P p);
```

#### Constraint

I is [InputIterator](#) and O is [OutputIterator](#) and O.ValueType is I.ValueType and P is [UnaryPredicate](#) and P.ArgumentType is I.ValueType

#### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of a sequence to copy.
end	I	An iterator pointing one past the end of a sequence to copy.
result	O	An iterator pointing to the beginning of a destination range.
p	P	A unary predicate.

#### Returns

O

Returns an iterator pointing one past the end of the destination range.

### 2.4.172 RemoveCopy(I, I, O, const T&) Function

Copies elements from given range to a destination range excluding elements that are equal to given value. Returns an iterator pointing one past the end of the destination range.

#### Syntax

```
public O RemoveCopy(I begin, I end, O result, const T& value);
```

#### Constraint

T is [Semiregular](#) and I is [InputIterator](#) and O is [OutputIterator](#) and O.ValueType is I.ValueType and [EqualityComparable<T, I.ValueType>](#)

#### Parameters

Name	Type	Description
begin	I	An iterator pointing to the beginning of a sequence to copy.
end	I	An iterator pointing one past the end of a sequence to copy.
result	O	An iterator pointing to the beginning of a destination range.
value	const T&	A value.

#### Returns

O

Returns an iterator pointing one past the end of the destination range.

## 2.4.173 RethrowException(System.ExceptionPtr&) Function

Rethrows an exception captured to an [ExceptionPtr](#).

### Syntax

```
public void RethrowException(System.ExceptionPtr& capturedException);
```

### Parameters

Name	Type	Description
capturedException	<a href="#">System.ExceptionPtr&amp;</a>	Exception captured to an <a href="#">ExceptionPtr</a> .

### Remarks

Used with function [CaptureCurrentException\(\)](#).

#### 2.4.173.1 Example

```
using System;

// This program writes the following message to the standard error stream:
// FooException at 'C:/Temp/exptr/System.ExceptionPtr.cm' line 32:
// exception from thread
// call stack:
// 1> function 'foo()' file C:/Temp/exptr/System.ExceptionPtr.cm line 32
// 0> function 'ThreadFunction(void*)' file C:/Temp/exptr/System.
// ExceptionPtr.cm line 40

public class FooException : Exception
{
    public FooException(const string& message_) : base(message_)
    {
    }
}

public class ThreadData
{
    public void SetException(ExceptionPtr&& exPtr_)
    {
        exPtr = exPtr_;
    }
    public ExceptionPtr GetException() const
    {
        return Rvalue(exPtr);
    }
    private ExceptionPtr exPtr;
}

void foo()
{
    throw FooException("exception from thread");
}
```

```
|| void ThreadFunction(void* data)
|| {
||     ThreadData* threadData = cast<ThreadData*>(data);
||     try
||     {
||         foo();
||     }
||     catch (const Exception& ex)
||     {
||         ExceptionPtr exPtr = CaptureCurrentException();
||         threadData->SetException(Rvalue(exPtr));
||     }
|| }
|
void main()
{
    try
    {
        ThreadData threadData;
        System.Threading.Thread thread(System.Threading.ThreadFun(
            ThreadFunction), &threadData);
        thread.Join();
        ExceptionPtr exPtr = threadData.GetException();
        if (exPtr.HasException())
        {
            RethrowException(exPtr);
        }
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.ToString() << endl();
    }
}
```

### 2.4.174 Reverse(I, I) Function

Reverses a sequence.

#### Syntax

```
public void Reverse(I begin, I end);
```

#### Constraint

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](#)

### 2.4.175 Reverse(I, I) Function

Reverses a sequence.

#### Syntax

```
public void Reverse(I begin, I end);
```

#### Constraint

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.

#### 2.4.175.1 Example

```
using System;
using System.Collections;

// Writes:
// 3, 2, 1

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

#### Implementation

[algorithm.cm](#), page 1

### 2.4.176 ReverseUntil(I, I, I) Function

Reverses a sequence until the given [middle](#) iterator is hit.

#### Syntax

```
public System.Pair<I, I> ReverseUntil(I first, I middle, I last);
```

#### Constraint

I is [BidirectionalIterator](#)

#### Parameters

Name	Type	Description
first	I	A bidirectional iterator pointing to the beginning of a sequence.
middle	I	A bidirectional iterator pointing a position between <a href="#">first</a> and <a href="#">last</a> inclusive.
last	I	A bidirectional iterator pointing one past the end of a sequence.

#### Returns

[System.Pair<I, I>](#)

Returns a pair of iterators. See remarks section.

#### Remarks

While [first](#) and [last](#) differ from [middle](#), decrements [last](#), swaps elements pointed by [first](#) and [last](#), and increments [first](#). Then returns a pair made of [first](#) and [last](#).

### 2.4.177 Rotate(I, I, I) Function

Rotates a sequence with respect to a [middle](#) iterator and returns an iterator pointing to the new middle.

#### Syntax

```
public I Rotate(I first, I middle, I last);
```

#### Constraint

I is [BidirectionalIterator](#)

#### Parameters

Name	Type	Description
first	I	A bidirectional iterator pointing to the beginning of a sequence.
middle	I	An iterator pointing to a position between <a href="#">first</a> and <a href="#">last</a> incusive.
last	I	A bidirectional iterator pointing one past the end of a sequence.

#### Returns

I

Returns an iterator pointing to the new middle.

#### Remarks

Shifts a sequence of elements pointed by iterators [first](#) and [middle](#) (where the [middle](#) iterator points one past the end of that range) to the end of the whole sequence [first](#) and [last](#).

## 2.4.178 Rvalue(T&&) Function

Converts an argument to an rvalue so that it can be moved.

### Syntax

```
public T&& Rvalue(T&& x);
```

### Parameters

Name	Type	Description
x	T&&	A argument to convert.

### Returns

T&&

Returns an rvalue reference of the argument.

### 2.4.178.1 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();
    }
}
```

### 2.4.179 Select\_0\_2(const T&, const T&, R) Function

Returns the smaller of two values according to the given ordering relation.

#### Syntax

```
public const T& Select_0_2(const T& a, const T& b, R r);
```

#### Constraint

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.

## 2.4.180 Select\_0\_3(const T&, const T&, const T&, R) Function

Returns the smallest of tree values according to the given ordering relation.

### Syntax

```
public const T& Select_0_3(const T& a, const T& b, const T& c, R r);
```

### Constraint

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.

### 2.4.181 Select\_1\_2(const T&, const T&, R) Function

Returns the larger of two values according to the given ordering relation.

#### Syntax

```
public const T& Select_1_2(const T& a, const T& b, R r);
```

#### Constraint

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.

## 2.4.182 Select\_1\_3(const T&, const T&, const T&, R) Function

Returns the median of three values according to the given ordering relation.

### Syntax

```
public const T& Select_1_3(const T& a, const T& b, const T& c, R r);
```

### Constraint

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.

### 2.4.183 Select\_1\_3\_ab(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 const T& Select_1_3_ab(const T& a, const T& b, const T& c, R r);
```

#### Constraint

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.

### 2.4.184 Select\_2\_3(const T&, const T&, const T&, R) Function

Returns the largest of three values according to the given ordering relation.

#### Syntax

```
public const T& Select_2_3(const T& a, const T& b, const T& c, R r);
```

#### Constraint

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.

### 2.4.185 Sort(C&) Function

Sorts the elements of a forward container to increasing order.

#### Syntax

```
public void Sort(C& c);
```

#### Constraint

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

## 2.4.186 Sort(C&) Function

Sorts the elements of a random access container to increasing order.

### Syntax

```
public void Sort(C& c);
```

### Constraint

C is [RandomAccessContainer](#) and C.Iterator.ValueType is [TotallyOrdered](#)

### Parameters

Name	Type	Description
c	C&	A random access container.

### 2.4.186.1 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

## 2.4.187 Sort(C&, R) Function

Sorts the elements of a random access container to order according to the given ordering relation.

### Syntax

```
public void Sort(C& c, R r);
```

### Constraint

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.

### 2.4.187.1 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

## 2.4.188 Sort(C&, R) Function

Sorts the elements of a forward container to order according to the given ordering relation.

### Syntax

```
public void Sort(C& c, R r);
```

### Constraint

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.

### 2.4.189 Sort(I, I) Function

Sorts the elements of a sequence to increasing order.

#### Syntax

```
public void Sort(I begin, I end);
```

#### Constraint

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](#)

## 2.4.190 Sort(I, I, R) Function

Sorts the elements of a sequence to order according to the given ordering relation.

### Syntax

```
public void Sort(I begin, I end, R r);
```

### Constraint

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

[algoritm.cm, page 11](#)

## 2.4.191 Swap(T&, T&) Function

Exchanges two values.

### Syntax

```
public void Swap(T& left, T& right);
```

### Constraint

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 System.Rvalue.T.T.rr function and exchanged by moving them.

### Implementation

[algorithm.cm](#), page 1

## 2.4.192 ToHexString(System.uhuge) Function

Returns a 128-bit value converted to hexadecimal representation.

### Syntax

```
public System.String ToHexString(System.uhuge x);
```

### Parameters

Name	Type	Description
x	<a href="#">System.uhuge</a>	A 128-bit value to convert.

### Returns

[System.String](#)

A hexadecimal string.

## 2.4.193 ToHexString(U) Function

Converts an unsigned integer value to hexadecimal string representation.

### Syntax

```
public System.String ToHexString(U x);
```

### Constraint

U is [UnsignedInteger](#) and [ExplicitlyConvertible<U, byte>](#)

### Parameters

Name	Type	Description
x	U	An unsigned integer value.

### Returns

[System.String](#)

Returns *x* converted to hexadecimal string representation.

### Implementation

[convert.cm, page 4](#)

### 2.4.194 ToHexString(byte) Function

Converts a `byte` to hexadecimal string representation.

#### Syntax

```
public System.String ToHexString(byte b);
```

#### Parameters

Name	Type	Description
b	byte	A <code>byte</code> .

#### Returns

[System.String](#)

Returns *b* converted to hexadecimal string representation.

#### Implementation

[convert.cm, page 4](#)

## 2.4.195 ToHexString(uint) Function

Converts an **uint** to hexadecimal string representation.

### Syntax

```
public System.String ToHexString(uint u);
```

### Parameters

Name	Type	Description
u	uint	An <b>uint</b> .

### Returns

[System.String](#)

Returns *u* converted to hexadecimal string representation.

### Implementation

[convert.cm, page 4](#)

## 2.4.196 ToHexString(ulong) Function

Converts an **ulong** to hexadecimal string representation.

### Syntax

```
public System.String ToHexString(ulong u);
```

### Parameters

Name	Type	Description
u	ulong	An <b>ulong</b> .

### Returns

[System.String](#)

Returns *u* converted to hexadecimal string representation.

### Implementation

[convert.cm, page 4](#)

### 2.4.197 ToHexString(ushort) Function

Converts an **ushort** to hexadecimal string representation.

#### Syntax

```
public System.String ToHexString(ushort u);
```

#### Parameters

Name	Type	Description
u	ushort	An <b>ushort</b> .

#### Returns

[System.String](#)

Returns *u* converted to hexadecimal string representation.

#### Implementation

[convert.cm, page 4](#)

## 2.4.198 ToLower(const System.String&) Function

Converts a string to lower case.

### Syntax

```
public System.String ToLower(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string.

### Returns

[System.String](#)

Returns *s* converted to lower case.

### Implementation

[string.cm, page 9](#)

## 2.4.199 `ToString(I)` Function

Converts a signed integer value to string representation.

### Syntax

```
public System.String ToString(I x);
```

### Constraint

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

[System.String](#)

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 1](#)

## 2.4.200 `ToString(System.Date)` Function

Converts a [Date](#) to a string representation.

### Syntax

```
public System.String ToString(System.Date date);
```

### Parameters

Name	Type	Description
date	<a href="#">System.Date</a>	A date to convert.

### Returns

[System.String](#)

Returns a string YYYY-MM-DD where YYYY is year of date in four digits, MM is month of date in two digits and DD is day of date in two digits.

## 2.4.201 `ToString(System.uhuge)` Function

Converts a 128-bit value to string representation.

### Syntax

```
public System.String ToString(System.uhuge x);
```

### Parameters

Name	Type	Description
x	<a href="#">System.uhuge</a>	A 128-bit value.

### Returns

[System.String](#)

Returns *x* converted to string.

## 2.4.202 `ToString(U)` Function

Converts an unsigned integer value to string representation.

### Syntax

```
public System.String ToString(U x);
```

### Constraint

U is `UnsignedInteger` and `ExplicitlyConvertible<U, byte>`

### Parameters

Name	Type	Description
x	U	An unsigned integer value.

### Returns

`System.String`

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 2](#)

### 2.4.203 `ToString(bool)` Function

Converts a Boolean value to string representation.

#### Syntax

```
public System.String ToString(bool b);
```

#### Parameters

Name	Type	Description
b	bool	A Boolean value.

#### Returns

[System.String](#)

If *b* returns “true” else returns “false”.

#### Implementation

[convert.cm, page 3](#)

## 2.4.204 `ToString(byte)` Function

Converts a `byte` to string representation.

### Syntax

```
public System.String ToString(byte x);
```

### Parameters

Name	Type	Description
x	byte	A <code>byte</code> .

### Returns

[System.String](#)

Returns a `byte` converted to string representation.

### Implementation

[convert.cm, page 2](#)

## 2.4.205 `ToString(char)` Function

Converts a character to string representation.

### Syntax

```
public System.String ToString(char c);
```

### Parameters

Name	Type	Description
c	char	A character.

### Returns

[System.String](#)

Returns *c* converted to string representation.

### Implementation

[convert.cm, page 3](#)

## 2.4.206 `ToString(double)` Function

Converts a `double` to string representation.

### Syntax

```
public System.String ToString(double x);
```

### Parameters

Name	Type	Description
x	double	A <code>double</code> .

### Returns

[System.String](#)

Returns *x* converted to string representation.

### Remarks

The maximum number of decimal places in the conversion is 15.

### Implementation

[convert.cm, page 3](#)

## 2.4.207 `ToString(double, int)` Function

Converts a **double** to string representation using the given maximum number of decimal places.

### Syntax

```
public System.String ToString(double x, int maxNumDecimals);
```

### Parameters

Name	Type	Description
x	double	A <b>double</b> .
maxNumDecimals	int	Maximum number of decimal digits.

### Returns

[System.String](#)

Returns *x* converted to string representation using the given maximum number of decimal places.

### Implementation

[convert.cm, page 3](#)

## 2.4.208 `ToString(int)` Function

Converts an `int` to string representation.

### Syntax

```
public System.String ToString(int x);
```

### Parameters

Name	Type	Description
x	int	An <code>int</code> .

### Returns

[System.String](#)

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 2](#)

## 2.4.209 `ToString(long)` Function

Converts a `long` to string representation.

### Syntax

```
public System.String ToString(long x);
```

### Parameters

Name	Type	Description
x	long	A long.

### Returns

[System.String](#)

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 2](#)

## 2.4.210 `ToString(sbyte)` Function

Converts an `sbyte` to string representation.

### Syntax

```
public System.String ToString(sbyte x);
```

### Parameters

Name	Type	Description
x	sbyte	An <code>sbyte</code> .

### Returns

[System.String](#)

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 2](#)

### 2.4.211 `ToString(short)` Function

Converts a `short` to string representation.

#### Syntax

```
public System.String ToString(short x);
```

#### Parameters

Name	Type	Description
x	short	An <code>short</code> .

#### Returns

[System.String](#)

Returns *x* converted to string representation.

#### Implementation

[convert.cm, page 2](#)

### 2.4.212 `ToString(uint)` Function

Converts an `uint` to string representation.

#### Syntax

```
public System.String ToString(uint x);
```

#### Parameters

Name	Type	Description
x	uint	An <code>uint</code> .

#### Returns

[System.String](#)

Returns *x* converted to string representation.

#### Implementation

[convert.cm, page 2](#)

### 2.4.213 `ToString(ulong)` Function

Converts an `ulong` to string representation.

#### Syntax

```
public System.String ToString(ulong x);
```

#### Parameters

Name	Type	Description
x	ulong	An <code>ulong</code> .

#### Returns

[System.String](#)

Returns *x* converted to string representation.

#### Implementation

[convert.cm, page 2](#)

## 2.4.214 `ToString(ushort)` Function

Converts an `ushort` to string representation.

### Syntax

```
public System.String ToString(ushort x);
```

### Parameters

Name	Type	Description
x	ushort	An <code>ushort</code> .

### Returns

[System.String](#)

Returns *x* converted to string representation.

### Implementation

[convert.cm, page 2](#)

## 2.4.215 ToUpper(const System.String&) Function

Converts a string to upper case.

### Syntax

```
public System.String ToUpper(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string.

### Returns

[System.String](#)

Returns *s* converted to upper case.

### Implementation

[string.cm, page 10](#)

## 2.4.216 ToUtf8(uint) Function

Converts the given Unicode code point to UTF-8 representation.

### Syntax

```
public System.String ToUtf8(uint c);
```

### Parameters

Name	Type	Description
c	uint	A Unicode code point.

### Returns

[System.String](#)

Returns a UTF-8 encoded string.

## 2.4.217 Transform(I, I, O, F) Function

Transforms an input sequence to an output sequence using a unary function.

### Syntax

```
public O Transform(I begin, I end, O to, F fun);
```

### Constraint

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.

### 2.4.217.1 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](#)

## 2.4.218 Transform(I1, I1, I2, O, F) Function

Transforms two input sequences to an ouput sequence using a binary function.

### Syntax

```
public O Transform(I1 begin1, I1 end1, I2 begin2, O to, F fun);
```

### Constraint

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.

### 2.4.218.1 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

## 2.4.219 UpperBound(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 I UpperBound(I first, I last, const T& value);
```

### Constraint

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*.

### 2.4.219.1 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

## 2.4.220 `UpperBound(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 I UpperBound(I first, I last, const T& value, R r);
```

### Constraint

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

## 2.4.221 endl() Function

Returns [EndLine](#) object that represents an end of line character.

### Syntax

```
public System.EndLine endl();
```

### Returns

[System.EndLine](#)

Returns [EndLine](#) object that represents an end of line character.

## 2.5 Enumerations

Enumeration	Description
CharClass	A character classification enumeration.

### 2.5.221.1 CharClass Enumeration

A character classification enumeration.

#### Enumeration Constants

Constant	Value	Description
alnum	7	Alphanumeric character.
alpha	3	Alphabetic character.
cntrl	16	Control character.
digit	4	Digit character.
graph	32	Graphical character.
lower	1	Lower case alphabetic character.
none	0	No classification.
print	64	Printable character.
punct	128	Punctuation character.
space	256	White space character.
upper	2	Upper case alphabetic character.
xdigit	8	Hexadecimal digit character.

## 2.6 Constants

Constant	Type	Value	Description
EXIT_CHAR_- CLASS_TABLE_- ALLOCATE	int	249	Program exit status when the character class table could not be allocated.
InsertionSortThreshold	int	16	A threshold value for doing insertion sort.

# 3 System.Collections Namespace

Contains collection classes and functions that operate on collections.

Figure 3.1 contains the classes in this namespace.

Figure 3.1: 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>

## 3.7 Classes

Class	Description
<code>BitSet</code>	A compact sequence of bits.
<code>Bucket&lt;T&gt;</code>	A hash table bucket.
<code>ForwardList&lt;T&gt;</code>	A singly linked list of elements.
<code>ForwardListNode&lt;T&gt;</code>	A forward list node type.
<code>ForwardListNodeIterator&lt;T, R, P&gt;</code>	A forward iterator that iterates through a <code>ForwardList&lt;T&gt;</code> .
<code>HashMap&lt;K, T, H, C&gt;</code>	An associative container of key-value pairs organized in a hash table. The keys need not be ordered.
<code>HashSet&lt;T, H, C&gt;</code>	A set of unique elements organized in a hash table. The elements need not be ordered.
<code>Hasher&lt;T&gt;</code>	Default hash function.
<code>Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;</code>	A hash table of unique elements used to implement <code>HashMap&lt;K, T, H, C&gt;</code> and <code>HashSet&lt;T, H, C&gt;</code> . The keys of elements need not be ordered.
<code>HashtableBase&lt;T&gt;</code>	Implementation detail.
<code>HashtableIterator&lt;T, R, P&gt;</code>	A hash table iterator type.
<code>LinkedList&lt;T&gt;</code>	A doubly linked list class.
<code>LinkedListBase</code>	Implementation detail.
<code>LinkedListNode&lt;T&gt;</code>	A linked list node type.
<code>LinkedListNodeBase</code>	Implementation detail.
<code>LinkedListNodeIterator&lt;T, R, P&gt;</code>	A linked list iterator type.
<code>List&lt;T&gt;</code>	A container of elements in which the contained elements are in consecutive locations in memory.
<code>Map&lt;Key, Value, KeyCompare&gt;</code>	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 red-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.

### 3.7.1 BitSet Class

A compact sequence of bits.

#### Syntax

```
public class BitSet;
```

##### 3.7.1.1 Example

```
using System;
using System.Collections;

// Writes:
// 1111111
// 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();
    }
}
```

{}

### 3.7.1.2 Member Functions

Member Function	Description
<code>BitSet()</code>	Constructor. Constructs an empty bit set.
<code>BitSet(const System.Collections.BitSet&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.BitSet&amp;)</code>	Copy assignment.
<code>BitSet(System.Collections.BitSet&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.BitSet&amp;&amp;)</code>	Move assignment.
<code>All() const</code>	Returns true if all the bits are 1, false otherwise.
<code>Any() const</code>	Returns true if any bit is 1, false otherwise.
<code>BitSet(const System.String&amp;)</code>	Constructor. Constructs a bit set from a string of bits.
<code>BitSet(int)</code>	Constructor. Constructs a bit set capable of holding given number of bits.
<code>Clear()</code>	Makes the bit set empty.
<code>Count() const</code>	Returns the number of bits the bit set contains.
<code>Flip()</code>	Toggles the values of the bits that the bit set contains.
<code>Flip(int)</code>	Toggles the value of the bit with the given index.
<code>None() const</code>	Returns true if all the bits are 0, false otherwise.
<code>Reset()</code>	Resets all the bits to 0.
<code>Reset(int)</code>	Resets the bit with the given index to 0.
<code>Resize(int)</code>	Resizes the bit set to contain the given number of bits.
<code>Set()</code>	Sets all the bits to 1.
<code>Set(int)</code>	Sets the bit with the given index to 1.
<code>Set(int, bool)</code>	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.
operator==(const System.Collections.BitSet&)	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.
~BitSet()	Destructor.

**BitSet() Member Function**

Constructor. Constructs an empty bit set.

**Syntax**

```
public BitSet();
```

**Implementation**

[bitset.cm, page 1](#)

**BitSet(const System.Collections.BitSet&) Member Function**

Copy constructor.

**Syntax**

```
public BitSet(const System.Collections.BitSet& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	const System.Collections.BitSet&	

**operator=(const System.Collections.BitSet&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.BitSet& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	const System.Collections.BitSet&	

**BitSet(System.Collections.BitSet&&) Member Function**

Move constructor.

**Syntax**

```
public BitSet(System.Collections.BitSet&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	<a href="#">System.Collections.BitSet&amp;&amp;</a>	

**operator=(System.Collections.BitSet&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.BitSet&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.Collections.BitSet&&	

**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](#)

**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](#)

**BitSet(const System.String&) Member Function**

Constructor. Constructs a bit set from a string of bits.

**Syntax**

```
public BitSet(const System.String& bits_);
```

**Parameters**

Name	Type	Description
bits_	const System.String&	A string of bits.

**Implementation**

[bitset.cm, page 1](#)

**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](#)

**Clear() Member Function**

Makes the bit set empty.

**Syntax**

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

**Implementation**

[bitset.cm, page 2](#)

**Count() const Member Function**

Returns the number of bits the bit set contains.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of bits the bit set contains.

**Implementation**

[bitset.cm, page 2](#)

**Flip() Member Function**

Toggles the values of the bits that the bit set contains.

**Syntax**

```
public void Flip();
```

**Implementation**

[bitset.cm, page 3](#)

**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](#)

**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](#)

**Reset() Member Function**

Resets all the bits to 0.

**Syntax**

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

**Implementation**

[bitset.cm, page 2](#)

**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](#)

**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](#)

**Set() Member Function**

Sets all the bits to 1.

**Syntax**

```
public void Set();
```

**Implementation**

[bitset.cm, page 2](#)

**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](#)

**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](#)

**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](#)

**ToString() const Member Function**

Returns the string representation of the bit set.

**Syntax**

```
public System.String ToString() const;
```

**Returns**

[System.String](#)

Returns the string representation of the bit set.

**Implementation**

[bitset.cm, page 5](#)

**operator==(const System.Collections.BitSet&) const Member Function**

Compares this bit set and given bit set for equality.

**Syntax**

```
public bool operator==(const System.Collections.BitSet& that) const;
```

**Parameters**

Name	Type	Description
that	const System.Collections.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](#)

**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](#)

**~BitSet() Member Function**

Destructor.

**Syntax**

```
public ~BitSet();
```

### 3.7.2 Bucket<T> Class

A hash table bucket.

#### Syntax

```
public class Bucket<T>;
```

#### Constraint

T is [Semiregular](#)

### 3.7.2.1 Type Definitions

Name	Type	Description
ValueType	T	Type of value contained by the bucket.

### 3.7.2.2 Member Functions

Member Function	Description
Bucket<T>()	Default constructor.
Bucket<T>(const System.Collections.Bucket<T>&)	Copy constructor.
operator=(const System.Collections.Bucket<T>&)	Copy assignment.
Bucket<T>(System.Collections.Bucket<T>&&)	Move constructor.
operator=(System.Collections.Bucket<T>&&)	Move assignment.
Bucket<T>(const T&, System.Collections.Bucket<T>*)	Constructor. Initializes the bucket with a pointer to the next bucket and a value.
Next() const	Returns a pointer to the next bucket.
SetNext(System.Collections.Bucket<T>*)	Sets the pointer to the next bucket.
Value() const	Returns a constant reference to the value contained by the bucket.

**Bucket<T>() Member Function**

Default constructor.

**Syntax**

```
public Bucket<T>();
```

**Bucket<T>(const System.Collections.Bucket<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Bucket<T>(const System.Collections.Bucket<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Bucket<T>&	Argument to copy.

**operator=(const System.Collections.Bucket<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.Bucket<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Bucket<T>&	Argument to assign.

**Bucket<T>(System.Collections.Bucket<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Bucket<T>(System.Collections.Bucket<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.Bucket&lt;T&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.Collections.Bucket<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Bucket<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.Bucket&lt;T&gt;&amp;&amp;</a>	Argument to assign from.

**Bucket<T>(const T&, System.Collections.Bucket<T>\*) Member Function**

Constructor. Initializes the bucket with a pointer to the next bucket and a value.

**Syntax**

```
public Bucket<T>(const T& value_, System.Collections.Bucket<T>* next_);
```

**Parameters**

Name	Type	Description
value_	const T&	A value contained by the bucket.
next_	System.Collections.Bucket<T>*	Pointer to the next bucket.

**Next() const Member Function**

Returns a pointer to the next bucket.

**Syntax**

```
public System.Collections.Bucket<T>* Next() const;
```

**Returns**

[System.Collections.Bucket<T>\\*](#)

Returns a pointer to the next bucket.

**SetNext(System.Collections.Bucket<T>\*) Member Function**

Sets the pointer to the next bucket.

**Syntax**

```
public void SetNext(System.Collections.Bucket<T>* next_);
```

**Parameters**

Name	Type	Description
next_	<a href="#">System.Collections.Bucket&lt;T&gt;*</a>	A pointer to the next bucket.

**Value() const Member Function**

Returns a constant reference to the value contained by the bucket.

**Syntax**

```
public const T& Value() const;
```

**Returns**

const T&

Returns a constant reference to the value contained by the bucket.

### 3.7.3 `ForwardList<T>` Class

A singly linked list of elements.

#### Syntax

```
public class ForwardList<T>;
```

#### Constraint

T is [Regular](#)

#### Model of

[ForwardContainer<T>](#)

#### 3.7.3.1 Remarks

`ForwardList<T>` is suitable only for very short sequences of items. Often a [List<T>](#) is more appropriate.

### 3.7.3.2 Type Definitions

Name	Type	Description
ConstIterator	System.Collections.- ForwardListNodeIterator<T, const T&, const T*>	A constant iterator type.
Iterator	System.Collections.- ForwardListNodeIterator<T, T&, T*>	An iterator type.
ValueType	T	The type of the contained element.

### 3.7.3.3 Member Functions

Member Function	Description
ForwardList<T>()	Constructor. Constructs an empty forward list.
ForwardList<T>(const ForwardList<T>&)	Copy constructor.
operator=(const ForwardList<T>&)	Copy assignment.
ForwardList<T>(System.Collections.-ForwardList<T>&&)	Move constructor.
operator=(System.Collections.ForwardList<T>&&)	Move assignment.
Begin()	Returns an iterator pointing to the first element of the forward list, or <a href="#">End()</a> if the forward list is empty.
Begin() const	Returns a constant iterator pointing to the first element of the forward list, or <a href="#">CEnd() const</a> if the forward list is empty.
CBegin() const	Returns a constant iterator pointing to the first element of the forward list, or <a href="#">CEnd() const</a> 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(System.Collections.- ForwardListNodeIterator<T, T&, T*>, const T&)	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 T&)	Inserts an element to the front of the forward list.
IsEmpty() const	Returns true if the forward list is empty, false otherwise.
Remove(const T&)	Removes all occurrences of the given value from the forward list.
RemoveAfter(System.Collections.- ForwardListNodeIterator<T, T&, T*>)	Removes an element after the position pointed by the given iterator.
RemoveFront()	Removes an element from the front of the forward list.
~ForwardList<T>()	Destructor.

**ForwardList<T>() Member Function**

Constructor. Constructs an empty forward list.

**Syntax**

```
public ForwardList<T>();
```

**Implementation**

[fwdlist.cm, page 3](#)

**ForwardList<T>(const System.Collections.ForwardList<T>&) Member Function**

Copy constructor.

**Syntax**

```
public ForwardList<T>(const System.Collections.ForwardList<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.ForwardList<T>&	A forward list to copy.

**Implementation**

[fwlist.cm, page 3](#)

**operator=(const System.Collections.ForwardList<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.ForwardList<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.ForwardList<T>&	A forward list to assign.

**Implementation**

[fwlist.cm, page 3](#)

**ForwardList<T>(System.Collections.ForwardList<T>&&) Member Function**

Move constructor.

**Syntax**

```
public ForwardList<T>(System.Collections.ForwardList<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardList<T>&&	A forward list to move from.

**Implementation**

[fwlist.cm, page 3](#)

**operator=(System.Collections.ForwardList<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.ForwardList<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardList<T>&&	A forward list to move from.

**Implementation**

[fwlist.cm](#), page 3

**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 System.Collections.ForwardListNodeIterator<T, T&, T*> Begin();
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, T&, T\\*>](#)

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](#)

**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 System.Collections.ForwardListNodeIterator<T, const T&, const T*> Begin() const;
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, const T&, const T\\*>](#)

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](#)

**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 System.Collections.ForwardListNodeIterator<T, const T&, const T*> CBegin()  
const;
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, const T&, const T\\*>](#)

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](#)

**CEnd() const Member Function**

Returns a constant iterator pointing to one past the end of the forward list.

**Syntax**

```
public System.Collections.ForwardListNodeIterator<T, const T&, const T*> CEnd() const;
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to one past the end of the forward list.

**Implementation**

[fwdlist.cm, page 4](#)

**Clear() Member Function**

Makes the forward list empty.

**Syntax**

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

**Implementation**

[fwdlist.cm, page 3](#)

**Count() const Member Function**

Counts the number of elements in the forward list.

**Syntax**

```
public 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](#)

**End() Member Function**

Returns an iterator pointing to one past the end of the forward list.

**Syntax**

```
public System.Collections.ForwardListNodeIterator<T, T&, T*> End();
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to one past the end of the forward list.

**Implementation**

[fwlist.cm, page 4](#)

**End() const Member Function**

Returns a constant iterator pointing to one past the end of the forward list.

**Syntax**

```
public System.Collections.ForwardListNodeIterator<T, const T&, const T*> End() const;
```

**Returns**

[System.Collections.ForwardListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to one past the end of the forward list.

**Implementation**

[fwdlist.cm, page 4](#)

**Front() const Member Function**

Returns a constant reference to the first element in the forward list.

**Syntax**

```
public const T& Front() const;
```

**Returns**

const T&

Returns a constant reference to the first element in the forward list.

**Implementation**

[fwlist.cm, page 4](#)

**InsertAfter(System.Collections.ForwardListNodeIterator<T, T&, T\*>, const T&) 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 System.Collections.ForwardListNodeIterator<T, T&, T*> InsertAfter(System.Collections.ForwardList<T, T*> pos, const T& value);
```

**Parameters**

Name	Type	Description
pos	<a href="#">System.Collections.ForwardListNodeIterator&lt;T, T&amp;, T*&gt;</a>	An iterator.
value	const T&	A value to insert.

**Returns**

[System.Collections.ForwardListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**Implementation**

[fwdlist.cm, page 4](#)

**InsertFront(const T&)** Member Function

Inserts an element to the front of the forward list.

**Syntax**

```
public System.Collections.ForwardListNodeIterator<T, T&, T*> InsertFront(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	A value to insert.

**Returns**

[System.Collections.ForwardListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**Implementation**

[fwdlist.cm, page 4](#)

**IsEmpty() const Member Function**

Returns true if the forward list is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the forward list is empty, false otherwise.

**Implementation**

[fwlist.cm, page 3](#)

**Remove(const T&)** Member Function

Removes all occurrences of the given value from the forward list.

**Syntax**

```
public void Remove(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	A value to remove.

**Implementation**

[fwlist.cm, page 5](#)

**RemoveAfter(System.Collections.ForwardListNodeIterator<T, T&, T\*>) Member Function**

Removes an element after the position pointed by the given iterator.

**Syntax**

```
public void RemoveAfter(System.Collections.ForwardListNodeIterator<T, T&, T*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.ForwardListNodeIterator<T, T&, T*>	An iterator.

**Implementation**

[fwdlist.cm, page 5](#)

**RemoveFront() Member Function**

Removes an element from the front of the forward list.

**Syntax**

```
public void RemoveFront();
```

**Implementation**

[fwdlist.cm, page 4](#)

**`~ForwardList<T>()` Member Function**

Destructor.

**Syntax**

```
public ~ForwardList<T>();
```

**Implementation**

[fwdlist.cm, page 3](#)

**3.7.3.4 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.Collections.-ForwardList&lt;T&gt;&amp;, const System.Collections.-ForwardList&lt;T&gt;&amp;)</code>	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.
<code>operator==(const System.Collections.-ForwardList&lt;T&gt;&amp;, const System.Collections.-ForwardList&lt;T&gt;&amp;)</code>	Compares two forward lists for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.

**operator<(const System.Collections.ForwardList<T>&, const System.Collections.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<(const System.Collections.ForwardList<T>& left, const System.Collections.F
```

**Constraint**

T is [TotallyOrdered](#)

**Parameters**

Name	Type	Description
left	const System.Collections.ForwardList<T>&	The first forward list.
right	const System.Collections.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

**operator==(const System.Collections.ForwardList<T>&, const System.Collections.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==(const System.Collections.ForwardList<T>& left, const System.Collections.ForwardList<T>& right);
```

**Constraint**

T is [Regular](#)

**Parameters**

Name	Type	Description
left	const System.Collections.ForwardList<T>&	The first forward list.
right	const System.Collections.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

### 3.7.4 ForwardListNode<T> Class

A forward list node type.

#### Syntax

```
public class ForwardListNode<T>;
```

#### Constraint

T is [Semiregular](#)

#### 3.7.4.1 Member Functions

Member Function	Description
ForwardListNode<T>()	Default constructor.
ForwardListNode<T>(const System.Collections.ForwardListNode<T>&)	Copy constructor.
operator=(const System.Collections.ForwardListNode<T>&)	Copy assignment.
ForwardListNode<T>(System.Collections.-ForwardListNode<T>&&)	Move constructor.
operator=(System.Collections.-ForwardListNode<T>&&)	Move assignment.
ForwardListNode<T>(System.Collections.-ForwardListNode<T>*, const T&)	Constructor. Initializes the forward list node with a pointer to the next forward list node and a value.
Next() const	Returns a pointer to the next forward list node.
SetNext(System.Collections.ForwardListNode<T>*)	Sets the pointer to the next forward list node.
Value()	Returns a reference to the value contained by the forward list node.
Value() const	Returns a constant reference to the value contained by the forward list node.

**ForwardListNode<T>() Member Function**

Default constructor.

**Syntax**

```
public ForwardListNode<T>();
```

**ForwardListNode<T>(const System.Collections.ForwardListNode<T>&)** Member Function

Copy constructor.

**Syntax**

```
public ForwardListNode<T>(const System.Collections.ForwardListNode<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.ForwardListNode<T>&	Argument to copy.

**operator=(const System.Collections.ForwardListNode<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.ForwardListNode<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.ForwardListNode<T>&	Argument to assign.

**ForwardListNode<T>(System.Collections.ForwardListNode<T>&&) Member Function**

Move constructor.

**Syntax**

```
public ForwardListNode<T>(System.Collections.ForwardListNode<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardListNode<T>&&	Argument to move from.

**operator=(System.Collections.ForwardListNode<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.ForwardListNode<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardListNode<T>&&	Argument to assign from.

**ForwardListNode<T>(System.Collections.ForwardListNode<T>\*, const T&) Member Function**

Constructor. Initializes the forward list node with a pointer to the next forward list node and a value.

**Syntax**

```
public ForwardListNode<T>(System.Collections.ForwardListNode<T>* next_, const T& value_);
```

**Parameters**

Name	Type	Description
next_	System.Collections.ForwardListNode<T>*	A pointer to the next forward list node.
value_	const T&	A value.

**Next() const Member Function**

Returns a pointer to the next forward list node.

**Syntax**

```
public System.Collections.ForwardListNode<T>* Next() const;
```

**Returns**

[System.Collections.ForwardListNode<T>\\*](#)

Returns a pointer to the next forward list node.

**SetNext(System.Collections.ForwardListNode<T>\*) Member Function**

Sets the pointer to the next forward list node.

**Syntax**

```
public void SetNext(System.Collections.ForwardListNode<T>* next_);
```

**Parameters**

Name	Type	Description
next_	System.Collections.ForwardListNode<T>*	A pointer to the next forward list node.

**Value() Member Function**

Returns a reference to the value contained by the forward list node.

**Syntax**

```
public T& Value();
```

**Returns**

T&

Returns a reference to the value contained by the forward list node.

**Value() const Member Function**

Returns a constant reference to the value contained by the forward list node.

**Syntax**

```
public const T& Value() const;
```

**Returns**

```
const T&
```

Returns a constant reference to the value contained by the forward list node.

### 3.7.5 **ForwardListNodeIterator<T, R, P>** Class

A forward iterator that iterates through a [ForwardList<T>](#).

#### Syntax

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

#### Model of

[ForwardIterator<T>](#)

### 3.7.5.1 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.

### 3.7.5.2 Member Functions

Member Function	Description
<code>ForwardListNodeIterator&lt;T, R, P&gt;()</code>	Constructor. Default constructs a forward list node iterator.
<code>ForwardListNodeIterator&lt;T, R, P&gt;(const System.Collections.ForwardListNodeIterator&lt;T, R, P&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.ForwardListNodeIterator&lt;T, R, P&gt;&amp;)</code>	Copy assignment.
<code>ForwardListNodeIterator&lt;T, R, P&gt;(System.Collections.ForwardListNodeIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.ForwardListNodeIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move assignment.
<code>ForwardListNodeIterator&lt;T, R, P&gt;(System.Collections.ForwardListNode&lt;T&gt;*)</code>	Constructor. Constructs a forward list node iterator pointing to a forward list node.
<code>GetNode() const</code>	Returns the contained pointer to an element.
<code>operator*() const</code>	Returns a reference to an element.
<code>operator++()</code>	Advances the iterator pointing to the next element in the forward list.
<code>operator-&gt;() const</code>	Returns a pointer to an element.

**ForwardListNodeIterator<T, R, P>() Member Function**

Constructor. Default constructs a forward list node iterator.

**Syntax**

```
public ForwardListNodeIterator<T, R, P>();
```

**Implementation**

[fwdlist.cm, page 2](#)

**ForwardListNodeIterator<T, R, P>(const System.Collections.ForwardListNodeIterator<T, R, P>&) Member Function**

Copy constructor.

**Syntax**

```
public ForwardListNodeIterator<T, R, P>(const System.Collections.ForwardListNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const <a href="#">System.Collections.ForwardListNodeIterator&lt;T, R, P&gt;&amp;</a>	Argument to copy.

**operator=(const System.Collections.ForwardListNodeIterator<T, R, P>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.ForwardListNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.ForwardListNodeIterator<T, R, P>&	Argument to assign.

**ForwardListNodeIterator<T, R, P>(System.Collections.ForwardListNodeIterator<T, R, P>&&) Member Function**

Move constructor.

**Syntax**

```
public ForwardListNodeIterator<T, R, P>(System.Collections.ForwardListNodeIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardListNodeIterator<T, R, P>&&	Argument to move from.

**operator=(System.Collections.ForwardListNodeIterator<T, R, P>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.ForwardListNodeIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.ForwardListNodeIterator<T, P>&&	R, Argument to assign from.

**ForwardListNodeIterator<T, R, P>(System.Collections.ForwardListNode<T>\*) Member Function**

Constructor. Constructs a forward list node iterator pointing to a forward list node.

**Syntax**

```
public ForwardListNodeIterator<T, R, P>(System.Collections.ForwardListNode<T>* node_);
```

**Parameters**

Name	Type	Description
node_	System.Collections.ForwardListNode<T>*	A pointer to a forward list node.

**Implementation**

[fwdlist.cm, page 2](#)

**GetNode() const Member Function**

Returns the contained pointer to an element.

**Syntax**

```
public System.Collections.ForwardListNode<T>* GetNode() const;
```

**Returns**

[System.Collections.ForwardListNode<T>\\*](#)

Returns the contained pointer to an element.

**Implementation**

[fwlist.cm, page 2](#)

**operator\*() const Member Function**

Returns a reference to an element.

**Syntax**

```
public R operator*() const;
```

**Returns**

R

Returns a reference to an element.

**Implementation**

[fwlist.cm, page 2](#)

**operator++() Member Function**

Advances the iterator pointing to the next element in the forward list.

**Syntax**

```
public System.Collections.ForwardListNodeIterator<T, R, P>& operator++();
```

**Returns**

`System.Collections.ForwardListNodeIterator<T, R, P>&`

Returns a reference to the iterator.

**Implementation**

[fwlist.cm, page 2](#)

**operator->() const Member Function**

Returns a pointer to an element.

**Syntax**

```
public P operator->() const;
```

**Returns**

P

Returns a pointer to an element.

**Implementation**

[fwdlist.cm, page 2](#)

### 3.7.5.3 Nonmember Functions

Function	Description
<code>operator==(System.Collections.- ForwardListNodeIterator&lt;T, R, P&gt;, System. Collections.ForwardListNodeIterator&lt;T, R, P&gt;)</code>	Compares two forward list node iterators for equality.

**operator==(System.Collections.ForwardListNodeIterator<T, R, P>, System.Collections.ForwardListNodeIterator<T, R, P>) Function**

Compares two forward list node iterators for equality.

**Syntax**

```
public bool operator==(System.Collections.ForwardListNodeIterator<T, R, P> left,  
System.Collections.ForwardListNodeIterator<T, R, P> right);
```

**Parameters**

Name	Type	Description
left	System.Collections.ForwardListNodeIterator<T, R, P>	R, The first forward list node iterator.
right	System.Collections.ForwardListNodeIterator<T, R, P>	R, 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**

[fwlist.cm](#), page 6

### 3.7.6 `HashMap<K, T, H, C>` Class

An associative container of key-value pairs organized in a hash table. The keys need not be ordered.

#### Syntax

```
public class HashMap<K, T, H, C>;
```

#### Constraint

K is [Semiregular](#) and T is [Semiregular](#) and HashFunction<H, K> and C is Relation and  
C.Domain is K

#### Model of

[ForwardContainer<T>](#)

#### Default Template Arguments

H = [System.Collections.Hasher<K>](#)

C = [System.EqualTo<K>](#)

### 3.7.6.1 Type Definitions

Name	Type	Description
Compare	C	A relation used to compare keys.
ConstIterator	System.Collections.HashtableIterator<- System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*>	A constant iterator type.
HashFun	H	A hash function type.
Iterator	System.Collections.HashtableIterator<- System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*>	An iterator type.
KeyType	K	The type of the key.
MappedType	T	The type associated with the key.
ValueType	System.Pair<K, T>	A pair composed of key type and mapped type.

### 3.7.6.2 Member Functions

Member Function	Description
HashMap<K, T, H, C>()	Default constructor.
HashMap<K, T, H, C>(const System.- Collections.HashMap<K, T, H, C>&)	Copy constructor.
operator=(const System.Collections.HashMap<- K, T, H, C>&)	Copy assignment.
HashMap<K, T, H, C>(System.Collections.- HashMap<K, T, H, C>&&)	Move constructor.
operator=(System.Collections.HashMap<K, T, H, C>&&)	Move assignment.
Begin()	Returns an iterator to the beginning of the hash map, or <a href="#">End()</a> if the hash map is empty.
Begin() const	Returns a constant iterator to the beginning of the hash map, or <a href="#">CEnd() const</a> if the hash map is empty.
CBegin() const	Returns a constant iterator to the beginning of the hash map, or <a href="#">CEnd() const</a> if the hash map is empty.
CEnd() const	Returns a constant iterator pointing one past the end of the hash map.

<code>CFind(const K&amp;) const</code>	Searches a key from the hash map and returns a constant iterator pointing to the found element, or <code>CEnd() const</code> if the key is not found in the hash map.
<code>Clear()</code>	Makes the hash map empty.
<code>Count() const</code>	Returns the number of elements in the hash map.
<code>End()</code>	Returns an iterator pointing one past the end of the hash map.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the hash map.
<code>Find(const K&amp;)</code>	Searches a key from the hash map and returns an iterator pointing to the found element, or <code>End()</code> if the key is not found in the hash map.
<code>Find(const K&amp;) const</code>	Searches a key from the hash map and returns a constant iterator pointing to the found element, or <code>CEnd() const</code> if the key is not found in the hash map.
<code>Insert(const System.Pair&lt;K, T&gt;&amp;)</code>	Inserts a key-value pair to the hash map if the hash map does not already contain the key. In that case returns a pair consisting a pair of an iterator pointing to the inserted element and <code>true</code> . Otherwise does not insert an element, but returns a pair consisting of an iterator pointing to the previously inserted element and <code>false</code> .
<code>IsEmpty() const</code>	Returns true if the hash map is empty, false otherwise.
<code>Remove(System.Collections.HashtableIterator&lt;-System.Pair&lt;K, T&gt;, System.Pair&lt;K, T&gt;&amp;, System.Pair&lt;K, T&gt;*&gt;)</code> <code>Remove(const K&amp;)</code>	Removes an element pointed by the given iterator from the hash map.
<code>operator[](const K&amp;)</code>	Removes an element with the given key from the hash map.
	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 to the hash map.



**HashMap<K, T, H, C>() Member Function**

Default constructor.

**Syntax**

```
public HashMap<K, T, H, C>();
```

**HashMap<K, T, H, C>(const System.Collections.HashMap<K, T, H, C>&) Member Function**

Copy constructor.

**Syntax**

```
public HashMap<K, T, H, C>(const System.Collections.HashMap<K, T, H, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.HashMap<K, T, H, C>&	Argument to copy.

**operator=(const System.Collections.HashMap<K, T, H, C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.HashMap<K, T, H, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.HashMap<K, T, H, C>&	Argument to assign.

**HashMap<K, T, H, C>(System.Collections.HashMap<K, T, H, C>&&) Member Function**

Move constructor.

**Syntax**

```
public HashMap<K, T, H, C>(System.Collections.HashMap<K, T, H, C>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.HashMap<K, T, H, C>&&	Argument to move from.

**operator=(System.Collections.HashMap<K, T, H, C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.HashMap<K, T, H, C>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.HashMap<K, T, H, C>&&	Argument to assign from.

**Begin() Member Function**

Returns an iterator to the beginning of the hash map, or [End\(\)](#) if the hash map is empty.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*> Begin();
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>\\*>](#)

Returns an iterator to the beginning of the hash map, or [End\(\)](#) if the hash map is empty.

**Begin() const Member Function**

Returns a constant iterator to the beginning of the hash map, or [CEnd\(\) const](#) if the hash map is empty.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> Begin() const;
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator to the beginning of the hash map, or [CEnd\(\) const](#) if the hash map is empty.

**CBegin() const Member Function**

Returns a constant iterator to the beginning of the hash map, or [CEnd\(\) const](#) if the hash map is empty.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> CBegin() const;
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator to the beginning of the hash map, or [CEnd\(\) const](#) if the hash map is empty.

**CEnd() const Member Function**

Returns a constant iterator pointing one past the end of the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> CEnd() const;
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator pointing one past the end of the hash map.

**CFind(const K&) const Member Function**

Searches a key from the hash map and returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) if the key is not found in the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> CFind(const K& key) const;
```

**Parameters**

Name	Type	Description
key	const K&	A key to search.

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) if the key is not found in the hash map.

**Clear() Member Function**

Makes the hash map empty.

**Syntax**

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

**Count() const Member Function**

Returns the number of elements in the hash map.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the hash map.

**End() Member Function**

Returns an iterator pointing one past the end of the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*> End();
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>\\*>](#)

Returns an iterator pointing one past the end of the hash map.

**End() const Member Function**

Returns a constant iterator pointing one past the end of the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> End() const;
```

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator pointing one past the end of the hash map.

**Find(const K&) Member Function**

Searches a key from the hash map and returns an iterator pointing to the found element, or [End\(\)](#) if the key is not found in the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*> Find(const K& key);
```

**Parameters**

Name	Type	Description
key	const K&	A key to search.

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>\\*>](#)

Returns an iterator pointing to the found element, or [End\(\)](#) if the key is not found in the hash map.

**Find(const K&) const Member Function**

Searches a key from the hash map and returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) if the key is not found in the hash map.

**Syntax**

```
public System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>*> Find(const K& key) const;
```

**Parameters**

Name	Type	Description
key	const K&	A key to search.

**Returns**

[System.Collections.HashtableIterator<System.Pair<K, T>, const System.Pair<K, T>&, const System.Pair<K, T>\\*>](#)

Returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) if the key is not found in the hash map.

**Insert(const System.Pair<K, T>&) Member Function**

Inserts a key-value pair to the hash map if the hash map does not already contain the key. In that case returns a pair consisting a pair 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 System.Pair<System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*>, bool> Insert(const System.Pair<K, T>& value);
```

**Parameters**

Name	Type	Description
value	const System.Pair<K, T>&	A key-value pair to insert.

**Returns**

System.Pair<System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>\*>, 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 hash map.

**IsEmpty() const Member Function**

Returns true if the hash map is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the hash map is empty, false otherwise.

**Remove(System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>\*>) Member Function**

Removes an element pointed by the given iterator from the hash map.

**Syntax**

```
public void Remove(System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.HashtableIterator<System.Pair<K, T>, System.Pair<K, T>&, System.Pair<K, T>*>	Iterator pointing to an element to remove.

**Remove(const K&) Member Function**

Removes an element with the given key from the hash map.

**Syntax**

```
public void Remove(const K& key);
```

**Parameters**

Name	Type	Description
key	const K&	Key of element to remove.

**operator[](const K&) 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 to the hash map.

**Syntax**

```
public T& operator[](const K& key);
```

**Parameters**

Name	Type	Description
key	const K&	A key.

**Returns**

T&

A value associated with the key.

**3.7.6.3 Nonmember Functions**

Function	Description
operator==(const System.Collections. HashMap<K, T, H, C>&, const System. Collections.HashMap<K, T, H, C>&)	Compares two hash maps for equality and returns true if the first hash map contains same number of equal elements as the second hash map, false otherwise.

**operator==(const System.Collections.HashMap<K, T, H, C>&, const System.Collections.HashMap<K, T, H, C>&) Function**

Compares two hash maps for equality and returns true if the first hash map contains same number of equal elements as the second hash map, false otherwise.

**Syntax**

```
public bool operator==(const System.Collections.HashMap<K, T, H, C>& left, const System.Collections.HashMap<K, T, H, C>& right);
```

**Constraint**

K is [Semiregular](#) and T is [Semiregular](#) and [HashFunction<H, K>](#) and C is [Relation](#) and C.Domain is K

**Parameters**

Name	Type	Description
left	const System.Collections.HashMap<K, T, H, C>&	The first hash map.
right	const System.Collections.HashMap<K, T, H, C>&	The second hash map.

**Returns**

bool

Returns true if the first hash map contains same number of equal elements as the second hash map, false otherwise

### 3.7.7 HashSet<T, H, C> Class

A set of unique elements organized in a hash table. The elements need not be ordered.

#### Syntax

```
public class HashSet<T, H, C>;
```

#### Constraint

T is [Semiregular](#) and [HashFunction<H, T>](#) and C is [Relation](#) and C.Domain is T

#### Model of

[ForwardContainer<T>](#)

#### Default Template Arguments

H = [System.Collections.Hasher<T>](#)

C = [System.EqualTo<T>](#)

### 3.7.7.1 Type Definitions

Name	Type	Description
Compare	C	A relation used to compare elements.
ConstIterator	System.Collections.HashtableIterator<T, const T&, const T*>	A constant iterator type.
HashFun	H	A hash function type.
Iterator	System.Collections.HashtableIterator<T, T&, T*>	An iterator type.
KeyType	T	The key type is equal to the <b>ValueType</b> for the hash table.
ValueType	T	The type of the element in the hash set.

### 3.7.7.2 Member Functions

Member Function	Description
HashSet<T, H, C>()	Default constructor.
HashSet<T, H, C>(const System.Collections.- HashSet<T, H, C>&)	Copy constructor.
operator=(const System.Collections.HashSet<T, H, C>&)	Copy assignment.
HashSet<T, H, C>(System.Collections.- HashSet<T, H, C>&&)	Move constructor.
operator=(System.Collections.HashSet<T, H, C>&&)	Move assignment.
Begin()	Returns an iterator pointing to the beginning of the hash set, or <b>End()</b> if the hash set is empty.
Begin() const	Returns a constant iterator pointing to the beginning of the hash set, or <b>CEnd() const</b> if the hash set is empty.
CBegin() const	Returns a constant iterator pointing to the beginning of the hash set, or <b>CEnd() const</b> if the hash set is empty.
CEnd() const	Returns a constant iterator pointing one past the end of the hash set.
CFind(const T&) const	Searches an element in the hash set and returns a constant iterator pointing to it if found, or <b>CEnd() const</b> otherwise.

<code>Clear()</code>	Makes the hash set empty.
<code>Count() const</code>	Returns the number of elements in the hash set.
<code>End()</code>	Returns an iterator pointing one past the end of the hash set.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the hash set.
<code>Find(const T&amp;)</code>	Searches an element in the hash set and returns an iterator pointing to it if found, or <code>End()</code> iterator otherwise.
<code>Find(const T&amp;) const</code>	Searches an element in the hash set and returns a constant iterator pointing to it if found, or <code>CEnd() const</code> iterator otherwise.
<code>Insert(const T&amp;)</code>	Inserts an element into the hash set, if it is not already there.
<code>IsEmpty() const</code>	Returns true if the hash set is empty, false otherwise.
<code>Remove(System.Collections.HashtableIterator&lt; T, T&amp;, T*&gt;)</code>	Removes an element pointed by the given iterator from the hash set.
<code>Remove(const T&amp;)</code>	Removes an element from the hash set. If element was not found, does nothing.

**HashSet<T, H, C>() Member Function**

Default constructor.

**Syntax**

```
public HashSet<T, H, C>();
```

**HashSet<T, H, C>(const System.Collections.HashSet<T, H, C>&) Member Function**

Copy constructor.

**Syntax**

```
public HashSet<T, H, C>(const System.Collections.HashSet<T, H, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.HashSet<T, H, C>&	Argument to copy.

**operator=(const System.Collections.HashSet<T, H, C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.HashSet<T, H, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.HashSet<T, H, C>&	Argument to assign.

**HashSet<T, H, C>(System.Collections.HashSet<T, H, C>&&) Member Function**

Move constructor.

**Syntax**

```
public HashSet<T, H, C>(System.Collections.HashSet<T, H, C>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.HashSet&lt;T, H, C&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.Collections.HashSet<T, H, C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.HashSet<T, H, C>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.HashSet&lt;T, H, C&gt;&amp;&amp;</a>	Argument to assign from.

**Begin() Member Function**

Returns an iterator pointing to the beginning of the hash set, or [End\(\)](#) if the hash set is empty.

**Syntax**

```
public System.Collections.HashtableIterator<T, T&, T*> Begin();
```

**Returns**

[System.Collections.HashtableIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the beginning of the hash set, or [End\(\)](#) if the hash set is empty.

**Begin() const Member Function**

Returns a constant iterator pointing to the beginning of the hash set, or [CEnd\(\) const](#) if the hash set is empty.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> Begin() const;
```

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginning of the hash set, or [CEnd\(\) const](#) if the hash set is empty.

**CBegin() const Member Function**

Returns a constant iterator pointing to the beginning of the hash set, or [CEnd\(\) const](#) if the hash set is empty.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> CBegin() const;
```

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginning of the hash set, or [CEnd\(\) const](#) if the hash set is empty.

**CEnd() const Member Function**

Returns a constant iterator pointing one past the end of the hash set.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> CEnd() const;
```

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the hash set.

**CFind(const T&) const Member Function**

Searches an element in the hash set and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> CFind(const T& key) const;
```

**Parameters**

Name	Type	Description
key	const T&	An element to search.

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the found element if the search was successful, or [CEnd\(\) const](#) iterator otherwise.

**Clear() Member Function**

Makes the hash set empty.

**Syntax**

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

**Count() const Member Function**

Returns the number of elements in the hash set.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the hash set.

**End() Member Function**

Returns an iterator pointing one past the end of the hash set.

**Syntax**

```
public System.Collections.HashtableIterator<T, T&, T*> End();
```

**Returns**

[System.Collections.HashtableIterator<T, T&, T\\*>](#)

Returns an iterator pointing one past the end of the hash set.

**End() const Member Function**

Returns a constant iterator pointing one past the end of the hash set.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> End() const;
```

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the hash set.

**Find(const T&) Member Function**

Searches an element in the hash set and returns an iterator pointing to it if found, or [End\(\)](#) iterator otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<T, T&, T*> Find(const T& key);
```

**Parameters**

Name	Type	Description
key	const T&	An element to seach.

**Returns**

[System.Collections.HashtableIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the found element if the search was successful, or [End\(\)](#) iterator otherwise.

**Find(const T&) const Member Function**

Searches an element in the hash set and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<T, const T&, const T*> Find(const T& key) const;
```

**Parameters**

Name	Type	Description
key	const T&	An element to seach.

**Returns**

[System.Collections.HashtableIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the found element if the search was successful, or [CEnd\(\) const](#) iterator otherwise.

**Insert(const T&)** Member Function

Inserts an element into the hash set, if it is not already there.

**Syntax**

```
public System.Pair<System.Collections.HashtableIterator<T, T&, T*>, bool> Insert(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	An element to insert.

**Returns**

[System.Pair<System.Collections.HashtableIterator<T, T&, T\\*>, 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.

**IsEmpty() const Member Function**

Returns true if the hash set is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the hash set is empty, false otherwise.

**Remove(System.Collections.HashtableIterator<T, T&, T\*>) Member Function**

Removes an element pointed by the given iterator from the hash set.

**Syntax**

```
public void Remove(System.Collections.HashtableIterator<T, T&, T*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.HashtableIterator<T, T&, T*>	An iterator pointing to the element to remove.

**Remove(const T&)** Member Function

Removes an element from the hash set. If element was not found, does nothing.

**Syntax**

```
public void Remove(const T& key);
```

**Parameters**

Name	Type	Description
key	const T&	An element to remove.

**3.7.7.3 Nonmember Functions**

Function	Description
operator==(const System.Collections.HashSet<T, H, C>&, const System.Collections.HashSet<T, H, C>&)	Compares two hash sets for equality and returns true if the first hash set contains same number of equal elements as the second hash set, false otherwise.

**operator==(const System.Collections.HashSet<T, H, C>&, const System.Collections.HashSet<T, H, C>&) Function**

Compares two hash sets for equality and returns true if the first hash set contains same number of equal elements as the second hash set, false otherwise.

**Syntax**

```
public bool operator==(const System.Collections.HashSet<T, H, C>& left, const System.Collections.HashSet<T, H, C>& right);
```

**Constraint**

T is [Semiregular](#) and [HashFunction<H, T>](#) and C is [Relation](#) and C.Domain is T

**Parameters**

Name	Type	Description
left	const System.Collections.HashSet<T, H, C>&	The first hash set.
right	const System.Collections.HashSet<T, H, C>&	The second hash set.

**Returns**

bool

Returns true if the first hash set contains same number of equal elements as the second hash set, false otherwise.

### 3.7.8 Hasher<T> Class

Default hash function.

#### Syntax

```
public class Hasher<T>;
```

#### Model of

[HashFunction<T, Key>](#)

#### Base Class

[System.UnaryFun<T, ulong>](#)

#### 3.7.8.1 Member Functions

Member Function	Description
<code>Hasher&lt;T&gt;()</code>	Default constructor.
<code>Hasher&lt;T&gt;(const System.Collections.Hasher&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.Hasher&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Hasher&lt;T&gt;(System.Collections.Hasher&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.Hasher&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>operator()(const T&amp;)</code>	Calls the overloaded <i>GetHashCode(T)</i> function to return a hash code for the given key.

**Hasher<T>() Member Function**

Default constructor.

**Syntax**

```
public Hasher<T>();
```

**Hasher<T>(const System.Collections.Hasher<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Hasher<T>(const System.Collections.Hasher<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Hasher<T>&	Argument to copy.

**operator=(const System.Collections.Hasher<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.Hasher<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Hasher<T>&	Argument to assign.

**Hasher<T>(System.Collections.Hasher<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Hasher<T>(System.Collections.Hasher<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.Hasher&lt;T&gt;&amp;&amp;</a>	Argument to move from.

**operator=(System.Collections.Hasher<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Hasher<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.Hasher&lt;T&gt;&amp;&amp;</a>	Argument to assign from.

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

Calls the overloaded *GetHashCode(T)* function to return a hash code for the given key.

**Syntax**

```
public ulong operator()(const T& x);
```

**Parameters**

Name	Type	Description
x	const T&	A key.

**Returns**

ulong

Returns a hash code for the given key.

**Remarks**

By overloading the *GetHashCode(T)* function for the key type, one can use the default hash function as is.

### 3.7.9 `Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>` Class

A hash table of unique elements used to implement `HashMap<K, T, H, C>` and `HashSet<T, H, C>`. The keys of elements need not be ordered.

#### Syntax

```
public class Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>;
```

#### Constraint

`KeyType` is [Semiregular](#) and `ValueType` is [Semiregular](#) and `KeySelectionFunction<KeyOfValue, KeyType, ValueType>` and `HashFunction<HashFun, KeyType>` and `Compare` is [Relation](#) and `Compare.Domain` is `KeyType`

#### Model of

`ForwardContainer<T>`

#### Default Template Arguments

`HashFun` = [System.Collections.Hasher<KeyType>](#)

`Compare` = [System.EqualTo<KeyType>](#)

#### Base Class

[System.Collections.HashtableBase<ValueType>](#)

### 3.7.9.1 Type Definitions

Name	Type	Description
ConstIterator	<code>System.Collections.HashtableIterator&lt;- ValueType, const ValueType&amp;, const ValueType*&gt;</code>	A constant iterator type.
Iterator	<code>System.Collections.HashtableIterator&lt;- ValueType, ValueType&amp;, ValueType*&gt;</code>	An iterator type.

### 3.7.9.2 Member Functions

Member Function	Description
<code>Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;()</code>	Default constructor. Constructs an empty hash table.
<code>Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;(const System.Collections.- Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.Hashtable&lt;- KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;&amp;)</code>	Copy assignment.
<code>Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;(System.Collections.- Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.Hashtable&lt;- KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;&amp;&amp;)</code>	Move assignment.
<code>Begin()</code>	Returns an iterator to the beginning of the hash table, or <code>End()</code> if the hash table is empty.
<code>Begin() const</code>	Returns a constant iterator to the beginning of the hash table, or <code>CEnd() const</code> if the hash table is empty.
<code>CBegin() const</code>	Returns a constant iterator to the beginning of the hash table, or <code>CEnd() const</code> if the hash table is empty.
<code>CEnd() const</code>	Returns a constant iterator one past the end of the hash table.
<code>CFind(const KeyType&amp;) const</code>	Searches an element with the given key from the hash table and returns a constant iterator point- ing to it if found, or <code>CEnd() const</code> iterator other- wise.

<code>Clear()</code>	Makes the hash table empty.
<code>Count() const</code>	Returns the number of elements in the hash table.
<code>End()</code>	Returns an iterator pointing one past the end of the hash table.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the hash table.
<code>Find(const KeyType&amp;)</code>	Searches an element with the given key from the hash table and returns an iterator pointing to it if found, or <code>End()</code> iterator otherwise.
<code>Find(const KeyType&amp;) const</code>	Searches an element with the given key from the hash table and returns a constant iterator pointing to it if found, or <code>CEnd() const</code> iterator otherwise.
<code>GetBucket(int) const</code>	Returns a pointer to first bucket with the given index.
<code>GetBucketCount() const</code>	Returns the number of buckets in the hash table.
<code>GetBucketIndex(const ValueType&amp;) const</code>	Returns index of bucket with the given element.
<code>Insert(const ValueType&amp;)</code>	Inserts an element to the hash table and returns a pair consisting of an iterator to the inserted element and <code>true</code> if the element was not already in the hash table, or a pair consisting of an iterator pointing to an existing element and <code>false</code> otherwise.
<code>IsEmpty() const</code>	Returns true if the hash table is empty, false otherwise.
<code>Remove(System.Collections.HashtableIterator&lt;ValueType, ValueType&amp;, ValueType*&gt;)</code>	Removes an element pointed by the given iterator from the hash table.
<code>Remove(const KeyType&amp;)</code>	Removes an element with the given key from the hash table.
<code>SetMaxLoadFactor(double)</code>	Sets maximum load factor of the hash table.
<code>~Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;()</code>	Destructor.

**Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>() Member Function**

Default constructor. Constructs an empty hash table.

**Syntax**

```
public Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>();
```

`Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>(const System.Collections.Hashtable<ValueType, KeyOfValue, HashFun, Compare>&)` Member Function

Copy constructor.

### Syntax

```
public Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>(const System.Collections.Hashtable<ValueType, KeyOfValue, HashFun, Compare>& that);
```

### Parameters

Name	Type	Description
that	const System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>&	A hash table to copy from.

**operator=(const System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>&) Member Function**

Copy assignment.

#### Syntax

```
public void operator=(const System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>& that);
```

#### Parameters

Name	Type	Description
that	const System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>&	A hash table to assign.

`Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>(System.Collections.Hashtable<ValueType, KeyOfValue, HashFun, Compare>&&) Member Function`

Move constructor.

### Syntax

```
public Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>(System.Collections.Hashtable<ValueType, KeyOfValue, HashFun, Compare>&& __parameter0);
```

### Parameters

Name	Type	Description
<code>__parameter0</code>	<code>System.Collections.Hashtable&lt;KeyType, ValueType, KeyOfValue, HashFun, Compare&gt;&amp;&amp;</code>	A hash table to move from.

**operator=(System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Hashtable<KeyType, ValueType, KeyOfValue,  
HashFun, Compare>&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.Collections.Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>&&	A hash table to move from.

**Begin() Member Function**

Returns an iterator to the beginning of the hash table, or [End\(\)](#) is the hash table is empty.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*> Begin();
```

**Returns**

[System.Collections.HashtableIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns an iterator to the beginning of the hash table, or [End\(\)](#) is the hash table is empty.

**Begin() const Member Function**

Returns a constant iterator to the beginning of the hash table, or [CEnd\(\) const](#) if the hash table is empty.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
Begin() const;
```

**Returns**

[System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType\\*>](#)

Returns a constant iterator to the beginning of the hash table, or [CEnd\(\) const](#) if the hash table is empty.

**CBegin() const Member Function**

Returns a constant iterator to the beginning of the hash table, or [CEnd\(\) const](#) if the hash table is empty.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
CBegin() const;
```

**Returns**

```
System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
```

Returns a constant iterator to the beginning of the hash table, or [CEnd\(\) const](#) if the hash table is empty.

**CEnd() const Member Function**

Returns a constant iterator one past the end of the hash table.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
CEnd() const;
```

**Returns**

[System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType\\*>](#)

Returns a constant iterator one past the end of the hash table.

**CFind(const KeyType&) const Member Function**

Searches an element with the given key from the hash table and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
CFind(const KeyType& key) const;
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType\\*>](#)

Returns a constant iterator pointing to the found element, if the search was successful, or [CEnd\(\) const](#) iterator otherwise.

**Clear() Member Function**

Makes the hash table empty.

**Syntax**

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

**Count() const Member Function**

Returns the number of elements in the hash table.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the hash table.

**End() Member Function**

Returns an iterator pointing one past the end of the hash table.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*> End();
```

**Returns**

[System.Collections.HashtableIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns an iterator pointing one past the end of the hash table.

**End() const Member Function**

Returns a constant iterator pointing one past the end of the hash table.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
End() const;
```

**Returns**

[System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType\\*>](#)

Returns a constant iterator pointing one past the end of the hash table.

**Find(const KeyType&)** Member Function

Searches an element with the given key from the hash table and returns an iterator pointing to it if found, or [End\(\)](#) iterator otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*> Find(const  
KeyType& key);
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.HashtableIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns an iterator pointing to the found element, if the search was successful, or [End\(\)](#) iterator otherwise.

**Find(const KeyType&) const Member Function**

Searches an element with the given key from the hash table and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Syntax**

```
public System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType*>
Find(const KeyType& key) const;
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.HashtableIterator<ValueType, const ValueType&, const ValueType\\*>](#)

Returns a constant iterator pointing to the found element, if the search was successful, or [CEnd\(\) const](#) iterator otherwise.

**GetBucket(int) const Member Function**

Returns a pointer to first bucket with the given index.

**Syntax**

```
public System.Collections.Bucket<ValueType>* GetBucket(int index) const;
```

**Parameters**

Name	Type	Description
index	int	Index of the bucket.

**Returns**

[System.Collections.Bucket<ValueType>\\*](#)

Returns a pointer to first bucket with the given index.

**GetBucketCount() const Member Function**

Returns the number of buckets in the hash table.

**Syntax**

```
public int GetBucketCount() const;
```

**Returns**

int

Returns the number of buckets in the hash table.

**GetBucketIndex(const ValueType&) const Member Function**

Returns index of bucket with the given element.

**Syntax**

```
public int GetBucketIndex(const ValueType& value) const;
```

**Parameters**

Name	Type	Description
value	const ValueType&	An element.

**Returns**

int

Returns index of bucket with the given element.

**Insert(const ValueType&)** Member Function

Inserts an element to the hash table and returns a pair consisting of an iterator to the inserted element and **true** if the element was not already in the hash table, or a pair consisting of an iterator pointing to an existing element and **false** otherwise.

**Syntax**

```
public System.Pair<System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*>,  
bool> Insert(const ValueType& value);
```

**Parameters**

Name	Type	Description
value	const ValueType&	A value to insert.

**Returns**

[System.Pair<System.Collections.HashtableIterator<ValueType, ValueType&, ValueType\\*>, bool>](#)

Returns a pair consisting of an iterator to the inserted element and **true** if the element was not already in the hash table, or a pair consisting of an iterator pointing to an existing element and **false** otherwise.

**IsEmpty() const Member Function**

Returns true if the hash table is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the hash table is empty, false otherwise.

**Remove(System.Collections.HashtableIterator<ValueType, ValueType&, ValueType\*>)**  
**Member Function**

Removes an element pointed by the given iterator from the hash table.

**Syntax**

```
public void Remove(System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*>
pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.HashtableIterator<ValueType, ValueType&, ValueType*>	An iterator pointing to the ele- ment to remove.

**Remove(const KeyType&)** Member Function

Removes an element with the given key from the hash table.

**Syntax**

```
public void Remove(const KeyType& key);
```

**Parameters**

Name	Type	Description
key	const KeyType&	Key of element to remove.

**SetMaxLoadFactor(double) Member Function**

Sets maximum load factor of the hash table.

**Syntax**

```
public void SetMaxLoadFactor(double maxLoadFactor_);
```

**Parameters**

Name	Type	Description
maxLoadFactor_	double	New maximum load factor.

**Remarks**

By default maximum load factor is 0.8.

**`~Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>()`** Member Function

Destructor.

#### Syntax

```
public ~Hashtable<KeyType, ValueType, KeyOfValue, HashFun, Compare>();
```

### 3.7.10 HashtableBase<T> Class

Implementation detail.

#### Syntax

```
public abstract class HashtableBase<T>;
```

#### Constraint

T is [Semiregular](#)

### 3.7.10.1 Type Definitions

Name	Type	Description
------	------	-------------

### 3.7.10.2 Member Functions

Member Function	Description
-----------------	-------------

### 3.7.11 **HashtableIterator<T, R, P>** Class

A hash table iterator type.

#### Syntax

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

#### Model of

[ForwardIterator<T>](#)

### 3.7.11.1 Type Definitions

Name	Type	Description
PointerType	P	Type of pointer to hash table element.
ReferenceType	R	Type of reference to hash table element.
ValueType	T	Type of hash table element.

### 3.7.11.2 Member Functions

Member Function	Description
<code>HashtableIterator&lt;T, R, P&gt;()</code>	Default constructor.
<code>HashtableIterator&lt;T, R, P&gt;(const System.Collections.HashtableIterator&lt;T, R, P&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.HashtableIterator&lt;T, R, P&gt;&amp;)</code>	Copy assignment.
<code>HashtableIterator&lt;T, R, P&gt;&amp; operator=(System.Collections.HashtableIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.HashtableIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move assignment.
<code>GetBucket() const</code>	Returns a pointer to the bucket the iterator points to.
<code>HashtableIterator&lt;T, R, P&gt;(System.Collections.HashtableBase&lt;T&gt;*, System.Collections.Bucket&lt;T&gt;*)</code>	Constructor. Initializes the hash table iterator with the given pointer to hash table and pointer to bucket.
<code>operator++()</code>	Advances the iterator to point to next element in the hash table.
<code>operator-&gt;() const</code>	Returns a pointer to the hash table element the iterator points to.

**HashtableIterator<T, R, P>() Member Function**

Default constructor.

**Syntax**

```
public HashtableIterator<T, R, P>();
```

**HashtableIterator<T, R, P>(const System.Collections.HashtableIterator<T, R, P>&)**  
**Member Function**

Copy constructor.

**Syntax**

```
public HashtableIterator<T, R, P>(const System.Collections.HashtableIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.HashtableIterator<T, R, P>-&	Argument to copy.

**operator=(const System.Collections.HashtableIterator<T, R, P>&)** Member Function

Copy assignment.

#### Syntax

```
public void operator=(const System.Collections.HashtableIterator<T, R, P>& that);
```

#### Parameters

Name	Type	Description
that	const System.Collections.HashtableIterator<T, R, P> &	Argument to assign.

**HashtableIterator<T, R, P>(System.Collections.HashtableIterator<T, R, P>&&) Member Function**

Move constructor.

**Syntax**

```
public HashtableIterator<T, R, P>(System.Collections.HashtableIterator<T, R, P>&&  
that);
```

**Parameters**

Name	Type	Description
that	System.Collections.HashtableIterator<T, R, P>&&	Argument to move from.

**operator=(System.Collections.HashtableIterator<T, R, P>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.HashtableIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.HashtableIterator<T, R, P>&&	Argument to assign from.

**GetBucket() const Member Function**

Returns a pointer to the bucket the iterator points to.

**Syntax**

```
public System.Collections.Bucket<T>* GetBucket() const;
```

**Returns**

[System.Collections.Bucket<T>\\*](#)

Returns a pointer to the bucket the iterator points to.

**HashtableIterator<T, R, P>(System.Collections.HashtableBase<T>\*, System.Collections.Bucket<T>)**  
**Member Function**

Constructor. Initializes the hash table iterator with the given pointer to hash table and pointer to bucket.

**Syntax**

```
public HashtableIterator<T, R, P>(System.Collections.HashtableBase<T>* table_, System.Collections.Bucket<T> bucket_);
```

**Parameters**

Name	Type	Description
table_	System.Collections.HashtableBase<T>*	A pointer to hash table.
bucket_	System.Collections.Bucket<T>*	A pointer to bucket.

**operator++() Member Function**

Advances the iterator to point to next element in the hash table.

**Syntax**

```
public System.Collections.HashtableIterator<T, R, P>& operator++();
```

**Returns**

[System.Collections.HashtableIterator<T, R, P>&](#)

Returns the iterator.

**operator->() const Member Function**

Returns a pointer to the hash table element the iterator points to.

**Syntax**

```
public P operator->() const;
```

**Returns**

P

Returns a pointer to the hash table element the iterator points to.

### 3.7.11.3 Nonmember Functions

Function	Description
<code>operator==(const System.Collections.- HashtableIterator&lt;T, R, P&gt;&amp;, const System.- Collections.HashtableIterator&lt;T, R, P&gt;&amp;)</code>	Compares two hash table iterators for equality and returns true if they point to the same hash table node if the table is not empty, or both are <code>CEnd() const</code> iterators otherwise.

**operator==(const System.Collections.HashtableIterator<T, R, P>&, const System.Collections.HashtableIterator<T, R, P>&) Function**

Compares two hash table iterators for equality and returns true if they point to the same hash table node if the table is not empty, or both are [CEnd\(\)](#) [const](#) iterators otherwise.

**Syntax**

```
public bool operator==(const System.Collections.HashtableIterator<T, R, P>& left,  
const System.Collections.HashtableIterator<T, R, P>& right);
```

**Parameters**

Name	Type	Description
left	const System.Collections.HashtableIterator<T, R, P> &	The first iterator.
right	const System.Collections.HashtableIterator<T, R, P> &	The second iterator.

**Returns**

bool

Returns true if both point to same hash table node, or both are [CEnd\(\)](#) [const](#) iterators.

### 3.7.12 `LinkedList<T>` Class

A doubly linked list class.

#### Syntax

```
public class LinkedList<T>;
```

#### Constraint

T is [Regular](#)

#### Model of

[BidirectionalContainer<T>](#)

[BackInsertionSequence<T>](#)

[FrontInsertionSequence<T>](#)

[InsertionSequence<T>](#)

#### Base Class

[System.Collections.LinkedListBase](#)

### 3.7.12.1 Type Definitions

Name	Type	Description
ConstIterator	System.Collections.- LinkedListNodeIterator<T, const T&, const T*>	Type of constant iterator.
Iterator	System.Collections.- LinkedListNodeIterator<T, T&, T*>	Type of iterator.
ValueType	T	Type of element the linked list contains.

### 3.7.12.2 Member Functions

Member Function	Description
LinkedList<T>()	Default constructor. Constructs an empty linked list.
LinkedList<T>(const LinkedList<T>&)	Copy constructor.
operator=(const LinkedList<T>&)	Copy assignment.
LinkedList<T>(System.Collections.LinkedList<- T>&&)	Move constructor.
operator=(System.Collections.LinkedList<T>&- &)	Move assignment.
Add(const T&)	Adds an element to the end of the linked list.
Back() const	Returns a constant reference to the last element in the linked list.
Begin()	Returns an iterator pointing to the beginning of the linked list, or <a href="#">End()</a> if the linked list is empty.
Begin() const	Returns a constant iterator pointing to the beginning of the linked list, or <a href="#">CEnd() const</a> if the linked list is empty.
CBegin() const	Returns a constant iterator pointing to the beginning of the linked list, or <a href="#">CEnd() const</a> if the linked list is empty.
CEnd() const	Returns a constant iterator pointing one past the end of the linked list.
Clear()	Makes the linked list empty.
Count() const	Returns the number of elements in the linked list.

<code>End()</code>	Returns an iterator pointing one past the end of the linked list.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the linked list.
<code>Front() const</code>	Returns a constant reference to the first element in the linked list.
<code>Insert(System.Collections.- LinkedListNodeIterator&lt;T, T&amp;, T*&gt;, const T&amp;)</code>	Inserts an element before position pointed by the given iterator to the linked list.
<code>InsertFront(const T&amp;)</code>	Inserts an element to the head of the linked list.
<code>IsEmpty() const</code>	Returns true if the linked list is empty, false otherwise.
<code>Remove(System.Collections.- LinkedListNodeIterator&lt;T, T&amp;, T*&gt;)</code>	Removes an element pointed by the given iterator from the linked list.
<code>Remove(const T&amp;)</code>	Removes elements that are equal to the given value from the linked list.
<code>RemoveFirst()</code>	Removes the first element from the linked list.
<code>RemoveLast()</code>	Removes the last element from the linked list.
<code>~LinkedList&lt;T&gt;()</code>	Destructor.

**LinkedList<T>() Member Function**

Default constructor. Constructs an empty linked list.

**Syntax**

```
public LinkedList<T>();
```

**LinkedList<T>(const System.Collections.LinkedList<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LinkedList<T>(const System.Collections.LinkedList<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.LinkedList<T>&	A linked list to copy from.

**operator=(const System.Collections.LinkedList<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.LinkedList<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.LinkedList<T>&	A linked list to assign from.

**LinkedList<T>(System.Collections.LinkedList<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LinkedList<T>(System.Collections.LinkedList<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.LinkedList&lt;T&gt;&amp;&amp;</a>	A linked list to move from.

**operator=(System.Collections.LinkedList<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.LinkedList<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.LinkedList&lt;T&gt;&amp;&amp;</a>	A linked list to move from.

**Add(const T&)** Member Function

Adds an element to the end of the linked list.

**Syntax**

```
public void Add(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	An element to add.

**Back() const Member Function**

Returns a constant reference to the last element in the linked list.

**Syntax**

```
public const T& Back() const;
```

**Returns**

const T&

Returns a constant reference to the last element in the linked list.

**Begin() Member Function**

Returns an iterator pointing to the beginning of the linked list, or [End\(\)](#) if the linked list is empty.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, T&, T*> Begin();
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the beginning of the linked list, or [End\(\)](#) if the linked list is empty.

**Begin() const Member Function**

Returns a constant iterator pointing to the beginning of the linked list, or [CEnd\(\) const](#) if the linked list is empty.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, const T&, const T*> Begin() const;
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginning of the linked list, or [CEnd\(\) const](#) if the linked list is empty.

**CBegin() const Member Function**

Returns a constant iterator pointing to the beginning of the linked list, or [CEnd\(\) const](#) if the linked list is empty.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, const T&, const T*> CBegin()  
const;
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginning of the linked list, or [CEnd\(\) const](#) if the linked list is empty.

**CEnd() const Member Function**

Returns a constant iterator pointing one past the end of the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, const T&, const T*> CEnd() const;
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the linked list.

**Clear() Member Function**

Makes the linked list empty.

**Syntax**

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

**Count() const Member Function**

Returns the number of elements in the linked list.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the linked list.

**End() Member Function**

Returns an iterator pointing one past the end of the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, T&, T*> End();
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing one past the end of the linked list.

**End() const Member Function**

Returns a constant iterator pointing one past the end of the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, const T&, const T*> End() const;
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the linked list.

**Front() const Member Function**

Returns a constant reference to the first element in the linked list.

**Syntax**

```
public const T& Front() const;
```

**Returns**

const T&

Returns a constant reference to the first element in the linked list.

**Insert(System.Collections.LinkedListNodeIterator<T, T&, T\*>, const T&) Member Function**

Inserts an element before position pointed by the given iterator to the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, T&, T*> Insert(System.Collections.LinkedListNodeIterator<T, T&, T*> pos, const T& value);
```

**Parameters**

Name	Type	Description
pos	System.Collections.LinkedListNodeIterator<T, T&, T*>	An iterator pointing to a position before to insert.
value	const T&	An element to insert.

**Returns**

[System.Collections.LinkedListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**InsertFront(const T&)** Member Function

Inserts an element to the head of the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, T&, T*> InsertFront(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	An element to insert.

**Returns**

[System.Collections.LinkedListNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**IsEmpty() const Member Function**

Returns true if the linked list is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the linked list is empty, false otherwise.

**Remove(System.Collections.LinkedListNodeIterator<T, T&, T\*>) Member Function**

Removes an element pointed by the given iterator from the linked list.

**Syntax**

```
public void Remove(System.Collections.LinkedListNodeIterator<T, T&, T*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.LinkedListNodeIterator<T, T&, T*>	An iterator pointing to the element to remove.

**Remove(const T&)** Member Function

Removes elements that are equal to the given value from the linked list.

**Syntax**

```
public void Remove(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	A value to remove.

**RemoveFirst() Member Function**

Removes the first element from the linked list.

**Syntax**

```
public void RemoveFirst();
```

**RemoveLast() Member Function**

Removes the last element from the linked list.

**Syntax**

```
public void RemoveLast();
```

**~LinkedList<T>() Member Function**

Destructor.

**Syntax**

```
public ~LinkedList<T>();
```

**3.7.12.3 Nonmember Functions**

<b>Function</b>	<b>Description</b>		
<code>operator&lt;(const LinkedList&lt;T&gt;&amp;, const LinkedList&lt;T&gt;&amp;)</code>	<code>System.Collections.- System.Collections.- System.Collections.-</code>		
<code>operator==(const LinkedList&lt;T&gt;&amp;, const LinkedList&lt;T&gt;&amp;)</code>	<code>System.Collections.- System.Collections.- System.Collections.-</code>		

**operator<(const System.Collections.LinkedList<T>&, const System.Collections.LinkedList<T>&)**  
**Function**

Compares two linked listys for less than relationship and returns true if the first linked list comes lexicographically before the second linked list, false otherwise.

**Syntax**

```
public bool operator<(const System.Collections.LinkedList<T>& left, const System.Collections.LinkedList<T>& right);
```

**Constraint**

T is [TotallyOrdered](#)

**Parameters**

Name	Type	Description
left	const System.Collections.LinkedList<T>&	The first linked list.
right	const System.Collections.LinkedList<T>&	The second linked list.

**Returns**

bool

Returns true if the first linked list comes lexicographically before the second linked list, false otherwise.

**operator==(const System.Collections.LinkedList<T>&, const System.Collections.LinkedList<T>&)**  
**Function**

Compares two linked lists for equality and returns true if they contain same number of pairwise equal elements, false otherwise.

**Syntax**

```
public bool operator==(const System.Collections.LinkedList<T>& left, const System.Collections.LinkedList<T>& right);
```

**Constraint**

T is [Regular](#)

**Parameters**

Name	Type	Description
left	const System.Collections.LinkedList<T>&	The first linked list.
right	const System.Collections.LinkedList<T>&	The second linked list-.

**Returns**

bool

Returns true if both linked lists contain the same number of pairwise equal elements, false otherwise.

### 3.7.13 LinkedListBase Class

Implementation detail.

#### Syntax

```
public abstract class LinkedListBase;
```

#### 3.7.13.1 Member Functions

<u>Member Function</u>	<u>Description</u>
------------------------	--------------------

### 3.7.14 `LinkedListNode<T>` Class

A linked list node type.

#### Syntax

```
public class LinkedListNode<T>;
```

#### Base Class

[System.Collections.LinkedListNodeBase](#)

### 3.7.14.1 Type Definitions

Name	Type	Description
ValueType	T	Type of linked list element.

### 3.7.14.2 Member Functions

Member Function	Description	
LinkedListNode<T>()		Default constructor.
LinkedListNode<T>(const System.Collections.- LinkedListNode<T>&)		Copy constructor.
operator=(const System.Collections.- LinkedListNode<T>&)		Copy assignment.
LinkedListNode<T>(System.Collections.- LinkedListNode<T>&&)		Move constructor.
operator=(System.Collections.LinkedListNode<- T>&&)		Move assignment.
LinkedListNode<T>(const T&, System.- Collections.LinkedListNodeBase*, System.- Collections.LinkedListNodeBase*)		Constructor. Initializes the linked list node with the given value and pointers to previous and next linked list nodes.
Value()		Returns a reference to the contained linked list element.
Value() const		Returns a constant reference to the contained linked list element.

**LinkedListNode<T>() Member Function**

Default constructor.

**Syntax**

```
public LinkedListNode<T>();
```

**LinkedListNode<T>(const System.Collections.LinkedListNode<T>&) Member Function**

Copy constructor.

**Syntax**

```
public LinkedListNode<T>(const System.Collections.LinkedListNode<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.LinkedListNode<T>&	Argument to copy.

**operator=(const System.Collections.LinkedListNode<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.LinkedListNode<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.LinkedListNode<T>&	Argument to assign.

**LinkedListNode<T>(System.Collections.LinkedListNode<T>&&) Member Function**

Move constructor.

**Syntax**

```
public LinkedListNode<T>(System.Collections.LinkedListNode<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.LinkedListNode<T>&&	Argument to move from.

**operator=(System.Collections.LinkedListNode<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.LinkedListNode<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.LinkedListNode<T>&&	Argument to assign from.

**LinkedListNode<T>(const T&, System.Collections.LinkedListNodeBase\*, System.Collections.LinkedListNodeBase\*) Member Function**

Constructor. Initializes the linked list node with the given value and pointers to previous and next linked list nodes.

**Syntax**

```
public LinkedListNode<T>(const T& value_, System.Collections.LinkedListNodeBase* prev_, System.Collections.LinkedListNodeBase* next_);
```

**Parameters**

Name	Type	Description
value_	const T&	A linked list element.
prev_	System.Collections.LinkedListNodeBase*	A pointer to previous linked list node.
next_	System.Collections.LinkedListNodeBase*	A pointer to next linked list node.

**Value() Member Function**

Returns a reference to the contained linked list element.

**Syntax**

```
public T& Value();
```

**Returns**

T&

Returns a reference to the contained linked list element.

**Value() const Member Function**

Returns a constant reference to the contained linked list element.

**Syntax**

```
public const T& Value() const;
```

**Returns**

const T&

Returns a constant reference to the contained linked list element.

### 3.7.15 LinkedListNodeBase Class

Implementation detail.

#### Syntax

```
public class LinkedListNodeBase;
```

#### 3.7.15.1 Member Functions

Member Function	Description
<a href="#">LinkedListNodeBase()</a>	Default constructor.

**LinkedListNodeBase() Member Function**

Default constructor.

**Syntax**

```
public LinkedListNodeBase();
```

### 3.7.16 `LinkedListNodeIterator<T, R, P>` Class

A linked list iterator type.

#### Syntax

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

#### Model of

[BidirectionalIterator<T>](#)

### 3.7.16.1 Type Definitions

Name	Type	Description
PointerType	P	Type of pointer to linked list element.
ReferenceType	R	Type of reference to linked list element.
ValueType	T	Type of linked list element.

### 3.7.16.2 Member Functions

Member Function	Description
<code>LinkedListNodeIterator&lt;T, R, P&gt;()</code>	Default constructor.
<code>LinkedListNodeIterator&lt;T, R, P&gt;(const System.Collections.LinkedListIterator&lt;- T, R, P&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.- LinkedListIterator&lt;T, R, P&gt;&amp;)</code>	Copy assignment.
<code>LinkedListNodeIterator&lt;T, R, P&gt;(System.- Collections.LinkedListIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.- LinkedListIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move assignment.
<code>GetNode() const</code>	Returns a pointer to linked list node the iterator points to.
<code>LinkedListNodeIterator&lt;T, R, P&gt;(System.- Collections.LinkedListBase*, System.- Collections.LinkedListNode&lt;T&gt;*)</code>	Constructor. Initializes the linked list node iterator with the given pointer to linked list and pointer to linked list node.
<code>operator*() const</code>	Returns a reference to the linked list element the iterator points to.
<code>operator++()</code>	Advances the linked list node iterator to point to next element in the linked list.
<code>operator--()</code>	Backs up the linked list node iterator to point to previous element in the linked list.
<code>operator-&gt;() const</code>	Returns a pointer the linked list element the iterator points to.

**LinkedListNodeIterator<T, R, P>() Member Function**

Default constructor.

**Syntax**

```
public LinkedListNodeIterator<T, R, P>();
```

**LinkedListNodeIterator<T, R, P>(const System.Collections.LinkedListNodeIterator<T, R, P>&) Member Function**

Copy constructor.

**Syntax**

```
public LinkedListNodeIterator<T, R, P>(const System.Collections.LinkedListNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.LinkedListNodeIterator<T, R, P>&	Argument to copy.

**operator=(const System.Collections.LinkedListNodeIterator<T, R, P>&) Member Function**

Copy assignment.

#### Syntax

```
public void operator=(const System.Collections.LinkedListNodeIterator<T, R, P>& that);
```

#### Parameters

Name	Type	Description
that	const System.Collections.LinkedListNodeIterator<T, R, P>&	Argument to assign.

**LinkedListNodeIterator<T, R, P>(System.Collections.LinkedListNodeIterator<T, R, P>& z) Member Function**

Move constructor.

**Syntax**

```
public LinkedListNodeIterator<T, R, P>(System.Collections.LinkedListNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.LinkedListNodeIterator&lt;T, R, P&gt;</a> &&	Argument to move from.

**operator=(System.Collections.LinkedListNodeIterator<T, R, P>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.LinkedListNodeIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.LinkedListNodeIterator<T, R, P> &&	Argument to assign from.

**GetNode() const Member Function**

Returns a pointer to linked list node the iterator points to.

**Syntax**

```
public System.Collections.LinkedListNode<T>* GetNode() const;
```

**Returns**

[System.Collections.LinkedListNode<T>\\*](#)

Returns a pointer to linked list node the iterator points to.

**LinkedListNodeIterator<T, R, P>(System.Collections.LinkedListBase\*, System.Collections.LinkedListNode<T>\*) Member Function**

Constructor. Initializes the linked list node iterator with the given pointer to linked list and pointer to linked list node.

**Syntax**

```
public LinkedListNodeIterator<T, R, P>(System.Collections.LinkedListBase* list_,  
System.Collections.LinkedListNode<T>* node_);
```

**Parameters**

Name	Type	Description
list_	<a href="#">System.Collections.LinkedListBase*</a>	A pointer to linked list.
node_	<a href="#">System.Collections.LinkedListNode&lt;T&gt;*</a>	A pointer to linked list node.

**operator\*() const Member Function**

Returns a reference to the linked list element the iterator points to.

**Syntax**

```
public R operator*() const;
```

**Returns**

R

Returns a reference to the linked list element the iterator points to.

**operator++() Member Function**

Advances the linked list node iterator to point to next element in the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, R, P>& operator++();
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, R, P>&](#)

Returns the iterator.

**operator–() Member Function**

Backs up the linked list node iterator to point to previous element in the linked list.

**Syntax**

```
public System.Collections.LinkedListNodeIterator<T, R, P>& operator--();
```

**Returns**

[System.Collections.LinkedListNodeIterator<T, R, P>&](#)

Returns the iterator.

**operator->() const Member Function**

Returns a pointer the linked list element the iterator points to.

**Syntax**

```
public P operator->() const;
```

**Returns**

P

Returns a pointer the linked list element the iterator points to.

### 3.7.16.3 Nonmember Functions

Function	Description
<code>operator==(const System.Collections.LinkedListIterator&lt;T, R, P&gt;&amp;, const System.Collections.LinkedListIterator&lt;T, R, P&gt;&amp;)</code>	Compares two linked list node iterators and returns true if they point to same linked list node, or both are <code>CEnd() const</code> iterators, false otherwise.

**operator==(const System.Collections.LinkedListNodeIterator<T, R, P>&, const System.Collections.LinkedListNodeIterator<T, R, P>&) Function**

Compares two linked list node iterators and returns true if they point to same linked list node, or both are [CEnd\(\) const](#) iterators, false otherwise.

**Syntax**

```
public bool operator==(const System.Collections.LinkedListNodeIterator<T, R, P>& left, const System.Collections.LinkedListNodeIterator<T, R, P>& right);
```

**Parameters**

Name	Type	Description
left	<a href="#">const System.Collections.LinkedListNodeIterator&lt;T, R, P&gt;&amp;</a>	The first linked list node iterator.
right	<a href="#">const System.Collections.LinkedListNodeIterator&lt;T, R, P&gt;&amp;</a>	The second linked list node iterator.

**Returns**

bool

Returns true if both iterators point to same linked list node, or both are [CEnd\(\) const](#) iterators, false otherwise.

### 3.7.17 List<T> Class

A container of elements in which the contained elements are in consecutive locations in memory.

#### Syntax

```
public class List<T>;
```

#### Constraint

T is [Semiregular](#)

#### Model of

[BackInsertionSequence<T>](#)

[InsertionSequence<T>](#)

[FrontInsertionSequence<T>](#)

[RandomAccessContainer<T>](#)

#### 3.7.17.1 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());
}
```

### 3.7.17.2 Type Definitions

Name	Type	Description
ConstIterator	<code>System.RandomAccessIter&lt;T, const T&amp;, const T*&gt;</code>	A constant iterator type.
Iterator	<code>System.RandomAccessIter&lt;T, T&amp;, T*&gt;</code>	An iterator type.
ValueType	<code>T</code>	The type of the contained element.

### 3.7.17.3 Member Functions

Member Function	Description
<code>List&lt;T&gt;()</code>	Constructor. Constructs an empty list.
<code>List&lt;T&gt;(const System.Collections.List&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.List&lt;T&gt;&amp;)</code>	Copy assignment.
<code>List&lt;T&gt;(System.Collections.List&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.List&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>Add(T&amp;&amp;)</code>	Moves an element to the end of the list.
<code>Add(const T&amp;)</code>	Copies an element to the end of the list.
<code>Back()</code>	Returns a reference to the last element in the list.
<code>Back() const</code>	Returns a constant reference to the last element in the list.
<code>Begin()</code>	Returns an iterator pointing to the first element of the list, or <code>End()</code> if the list is empty.
<code>Begin() const</code>	Returns a constant iterator pointing to the first element of the list, or <code>CEnd() const</code> if the list is empty.
<code>CBegin() const</code>	Returns a constant iterator pointing to the first element of the list, or <code>CEnd() const</code> if the list is empty.
<code>CEnd() const</code>	Returns a constant iterator pointing one past the end of the list.
<code>Capacity() const</code>	Returns the number of elements that the list can contain without a memory allocation.

<code>Clear()</code>	Makes the list empty.
<code>Count() const</code>	Returns the number of elements in the list.
<code>End()</code>	Returns an iterator pointing one past the end of the list.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the list.
<code>Front()</code>	Returns a reference to the first element in the list.
<code>Front() const</code>	Returns a constant reference to the first element in the list.
<code>Insert(System.RandomAccessIter&lt;T, T&amp;, T*&gt;, T&amp;&amp;)</code>	Moves an element before the given position in the list.
<code>Insert(System.RandomAccessIter&lt;T, T&amp;, T*&gt;, const T&amp;)</code>	Copies an element before the given position in the list.
<code>InsertFront(T&amp;&amp;)</code>	Moves an element to the head of the list.
<code>InsertFront(const T&amp;)</code>	Copies an element to the head of the list.
<code>IsEmpty() const</code>	Returns true if the list is empty, false otherwise.
<code>List&lt;T&gt;(int, const T&amp;)</code>	Constructs a list consisting of given number of copies of given value.
<code>Remove(System.RandomAccessIter&lt;T, T*&gt;, T&amp;)</code>	Removes an element pointed by the given iterator from the list and returns the removed element.
<code>RemoveFirst()</code>	Removes the first element from the list.
<code>RemoveLast()</code>	Removes the last element from the list.
<code>Reserve(int)</code>	Makes the capacity of the list at least the given number of elements.
<code>Resize(int)</code>	Grows or shrinks the list so that it contains given number of elements.
<code>operator[](int)</code>	Returns a reference to the element with the given index.

<code>operator[](int) const</code>	Returns a constant reference to the element with the given index.
<code>~List&lt;T&gt;()</code>	Destructor.

**List<T>() Member Function**

Constructor. Constructs an empty list.

**Syntax**

```
public List<T>();
```

**Implementation**

[list.cm, page 1](#)

**List<T>(const System.Collections.List<T>&) Member Function**

Copy constructor.

**Syntax**

```
public List<T>(const System.Collections.List<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.List<T>&	A list to copy from.

**operator=(const System.Collections.List<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.List<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.List<T>&	A list to assign from.

**List<T>(System.Collections.List<T>&&) Member Function**

Move constructor.

**Syntax**

```
public List<T>(System.Collections.List<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.List<T>&&	A list to move from.

**operator=(System.Collections.List<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.List<T>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.List&lt;T&gt;&amp;&amp;</a>	A list to move from.

**Add(T&&)** Member Function

Moves an element to the end of the list.

**Syntax**

```
public void Add(T&& item);
```

**Parameters**

Name	Type	Description
item	T&&	An element to move.

**Add(const T&)** Member Function

Copies an element to the end of the list.

**Syntax**

```
public void Add(const T& item);
```

**Parameters**

Name	Type	Description
item	const T&	An element to add.

**Back() Member Function**

Returns a reference to the last element in the list.

**Syntax**

```
public T& Back();
```

**Returns**

T&

Returns a reference to the last element in the list.

**Implementation**

[list.cm, page 6](#)

**Back() const Member Function**

Returns a constant reference to the last element in the list.

**Syntax**

```
public const T& Back() const;
```

**Returns**

const T&

Returns a constant reference to the last element in the list.

**Implementation**

[list.cm, page 6](#)

**Begin() Member Function**

Returns an iterator pointing to the first element of the list, or [End\(\)](#) if the list is empty.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> Begin();
```

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing to the first element of the list, or [End\(\)](#) if the list is empty.

**Implementation**

[list.cm, page 5](#)

**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 System.RandomAccessIter<T, const T&, const T*> Begin() const;
```

**Returns**

[System.RandomAccessIter<T, const T&, const T\\*>](#)

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](#)

**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 System.RandomAccessIter<T, const T&, const T*> CBegin() const;
```

**Returns**

[System.RandomAccessIter<T, const T&, const T\\*>](#)

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](#)

**CEnd() const Member Function**

Returns a constant iterator pointing one past the end of the list.

**Syntax**

```
public System.RandomAccessIter<T, const T&, const T*> CEnd() const;
```

**Returns**

[System.RandomAccessIter<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the list.

**Implementation**

[list.cm, page 5](#)

**Capacity() const Member Function**

Returns the number of elements that the list can contain without a memory allocation.

**Syntax**

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

**Returns**

int

Returns the number of elements that the list can contain without a memory allocation.

**Implementation**

[list.cm, page 2](#)

**Clear() Member Function**

Makes the list empty.

**Syntax**

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

**Implementation**

[list.cm, page 3](#)

**Count() const Member Function**

Returns the number of elements in the list.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the list.

**Implementation**

[list.cm, page 3](#)

**End() Member Function**

Returns an iterator pointing one past the end of the list.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> End();
```

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing one past the end of the list.

**Implementation**

[list.cm, page 5](#)

**End() const Member Function**

Returns a constant iterator pointing one past the end of the list.

**Syntax**

```
public System.RandomAccessIter<T, const T&, const T*> End() const;
```

**Returns**

[System.RandomAccessIter<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the list.

**Implementation**

[list.cm, page 5](#)

**Front() Member Function**

Returns a reference to the first element in the list.

**Syntax**

```
public T& Front();
```

**Returns**

T&

Returns a reference to the first element int the list.

**Implementation**

[list.cm, page 5](#)

**Front() const Member Function**

Returns a constant reference to the first element in the list.

**Syntax**

```
public const T& Front() const;
```

**Returns**

const T&

Returns a constant reference to the first element in the list.

**Implementation**

[list.cm, page 5](#)

**Insert(System.RandomAccessIter<T, T&, T\*>, T&&) Member Function**

Moves an element before the given position in the list.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> Insert(System.RandomAccessIter<T, T&, T*>
pos, T&& item);
```

**Parameters**

Name	Type	Description
pos	<a href="#">System.RandomAccessIter&lt;T, T&amp;, T*&gt;</a>	An iterator pointing to the position before which to insert.
item	T&&	An element to move.

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**Insert(System.RandomAccessIter<T, T&, T\*>, const T&) Member Function**

Copies an element before the given position in the list.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> Insert(System.RandomAccessIter<T, T&, T*>
pos, const T& item);
```

**Parameters**

Name	Type	Description
pos	<a href="#">System.RandomAccessIter&lt;T, T&amp;, T*&gt;</a>	An iterator pointing to the position before which to insert.
item	const T&	An element to insert.

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**InsertFront(T&&) Member Function**

Moves an element to the head of the list.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> InsertFront(T&& item);
```

**Parameters**

Name	Type	Description
item	T&&	An element to move.

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**InsertFront(const T&)** Member Function

Copies an element to the head of the list.

**Syntax**

```
public System.RandomAccessIter<T, T&, T*> InsertFront(const T& item);
```

**Parameters**

Name	Type	Description
item	const T&	An element to insert.

**Returns**

[System.RandomAccessIter<T, T&, T\\*>](#)

Returns an iterator pointing to the inserted element.

**IsEmpty() const Member Function**

Returns true if the list is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the list is empty, false otherwise.

**Implementation**

[list.cm, page 3](#)

**List<T>(int, const T&)** Member Function

Constructs a list consisting of given number of copies of given value.

**Syntax**

```
public List<T>(int n, const T& value);
```

**Parameters**

Name	Type	Description
n	int	Number of elements to construct.
value	const T&	Element to copy.

**Remove(System.RandomAccessIter<T, T&, T\*>) Member Function**

Removes an element pointed by the given iterator from the list and returns the removed element.

**Syntax**

```
public T Remove(System.RandomAccessIter<T, T&, T*> pos);
```

**Parameters**

Name	Type	Description
pos	System.RandomAccessIter<T, T&, T*>	An iterator pointing to the element to remove.

**Returns**

T

Returns the removed element.

**Implementation**

[list.cm, page 4](#)

**RemoveFirst() Member Function**

Removes the first element from the list.

**Syntax**

```
public T RemoveFirst();
```

**Returns**

T

Returns the removed element.

**Implementation**

[list.cm, page 4](#)

**RemoveLast() Member Function**

Removes the last element from the list.

**Syntax**

```
public T RemoveLast();
```

**Returns**

T

Returns the removed element.,,

**Implementation**

[list.cm, page 4](#)

**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](#)

**Resize(int) Member Function**

Grows or shrinks the list so that it contains given number of elements.

**Syntax**

```
public void Resize(int newCount);
```

**Parameters**

Name	Type	Description
newCount	int	New number of elements.

**operator[](int) Member Function**

Returns a reference to the element with the given index.

**Syntax**

```
public T& operator[](int index);
```

**Parameters**

Name	Type	Description
index	int	An index.

**Returns**

T&

Returns a reference to the element with the given index.

**Implementation**

[list.cm, page 5](#)

**operator[](int) const Member Function**

Returns a constant reference to the element with the given index.

**Syntax**

```
public const T& operator[](int index) const;
```

**Parameters**

Name	Type	Description
index	int	An index.

**Returns**

const T&

Returns a constant reference to the element with the given index.

**Implementation**

[list.cm, page 5](#)

**~List<T>() Member Function**

Destructor.

**Syntax**

```
public ~List<T>();
```

**3.7.17.4 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.Collections.List&lt;T&gt;&amp;, const System.Collections.List&lt;T&gt;&amp;)</code>	Compares two lists for less than relationship and returns true if the first list comes lexicographically before the second list, false otherwise.
<code>operator==(const System.Collections.List&lt;T&gt;&amp;, const System.Collections.List&lt;T&gt;&amp;)</code>	Compares two lists for equality and returns true if both lists contain the same number of pairwise equal elements, false otherwise.

**operator<(const System.Collections.List<T>&, const System.Collections.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 bool operator<(const System.Collections.List<T>& left, const System.Collections.List<T>& right);
```

**Constraint**

T is [TotallyOrdered](#)

**Parameters**

Name	Type	Description
left	const System.Collections.List<T>&	The first list.
right	const System.Collections.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](#)

**operator==(const System.Collections.List<T>&, const System.Collections.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 bool operator==(const System.Collections.List<T>& left, const System.Collections.List<T>& right);
```

**Constraint**

T is [Regular](#)

**Parameters**

Name	Type	Description
left	const System.Collections.List<T>&	The first list.
right	const System.Collections.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

### 3.7.18 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

Key is [Semiregular](#) and Value is [Semiregular](#) and KeyCompare is [Relation](#) and KeyCompare.Domain is Key

#### Model of

[BidirectionalContainer<T>](#)

#### Default Template Arguments

KeyCompare = [System.Less<Key>](#)

#### 3.7.18.1 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();
}
```

### 3.7.18.2 Type Definitions

Name	Type	Description
Compare	KeyCompare	The type of a relation used to compare keys.
ConstIterator	System.Collections.- RedBlackTreeNodeIterator<System.Pair<- Key, Value>, const System.Pair<Key, Value>&, const System.Pair<Key, Value>- *>	A constant iterator type.
Iterator	System.Collections.- RedBlackTreeNodeIterator<System.Pair<- Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*>	An iterator type.
KeyType	Key	The type of key.
MappedType	Value	The type associated with the key.
ValueType	System.Pair<Key, Value>	A pair composed of key type and mapped type.

### 3.7.18.3 Member Functions

Member Function	Description
Map<Key, Value, KeyCompare>()	Default constructor. Constructs an empty map.
Map<Key, Value, KeyCompare>(const System.- Collections.Map<Key, Value, KeyCompare>&)	Copy constructor.
operator=(const System.Collections.Map<Key, Value, KeyCompare>&)	Copy assignment.
Map<Key, Value, KeyCompare>(System.- Collections.Map<Key, Value, KeyCompare>&&)	Move constructor.
operator=(System.Collections.Map<Key, Value, KeyCompare>&&)	Move assignment.
Begin()	Returns an iterator pointing to the beginning of the map, or <code>End()</code> if the map is empty.
Begin() const	Returns a constant iterator pointing to the beginning of the map.
CBegin() const	Returns a constant iterator pointing to the beginning of the map, or <code>CEnd() const</code> if the map is empty.
CEnd() const	Returns a constant iterator pointing to one past the end of the map.

<code>CFind(const Key&amp;) const</code>	Searches the given key from the map and returns a constant iterator pointing to the found element, or <code>CEnd() const</code> iterator if the key is not found.
<code>Clear()</code>	Makes the map empty.
<code>Count() const</code>	Returns the number of key-value pairs in the map.
<code>End()</code>	Returns an iterator pointing to one past the end of the map.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the map.
<code>Find(const Key&amp;)</code>	Searches the given key in the map and returns an iterator pointing to it, if found, or an <code>System.Collections.Map.End</code> iterator otherwise.
<code>Find(const Key&amp;) const</code>	Searches the given key in the map and returns a constant iterator pointing to the found element, or <code>CEnd() const</code> iterator if the key is not found.
<code>Insert(const System.Pair&lt;Key, Value&gt;&amp;)</code>	Inserts an element to the map if the key of the element is not already found in the map.
<code>IsEmpty() const</code>	Returns true if the map is empty, false otherwise.
<code>Remove(System.Collections.-RedBlackTreeNodeIterator&lt;System.Pair&lt;Key, Value&gt;, System.Pair&lt;Key, Value&gt;&amp;, System.Pair&lt;Key, Value&gt;*&gt;)</code>	Removes an element pointed by the given iterator from the map.
<code>Remove(const Key&amp;)</code>	Removes a key-value pair associated with the given key from the map.
<code>operator[](const Key&amp;)</code>	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.

**Map<Key, Value, KeyCompare>() Member Function**

Default constructor. Constructs an empty map.

**Syntax**

```
public Map<Key, Value, KeyCompare>();
```

**Implementation**

[map.cm, page 1](#)

`Map<Key, Value, KeyCompare>(const System.Collections.Map<Key, Value, KeyCompare>&)`  
**Member Function**

Copy constructor.

### Syntax

```
public Map<Key, Value, KeyCompare>(const System.Collections.Map<Key, Value, KeyCompare>&
that);
```

### Parameters

Name	Type	Description
that	const System.Collections.Map<Key, Value, KeyCompare>&	Argument to copy.

**operator=(const System.Collections.Map<Key, Value, KeyCompare>&) Member Function**

Copy assignment.

#### Syntax

```
public void operator=(const System.Collections.Map<Key, Value, KeyCompare>& that);
```

#### Parameters

Name	Type	Description
that	const System.Collections.Map<Key, Value, KeyCompare>&	Argument to assign.

**Map<Key, Value, KeyCompare>(System.Collections.Map<Key, Value, KeyCompare>&&) Member Function**

Move constructor.

**Syntax**

```
public Map<Key, Value, KeyCompare>(System.Collections.Map<Key, Value, KeyCompare>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Map<Key, Value, KeyCompare>&	Argument to move from.

**operator=(System.Collections.Map<Key, Value, KeyCompare>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Map<Key, Value, KeyCompare>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Map<Key, Value, KeyCompare>&	Argument to assign from. &

**Begin() Member Function**

Returns an iterator pointing to the beginning of the map, or [End\(\)](#) if the map is empty.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*> Begin();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>\\*>](#)

Returns an iterator pointing to the beginning of the map, or [End\(\)](#) if the map is empty.

**Implementation**

[map.cm, page 1](#)

**Begin() const Member Function**

Returns a constant iterator pointing to the beginning of the map.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const  
System.Pair<Key, Value>&, const System.Pair<Key, Value>*> Begin() const;
```

**Returns**

`System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<-  
Key, Value>&, const System.Pair<Key, Value>*>`

Returns a constant iterator pointing to the beginning of the map, or [CEnd\(\) const](#) if the map is empty.

**CBegin() const Member Function**

Returns a constant iterator pointing to the beginning of the map, or [CEnd\(\) const](#) if the map is empty.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const  
System.Pair<Key, Value>&, const System.Pair<Key, Value>*> CBegin() const;
```

**Returns**

`System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<-  
Key, Value>&, const System.Pair<Key, Value>*>`

Returns a constant iterator pointing to the beginning of the map, or [CEnd\(\) const](#) if the map is empty.

**Implementation**

[map.cm, page 2](#)

**CEnd() const Member Function**

Returns a constant iterator pointing to one past the end of the map.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const  
System.Pair<Key, Value>&, const System.Pair<Key, Value>*> CEnd() const;
```

**Returns**

`System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<-  
Key, Value>&, const System.Pair<Key, Value>*>`

Returns a constant iterator pointing to one past the end of the map.

**Implementation**

[map.cm, page 2](#)

**CFind(const Key&)** const Member Function

Searches the given key from the map and returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) iterator if the key is not found.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<Key, Value>&, const System.Pair<Key, Value>*> CFind(const Key& key) const;
```

**Parameters**

Name	Type	Description
key	const Key&	A key to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<Key, Value>&, const System.Pair<Key, Value>\\*>](#)

Returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) iterator if the key is not found.

**Clear() Member Function**

Makes the map empty.

**Syntax**

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

**Implementation**

[map.cm, page 2](#)

**Count() const Member Function**

Returns the number of key-value pairs in the map.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of key-value pairs in the map.

**Implementation**

[map.cm, page 2](#)

**End() Member Function**

Returns an iterator pointing to one past the end of the map.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*> End();
```

**Returns**

`System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*>`

Returns an iterator pointing to one past the end of the map.

**Implementation**

[map.cm, page 2](#)

**End() const Member Function**

Returns a constant iterator pointing one past the end of the map.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const  
System.Pair<Key, Value>&, const System.Pair<Key, Value>*> End() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<-  
Key, Value>&, const System.Pair<Key, Value>\\*>](#)

Returns a constant iterator pointing one past the end of the map.

**Find(const Key&)** Member Function

Searches the given key in the map and returns an iterator pointing to it, if found, or an System.Collections.Map.End iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*> Find(const Key& key);
```

**Parameters**

Name	Type	Description
key	const Key&	A key to search.

**Returns**

System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>\*>

Returns an iterator pointing to the found key-value pair, if the search was successful, or System.Collections.Map.End iterator otherwise.

**Implementation**

[map.cm, page 2](#)

**Find(const Key&) const Member Function**

Searches the given key in the map and returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) iterator if the key is not found.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<Key, Value>&, const System.Pair<Key, Value>*> Find(const Key& key) const;
```

**Parameters**

Name	Type	Description
key	const Key&	A key to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, const System.Pair<Key, Value>&, const System.Pair<Key, Value>\\*>](#)

Returns a constant iterator pointing to the found element, or [CEnd\(\) const](#) iterator if the key is not found.

**Insert(const System.Pair<Key, Value>&) Member Function**

Inserts an element to the map if the key of the element is not already found in the map.

**Syntax**

```
public System.Pair<System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*>, bool> Insert(const System.Pair<Key, Value>& value);
```

**Parameters**

Name	Type	Description
value	const System.Pair<Key, Value>&	A key-value pair to insert.

**Returns**

System.Pair<System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>\*>, bool>

If the key of the element is not already in the map returns a pair consisting of an iterator pointing to the inserted element and **true**, or a pair consisting of an iterator pointing to an existing element and **false** otherwise.

**IsEmpty() const Member Function**

Returns true if the map is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the map is empty, false otherwise.

**Implementation**

[map.cm, page 2](#)

**Remove(System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>\*>) Member Function**

Removes an element pointed by the given iterator from the map.

**Syntax**

```
public void Remove(System.Collections.RedBlackTreeNodeIterator<System.Pair<Key, Value>, System.Pair<Key, Value>&, System.Pair<Key, Value>*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.RedBlackTreeNodeIterator<- System.Pair<Key, Value>, System.Pair<Key, Value>- &, System.Pair<Key, Value>*>	An iterator pointing to the ele- ment to remove.

**Remove(const Key&)** Member Function

Removes a key-value pair associated with the given key from the map.

**Syntax**

```
public bool Remove(const Key& key);
```

**Parameters**

Name	Type	Description
key	const Key&	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](#)

**operator[](const Key&) 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 Value& operator[](const Key& key);
```

**Parameters**

Name	Type	Description
key	const Key&	A key.

**Returns**

Value&

A value associated with the key.

**Implementation**

[map.cm, page 2](#)

**3.7.18.4 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.Collections.Map&lt;Key, Value, KeyCompare&gt;&amp;, const System.Collections.Map&lt;Key, Value, KeyCompare&gt;&amp;)</code>	Compares two maps for less than relationship and returns true if the first map comes lexicographically before the second map, false otherwise.
<code>operator==(const System.Collections.Map&lt;Key, Value, KeyCompare&gt;&amp;, const System.Collections.Map&lt;Key, Value, KeyCompare&gt;&amp;)</code>	Compares two maps for equality and returns true if both maps contain the same number of pairwise equal elements, false otherwise.

**operator<(const System.Collections.Map<Key, Value, KeyCompare>&, const System.Collections.Map<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 bool operator<(const System.Collections.Map<Key, Value, KeyCompare>& left,
const System.Collections.Map<Key, Value, KeyCompare>& right);
```

**Constraint**

Key is [Semiregular](#) and Value is [TotallyOrdered](#) and KeyCompare is [Relation](#) and KeyCompare.Domain is Key

**Parameters**

Name	Type	Description
left	const System.Collections.Map<Key, Value, KeyCompare>&	The first map.
right	const System.Collections.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

### **operator==(const System.Collections.Map<Key, Value, KeyCompare>&, const System.Collections.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 bool operator==(const System.Collections.Map<Key, Value, KeyCompare>& left,
const System.Collections.Map<Key, Value, KeyCompare>& right);
```

#### Constraint

Key is [Regular](#) and Value is [Regular](#) and KeyCompare is [Relation](#) and KeyCompare.Domain is [Key](#)

#### Parameters

Name	Type	Description
left	const System.Collections.Map<Key, Value, KeyCompare>&	The first map to compare.
right	const System.Collections.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](#)

### 3.7.19 Queue<T> Class

A first-in first-out data structure.

#### Syntax

```
public class Queue<T>;
```

#### Constraint

T is [Semiregular](#)

#### 3.7.19.1 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();
}
```

### 3.7.19.2 Type Definitions

Name	Type	Description
ValueType	T	The type of the queue element.

### 3.7.19.3 Member Functions

Member Function	Description
<code>Queue&lt;T&gt;()</code>	Default constructor. Constructs an empty queue.
<code>Queue&lt;T&gt;(const System.Collections.Queue&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.Queue&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Queue&lt;T&gt;(System.Collections.Queue&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.Queue&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>Clear()</code>	Makes the queue empty.
<code>Count() const</code>	Returns the number of elements in the queue.
<code>Front() const</code>	Returns a constant reference to the first element in the queue.
<code>Get()</code>	Removes the first element from the queue and returns it.
<code>IsEmpty() const</code>	Returns true if the queue is empty, false otherwise.
<code>Put(T&amp;&amp;)</code>	Moves an element to the back of the queue.
<code>Put(const T&amp;)</code>	Puts an element to the back of the queue.

**Queue<T>() Member Function**

Default constructor. Constructs an empty queue.

**Syntax**

```
public Queue<T>();
```

**Implementation**

[queue.cm, page 1](#)

**Queue<T>(const System.Collections.Queue<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Queue<T>(const System.Collections.Queue<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Queue<T>&	Argument to copy.

**operator=(const System.Collections.Queue<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.Queue<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Queue<T>&	Argument to assign.

**Queue<T>(System.Collections.Queue<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Queue<T>(System.Collections.Queue<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Queue<T>&&	Argument to move from.

**Implementation**

[queue.cm, page 1](#)

**operator=(System.Collections.Queue<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Queue<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Queue<T>&&	Argument to assign from.

**Implementation**

[queue.cm, page 1](#)

**Clear() Member Function**

Makes the queue empty.

**Syntax**

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

**Implementation**

[queue.cs, page 2](#)

**Count() const Member Function**

Returns the number of elements in the queue.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the queue.

**Implementation**

[queue.cm, page 1](#)

**Front() const Member Function**

Returns a constant reference to the first element in the queue.

**Syntax**

```
public const T& Front() const;
```

**Returns**

const T&

Returns a constant reference to the first element in the queue.

**Implementation**

[queue.cm, page 2](#)

**Get() Member Function**

Removes the first element from the queue and returns it.

**Syntax**

```
public T Get();
```

**Returns**

T

Returns the removed first element of the queue.

**Implementation**

[queue.cm, page 1](#)

**IsEmpty() const Member Function**

Returns true if the queue is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the queue is empty, false otherwise.

**Implementation**

[queue.cm, page 1](#)

**Put(T&&) Member Function**

Moves an element to the back of the queue.

**Syntax**

```
public void Put(T&& item);
```

**Parameters**

Name	Type	Description
item	T&&	An element to insert.

**Implementation**

[queue.cm, page 1](#)

**Put(const T&)** Member Function

Puts an element to the back of the queue.

**Syntax**

```
public void Put(const T& item);
```

**Parameters**

Name	Type	Description
item	const T&	An element to put.

**Implementation**

[queue.cm, page 1](#)

### 3.7.20 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

KeyType is [Semiregular](#) and ValueType is [Semiregular](#) and [KeySelectionFunction<KeyOfValue, KeyType, ValueType>](#) and Compare is [Relation](#) and Compare.Domain is KeyType

### 3.7.20.1 Type Definitions

Name	Type	Description
ConstIterator	System.Collections.- RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType*>	A constant iterator type.
Iterator	System.Collections.- RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>	An iterator type.

### 3.7.20.2 Member Functions

Member Function	Description
RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>()	Default constructor. Constructs an empty red-black tree.
RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(const System.Collections.- RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&)	Copy constructor.
operator=(const System.Collections.- RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&)	Copy assignment.
RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(&System.Collections.- RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&)	Move constructor.
operator=(System.Collections.RedBlackTree<- KeyType, ValueType, KeyOfValue, Compare>- &&)	Move assignment.
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.
CFind(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 System.Collections.-RedBlackTree.CEnd iterator otherwise.

<code>Clear()</code>	Makes the red-black tree empty.
<code>Count() const</code>	Returns the number of elements in the red-black tree.
<code>End()</code>	Returns a bidirectional iterator pointing to one past the end of the red-black tree.
<code>End() const</code>	Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.
<code>Find(const KeyType&amp;)</code>	Finds an element with the given key in the red-black tree and returns an iterator pointing to it if found, or System.Collections.RedBlackTree.End iterator otherwise.
<code>Find(const KeyType&amp;) const</code>	Finds an element with the given key in the red-black tree and returns a constant iterator pointing to it if found, or System.Collections.-RedBlackTree.CEnd.const iterator otherwise.
<code>IsEmpty() const</code>	Returns true if the red-black tree is empty, false otherwise.
<code>Remove(System.Collections.-RedBlackTreeNodeIterator&lt;ValueType, ValueType&amp;, ValueType*&gt;)</code>	Removes an element pointed by the given iterator from the red-black tree.
<code>Remove(const KeyType&amp;)</code>	Removes an element with the given key from the red-black tree. If an element with the given is not found, does nothing.
<code>~RedBlackTree&lt;KeyType, ValueType, KeyOf-Value, Compare&gt;()</code>	Destructor.

**RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>() Member Function**

Default constructor. Constructs an empty red-black tree.

**Syntax**

```
public RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>();
```

**Implementation**

[rbtree.cm, page 12](#)

`RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(const System.Collections.RedBlackTree<ValueType, KeyOfValue, Compare>&)` Member Function

Copy constructor.

### Syntax

```
public RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(const System.Collections.RedBlackTree<ValueType, KeyOfValue, Compare>& that);
```

### Parameters

Name	Type	Description
that	const System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&	A red-black tree to copy.

**operator=(const System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&	A red-black tree to assign.

**RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&) Member Function**

Move constructor.

### Syntax

```
public RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>(System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&& that);
```

### Parameters

Name	Type	Description
that	System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&	A red-black tree to move from.

**operator=(System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&)**  
**Member Function**

Move assignment.

### Syntax

```
public void operator=(System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue,  
Compare>&& that);
```

### Parameters

Name	Type	Description
that	System.Collections.RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>&&	A red-black tree to assign.

**Begin() Member Function**

Returns a bidirectional iterator pointing to the beginning of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>
Begin();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns a bidirectional iterator pointing to the beginning of the red-black tree.

**Implementation**

[rbtree.cm, page 12](#)

**Begin() const Member Function**

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> Begin() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>-](#)

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

**Implementation**

[rbtree.cm, page 12](#)

**CBegin() const Member Function**

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> CBegin() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>-](#)

Returns a constant bidirectional iterator pointing to the beginning of the red-black tree.

**Implementation**

[rbtree.cm, page 12](#)

**CEnd() const Member Function**

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> CEnd() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>-](#)

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

**Implementation**

[rbtree.cm, page 13](#)

**CFind(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 System.Collections.RedBlackTree.CEnd iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> CFind(const KeyType& key) const;
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>](#)-

Returns a constant iterator pointing to it if found, or System.Collections.RedBlackTree.CEnd iterator otherwise.

**Clear() Member Function**

Makes the red-black tree empty.

**Syntax**

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

**Implementation**

[rbtree.cm, page 13](#)

**Count() const Member Function**

Returns the number of elements in the red-black tree.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the red-black tree.

**Implementation**

[rbtree.cm, page 13](#)

**End() Member Function**

Returns a bidirectional iterator pointing to one past the end of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>
End();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns a bidirectional iterator pointing to one past the end of the red-black tree.

**Implementation**

[rbtree.cm, page 13](#)

**End() const Member Function**

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> End() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>-](#)

Returns a constant bidirectional iterator pointing to one past the end of the red-black tree.

**Implementation**

[rbtree.cm, page 12](#)

**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 System.Collections.RedBlackTree.End iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*>
Find(const KeyType& key);
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType\\*>](#)

Returns an iterator pointing to the found element if search is successful, or System.Collections.RedBlackTree.End iterator otherwise.

**Implementation**

[rbtree.cm, page 13](#)

**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 System.Collections.RedBlackTree.CEnd.const iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const  
ValueType*> Find(const KeyType& key) const;
```

**Parameters**

Name	Type	Description
key	const KeyType&	A key to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<ValueType, const ValueType&, const ValueType\\*>](#)-

Returns a constant iterator pointing to the found element if search is successful, or System.Collections.RedBlackTree.CEnd.const iterator otherwise.

**Implementation**

[rbtree.cm, page 14](#)

**IsEmpty() const Member Function**

Returns true if the red-black tree is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the red-black tree is empty, false otherwise.

**Implementation**

[rbtree.cm, page 13](#)

**Remove(System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType\*>)**  
**Member Function**

Removes an element pointed by the given iterator from the red-black tree.

**Syntax**

```
public void Remove(System.Collections.RedBlackTreeNodeIterator<ValueType, ValueType&, ValueType*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.RedBlackTreeNodeIterator<- ValueType, ValueType&, ValueType*>	An iterator pointing to the ele- ment to be removed.

**Implementation**

[rbtree.cm, page 17](#)

**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 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](#)

`~RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>()` Member Function  
Destructor.

#### Syntax

```
public ~RedBlackTree<KeyType, ValueType, KeyOfValue, Compare>();
```

#### Implementation

[rbtree.cm, page 12](#)

### 3.7.21 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>](#)

### 3.7.21.1 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.

### 3.7.21.2 Member Functions

Member Function	Description
<code>RedBlackTreeNodeIterator&lt;T, R, P&gt;()</code>	Constructor. Default constructs a red-black tree node iterator.
<code>RedBlackTreeNodeIterator&lt;T, R, P&gt;(const System.Collections.RedBlackTreeNodeIterator&lt;- T, R, P&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.- RedBlackTreeNodeIterator&lt;T, R, P&gt;&amp;)</code>	Copy assignment.
<code>RedBlackTreeNodeIterator&lt;T, R, P&gt;(System.- Collections.RedBlackTreeNodeIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.- RedBlackTreeNodeIterator&lt;T, R, P&gt;&amp;&amp;)</code>	Move assignment.
<code>GetNode() const</code>	Returns a pointer to a red-black tree node.
<code>RedBlackTreeNodeIterator&lt;T, R, P&gt;(System.- Collections.RedBlackTreeNode&lt;T&gt;*)</code>	Constructor. Constructs an iterator pointing to a red-black tree node.
<code>operator*() const</code>	Returns a reference to an element.
<code>operator++()</code>	Advances the iterator pointing to the next element in the red-black tree.
<code>operator--()</code>	Backs the iterator pointing to the previous element in the red-black tree.
<code>operator-&gt;() const</code>	Returns a pointer to an element.

**RedBlackTreeNodeIterator<T, R, P>() Member Function**

Constructor. Default constructs a red-black tree node iterator.

**Syntax**

```
public RedBlackTreeNodeIterator<T, R, P>();
```

**Implementation**

[rbtree.cm, page 11](#)

**RedBlackTreeNodeIterator<T, R, P>(const System.Collections.RedBlackTreeNodeIterator<T, R, P>&) Member Function**

Copy constructor.

**Syntax**

```
public RedBlackTreeNodeIterator<T, R, P>(const System.Collections.RedBlackTreeNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.RedBlackTreeNodeIterator<- T, R, P>&	Argument to copy.

**operator=(const System.Collections.RedBlackTreeNodeIterator<T, R, P>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.RedBlackTreeNodeIterator<T, R, P>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.RedBlackTreeNodeIterator<T, R, P>&	Argument to assign.

**RedBlackTreeNodeIterator<T, R, P>(System.Collections.RedBlackTreeNodeIterator<T, R, P>&&) Member Function**

Move constructor.

**Syntax**

```
public RedBlackTreeNodeIterator<T, R, P>(System.Collections.RedBlackTreeNodeIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.RedBlackTreeNodeIterator<T, R, P>&&	Argument to move from.

**operator=(System.Collections.RedBlackTreeNodeIterator<T, R, P>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.RedBlackTreeNodeIterator<T, R, P>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.RedBlackTreeNodeIterator<T, R, P>&&	Argument to assign from.

**GetNode() const Member Function**

Returns a pointer to a red-black tree node.

**Syntax**

```
public System.Collections.RedBlackTreeNode<T>* GetNode() const;
```

**Returns**

[System.Collections.RedBlackTreeNode<T>\\*](#)

Returns a pointer to a red-black tree node.

**Implementation**

[rbtree.cm, page 11](#)

**RedBlackTreeNodeIterator<T, R, P>(System.Collections.RedBlackTreeNode<T>\*) Member Function**

Constructor. Constructs an iterator pointing to a red-black tree node.

**Syntax**

```
public RedBlackTreeNodeIterator<T, R, P>(System.Collections.RedBlackTreeNode<T>*  
node_);
```

**Parameters**

Name	Type	Description
node_	System.Collections.RedBlackTreeNode<T>*	A pointer to a red-black tree node.

**Implementation**

[rbtree.cm, page 11](#)

**operator\*() const Member Function**

Returns a reference to an element.

**Syntax**

```
public R operator*() const;
```

**Returns**

R

Returns a reference to an element.

**Implementation**

[rbtree.cm, page 11](#)

**operator++() Member Function**

Advances the iterator pointing to the next element in the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, R, P>& operator++();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, R, P>&](#)

Returns a reference to the iterator.

**Implementation**

[rbtree.cm, page 11](#)

**operator–() Member Function**

Backs the iterator pointing to the previous element in the red-black tree.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, R, P>& operator--();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, R, P>&](#)

Returns a reference to the iterator.

**Implementation**

[rbtree.cm, page 11](#)

**operator->() const Member Function**

Returns a pointer to an element.

**Syntax**

```
public P operator->() const;
```

**Returns**

P

Returns a pointer to an element.

**Implementation**

[rbtree.cm, page 11](#)

### 3.7.21.3 Nonmember Functions

Function	Description
<code>operator==(const System.Collections.-RedBlackTreeNodeIterator&lt;T, R, P&gt;&amp;, const System.Collections.RedBlackTreeNodeIterator&lt;-T, R, P&gt;&amp;)</code>	Compares two red-black tree node iterators for equality.

**operator==(const System.Collections.RedBlackTreeNodeIterator<T, R, P>&, const System.Collections.RedBlackTreeNodeIterator<T, R, P>&) Function**

Compares two red-black tree node iterators for equality.

**Syntax**

```
public bool operator==(const System.Collections.RedBlackTreeNodeIterator<T, R, P>&
left, const System.Collections.RedBlackTreeNodeIterator<T, R, P>& right);
```

**Parameters**

Name	Type	Description
left	const System.Collections.RedBlackTreeNodeIterator<T, R, P>&	The first red-black tree node iterator.
right	const System.Collections.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

### 3.7.22 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

T is [Semiregular](#) and C is [Relation](#) and C.Domain is T

#### Model of

[BidirectionalContainer<T>](#)

#### Default Template Arguments

C = [System.Less<T>](#)

#### 3.7.22.1 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();
```

### 3.7.22.2 Type Definitions

Name	Type	Description
Compare	C	A relation used to order elements in the set.
ConstIterator	System.Collections.- RedBlackTreeNodeIterator<T, T&, const T*>	A constant iterator type.
Iterator	System.Collections.- RedBlackTreeNodeIterator<T, T&, T*>	An iterator type.
KeyType	T	The key type is equal to the System.Collections.Set.-ValueType for the red-black tree.
ValueType	T	The type of the element in the set.

### 3.7.22.3 Member Functions

Member Function	Description
<code>Set&lt;T, C&gt;()</code>	Default constructor. Constructs an empty set.
<code>Set&lt;T, C&gt;(const System.Collections.Set&lt;T, C&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.Set&lt;T, C&gt;&amp;)</code>	Copy assignment.
<code>Set&lt;T, C&gt;(System.Collections.Set&lt;T, C&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.Set&lt;T, C&gt;&amp;&amp;)</code>	Move assignment.
<code>Begin()</code>	Returns an iterator pointing to the beginning of the set, or <code>PEnd()</code> if the set is empty.
<code>Begin() const</code>	Returns a constant iterator pointing to the beginning of the set, or <code>PEnd() const</code> if the set is empty.
<code>CBegin() const</code>	Returns a constant iterator pointing to the beginning of the set, or <code>PEnd() const</code> if the set is empty.
<code>PEnd() const</code>	Returns a constant iterator pointing one past the end of the set.
<code>CFind(const T&amp;) const</code>	Searches an element from the set and returns a constant iterator pointing to it if found, or <code>PEnd() const</code> iterator otherwise.

<code>Clear()</code>	Makes the set empty.
<code>Count() const</code>	Returns the number of elements in the set.
<code>End()</code>	Returns an iterator pointing one past the end of the set.
<code>End() const</code>	Returns a constant iterator pointing one past the end of the set.
<code>Find(const T&amp;)</code>	Searches an element from the set and returns an iterator pointing to it if found, or <code>End()</code> iterator otherwise.
<code>Find(const T&amp;) const</code>	Searches an element from the set and returns a constant iterator pointing to it if found, or <code>CEnd() const</code> iterator otherwise.
<code>Insert(const T&amp;)</code>	Inserts an element to the set if it is not already there and returns a pair consisting of an iterator pointing to the inserted element and <code>true</code> if element was inserted, and a pair consisting of an iterator pointing to the existing element and <code>false</code> otherwise.
<code>IsEmpty() const</code>	Returns true if the set is empty, false otherwise.
<code>Remove(System.Collections.-RedBlackTreeNodeIterator&lt;T, T&amp;, T*&gt;)</code>	Removes an element pointed by the given iterator from the set.
<code>Remove(const T&amp;)</code>	Removes an element from the set. If the element was not found, does nothing.

**Set<T, C>() Member Function**

Default constructor. Constructs an empty set.

**Syntax**

```
public Set<T, C>();
```

**Implementation**

[set.cm](#), page 1

**Set<T, C>(const System.Collections.Set<T, C>&) Member Function**

Copy constructor.

**Syntax**

```
public Set<T, C>(const System.Collections.Set<T, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Set<T, C>&	Argument to copy.

**operator=(const System.Collections.Set<T, C>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.Set<T, C>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Set<T, C>&	Argument to assign.

**Set<T, C>(System.Collections.Set<T, C>&&) Member Function**

Move constructor.

**Syntax**

```
public Set<T, C>(System.Collections.Set<T, C>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Set<T, C>&&	Argument to move from.

**operator=(System.Collections.Set<T, C>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Set<T, C>&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Collections.Set&lt;T, C&gt;&amp;&amp;</a>	Argument to assign from.

**Begin() Member Function**

Returns an iterator pointing to the beginning of the set, or [End\(\)](#) if the set is empty.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, T&, T*> Begin();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the beginning of the set, or [End\(\)](#) if the set is empty.

**Implementation**

[set.cm, page 1](#)

**Begin() const Member Function**

Returns a constant iterator pointing to the beginnig of the set, or CEnd() const ir the set is empty.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> Begin()  
const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginnig of the set, or CEnd() const ir the set is empty.

**CBegin() const Member Function**

Returns a constant iterator pointing to the beginning of the set, or [CEnd\(\) const](#) if the set is empty.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> CBegin()  
const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to the beginning of the set, or [CEnd\(\) const](#) if the set is empty.

**Implementation**

[set.cm, page 1](#)

**CEnd() const Member Function**

Returns a constant iterator pointing one past the end of the set.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> CEnd()  
const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the set.

**Implementation**

[set.cm, page 2](#)

**CFind(const T&) const Member Function**

Searches an element from the set and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> CFind(const T& key) const;
```

**Parameters**

Name	Type	Description
key	const T&	An element to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Clear() Member Function**

Makes the set empty.

**Syntax**

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

**Implementation**

[set.cm, page 2](#)

**Count() const Member Function**

Returns the number of elements in the set.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the set.

**Implementation**

[set.cm, page 2](#)

**End() Member Function**

Returns an iterator pointing one past the end of the set.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, T&, T*> End();
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing one past the end of the set.

**Implementation**

[set.cm, page 2](#)

**End() const Member Function**

Returns a constant iterator pointing one past the end of the set.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> End() const;
```

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing one past the end of the set.

**Find(const T&) Member Function**

Searches an element from the set and returns an iterator pointing to it if found, or [End\(\)](#) iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, T&, T*> Find(const T& key);
```

**Parameters**

Name	Type	Description
key	const T&	An element to seach.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, T&, T\\*>](#)

Returns an iterator pointing to the found element if the search was successful, or [End\(\)](#) iterator otherwise.

**Implementation**

[set.cm, page 2](#)

**Find(const T&) const Member Function**

Searches an element from the set and returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Syntax**

```
public System.Collections.RedBlackTreeNodeIterator<T, const T&, const T*> Find(const T& key) const;
```

**Parameters**

Name	Type	Description
key	const T&	An element to search.

**Returns**

[System.Collections.RedBlackTreeNodeIterator<T, const T&, const T\\*>](#)

Returns a constant iterator pointing to it if found, or [CEnd\(\) const](#) iterator otherwise.

**Insert(const T&)** Member Function

Inserts an element to the set if it is not already there and returns a pair consisting of an iterator pointing to the inserted element and **true** if element was inserted, and a pair consisting of an iterator pointing to the existing element and **false** otherwise.

**Syntax**

```
public System.Pair<System.Collections.RedBlackTreeNodeIterator<T, T&, T*>, bool>
Insert(const T& value);
```

**Parameters**

Name	Type	Description
value	const T&	An element to insert.

**Returns**

[System.Pair<System.Collections.RedBlackTreeNodeIterator<T, T&, T\\*>, bool>](#)

Returns a pair consisting of an iterator pointing to the inserted element and **true** if element was inserted, and a pair consisting of an iterator pointing to the existing element and **false** otherwise.

**IsEmpty() const Member Function**

Returns true if the set is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the set is empty, false otherwise.

**Implementation**

[set.cm, page 2](#)

**Remove(System.Collections.RedBlackTreeNodeIterator<T, T&, T\*>) Member Function**

Removes an element pointed by the given iterator from the set.

**Syntax**

```
public void Remove(System.Collections.RedBlackTreeNodeIterator<T, T&, T*> pos);
```

**Parameters**

Name	Type	Description
pos	System.Collections.RedBlackTreeNodeIterator<T, T&, T*>	An iterator pointing to the element to remove.

**Remove(const T&)** Member Function

Removes an element from the set. If the element was not found, does nothing.

**Syntax**

```
public bool Remove(const T& key);
```

**Parameters**

Name	Type	Description
key	const T&	An element to remove.

**Returns**

bool

Returns true if element was removed, false otherwise.

**Implementation**

[set.cm, page 2](#)

**3.7.22.4 Nonmember Functions**

Function	Description
<code>operator&lt;(const System.Collections.Set&lt;T, C&gt; &amp;, const System.Collections.Set&lt;T, C&gt;&amp;)</code>	Compares two sets for less than relationship and returns true if the first set comes lexicographically before the second set, false otherwise.
<code>operator==(const System.Collections.Set&lt;T, C&gt;&amp;, const System.Collections.Set&lt;T, C&gt;&amp;)</code>	Compares two sets for equality and returns true if both contain the same number of pairwise equal elements, false otherwise.

### **operator<(const System.Collections.Set<T, C>&, const System.Collections.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 bool operator<(const System.Collections.Set<T, C>& left, const System.Collections.Set<T, C>& right);
```

#### Constraint

T is [Semiregular](#) and C is [Relation](#) and C.Domain is T

#### Parameters

Name	Type	Description
left	const System.Collections.Set<T, C>&	The first set.
right	const System.Collections.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[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](#)

### **operator==(const System.Collections.Set<T, C>&, const System.Collections.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 bool operator==(const System.Collections.Set<T, C>& left, const System.Collections.Set<T, C>& right);
```

#### **Constraint**

T is [Regular](#) and C is [Relation](#) and C.Domain is T

#### **Parameters**

Name	Type	Description
left	const System.Collections.Set<T, C>&	The first set.
right	const System.Collections.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

### 3.7.23 Stack<T> Class

A last-in first-out data structure.

## Syntax

```
public class Stack<T>;
```

## Constraint

T is Semiregular

### 3.7.23.1 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();
    }
}
```

### 3.7.23.2 Type Definitions

Name	Type	Description
ValueType	T	The type of the stack element.

### 3.7.23.3 Member Functions

Member Function	Description
<code>Stack&lt;T&gt;()</code>	Default constructor. Constructs an empty stack.
<code>Stack&lt;T&gt;(const System.Collections.Stack&lt;T&gt;&amp;)</code>	Copy constructor.
<code>operator=(const System.Collections.Stack&lt;T&gt;&amp;)</code>	Copy assignment.
<code>Stack&lt;T&gt;(System.Collections.Stack&lt;T&gt;&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Collections.Stack&lt;T&gt;&amp;&amp;)</code>	Move assignment.
<code>Clear()</code>	Makes the stack empty.
<code>Count() const</code>	Returns the number of elements in the stack.
<code>IsEmpty() const</code>	Returns true if the stack is empty, false otherwise.
<code>Pop()</code>	Removes an element from the top of the stack and returns it.
<code>Push(T&amp;&amp;)</code>	Moves an element to the top of the stack.
<code>Push(const T&amp;)</code>	Copies an element to the top of the stack.
<code>Top()</code>	Return a reference to the element at the top of the stack.
<code>Top() const</code>	Returns a constant reference to the element at the top of the stack.

**Stack<T>() Member Function**

Default constructor. Constructs an empty stack.

**Syntax**

```
public Stack<T>();
```

**Implementation**

[stack.cm, page 1](#)

**Stack<T>(const System.Collections.Stack<T>&) Member Function**

Copy constructor.

**Syntax**

```
public Stack<T>(const System.Collections.Stack<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Stack<T>&	Argument to copy.

**operator=(const System.Collections.Stack<T>&) Member Function**

Copy assignment.

**Syntax**

```
public void operator=(const System.Collections.Stack<T>& that);
```

**Parameters**

Name	Type	Description
that	const System.Collections.Stack<T>&	Argument to assign.

**Stack<T>(System.Collections.Stack<T>&&) Member Function**

Move constructor.

**Syntax**

```
public Stack<T>(System.Collections.Stack<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Stack<T>&&	Argument to move from.

**Implementation**

[stack.cm, page 1](#)

**operator=(System.Collections.Stack<T>&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Collections.Stack<T>&& that);
```

**Parameters**

Name	Type	Description
that	System.Collections.Stack<T>&&	Argument to assign from.

**Implementation**

[stack.cm, page 1](#)

**Clear() Member Function**

Makes the stack empty.

**Syntax**

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

**Implementation**

[stack.cm](#), page 2

**Count() const Member Function**

Returns the number of elements in the stack.

**Syntax**

```
public int Count() const;
```

**Returns**

int

Returns the number of elements in the stack.

**Implementation**

[stack.cm, page 1](#)

**IsEmpty() const Member Function**

Returns true if the stack is empty, false otherwise.

**Syntax**

```
public bool IsEmpty() const;
```

**Returns**

bool

Returns true if the stack is empty, false otherwise.

**Implementation**

[stack.cm, page 1](#)

**Pop() Member Function**

Removes an element from the top of the stack and returns it.

**Syntax**

```
public T Pop();
```

**Returns**

T

Returns the removed element.

**Push(T&&)** Member Function

Moves an element to the top of the stack.

**Syntax**

```
public void Push(T&& item);
```

**Parameters**

Name	Type	Description
item	T&&	An element to push.

**Push(const T&)** Member Function

Copies an element to the top of the stack.

**Syntax**

```
public void Push(const T& item);
```

**Parameters**

Name	Type	Description
item	const T&	An element to push.

**Top() Member Function**

Return a reference to the element at the top of the stack.

**Syntax**

```
public T& Top();
```

**Returns**

T&

Return a reference to the element at the top of the stack.

**Implementation**

[stack.cm, page 2](#)

**Top() const Member Function**

Returns a constant reference to the element at the top of the stack.

**Syntax**

```
public const T& Top() const;
```

**Returns**

const T&

Returns a constant reference to the element at the top of the stack.

**Implementation**

[stack.cm, page 2](#)

## 3.8 Functions

Function	Description
<code>ConstructiveCopy(ValueType*, ValueType*, int)</code>	Copies a sequence of values by constructing them into raw memory.
<code>ConstructiveMove(ValueType*, ValueType*, int)</code>	Moves a sequence of values by moving them into raw memory.
<code>Destroy(ValueType*, int)</code>	Destroys a sequence of values but does not release the memory allocated for them.
<code>GetHashCode(char)</code>	Returns a hash code for given character.
<code>GetHashCode(const System.String&amp;)</code>	Computes a hash code for the given string and returns it.
<code>GetHashCode(long)</code>	Returns a hash code for the given long value.
<code>GetHashCode(ulong)</code>	Returns a hash code for the given ulong value.
<code>GetHashCode(void*)</code>	Returns a hash code for the given pointer.

### 3.8.24 ConstructiveCopy(ValueType\*, ValueType\*, int) Function

Copies a sequence of values by constructing them into raw memory.

#### Syntax

```
public void ConstructiveCopy(ValueType* to, ValueType* from, int count);
```

#### Constraint

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

### 3.8.25 ConstructiveMove(ValueType\*, ValueType\*, int) Function

Moves a sequence of values by moving them into raw memory.

#### Syntax

```
public void ConstructiveMove(ValueType* to, ValueType* from, int count);
```

#### Constraint

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

### 3.8.26 Destroy(ValueType\*, int) Function

Destroys a sequence of values but does not release the memory allocated for them.

#### Syntax

```
public void Destroy(ValueType* items, int count);
```

#### Constraint

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

### 3.8.27 GetHashCode(char) Function

Returns a hash code for given character.

#### Syntax

```
public ulong GetHashCode(char x);
```

#### Parameters

Name	Type	Description
x	char	A character.

#### Returns

ulong

Returns a hash code for the given character.

### 3.8.28 GetHashCode(const System.String&) Function

Computes a hash code for the given string and returns it.

#### Syntax

```
public ulong GetHashCode(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string.

#### Returns

ulong

Returns a hash code for the given string.

### 3.8.29 GetHashCode(long) Function

Returns a hash code for the given long value.

#### Syntax

```
public ulong GetHashCode(long x);
```

#### Parameters

Name	Type	Description
x	long	A long value.

#### Returns

ulong

Returns a hash code for the given long value.

### 3.8.30 GetHashCode(ulong) Function

Returns a hash code for the given ulong value.

#### Syntax

```
public ulong GetHashCode(ulong x);
```

#### Parameters

Name	Type	Description
x	ulong	A ulong value.

#### Returns

ulong

Returns a hash code for the given ulong value.

### 3.8.31 GetHashCode(void\*) Function

Returns a hash code for the given pointer.

#### Syntax

```
public ulong GetHashCode(void* x);
```

#### Parameters

Name	Type	Description
x	void*	A pointer.

#### Returns

ulong

Returns a hash code for the given pointer.

# 4 System.Concepts Namespace

Contains system library concepts.

Figures 4.1, 4.2, 4.3, 4.4 and 4.5 contain the concepts in this namespace.

Figure 4.1: Concept Diagram 1: Basic Concepts



Figure 4.2: Concept Diagram 2: Iterator Concepts

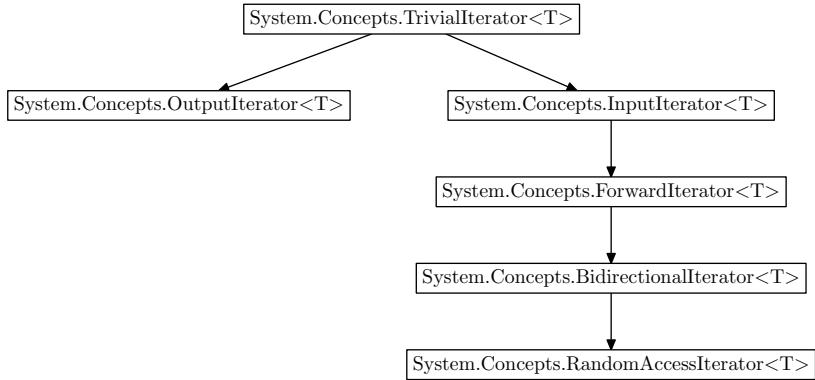


Figure 4.3: Concept Diagram 3: Container Concepts

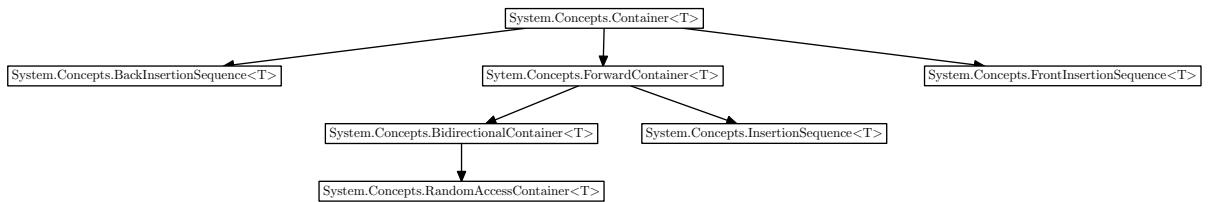


Figure 4.4: Concept Diagram 4: Functional Concepts

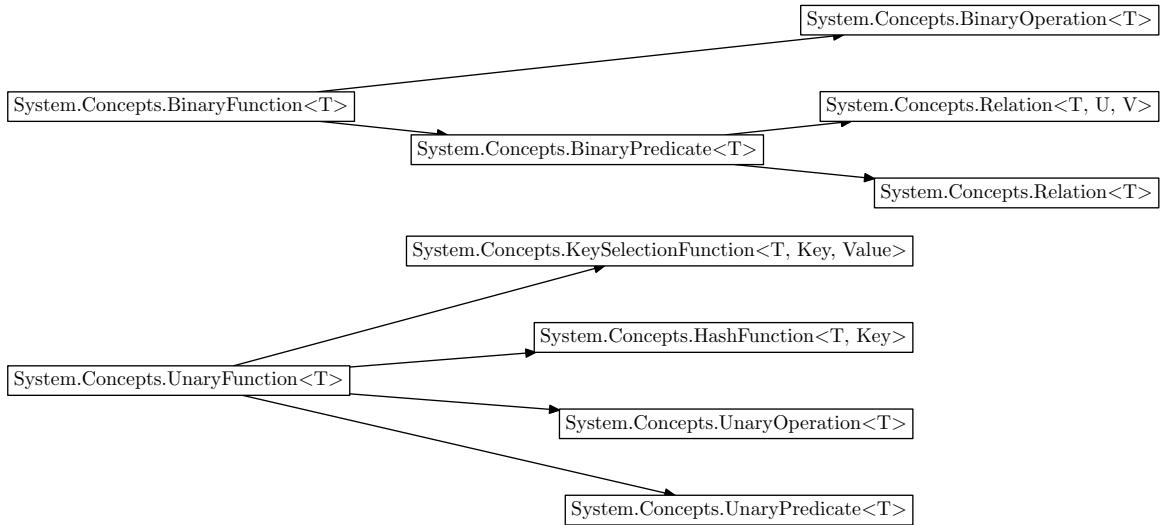
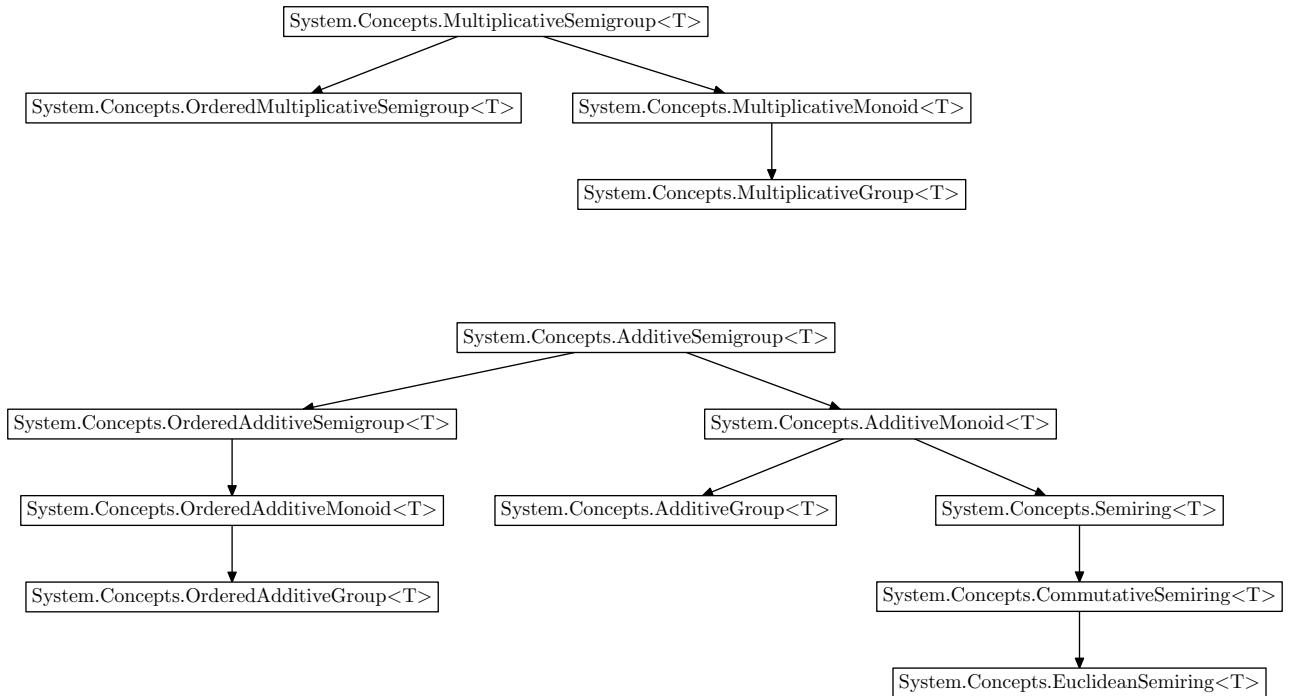


Figure 4.5: Concept Diagram 5: Algebraic Concepts



## 4.9 Concepts

Concept	Description
<a href="#">AdditiveGroup&lt;T&gt;</a>	An additive group is an additive monoid that has an inverse of addition operation.
<a href="#">AdditiveMonoid&lt;T&gt;</a>	An additive semigroup that has an identity element 0 is called an additive monoid.
<a href="#">AdditiveSemigroup&lt;T&gt;</a>	A set with an associative and commutative addition operation + is called an additive semigroup.
<a href="#">BackInsertionSequence&lt;T&gt;</a>	A container with an <b>Add</b> member function, that adds elements to the end of the container is called a back insertion sequence.
<a href="#">BidirectionalContainer&lt;T&gt;</a>	A container whose iterators are bidirectional iterators is a bidirectional container.
<a href="#">BidirectionalIterator&lt;T&gt;</a>	An iterator that can be incremented and decremented is a bidirectional iterator.
<a href="#">BinaryFunction&lt;T&gt;</a>	A function object that implements an <b>operator()</b> member function that accepts two parameters is called a binary function.
<a href="#">BinaryOperation&lt;T&gt;</a>	A binary function whose result type is same as its first argument type is called a binary operation.
<a href="#">BinaryPredicate&lt;T&gt;</a>	A binary function whose result type is <b>bool</b> is called a binary predicate.
<a href="#">CommutativeSemiring&lt;T&gt;</a>	A semiring with commutative multiplication operation is called a commutative semiring.
<a href="#">Container&lt;T&gt;</a>	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 <b>ValueType</b> , and provide iterator types for iterating through the container, for example.
<a href="#">CopyAssignable&lt;T, U&gt;</a>	Types T and U satisfy the <a href="#">CopyAssignable&lt;T, U&gt;</a> concept if type T has an copy assignment operator taking a parameter of type U.
<a href="#">CopyAssignable&lt;T&gt;</a>	Copy assignable type can be target of an copy assignment operation.

<code>CopyConstructible&lt;T&gt;</code>	A copy constructible type can be copied.
<code>Copyable&lt;T&gt;</code>	A copy constructible and copy assignable type is copyable.
<code>DefaultConstructible&lt;T&gt;</code>	A default constructible type has a constructor that takes no parameters.
<code>Destructible&lt;T&gt;</code>	A destructible type has a user defined or compiler generated destructor, or is trivially destructible.
<code>EqualityComparable&lt;T, U&gt;</code>	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.
<code>EqualityComparable&lt;T&gt;</code>	An equality comparable type can be compared for equality and inequality.
<code>EuclideanSemiring&lt;T&gt;</code>	Euclidean semiring is a commutative semiring that has division and remainder operations.
<code>ForwardContainer&lt;T&gt;</code>	A container whose iterators are forward iterators is a forward container.
<code>ForwardIterator&lt;T&gt;</code>	A multipass iterator that can be incremented is a forward iterator.
<code>FrontInsertionSequence&lt;T&gt;</code>	A container with an <b>InsertFront</b> member function is called a front insertion sequence.
<code>HashFunction&lt;T, Key&gt;</code>	A unary function that computes a hash code from a key is a hash function.
<code>InputIterator&lt;T&gt;</code>	A one-pass iterator that can be incremented and compared for equality is an input iterator.
<code>InsertionSequence&lt;T&gt;</code>	A container with an <b>Insert</b> member function is called an insertion sequence.
<code>Integer&lt;I&gt;</code>	An integer type supports the usual integer operations.
<code>KeySelectionFunction&lt;T, Key, Value&gt;</code>	A unary function that extracts a key from a value type is called a key selection function.

<a href="#">LessThanComparable&lt;T, U&gt;</a>	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.
<a href="#">LessThanComparable&lt;T&gt;</a>	An less than comparable type can be compared for less than, greater than, less than or equal to and greater than or equal to relations.
<a href="#">Movable&lt;T&gt;</a>	A move constructible and move assignable type is movable.
<a href="#">MoveAssignable&lt;T&gt;</a>	A type equipped with move assignment operator is called move assignable.
<a href="#">MoveConstructible&lt;T&gt;</a>	A type equipped with move constructor is called move constructible.
<a href="#">MultiplicativeGroup&lt;T&gt;</a>	A multiplicative monoid with a division operation is called a multiplicative group.
<a href="#">MultiplicativeMonoid&lt;T&gt;</a>	A multiplicative semigroup with an identity element is called a multiplicative monoid.
<a href="#">MultiplicativeSemigroup&lt;T&gt;</a>	A set with an associative multiplication operation is called a multiplicative semigroup.
<a href="#">OrderedAdditiveGroup&lt;T&gt;</a>	An ordered additive monoid that forms also an additive group is called an ordered additive group.
<a href="#">OrderedAdditiveMonoid&lt;T&gt;</a>	An ordered additive semigroup that forms also an additive monoid is called an ordered additive monoid.
<a href="#">OrderedAdditiveSemigroup&lt;T&gt;</a>	An additive semigroup with a total ordering relation on its elements is called an ordered additive semigroup.
<a href="#">OrderedMultiplicativeSemigroup&lt;T&gt;</a>	A multiplicative semigroup with a total ordering relation on its elements is called an ordered multiplicative semigroup.
<a href="#">OutputIterator&lt;T&gt;</a>	A writable iterator that is incrementable is an output iterator.
<a href="#">RandomAccessContainer&lt;T&gt;</a>	A container whose iterators are random access iterators is a random access container.

<a href="#">RandomAccessIterator&lt;T&gt;</a>	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.
<a href="#">Regular&lt;T&gt;</a>	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.
<a href="#">Relation&lt;T, U, V&gt;</a>	A binary predicate with two not necessarily same argument types satisfy a multiparameter relation concept.
<a href="#">Relation&lt;T&gt;</a>	A binary predicate whose argument types are same is called a relation.
<a href="#">Semiregular&lt;T&gt;</a>	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.
<a href="#">Semiring&lt;T&gt;</a>	A set with addition and multiplication operations that are connected with given axioms is called a semiring.
<a href="#">SignedInteger&lt;I&gt;</a>	An integer type with a conversion from <b>sbyte</b> is called a signed integer.
<a href="#">TotallyOrdered&lt;T, U&gt;</a>	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.
<a href="#">TotallyOrdered&lt;T&gt;</a>	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.
<a href="#">TrivialIterator&lt;T&gt;</a>	A trivial iterator concept collects together requirements common to all iterator types.
<a href="#">UnaryFunction&lt;T&gt;</a>	A function object that implements an <b>operator()</b> 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.

### 4.9.1 AdditiveGroup<T> Concept

An additive group is an additive monoid that has an inverse of addition operation.

#### Syntax

```
public concept AdditiveGroup<T>;
```

#### Refines

[System.Concepts.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.

## 4.9.2 AdditiveMonoid<T> Concept

An additive semigroup that has an identity element 0 is called an additive monoid.

### Syntax

```
public concept AdditiveMonoid<T>;
```

### Refines

[System.Concepts.AdditiveSemigroup<T>](#)

### Constraints

[ConversionFromSByte<T>](#) or [ConversionFromByte<T>](#)

### Axioms

```
zeroIsIdentityElement(T a)
{
    a + 0 == a && 0 + a == a;
}
```

### Models

Integer and floating-point types with + are partial models of an additive monoid.

### 4.9.3 AdditiveSemigroup<T> Concept

A set with an associative and commutative addition operation `+` is called an additive semigroup.

#### Syntax

```
public concept AdditiveSemigroup<T>;
```

#### Constraints

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.

#### 4.9.4 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

[System.Concepts.Container<T>](#)

##### Constraints

```
void T.Add(T.ValueType);
```

##### Models

[List<T>](#) and [LinkedList<T>](#) are a back insertion sequences.

### 4.9.5 BidirectionalContainer<T> Concept

A container whose iterators are bidirectional iterators is a bidirectional container.

#### Syntax

```
public concept BidirectionalContainer<T>;
```

#### Refines

[System.Concepts.ForwardContainer<T>](#)

#### Constraints

T.Iterator is [BidirectionalIterator](#) and T.ConstIterator is [BidirectionalIterator](#)

#### Models

[List<T>](#), [LinkedList<T>](#), [Set<T, C>](#) and [Map<Key, Value, KeyCompare>](#) are bidirectional containers.

### 4.9.6 BidirectionalIterator<T> Concept

An iterator that can be incremented and decremented is a bidirectional iterator.

#### Syntax

```
public concept BidirectionalIterator<T>;
```

#### Refines

[System.Concepts.ForwardIterator<T>](#)

#### Constraints

T& operator-();

#### Models

[LinkedListNodeIterator<T, R, P>](#) and [RedBlackTreeNodeIterator<T, R, P>](#) are bidirectional iterators.

### 4.9.7 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

T is [Semiregular](#)

typename T.FirstArgumentType;

typename T.SecondArgumentType;

typename T.ResultType;

T.FirstArgumentType is [Semiregular](#) and T.SecondArgumentType is [Semiregular](#)

T.ResultType operator()(T.FirstArgumentType, T.SecondArgumentType);

#### Models

[Plus<T>](#), [Minus<T>](#), [Multiplies<T>](#),  
[Divides<T>](#) and [Remainder<T>](#) are  
binary functions.

## 4.9.8 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

[System.Concepts.BinaryFunction<T>](#)

### Constraints

T.ResultType is T.FirstArgumentType

### Models

[Plus<T>](#), [Minus<T>](#), [Multiplies<T>](#),  
[Divides<T>](#) and [Remainder<T>](#) are  
binary operations.

### 4.9.9 BinaryPredicate<T> Concept

A binary function whose result type is **bool** is called a binary predicate.

#### Syntax

```
public concept BinaryPredicate<T>;
```

#### Refines

[System.Concepts.BinaryFunction<T>](#)

#### Constraints

T.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.

## 4.9.10 CommutativeSemiring<T> Concept

A semiring with commutative multiplication operation is called a commutative semiring.

### Syntax

```
public concept CommutativeSemiring<T>;
```

### Refines

[System.Concepts.Semiring<T>](#)

### Axioms

```
multiplicationIsCommutative(T a, T b)
{
    a * b == b * a;
}
```

### Models

Integer types with + and \* are partial models of a commutative semiring.

### 4.9.11 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

`T` is [Semiregular](#)

`typename T.ValueType;`

`typename T.Iterator;`

`typename T.ConstIterator;`

`T.Iterator` is [TrivialIterator](#) and `T.ConstIterator` is [TrivialIterator](#) and `T.ValueType` is `T.Iterator.ValueType`

`T.Iterator T.Begin();`

`T.ConstIterator T.CBegin();`

`T.Iterator T.End();`

`T.ConstIterator T.CEnd();`

`int T.Count();`

`bool T.IsEmpty();`

#### Models

`List<T>,`  
`LinkedList<T>,`  
`Set<T, C>,`  
`Map<Key, Value, KeyCompare>,`  
`HashSet<T, H, C>,`  
`HashMap<K, T, H, C>` and  
`ForwardList<T>` are containers.

## 4.9.12 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&);
```

#### 4.9.12.1 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
}
```

### 4.9.13 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 [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.

#### 4.9.13.1 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
}
```

## 4.9.14 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 [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.

### 4.9.14.1 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
}
```

### 4.9.15 Copyable<T> Concept

A copy constructible and copy assignable type is copyable.

#### Syntax

```
public concept Copyable<T>;
```

#### Constraints

T is [CopyConstructible](#) and T is [CopyAssignable](#)

## 4.9.16 DefaultConstructible<T> Concept

A default constructible type has a constructor that takes no parameters.

### Syntax

```
public concept DefaultConstructible<T>;
```

### Constraints

`T();`

`NonReferenceType<T>`

### Models

All built-in types (`int`, `double`, `bool`, `char`, etc...) are default constructible.

All types in the `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...

#### 4.9.16.1 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..."'
}
```

### 4.9.17 Destructible<T> Concept

A destructible type has a user defined or compiler generated destructor, or is trivially destructible.

#### Syntax

```
public concept Destructible<T>;
```

#### Constraints

```
 $\sim T();$ 
```

#### Models

All types in Cmajor are destructible.

#### 4.9.17.1 Example

```
using System;
using System.Concepts;

// A has trivial destructor:

class A
{
}

// B has compiler generated destructor:

class B
{
    public default  $\sim B()$ ;
}

// C has user defined destructor:

class C
{
    public  $\sim C()$  {}
}

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
}
```



### 4.9.18 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

T is [EqualityComparable](#) and U is [EqualityComparable](#) and CommonType is [EqualityComparable](#)

### 4.9.19 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);
}
```

## 4.9.20 EuclideanSemiring<T> Concept

Euclidean semiring is a commutative semiring that has division and remainder operations.

### Syntax

```
public concept EuclideanSemiring<T>;
```

### Refines

[System.Concepts.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;
}
```

## 4.9.21 ForwardContainer<T> Concept

A container whose iterators are forward iterators is a forward container.

### Syntax

```
public concept ForwardContainer<T>;
```

### Refines

[System.Concepts.Container<T>](#)

### Constraints

T.Iterator is [ForwardIterator](#) and T.ConstIterator is [ForwardIterator](#)

### Models

[ForwardList<T>](#) is a proper forward container.

All containers in the [Collections](#) namespace are also forward containers.

## 4.9.22 `ForwardIterator<T>` Concept

A multipass iterator that can be incremented is a forward iterator.

### Syntax

```
public concept ForwardIterator<T>;
```

### Refines

[System.Concepts.InputIterator<T>](#)

### Constraints

T is [OutputIterator](#)

### Models

[ForwardListNodeIterator<T, R, P>](#) is a proper forward iterator.

All Cmajor iterators are also forward iterators.

### 4.9.23 FrontInsertionSequence<T> Concept

A container with an **InsertFront** member function is called a front insertion sequence.

#### Syntax

```
public concept FrontInsertionSequence<T>;
```

#### Refines

[System.Concepts.Container<T>](#)

#### Constraints

```
T.Iterator T.InsertFront(T.ValueType);
```

#### Models

[List<T>](#), [LinkedList<T>](#) and [ForwardList<T>](#) are front insertion sequences.

## 4.9.24 HashFunction<T, Key> Concept

A unary function that computes a hash code from a key is a hash function.

### Syntax

```
public concept HashFunction<T, Key>;
```

### Refines

[System.Concepts.UnaryFunction<T>](#)

### Constraints

T.ArgumentType is Key and T.ResultType is ulong

### Models

[Hasher<T>](#) is a hash function.

## 4.9.25 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

[System.Concepts.TrivialIterator<T>](#)

### Constraints

T& operator++();

T is [Regular](#)

## 4.9.26 InsertionSequence<T> Concept

A container with an **Insert** member function is called an insertion sequence.

### Syntax

```
public concept InsertionSequence<T>;
```

### Refines

[System.Concepts.ForwardContainer<T>](#)

### Constraints

```
T.Iterator T.Insert(T.Iterator, T.ValueType);
```

### Models

[List<T>](#) and [LinkedList<T>](#) are insertion sequences.

### 4.9.27 Integer<I> Concept

An integer type supports the usual integer operations.

#### Syntax

```
public concept Integer<I>;
```

#### Constraints

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.

## 4.9.28 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

[System.Concepts.UnaryFunction<T>](#)

### Constraints

T.ArgumentType is Value and T.ResultType is Key

### Models

[SelectFirst<T, U>](#) is a key selection function.

## 4.9.29 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

T is [LessThanComparable](#) and U is [LessThanComparable](#) and CommonType is [LessThanComparable](#)

### 4.9.30 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 relations.

#### 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);
}
```

### 4.9.31 **Movable<T>** Concept

A move constructible and move assignable type is movable.

#### Syntax

```
public concept Movable<T>;
```

#### Constraints

T is [MoveConstructible](#) and T is [MoveAssignable](#)

### 4.9.32 MoveAssignable<T> Concept

A type equipped with move assignment operator is called move assignable.

#### Syntax

```
public concept MoveAssignable<T>;
```

#### Constraints

```
void operator=(T&&);
```

### 4.9.33 MoveConstructible<T> Concept

A type equipped with move constructor is called move constructible.

#### Syntax

```
public concept MoveConstructible<T>;
```

#### Constraints

```
T(T&&);
```

### 4.9.34 MultiplicativeGroup<T> Concept

A multiplicative monoid with a division operation is called a multiplicative group.

#### Syntax

```
public concept MultiplicativeGroup<T>;
```

#### Refines

[System.Concepts.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);
}
```

### 4.9.35 MultiplicativeMonoid<T> Concept

A multiplicative semigroup with an identity element is called a multiplicative monoid.

#### Syntax

```
public concept MultiplicativeMonoid<T>;
```

#### Refines

[System.Concepts.MultiplicativeSemigroup<T>](#)

#### Constraints

[ConversionFromSByte<T>](#) or [ConversionFromByte<T>](#)

#### Axioms

```
oneIsIdentityElement(T a)
{
    a * 1 == a && 1 * a == a;
}
```

### 4.9.36 MultiplicativeSemigroup<T> Concept

A set with an associative multiplication operation is called a multiplicative semigroup.

#### Syntax

```
public concept MultiplicativeSemigroup<T>;
```

#### Constraints

T is [Regular](#)

T operator\*(T, T);

#### Axioms

```
multiplicationIsAssociative(T a, T b, T c)
{
    (a * b) * c == a * (b * c);
}
```

### 4.9.37 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

[System.Concepts.OrderedAdditiveMonoid<T>](#)

#### Constraints

T is [AdditiveGroup](#)

## 4.9.38 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

[System.Concepts.OrderedAdditiveSemigroup<T>](#)

### Constraints

T is [AdditiveMonoid](#)

### 4.9.39 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

[System.Concepts.AdditiveSemigroup<T>](#)

#### Constraints

T is [TotallyOrdered](#)

#### Axioms

```
additionPreservesOrder(T a, T b, T c)
{
    a < b => a + c < b + c;
}
```

## 4.9.40 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

[System.Concepts.MultiplicativeSemigroup<T>](#)

### Constraints

T is [TotallyOrdered](#)

#### 4.9.41 `OutputIterator<T>` Concept

A writable iterator that is incrementable is an output iterator.

##### Syntax

```
public concept OutputIterator<T>;
```

##### Refines

[System.Concepts.TrivialIterator<T>](#)

##### Constraints

```
T& operator++();
```

## 4.9.42 RandomAccessContainer<T> Concept

A container whose iterators are random access iterators is a random access container.

### Syntax

```
public concept RandomAccessContainer<T>;
```

### Refines

[System.Concepts.BidirectionalContainer<T>](#)

### Constraints

T.Iterator is [RandomAccessIterator](#) and T.ConstIterator is [RandomAccessIterator](#)

### Models

[List<T>](#) is a random access container.

### 4.9.43 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

[System.Concepts.BidirectionalIterator<T>](#)

#### Constraints

T.ReferenceType operator[](int index);

T operator+(T, int);

T operator+(int, T);

T operator-(T, int);

int operator-(T, T);

T is [LessThanComparable](#)

#### Models

[RandomAccessIter<T, R, P>](#) is a random access iterator.

## 4.9.44 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

[System.Concepts.Semiregular<T>](#)

### Constraints

T is [EqualityComparable](#)

### Models

All built-in types (**int**, **double**, **bool**, **char**, etc...) are regular.

All types in the [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.

#### 4.9.44.1 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
}
```

#### 4.9.45 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

[System.Concepts.BinaryPredicate<T>](#)

##### Constraints

T.FirstArgumentType is U and T.SecondArgumentType is V

## 4.9.46 Relation<T> Concept

A binary predicate whose argument types are same is called a relation.

### Syntax

```
public concept Relation<T>;
```

### Refines

[System.Concepts.BinaryPredicate<T>](#)

### Constraints

typename T.Domain;

[Same<T.Domain, T.FirstArgumentType>](#) and [Same<T.SecondArgumentType, T.Domain>](#)

## 4.9.47 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

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 [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.

#### 4.9.47.1 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
}

```

## 4.9.48 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

[System.Concepts.AdditiveMonoid<T>](#)

### Constraints

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;
}
```

#### 4.9.49 SignedInteger<I> Concept

An integer type with a conversion from `sbyte` is called a signed integer.

##### Syntax

```
public concept SignedInteger<I>;
```

##### Constraints

I is [Integer](#)

I(sbyte);

##### Models

`sbyte`, `short`, `int` and `long` are signed integer types.

## 4.9.50 `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

T is [TotallyOrdered](#) and U is [TotallyOrdered](#) and CommonType is [TotallyOrdered](#)

### 4.9.51 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

[System.Concepts.Regular<T>](#)

#### Constraints

T is [LessThanComparable](#)

#### Models

Arithmetic and character types (**int**, **double**, **char**, etc...) are totally ordered. [List<T>](#), [LinkedList<T>](#), [Set<T, C>](#) and [Map<Key, Value, KeyCompare>](#) are totally ordered if their value type is totally ordered.

#### Remarks

If a type implements the < operator, the compiler implements the >, <= and >= operators automatically.

#### 4.9.51.1 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
}

```

}

## 4.9.52 TrivialIterator<T> Concept

A trivial iterator concept collects together requirements common to all iterator types.

### Syntax

```
public concept TrivialIterator<T>;
```

### Constraints

T is [Semiregular](#)

typename T.ValueType;

T.ValueType is [Semiregular](#)

typename T.ReferenceType;

T.ReferenceType is ValueType&

T.ReferenceType operator\*();

typename T.PointerType;

T.PointerType is ValueType\*

T.PointerType operator->();

### 4.9.53 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

T is [Semiregular](#)

typename T.ArgumentType;

typename T.ResultType;

T.ArgumentType is [Semiregular](#)

T.ResultType operator()(T.ArgumentType);

#### Models

[Negate<T>](#) and [Identity<T>](#) are unary functions.

### 4.9.54 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

[System.Concepts.UnaryFunction<T>](#)

#### Constraints

T.ResultType is T.ArgumentType

#### Models

[Negate<T>](#) and [Identity<T>](#) are unary operations.

### 4.9.55 UnaryPredicate<T> Concept

A unary function whose result type is **bool** is called a unary predicate.

#### Syntax

```
public concept UnaryPredicate<T>;
```

#### Refines

[System.Concepts.UnaryFunction<T>](#)

#### Constraints

T.ResultType is bool

### 4.9.56 UnsignedInteger<U> Concept

An integer type with a conversion from **byte** is called an unsigned integer.

#### Syntax

```
public concept UnsignedInteger<U>;
```

#### Constraints

U is [Integer](#)

U(byte);

#### Models

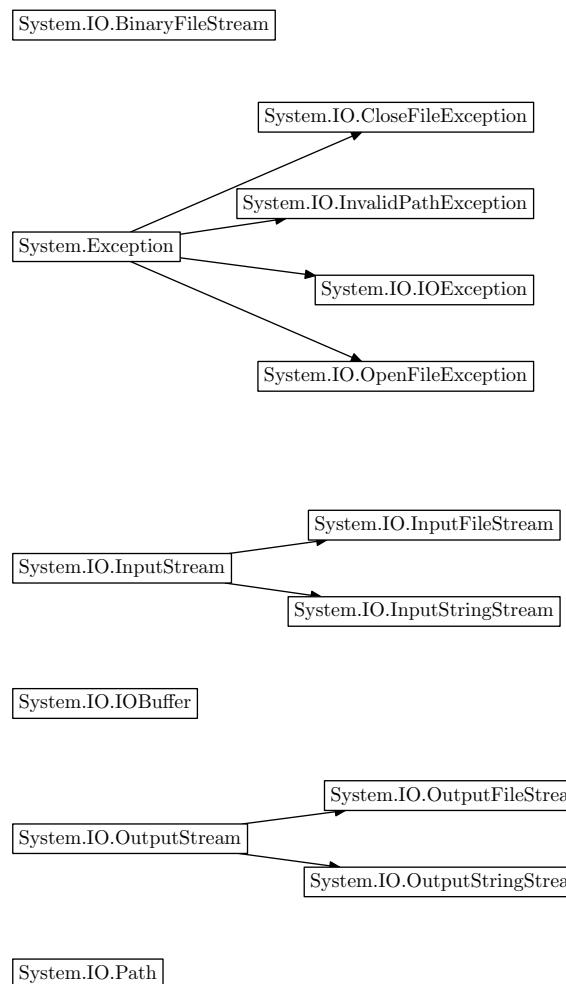
**byte**, **ushort**, **uint** and **ulong** are unsigned integer types.

# 5 System.IO Namespace

Contains classes and functions for doing input and output.

Figure 5.1 contains the classes in this namespace.

Figure 5.1: Class Diagram: I/O Classes



## 5.10 Classes

<b>Class</b>	<b>Description</b>
<a href="#">BinaryFileStream</a>	A stream of bytes connected to a file.
<a href="#">ByteStream</a>	Abstract base class for byte stream classes. Represents a stream of bytes. Byte stream can be read from or written to.
<a href="#">CloseFileException</a>	An exception thrown when closing a file fails.
<a href="#">FileByteStream</a>	Represents a stream of bytes connected to a file. The stream can be read from or written to but not both.
<a href="#">FileMapping</a>	Represents memory mapped input file.
<a href="#">FileMappingFailure</a>	Exception thrown when opening a memory mapped file fails.
<a href="#">IOBuffer</a>	A handle to dynamically allocated memory.
<a href="#">IOException</a>	An exception thrown when an I/O operation fails.
<a href="#">InputFileStream</a>	A stream of characters connected to an input file.
<a href="#">InputStream</a>	An abstract base class for input stream classes.
<a href="#">InputStreamString</a>	A class for reading from a string.
<a href="#">InvalidPathException</a>	An exception class that is thrown if a path contains too many .. components.
<a href="#">MemoryByteStream</a>	Represents a stream of bytes in memory. Writing to the stream writes data to the end of the stream. Reading from the stream reads data starting from the beginning of the stream.
<a href="#">OpenFileException</a>	An exception class thrown if opening a file fails.
<a href="#">OutputFileStream</a>	A stream of characters connected to an output file.
<a href="#">OutputStream</a>	An abstract base class for output stream classes.
<a href="#">OutputStringStream</a>	A class for writing to a string.
<a href="#">Path</a>	A static class for manipulating paths.

### 5.10.1 BinaryFileStream Class

A stream of bytes connected to a file.

#### Syntax

```
public class BinaryFileStream;
```

#### 5.10.1.1 Member Functions

Member Function	Description
<code>BinaryFileStream(System.IO.- BinaryFileStream&amp;&amp;) operator=(System.IO.BinaryFileStream&amp;&amp;)</code>	Move constructor. Move constructor.
<code>BinaryFileStream(const System.IO.OpenMode)</code>	<code>System.String&amp;,</code> Opens a binary file stream with the specified open mode and file name.
<code>BinaryFileStream(const System.IO.OpenMode, int)</code>	<code>System.String&amp;,</code> Opens a binary file stream with the specified open mode, file name and permission mode.
<code>Close()</code>	If the binary file stream is connected to an open file, closes the file, otherwise throws <code>CloseFileException</code> .
<code>GetFileSize()</code>	Returns the size of the file binary file stream is connected to.
<code>Open(const System.String&amp;, System.IO.- OpenMode, int)</code>	Opens a file with the specified open mode, file name and permission mode and connects it with the binary file stream.
<code>Read(void*, ulong)</code>	Reads <code>size</code> bytes from binary file stream into <code>buffer</code> and returns the number of bytes read.
<code>ReadBool()</code>	Reads a Boolean value from the binary file stream and returns it.
<code>.ReadByte()</code>	Reads a <code>byte</code> from the binary file stream and returns it.
<code>ReadChar()</code>	Reads a character from the binary file stream and returns it.
<code>ReadDouble()</code>	Reads a <code>double</code> from the binary file stream and returns it.
<code>ReadFloat()</code>	Reads a <code>float</code> from the binary file stream and returns it.

<a href="#">ReadInt()</a>	Reads an <b>int</b> from the binary file stream and returns it.
<a href="#">ReadLong()</a>	Reads an <b>long</b> from the binary file stream and returns it.
<a href="#">ReadSByte()</a>	Reads an <b>sbyte</b> from the binary file stream and returns it.
<a href="#">ReadShort()</a>	Reads a <b>short</b> from the binary file stream and returns it.
<a href="#">ReadSize(void*, ulong)</a>	Reads exactly <b>size</b> bytes from the binary file stream into <b>buffer</b> . If the number of bytes read is not equal to <b>size</b> throws an <a href="#">IOException</a> .
<a href="#">ReadString()</a>	Reads the length of a string (an <b>int</b> ) followed by the contents of the string from the binary file stream and returns the string.
<a href="#">ReadUInt()</a>	Reads a <b>uint</b> from the binary file stream and returns it.
<a href="#">ReadULong()</a>	Reads a <b>ulong</b> from the binary file stream and returns it.
<a href="#">ReadUShort()</a>	Reads a <b>ushort</b> from the binary file stream and returns it.
<a href="#">Seek(long, int)</a>	Sets the current file position.
<a href="#">Tell()</a>	Returns the current file position.
<a href="#">Write(bool)</a>	Writes a Boolean value to the binary file stream.
<a href="#">Write(byte)</a>	Writes a <b>byte</b> to the binary file stream.
<a href="#">Write(char)</a>	Writes a character to the binary file stream.
<a href="#">Write(const System.String&amp;)</a>	Writes a length (an <b>int</b> ) of the given string followed by the contents of the string to the binary file stream.
<a href="#">Write(const char*)</a>	Writes a length (an <b>int</b> ) of the given C-style string followed by the contents of the C-style string to the binary file stream.

<a href="#">Write(double)</a>	Writes a <b>double</b> to the binary file stream.
<a href="#">Write(float)</a>	Writes a <b>float</b> to the binary file stream.
<a href="#">Write(int)</a>	Writes an <b>int</b> to the binary file stream.
<a href="#">Write(long)</a>	Writes a <b>long</b> to the binary file stream.
<a href="#">Write(sbyte)</a>	Writes an <b>sbyte</b> to the binary file stream.
<a href="#">Write(short)</a>	Writes a <b>short</b> to the binary file stream.
<a href="#">Write(uint)</a>	Writes a <b>uint</b> to the binary file stream.
<a href="#">Write(ulong)</a>	Writes a <b>ulong</b> to the binary file stream.
<a href="#">Write(ushort)</a>	Writes a <b>ushort</b> to the binary file stream.
<a href="#">Write(void*, ulong)</a>	Writes <b>size</b> bytes from <b>buffer</b> to the binary file stream. If the number of bytes written is not equal to <b>size</b> throws an <a href="#">IOException</a> .
<a href="#">~BinaryFileStream()</a>	If the binary file stream is connected to an open file, closes the file.

**BinaryFileStream(System.IO.BinaryFileStream&&)** Member Function

Move constructor.

**Syntax**

```
public BinaryFileStream(System.IO.BinaryFileStream&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.BinaryFileStream&&	A binary file stream to move from.

**operator=(System.IO.BinaryFileStream&&) Member Function**

Move constructor.

**Syntax**

```
public void operator=(System.IO.BinaryFileStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.BinaryFileStream&&	A binary file stream to move from.

**BinaryFileStream(const System.String&, System.IO.OpenMode) Member Function**

Opens a binary file stream with the specified open mode and file name.

**Syntax**

```
public BinaryFileStream(const System.String& fileName_, System.IO.OpenMode mode_);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	The name of file to open.
mode_	System.IO.OpenMode	An open mode.

**BinaryFileStream(const System.String&, System.IO.OpenMode, int) Member Function**

Opens a binary file stream with the specified open mode, file name and permission mode.

**Syntax**

```
public BinaryFileStream(const System.String& fileName_, System.IO.OpenMode mode_,  
int pmode);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	The name of file to open.
mode_	System.IO.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).

**Close() Member Function**

If the binary file stream is connected to an open file, closes the file, otherwise throws [CloseFileException](#).

**Syntax**

```
public void Close();
```

**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.

**Open(const System.String&, System.IO.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 System.String& fileName_, System.IO.OpenMode mode_, int pmode);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	The name of the file to open.
mode_	System.IO.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.

**Read(void\*, ulong) Member Function**

Reads `size` bytes from binary file stream into `buffer` and returns the number of bytes read.

**Syntax**

```
public int Read(void* buffer, ulong size);
```

**Parameters**

Name	Type	Description
buffer	void*	A buffer to read to.
size	ulong	Number of bytes to read.

**Returns**

int

Returns the number of bytes read.

**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.

**ReadByte() Member Function**

Reads a **byte** from the binary file stream and returns it.

**Syntax**

```
public byte ReadByte();
```

**Returns**

**byte**

Returns the read **byte**.

**ReadChar() Member Function**

Reads a character from the binary file stream and returns it.

**Syntax**

```
public char ReadChar();
```

**Returns**

char

Returns the read character.

**ReadDouble() Member Function**

Reads a **double** from the binary file stream and returns it.

**Syntax**

```
public double ReadDouble();
```

**Returns**

**double**

Returns the read **double**.

**ReadFloat() Member Function**

Reads a **float** from the binary file stream and returns it.

**Syntax**

```
public float ReadFloat();
```

**Returns**

**float**

Returns the read **float**.

**ReadInt() Member Function**

Reads an **int** from the binary file stream and returns it.

**Syntax**

```
public int ReadInt();
```

**Returns**

**int**

Returns the read **int**.

**ReadLong() Member Function**

Reads an **long** from the binary file stream and returns it.

**Syntax**

```
public long ReadLong();
```

**Returns**

**long**

Returns the read **long**.

**ReadSByte() Member Function**

Reads an **sbyte** from the binary file stream and returns it.

**Syntax**

```
public sbyte ReadSByte();
```

**Returns**

**sbyte**

Returns the read **sbyte**.

**ReadShort() Member Function**

Reads a **short** from the binary file stream and returns it.

**Syntax**

```
public short ReadShort();
```

**Returns**

**short**

Returns the read **short**.

**ReadSize(void\*, ulong) Member Function**

Reads exactly `size` bytes from the binary file stream into `buffer`. If the number of bytes read is not equal to `size` throws an `IOException`.

**Syntax**

```
public void ReadSize(void* buffer, ulong size);
```

**Parameters**

Name	Type	Description
buffer	void*	A buffer to read to.
size	ulong	Number of bytes to read.

**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 System.String ReadString();
```

**Returns**

[System.String](#)

Returns the read string.

**ReadUInt() Member Function**

Reads a **uint** from the binary file stream and returns it.

**Syntax**

```
public uint ReadUInt();
```

**Returns**

**uint**

Returns the read **uint**.

**ReadULong() Member Function**

Reads a **ulong** from the binary file stream and returns it.

**Syntax**

```
public ulong ReadULong();
```

**Returns**

**ulong**

Returns the read **ulong**.

**ReadUShort() Member Function**

Reads a **ushort** from the binary file stream and returns it.

**Syntax**

```
public ushort ReadUShort();
```

**Returns**

**ushort**

Returns the read **ushort**.

**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.

**Tell() Member Function**

Returns the current file position.

**Syntax**

```
public long Tell();
```

**Returns**

long

Returns the current file position.

**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.

**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 <b>byte</b> to write.

**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.

**Write(const System.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 System.String& s);
```

**Parameters**

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

**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.

**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 <b>double</b> to write.

**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 <b>float</b> to write.

**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 <b>int</b> to write.

**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.

**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 <b>sbyte</b> to write.

**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 <b>short</b> to write.

**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 <b>uint</b> to write.

**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 <b>ulong</b> to write.

**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 <b>ushort</b> to write.

**Write(void\*, ulong) Member Function**

Writes `size` bytes from `buffer` to the binary file stream. If the number of bytes written is not equal to `size` throws an `IOException`.

**Syntax**

```
public void Write(void* buffer, ulong size);
```

**Parameters**

Name	Type	Description
buffer	void*	A buffer to write from.
size	ulong	Number of bytes to write.

**~BinaryFileStream() Member Function**

If the binary file stream is connected to an open file, closes the file.

**Syntax**

```
public ~BinaryFileStream();
```

## 5.10.2 ByteStream Class

Abstract base class for byte stream classes. Represents a stream of bytes. Byte stream can be read from or written to.

### Syntax

```
public abstract class ByteStream;
```

#### 5.10.2.1 Member Functions

Member Function	Description
<code>ByteStream()</code>	Default constructor.
<code>ByteStream(System.IO.ByteStream&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.ByteStream&amp;&amp;)</code>	Move assignment.
<code>CopyTo(System.IO.ByteStream&amp;)</code>	Copies this byte stream to another. Uses default buffer size 16K for copying.
<code>CopyTo(System.IO.ByteStream&amp;, int)</code>	Copies this byte stream to another using given buffer size.
<code>Read(byte*, int)</code>	Abstract member function that derived byte stream class overrides. Reads at most given number of bytes from the byte stream to the given buffer. Returns number of bytes read that might be less than the number of bytes requested.
<code>.ReadByte()</code>	Abstract member function that derived byte stream class overrides. Reads one byte from the byte stream. Returns the byte read or -1 if end of stream is encountered.
<code>Write(byte)</code>	Abstract member function that derived byte stream class overrides. Writes one byte of data to the byte stream.
<code>Write(byte*, int)</code>	Abstract member function that derived byte stream class overrides. Writes given number of bytes from the given buffer to the byte stream.

**ByteStream() Member Function**

Default constructor.

**Syntax**

```
public ByteStream();
```

**ByteStream(System.IO.ByteStream&&) Member Function**

Move constructor.

**Syntax**

```
public ByteStream(System.IO.ByteStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.ByteStream&&	

**operator=(System.IO.ByteStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.ByteStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.ByteStream&&	

**CopyTo(System.IO.ByteStream&)** Member Function

Copies this byte stream to another. Uses default buffer size 16K for copying.

**Syntax**

```
public void CopyTo(System.IO.ByteStream& destination);
```

**Parameters**

Name	Type	Description
destination	System.IO.ByteStream&	A byte stream to copy to.

**CopyTo(System.IO.ByteStream&, int) Member Function**

Copies this byte stream to another using given buffer size.

**Syntax**

```
public void CopyTo(System.IO.ByteStream& destination, int bufferSize);
```

**Parameters**

Name	Type	Description
destination	System.IO.ByteStream&	A byte stream to copy to.
bufferSize	int	Buffer size.

**Read(byte\*, int) Member Function**

Abstract member function that derived byte stream class overrides. Reads at most given number of bytes from the byte stream to the given buffer. Returns number of bytes read that might be less than the number of bytes requested.

**Syntax**

```
public abstract int Read(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer to read data to.
count	int	Maximum number of bytes to read.

**Returns**

int

Returns number of bytes read. Might be less than the number of bytes requested but always non-negative. Return value 0 indicates end of stream.

**Remarks**

Throws an exception if reading fails.

**ReadByte() Member Function**

Abstract member function that derived byte stream class overrides. Reads one byte from the byte stream. Returns the byte read or -1 if end of stream is encountered.

**Syntax**

```
public abstract int ReadByte();
```

**Returns**

int

Returns the byte read or -1 if end of stream is encountered.

**Remarks**

Throws an exception if reading fails.

**Write(byte) Member Function**

Abstract member function that derived byte stream class overrides. Writes one byte of data to the byte stream.

**Syntax**

```
public abstract void Write(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to write.

**Remarks**

Throws an exception if writing fails.

**Write(byte\*, int) Member Function**

Abstract member function that derived byte stream class overrides. Writes given number of bytes from the given buffer to the byte stream.

**Syntax**

```
public abstract void Write(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer of data to write.
count	int	Number of bytes to write.

### 5.10.3 CloseFileException Class

An exception thrown when closing a file fails.

#### Syntax

```
public class CloseFileException;
```

#### Base Class

[System.Exception](#)

#### 5.10.3.1 Member Functions

Member Function	Description
<code>CloseFileException()</code>	Default constructor.
<code>CloseFileException(const CloseFileException&amp;)</code>	<code>System.IO.-</code> Copy constructor.
<code>operator=(const CloseFileException&amp;)</code>	<code>System.IO.-</code> Copy assignment.
<code>CloseFileException(System.IO.- CloseFileException&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.CloseFileException&amp;&amp;)</code>	Move assignment.
<code>CloseFileException(const System.String&amp;)</code>	Constructor. Initializes the close file exception with the given error message.
<code>~CloseFileException()</code>	Destructor.

**CloseFileException() Member Function**

Default constructor.

**Syntax**

```
public CloseFileException();
```

**CloseFileException(const System.IO.CloseFileException&) Member Function**

Copy constructor.

**Syntax**

```
public CloseFileException(const System.IO.CloseFileException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.CloseFileException&	Argument to copy.

**operator=(const System.IO.CloseFileException&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.IO.CloseFileException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.CloseFileException&	Argument to assign.

**CloseFileException(System.IO.CloseFileException&&)** Member Function

Move constructor.

**Syntax**

```
public CloseFileException(System.IO.CloseFileException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.CloseFileException&&	Argument to move from.

**operator=(System.IO.CloseFileException&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.CloseFileException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.CloseFileException&&	Argument to assign from.

**CloseFileException(const System.String&) Member Function**

Constructor. Initializes the close file exception with the given error message.

**Syntax**

```
public CloseFileException(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message.

**~CloseFileException() Member Function**

Destructor.

**Syntax**

```
public ~CloseFileException();
```

### 5.10.4 FileByteStream Class

Represents a stream of bytes connected to a file. The stream can be read from or written to but not both.

#### Syntax

```
public class FileStream;
```

#### Base Class

[System.IO.ByteStream](#)

#### 5.10.4.1 Member Functions

Member Function	Description
<a href="#">FileStream(System.IO.FileByteStream&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.IO.FileByteStream&amp;&amp;)</a>	Move assignment.
<a href="#">FileStream(const System.String&amp;, System.IO.FileMode)</a>	Constructor. Initializes the byte stream with the given file name and open mode.
<a href="#">Read(byte*, int)</a>	Reads at most given number of bytes from the file to the given buffer. Returns number of bytes
<a href="#">.ReadByte()</a>	Reads a byte from the file. Returns the byte read, or -1 if end of file is encountered.
<a href="#">Write(byte)</a>	Writes a byte of data to the file.
<a href="#">Write(byte*, int)</a>	Writes given number of bytes from the given buffer to the file.
<a href="#">~FileStream()</a>	Destructor. Closes the file.

**FileByteStream(System.IO.FileByteStream&&)** Member Function

Move constructor.

**Syntax**

```
public FileByteStream(System.IO.FileByteStream&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.FileByteStream&&	A file byte stream to move from.

**operator=(System.IO.FileByteStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.FileByteStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.FileByteStream&&	A file byte stream to assign from.

**FileByteStream(const System.String&, System.IO.FileMode) Member Function**

Constructor. Initializes the byte stream with the given file name and open mode.

**Syntax**

```
public FileByteStream(const System.String& fileName_, System.IO.FileMode mode);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	Name of the file to open or create.
mode	System.IO.FileMode	Open mode.

**Read(byte\*, int) Member Function**

Reads at most given number of bytes from the file to the given buffer. Returns number of bytes

**Syntax**

```
public int Read(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer to read to.
count	int	Maximum number of bytes to read.

**Returns**

int

Returns the number of bytes read. Return value of 0 indicates end of file.

**Remarks**

Throws [IOException](#) if reading fails.

**ReadByte() Member Function**

Reads a byte from the file. Returns the byte read, or -1 if end of file is encountered.

**Syntax**

```
public int ReadByte();
```

**Returns**

int

Returns the byte read, or -1 if end of file is encountered.

**Remarks**

Throws [IOException](#) if reading fails.

**Write(byte) Member Function**

Writes a byte of data to the file.

**Syntax**

```
public void Write(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to write.

**Remarks**

Throws [IOException](#) if writing fails.

**Write(byte\*, int) Member Function**

Writes given number of bytes from the given buffer to the file.

**Syntax**

```
public void Write(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer of data to write.
count	int	The number of bytes to write.

**Remarks**

Throws [IOException](#) if writing fails.

**~FileByteStream() Member Function**

Destructor. Closes the file.

**Syntax**

```
public ~FileByteStream();
```

## 5.10.5 FileMapping Class

Represents memory mapped input file.

### Syntax

```
public class FileMapping;
```

#### 5.10.5.1 Member Functions

Member Function	Description
<code>FileMapping(System.IO.FileMapping&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.FileMapping&amp;&amp;)</code>	Move assignment.
<code>Begin() const</code>	Returns a pointer to the beginning of the file's data in memory.
<code>End() const</code>	Returns a pointer one past the end of the file's data in memory.
<code>FileMapping(const System.String&amp;)</code>	Constructor. Opens the file with the given name and maps its content to memory.
<code>FilePath() const</code>	Returns the name of the file.
<code>~FileMapping()</code>	Destructor. Closes the file and releases memory.

**FileMapping(System.IO.FileMapping&&) Member Function**

Move construcor.

**Syntax**

```
public FileMapping(System.IO.FileMapping&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.FileMapping&&	A file mapping to move from.

**operator=(System.IO.FileMapping&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.FileMapping&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.FileMapping&&	A file mapping to assign.

**Begin() const Member Function**

Returns a pointer to the beginning of the file's data in memory.

**Syntax**

```
public const char* Begin() const;
```

**Returns**

const char\*

Returns a pointer to the beginning of the file's data in memory.

**End() const Member Function**

Returns a pointer one past the end of the file's data in memory.

**Syntax**

```
public const char* End() const;
```

**Returns**

const char\*

Returns a pointer one past the end of the file's data in memory.

**FileMapping(const System.String&) Member Function**

Constructor. Opens the file with the given name and maps its content to memory.

**Syntax**

```
public FileMapping(const System.String& filePath_);
```

**Parameters**

Name	Type	Description
filePath_	const System.String&	The name of the file to open.

**Remarks**

Throws [FileMappingFailure](#) if opening fails.

**FilePath() const Member Function**

Returns the name of the file.

**Syntax**

```
public const System.String& FilePath() const;
```

**Returns**

`const System.String&`

Returns the name of the file.

**~FileMapping() Member Function**

Destructor. Closes the file and releases memory.

**Syntax**

```
public ~FileMapping();
```

## 5.10.6 FileMappingFailure Class

Exception thrown when opening a memory mapped file fails.

### Syntax

```
public class FileMappingFailure;
```

### Base Class

[System.Exception](#)

### 5.10.6.1 Member Functions

Member Function	Description
<code>FileMappingFailure()</code>	Default constructor.
<code>FileMappingFailure(const System.IO.FileMappingFailure&amp;)</code>	Copy constructor.
<code>operator=(const System.IO.FileMappingFailure&amp;)</code>	Copy assignment.
<code>FileMappingFailure(System.IO.FileMappingFailure&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.FileMappingFailure&amp;&amp;)</code>	Move assignment.
<code>FileMappingFailure(const System.String&amp;)</code>	Constructor. Initializes the file mapping failure class with the given error message.

**FileMappingFailure() Member Function**

Default constructor.

**Syntax**

```
public FileMappingFailure();
```

**FileMappingFailure(const System.IO.FileMappingFailure&) Member Function**

Copy constructor.

**Syntax**

```
public FileMappingFailure(const System.IO.FileMappingFailure& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.FileMappingFailure&	Argument to copy.

**operator=(const System.IO.FileMappingFailure&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.IO.FileMappingFailure& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.FileMappingFailure&	Argument to assign.

**FileMappingFailure(System.IO.FileMappingFailure&&) Member Function**

Move constructor.

**Syntax**

```
public FileMappingFailure(System.IO.FileMappingFailure&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.FileMappingFailure&&	Argument to move from.

**operator=(System.IO.FileMappingFailure&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.FileMappingFailure&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.FileMappingFailure&&	Argument to assign from.

**FileMappingFailure(const System.String&) Member Function**

Constructor. Initializes the file mapping failure class with the given error message.

**Syntax**

```
public FileMappingFailure(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message string.

## 5.10.7 IOBuffer Class

A handle to dynamically allocated memory.

### Syntax

```
public class IOBuffer;
```

#### 5.10.7.1 Member Functions

Member Function	Description
<code>IOBuffer(System.IO.IOBuffer&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.IOBuffer&amp;&amp;)</code>	Move assignment.
<code>IOBuffer(ulong)</code>	Constructor. Creates an I/O buffer of <code>size_</code> bytes.
<code>Mem() const</code>	Returns a generic pointer to the allocated memory.
<code>Size() const</code>	Returns the size of the I/O buffer.
<code>~IOBuffer()</code>	Destructor. Releases the allocated memory back to system.

**IOBuffer(System.IO.IOBuffer&&) Member Function**

Move constructor.

**Syntax**

```
public IOBuffer(System.IO.IOBuffer&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.IO.IOBuffer&amp;&amp;</a>	An I/O buffer to move from.

**operator=(System.IO.IOBuffer&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.IOBuffer&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.IO.IOBuffer&amp;&amp;</a>	An I/O buffer to assign from.

**IOBuffer(ulong) Member Function**

Constructor. Creates an I/O buffer of `size_` bytes.

**Syntax**

```
public IOBuffer(ulong size_);
```

**Parameters**

Name	Type	Description
<code>size_</code>	ulong	Number of bytes to allocate.

**Mem() const Member Function**

Returns a generic pointer to the allocated memory.

**Syntax**

```
public void* Mem() const;
```

**Returns**

void\*

Returns a generic pointer to the allocated memory.

**Size() const Member Function**

Returns the size of the I/O buffer.

**Syntax**

```
public ulong Size() const;
```

**Returns**

ulong

Returns the size of the I/O buffer.

**~IOBuffer() Member Function**

Destructor. Releases the allocated memory back to system.

**Syntax**

```
public ~IOBuffer();
```

## 5.10.8 IOException Class

An exception thrown when an I/O operation fails.

### Syntax

```
public class IOException;
```

### Base Class

[System.Exception](#)

### 5.10.8.1 Member Functions

Member Function	Description
<code>IOException()</code>	Default constructor.
<code>IOException(const System.IO.IOException&amp;)</code>	Copy constructor.
<code>operator=(const System.IO.IOException&amp;)</code>	Copy assignment.
<code>IOException(System.IO.IOException&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.IOException&amp;&amp;)</code>	Move assignment.
<code>IOException(const System.String&amp;)</code>	Constructor. Initializes the exception with the given error message.
<code>~IOException()</code>	Destructor.

**IOException() Member Function**

Default constructor.

**Syntax**

```
public IOException();
```

**IOException(const System.IO.IOException&) Member Function**

Copy constructor.

**Syntax**

```
public IOException(const System.IO.IOException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.IOException&	Argument to copy.

**operator=(const System.IO.IOException&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.IO.IOException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.IOException&	Argument to assign.

**IOException(System.IO.IOException&&) Member Function**

Move constructor.

**Syntax**

```
public IOException(System.IO.IOException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.IOException&&	Argument to move from.

**operator=(System.IO.IOException&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.IOException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.IOException&&	Argument to assign from.

**IOException(const System.String&) Member Function**

Constructor. Initializes the exception with the given error message.

**Syntax**

```
public IOException(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message.

**~IOException() Member Function**

Destructor.

**Syntax**

```
public ~IOException();
```

### 5.10.9 InputStream Class

A stream of characters connected to an input file.

#### Syntax

```
public class InputStream;
```

#### Base Class

[System.IO.InputStream](#)

#### 5.10.9.1 Member Functions

Member Function	Description
<a href="#">InputStream()</a>	Default constructor. Initializes the input file stream and connects it to standard input stream.
<a href="#">InputStream(System.IO.InputStream&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.IO.InputStream&amp;&amp;)</a>	Move assignment.
<a href="#">Close()</a>	If the input file stream is connected to an open file, closes the file, otherwise throws <a href="#">CloseFileException</a> ;
<a href="#">EndOfStream() const</a>	Returns true if the end of the stream is encountered, false otherwise.
<a href="#">FileName() const</a>	Returns the name of the file input file stream is connected to.
<a href="#">Handle() const</a>	Returns the file handler the input file stream is connected to.
<a href="#">InputStream(const System.String&amp;)</a>	Constructor. Initializes the input file stream and connects it to a file with the given name.
<a href="#">InputStream(const System.String&amp;, uint)</a>	Constructor. Initializes the input file stream with the given buffer size and connects it to a file with the given name.
<a href="#">InputStream(int, uint)</a>	Constructor. Initializes the input file stream with the given buffer size and connects it to the file with the given file handle.
<a href="#">Open(const System.String&amp;)</a>	Opens an input file and connects it to the input file stream.

<a href="#">ReadLine()</a>	Reads a line of text from the input file stream and returns it.
<a href="#">ReadToEnd()</a>	Reads the content of the input file into a string and returns it.
<a href="#">~InputFileStream()</a>	Destructor. If the input file stream is connected to an open file, closes the file.

**InputFileStream() Member Function**

Default constructor. Initializes the input file stream and connects it to standard input stream.

**Syntax**

```
public InputFileStream();
```

**InputStream(System.IO.InputStream&&) Member Function**

Move constructor.

**Syntax**

```
public InputStream(System.IO.InputStream&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.InputStream&&	An input file stream to move from.

**operator=(System.IO.InputStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.InputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.InputStream&&	An input file stream to assign from.

**Close() Member Function**

If the input file stream is connected to an open file, closes the file, otherwise throws [CloseFileException](#);

**Syntax**

```
public void Close();
```

**EndOfStream() const Member Function**

Returns true if the end of the stream is encountered, false otherwise.

**Syntax**

```
public bool EndOfStream() const;
```

**Returns**

bool

Returns true if the end of the stream is encountered, false otherwise.

**FileName() const Member Function**

Returns the name of the file input file stream is connected to.

**Syntax**

```
public const System.String& FileName() const;
```

**Returns**

**const System.String&**

Returns the name of the file input file stream is connected to.

**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.

**InputStream(const System.String&) Member Function**

Constructor. Initializes the input file stream and connects it to a file with the given name.

**Syntax**

```
public InputStream(const System.String& fileName_);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	The name of the file to open.

**InputFileStream(const System.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 System.String& fileName_, uint bufferSize_);
```

**Parameters**

Name	Type	Description
fileName_	const System.String&	The name of the file to open.
bufferSize_	uint	The size of the input buffer.

**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 InputFileStream(int handle_, uint bufferSize_);
```

**Parameters**

Name	Type	Description
handle_	int	A file handle.
bufferSize_	uint	The size of the input buffer.

**Open(const System.String&)** Member Function

Opens an input file and connects it to the input file stream.

**Syntax**

```
public void Open(const System.String& fileName_);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**ReadLine() Member Function**

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

**Syntax**

```
public System.String ReadLine();
```

**Returns**

[System.String](#)

Returns the line of text read.

**ReadToEnd() Member Function**

Reads the content of the input file into a string and returns it.

**Syntax**

```
public System.String ReadToEnd();
```

**Returns**

[System.String](#)

Returns the contents of the file read.

**~InputFileStream() Member Function**

Destructor. If the input file stream is connected to an open file, closes the file.

**Syntax**

```
public ~InputFileStream();
```

### 5.10.10 InputStream Class

An abstract base class for input stream classes.

#### Syntax

```
public abstract class InputStream;
```

#### 5.10.10.1 Member Functions

Member Function	Description
<code>InputStream()</code>	Default constructor.
<code>InputStream(System.IO.InputStream&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.InputStream&amp;&amp;)</code>	Move assignment.
<code>EndOfFile() const</code>	Abstract member function for returning the end-of-file status.
<code>ReadLine()</code>	Abstract member function for reading a line of text from the input stream.
<code>ReadToEnd()</code>	Abstract member function for reading the contents of the stream into a string.
<code>~InputStream()</code>	Destructor.

**InputStream() Member Function**

Default constructor.

**Syntax**

```
public InputStream();
```

**InputStream(System.IO.InputStream&&)** Member Function

Move constructor.

**Syntax**

```
public InputStream(System.IO.InputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.InputStream&&	

**operator=(System.IO.InputStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.InputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.InputStream&&	

**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.

**ReadLine() Member Function**

Abstract member function for reading a line of text from the input stream.

**Syntax**

```
public abstract System.String ReadLine();
```

**Returns**

[System.String](#)

Returns the line read.

**ReadToEnd() Member Function**

Abstract member function for reading the contents of the stream into a string.

**Syntax**

```
public abstract System.String ReadToEnd();
```

**Returns**

[System.String](#)

Returns the contents of the stream.

**~InputStream() Member Function**

Destructor.

**Syntax**

```
public ~InputStream();
```

### 5.10.11 InputStringStream Class

A class for reading from a string.

#### Syntax

```
public class InputStringStream;
```

#### Base Class

[System.IO.InputStream](#)

#### 5.10.11.1 Member Functions

Member Function	Description
<code>InputStringStream()</code>	Default constructor.
<code>InputStringStream(System.IO.-&lt;br&gt;InputStringStream&amp;&amp;)-&lt;br&gt;operator=(System.IO.InputStringStream&amp;&amp;)</code>	Move constructor.
<code>EndOfStream() const</code>	Move assignment.
<code>GetStr() const</code>	Returns true if the end of the string has been encountered, false otherwise.
<code>InputStringStream(const System.String&amp;)</code>	Returns the contained string.
<code>ReadLine()</code>	Constructor. Initializes the input string stream with the given string.
<code>ReadToEnd()</code>	Reads a line of text from the string and returns it.
<code>SetStr(const System.String&amp;)</code>	Returns the contents of the rest of the string.
<code>~InputStringStream()</code>	Sets the contained string.
	Destructor.

**InputStringStream() Member Function**

Default constructor.

**Syntax**

```
public InputStringStream();
```

**InputStringStream(System.IO.InputStringStream&&) Member Function**

Move constructor.

**Syntax**

```
public InputStringStream(System.IO.InputStringStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.InputStringStream&&	

**operator=(System.IO.InputStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.InputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.InputStream&&	

**EndOfStream() const Member Function**

Returns true if the end of the string has been encountered, false otherwise.

**Syntax**

```
public bool EndOfStream() const;
```

**Returns**

bool

Returns true if the end of the string has been encountered, false otherwise.

**GetStr() const Member Function**

Returns the contained string.

**Syntax**

```
public const System.String& GetStr() const;
```

**Returns**

**const System.String&**

Returns the contained string.

**InputStringStream(const System.String&) Member Function**

Constructor. Initializes the input string stream with the given string.

**Syntax**

```
public InputStringStream(const System.String& str_);
```

**Parameters**

Name	Type	Description
str_	const System.String&	A string to read from.

**ReadLine() Member Function**

Reads a line of text from the string and returns it.

**Syntax**

```
public System.String ReadLine();
```

**Returns**

[System.String](#)

Returns the line read.

**ReadToEnd() Member Function**

Returns the contents of the rest of the string.

**Syntax**

```
public System.String ReadToEnd();
```

**Returns**

[System.String](#)

Returns the contents of the rest of the string.

**SetStr(const System.String&)** Member Function

Sets the contained string.

**Syntax**

```
public void SetStr(const System.String& str_);
```

**Parameters**

Name	Type	Description
str_	const System.String&	A string.

**~InputStringStream() Member Function**

Destructor.

**Syntax**

```
public ~InputStringStream();
```

### 5.10.12 InvalidPathException Class

An exception class that is thrown if a path contains too many .. components.

#### Syntax

```
public class InvalidPathException;
```

#### Base Class

[System.Exception](#)

#### 5.10.12.1 Member Functions

Member Function	Description
InvalidPathException()	Default constructor.
InvalidPathException(const System.IO.InvalidPathException&)	Copy constructor.
operator=(const System.IO.InvalidPathException&)	Copy assignment.
InvalidPathException(System.IO.InvalidPathException&&)	Move constructor.
operator=(System.IO.InvalidPathException&&)	Move assignment.
InvalidPathException(const System.String&)	Constructor. Initializes the exception with the given error message.
~InvalidPathException()	Destructor.

**InvalidPathException() Member Function**

Default constructor.

**Syntax**

```
public InvalidPathException();
```

**InvalidPathException(const System.IO.InvalidPathException&) Member Function**

Copy constructor.

**Syntax**

```
public InvalidPathException(const System.IO.InvalidPathException& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	const System.IO.InvalidPathException&	

**operator=(const System.IO.InvalidPathException&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.IO.InvalidPathException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.InvalidPathException&	Argument to assign.

**InvalidPathException(System.IO.InvalidPathException&&) Member Function**

Move constructor.

**Syntax**

```
public InvalidPathException(System.IO.InvalidPathException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.InvalidPathException&&	Argument to move from.

**operator=(System.IO.InvalidPathException&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.InvalidPathException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.InvalidPathException&&	Argument to assign from.

**InvalidPathException(const System.String&) Member Function**

Constructor. Initializes the exception with the given error message.

**Syntax**

```
public InvalidPathException(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message.

**~InvalidPathException() Member Function**

Destructor.

**Syntax**

```
public ~InvalidPathException();
```

### 5.10.13 MemoryStream Class

Represents a stream of bytes in memory. Writing to the stream writes data to the end of the stream. Reading from the stream reads data starting from the beginning of the stream.

#### Syntax

```
public class MemoryStream;
```

#### Base Class

[System.IO.ByteStream](#)

#### 5.10.13.1 Member Functions

Member Function	Description
<a href="#">MemoryByteStream()</a>	Default constructor. Constructs an empty memory byte stream.
<a href="#">MemoryByteStream(System.IO.-MemoryByteStream&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.IO.MemoryByteStream&amp;&amp;)</a>	Move assignment.
<a href="#">Data() const</a>	Returns a pointer to the beginning of data.
<a href="#">MemoryByteStream(byte*, int)</a>	Constructor. Initializes the memory byte stream with given data.
<a href="#">Read(byte*, int)</a>	Reads at most given number of bytes from the memory stream to the given buffer. Returns the number of bytes read. Return value of 0 indicates end of stream.
<a href="#">.ReadByte()</a>	Reads one byte of data from the stream. Returns the byte read, or -1 if end of stream is encountered.
<a href="#">ReadPos() const</a>	Returns reading position.
<a href="#">Size() const</a>	Returns the number of bytes contained in the stream.
<a href="#">Write(byte)</a>	Writes one byte of data to the end of the stream.
<a href="#">Write(byte*, int)</a>	Writes given number of bytes from the given buffer to the end of the stream.
<a href="#">~MemoryByteStream()</a>	Destructor. Releases memory occupied by the stream.



**MemoryByteStream() Member Function**

Default constructor. Constructs an empty memory byte stream.

**Syntax**

```
public MemoryByteStream();
```

**MemoryByteStream(System.IO.MemoryByteStream&&)** Member Function

Move constructor.

**Syntax**

```
public MemoryByteStream(System.IO.MemoryByteStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	<a href="#">System.IO.MemoryByteStream&amp;&amp;</a>	

**operator=(System.IO.MemoryByteStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.MemoryByteStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.MemoryByteStream&&	

**Data() const Member Function**

Returns a pointer to the beginning of data.

**Syntax**

```
public const byte* Data() const;
```

**Returns**

const byte\*

Returns a pointer to the beginning of data.

**MemoryByteStream(byte\*, int) Member Function**

Constructor. Initializes the memory byte stream with given data.

**Syntax**

```
public MemoryByteStream(byte* data, int count);
```

**Parameters**

Name	Type	Description
data	byte*	Buffer of data.
count	int	Number of bytes in data buffer.

**Read(byte\*, int) Member Function**

Reads at most given number of bytes from the memory stream to the given buffer. Returns the number of bytes read. Return value of 0 indicates end of stream.

**Syntax**

```
public int Read(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer to read to.
count	int	Maximum number of bytes to read.

**Returns**

int

Returns the number of bytes read.

**ReadByte() Member Function**

Reads one byte of data from the stream. Returns the byte read, or -1 if end of stream is encountered.

**Syntax**

```
public int ReadByte();
```

**Returns**

int

Returns the byte read, or -1 if end of stream is encountered.

**ReadPos() const Member Function**

Returns reading position.

**Syntax**

```
public int ReadPos() const;
```

**Returns**

int

Returns reading position.

**Size() const Member Function**

Returns the number of bytes contained in the stream.

**Syntax**

```
public int Size() const;
```

**Returns**

int

Returns the number of bytes contained in the stream.

**Write(byte) Member Function**

Writes one byte of data to the end of the stream.

**Syntax**

```
public void Write(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to write.

**Write(byte\*, int) Member Function**

Writes given number of bytes from the given buffer to the end of the stream.

**Syntax**

```
public void Write(byte* buf, int count);
```

**Parameters**

Name	Type	Description
buf	byte*	A buffer of data to write.
count	int	Number of bytes to write.

**~MemoryByteStream() Member Function**

Destructor. Releases memory occupied by the stream.

**Syntax**

```
public ~MemoryByteStream();
```

### 5.10.14 OpenFileException Class

An exception class thrown if opening a file fails.

#### Syntax

```
public class OpenFileException;
```

#### Base Class

[System.Exception](#)

#### 5.10.14.1 Member Functions

Member Function	Description
<code>OpenFileException()</code>	Default constructor.
<code>OpenFileException(const OpenFileException&amp;)</code>	<code>System.IO.-</code> Copy constructor.
<code>operator=(const OpenFileException&amp;)</code>	<code>System.IO.-</code> Copy assignment.
<code>OpenFileException(System.IO.- OpenFileException&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.OpenFileException&amp;&amp;)</code>	Move assignment.
<code>OpenFileException(const System.String&amp;)</code>	Constructor. Initializes the exception with the given error message.
<code>~OpenFileException()</code>	Destructor.

**OpenFileException() Member Function**

Default constructor.

**Syntax**

```
public OpenFileException();
```

**OpenFileException(const System.IO.OpenFileException&) Member Function**

Copy constructor.

**Syntax**

```
public OpenFileException(const System.IO.OpenFileException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.OpenFileException&	Argument to copy.

**operator=(const System.IO.OpenFileException&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.IO.OpenFileException& that);
```

**Parameters**

Name	Type	Description
that	const System.IO.OpenFileException&	Argument to assign.

**OpenFileException(System.IO.OpenFileException&&)** Member Function

Move constructor.

**Syntax**

```
public OpenFileException(System.IO.OpenFileException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.OpenFileException&&	Argument to move from.

**operator=(System.IO.OpenFileException&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.OpenFileException&& that);
```

**Parameters**

Name	Type	Description
that	System.IO.OpenFileException&&	Argument to assign from.

**OpenFileException(const System.String&) Member Function**

Constructor. Initializes the exception with the given error message.

**Syntax**

```
public OpenFileException(const System.String& message_);
```

**Parameters**

Name	Type	Description
message_	const System.String&	An error message.

**~OpenFileException() Member Function**

Destructor.

**Syntax**

```
public ~OpenFileException();
```

### 5.10.15 OutputStream Class

A stream of characters connected to an output file.

#### Syntax

```
public class OutputStream;
```

#### Base Class

[System.IO.OutputStream](#)

#### 5.10.15.1 Member Functions

Member Function	Description
<code>OutputStream()</code>	Default constructor. Connects the output file stream with standard output stream.
<code>OutputStream(System.IO.- OutputStream&amp;&amp;) operator=(System.IO.OutputStream&amp;&amp;)</code>	Move constructor.
<code>Close()</code>	Move assignment.
<code>Close()</code>	If the output file stream is connected to an open file, the file is closed, otherwise throws <a href="#">CloseFileException</a> .
<code>FileName() const</code>	Returns the name of the output file the output file stream is connected to.
<code>Handle() const</code>	Returns the file handle of the file the output file stream is connected to.
<code>Open(const System.String&amp;)</code>	Opens the given output file and connects it to the output file stream.
<code>Open(const System.String&amp;, bool)</code>	Opens the given output file and connects it to the output file stream.
<code>Open(const System.String&amp;, int)</code>	Opens the given output file and connects it to the output file stream using given permissions.
<code>Open(const System.String&amp;, int, bool)</code>	Opens the given output file and connects it to the output file stream using given permissions.
<code>OutputStream(const System.String&amp;)</code>	Constructor. Connects the output file stream with the output file of the given name.
<code>OutputStream(const System.String&amp;, bool)</code>	Constructor. Connects the output file stream to an output file of the given name.

<code>OutputFileStream(const System.String&amp;, int)</code>	Constructor. Connects the output file stream to an output file with the given name using given permissions.
<code>OutputFileStream(const System.String&amp;, int, bool)</code>	Constructor. Connects the output file stream to an output file with the given name using given permissions.
<code>OutputFileStream(int)</code>	Constructor. Connects the output file stream to an output file with the given file handle.
<code>Write(bool)</code>	Writes the given Boolean value to the output file stream.
<code>Write(byte)</code>	Writes the given <b>byte</b> to the output file stream.
<code>Write(char)</code>	Writes the given character to the output file stream.
<code>Write(const System.String&amp;)</code>	Writes the given string to the output file stream.
<code>Write(const char*)</code>	Writes the given C-style string to the output file stream.
<code>Write(double)</code>	Writes the given <b>double</b> value to the output file stream.
<code>Write(float)</code>	Writes the given <b>float</b> value to the output file stream.
<code>Write(int)</code>	Writes the given <b>int</b> value to the output file stream.
<code>Write(long)</code>	Writes the given <b>long</b> value to the output file stream.
<code>Write(sbyte)</code>	Writes the given <b>sbyte</b> value to the output file stream.
<code>Write(short)</code>	Writes the given <b>short</b> value to the output file stream.
<code>Write(uint)</code>	Writes the given <b>uint</b> value to the output file stream.
<code>Write(ulong)</code>	Writes the given <b>ulong</b> value to the output file stream.

<a href="#">Write(ushort)</a>	Writes the given <b>ushort</b> value to the output file stream.
<a href="#">WriteLine()</a>	Writes a new line to the output file stream.
<a href="#">WriteLine(bool)</a>	Writes the given Boolean value followed by a new line to the output file stream.
<a href="#">WriteLine(byte)</a>	Writes the given <b>byte</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(char)</a>	Writes the given character followed by a new line to the output file stream.
<a href="#">WriteLine(const System.String&amp;)</a>	Writes the given string followed by a new line to the output file stream.
<a href="#">WriteLine(const char*)</a>	Writes the given C-style string followed by a new line to the output file stream.
<a href="#">WriteLine(double)</a>	Writes the given <b>double</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(float)</a>	Writes the given <b>float</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(int)</a>	Writes the given <b>int</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(long)</a>	Writes the given <b>long</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(sbyte)</a>	Writes the given <b>sbyte</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(short)</a>	Writes the given <b>short</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(uint)</a>	Writes the given <b>uint</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(ulong)</a>	Writes the given <b>ulong</b> value followed by a new line to the output file stream.
<a href="#">WriteLine(ushort)</a>	Writes the given <b>ushort</b> value followed by a new line to the output file stream.

[`~OutputFileStream\(\)`](#)

Destructor. If the output file stream is connected to an open file, the file is closed.

**OutputFileStream() Member Function**

Default constructor. Connects the output file stream with standard output stream.

**Syntax**

```
public OutputFileStream();
```

**OutputFileStream(System.IO.OutputStream&&)** Member Function

Move constructor.

**Syntax**

```
public OutputFileStream(System.IO.OutputStream&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.IO.OutputStream&amp;&amp;</a>	An output file stream to move from.

**operator=(System.IO.OutputStream&&)** Member Function

Move assignment.

**Syntax**

```
public void operator=(System.IO.OutputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.OutputStream&&	

**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();
```

**FileName() const Member Function**

Returns the name of the output file the output file stream is connected to.

**Syntax**

```
public const System.String& FileName() const;
```

**Returns**

**const System.String&**

Returns the name of the output file.

**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.

**Open(const System.String&) Member Function**

Opens the given output file and connects it to the output file stream.

**Syntax**

```
public void Open(const System.String& fileName_);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**Open(const System.String&, bool) Member Function**

Opens the given output file and connects it to the output file stream.

**Syntax**

```
public void Open(const System.String& fileName_, bool append);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**Open(const System.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 System.String& fileName_, int pmode);
```

**Parameters**

Name	Type	Description
fileName_	const System.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).

**Open(const System.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 System.String& fileName_, int pmode, bool append);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**OutputFileStream(const System.String&) Member Function**

Constructor. Connects the output file stream with the output file of the given name.

**Syntax**

```
public OutputFileStream(const System.String& fileName_);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**OutputFileStream(const System.String&, bool) Member Function**

Constructor. Connects the output file stream to an output file of the given name.

**Syntax**

```
public OutputFileStream(const System.String& fileName_, bool append);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**OutputFileStream(const System.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 System.String& fileName_, int pmode);
```

**Parameters**

Name	Type	Description
fileName_	const System.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).

**OutputFileStream(const System.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 System.String& fileName_, int pmode, bool append);
```

**Parameters**

Name	Type	Description
fileName_	const System.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.

**OutputFileStream(int) Member Function**

Constructor. Connects the output file stream to an output file with the given file handle.

**Syntax**

```
public OutputFileStream(int handle_);
```

**Parameters**

Name	Type	Description
handle_	int	A file handle.

**Write(bool) Member Function**

Writes the given Boolean value to the output file stream.

**Syntax**

```
public 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”.

**Write(byte) Member Function**

Writes the given **byte** to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
b	byte	A <b>byte</b> to write.

**Write(char) Member Function**

Writes the given character to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
c	char	A character to write.

**Write(const System.String&)** Member Function

Writes the given string to the output file stream.

**Syntax**

```
public void Write(const System.String& s);
```

**Parameters**

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

**Write(const char\*) Member Function**

Writes the given C-style string to the output file stream.

**Syntax**

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

**Parameters**

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

**Write(double) Member Function**

Writes the given **double** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
d	double	A <b>double</b> to write.

**Write(float) Member Function**

Writes the given **float** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
f	float	A <b>float</b> to write.

**Write(int) Member Function**

Writes the given **int** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
i	int	An <b>int</b> to write.

**Write(long) Member Function**

Writes the given **long** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
l	long	A long to write.

**Write(sbyte) Member Function**

Writes the given **sbyte** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
s	sbyte	An <b>sbyte</b> to write.

**Write(short) Member Function**

Writes the given **short** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
s	short	A <b>short</b> to write.

**Write(uint) Member Function**

Writes the given **uint** value to the output file stream.

**Syntax**

```
public void Write(uint i);
```

**Parameters**

Name	Type	Description
i	uint	A <b>uint</b> to write.

**Write(ulong) Member Function**

Writes the given **ulong** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ulong	A <b>ulong</b> to write.

**Write(ushort) Member Function**

Writes the given **ushort** value to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ushort	A <b>ushort</b> to write.

**WriteLine() Member Function**

Writes a new line to the output file stream.

**Syntax**

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

**WriteLine(bool) Member Function**

Writes the given Boolean value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
b	bool	A Boolean value to write.

**WriteLine(byte) Member Function**

Writes the given **byte** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
b	byte	A <b>byte</b> to write.

**WriteLine(char) Member Function**

Writes the given character followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
c	char	A character to write.

**WriteLine(const System.String&) Member Function**

Writes the given string followed by a new line to the output file stream.

**Syntax**

```
public void WriteLine(const System.String& s);
```

**Parameters**

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

**WriteLine(const char\*) Member Function**

Writes the given C-style string followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

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

**WriteLine(double) Member Function**

Writes the given **double** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
d	double	A <b>double</b> to write.

**WriteLine(float) Member Function**

Writes the given **float** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
f	float	A <b>float</b> to write.

**WriteLine(int) Member Function**

Writes the given **int** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
i	int	An <b>int</b> to write.

**WriteLine(long) Member Function**

Writes the given **long** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
l	long	A long to write.

**WriteLine(sbyte) Member Function**

Writes the given **sbyte** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
s	sbyte	An <b>sbyte</b> to write.

**WriteLine(short) Member Function**

Writes the given **short** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
s	short	A <b>short</b> to write.

**WriteLine(uint) Member Function**

Writes the given **uint** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
u	uint	A <b>uint</b> to write.

**WriteLine(ulong) Member Function**

Writes the given **ulong** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ulong	A <b>ulong</b> to write.

**WriteLine(ushort) Member Function**

Writes the given **ushort** value followed by a new line to the output file stream.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ushort	A <b>ushort</b> to write.

**~OutputFileStream() Member Function**

Destructor. If the output file stream is connected to an open file, the file is closed.

**Syntax**

```
public ~OutputFileStream();
```

## 5.10.16 OutputStream Class

An abstract base class for output stream classes.

### Syntax

```
public abstract class OutputStream;
```

#### 5.10.16.1 Member Functions

Member Function	Description
<code>OutputStream()</code>	Default constructor.
<code>OutputStream(System.IO.OutputStream&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.OutputStream&amp;&amp;)</code>	Move assignment.
<code>Write(bool)</code>	Writes a Boolean value to output stream.
<code>Write(byte)</code>	Writes a <b>byte</b> to the output stream.
<code>Write(char)</code>	Writes a character to the output stream.
<code>Write(const System.String&amp;)</code>	Writes a string to the output stream.
<code>Write(const char*)</code>	Writes a C-style string to the output steram.
<code>Write(double)</code>	Writes a <b>double</b> to the output stream.
<code>Write(float)</code>	Writes a <b>float</b> to the output stream.
<code>Write(int)</code>	Writes an <b>int</b> to the output stream.
<code>Write(long)</code>	Writes a <b>long</b> to the output stream.
<code>Write(sbyte)</code>	Writes an <b>sbyte</b> to the output stream.
<code>Write(short)</code>	Writes a <b>short</b> to the output stream.
<code>Write(uint)</code>	Writes a <b>uint</b> to the output stream.
<code>Write(ulong)</code>	Writes a <b>ulong</b> to the output stream.
<code>Write(ushort)</code>	Writes a <b>ushort</b> to the output stream.
<code>WriteLine()</code>	Writes a new line to the output stream.
<code>WriteLine(bool)</code>	Writes a Boolean followed by a new line to the output stream.

<code>WriteLine(byte)</code>	Writes a <b>byte</b> followed by a new line to the output stream.
<code>WriteLine(char)</code>	Writes a character followed by a new line to the output stream.
<code>WriteLine(const System.String&amp;)</code>	Writes a string followed by a new line to the output stream.
<code>WriteLine(const char*)</code>	Writes a C-style string followed by a new line to the output stream.
<code>WriteLine(double)</code>	Writes a <b>double</b> followed by a new line to the output stream.
<code>WriteLine(float)</code>	Writes a <b>float</b> followed by a new line to the output stream.
<code>WriteLine(int)</code>	Writes an <b>int</b> followed by a new line to the output stream.
<code>WriteLine(long)</code>	Writes a <b>long</b> followed by a new line to the output stream.
<code>WriteLine(sbyte)</code>	Writes an <b>sbyte</b> followed by a new line to the output stream.
<code>WriteLine(short)</code>	Writes a <b>short</b> followed by a new line to the output stream.
<code>WriteLine(uint)</code>	Writes a <b>uint</b> followed by a new line to the output stream.
<code>WriteLine(ulong)</code>	Writes a <b>ulong</b> followed by a new line to the output stream.
<code>WriteLine(ushort)</code>	Writes a <b>ushort</b> followed by a new line to the output stream.
<code>~OutputStream()</code>	Destructor.

**OutputStream() Member Function**

Default constructor.

**Syntax**

```
public OutputStream();
```

**OutputStream(System.IO.OutputStream&&) Member Function**

Move constructor.

**Syntax**

```
public OutputStream(System.IO.OutputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.OutputStream&&	

**operator=(System.IO.OutputStream&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.IO.OutputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.OutputStream&&	

**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.

**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 <b>byte</b> to write.

**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.

**Write(const System.String&)** Member Function

Writes a string to the output stream.

**Syntax**

```
public abstract void Write(const System.String& s);
```

**Parameters**

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

**Write(const char\*) Member Function**

Writes a C-style string to the output stream.

**Syntax**

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

**Parameters**

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

**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 <b>double</b> to write.

**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 <b>float</b> to write.

**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 <b>int</b> to write.

**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.

**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 <b>sbyte</b> to write.

**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 <b>short</b> to write.

**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 <b>uint</b> to write.

**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 <b>ulong</b> to write.

**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 <b>ushort</b> to write.

**WriteLine() Member Function**

Writes a new line to the output stream.

**Syntax**

```
public abstract void WriteLine();
```

**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.

**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.

**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.

**WriteLine(const System.String&)** Member Function

Writes a string followed by a new line to the output stream.

**Syntax**

```
public abstract void WriteLine(const System.String& s);
```

**Parameters**

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

**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.

**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 <b>double</b> to write.

**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 <b>float</b> to write.

**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 <b>int</b> to write.

**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.

**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 <b>sbyte</b> to write.

**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 <b>short</b> to write.

**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 <b>uint</b> to write.

**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 <b>ulong</b> to write.

**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 <b>ushort</b> to write.

**~OutputStream() Member Function**

Destructor.

**Syntax**

```
public ~OutputStream();
```

**5.10.16.2 Nonmember Functions**

<b>Function</b>	<b>Description</b>
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, System.Date)</code>	Writes the given date to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, System.EndLine)</code>	Writes end line character to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, bool)</code>	Writes the given Boolean value to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, byte)</code>	Writes the given byte to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, char)</code>	Writes the given character to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, const C&amp;)</code>	Writes the given forward container containing integers to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, const System.String&amp;)</code>	Writes the given string to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, const char*)</code>	Writes the given C-style string to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, double)</code>	Writes the given double to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, float)</code>	Writes the given float to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, int)</code>	Writes the given int to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, long)</code>	Writes the given long to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, sbyte)</code>	Writes the given sbyte to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, short)</code>	Writes the given short to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, uint)</code>	Writes the given uint to the given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, ulong)</code>	Writes the given ulong to given output stream.
<code>operator&lt;&lt;(System.IO.OutputStream&amp;, ushort)</code>	Writes the given ushort to the given output stream.

**operator<<(System.IO.OutputStream&, System.Date) Function**

Writes the given date to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, System.Date date);
```

**Parameters**

Name	Type	Description
s	<a href="#">System.IO.OutputStream&amp;</a>	An output stream.
date	<a href="#">System.Date</a>	A date to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, System.EndLine) Function**

Writes end line character to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, System.EndLine  
__parameter1);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
__parameter1	System.EndLine	

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, bool) Function**

Writes the given Boolean value to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, bool b);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
b	bool	A Boolean to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, byte) Function**

Writes the given byte to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, byte b);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
b	byte	A byte to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, char) Function**

Writes the given character to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, char c);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
c	char	A character to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, const C&) Function**

Writes the given forward container containing integers to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, const C& c);
```

**Constraint**

C is [ForwardContainer](#) and C.ValueType is int

**Parameters**

Name	Type	Description
s	<a href="#">System.IO.OutputStream&amp;</a>	An output stream.
c	const C&	A forward container.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, const System.String&) Function**

Writes the given string to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, const System.String& str);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
str	const System.String&	A string to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, const char\*) Function**

Writes the given C-style string to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, const char* str);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
str	const char*	A C-style string.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, double) Function**

Writes the given double to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, double d);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
d	double	A double to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, float) Function**

Writes the given float to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, float f);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
f	float	A float to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, int) Function**

Writes the given int to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, int i);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
i	int	An int to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, long) Function**

Writes the given long to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, long l);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	A output stream.
l	long	A long to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, sbyte) Function**

Writes the given sbyte to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, sbyte b);
```

**Parameters**

Name	Type	Description
s	<a href="#">System.IO.OutputStream&amp;</a>	An output stream.
b	sbyte	An sbyte to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, short) Function**

Writes the given short to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, short x);
```

**Parameters**

Name	Type	Description
s	<a href="#">System.IO.OutputStream&amp;</a>	An output stream.
x	short	A short to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, uint) Function**

Writes the given uint to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, uint u);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
u	uint	A uint to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, ulong) Function**

Writes the given ulong to given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, ulong u);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
u	ulong	A ulong to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

**operator<<(System.IO.OutputStream&, ushort) Function**

Writes the given ushort to the given output stream.

**Syntax**

```
public System.IO.OutputStream& operator<<(System.IO.OutputStream& s, ushort u);
```

**Parameters**

Name	Type	Description
s	System.IO.OutputStream&	An output stream.
u	ushort	A ushort to write.

**Returns**

[System.IO.OutputStream&](#)

Returns a reference to the output stream.

### 5.10.17 OutputStream Class

A class for writing to a string.

#### Syntax

```
public class OutputStream;
```

#### Base Class

[System.IO.OutputStream](#)

#### 5.10.17.1 Member Functions

Member Function	Description
<code>OutputStream()</code>	Default constructor. Write to an empty string.
<code>OutputStream(System.IO.-&lt;br&gt;OutputStream&amp;&amp;)</code>	Move constructor.
<code>operator=(System.IO.OutputStream&amp;&amp;)</code>	Move assignment.
<code>GetStr() const</code>	Returns the contained string.
<code>OutputStream(const System.String&amp;)</code>	Constructor. Write to the end of the given string.
<code>SetStr(const System.String&amp;)</code>	Sets the contained string.
<code>Write(bool)</code>	Writes a Boolean value to the end of the contained string.
<code>Write(byte)</code>	Writes a <b>byte</b> to the end of the contained string.
<code>Write(char)</code>	Writes a character to the end of the contained string.
<code>Write(const System.String&amp;)</code>	Writes a string to the end of the contained string.
<code>Write(const char*)</code>	Writes a C-style string to the end of the contained string.
<code>Write(double)</code>	Writes a <b>double</b> to the end of the contained string.
<code>Write(float)</code>	Writes a <b>float</b> to the end of the contained string.
<code>Write(int)</code>	Writes an <b>int</b> to the end of the contained string.
<code>Write(long)</code>	Writes a <b>long</b> to the end of the contained string.

<code>Write(sbyte)</code>	Writes an <b>sbyte</b> to the end of the contained string.
<code>Write(short)</code>	Writes a <b>short</b> to the end of the contained string.
<code>Write(uint)</code>	Writes a <b>uint</b> to the end of the contained string.
<code>Write(ulong)</code>	Writes a <b>ulong</b> to the end of the contained string.
<code>Write(ushort)</code>	Writes a <b>ushort</b> to the end of the contained string.
<code>WriteLine()</code>	Writes a new line to the end of the contained string.
<code>WriteLine(bool)</code>	Writes a Boolean followed by a new line to the end of the contained string.
<code>WriteLine(byte)</code>	Writes a <b>byte</b> followed by a new line to the end of the contained string.
<code>WriteLine(char)</code>	Writes a character followed by a new line to the end of the contained string.
<code>WriteLine(const System.String&amp;)</code>	Writes a string followed by a new line to the end of the contained string.
<code>WriteLine(const char*)</code>	Writes a C-style string followed by a new line to the end of the contained string.
<code>WriteLine(double)</code>	Writes a <b>double</b> followed by a new line to the end of the contained string.
<code>WriteLine(float)</code>	Writes a <b>float</b> followed by a new line to the end of the contained string.
<code>WriteLine(int)</code>	Writes an <b>int</b> followed by a new line to the end of the contained string.
<code>WriteLine(long)</code>	Writes a <b>long</b> followed by a new line to the end of the contained string.
<code>WriteLine(sbyte)</code>	Writes an <b>sbyte</b> followed by a new line to the end of the contained string.
<code>WriteLine(short)</code>	Writes a <b>short</b> followed by a new line to the end of the contained string.

<a href="#">WriteLine(uint)</a>	Writes a <b>uint</b> followed by a new line to the end of the contained string.
<a href="#">WriteLine(ulong)</a>	Writes a <b>ulong</b> followed by a new line to the end of the contained string.
<a href="#">WriteLine(ushort)</a>	Writes a <b>ushort</b> followed by a new line to the end of the contained string.
<a href="#">~OutputStringStream()</a>	Destructor.

**OutputStringStream() Member Function**

Default constructor. Write to an empty string.

**Syntax**

```
public OutputStringStream();
```

**OutputStringStream(System.IO.OutputStringStream&&) Member Function**

Move constructor.

**Syntax**

```
public OutputStringStream(System.IO.OutputStringStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.OutputStringStream&&	

**operator=(System.IO.OutputStream&&)** Member Function

Move assignment.

**Syntax**

```
public void operator=(System.IO.OutputStream&& __parameter0);
```

**Parameters**

Name	Type	Description
__parameter0	System.IO.OutputStream&&	

**GetStr() const Member Function**

Returns the contained string.

**Syntax**

```
public const System.String& GetStr() const;
```

**Returns**

**const System.String&**

Returns the contained string.

**OutputStringStream(const System.String&) Member Function**

Constructor. Write to the end of the given string.

**Syntax**

```
public OutputStringStream(const System.String& str_);
```

**Parameters**

Name	Type	Description
str_	const System.String&	A string to write to.

**SetStr(const System.String&)** Member Function

Sets the contained string.

**Syntax**

```
public void SetStr(const System.String& str_);
```

**Parameters**

Name	Type	Description
str_	const System.String&	A string to write to.

**Write(bool) Member Function**

Writes a Boolean value to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
b	bool	A Boolean value to write.

**Write(byte) Member Function**

Writes a **byte** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
b	byte	A byte to write.

**Write(char) Member Function**

Writes a character to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
c	char	A character to write.

**Write(const System.String&)** Member Function

Writes a string to the end of the contained string.

**Syntax**

```
public void Write(const System.String& s);
```

**Parameters**

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

**Write(const char\*) Member Function**

Writes a C-style string to the end of the contained string.

**Syntax**

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

**Parameters**

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

**Write(double) Member Function**

Writes a **double** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
d	double	A <b>double</b> to write.

**Write(float) Member Function**

Writes a **float** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
f	float	A <b>float</b> to write.

**Write(int) Member Function**

Writes an **int** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
i	int	An <b>int</b> to write.

**Write(long) Member Function**

Writes a **long** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
l	long	A long to write.

**Write(sbyte) Member Function**

Writes an **sbyte** to the end of the contained string.

**Syntax**

```
public void Write(sbyte b);
```

**Parameters**

Name	Type	Description
b	sbyte	An <b>sbyte</b> to write.

**Write(short) Member Function**

Writes a **short** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
s	short	A <b>short</b> to write.

**Write(uint) Member Function**

Writes a **uint** to the end of the contained string.

**Syntax**

```
public void Write(uint i);
```

**Parameters**

Name	Type	Description
i	uint	A <b>uint</b> to write.

**Write(ulong) Member Function**

Writes a **ulong** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ulong	A <b>ulong</b> to write.

**Write(ushort) Member Function**

Writes a **ushort** to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ushort	A <b>ushort</b> to write.

**WriteLine() Member Function**

Writes a new line to the end of the contained string.

**Syntax**

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

**WriteLine(bool) Member Function**

Writes a Boolean followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
b	bool	A Boolean to write.

**WriteLine(byte) Member Function**

Writes a **byte** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
b	byte	A <b>byte</b> to write.

**WriteLine(char) Member Function**

Writes a character followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
c	char	A character to write.

**WriteLine(const System.String&)** Member Function

Writes a string followed by a new line to the end of the contained string.

**Syntax**

```
public void WriteLine(const System.String& s);
```

**Parameters**

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

**WriteLine(const char\*) Member Function**

Writes a C-style string followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

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

**WriteLine(double) Member Function**

Writes a **double** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
d	double	A <b>double</b> to write.

**WriteLine(float) Member Function**

Writes a **float** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
f	float	A <b>float</b> to write.

**WriteLine(int) Member Function**

Writes an **int** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
i	int	An <b>int</b> to write.

**WriteLine(long) Member Function**

Writes a **long** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
l	long	A long to write.

**WriteLine(sbyte) Member Function**

Writes an **sbyte** followed by a new line to the end of the contained string.

**Syntax**

```
public void WriteLine(sbyte b);
```

**Parameters**

Name	Type	Description
b	sbyte	An <b>sbyte</b> to write.

**WriteLine(short) Member Function**

Writes a **short** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
s	short	A <b>short</b> to write.

**WriteLine(uint) Member Function**

Writes a **uint** followed by a new line to the end of the contained string.

**Syntax**

```
public void WriteLine(uint i);
```

**Parameters**

Name	Type	Description
i	uint	A <b>uint</b> to write.

**WriteLine(ulong) Member Function**

Writes a **ulong** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ulong	A <b>ulong</b> to write.

**WriteLine(ushort) Member Function**

Writes a **ushort** followed by a new line to the end of the contained string.

**Syntax**

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

**Parameters**

Name	Type	Description
u	ushort	A <b>ushort</b> to write.

**~OutputStringStream() Member Function**

Destructor.

**Syntax**

```
public ~OutputStringStream();
```

### 5.10.18 Path Class

A static class for manipulating paths.

#### Syntax

```
public static class Path;
```

#### 5.10.18.1 Member Functions

Member Function	Description
ChangeExtension(const System.String&, const System.String&) const	Changes an extension of a path.
Combine(const System.String&, const System.String&) const	Combines two paths.
GetDirectoryName(const System.String&) const	Returns the directory name part of a path.
GetExtension(const System.String&) const	Returns the extension of a path.
GetFileName(const System.String&) const	Returns the file name part of a path.
GetFileNameWithoutExtension(const System.String&) const	Returns the file name without an extension part of a path.
GetParent(const System.String&) const	Returns a given path with last component removed.
HasExtension(const System.String&) const	Returns true, if the given path has an extension, false otherwise.
IsAbsolute(const System.String&) const	Returns true if the given path is absolute, false otherwise.
IsRelative(const System.String&) const	Returns true if the given path is relative, false otherwise.
MakeCanonical(const System.String&) const	Returns a canonical representation of a path.

**ChangeExtension(const System.String&, const System.String&) const Member Function**

Changes an extension of a path.

**Syntax**

```
public static System.String ChangeExtension(const System.String& path, const System.String& extension) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.
extension	const System.String&	A new extension.

**Returns**

[System.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.

**Combine(const System.String&, const System.String&) const Member Function**

Combines two paths.

**Syntax**

```
public static System.String Combine(const System.String& path1, const System.String& path2) const;
```

**Parameters**

Name	Type	Description
path1	const System.String&	The first path.
path2	const System.String&	The second path.

**Returns**

[System.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.

**GetDirectoryName(const System.String&) const Member Function**

Returns the directory name part of a path.

**Syntax**

```
public static System.String GetDirectoryName(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.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.

**GetExtension(const System.String&) const Member Function**

Returns the extension of a path.

**Syntax**

```
public static System.String GetExtension(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.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.

**GetFileName(const System.String&) const Member Function**

Returns the file name part of a path.

**Syntax**

```
public static System.String GetFileName(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.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.

**GetFileNameWithoutExtension(const System.String&) const Member Function**

Returns the file name without an extension part of a path.

**Syntax**

```
public static System.String GetFileNameWithoutExtension(const System.String& path)  
const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.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.

**GetParent(const System.String&) const Member Function**

Returns a given path with last component removed.

**Syntax**

```
public static System.String GetParent(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.String](#)

Returns a given path with last component removed.

**HasExtension(const System.String&) const Member Function**

Returns true, if the given path has an extension, false otherwise.

**Syntax**

```
public static bool HasExtension(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.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.

**IsAbsolute(const System.String&) const Member Function**

Returns true if the given path is absolute, false otherwise.

**Syntax**

```
public static bool IsAbsolute(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.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.

**IsRelative(const System.String&) const Member Function**

Returns true if the given path is relative, false otherwise.

**Syntax**

```
public static bool IsRelative(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.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 System.IO.Path.IsAbsolute(System.String).const.ref function for the given path.

**MakeCanonical(const System.String&) const Member Function**

Returns a canonical representation of a path.

**Syntax**

```
public static System.String MakeCanonical(const System.String& path) const;
```

**Parameters**

Name	Type	Description
path	const System.String&	A path.

**Returns**

[System.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.

## 5.11 Functions

Function	Description
CreateDirectories(const System.String&)	Creates all directories along the given directory path.
DirectoryExists(const System.String&)	Returns true if a directory with a given path name exists, false otherwise.
FileContentsEqual(const System.String&, const System.String&)	Returns true if the given files have equal content, false otherwise.
FileExists(const System.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 System.String&)	Returns an absolute path corresponding to the given absolute or relative path.
PathExists(const System.String&)	Returns true if the given path exists, false otherwise.
ReadFile(const System.String&)	Reads a file with the given name into a string and returns it.

### 5.11.19 CreateDirectories(const System.String&) Function

Creates all directories along the given directory path.

#### Syntax

```
public void CreateDirectories(const System.String& directoryPath);
```

#### Parameters

Name	Type	Description
directoryPath	const System.String&	A directory path.

### 5.11.20 DirectoryExists(const System.String&) Function

Returns true if a directory with a given path name exists, false otherwise.

#### Syntax

```
public bool DirectoryExists(const System.String& directoryPath);
```

#### Parameters

Name	Type	Description
directoryPath	const System.String&	A path to test.

#### Returns

bool

Returns true if a directory with a given path name exists, false otherwise.

### 5.11.21 FileContentsEqual(const System.String&, const System.String&) Function

Returns true if the given files have equal content, false otherwise.

#### Syntax

```
public bool FileContentsEqual(const System.String& fileName1, const System.String& fileName2);
```

#### Parameters

Name	Type	Description
fileName1	const System.String&	The first file.
fileName2	const System.String&	The second file.

#### Returns

bool

Returns a given path with last component starting with '/' character removed.

### 5.11.22 FileExists(const System.String&) Function

Returns true if a file with a given path name exists, false otherwise.

#### Syntax

```
public bool FileExists(const System.String& filePath);
```

#### Parameters

Name	Type	Description
filePath	const System.String&	A path to test.

#### Returns

bool

Returns true if a file with a given path name exists, false otherwise.

### 5.11.23 GetCurrentWorkingDirectory() Function

Returns a path to the current working directory.

#### Syntax

```
public System.String GetCurrentWorkingDirectory();
```

#### Returns

[System.String](#)

Returns a path to the current working directory.

### 5.11.24 GetFullPath(const System.String&) Function

Returns an absolute path corresponding to the given absolute or relative path.

#### Syntax

```
public System.String GetFullPath(const System.String& path);
```

#### Parameters

Name	Type	Description
path	const System.String&	A path.

#### Returns

[System.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.

### 5.11.25 PathExists(const System.String&) Function

Returns true if the given path exists, false otherwise.

#### Syntax

```
public bool PathExists(const System.String& path);
```

#### Parameters

Name	Type	Description
path	const System.String&	A path.

#### Returns

bool

Returns true if the given path exists, false otherwise.

### 5.11.26 ReadFile(const System.String&) Function

Reads a file with the given name into a string and returns it.

#### Syntax

```
public System.String ReadFile(const System.String& fileName);
```

#### Parameters

Name	Type	Description
fileName	const System.String&	The name of the file to read.

#### Returns

[System.String](#)

Returns the contents of the given file.

## 5.12 Enumerations

Enumeration	Description
FileMode	Open mode for <a href="#">FileByteStream</a> .
OpenMode	An open mode for a binary file stream.

### 5.12.26.1 FileMode Enumeration

Open mode for [FileByteStream](#).

#### Enumeration Constants

Constant	Value	Description
append	0	Opens the file for appending to the end of it.
create	1	Creates a file if it does not exists, or truncates it to zero bytes if exists. The file is opened for writing.
open	2	Opens the file for reading.

### 5.12.26.2 OpenMode Enumeration

An open mode for a binary file stream.

#### Enumeration Constants

Constant	Value	Description
readOnly	0	Open the file with read only access.
readWrite	2	Open the file with read and write access.
writeOnly	1	Open the file with write only access. Truncates the file if it exists.

# **6 System.Security Namespace**

Contains classes and functions for computing cryptographic hash functions.

## 6.13 Classes

Class	Description
<a href="#">Sha1</a>	Class for computing SHA-1 hash value.
<a href="#">Sha256</a>	Class for computing SHA-256 hash value.
<a href="#">Sha512</a>	Class for computing SHA-512 hash value.

### 6.13.1 Sha1 Class

Class for computing SHA-1 hash value.

#### Syntax

```
public class Sha1;
```

#### 6.13.1.1 Member Functions

Member Function	Description
<code>Sha1()</code>	Default constructor.
<code>Sha1(const System.Security Sha1&amp;)</code>	Copy constructor.
<code>operator=(const System.Security Sha1&amp;)</code>	Copy assignment.
<code>Sha1(System.Security Sha1&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Security Sha1&amp;&amp;)</code>	Move assignment.
<code>GetDigest()</code>	Returns computed SHA-1 message digest as a hexadecimal string.
<code>Process(byte)</code>	Process one byte.
<code>Process(const void*, const void*)</code>	Process bytes in range <code>begin</code> and <code>end</code> .
<code>Process(const void*, int)</code>	Process a block of memory.
<code>Reset()</code>	Reset SHA-1 computation ready for computing another message.

**Sha1() Member Function**

Default constructor.

**Syntax**

```
public Sha1();
```

**Sha1(const System.Security Sha1&) Member Function**

Copy constructor.

**Syntax**

```
public Sha1(const System.Security Sha1& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha1&	Argument to copy.

**operator=(const System.Security Sha1&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Security Sha1& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha1&	Argument to assign.

**Sha1(System.Security Sha1&&) Member Function**

Move constructor.

**Syntax**

```
public Sha1(System.Security Sha1&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha1&&	Argument to move from.

**operator=(System.Security Sha1&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Security Sha1&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha1&&	Argument to assign from.

**GetDigest() Member Function**

Returns computed SHA-1 message digest as a hexadecimal string.

**Syntax**

```
public System.String GetDigest();
```

**Returns**

[System.String](#)

Returns computed SHA-1 message digest as a hexadecimal string.

**Process(byte) Member Function**

Process one byte.

**Syntax**

```
public void Process(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to process.

**Process(const void\*, const void\*) Member Function**

Process bytes in range `begin` and `end`.

**Syntax**

```
public void Process(const void* begin, const void* end);
```

**Parameters**

Name	Type	Description
begin	const void*	Pointer to the beginning of bytes to process.
end	const void*	Pointer to the end of bytes to process.

**Process(const void\*, int) Member Function**

Process a block of memory.

**Syntax**

```
public void Process(const void* buf, int count);
```

**Parameters**

Name	Type	Description
buf	const void*	Pointer to memory buffer.
count	int	Number of bytes to process.

**Reset() Member Function**

Reset SHA-1 computation ready for computing another message.

**Syntax**

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

### 6.13.2 Sha256 Class

Class for computing SHA-256 hash value.

#### Syntax

```
public class Sha256;
```

#### 6.13.2.1 Member Functions

Member Function	Description
<code>Sha256()</code>	Default constructor.
<code>Sha256(const System.Security Sha256&amp;)</code>	Copy constructor.
<code>operator=(const System.Security Sha256&amp;)</code>	Copy assignment.
<code>Sha256(System.Security Sha256&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Security Sha256&amp;&amp;)</code>	Move assignment.
<code>GetDigest()</code>	Returns computed SHA-256 message digest as a hexadecimal string.
<code>Process(byte)</code>	Process one byte.
<code>Process(const void*, const void*)</code>	Process bytes in range <code>begin</code> and <code>end</code> .
<code>Process(const void*, int)</code>	Process a block of memory.
<code>Reset()</code>	Reset SHA-256 computation ready for computing another message.

**Sha256() Member Function**

Default constructor.

**Syntax**

```
public Sha256();
```

**Sha256(const System.Security Sha256&) Member Function**

Copy constructor.

**Syntax**

```
public Sha256(const System.Security Sha256& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha256&	Argument to copy.

**operator=(const System.Security Sha256&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Security Sha256& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha256&	Argument to assign.

**Sha256(System.Security Sha256&&) Member Function**

Move constructor.

**Syntax**

```
public Sha256(System.Security Sha256&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha256&&	Argument to move from.

**operator=(System.Security Sha256&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Security Sha256&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha256&&	Argument to assign from.

**GetDigest() Member Function**

Returns computed SHA-256 message digest as a hexadecimal string.

**Syntax**

```
public System.String GetDigest();
```

**Returns**

[System.String](#)

Returns computed SHA-256 message digest as a hexadecimal string.

**Process(byte) Member Function**

Process one byte.

**Syntax**

```
public void Process(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to process.

**Process(const void\*, const void\*) Member Function**

Process bytes in range `begin` and `end`.

**Syntax**

```
public void Process(const void* begin, const void* end);
```

**Parameters**

Name	Type	Description
begin	const void*	Pointer to the beginning of bytes to process.
end	const void*	Pointer to the end of bytes to process.

**Process(const void\*, int) Member Function**

Process a block of memory.

**Syntax**

```
public void Process(const void* buf, int count);
```

**Parameters**

Name	Type	Description
buf	const void*	Pointer to memory buffer.
count	int	Number of bytes to process.

**Reset() Member Function**

Reset SHA-256 computation ready for computing another message.

**Syntax**

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

### 6.13.3 Sha512 Class

Class for computing SHA-512 hash value.

#### Syntax

```
public class Sha512;
```

#### 6.13.3.1 Member Functions

Member Function	Description
<code>Sha512()</code>	Default constructor.
<code>Sha512(const System.Security Sha512&amp;)</code>	Copy constructor.
<code>operator=(const System.Security Sha512&amp;)</code>	Copy assignment.
<code>Sha512(System.Security Sha512&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Security Sha512&amp;&amp;)</code>	Move assignment.
<code>GetDigest()</code>	Returns computed SHA-512 message digest as a hexadecimal string.
<code>Process(byte)</code>	Process one byte.
<code>Process(const void*, const void*)</code>	Process bytes in range <code>begin</code> and <code>end</code> .
<code>Process(const void*, int)</code>	Process a block of memory.
<code>Reset()</code>	Reset SHA-512 computation ready for computing another message.

**Sha512() Member Function**

Default constructor.

**Syntax**

```
public Sha512();
```

**Sha512(const System.Security Sha512&) Member Function**

Copy constructor.

**Syntax**

```
public Sha512(const System.Security Sha512& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha512&	Argument to copy.

**operator=(const System.Security Sha512&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Security Sha512& that);
```

**Parameters**

Name	Type	Description
that	const System.Security Sha512&	Argument to assign.

**Sha512(System.Security Sha512&&) Member Function**

Move constructor.

**Syntax**

```
public Sha512(System.Security Sha512&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha512&&	Argument to move from.

**operator=(System.Security Sha512&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Security Sha512&& that);
```

**Parameters**

Name	Type	Description
that	System.Security Sha512&&	Argument to assign from.

**GetDigest() Member Function**

Returns computed SHA-512 message digest as a hexadecimal string.

**Syntax**

```
public System.String GetDigest();
```

**Returns**

[System.String](#)

Returns computed SHA-512 message digest as a hexadecimal string.

**Process(byte) Member Function**

Process one byte.

**Syntax**

```
public void Process(byte x);
```

**Parameters**

Name	Type	Description
x	byte	A byte to process.

**Process(const void\*, const void\*) Member Function**

Process bytes in range `begin` and `end`.

**Syntax**

```
public void Process(const void* begin, const void* end);
```

**Parameters**

Name	Type	Description
begin	const void*	Pointer to the beginning of bytes to process.
end	const void*	Pointer to the end of bytes to process.

**Process(const void\*, int) Member Function**

Process a block of memory.

**Syntax**

```
public void Process(const void* buf, int count);
```

**Parameters**

Name	Type	Description
buf	const void*	Pointer to memory buffer.
count	int	Number of bytes to process.

**Reset() Member Function**

Reset SHA-512 computation ready for computing another message.

**Syntax**

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

## 6.14 Functions

Function	Description
<code>GetSha1FileDigest(const System.String&amp;)</code>	Returns SHA-1 hash value for a contents of given file as a hexadecimal string.
<code>GetSha1MessageDigest(const System.String&amp;)</code>	Returns SHA-1 hash value for a given string as a hexadecimal string.
<code>GetSha256FileDigest(const System.String&amp;)</code>	Returns SHA-256 hash value for a contents of given file as a hexadecimal string.
<code>GetSha256MessageDigest(const System.String&amp;)</code>	Returns SHA-256 hash value for a given string as a hexadecimal string.
<code>GetSha512FileDigest(const System.String&amp;)</code>	Returns SHA-512 hash value for a contents of given file as a hexadecimal string.
<code>GetSha512MessageDigest(const System.String&amp;)</code>	Returns SHA-512 hash value for a given string.
<code>LeftRotate(uint, uint)</code>	Rotates bits of given uint left by given amount.
<code>RightRotate(uint, uint)</code>	Rotates bits of given uint right by given amount.
<code>RightRotate(ulong, ulong)</code>	Rotates bits of given ulong right by given amount.

### 6.14.4 GetSha1FileDigest(const System.String&) Function

Returns SHA-1 hash value for a contents of given file as a hexadecimal string.

#### Syntax

```
public System.String GetSha1FileDigest(const System.String& filePath);
```

#### Parameters

Name	Type	Description
filePath	const System.String&	A file path.

#### Returns

[System.String](#)

Returns SHA-1 hash value for a contents of given file as a hexadecimal string.

### 6.14.5 GetSha1MessageDigest(const System.String&) Function

Returns SHA-1 hash value for a given string as a hexadecimal string.

#### Syntax

```
public System.String GetSha1MessageDigest(const System.String& message);
```

#### Parameters

Name	Type	Description
message	const System.String&	A string.

#### Returns

[System.String](#)

Returns SHA-1 hash value for a given string as a hexadecimal string.

## 6.14.6 GetSha256FileDigest(const System.String&) Function

Returns SHA-256 hash value for a contents of given file as a hexadecimal string.

### Syntax

```
public System.String GetSha256FileDigest(const System.String& filePath);
```

### Parameters

Name	Type	Description
filePath	const System.String&	A file path.

### Returns

[System.String](#)

A file path.

### 6.14.7 GetSha256MessageDigest(const System.String&) Function

Returns SHA-256 hash value for a given string as a hexadecimal string.

#### Syntax

```
public System.String GetSha256MessageDigest(const System.String& message);
```

#### Parameters

Name	Type	Description
message	const System.String&	A string.

#### Returns

[System.String](#)

Returns SHA-256 hash value for a given string as a hexadecimal string.

## 6.14.8 GetSha512FileDigest(const System.String&) Function

Returns SHA-512 hash value for a contents of given file as a hexadecimal string.

### Syntax

```
public System.String GetSha512FileDigest(const System.String& filePath);
```

### Parameters

Name	Type	Description
filePath	const System.String&	A file path.

### Returns

[System.String](#)

Returns SHA-512 hash value for a contents of given file as a hexadecimal string.

### 6.14.9 GetSha512MessageDigest(const System.String&) Function

Returns SHA-512 hash value for a given string.

#### Syntax

```
public System.String GetSha512MessageDigest(const System.String& message);
```

#### Parameters

Name	Type	Description
message	const System.String&	A string.

#### Returns

[System.String](#)

Returns SHA-256 hash value for a given string.

### 6.14.10 LeftRotate(uint, uint) Function

Rotates bits of given uint left by given amount.

#### Syntax

```
public uint LeftRotate(uint x, uint n);
```

#### Parameters

Name	Type	Description
x	uint	Value to rotate.
n	uint	Number of bits to rotate.

#### Returns

uint

Returns rotated value.

### 6.14.11 RightRotate(uint, uint) Function

Rotates bits of given uint right by given amount.

#### Syntax

```
public uint RightRotate(uint x, uint n);
```

#### Parameters

Name	Type	Description
x	uint	Value to rotate.
n	uint	Number of bits to rotate.

#### Returns

uint

Returns rotated value.

### 6.14.12 RightRotate(ulong, ulong) Function

Rotates bits of given ulong right by given amount.

#### Syntax

```
public ulong RightRotate(ulong x, ulong n);
```

#### Parameters

Name	Type	Description
x	ulong	Value to rotate.
n	ulong	Number of bits to rotate.

#### Returns

ulong

Returns rotated value.

# 7 System.Text Namespace

Contains classes and functions for manipulating text.

## 7.15 Classes

Class	Description
<a href="#">CodeFormatter</a>	A class for generating indented text.

### 7.15.1 CodeFormatter Class

A class for generating indented text.

#### Syntax

```
public class CodeFormatter;
```

#### 7.15.1.1 Member Functions

Member Function	Description
<code>CodeFormatter()</code>	Default constructor.
<code>CodeFormatter(System.IO.OutputStream&amp;)</code>	Constructor. Initializes the the code formatter with the given output stream.
<code>CurrentIndent() const</code>	Returns the current indent in characters.
<code>DecIndent()</code>	Decreases the indent.
<code>IncIndent()</code>	Increases the indent.
<code>Indent() const</code>	Returns the indent level.
<code>IndentSize() const</code>	Returns the number of characters to indent.
<code>Line() const</code>	Returns current line number.
<code>SetIndentSize(int)</code>	Sets the number of characters to indent.
<code>SetLine(int)</code>	Sets current line number.
<code>Write(const System.String&amp;)</code>	If at the beginning of a line, writes a string indented with the current indent. Otherwise writes the given string.
<code>WriteLine()</code>	Writes end-of-line character.
<code>WriteLine(const System.String&amp;)</code>	Writes given string using System.Text.-CodeFormatter.WriteLine(System.String.const.ref function and then writes end-of-line character.

**CodeFormatter() Member Function**

Default constructor.

**Syntax**

```
public CodeFormatter();
```

**CodeFormatter(System.IO.OutputStream&)** Member Function

Constructor. Initializes the the code formatter with the given output stream.

**Syntax**

```
public CodeFormatter(System.IO.OutputStream& stream_);
```

**Parameters**

Name	Type	Description
stream_	<a href="#">System.IO.OutputStream&amp;</a>	An output stream.

**CurrentIndent() const Member Function**

Returns the current indent in characters.

**Syntax**

```
public int CurrentIndent() const;
```

**Returns**

int

Returns the current indent in characters.

**DecIndent() Member Function**

Decreases the indent.

**Syntax**

```
public void DecIndent();
```

**IncIndent() Member Function**

Increases the indent.

**Syntax**

```
public void IncIndent();
```

**Indent() const Member Function**

Returns the indent level.

**Syntax**

```
public int Indent() const;
```

**Returns**

int

Returns the indent level.

**IndentSize() const Member Function**

Returns the number of characters to indent.

**Syntax**

```
public int IndentSize() const;
```

**Returns**

int

Returns the number of characters to indent.

**Line() const Member Function**

Returns current line number.

**Syntax**

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

**Returns**

int

Returns current line number.

**SetIndentSize(int) Member Function**

Sets the number of characters to indent.

**Syntax**

```
public void SetIndentSize(int indentSize_);
```

**Parameters**

Name	Type	Description
indentSize_	int	The number of characters to indent.

**SetLine(int) Member Function**

Sets current line number.

**Syntax**

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

**Parameters**

Name	Type	Description
line_	int	Line number.

**Write(const System.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 System.String& text);
```

**Parameters**

Name	Type	Description
text	const System.String&	A string to write.

**WriteLine() Member Function**

Writes end-of-line character.

**Syntax**

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

**WriteLine(const System.String&) Member Function**

Writes given string using System.Text.CodeFormatter.Write.System.String.const.ref function and then writes end-of-line character.

**Syntax**

```
public void WriteLine(const System.String& text);
```

**Parameters**

Name	Type	Description
text	const System.String&	A string to write

## 7.16 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 System.String&)	Returns a string literal representation of a string.
StringStr(const System.String&)	Returns a string where control characters of given string are escaped.
Trim(const System.String&)	Returns a string where white space from the beginning and end of given string are removed.
TrimAll(const System.String&)	Returns a string where white space from the beginning, middle and end of given string are removed. Replaces occurrences of white space in the middle of the given string with one space character.

## 7.16.2 CharStr(char) Function

Returns a string representation of a character.

### Syntax

```
public System.String CharStr(char c);
```

### Parameters

Name	Type	Description
c	char	A character to convert.

### Returns

[System.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.

### 7.16.3 HexEscape(char) Function

Returns a hexadecimal representation of the given character.

#### Syntax

```
public System.String HexEscape(char c);
```

#### Parameters

Name	Type	Description
c	char	A character.

#### Returns

[System.String](#)

Returns a hexadecimal representation of the given character.

## 7.16.4 MakeCharLiteral(char) Function

Returns a character literal representation of a character.

### Syntax

```
public System.String MakeCharLiteral(char c);
```

### Parameters

Name	Type	Description
c	char	A character to convert.

### Returns

[System.String](#)

Returns a character literal representation of a character.

### Remarks

Returns the character enclosed in apostrophes and escaped if necessary.

## 7.16.5 MakeStringLiteral(const System.String&) Function

Returns a string literal representation of a string.

### Syntax

```
public System.String MakeStringLiteral(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to convert.

### Returns

[System.String](#)

Returns a string literal representation of a string.

### Remarks

Returns the string enclosed in quotes and characters escaped if necessary.

## 7.16.6 StringStr(const System.String&) Function

Returns a string where control characters of given string are escaped.

### Syntax

```
public System.String StringStr(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string.

### Returns

[System.String](#)

Returns a string where control characters of given string are escaped.

### 7.16.7 Trim(const System.String&) Function

Returns a string where white space from the beginning and end of given string are removed.

#### Syntax

```
public System.String Trim(const System.String& s);
```

#### Parameters

Name	Type	Description
s	const System.String&	A string to trim.

#### Returns

[System.String](#)

Returns a string where white space from the beginning and end of given string are removed.

## 7.16.8 TrimAll(const System.String&) Function

Returns a string where white space from the beginning, middle and end of given string are removed. Replaces occurrences of white space in the middle of the given string with one space character.

### Syntax

```
public System.String TrimAll(const System.String& s);
```

### Parameters

Name	Type	Description
s	const System.String&	A string to trim.

### Returns

[System.String](#)

Returns a string where white space from the beginning, middle and end of given string are removed.

# 8 System.Threading Namespace

Contains classes and functions for controlling multiple threads of execution.

## 8.17 Concepts

Concept	Description
<a href="#">Lockable&lt;M&gt;</a>	Lockable class contains Lock() and Unlock() member functions.

### 8.17.1 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.

## 8.18 Classes

Class	Description
<a href="#">ConditionVariable</a>	Condition variables can be used as a communication mechanism among threads.
<a href="#">LockGuard&lt;M&gt;</a>	Helper class for locking and unlocking lockable object (mutex or recursive mutex).
<a href="#">Mutex</a>	Mutexes are a synchronization mechanism for controlling threads' access to some data.
<a href="#">RecursiveMutex</a>	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.
<a href="#">Thread</a>	Represents thread of execution.
<a href="#">ThreadingException</a>	Exception class thrown when some thread operation fails.

## 8.18.2 ConditionVariable Class

Condition variables can be used as a communication mechanism among threads.

### Syntax

```
public class ConditionVariable;
```

#### 8.18.2.1 Member Functions

Member Function	Description
<a href="#">ConditionVariable()</a>	Constructor. Initializes the condition variable.
<a href="#">NotifyAll()</a>	Unblock all threads waiting on this condition variable.
<a href="#">NotifyOne()</a>	Unblock one thread waiting on this condition variable.
<a href="#">Wait(System.Threading.Mutex&amp;)</a>	Wait on a condition variable to become signaled (notified). Before calling this function, the mutex associated with this condition variable must be locked.
<a href="#">WaitFor(System.Threading.Mutex&amp;, System.Duration)</a>	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.
<a href="#">WaitUntil(System.Threading.Mutex&amp;, System.TimePoint)</a>	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.
<a href="#">~ConditionVariable()</a>	Destructor. Destroys the condition variable.

**ConditionVariable() Member Function**

Constructor. Initializes the condition variable.

**Syntax**

```
public ConditionVariable();
```

**NotifyAll() Member Function**

Unblock all threads waiting on this condition variable.

**Syntax**

```
public void NotifyAll();
```

**NotifyOne() Member Function**

Unblock one thread waiting on this condition variable.

**Syntax**

```
public void NotifyOne();
```

**Wait(System.Threading.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(System.Threading.Mutex& m);
```

**Parameters**

Name	Type	Description
m	System.Threading.Mutex&	Mutex associated with this condition variable.

**WaitFor(System.Threading.Mutex&, System.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(System.Threading.Mutex& m, System.Duration d);
```

**Parameters**

Name	Type	Description
m	<a href="#">System.Threading.Mutex&amp;</a>	Mutex associated with this condition variable.
d	<a href="#">System.Duration</a>	Duration to wait.

**Returns**

bool

Returns true, if specified duration has elapsed without the condition variable to become signaled, false otherwise.

**WaitUntil(System.Threading.Mutex&, System.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(System.Threading.Mutex& m, System.TimePoint tp);
```

**Parameters**

Name	Type	Description
m	<a href="#">System.Threading.Mutex&amp;</a>	Mutex associated with this condition variable.
tp	<a href="#">System.TimePoint</a>	Due time.

**Returns**

bool

Returns true, if specified time point has been reached without the condition variable to become signaled, false otherwise.

**~ConditionVariable() Member Function**

Destructor. Destroys the condition variable.

**Syntax**

```
public ~ConditionVariable();
```

### 8.18.3 LockGuard<M> Class

Helper class for locking and unlocking lockable object (mutex or recursive mutex).

#### Syntax

```
public class LockGuard<M>;
```

#### Constraint

M is [Lockable](#)

#### 8.18.3.1 Member Functions

Member Function	Description
<a href="#">LockGuard&lt;M&gt;()</a>	Default constructor.
<a href="#">GetLock()</a>	Returns the lockable object.
<a href="#">LockGuard&lt;M&gt;(M&amp;)</a>	Locks the lockable object.
<a href="#">~LockGuard&lt;M&gt;()</a>	Unlocks the lockable object.

**LockGuard<M>() Member Function**

Default constructor.

**Syntax**

```
public LockGuard<M>();
```

**GetLock() Member Function**

Returns the lockable object.

**Syntax**

```
public M& GetLock();
```

**Returns**

M&

Returns the lockable object.

**LockGuard<M>(M&) Member Function**

Locks the lockable object.

**Syntax**

```
public LockGuard<M>(M& m_);
```

**Parameters**

Name	Type	Description
m_	M&	A lockable object.

**~LockGuard<M>() Member Function**

Unlocks the lockable object.

**Syntax**

```
public ~LockGuard<M>();
```

## 8.18.4 Mutex Class

Mutexes are a synchronization mechanism for controlling threads' access to some data.

### Syntax

```
public class Mutex;
```

#### 8.18.4.1 Remarks

Basic mutex cannot be locked recursively many times by same thread.

#### 8.18.4.2 Member Functions

Member Function	Description
<a href="#">Mutex()</a>	Constructor. Initializes the mutex.
<a href="#">Handle() const</a>	Returns pointer to the mutex handle.
<a href="#">Lock()</a>	Locks the mutex.
<a href="#">TryLock()</a>	If the mutex is currently unlocked, locks the mutex and returns true. Otherwise returns false.
<a href="#">Unlock()</a>	Unlocks the mutex.
<a href="#">~Mutex()</a>	Destructor. Destroys the mutex.

**Mutex() Member Function**

Constructor. Initializes the mutex.

**Syntax**

```
public Mutex();
```

**Handle() const Member Function**

Returns pointer to the mutex handle.

**Syntax**

```
public void** Handle() const;
```

**Returns**

void\*\*

Returns pointer to the mutex handle.

**Lock() Member Function**

Locks the mutex.

**Syntax**

```
public void Lock();
```

**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.

**Unlock() Member Function**

Unlocks the mutex.

**Syntax**

```
public void Unlock();
```

**~Mutex() Member Function**

Destructor. Destroys the mutex.

**Syntax**

```
public ~Mutex();
```

### 8.18.5 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

[System.Threading.Mutex](#)

#### 8.18.5.1 Member Functions

Member Function	Description
<a href="#">RecursiveMutex()</a>	Constructor. Initializes the recursive mutex.
<a href="#">~RecursiveMutex()</a>	Destructor.

**RecursiveMutex() Member Function**

Constructor. Initializes the recursive mutex.

**Syntax**

```
public RecursiveMutex();
```

**~RecursiveMutex() Member Function**

Destructor.

**Syntax**

```
public ~RecursiveMutex();
```

## 8.18.6 Thread Class

Represents thread of execution.

### Syntax

```
public class Thread;
```

#### 8.18.6.1 Member Functions

Member Function	Description
<a href="#">Thread()</a>	Default constructor. Initializes a thread that is not associated with operating system thread.
<a href="#">Thread(System.Threading.Thread&amp;&amp;)</a>	Move constructor.
<a href="#">operator=(System.Threading.Thread&amp;&amp;)</a>	Move assignment.
<a href="#">Detach()</a>	Detaches the thread. Detached thread cannot be joined.
<a href="#">Handle() const</a>	Returns thread handle.
<a href="#">Join()</a>	Waits for the thread to terminate.
<a href="#">Joinable() const</a>	Returns true, if the thread can be joined, false otherwise.
<a href="#">Thread(System.Threading.ThreadFun, void*)</a>	Constructor. Associates the thread with an operating system thread and starts executing the specified start function asynchronously.
<a href="#">~Thread()</a>	If the thread is associated with an operating system thread and it is not joined or detached, calls exit with <a href="#">EXIT_THREADS_NOT_JOINED</a> exit status.

**Thread() Member Function**

Default constructor. Initializes a thread that is not associated with operating system thread.

**Syntax**

```
public Thread();
```

**Thread(System.Threading.Thread&&) Member Function**

Move constructor.

**Syntax**

```
public Thread(System.Threading.Thread&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Threading.Thread&amp;&amp;</a>	A thread to move from.

**operator=(System.Threading.Thread&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Threading.Thread&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Threading.Thread&amp;&amp;</a>	A thread to assign from.

**Detach() Member Function**

Detaches the thread. Detached thread cannot be joined.

**Syntax**

```
public void Detach();
```

**Handle() const Member Function**

Returns thread handle.

**Syntax**

```
public ulong Handle() const;
```

**Returns**

ulong

Returns thread handle.

**Join() Member Function**

Waits for the thread to terminate.

**Syntax**

```
public void Join();
```

**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.

**Thread(System.Threading.ThreadFun, void\*) Member Function**

Constructor. Associates the thread with an operating system thread and starts executing the specified start function asynchronously.

**Syntax**

```
public Thread(System.Threading.ThreadFun start, void* arg);
```

**Parameters**

Name	Type	Description
start	<a href="#">System.Threading.ThreadFun</a>	A thread start function.
arg	void*	Argument to the thread start function.

**~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 ~Thread();
```

**8.18.6.2 Nonmember Functions**

Function	Description
<code>operator==(const System.Threading.Thread&amp;, const System.Threading.Thread&amp;)</code>	Returns true, if the specified threads are associated with the same operation system thread, false otherwise.

**operator==(const System.Threading.Thread&, const System.Threading.Thread&) Function**

Returns true, if the specified threads are associated with the same operation system thread, false otherwise.

**Syntax**

```
public bool operator==(const System.Threading.Thread& t1, const System.Threading.Thread& t2);
```

**Parameters**

Name	Type	Description
t1	const System.Threading.Thread&	The first thread.
t2	const System.Threading.Thread&	The second thread.

**Returns**

bool

Returns true, if the specified threads are associated with the same operation system thread, false otherwise.

### 8.18.7 ThreadingException Class

Exception class thrown when some thread operation fails.

#### Syntax

```
public class ThreadingException;
```

#### Base Class

[System.Exception](#)

#### 8.18.7.1 Member Functions

Member Function	Description
<code>ThreadingException()</code>	Default constructor.
<code>ThreadingException(const System.Threading.-ThreadingException&amp;)</code>	Copy constructor.
<code>operator=(const System.Threading.-ThreadingException&amp;)</code>	Copy assignment.
<code>ThreadingException(System.Threading.-ThreadingException&amp;&amp;)</code>	Move constructor.
<code>operator=(System.Threading.-ThreadingException&amp;&amp;)</code>	Move assignment.
<code>ThreadingException(const System.String&amp;, const System.String&amp;)</code>	Constructor. Initializes the threading exception with the specified operation description and failure reason.
<code>~ThreadingException()</code>	Destructor.

**ThreadingException() Member Function**

Default constructor.

**Syntax**

```
public ThreadingException();
```

**ThreadingException(const System.Threading.ThreadingException&) Member Function**

Copy constructor.

**Syntax**

```
public ThreadingException(const System.Threading.ThreadingException& that);
```

**Parameters**

Name	Type	Description
that	const System.Threading.ThreadingException&	Argument to copy.

**operator=(const System.Threading.ThreadingException&)** Member Function

Copy assignment.

**Syntax**

```
public void operator=(const System.Threading.ThreadingException& that);
```

**Parameters**

Name	Type	Description
that	const System.Threading.ThreadingException&	Argument to assign.

**ThreadingException(System.Threading.ThreadingException&&) Member Function**

Move constructor.

**Syntax**

```
public ThreadingException(System.Threading.ThreadingException&& that);
```

**Parameters**

Name	Type	Description
that	System.Threading.ThreadingException&&	Argument to move from.

**operator=(System.Threading.ThreadingException&&) Member Function**

Move assignment.

**Syntax**

```
public void operator=(System.Threading.ThreadingException&& that);
```

**Parameters**

Name	Type	Description
that	<a href="#">System.Threading.ThreadingException&amp;&amp;</a>	Argument to assign from.

**ThreadingException(const System.String&, const System.String&) Member Function**

Constructor. Initializes the threading exception with the specified operation description and failure reason.

**Syntax**

```
public ThreadingException(const System.String& operation, const System.String& reason);
```

**Parameters**

Name	Type	Description
operation	const System.String&	Description of the thread operation.
reason	const System.String&	Reason for failure.

**~ThreadingException() Member Function**

Destructor.

**Syntax**

```
public ~ThreadingException();
```

## 8.19 Functions

Function	Description
SleepFor(System.Duration)	Puts the calling thread to sleep for specified duration.
SleepUntil(System.TimePoint)	Puts the calling thread to sleep until specified time point has been reached.
ThreadStart(void*)	Implementation detail.

### 8.19.8 SleepFor(System.Duration) Function

Puts the calling thread to sleep for specified duration.

#### Syntax

```
public void SleepFor(System.Duration d);
```

#### Parameters

Name	Type	Description
d	System.Duration	Duration to sleep.

### 8.19.9 SleepUntil(System.TimePoint) Function

Puts the calling thread to sleep until specified time point has been reached.

#### Syntax

```
public void SleepUntil(System.TimePoint tp);
```

#### Parameters

Name	Type	Description
tp	<a href="#">System.TimePoint</a>	Time point to sleep until.

### 8.19.10 ThreadStart(void\*) Function

Implementation detail.

#### Syntax

```
public void ThreadStart(void* arg);
```

#### Parameters

Name	Type	Description
arg	void*	

## 8.20 Delegates

Delegate	Description
<a href="#">ThreadFun</a>	A thread start function delegate.

### 8.20.11 ThreadFun Delegate

A thread start function delegate.

#### Syntax

```
public delegate void ThreadFun(void* arg);
```

#### Parameters

Name	Type	Description
arg	void*	Argument given to the thread start function.

## 8.21 Constants

Constant	Type	Value	Description
EXIT_THREADS_NOT_JOINED	int	250	Program exit status when all threads are not joined or detached.