-- Operator (Visual APL Reference)

The decrement operator (--) decrements its operand by 1. The decrement operator can appear only after its operand:

Remarks

The postfix decrement operation. The result of the operation is the value of the operand before it has been decremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x--;
    print x
    x = 1.5 10 20;
    print x--;
    print x;
}
```

Output

```
1.5
0.5
1.5 10 20
0.5 9 19
```

-- Operator (Visual APL Reference)

The decrement operator (--) decrements its operand by 1. The decrement operator can appear only after its operand:

Remarks

The postfix decrement operation. The result of the operation is the value of the operand before it has been decremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x--;
    print x
    x = 1.5 10 20;
    print x--;
    print x;
}
```

Output

```
1.5
0.5
1.5 10 20
0.5 9 19
```

>> Operator (Visual APL Reference)

The right-shift operator (>>) shifts its first operand right by the number of bits specified by its second operand.

Remarks

If the first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of the second operand (second operand & 0x1f).

If the first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of the second operand (second operand & 0x3f).

If the first operand is an int or long, the right-shift is an arithmetic shift (high-order empty bits are set to the sign bit). If the first operand is of type uint or ulong, the right-shift is a logical shift (high-order bits are zero-filled).

User-defined types can overload the >> operator; the type of the first operand must be the user-defined type, and the type of the second operand must be int. For more information, see operator. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
using System;
function fn()
{
    i = -1000;
    print i >> 3;
}
```

Output

```
-125
```

>= Operator (Visual APL Reference)

All numeric and enumeration types define a "greater than or equal" relational operator, >= that returns true if the first operand is greater than or equal to the second, false otherwise.

Remarks

User-defined types can overload the >= operator. For more information, see operator. If >= is overloaded, <= must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print 1.1 >= 1;
    print 1.1 >= 1.1;
}
```

Output

```
true
true
```

== Operator (Visual APL Reference)

For predefined value types, the equality operator (==) returns true if the values of its operands are equal, false otherwise. For reference types other than string, == returns true if its two operands refer to the same object. For the string type, == compares the values of the strings.

Remarks

User-defined value types can overload the == operator (see operator). So can user-defined reference types, although by default == behaves as described above for both predefined and user-defined reference types. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    // Numeric equality:  True
    print (2 + 2) == 4;

    // Reference equality:  different objects,
    // same boxed value:  true.
    s = 1;
    t = 1;
    print s == t;

    // Define some strings:
    a =  hello ;                          "     "
    b =  String.Copy(a);
    c =  hello ;                          "     "

    // Compare string values of a constant and an instance:  True
    print a == b;
    print a == c

}
```

Output

```
true
true
true
true
```

<= Operator (Visual APL Reference)

All numeric and enumeration types define a "less than or equal" relational operator (<=) that returns true if the first operand is less than or equal to the second, false otherwise.

Remarks

User-defined types can overload the <= operator. For more information, see operator. If <= is overloaded, >= must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print 1 <= 1.1
    print 1.1 <= 1.1
}
```

Output

```
true
true
```

++ Operator (Visual APL Reference)

The increment operator (++) increments its operand by 1. The increment operator can appear only after its operand:

Remarks

The postfix increment operation. The result of the operation is the value of the operand before it has been incremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x++;
    print x
    x = 1.5 10 20;
    print x++;
    print x;
}
```

Output

```
1.5
2.5
1.5 10 20
2.5 11 21
```

|| Operator (Visual APL Reference)

The conditional-OR operator (||) performs a logical-OR of its bool operands, but only evaluates its second operand if necessary.

Remarks

The operation

x || y

corresponds to the operation

x | y

except that if x is true, y is not evaluated (because the result of the OR operation is true no matter what the value of y might be). This is known as "short-circuit" evaluation.

The conditional-OR operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators.

Example

In the following example, observe that the expression using || evaluates only the first operand.

```
using System;
function Method1()
{
    print  Method1 called ;                  "              "
    return true;
}

static bool Method2()
{
    print  Method2 called ;                  "              "
    return false;
}

static void Main()
{
    print  regular OR: ;                     "           "
    print string.Format( result is {0} , Method1() | Method2());        "
    print  short-circuit OR: ;               "                  "
    print string.Format( result is {0} , Method1() || Method2());       "
}
```

Output

```
regular OR:
Method1 called
Method2 called
result is true
short-circuit OR:
Method1 called
result is true
```

| Operator (Visual APL Reference)

Binary | operators are predefined for the integral types and bool. For integral types, | computes the bitwise OR of its operands. For bool operands, | computes the logical OR of its operands; that is, the result is false if and only if both its operands are false.

Remarks

The | operator evaluates both operators regardless of the first one's value. For example:

Example

```
using System;
function fn() {
    print true | false;  // logical or
    print false | false; // logical or
    print string.Format( 0x{0:x} , 0xf8 | 0x3f);    // bitwise or  "
}
```

Output

```
True
False
0xff
```

^ Operator (Visual APL Reference)

Binary ^ operators are predefined for the integral types and bool. For integral types, ^ computes the bitwise exclusive-OR of its operands. For bool operands, ^ computes the logical exclusive-or of its operands; that is, the result is true if and only if exactly one of its operands is true.

Remarks

Operations on integral types are generally allowed on enumeration.

Note
This operator is created with the shift-6 key, not to be confused with the alt-0 key which creates the array and operator.

Example

```
using System;
function fn()
{
    print true ^ false;  // logical exclusive-or
    print false ^ false; // logical exclusive-or
    // Bitwise exclusive-or:
    print string.Format( 0x{0:x} , 0xf8 ^ 0x3f);          "       "
}
```

Output

```
True
False
0xc7
```

&& Operator (Visual APL Reference)

The conditional-AND operator (&&) performs a logical-AND of its bool operands, but only evaluates its second operand if necessary.

Remarks

The operation

x && y

corresponds to the operation

x & y

except that if x is false, y is not evaluated (because the result of the AND operation is false no matter what the value of y may be). This is known as "short-circuit" evaluation.

The conditional-AND operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators.

Example

In the following example, observe that the expression using && evaluates only the first operand.

```
using System;
function Method1()
{
   print  Method1 called ;                  "               "
   return false;
}

function Method2()
{
   print  Method2 called ;                  "               "
   return true;
}

function fn()
{
   print  regular AND: ;                    "            "
   print string.Format( result is {0} , Method1() & Method2());        "
   print  short-circuit AND: ;              "                  "
   print string.Format ( result is {0} , Method1() && Method2());       "
}
```

Output

```
regular AND:
Method1 called
Method2 called
result is false
short-circuit AND:
Method1 called
result is false
```

& Operator (Visual APL Reference)

The & operator is a binary operator and works only on scalars.

Remarks

Binary & operators are predefined for the integral types and bool. For integral types, & computes the logical bitwise AND of its operands. For bool operands, & computes the logical AND of its operands; that is, the result is true if and only if both its operands are true.

The & operator evaluates both operators regardless of the first one's value.

For example:

```
int i = 1;
if (false & i == 1)
{
    // i is incremented, but the conditional
    // expression evaluates to false, so
    // this block does not execute.
}
```

Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print  true & false; // logical and
    print  true & true ;  // logical and
    print string.Format(0x{0:x} , 0xf8 & 0x3f);  // bitwise and  "
}
```

Output

```
False
True
0x38
```

!= Operator (Visual APL Reference)

The inequality operator (!=) returns false if its operands are equal, true otherwise. Inequality operators are predefined for all types, including string and object. User-defined types can overload the != operator.

Remarks

For predefined value types, the inequality operator (!=) returns true if the values of its operands are different, false otherwise. For reference types other than string, != returns true if its two operands refer to different objects. For the string type, != compares the values of the strings.

User-defined value types can overload the != operator (see operator). So can user-defined reference types, although by default != behaves as described above for both predefined and user-defined reference types. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    // Numeric inequality:
    print (2 + 2)  = 4;                !

    // Reference equality:  two objects, same boxed value
    s = 1;
    t = 1;
    print s  = t);               !

    // String equality:  same string value, same string objects
    a =  hello ;                            "     "
    b =  hello ;                            "     "

    // compare string values
    print a  = b;                !

}
```

Output

```
false
false
false
```