Computer Programming

Operator

Arithmetic Operators

An expression is a sequence of *operators* and their *operands*, that specifies a computation

- Operator
  - An operator is a *pre-defined symbol* that tells the compiler to perform specific mathematical or logical manipulations on the operand(s)
- Arithmetic operators

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm or $b \cdot m$	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	%	$r \bmod s$	`r % s`

Only for integer operands

Simple Arithmetic Operations

```
#include <iostream>
using namespace std;

int main( )
{
    int k=11, p=3, h=-11, m, n;
    double a, b, x=3.0, y=4.0;
    a = x*y;
    b = x/y;
    m = k%p;
    n = h%p;

    cout <<"a=" <<a <<"\t b=" <<b <<endl;
    cout <<"m=" <<m <<"\t n=" <<n <<endl;
}
```

The fractional part in integer division is truncated--*not rounded*

Compare `11/3` vs. `11/3.0`

Compare `(-11)%3` vs. `11%(-3)`

`n = h%p` has the same sign as **`h`**

```
a=12        b=0.75
m=2         n=-2
```

Assignment Operators

The value of `9` is returned as the evaluation result for `x=9;` (if the returned value is needed)

- Assignment operator =
  - ☞ LHS (left-hand side) vs. RHS (right-hand side) operands
  - An operator performs an operation and *returns a value*
  - Assignment operator returns the *assigned value*
  - *Type conversion* will be performed *implicitly* if LHS and RHS belong to different data types
- Compound assignment operators

Assigning a float-point value to an integer results in ***truncation***

```
k = k+2;        k += 2;
k = k-2;        k -= 2;
k = k*2;   ⟺    k *= 2;
k = k/2;        k /= 2;
k = k%2;        k %= 2;
```

Operator	Name
+=	addition and assignment
-=	subtraction and assignment
*=	multiplication and assignment
/=	division and assignment
%=	remainder and assignment
=	assignment

Increment & Decrement Operators

- Increment or decrement an integer by 1
  - ① Post-increment (postfix) `i++;`
  - ☞ Postfix operator modifies the value of the variable by 1 and returns the value *before modification*

```
k = i++;  {  k = i;
             i = i + 1;
```

An operator is said to have a "*side effect*" if it modifies the operand(s)

(compound) assignment operator and increment/decrement operator

  - ② Pre-increment (prefix) `++i;`
  - ☞ Prefix operator modifies the value of the variable by 1 and returns the value (variable) *after modification*

```
k = ++i;  {  i = i + 1;
             k = i;
```

The cases for `--i` and `i--` are similar

Operators with Side Effects

```
#include <iostream>
using namespace std;

int main( )
{
    int c;
    c = 5;
    cout << c-- << '\n' << endl;
    c = 5;
    cout << --c << '\n' << endl;
    c = 5;
    cout << c << '\n' << c-- << '\n' << endl;
    c = 5;
    cout << c << '\n' << --c << '\n' << endl;
}
```

The result may differ for different compilers since there are multiple accesses (with modification) to a variable between sequence points

Avoid using `++` or `--` on a variable with *multiple occurrences* in a complex expression; break one such expression into multiple expressions

Find the value of **`j`** and **`k`**:

```
i = 2;
j = 3*(i++) - 2;
i = 2;
k = 3*(++i) - 2;
```

- The order of execution of expressions between *sequence points* (e.g. statements ended by semicolons) is *undefined* to allow for latitude of compiler optimization

Relational Operators

Note that the expression
`1 < x < 3`
will result in unexpected answer

- Comparison of relation
  - Return true or false depending on the comparison result

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal to

Do not confuse == (comparison) with = (assignment)

- `bool`
  - A data type like `char`, `int`, `long`, …
  - Only two values (states): `true` or `false`

`true` and `false` are C++ keywords

```
bool x = true;
cout << "The value of x = " << x;
```

use manipulators
`boolalpha`
`noboolalpha`

Logical Operators

!! is double negative → no effect
(i.e. !!true → true)

- Logical operators

How to express exclusive OR?

Operator	Name	Operation	Operator type
!	Logical NOT	Negation	Unary
&&	Logical AND	Conjunction	Binary
||	Logical OR	Inclusive disjunction	Binary

- Truth table

A	B	A && B	A || B	!A	!B
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

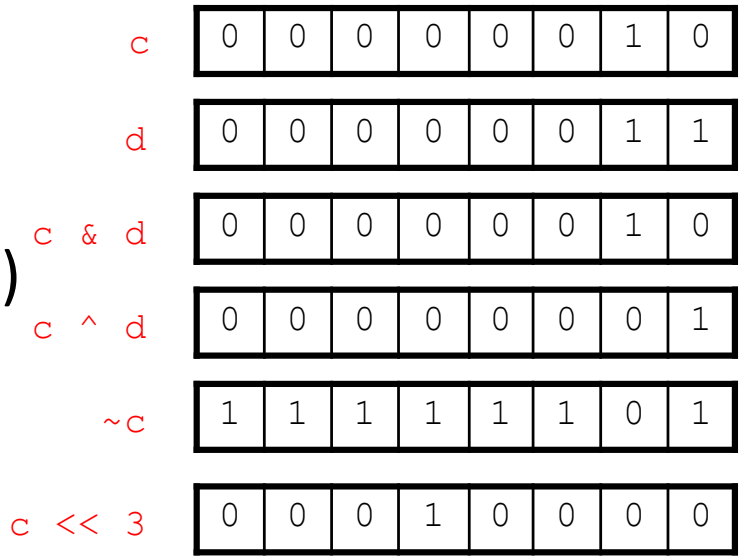
☞ To test if `(1<x<3)`, use `(1<x) && (x<3)`

Bitwise Operators

Note that these operators *do not change the value* of the operand(s)

- Operation in the bit level
  - & is bitwise AND
  - | is bitwise (inclusive) OR
  - ^ is bitwise exclusive OR (XOR)
  - ~ is bitwise complement
  - << is bitwise left shift
  - >> is bitwise right shift

`cout` and `cin` use the two operators for different purposes (operator overloading)




```
int c=2, d=3;
cout << "c & d = " << (c & d) << endl;
cout << "c | d = " << (c | d) << endl;
cout << "c ^ d = " << (c ^ d) << endl;
cout << "~c = " << (~c) << endl;
cout << "c << d = " << (c << d) << endl;
cout << "c >> d = " << (c >> d) << endl;
```

For any integer c, ~c+1 is -c (two's complement)

```
c & d = 2
c | d = 3
c ^ d = 1
~c = -3
c << d = 16
c >> d = 0
```

Operator Precedence

Associativity of an operator determines how operators of *the same precedence* are grouped for evaluation

x *op* y *op* z
L-to-R: (x *op* y) *op* z
R-to-L: x op (y *op* z)

unary

binary

Operators	Associativity	Type
()	left to right	parentheses
++ --	left to right	postfix (unary)
+ - ~ ! ++ --	right to left	sign, NOT, prefix (unary)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	bitwise shift
< <= > >=	left to right	relational
== !=	left to right	equality, inequality
&	left to right	bitwise AND
^	left to right	bitwise XOR
|	left to right	bitwise OR
&&	left to right	logical AND
||	left to right	logical OR
= += -= *= /= %=	right to left	assignment

Math Functions

The *argument* to each function can be any variable of the floating-point data type to get the desired precision level

Function	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 ) is 10.0` `ceil( -9.8 ) is -9.0`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 ) is 9.0` `floor( -9.8 ) is -10.0`
`round( x )`	rounds *x* to the integer nearest to *x*	`round( 9.2 ) is 9.0` `round( -9.8 ) is -10.0`
`trunc( x )`	nearest integer not larger in magnitude than *x*	`trunc( 9.2 ) is 9.0` `trunc( -9.8 ) is -9.0`
`fabs( x )`	absolute value of *x*	`fabs( 5.1 ) is 5.1` `fabs( -8.76 ) is 8.76`
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 2.6, 1.2 ) is 0.2`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 ) is 1.0`
`log2( x )`	logarithm of *x* (base 2)	`log2( 2.0 ) is 1.0`
`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 ) is 1.0`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 ) is 128` `pow( 9, .5 ) is 3`
`exp( x )`	exponential function e^x	`exp( 1.0 ) is 2.71828`
`sqrt( x )`	square root of *x* (where *x* is a nonnegative value)	`sqrt( 9.0 ) is 3.0`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 ) is 0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 ) is 1.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 ) is 0`
`atan( x )`	principal arc tangent of *x*, in [-π/2, +π/2] radians	`atan( 1.0 ) is 0.785398`
`tanh( x )`	hyperbolic tangent of *x*	`tanh( 1.0 ) is 0.761594`

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{(e^x - e^{-x})/2}{(e^x + e^{-x})/2}$$

Using Math Functions

```
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    double x=3.0, y=4.0, z=5.9;
    double a, b, c, d, e, f, g, h;

    a = sin(x);     b = exp(x);
    c = log(x);     d = log10(x);

    e = sqrt(x);    f = pow(x, y);

    g = floor(x);   h = ceil(x);

    double w = fabs(-3.7);

    cout << "value of e = " << e << endl;
}
```

header file

Different Types of Math Functions

- Math functions for different data types
  - Different versions of functions (with different precision) for `float`, `double`, `long double` arguments are provided in C++

```
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    float  x = log(3.0f);
    double y = log(3.0);
    long double z = log(3.0l);

    cout << "float = \t" << x << "\ndouble = \t" << y
         << "\nlong double = \t" << z << endl;
}
```

Use `setprecision()` to control the number of output digits

Example

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main( )
{
     float  x = 4.0*atan(1.0f);
     double y = 4.0*atan(1.0);
     long double z = 4.0*atan(1.0l);

     cout << setprecision(20) << "float = \t" << x
          << "\ndouble = \t" << y << "\nlong double = \t" << z << endl;

     cout << fixed << setprecision(20) << "float = \t" << x
          << "\ndouble = \t" << y << "\nlong double = \t" << z << endl;
}
```

header file

The `fixed` manipulator results in the desired number of digits "after the decimal point" to be displayed for non-scientific float-field notation

Common Pitfalls with Variables

- Problems to be avoided
  - LHS of "=" does not refer to a memory location (L-value)

```
x/y = b;
```

wrong!

  - Use of uninitialized variables

```
int z;
cout << "Uninitialized value=" << z;
```

Use of uninitialized variables is dangerous!

  - Division by zero
  - Range violation

```
int n, z;
n = 1e+9;
z = n*10;
```

`int8_t, int16_t, int32_t, int64_t`

`uint8_t, uint16_t, uint32_t, uint64_t`

Review

- Literals
- Variables
  - Naming, declaration, assignment and initialization
  - Constant variable
  - Variable reference
- Standard input and output
- Operators