Equations and Operators

Operators

Operators perform some operation on one or more items (*operands*)

Binary & Unary Operators

A binary operator operates on two values (*operands*)

A unary operator operates on a single value (*operand*)

Type	Example
+ is a Binary Operator	3 + 4
− is a Binary Operator	3 - 4
−negation is a Unary Operator	-3

Whether '-' is binary or unary depends on the context in which it is used

- this is true of the '-' sign in algebra, too, so don't worry. =)

Operator Precedence

Operator precedence is simply the natural order in which operators are evaluated

- identical in concept to the "order of operations" concept in math
- as in math, we can override natural precedence by grouping with ( )'s

Examples:

- multiplication (*) and division (/) have higher precedence than addition (+) and subtraction (-)

```
1  // a has the value 52
2  int a = 10 * 5 + 4 / 2;
3
4  // b has the value 45
5  int b = 10 * (5 + 4) / 2;
6
7  // c has the value 70
8  int c = 10 * (5 + 4 / 2);
9
10 // d has the value 40 -- why?
11 int d = 10 * ((5 + 4) / 2);
12
```

Operator Associativity

Associativity determines the order of evaluation when operators have the same level of precedence.

Consider `5 * 10 % 3` with left-to-right (LTR) associativity:

- expression gets evaluated as: `(5 * 10) % 3 = 2`

Same as above, but with right-to-left (RTL) associativity:

- expression gets evaluated as: `5 * (10 % 3) = 5`

Note: binary arithmetic operators are LTR associative, so first example is correct.

- See Table 2.5 on page 56 for a concise listing, or http://www.cppreference.com/wiki/operator_precedence for an exhaustive one.
- Like precedence, associativity can be overridden with ( )'s

Arithmetic Operators

Addition (+), subtraction (-), multiplication (*), and division (/)

- all very straight-forward (elementary-school math!)

Modulus operator (%):

- `10 % 5 = ?`
  - `10 / 5 = 2, remainder 0`
  - `10 % 5 = 0`
- `11 % 2 = ?`
  - `11 / 2 = 5, remainder 1`
  - `11 % 2 = 1`
- `80 % 9 = ?`
  - `80 / 9 = 8, remainder 8`
  - `80 % 9 = 8`
- `100 % 0 = ?`
  - invalid (division by `0`)!

Assignment Operators

The assignment operator (=) sets a variable's value to something new.

```
// perform a single assignment
x = 3; // x now has the value 3
```

Assignment operators are right-to-left associative

- nearly all other operators have left-to-right associative, so just know that assignment operators are the oddball operators
- example of multiple chained assignments:

```
x = y = z = 3; // evaluates as x = (y = (z = 3));
```

Assignment Operators

Abbreviated assignment operators:

Operator	Example	Equivalent Statement
+=	`x += 3`	`x = x + 3`
-=	`x -= 2`	`x = x - 2`
*=	`y *= 10`	`y = y * 10`
/=	`y /= 5`	`y = y / 5`
%=	`y %= 2`	`y = y % 2`

Each of these operators changes the value of the variable!

```
x += 3; // the value of x increases by 3

x *= 2; // the value of x doubles
```

Increment & Decrement Operators

Increment (++): *increase* a variable by 1

Decrement (--): *decrease* a variable by 1

Operator	Example	Equivalent Statements
Prefix Increment	++x	x = x + 1 x += 1
Prefix Decrement	--x	x = x - 1 x -= 1
Postfix Increment	x++	x = x + 1 x += 1
Postfix Decrement	x--	x = x - 1 x -= 1

Prefix vs Postfix

Prefix and postfix differ *slightly* in how they work:

- `prefix  (++x)`: adds / subtracts 1, then returns the value of the operand
- `postfix (x++)`: returns the value of the operand, then adds / subtracts 1

Example:

- assume two variables, x & y, with the following declarations:

```
int x = 5;
int y = 2;
```

- Prefix version:

```
// add 1 to x, then y = x
y = ++x; // y = 6; x = 6
```

- Postfix version:

```
// y = x, then add 1 to x
y = x++; // y = 5; x = 6
```

Prefix vs Postfix

Prefix (++x / --x) operations change the variable BEFORE using it

Postfix (x++ / x--) operations change the variable AFTER using it

Underflow and Overflow

Type Specifier	Bytes (Bits)	Negative Range		Positive Range	
int	4 (32)	-2,147,483,648	0	0	2,147,483,647
double	16 (128)	-2.2251×10^{308}	-1.7977×10^{-308}	1.7977×10^{-308}	2.2251×10^{308}

Because data types are allocated a finite amount of space, they can only represent a finite range of values

Take the following operation involving an `int`:

`2,147,483,647 + 1 ≠ 2,147,483,648`

`2,147,483,647 + 1 = -2,147,483,648` ?????

Similarly, for a `double`:

$1 \times 10^{308} * 10 \neq 1 \times 10^{309}$

Both of these examples demonstrate "overflow", where a positive number is bigger than the largest storable value and thus "wraps around" to become negative.

- Underflow is the opposite phenomenon.

Integer Math

ints hold integers, so this make senses (note: truncate, don't round):

```
// a's value:  12

int a = 12.8;
```

But doubles can store decimal values, right???

```
// a's value:  1

double a = 10 / 9;
```

Expressions such as `10 / 9` result in temporary variables!

- the type of the temporary variable depends on the operands to the operator
- a temporary int variable results from dividing an int by and int, so any remainder is truncated before the value is stored into the double variable, *a*.

Integer Math

An int divided by an int (integer math) results in an int

- this may mean we truncate our answer when we don't actually want to!

To get the "correct" answer, one of the numbers must be a double

- we can use a double "literal", putting a decimal point on one of the numbers:

```
// a now has the correct value (1.1111111)

double a = 10.0 / 9;        // 10 is represented as a double
```

- or we can cast one of the numbers to a double:

```
// a now has the correct value (1.1111111)

double a = double(10) / 9; // 10 is CAST as a double
```

casting!

Math Functions

C++ offers many mathematical functions via its cmath library:

- to access these functions, you must include this code at the top of your program:

```
#include <cmath> // provides many math-related functions
```

Some useful functions:

- acos(x) // arccosine

```
const double PI = acos(-1); // pi to 15 decimal places
```

- sqrt(x) // square root of x

```
int n1 = sqrt(25);  // n1 = 5
```

- pow(x, y) // x to the power of y

```
int n2 = pow(2, 5); // n2 = 32
```

- abs(x) // absolute value of x

```
int n3 = abs(-139); // n3 = 139
```

Simple Equations

Assume we have the following variables declared:

```
int x = 5, y = 2;
int z;
```

Convert this equation to a single C++ statement:

$$y = x^2 + 2x + 1$$

Possible solutions:

```
y = x*x + 2*x + 1;
y = pow(x, 2) + 2*x + 1; // assuming you #include <cmath>
```

Simple Equations

Assume we have the following variables declared:

```
int x = 5, y = 2;
int z;
```

Convert this equation to a single C++ statement:

$$z = \frac{x^3}{y^2}$$

Possible solutions:

```
z = x * x * x / (y * y);
z = x * x * x / y / y;
z = pow(x, 3) / pow(y, 3); // assuming you #include <cmath>
```

Simple Equations

Assume we have the following variables declared:

```
int x = 5, y = 2;
int z;
```

Convert this equation to a single C++ statement:

$$y = \sqrt{x + z^2}$$

Possible solutions (both require `#include <cmath>`):

```
y = sqrt(x + z*z);
y = sqrt(x + pow(z, 2));
```

Simple Equations

Assume we have the following variables declared:

```
int x = 5, y = 2;
int z;
```

Convert this equation to a single C++ statement:

$$x = \cos\ y\ + \sin\ x$$

Possible solution:

```
x = cos(y) + sin(x); // assumes you #include <cmath>
```