Basic Elements II

DM2111
C++ Programming

Introduction

Introduction	Array and Strings
Problem solving	Array and Strings
Basic elements of C++	Pointers
Basic elements of C++	Pointers
Logic and branching	I/O operations
Repetition	Structs
Functions	Others
Functions	

Agenda

- Expressions
- Operator Types
- Arithmetic Operators
- Assignment Operators
- Increment / Decrement Operators
- Logical Operators
- Comparison Operators
- Operator Precedence
- Type Conversion

Operators, Operands and Expressions

An expression is a sequence of *operators* and their *operands*, that specifies a computation.
http://en.cppreference.com/w/cpp/language/expressions

Every expression yields a result.

```
2 + 3;
"C++";
x = 42;
std::cout << "Hello World";
C++;
;
```

Operator
Operands

null statement

Operator Types

Unary operator – one operand

```
~expr
&expr
expr--
+expr
```

Binary operator – two operands

```
expr1 = expr2
expr1 - expr2
expr1 | expr2
expr1 < expr2
```

Ternary operator - three operands (guess which one?)

Arithmetic Operators

```
+a        unary plus
-a        unary minus

a * b     multiplication
a / b     division
a % b     modulo

a + b     addition
a - b     subtraction
```

- Can be applied to any of the arithmetic types
- They group left to right if precedence levels are same

```
int i;
i = j * k / l;          // All operators have same precedence
i = ((j * k) / l);      // The order of evaluation, left to right
```

Arithmetic Operators

```
~a        bitwise NOT

a << b    left shift
a >> b    right shift

a & b     bitwise AND
a ^ b     bitwise XOR
a | b     bitwise OR
```

Bitwise operators

- Operands are of integral type
- Operators work on the bit level
- Recommend to work on unsigned values

Assignment Operators

```
=    assignment
```

The left hand operand must be a nonconst lvalue

```
int i, j;
1024 = i;       // Error: Literals are lvalues
i + j = 512;    // Error: Expressions are lvalues
i = 256;        // OK
```

lvalue : An object that persists beyond the expression

rvalue : A temporary value that does not persist after the expression

Only lvalues can be on the left side of assignment

lvalues and rvalues can be on the right side

Compound Assignment Operators

```
a += b    addition assignment
a -= b    subtraction assignment
a *= b    multiplication assignment
a /= b    division assignment
a %= b    modulo assignment

a &= b    bitwise AND assignment
a |= b    bitwise OR assignment
a ^= b    bitwise XOR assignment
a <<= b   bitwise left shift assignment
a >>= b   bitwise right shift assignment
```

Shortcut to do operation and assignment

```
int i;
i = i + 1;    // Add 1 to i, and assign the result back to i
i += 1;       // Same as above
```

Increment / Decrement Operators

```
++a      pre-increment
--a      pre-decrement
a++      post-increment
a--      post-decrement
```

Shortcut for adding or subtracting by 1

```
int i = 0, j;
j = ++i;       // i = 1, j = 1. Incremented value is assigned
j = i++;       // i = 2, j = 1. unincremented value is assigned
```

Prefix returns changed value

Postfix returns unchanged copy of the value. Hence we don't use postfix on objects, as it performs unnecessary copying.

tl;dr - Use prefix, always.

Logical Operators

```
!a        Negation
a && b    AND
a || b    Inclusive OR
```

Returns values of type `bool` (true or false)

0 is `false`, `true` otherwise

```
0       // False
'0'     // True: '0' is not a zero, has value of 48
1       // True: non-zero
'\0'    // False: \0 is a zero value
-1      // True: non-zero
```

Logical Operators

- ! (Not) Operator

<expr>	!<expr>
true (nonzero)	false (0)
false (0)	true (1)

Logical Operators

- && (And) Operator

<expr1>	<expr2>	<expr1> && <expr2>
false	false	false
false	true	false
true	false	false
true	true	true

Logical Operators

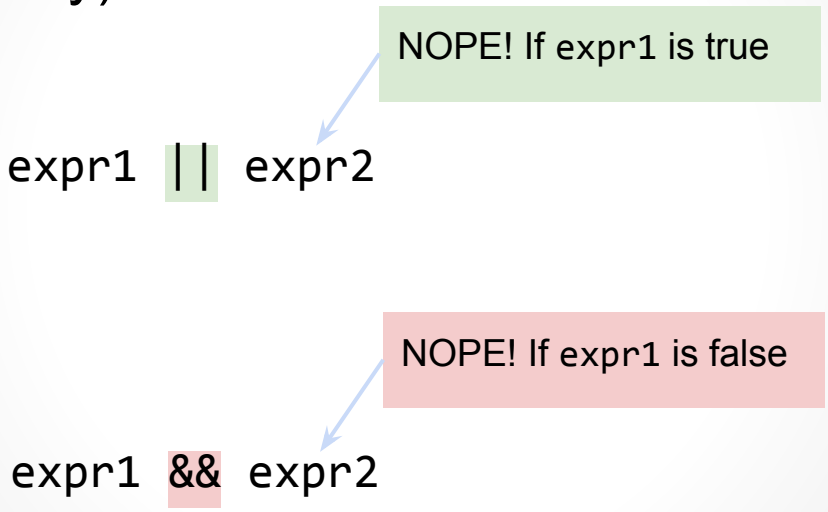
- || (Or) Operator

<expr1>	<expr2>	<expr1> || <expr2>
false	false	false
false	true	true
true	false	true
true	true	true

Logical Operators

AND and OR operators evaluate left operand first

- If left operand determines result, right operand is not evaluated
- Short-circuit (Lazy) Evaluation



Programmer's girlfriend -

"Are you going to sit and type in front of that thing all day or are you going out with me?"

Programmer - "Yes"

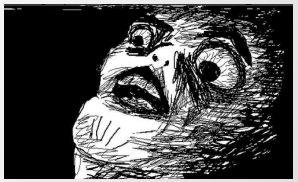
Comparison Operators

```
a == b        Equal to
a != b        Not equal to
a < b         Smaller than
a > b         Greater than
a <= b        Smaller than or equal to
a >= b        Greater than or equal to
```

Returns values of type `bool` (true or false)

They do not chain together

```
int i = 50;
10 < i < 20;        // Check if i is between 10 and 20
((10 < 50) < 20)    // Order of evaluation, left associative
(1 < 20)            // True result, converted to int
1                   // Expression returns true
```



Don’t be lazy, do it properly

```
(10 < i) and (i < 50)
```

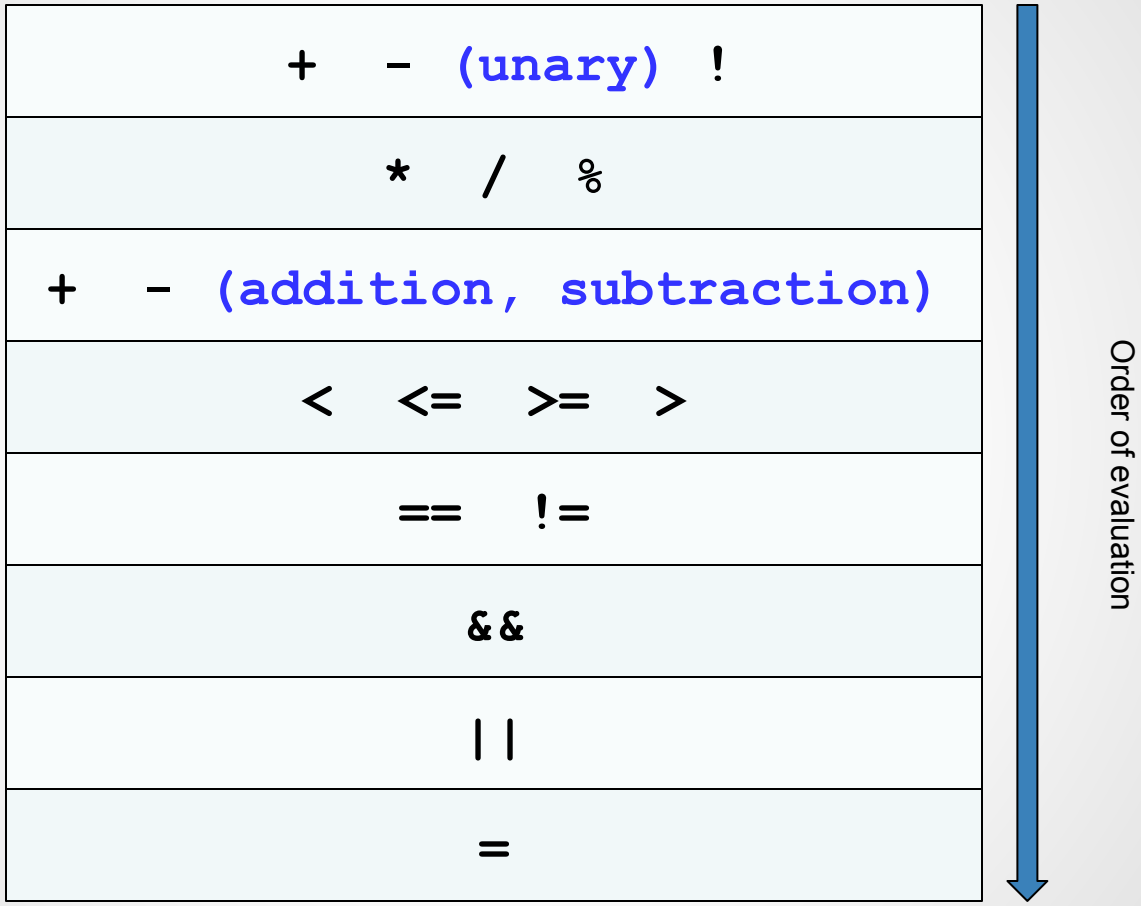
Comparison Operators

Do not confuse == with =

== is comparison
= is assignment

```
int i = 0, j = 10;
i == j;         // Returns false, 0 is not 10
i = j;          // Returns true, i has a value of 10
```

Operator Precedence



The list is much bigger than this. Go look it up

Operator Precedence

Operators with higher precedence gets evaluated first

```
4 / 3 + 2
(4 / 3) + 2

-26 / (26/2) + 45 * 3/5
(((-26) / (26/2)) + ((45 * 3)/5))
```

Parentheses override precedence

Why does the assignment operator have a low precedence?

Type Conversion

Implicit type conversion

```
5 / 2 returns 2           int / int returns int
5.0f / 2 returns 2.5f     float / int returns float
5 / 2.0f returns 2.5f     int / float returns float
```

The integer is promoted to the same data type before evaluation.
Data types are always promoted to a higher precision.

Explicit type conversion

```
static_cast<float> (5) / 2 returns 2.5f
(float) 5 / 2 returns 2.5f   Old way of conversion
```

Convert one or both of the operands to a higher precision in order to get the precision you need.