C Operators

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

Recall

- Data Types
- Constants

Topic for today

- C Operators

Arithmetic Operators

Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2

Arithmetic Operators

Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

A++, post-increment, first using the current value of variable A in the expression, then increment A for use in the next statement.

++A, pre-increment, first increment value of A, then use the new value of A in the current expression.

Relational Operators

- ==
- !=
- >
- >=
- <
- <=

Relational Operators

Demo of Relational Operators

relationOpDemo.c

Logical Operators

- Assume variable A holds 1 and variable B holds 0, then:
- && → Logical AND operator.
  - If both the operands are non-zero, then condition becomes true. E.g. (A && B) is false.
- || → Logical OR Operator.
  - If any of the two operands is non-zero, then condition becomes true. E.g. (A || B) is true.

Logical Operators

- Assume variable A holds 1 and variable B holds 0, then:
- ! → Logical NOT Operator.
  - Use to reverses the logical state of its operand.
  - If a condition is true then Logical NOT operator will make false.   !(A && B) is true.

Bitwise Operators

- &
- Binary AND Operator copies a 1-bit to the result if it exists in **both** operands.
- E.g.
  – Assume if A = 60; and B = 13;

  A = 0011 1100 → **in 2's complement**

  B = 0000 1101

A&B= 0000 1100, which is 12_{10}

Bitwise Operators

- |
- Binary OR Operator copies a 1-bit to the result if it exists in **either** operands.
- E.g.
  - Assume if A = 60; and B = 13;

  A = 0011 1100 →**in 2's complement**

  B = 0000 1101

A|B = 0011 1101, which is 61_{10}

Bitwise Operators

- ^
- Binary XOR Operator copies the 1-bit if it is set in one operand but **not both.**
- E.g.
  - Assume if A = 60; and B = 13;

A = 0011 1100 → **in 2's complement**

B = 0000 1101

A^B = 0011 0001, which is 49_{10}

Bitwise Operators

- ~
- Binary NOT Operator has the effect of 'flipping' bits. (Unary operator)
- E.g.
  - Assume if A = 60;
  - A = 0011 1100 → **in 2's complement**

  ~A = 1100 0011, which is -61_{10}

Bitwise Operators

- <<
- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
- E.g.
  - Assume if A = 30;

A = 0001 1110 → **in 2's complement**

A<<2 = 0111 1000, which is 120_{10}

Bitwise Operators

- <<
- If you shift left on an unsigned int by K bits, this is equivalent to multiplying by 2^K.
- Why?
- Thinking about changes in the contribution of each bit after the shift.

Bitwise Operators

- >>
- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
- E.g.
  - Assume if A = 60;

A = 0011 1100 → **in 2's complement**

A>>2 = 0000 1111, which is 15_{10}

Bitwise Operators

- >>
- When shifting to the right for unsigned int, bits fall off the least significant end, and 0's are shifted in from the most significant end. This is also known as logical right shift .
  - **equivalent to dividing by 2^K.**
- However, with signed int, the story is different.
  - What right shifting does depends on the compiler.

Bitwise Operators

- **Note that**
- a << 2 will not change value of a,
  - instead we should use a = a << 2;
- The same is with a >> 2;

Assignment Operators

- Same as in Java

  =, +=, -=, *=, /=, %=,

- Bitwise Assignment in C

  <<=, >>=, &=, ^=, |=,

Other Operators

- sizeof()
  - Returns the size of an variable or a type.
- &
  - Get the address of a variable, E.g   (&age)
- *
  - dereference a pointer variable,
  - int *p, (assume p is of type int *)
  - ***p** retrieves the value at the memory location that **p** points to (i.e. **p** holds a mem address)

Operator Precedence

- Operator precedence determines the grouping of terms in an expression.
- This affects how an expression is evaluated.
  - Which term is first evaluated, which is next?
  - E.g. x = y = z = 3; → x = (y = (z = 3)) ; → value of the whole expression is 3.
  - How about this?
    - a = 2;
    - c = a ++ + 1; → c = (a++) + 1
    - d = ++a + 1;

Category	Operator	Associativity
Postfix	() [] -> .	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	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	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Operator Precedence

- Associativity defines the order of evaluating a expression when operators have a same precedence.
- E.g.
  - int a = 2;
  - int b= 3;
  - int c = a ++ + ++ b;
  - int d = sizeof + ++a;
  - c ? =6
  - d? =4

Operator Precedence

- Associativity defines the order of evaluating a expression when operators have a same precedence.
- It does NOT hurt to use parens to change precedence in your expected way when you are not sure.

Summary

- Operators
- Bitwise Operators
- Operator Precedence