C Operators, Operands, Expressions & Statements

Hands-on, crash course with code examples

C OPERATORS, OPERANDS, EXPRESSION & STATEMENTS

- **Operators** are symbols which take one or more **operands** or **expressions** and perform arithmetic or logical computations.
- **Operands** are **variables** or **expressions** which are used in conjunction with operators to evaluate the expression.
- Combination of operands and operators form an e**xpression**.
- Expressions are sequences of operators, operands, and punctuators that specify a computation.
- Evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.
- Expression can result in **a value** and can produce **side effects**.
- A **side effect** is a change in the state of the execution environment.

C OPERATORS

- An **expression** is any valid set of literals, variables, operators, operands and expressions that evaluates to a single value.
- This value can be a number, a string or a logical value.
- For instance a = b + c; denotes an expression in which there are 3 operands a, b, c and two operator + and =.
- A **statement**, the smallest independent computational unit, specifies an action to be performed.
- In most cases, statements are executed in sequence.
- The number of operands of an operator is called its **arity**.
- Based on arity, operators are classified as **nullary** (no operands), **unary** (1 operand), **binary** (2 operands), **ternary** (3 operands).

C OPERATORS

Operators	Description	Example Usage
Postfix operators		

( )	Function call operator. A function call is an expression containing the function name followed by the function call operator, (). If the function has been defined to receive parameters, the values that are to be sent into the function are listed inside the parentheses of the function call operator. The argument list can contain any number of expressions separated by commas. It can also be empty.	`sumUp(inum1, inum2)`
		
		`displayName()`
		
		`student(cname, iage, caddress)`
		
		`Calculate(length, wide + 7)`

C OPERATORS

[ ]	Array subscripting operator. A postfix expression followed by an expression in [ ] (brackets) specifies an element of an array. The expression within the brackets is referred to as a subscript. The first element of an array has the subscript zero.	`#include <stdio.h>`
		
		`int main(void)`
		`{`
		`  int a[3] = { 11, 12, 33 };`
		`  printf("a[0] = %d\n", a[0]);`
		`  return 0;`
		`}`

.	Dot operator used to access class, structure, or union members type.	`objectVar.memberOfStructUnion`
->	Arrow operator used to access class, structure or union members using a pointer type.	`aptrTo->memberOfStructUnion`

1. Example 1: function call operator
2. Example 2: array subscripting operator
3. Example 3: pointer dot and arrow operator

C OPERATORS

Unary Operators		
+	Unary plus maintains the value of the operand. Any plus sign in front of a constant is not part of the constant.	`+aNumber`
–	Unary minus operator negates the value of the operand. For example, if `num` variable has the value 200, `-num` has the value -200. Any minus sign in front of a constant is not part of the constant.	`–342`

C OPERATORS

You can put the ++/-- before or after the operand. If it appears before the operand, the operand is incremented/decremented. The incremented value is then used in the expression. If you put the ++/-- after the operand, the value of the operand is used in the expression before the operand is incremented/decremented.		
++	Post-increment. After the result is obtained, the value of the operand is incremented by 1.	`aNumber++`
--	Post-decrement. After the result is obtained, the value of the operand is decremented by 1.	`aNumber--`
++	Pre-increment. The operand is incremented by 1 and its new value is the result of the expression.	`++yourNumber`
--	Pre-decrement. The operand is decremented by 1 and its new value is the result of the expression.	`--yourNumber`

C OPERATORS

&	The address-of operator (&) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the register storage-class specifier. The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.	`&addressOfData`
*	The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.	`*aPointerTo`

C OPERATORS

sizeof	sizeof operator for expressions	**sizeof** 723
sizeof()	sizeof operator for types	**sizeof**(char)
(*type*) cast-expression	Explicit conversion of the type of an object in a specific situation. This is also call automatic promotion.	(**float**)i

Program example: plus, minus, address-of, pre and post increment/decrement, sizeof() and type promotion

C OPERATORS

Multiplicative Operators		
*	The multiplication operator causes its two operands to be multiplied.	`p = q * r;`
/	The division operator causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules: 1. The result of division by 0 is undefined according to the ANSI C standard. The Microsoft C compiler generates an error at compile time or run time. 2. If both operands are positive or unsigned, the result is truncated toward 0. 3. If either operand is negative, whether the result of the operation is the largest integer less than or equal to the algebraic quotient or is the smallest integer greater than or equal to the algebraic quotient is implementation defined.	`a = b / c;`
%	The result of the remainder operator is the remainder when the first operand is divided by the second. When the division is inexact, the result is determined by the following rules: 1. If the right operand is zero, the result is undefined. 2. If both operands are positive or unsigned, the result is positive. 3. If either operand is negative and the result is inexact, the result is implementation defined.	`x = y % z;`

C OPERATORS

Addition and subtraction Operators		
+	addition	`d = e + f`
–	subtraction	`r = s – t;`

- The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.
- The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion.
- Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Program example: assignment, add, subtract, multiply, divide and modulus

C OPERATORS

- For relational expression, `0` is `FALSE`, `1` is `TRUE`.
- Any numeric value is interpreted as either `TRUE` or `FALSE` when it is used in a C / C++ expression or statement that is expecting a logical (true or false) value. The rules are:

1. A value of `0` represents `FALSE`.
2. Any non-zero (including negative numbers) value represents `TRUE`.

Program example: true, false and negate

C OPERATORS

Relational Inequality Operators		
The relational operators compare two operands and determine the validity of a relationship.		
<	Specifies whether the value of the left operand is less than the value of the right operand. The type of the result is `int` and has the value 1 if the specified relationship is true, and 0 if false.	`i < 7`
>	Specifies whether the value of the left operand is greater than the value of the right operand. The type of the result is `int` and has the value 1 if the specified relationship is true, and 0 if false.	`j > 5`
<=	Specifies whether the value of the left operand is less than or equal to the value of the right operand. The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false.	`k <= 4`
>=	Specifies whether the value of the left operand is greater than or equal to the value of the right operand. The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false.	`p >= 3`

C OPERATORS

Relational Equality Operators		
The equality operators, like the relational operators, compare two operands for the validity of a relationship.		

==	Indicates whether the value of the left operand is equal to the value of the right operand. The type of the result is `int` and has the value 1 if the specified relationship is true, and 0 if false. The equality operator (==) should not be confused with the assignment (=) operator. For example:
	if (x == 4) evaluates to true (or 1) if x is equal to four.
	while
	if (x = 4) is taken to be true because (x = 4) evaluates to a nonzero value (4). The expression also assigns the value 4 to x.
		`nChoice == 'Y'`

!=	Indicates whether the value of the left operand is not equal to the value of the right operand. The type of the result is `int` and has the value 1 if the specified relationship is true, and 0 if false.	`nChoice != 'Y'`

C OPERATORS

Expressions	Evaluates as
`(3 == 3) && (4 != 3)`	True (1) because both operands are true
`(4 > 2) || (7 < 11)`	True (1) because (either) one operand/expression is true
`(3 == 2) && (7 == 7)`	False (0) because one operand is false
`! (4 == 3)`	True (1) because the expression is false
`NOT(FALSE) = TRUE`	
`NOT(TRUE) = FALSE`	

Program example: less-than, greater-than, less-than and equal-to, greater-than and equal-to, not-equal, equal and assignment operators

C OPERATORS

Logical NOT Operator (unary)		
!	Logical not operator. Produces value 0 if its operand or expression is true (nonzero) and the value 1 if its operand or expression is false (0). The result has an `int` type. The operand must be an integral, floating, or pointer value.	`!(4 == 2)` `!x`

Operand (or expression)	Output
`!0`	`1 ( T )`
`!1`	`0 ( F )`

C OPERATORS

Logical AND Operator		
&&	Indicates whether both operands are true. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have an arithmetic or pointer type. The usual arithmetic conversions on each operand are performed. The logical `AND` (`&&`) should not be confused with the bitwise `AND` (`&`) operator. For example: `1 && 4` evaluates to `1` (`True && True = True`) while `1 & 4` (`0001 & 0100 = 0000`) evaluates to `0`	`x && y`

Operand1	Operand2	Output
0	0	0 (F)
0	1	0 (F)
1	0	0 (F)
1	1	1 (T)

C OPERATORS

Logical OR Operator		
`||`	Indicates whether either operand is true. If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed. The logical `OR` (`||`) should not be confused with the bitwise `OR` (`|`) operator. For example: `1 || 4` evaluates to `1` (or `True || True = True`) while `1 | 4` (`0001 | 0100 = 0101`) evaluates to `5`	`x || y`

Operand1	Operand2	Output
0	0	0 (F)
0	1	1 (T)
1	0	1 (T)
1	1	1 (T)

Program example: Logical-AND, logical-OR, logical-NOT, XOR operators

C OPERATORS

Conditional Operator (ternary)		
`?:`	The first operand/expression is evaluated, and its value determines whether the second or third operand/expression is evaluated: 1. If the value is true, the second operand/expression is evaluated. 2. If the value is false, the third operand/expression is evaluated. The result is the value of the second or third operand/expression. The syntax is: `First operand ? second operand : third operand`	`size != 0 ? size : 0`

Program example: tenary conditional operator

C OPERATORS

- The compound assignment operators consist of a binary operator and the simple assignment operator.
- They perform the operation of the binary operator on both operands and store the result of that operation into the left operand.
- The following table lists the simple and compound assignment operators and expression examples:

C OPERATORS

Simple Assignment Operator		
=	▪ The simple assignment operator has the following form: *lvalue* = *expr* ▪ The operator stores the value of the right operand *expr* in the object designated by the left operand *lvalue*. ▪ The left operand must be a modifiable lvalue. ▪ The type of an assignment operation is the type of the left operand.	`i = 5 + x;`

C OPERATORS

Compound Assignment Operator	Example	Equivalent expression
identifier operator= entity represents identifier = identifier operator entity		
+=	nindex += 3	index = nindex + 3
-=	*(paPter++) -= 1	* paPter = *(paPter ++) - 1
*=	fbonus *= fpercent	fbonus = fbonus * fpercent
/=	ftimePeriod /= fhours	ftimePeriod = ftimePeriod / fhours
%=	fallowance %= 80	fallowance = fallowance % 80
<<=	iresult <<= inum	iresult = iresult << inum
>>=	byleftForm >>= 1	byleftForm = byleftForm >> 1
&=	bybitMask &= 2	bybitMask = bybitMask & 2
^=	itestSet ^= imainTest	itestSet = itestSet ^ imainTest
|=	bflag |= bonBit	bflag = bflag | bonBit

Program example: compound operators

C OPERATORS

Comma Operator

,

A comma expression contains two operands of any type separated by a comma and has left-to-right associativity. The left operand is fully evaluated, possibly producing side effects, and its value, if there is one, is discarded. The right operand is then evaluated. The type and value of the result of a comma expression are those of its right operand, after the usual unary conversions. In some contexts where the comma character is used, parentheses are required to avoid ambiguity. The primary use of the comma operator is to produce side effects in the following situations:

1. Calling a function.
2. Entering or repeating an iteration loop.
3. Testing a condition.
4. Other situations where a side effect is required but the result of the expression is not immediately needed.

The use of the comma token as an operator is distinct from its use in function calls and definitions, variable declarations, enum declarations, and similar constructs, where it acts as a separator.

Because the comma operator discards its first operand, it is generally only useful where the first operand has desirable side effects, such as in the initializer or the counting expression of a for loop.

C OPERATORS

- The following table gives some examples of the uses of the comma operator.

Statement	Effects
`for (i=0; i<4; ++i, func());`	A `for` statement in which `i` is incremented and `func()` is called at each iteration.
`if (func(), ++i, i>1 ) { /* ... */ }`	An `if` statement in which function `func()` is called, variable `i` is incremented, and variable `i` is tested against a value. The first two expressions within this comma expression are evaluated before the expression `i>1`. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the `if` statement is processed.
`func1((++iaArg, func2(iaArg)));`	A function call to `func1()` in which `iaArg` is incremented, the resulting value is passed to a function `func2()`, and the return value of `func2()` is passed to `func1()`. The function `func1()` is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list.
`int inum=3, inum2=7, inum3 = 2;`	Comma acts as separator, not as an operator.

Program example: comma operator

C OPERATORS

Bitwise (complement) NOT Operators	
~	The ~ (bitwise negation) operator yields the bitwise (one) complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue (left value). The symbol used called tilde.

Suppose `byNum` variable represents the decimal value `8`. The 16-bit binary representation of `byNum` is:

```
00000000 00001000
```

The expression `~byNum` yields the following result (represented here as a 16-bit binary number):

```
11111111 11110111
```

Note that the ~ character can be represented by the trigraph ??-.
The 16-bit binary representation of `~0` (which is `~00000000 00000000`) is:

```
11111111 11111111
```

C OPERATORS

Bitwise Shift Operators		
<<	Left shift operator, shift their first operand left (<<) by the number of positions specified by the second operand.	`nbits << nshiftSize`
>>	Right shift operator, shift their first operand right (>>) by the number of positions specified by the second operand.	`nbits >> nshiftSize`

- Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.
- For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is unsigned, they are set to 0. Otherwise, they are filled with copies of the sign bit. For left-shift operators without overflow, the statement:

```
expression1 << expression2
```

- is equivalent to multiplication by $2^{expression2}$. For right-shift operators:

```
expression1 >> expression2
```

- is equivalent to division by $2^{expression2}$ if expression1 is unsigned or has a nonnegative value.
- The result of a shift operation is undefined if the second operand is negative, or if the right operand is greater than or equal to the width in bits of the promoted left operand.
- Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

C OPERATORS

For example, if `nleftShiftOp` and `nrightShiftOp` has the value 2030, the bit pattern (in 16-bit format) of `nleftShiftOp` and `nrightShiftOp` respectively is:

```
00000111 11101110
```

The expression `nleftShiftOp << 3` yields:

```
00111111 01110000

                0  0  0  0  0  1  1  1  1  1  1  0  1  1  1  0
                                                ?  ?  ?  ?
                0  0  1  1  1  1  1  1  0  1  1  1  0  0  0  0
```

The expression `nrightShiftOp >> 3` yields:

```
00000111 11101110

                0  0  0  0  0  1  1  1  1  1  1  0  1  1  1  0
                                                ?  ?  ?  ?
                0  0  0  0  0  0  0  0  1  1  1  1  1  1  0  1
```

Program example: bitwise shift left, bitwise shift right and bitwise-NOT operators

C OPERATORS

Bitwise AND Operator	
&	▪ The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0. ▪ Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands. ▪ Because the bitwise AND operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise AND operator. ▪ The bitwise AND (&) should not be confused with the logical AND. (&&) operator. For example: `1 & 4` evaluates to `0` (`0001 & 0100 = 0000`) while `1 && 4` evaluates to true [`True && True = True`]

C OPERATORS

The following example shows the values of num1, num2, and the result of num1 & num2 represented as 16-bit binary numbers:

Bit pattern num1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	0	0
Bit pattern num2	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1
Bit pattern num1 & num2	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0

Bitwise AND operation		
Operand 1	**Operand 2**	**Operand 1 & Operand 2**
0	0	0
0	1	0
1	0	0
1	1	1

C OPERATORS

Bitwise XOR Operator

^

- The bitwise exclusive `OR` operator (in EBCDIC, the `^` symbol is represented by the ¬ symbol) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are `1`'s or both bits are `0`'s, the corresponding bit of the result is set to `0`. Otherwise, it sets the corresponding result bit to `1`.
- Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue (left value).
- Because the bitwise exclusive `OR` operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise exclusive `OR` operator. Note that the `^` character can be represented by the trigraph ??'. The symbol used called caret.

C OPERATORS

The following example shows the values of num1, num2, and the result of num1 ^ num2 represented as 16-bit binary numbers:

Bit pattern num1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	0	0
Bit pattern num2	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1
Bit pattern num1 ^ num2	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1

Bitwise XOR operation		
Operand 1	**Operand 2**	**Operand 1 ^ Operand 2**
0	0	0
0	1	1
1	0	1
1	1	0

C OPERATORS

Bitwise (Inclusive) OR Operator

|

- The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.
- Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.
- Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator. Note that the | character can be represented by the trigraph ??!.
- The bitwise OR (|) should not be confused with the logical OR (||) operator. For example:

`1 | 4` evaluates to `5` (`0001 | 0100 = 0101`) while `1 || 4` (`True || True = True`) evaluates to true

C OPERATORS

The following example shows the values of num1, num2, and the result of num1 | num2 represented as 16-bit binary numbers:

Bit pattern num1	0	0	0	0	0	1	1	1	1	1	1	0	0	1	0	0
Bit pattern num2	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1
Bit pattern num1 ^ num2	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1

Bitwise (Inclusive) OR operation		
Operand 1	**Operand 2**	**Operand 1 | Operand 2**
0	0	0
0	1	1
1	0	1
1	1	1

Program example: bitwise-AND, bitwise-OR and eXclusive-OR (XOR) operators

OPERATOR PRECEDENCE

- Consider the following arithmetic operation:

  - left to right

  ```
  6 / 2 * 1 + 2 = 5
  ```

  - right to left

  ```
  6/2 * 1 + 2 = 1
  ```

  - using parentheses

  ```
  = 6 / (2 * 1) + 2
  = (6 / 2) + 2
  = 3 + 2
  = 5
  ```

Inconsistent answers!

OPERATOR PRECEDENCE

- Operator precedence: a rule used to clarify unambiguously which operations (operator and operands) should be performed first in the given (mathematical) expression.
- Use precedence levels that conform to the order *commonly* used in mathematics.
- However, parentheses take the highest precedence and operation performed from the innermost to the outermost parentheses.

OPERATOR PRECEDENCE

- Precedence and associativity of C operators affect the grouping and evaluation of operands in expressions.
- Is meaningful only if other operators with higher or lower precedence are present.
- Expressions with higher-precedence operators are evaluated first.

OPERATOR PRECEDENCE

Precedence and Associativity of C Operators		
Symbol	**Type of Operation**	**Associativity**
`[ ] ( ) . ->` postfix ++ and postfix --	Expression	Left to right
prefix ++ and prefix -- `sizeof` `& * + - ~ !`	Unary	Right to left
typecasts	Unary	Right to left
`* / %`	Multiplicative	Left to right
`+ -`	Additive	Left to right
`<< >>`	Bitwise shift	Left to right
`< > <= >=`	Relational	Left to right
`== !=`	Equality	Left to right
`&`	Bitwise-AND	Left to right
`^`	Bitwise-exclusive-OR	Left to right
`|`	Bitwise-inclusive-OR	Left to right
`&&`	Logical-AND	Left to right
`||`	Logical-OR	Left to right
`? :`	Conditional-expression	Right to left
`= *= /= %= += -= <<= >>= &= ^= |=`	Simple and compound assignment	Right to left
`,`	Sequential evaluation	Left to right

OPERATOR PRECEDENCE

- The precedence and associativity (the order in which the operands are evaluated) of C operators.
- In the order of precedence from highest to lowest.
- If several operators appear together, they have equal precedence and are evaluated according to their associativity.
- All simple and compound-assignment operators have equal precedence.

OPERATOR PRECEDENCE

- Operators with equal precedence such as + and −, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right.
- The direction of evaluation does not affect the results of expressions that include more than one multiplication (*), addition (+), or binary-bitwise (& | ^) operator at the same level.

OPERATOR PRECEDENCE

- e.g:

```
3 + 5 + (3 + 2) = 13 – right to left
(3 + 5) + 3 + 2 = 13 – left to right
3 + (5 + 3) + 2 = 13 – from middle
```

- Order of operations is not defined by the language.
- The compiler is free to evaluate such expressions in any order, if the compiler can guarantee a consistent result.

OPERATOR PRECEDENCE

- Only the sequential-evaluation (`,`), logical-`AND` (`&&`), logical-`OR` (`||`), conditional-expression (`?:`), and function-call operators constitute sequence points and therefore guarantee a particular order of evaluation for their operands.
- The sequential-evaluation operator (`,`) is guaranteed to evaluate its operands from left to right.
- The comma operator in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.

OPERATOR PRECEDENCE

- Logical operators also guarantee evaluation of their operands from left to right.
- But, they evaluate the smallest number of operands needed to determine the result of the expression.
- This is called "short-circuit" evaluation.
- Thus, some operands of the expression may not be evaluated.

OPERATOR PRECEDENCE

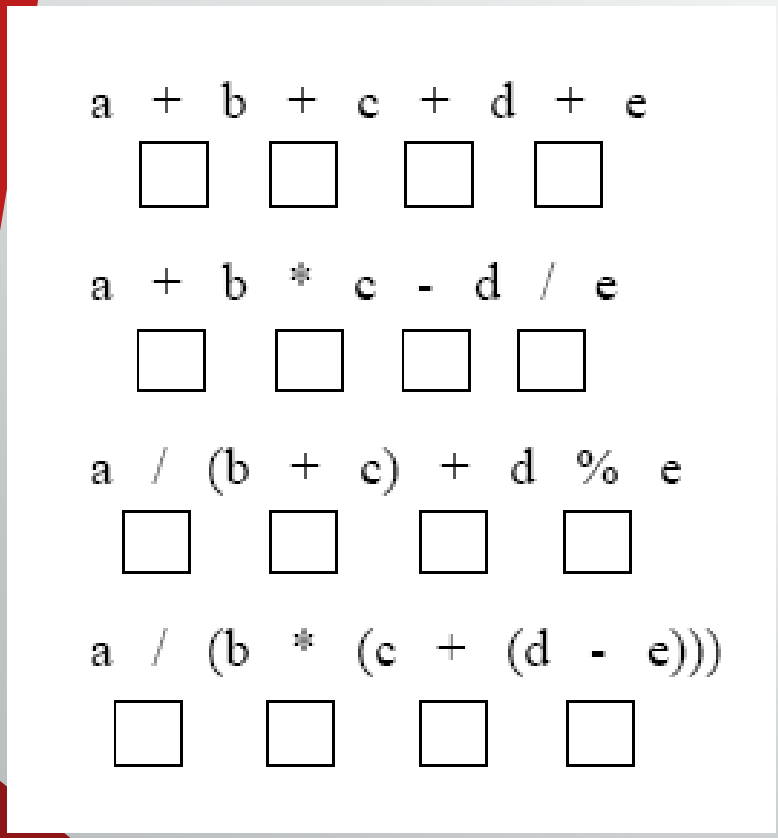
- For example:

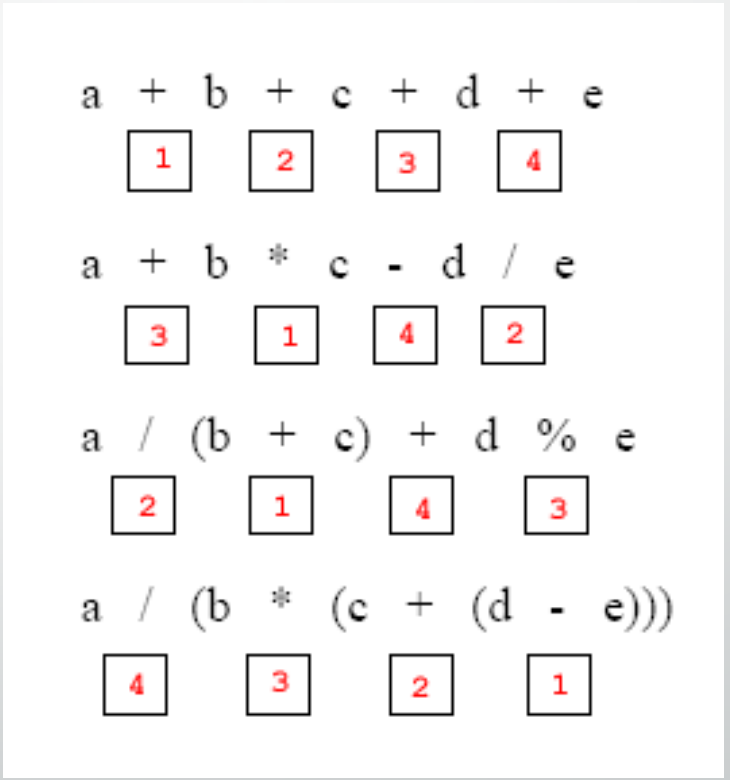
```
x  &&  y++
```

- The second operand, `y++`, is evaluated only if `x` is true (nonzero).
- Thus, `y` is not incremented if `x` is false (`0`).

OPERATOR PRECEDENCE

- Label the execution order for the following expressions





OPERATOR PRECEDENCE

- Convert the following operations to C expression

a. $rate^2 + delta$

b. $2(salary + bonus)$

c. $\dfrac{1}{time + 3mass}$

d. $\dfrac{a - 7}{t + 9v}$

a. (rate*rate) + delta
b. 2*(salary + bonus)
c. 1/(time + (3*mass))
d. (a - 7) / (t + (9 * v))

End of C Operators, Operands, Expressions & Statements