PROGRAMMAZIONE PROCEDURALE

A.A. 2022/2023

EXPRESSIONS AND OPERATORS

EXPRESSIONS

- An expression consists of a sequence of constants (literals), identifiers, and operators that the program evaluates by performing the operations indicated.
- The expression's purpose in the program may be to obtain the resulting *value*, or to produce *side effects* of the evaluation, or both.
- A single constant, a string literal, or the identifier of an object or function is in itself an expression.

SIDE EFFECTS

- An expression is said to have a side effect if it modifies some state outside its local environment, that is to say has an observable interaction with the outside world besides returning a value

PS: WRONG, not the right way to test conditions

```
if (a == 5) {
    printf("Hello World!\n");
}
```

Returns a value

```
if (a = 5) {
    printf("Hello World!\n");
}
```

Returns a value AND has a side effect on a

TYPE OF AN EXPRESSION

- Every expression has a type. An expression's type is the type of the value that results when the expression is evaluated. If the expression yields no value, it has the type *void*. If **a** has type **int**

Expression	Type
`'\n'`	`int`
`a + 1`	`int`
`a + 1.0`	`double`
`a < 77.7`	`int`
`"A string literal."`	`char *`
`abort()`	`void`
`sqrt(2.0)`	`double`

FUNDAMENTAL PRINCIPLES

- A few fundamental principles that will help you understand how C expressions are evaluated. The
  - ✓ *precedence* and
  - ✓ *associativity*
  - ✓ *sequence points* and
  - ✓ *lvalues and rvalues*

LVALUES

LVALUES

- An ***lvalue*** is an expression that designates an object.
- The term *object* refers to a location in memory whose contents can represent values.
- The simplest example is the name of a variable.
- The initial “l” in the term originally meant “left”: because an lvalue designates an object, **it can appear on the left side of an assignment operator (but also on the right of it)**, as in

  leftExpression = rightExpression
- Expressions that represent a value without designating an object are called, by analogy, ***rvalues***.
- **An *rvalue* is an expression that can appear on the right side of an assignment operator, but not the left.**

EXAMPLES

7	rvalue	~~7= a~~	a= 7
a	lvalue	a= 7	
a+1	rvalue	~~a + 1= 8~~	a= a +1
‘c’	rvalue	~~‘c’= a~~	a= ‘c’
a+b	rvalue	~~a+b= 11~~	a= a+b

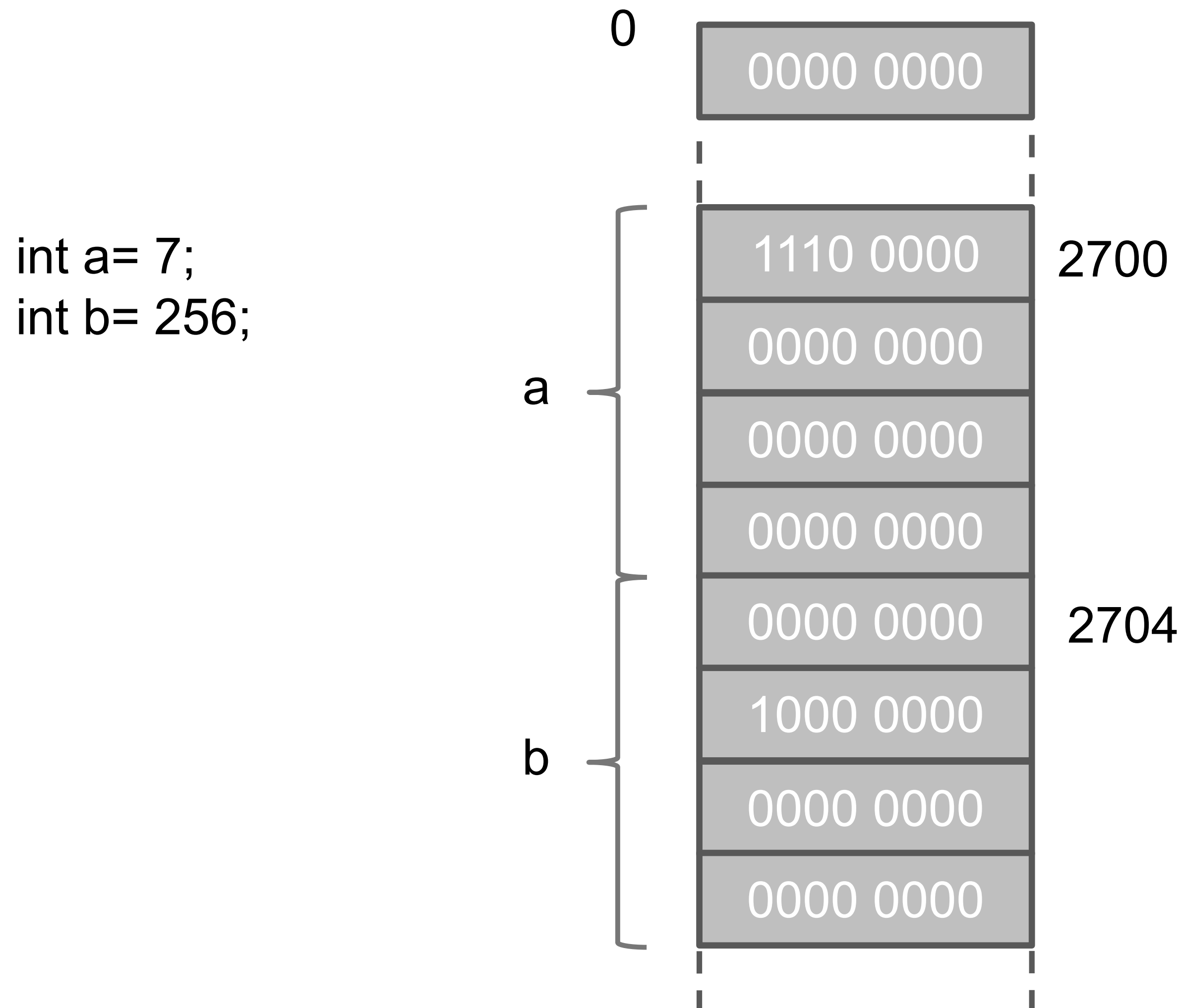
LVALUES CAN BE ALSO ON THE RIGHT OF =

a= 7   b= a

1. When a variable is on the left of =, we modify its value with the value of the expression on the right of =
2. When a variable is on the right of =, we take its value stored in memory

VARIABLES

- Variables are lvalues: they are containers of values
- Anything whose value can be changed in memory



CONSTANTS AND LVALUES

- An object may be declared as constant. If this is the case, you can't use it on the left side of an assignment, even though it is an lvalue, as the following example illustrates:

```
int a = 1;
const int b = 2;
b = 20;                    // Error: b is declared as const int.
```

LEFT VALUES AND ASSIGNMENTS

- The left operand of an assignment, but also a ***modifiable lvalue***
- A **modifiable** lvalue is an lvalue that is not declared as a const-qualified type
- Use constants when you want to be sure a value of a variable will not be changed during the execution of a program
- You have to give a value to it as soon as it is defined, since it is not modifiable after.

```
const double pi= 3.14;
```

```
const double pi;
pi= 3.14;
```

OPERATORS PRECEDENCE

OPERATOR PRECEDENCE

- An expression may contain several operators. In this case, the *precedence* of the operators determines which part of the expression is treated as the operand of each operator.
- For example, in keeping with the customary rules of arithmetic, the operators *, /, and % have higher precedence in an expression than the operators + and -.
- For example, the following expression: a – b * c is equivalent to

  a - (b * c).
- If you intend the operands to be grouped differently, you must use parentheses, thus: (a - b) * c

OPERATOR ASSOCIATIVITY

- If two operators in an expression have the same precedence, then their *associativity* determines whether they are grouped with operands in order from left to right, or from right to left.
- For example, arithmetic operators are associated with operands from left to right, and assignment operators from right to left:
- a / b / c
  - ✓(a / b) / c
- a = b = c
  - ✓a = (b = c) → int a= 5; int b= 7; Int c= 9; → int a= 9; int b= 9; Int c= 9;

PRECEDENCE AND ASSOCIATIVITY TABLE

Precedence	Operators	Associativity
1.	Postfix operators: [] () . -> ++ -- (*type name*){*list*}	Left to right
2.	Unary operators: ++ -- ! ~ + - * & sizeof	Right to left
3.	The cast operator: (*type name*)	Right to left
4.	Multiplicative operators: * / %	Left to right
5.	Additive operators: + -	Left to right
6.	Shift operators: << >>	Left to right
7.	Relational operators: < <= > >=	Left to right
8.	Equality operators: == !=	Left to right
9.	Bitwise AND: &	Left to right
10.	Bitwise exclusive OR: ^	Left to right
11.	Bitwise OR: |	Left to right
12.	Logical AND: &&	Left to right
13.	Logical OR: ||	Left to right
14.	The conditional operator: ? :	Right to left
15.	Assignment operators: = += -= *= /= %= &= ^= |= <<= >>=	Right to left
16.	The comma operator: ,	Left to right

COMMENTS TO TABLE

- A few of the operator tokens appear twice in the table.
  - ✓The increment and decrement operators, ++ and --, have a higher precedence when used as postfix operators (as in the expression x++) than the same tokens when used as prefix operators (as in ++x).
- Furthermore, the tokens +, -, *, and & represent both *unary operators* -that is, operators that work on a single operand- and *binary operators*, or operators that connect two operands.

ARITHMETIC OPERATORS

Operator	Meaning	Example	Result
*	Multiplication	x * y	The product of x and y
/	Division	x / y	The quotient of x by y
%	The modulo operation	x % y	The remainder of x divided by y
+	Addition	x + y	The sum of x and y
-	Subtraction	x - y	The difference of x and y
+ (unary)	Positive sign	+x	The value of x
- (unary)	Negative sign	-x	The arithmetic negation of x

ARITHMETIC OPERATORS (2)

- The operands of the arithmetic operators are subject to the following rules:
  - ✓ Only the % operator requires integer operands.
- The operands of all other operators may have any arithmetic type.

ARITHMETIC CONVERSIONS

- The operands are subject to the *usual arithmetic.* The result of *division* with two integer operands is also an integer! Example with
  - ✓short n = -5

Expression	Implicit type conversion	The expression's type	The expression's value
`-n`	Integer promotion.		
`n * -2L`	Integer promotion: the value of `n` is promoted to `long`, because the constant `-2L` has the type `long`.		
`8/n`	Integer promotion.		
`8%n`	Integer promotion.		
`8.0/n`	The value of `n` is converted to the type `double`, because 8.0 has the type `double`.		
`8.0%n`	Error: the modulo operation (%) requires integer operands.		

COMMENTS

- If both operands in a multiplication or a division have the same sign, the result is positive; otherwise, it is negative.
- However, the result of a modulo operation always has the same sign as the left operand.

ASSIGNMENT OPERATORS

- In an assignment operation, the left operand must be a *modifiable lvalue*; in other words, it must be an expression that designates an object whose value can be changed.
- In a *simple assignment* (that is, one performed using the operator =), the assignment operation stores the value of the right operand in this object.
- There are also *compound assignments*, which combine an arithmetic or a bitwise operation in the same step with the assignment.

Operator	Meaning	Example	Result
=	Simple assignment	x = y	Assign x the value of y.
+= -= *= /= %= &= ^= |= <<= >>=	Compound assignment	x *= y	For each binary arithmetic or binary bitwise operator *op*, x *op*= y is equivalent to x = x *op* (y).

COMPOUND ASSIGNMENTS

- A compound assignment is performed by any of the following operators:
  - ✓*= /= %= += -= (arithmetic operation and assignment)
  - ✓<<= >>= &= ^= |= (bitwise operation and assignment)
- Simply, in evaluating a compound assignment expression, the program combines the two operands with the specified operation and assigns the result to the left operand.

INCREMENT AND DECREMENT OPS.

- Each of the tokens ++ and -- represents both a postfix and a prefix operator.

Operator	Meaning	Side effect	Value of the expression
Postfix: *x*++	Increment	Increases the value of *x* by one (like *x* = *x* + 1).	The value of *x*++ is the value that *x* had *before* it was incremented.
Prefix: ++*x*			The value of ++*x* is the value that *x* has *after* it has been incremented.
Postfix: *x*--	Decrement	Decreases the value of *x* by one (like *x* = *x* - 1).	The value of *x*-- is the value that *x* had *before* it was decremented.
Prefix: --*x*			The value of --*x* is the value that *x* has *after* it has been decremented.

COMPARATIVE OPERATORS

- The *comparative operators*, also called the *relational operators* and the *equality operators*, compare two operands and yield a value of type int. The value is 1 if the specified relation holds, and 0 if it does not.

Operator	Meaning	Example	Result (1 = true, 0 = false)
<	Less than	x < y	1 if x is less than y, otherwise 0
<=	Less than or equal to	x <= y	1 if x is less than or equal to y, otherwise 0
>	Greater than	x > y	1 if x is greater than y, otherwise 0
>=	Greater than or equal to	x >= y	1 if x is greater than or equal to y, otherwise 0
==	Equal to	x == y	1 if x is equal to y, otherwise 0
!=	Not equal to	x != y	1 if x is not equal to y, otherwise 0

EXAMPLES

```
int a= 5
int b= 6
int c= 0

c = a < b;        c is equal to 1
c =  a == b;      c is equal to 0
c = b >= a        c is equal to 1
```

COMPARATIVE OPS. AND PREC

- The comparative operators have lower precedence than the arithmetic operators, but higher precedence than the logical operators.
  - ✓a < b && b < c + 1
  - ✓(a < b) && (b < (c + 1))
- Furthermore, the equality operators, == and !=, have lower precedence than the other comparative operators.
  - ✓a < b != b < c
  - ✓(a < b) != (b < c)

LOGICAL OPERATORS

- You can connect expressions using logical operators to form compound conditions: boolean operations AND, OR, and NOT.

Operator	Meaning	Example	Result (1 = true, 0 = false)
&&	logical AND	x && y	1 if each of the operands x and y is not equal to zero, otherwise 0
||	logical OR	x || y	0 if each of x and y is equal to zero, otherwise 1
!	logical NOT	!x	1 if x is equal to zero, otherwise 0

- Like comparative expressions, logical expressions have the type int. The result has the value 1 if the logical expression is true, and the value 0 if it is false.

LOGICAL OPERATORS (2)

- Any operand with a value of 0 is interpreted as false; any value other than 0 is treated as true.

all these conditions are equivalent

```
(a< -0.2) || (a> 0.2)
a< -0.2 || a> 0.2
!(a>= -0.2 && a<= 0.2)
```

The unary operator ! has a higher precedence.

ORDER OF EVALUATION

- The operators && and || have an important peculiarity: their operands are evaluated in order from left to right, and if the value of the left operand is sufficient to determine the result of the operation, then the right operand is not evaluated at all.
  - ✓The operator && evaluates the right operand only if the left operand yields a nonzero value.
  - ✓The operator || evaluates the right operand only if the left operand yields 0.

SHORT CIRCUIT

```c
int a= 1;
int b= 0;
int c;

c= a || ++b;
```

b is not incremented

```c
int a= 1;
int b= 0;
int c;

c= b && ++a;
```

a is not incremented

BITWISE OPERATORS

- The bitwise operators allow you to manipulate individual bits in a byte or in a larger data unit

Operator	Meaning	Example	Result (for each bit position) (1 = set, 0 = cleared)
&	Bitwise AND	x & y	1, if 1 in both x and y 0, if 0 in x or y, or both
|	Bitwise OR	x | y	1, if 1 in x or y, or both 0, if 0 in both x and y
^	Bitwise exclusive OR	x ^ y	1, if 1 either in x or in y, but not in both 0, if either value in both x and y
~	Bitwise NOT (one's complement)	~x	1, if 0 in x 0, if 1 in x

EXAMPLE

Expression (or declaration)	Bit pattern
`int a = 6;`	0 ... 0 0 1 1 0
`int b = 11;`	0 ... 0 1 0 1 1
`a & b`	0 ... 0 0 0 1 0
`a | b`	0 ... 0 1 1 1 1
`a ^ b`	0 ... 0 1 1 0 1
`~a`	1 ... 1 1 0 0 1

a &= 0xFF; // Equivalent a = a & 0xFF; // Clears all except the lowest eight bit

IN THE PROCESSOR

- In memory values are represented in little endian
- In the processor, the same values are translated to big endian
- So, all the operations in the processor are computed in big endian, and when the result is stored back into memory, they are saved in little endian.

SHIFT OPERATORS

- The shift operators transpose the bit pattern of the left operand by the number of bit positions indicated by the right operand.

Operator	Meaning	Example	Result
<<	Shift left	x << y	Each bit value in x is moved y positions to the left.
>>	Shift right	x >> y	Each bit value in x is moved y positions to the right.

- The operands of the shift operators must be integers. Before the actual bit-shift, the integer promotions are performed on both operands. The value of the right operand **must not be negative**, and must be less than the width of the left operand after integer promotion.

EXAMPLE

```
unsigned long n = 0xB,                    // Bit pattern: 0 ... 0 0 0 1 0 1 1
        result = 0;

result = n << 2;                          // 0 ... 0 1 0 1 1 0 0
result = n >> 2;                          // 0 ... 0 0 0 0 0 1 0
```

- A left shift through y bit positions is equivalent to multiplying the left operand by 2^y
- Right shift: $x/2^y$

OTHER OPERATORS

- There are six other operators in C that do not fall into any of the categories described in this round of slides.

Operator	Meaning	Example	Result
`()`	Function call	`log(x)`	Passes control to the specified function, with the specified arguments.
`sizeof`	Storage size of an object or type, in bytes	`sizeof x`	The number of bytes occupied in memory by x.
(*type name*)	Explicit type conversion, or "cast"	`(short) x`	The value of x converted to the type specified.
`?:`	Conditional evaluation	`x ? y : z`	The value of y, if x is true (i.e., nonzero); otherwise the value of z.
`,`	Sequential evaluation	`x,y`	Evaluates first x, then y. The result of the expression is the value of y. nonzero); otherwise the value of z.

CONDITIONAL OPERATOR

- The *conditional operator* is sometimes called the *ternary* or *trinary operator*, because it is the only one that has three operands:

  condition ? *expression 1* : *expression 2*

- The operation first evaluates the condition. Then, depending on the result, it evaluates one or the other of the two alternative expressions.
- There is a sequence point after the *condition* has been evaluated.
- If the result is not equal to 0 (in other words, if the *condition* is true), then only the second operand, *expression 1*, is evaluated, and the entire operation yields the value of *expression 1*. If on the other hand *condition* does yield 0 (i.e., false), then only the third operand, *expression 2*, is evaluated, and the entire operation yields the value of *expression 2*.

EXAMPLE

```
int a= 6
int b= 12;
int c;

c= b > a ? 100: 150;
```

c becomes 100

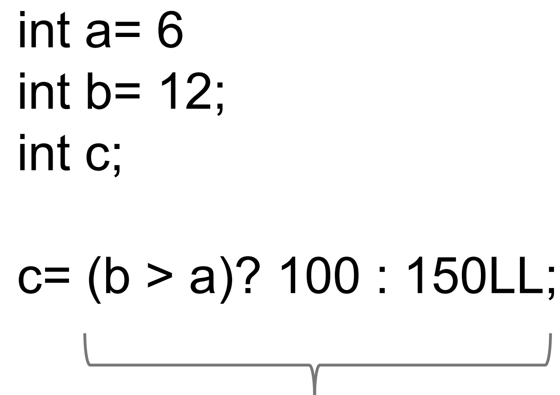
```
int a= 6
int b= 12;
int c;

c= a == b ? 100 : 150;
```

c becomes 150

TYPE OF CONDITIONAL STAT.

- The first operand of the conditional operator, *condition*, must have an arithmetic type
- Both of the alternative expressions have arithmetic types, in which case the result of the complete operation has the type that results from performing the usual arithmetic conversions on these operands.



The whole expression has type **long long**

COMMA OPERATOR

- The comma operator is a binary operator:

  expression 1 , expression 2

- The comma operator ensures sequential processing: first the left operand is evaluated, then the right operand. The result of the complete expression has the type and value of the right operand. The left operand is only evaluated for its side effects; its value is discarded.

EXAMPLE

a becomes 7
c becomes 14

```
int a= 6
int b= 12;
int c;

c= (a++, a + 7);
```

The value of the whole expression is 14

SEQUENCE POINTS AND SIDE EFFECTS

SIDE EFFECTS, SEQUENCE POINTS

- A function or expression is said to have a **side effect** if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world.
- A **sequence point** defines any point in a computer program's execution at which it is guaranteed that
  - ✓all side effects of previous evaluations will have been performed, and
  - ✓no side effects from subsequent evaluations have yet been performed.
- A sequence point is a point in program execution at which all side effects are evaluated before going on to the next step.

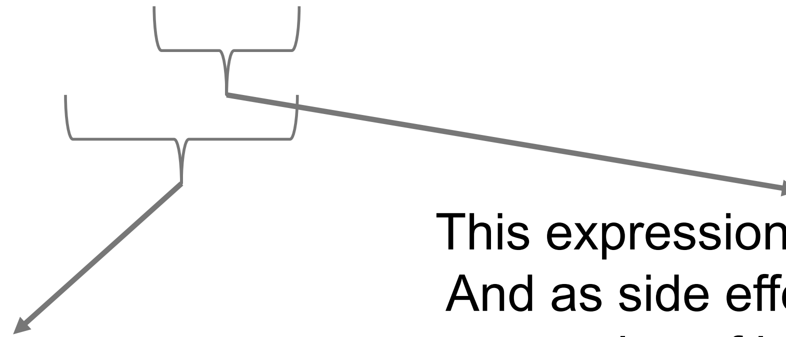
BE CAREFUL

- The modification of the left operand is a side effect of an *assignment expression*.
  - ✓a = b = 5
- The value of the entire assignment expression is the same as the value assigned to the left operand, and the assignment expression has the type of the left operand
- However, unlike its left operand, the assignment expression itself is not an lvalue.

EXAMPLE

```
int a= 6
int b= 12;
int c;

c= (b = a);
```



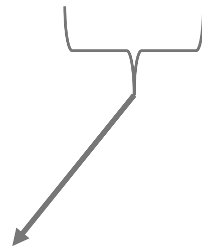
This expression has a value of 6
And as side effect, changes the
value of b in memory

This expression has a value of 6
And as side effect, changes the
value of c in memory

EXAMPLE

```
int a= 6
int b= 12;
int c;

(c= b) = a;
```



This expression has a value of 12
And as side effect, changes the
value of c in memory

However, the compiler returns an error because c = b is an rvalue and not an lvalue: we cannot assign the value of a to it

HOW MANY SIDE EFFECTS?

```
int a= 3;
int b= 4;
a= b++;
```

Expression a= b++ has
two side effects: one on b and one on a

REMINDER

- As a programmer, you must therefore remember not to modify **the same variable** more than once between two consecutive sequence points. An example:

```
i = 1;          // OK.
i = i++;        // Wrong: two modifications of i; behavior is undefined.
```

- Because the assignment and increment operations in the last statement may take place in either order, the resulting value of i is undefined.

```
int i = i++;        // OK, whatever it means, because int i = something
                    // is seen as the creation of i and an then its
                    // initialization. The first "=" is not seen as above
```

SEQUENCE POINTS

- The most important sequence points occur at the following positions:
  - ✓At the end of a command (;)
  - ✓After all the arguments in a function call have been evaluated, and before control passes to the statements in the function
  - ✓At the end of an expression which is not part of a larger expression. For example, each of the three controlling expressions in a **for** statement, the condition of an **if** or **while** statement, the expression in a **return** statement, and initializers (after eval of 5 in int a =5;).
  - ✓After the evaluation of the first operand of each of the following operators:
    - && (logical AND)
    - || (logical OR)
    - ?: (the conditional operator)
    - , (the comma operator)

EXAMPLE

- Thus the expression

  ++i < 100 ? f(i++) : (i = 0)

- is permissible, as there is a sequence point between the first modification of *i* and whichever of the other two modifications is performed.
- Also **a++ && a++** is ok
- **a++ + b++** is ok because in this expression there is one side effect on a and one side effect on b, hence on two different variables

EXERCISE

```
int main()
{
    int i = 0;
    int k = i++, j= i++;
    printf( "%d %d %d\n", k, j, i) ;

    int l= 0;
    int m= ++l, n= ++l;
    printf( "%d %d %d\n", m, n, l) ;

}
```

```
0 1 2
1 2 2
```

```
int main()
{
    int i = 0;
    int k = i++, j= i++;
    printf( "%d %d %d\n", k, j, i) ;

    int l= 0;
    int m= ++l, n= ++l;
    printf( "%d %d %d\n", m, n, l) ;

    int q;
    q= ++q;
    printf( "%d\n", q);
}
```

```
example.c:26:8: warning: multiple unsequenced modifications to 'q' [-Wunsequenced]
    q= ++q;
     ~ ^
1 warning generated.
```