S5_1

Operators

Objectives

- To learn and appreciate
  - Arithmetic Operators
  - Relational and Logical Operators
  - Increment and Decrement Operators
  - Bitwise Operators

Session outcome

- At the end of session student will be able to learn and understand
  - Arithmetic Operators
  - Relational and Logical Operators
  - Increment and Decrement Operators
  - Bitwise Operators

Operators

The different operators are:

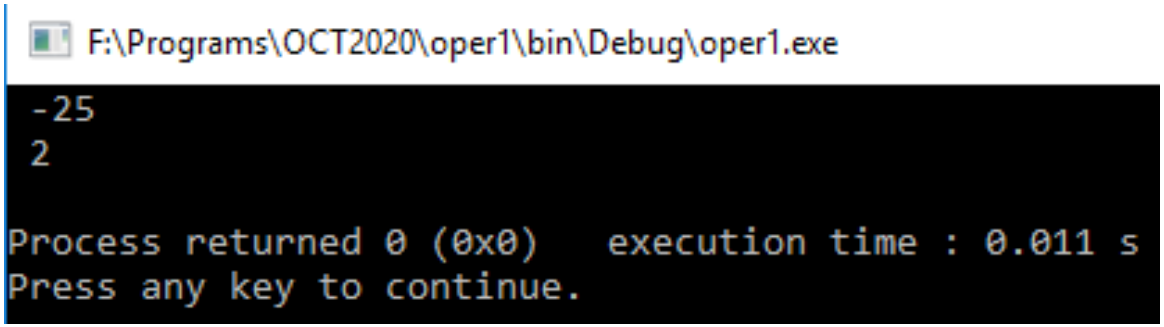
- Arithmetic
- Relational
- Logical
- Increment and Decrement
- Bitwise
- Assignment
- Conditional

Arithmetic Operators

- The binary arithmetic operators are `+`, `-`, `*`, `/` and the modulus operator `%`.
- The / operator when used with integers truncates any fractional part
  - **`E.g. 5/2 = 2 and not 2.5`**
- Therefore % operator produces the remainder when 5 is divided by 2
  - `i.e. 1`
- The % operator **cannot be applied** to float or double
  - E.g. x % y wherein % is the operator and x, y are operands

The **unary** minus operator

```
#include <stdio.h>
int main ()
{
   int a = 25;
   int b = -2;
   printf("%d\n",-a);
   printf("%d\n",-b);
   return 0;
}
```



Working with arithmetic expressions

- **Basic arithmetic operators**: **+, -, *, /, %**
- **Precedence**: One operator can have a higher priority, or *precedence*, over another operator. The operators within C are grouped hierarchically according to their ***precedence*** (i.e., order of evaluation)
  - Operations with a higher precedence are carried out before operations having a lower precedence.

    High priority operators * / %

    Low priority operators + -

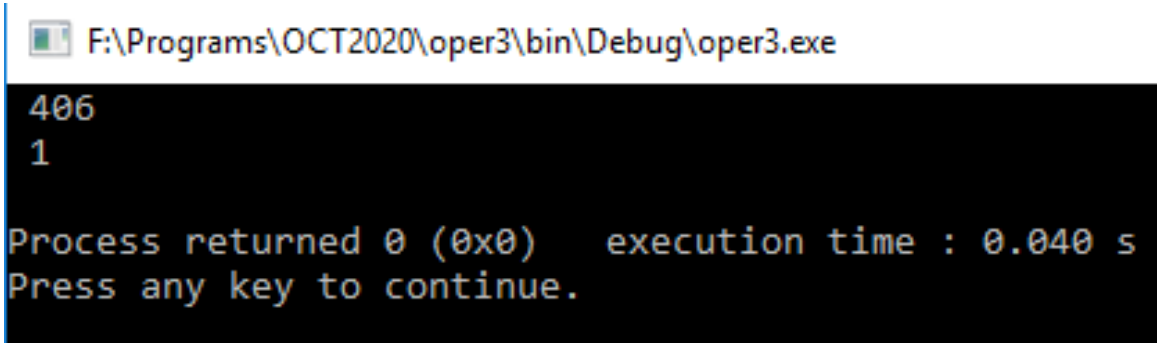
    - E.g.: *** has a higher precedence than +**

      a + b * c → a+(b*c)
  - If necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.
- **Associativity**: Expressions containing operators of the same precedence are evaluated either from ***left to right*** or ***from right to left***, depending on the operator. This is known as the ***Associative*** property of an operator.
    - E.g.: **+ has a *left to right* associativity**

For both the precedence group described above, ***associativity is "left to right"***.

Working with arithmetic expressions

```
#include <stdio.h>
int main ()
{
   int a = 100;
   int b = 4;
   int c = 24;
   int d = 4;
   int result;
   result = a * b + c / d;   // Associativity
   printf("%d\n",result);
   result = a / (b + c * d);// Precedence
   printf("%d\n",result);
   return 0;
}
```



Relational operators

Operator	Meaning
==	Is equal to
!=	Is not equal to
<	Is less than
<=	Is less or equal
>	Is greater than
>=	Is greater or equal

The relational operators have lower precedence than all arithmetic operators:

`a < b + c` **is evaluated as** `a < (b + c)`

ATTENTION !

the "`is equal to`" operator `==` and the "`assignment`" operator `=`

Relational operators

- An expression such as **a < b** containing a relational operator is called a ***relational expression***.

- The value of a relational expression is one, if the specified relation is true and zero if the relation is false.

E.g.:

10 < 20 is TRUE
20 < 10 is FALSE

- A simple relational expression contains only one relational operator and takes the following form.

ae1 relational operator ae2

ae1 & ae2 are arithmetic expressions, which may be simple constants, variables or combinations of them.

Relational operators

The arithmetic expressions will be evaluated first & then the results will be compared. That is, arithmetic operators have a higher priority over relational operators. **>** **>=** **<** **<=** all have the same precedence and below them are the next precedence equality operators i.e. **==** and **!=**

Suppose that i, j and k are integer variables whose values are 1, 2 and 3 respectively.

Expression	Interpretation	Value
i<j	true	1
(i+j)>=k	true	1
(j+k)>(i+5)	false	0
k!=3	false	0
j==2	true	1

Logical operators

Truth Table

op-1	op-2	value of expression	
		op-1 && op-2	op-1 || op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Operator	Symbol	Example
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
NOT	!	!expression1

The result of logical operators is always either 0 (FALSE) or 1 (TRUE)

Logical operators

op-1	op-2	value of expression op-1 && op-2	value of expression op-1 || op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Expressions	Evaluates As
`(5 == 5)&&(6 != 2)`	True (1) because both operands are true
`(5 > 1) || (6 < 1)`	True (1) because one operand is true
`(2 == 1)&&(5 ==5)`	False (0) because one operand is false
`! (5 == 4)`	True (1) because the operand is false
`!(FALSE) = TRUE` `!(TRUE) = FALSE`	

Increment and Decrement operators (++ and --)

- The operator **++** adds 1 to the operand.
- The operator **--** subtracts 1 from the operand.
- Both are unary operators.

- E.g.: `++ i` or `i ++` is equivalent to `i = i + 1`

- **They behave differently when they are used in expressions on the R.H.S of an assignment statement.**

Increment and Decrement operators

Ex:

```
m=5;
y = ++ m;
```

Prefix Mode

In this case, the value of y and m would be 6.

```
m=5;
y = m ++;
```

Postfix Mode

Here y continues to be 5. Only m changes to 6.

Prefix operator ++ appears before the variable.

Postfix operator ++ appears after the variable.

Increment and Decrement operators

Don’ts:

Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference.

Example:

```
++(5)
```
is a syntax error

```
++(x + 1)
```
is a syntax error

Poll Question

Go to chat box/posts for the link to the Poll question

Submit your solution in next 2 minutes

Click the result button to view your score

Bitwise Operators

- Bitwise Logical Operators
- Bitwise Shift Operators
- Ones Complement operator

Bitwise Logical operators

&(AND), |(OR), ^(EXOR)

op1	op2	&	|	^
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

- These are *binary operators* and require two integer operands.
- These work on their operands bit by bit starting from LSB (rightmost bit).

Example

Suppose x = 10, y = 15

z = x & y sets z=10

like this

0000000000001010 ← x

0000000000001111 ← y

0000000000001010 ← z = x & y

Same way **|,^** according to the truth table are computed.

Bitwise Shift operators

- **<< , >>**

- These are used to move bit patterns either to the left or right.

- They are used in the following form

- **op<<n** or **op>>n** here **op** is the **operand** to be shifted and **n** is number of **bit positions** to shift.

Bitwise Shift operator: <<

- << causes all the bits in the operand **op** to be shifted to the left by **n** positions.
- The ***leftmost*** **n** bits in the original bit pattern will be lost and the ***rightmost*** **n** bits that are vacated are filled with **0's**

Bitwise Shift operator: >>

- >> causes all the bits in operand op to be shifted to the right by n positions.
- The ***rightmost*** **n** bits will be lost and the **left most** vacated bits are filled with **0's** if number is unsigned integer.

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011

✓ x<<1 0000 0000 0001 0110 ⬅ Add ZEROS

✓ x>>1 Add ZEROS ➡ 0000 0000 0000 0101

Examples

- Suppose X is an unsigned integer whose bit pattern is 0000 0000 0000 1011 whose equivalent value in decimal number system is 11.

✓ x<<3 0000 0000 0101 1000 ⬅ Add ZEROS = 88

✓ x>>2 Add Z EROS ➡ 0000 0000 0000 0010 = 2

Note:

✓ **x=y<<1;** same as x=y*2 (Multiplication)

✓ **x=y>>1;** same as x=y/2 (Division)

Bitwise Shift operators

- Op and n can be constants or variables.

- There are 2 restrictions on the value of n
  - ✓ n cannot be –ve
  - ✓ n should not be greater than number of bits used to represent Op.

    (E.g.: suppose ***op*** is ***int*** and size is 2 bytes then ***n*** cannot be greater than 16).

Bitwise complement operator

- The complement operator(~) is an ***unary operator*** and inverts all the bits represented by its operand.

- Suppose x = 1001100010001111
  ~x = 0110011101110000 (complement)

- Also called as 1's complement operator.

Summary

- Arithmetic Operators
- Relational and Logical Operators
- Increment and Decrement Operators
- Bitwise Operators