LECTURE 03

Operators

Outcomes

1. To understand the basic rules of expressions and describe the five basic operators categories.
2. To be able to run and understand simple C program using operators.

Contents

1. Introduction of Expressions and Operators

C programming has various operators to perform tasks including arithmetic, relational, logical, assignment, and bitwise operations. You will learn about various C operators and how to use them in this note. So what is operator? An operator is a symbol which operates on a value or a variable. For example: **+** is an operator to perform addition. You will see many other symbols in C programming such as `=`, `>=`, `<=`, `||`, `&&`, `-`, `*`, and etc. All of these symbols have been categories into different category.

Before we go further into each of operator categories, we have to understand first the concept of expressions. An expression is a sequence of operands and operators that reduces to a single value. An operand is an object (e.g. variable) on which an operation is performed (receive operators action). A simple expression contains only one operator, for example `2+5`. A complex expression contains more than one operator, for example `2+5*7`. There are six categories of expression which are primary, postfix, prefix, unary, binary, and ternary. The summarization of these categories are illustrated in the below table.

Expression	Combination of operand and operator	Description	Example
primary	One operand, no operator	Operand can be name, identifier or variable	- **`hutang_negara`** - **`voltage`**
postfix	operand – operator	Consists of one operand and followed by one operator	- `a++` - `a--`
prefix	operator – operand	The operator comes before the operand	- `++a` - `--a`
unary	operator – operand	Like a prefix expression , consist of one operator, one operand	- `(float)x` - `sizeof x`
binary	operand – operator – operand	The most common expression category	- `a * b` - `x = x * (y + 3)`

The other part that we have to understand is about precedence and associativity. The precedence is used to determine the order in which different operators in a complex expression is evaluated. The associativity determines how operators with the same precedence are grouped together to form complex expression. Lets understand the precedence by the example given below:

```
int value = 10 + 20 * 10;
```

The value variable will give result 210 because `*` (multiplicative operator) is evaluated first before `+` (additive operator). On the other hand, associativity can be left-to-right or right-to-left. For example:

```
int jumlah = 3 * 8 / 4 % 4 * 5;
```

Here we have four operators with the same precedence (`* / % *`). Associativity determines how these subexpressions are grouped together. Since all of these operators have the same precedence, their associativity is from left-to-right as follows:

```
int jumlah = ((((3 * 8) / 4) % 4) * 5);
```

The `jumlah` value for this expression is 10. Several operators have right-to-left associativity. For example:

```
a = 3 + b = 5 * c = 8 - 5
```

When more than assignment operators occurs, the assignment operators must be interpreted from right-to-left. This means `c = 8 - 5` is evaluated first. Secondly, `b = 5 * c`, and thirdly `a = 3 + b`. The expression become:

```
(a = 3 + (b = 5 * (c = 8 - 5)))
```

The answer for this expression is 18.

2. Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, and etc. The following table presents the arithmetic operators.

Operator	Description	Example (`A = 10, B = 20`)
+	Add two variable	`A + B` = 30
-	Subtract second variable from first variable	`A - B` = -10
*	Multiply both variables	`A * B` = 200
/	Divides numerator by de-numerator	`B / A` = 20
%	Modulus operator – remainder of after an integer division	`B % A` = 0
++	Incremental operator – increase the integer value by one	`A++` = 11
--	Decrement operator – decrease the integer value by one	`A--` = 9

I have two examples regarding arithmetic operators which are as follows:

```
/*
This program is to show the use of arithmetic operators (+,-,*,/,%)
Written by: AFAN, FKP, UMP    Date: September 2016*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a = 9, b = 4, result;
    result = a + b;
    printf("a+b = %d\n", result);
    result = a - b;
    printf("a-b = %d\n", result);
    result = a * b;
    printf("a*b = %d\n", result);
    result = a / b;
    printf("a/b = %d\n", result);
    result = a % b;
    printf("Remainder when a divided by b = %d\n",result);
    return 0;
}
```

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b = 1
```

The operators `+`, `-` and `*` computes addition, subtraction and multiplication respectively as you might have expected. In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program. It is because both variables `a` and `b` are integers. Hence, the output is also an integer. The compiler neglects the term after decimal point and shows answer `2` instead of `2.25`. The modulus operator `%` computes the remainder. When `a = 9` is divided by `b = 4`, the remainder is `1`. The `%` operator can only be used with integers.

```
/*
This program is to show the use of arithmetic operators (++,--)
Written by: AFAN, FKP, UMP    Date: September 2016*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
    printf("++a = %d\n", ++a);
    printf("--b = %d\n", --b);
    printf("++c = %f\n", ++c);
    printf("--d = %f\n", --d);
    return 0;
}
```

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1. Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand. Here, the operators `++` and `--` are used as prefix. These two operators can also be used as postfix like `a++` and `a--`.

3. Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0. Relational operators are used in selection (`if-else`) and repetitive (`for`, `while`) statements that we will learn in the following chapter. The following table depicts the relational operators.

Operator	Description	Example (`A = 10, B = 20`)
==	Equal to – Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	`(A == B)` is not true
!=	Not equal to – Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	`(A != B)` is true
>	Greater than – Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	`(A > B)` is not true
<	Less than – Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	`(A < B)` is true
>=	Greater than or equal to – Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	`(A >= B)` is not true
<=	Less than or equal to – Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	`(A <= B)` is true

An example of relational operators usage is as below.

```
/*
C Program to demonstrate the working of arithmetic operators
Written by: AFAN, FKP, UMP    Date: September 2016*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a = 5, b = 5, c = 10;
    printf("%d == %d = %d\n", a, b, a == b);    //true
    printf("%d == %d = %d\n", a, c, a == c);    //false
    printf("%d > %d = %d\n", a, b, a > b);      //false
    printf("%d > %d = %d\n", a, c, a > c);      //false
    printf("%d < %d = %d\n", a, b, a < b);      //false
    printf("%d < %d = %d\n", a, c, a < c);      //true
    printf("%d != %d = %d\n", a, b, a != b);    //false
    printf("%d != %d = %d\n", a, c, a != c);    //true
    printf("%d >= %d = %d\n", a, b, a >= b);    //true
    printf("%d >= %d = %d\n", a, c, a >= c);    //false
    printf("%d <= %d = %d\n", a, b, a <= b);    //true
```

```
    printf("%d <= %d = %d\n", a, c, a <= c);     //true
    return 0;
}
```

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1
```

4. Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are also commonly used in selection (`if-else`) and repetitive (`for`, `while`) statements that we will learn in the following chapter. All of descriptions about logical operators are shows in the table on the next page.

Operator	Description	Example (`A = 1, B = 0`)
`&&`	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	`(A && B) is false`
`||`	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	`(A || B) is true`
`!`	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	`!(A && B) is true`

We have learned about logical data in previous chapter (Note 02, page 6). The logical operators is working on principal more or less like logical data. If `A = 1` (TRUE), `B = 0` (FALSE), then `A && B` (TRUE && FALSE) is FALSE (0). Following example shows the working of logical operators with combination of relational operators.

```
/*
C Program to demonstrate the working of logical operators
Written by: AFAN, FKP, UMP    Date: September 2016*/
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a = 5, b = 5, c = 10, result;
    result = (a = b) && (c > b);
    printf("(a = b) && (c > b) equals to %d\n", result);
    result = (a = b) && (c < b);
    printf("(a = b) && (c < b) equals to %d\n", result);
    result = (a = b) || (c < b);
    printf("(a = b) || (c < b) equals to %d\n", result);
    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) equals to %d\n", result);
    result = !(a != b);
    printf("!(a == b) equals to %d\n", result);
    result = !(a == b);
    printf("!(a == b) equals to %d\n", result);
    return 0;
}
```

```
(a = b) && (c > b) equals to 1
(a = b) && (c < b) equals to 0
(a = b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0
```

Explanations of logical operator program are:

1. `(a = b) && (c > 5)` evaluates to 1 because both operands `(a = b)` and `(c > b)` is 1 (TRUE).
2. `(a = b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (FALSE).
3. `(a = b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (TRUE).
4. `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (FALSE).
5. `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (FALSE). Hence, `!(a != b)` is 1 (TRUE).
6. `!(a == b)` evaluates to 0 because `(a == b)` is 1 (TRUE). Hence, `!(a == b)` is 0 (FALSE).

5. Bitwise Operators

In processor, mathematical operations like addition, subtraction, addition, and division are done in bit-level to process faster and save power. **To perform bit-level (binary) operations in C programming, bitwise operators are used**. The following table presents the bitwise operators.

Operator	Description	Example (`A = 12, B = 25`)
&	Bitwise AND Operator copies a bit to the result if it exists in both operands.	`(A & B) = 8`
|	Bitwise OR Operator copies a bit if it exists in either operand.	`(A | B) = 29`
^	Bitwise XOR Operator copies the bit if it is set in one operand but not both.	`(A ^ B) = 21`
~	Bitwise Ones Complement Operator is unary and has the effect of 'flipping' bits.	`(~A) = 243`
<<	Bitwise Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	`(A << 2) = 48`
>>	Bitwise Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	`(A >> 2) = 3`

In order to understand bitwise operation, lets say we have two integers `A = 12` and `B = 25`. `A` and `B` in binary are: `12 = 0000 1100` (`A` in Binary) `25 = 0001 1001` (`B` in Binary)

The output of bitwise AND is 1 if the corresponding bits of all operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0. The bitwise AND operation of (A & B) is:

```
   0000 1100
&  0001 1001
   0000 1000   = 8 (In decimal)
```

```
//C code for bitwise AND
int main(){
      int a = 12, b = 25;
      printf("Output = %d", a&b);
      return 0;}
```

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. The bitwise OR operation of `(A | B)` is:

```
  0000 1100
| 0001 1001
  0001 1101   = 29 (In decimal)
```

```
//C code for bitwise OR
int main(){
      int a = 12, b = 25;
      printf("Output = %d", a|b);
      return 0;}
```

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. The bitwise XOR operation of `(A ^ B)` is:

```
  0000 1100
^ 0001 1001
```

```
//C code for bitwise XOR
int main(){
      int a = 12, b = 25;
      printf("Output = %d", a^b);
```

```
  0001 0101   = 21 (In decimal)                    return 0;}
```

Bitwise Ones Compliment operator is an unary operator (works on one operand only). It changes the 1 to 0 and 0 to 1. The bitwise Ones Compliment operation of `(~A)` is:

```
~ 0000 1100
  1111 0011 = 243 (In decimal)
```

```
//C code for bitwise Compliment
int main(){
      printf("complement=%d",~12);
      return 0;}
```

Left shift operator shifts all bits towards left by certain number of specified bits. The bitwise Left Shift operation of `(A << 2)`, `(A << 3)` and `(A << 0)` are:

```
A = 0000 1100
A << 2 = 0000 1100 00 [left shift by two bits] = 48 (In decimal)
A << 3 = 0000 1100 000 [left shift by three bits] = 96 (In decimal)
A << 0 = 0000 1100 [no left shift] = 12 (In decimal)
```

Right shift operator shifts all bits towards right by certain number of specified bits. The bitwise Right Shift operation of `(A >> 2)`, `(A >> 3)` and `(A >> 0)` are:

```
A = 0000 1100
A >> 2 = 0000 0011 [right shift by two bits] = 3 (In decimal)
A >> 3 = 0000 0001 [right shift by three bits] = 1 (In decimal)
A >> 0 = 0000 1100 [no right shift] = 12 (In decimal)
```

Example of C program for right and left shift bitwise operators are as follow:

```
#include <stdio.h>
int main(){
    int num=12, i;
    for (i=0; i<=3; ++i)
        printf("Right shift by %d: %d\n", i, num>>i);
    for (i=0; i<=3; ++i)
        printf("Left shift by %d: %d\n", i, num<<i);
     return 0;}
```

6. Assignment Operators

An assignment operator is used for assigning a value to a variable. The next table discusses the assignment operators.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	`C = A + B` will assign the value of `A + B` to `C`
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	`C += A` is equivalent to `C = C + A`
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	`C -= A` is equivalent to `C = C - A`
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	`C *= A` is equivalent to `C = C * A`
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	`C /= A` is equivalent to `C = C / A`
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	`C %= A` is equivalent to `C = C % A`

```
/*C Program to demonstrate the working of assignment operators
Written by: AFAN, FKP, UMP          Date: September 2016*/
#include <stdio.h>
#include<stdlib.h>
int main(){
    int a = 5, c;
    c = a;
    printf("c = %d\n", c);
    c += a;                    // c = c + a
    printf("c = %d\n", c);
    c -= a;                    // c = c - a
    printf("c = %d\n", c);
    c *= a;                    // c = c * a
    printf("c = %d\n", c);
    c /= a;                    // c = c / a
    printf("c = %d\n", c);
    c %= a;                    // c = c % a
    printf("c = %d\n", c);
    return 0;
}
```

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```