Lecture 4.1

Topics

1. Operators and Operands
2. Primary, Binary, and Assignment Expressions

---

1. Operators and Operands

C coding will have many statements. What does a C statement have?

The general form of a C statement is as follows,

```
expression;
```

In the above, the expression can be either very simple or highly complex (depending on the underlined logic). One can form an expression using some combination of operators, operands and sub-expressions.

Then a C statement would include an expression terminated with a semicolon.

There are many operators that are built into C such as: `+, -, *, /, %,` etc. Each operator requires one or more operands so that they can be used to form the so-called **expression**. By combining these expressions together with semicolon terminator, one can then form a **statement**.

In C, expressions formed by operators must follow their rules of **precedence** and **associativity**. Some of these operators are given below.

Precedence	Operator	Operation	Association
1	`( )`	Parentheses : Parameter Evaluation, Function (Method) Invocation/Call	L to R
	`[ ]`	Array indexing	L to R
	`.`	Member reference	L to R
	`++`	Unary postfix increment	L to R
	`--`	Unary postfix decrement	L to R
2	`++`	Unary prefix increment	R to L
	`--`	Unary prefix decrement	R to L
	`~`	Bitwise NOT	R to L
	`!`	Logical NOT	R to L
3	`*`	De-reference	R to L
	`&`	Address of	R to L
	`(type)`	Type casting	R to L
4	`sizeof`	Size in bytes of a type	R to L
5	`*`	Multiplication	L to R
	`/`	Division	L to R
	`%`	Remainder	L to R
6	`+`	Addition	L to R
	`-`	Subtraction	L to R

7	`<`	Less than	L to R
	`<=`	Less than or equal to	L to R
	`>`	Greater than	L to R
	`>=`	Greater than or equal to	L to R
8	`==`	Equal to	L to R
	`!=`	Not equal to	L to R
9	`&`	Boolean AND	L to R
10	`^`	Boolean exclusive OR (XOR)	L to R
11	`|`	Boolean OR	L to R
12	`&&`	Logical AND	L to R
13	`||`	Logical OR	L to R
14	`? :`	Ternary conditional operator	R to L
15	`=`	Assignment	R to L
	`+=`	Addition, then assignment	R to L
	`+=`	String concatenation, then assignment	R to L
	`-=`	Subtraction, then assignment	R to L
	`*=`	Multiplication, then assignment	R to L
	`/=`	Division, then assignment	R to L
	`%=`	Remainder, then assignment	R to L

The rules of precedence and associativity are important because they provide proper instructions for identifying and evaluating expressions.

1.1 Precedence

Operator precedence establishes the ranking or priority of the operations to be performed in an expression. The application of operator precedence would resolve many of the possible confusion when trying to obtain the result of some expressions.

Given an expression with combination of several operators and operands, one can use the following rules:

(i) Starting from the left, scan through the expression to identify the operators,

(ii) Mark the operators with their rankings based on the list given in previous lecture,

(iii) Starting with the operators with highest rank to lowest rank, identify their operand(s),

(iv) Use parentheses to group operator and operand(s) for evaluation,

(v) Evaluate all possible parenthetical expressions,

(vi) Repeat Steps (i) through (v) until a single result is produced.

1.2 Associativity

Associativity can be from the **Left-to-Right** or **Right-to-Left** and it should be used to apply to operators and operands of the same level of precedence in an expression.

1.2.1 Left-to-Right Associativity

The illustration is given in the following example.

```
3 * 8 / 4 % 4 * 5                                        (expr 1)
```

All the operators have the same precedence and left-to-right associativity. Applying the above steps to the expression will result in the following parenthesized expression:

```
((((3 * 8) / 4) % 4) * 5)                                (expr 2)
```

1.2.2 Right-to-Left Associativity

There are only three types of expressions that have the right-to-left associativity:

(a) (Most of) the unary expressions,

(b) The conditional ternary expression, and

(c) The assignment expressions.

Let’s consider the assignment expressions here and the other expressions may be discussed in future lectures.

When several assignment operators are used in an expression, the association must start from right to left. That means the rightmost expression will be evaluated first; its result will be assigned to the left operand of the next expression for subsequent evaluation.

An example is given below,

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

The final parenthesized expression is found to be as follows,

```
(a += (b *= (c -= 5)))                                   (expr 4)
(a = a + (b = b * (c = c – 5)))                          (expr 5)
```

Thus, the result can be found easily from the last expression.

2. Primary, Binary, and Assignment Expressions

Recall that expressions are formed with combinations of **sub-expressions**, which can be identified as **primary group**, and **operator-operand combination group**.

Every expression must have a value and this value may or may not be usable depending how and where the expression was formed or defined.

2.1 Primary Expressions

Primary expression is the **most** basic type of expression. There are three primary expressions:

Identifiers	Any variable, function, defined name, object names
	`iVar     cVar     carCount     compObj     getValue()`
Constants	Any face-valued constants
	`5     9.5     ‘a’     “Monday”`
Parenthetical Expressions	Any complex expression can be enclosed in parentheses
	`(3 + 4 * 5)     (iVar = 5 * 4 – 1)`

2.2 Binary Expressions

A binary expression has an operator that requires two operands – the left (**`lOp`**) and right operands (**`rOp`**).

- The set of binary arithmetic operators may include addition, subtraction, multiplication, division, and modulo.
- There are other binary operators such as assignments, comparisons, logical, etc.

2.3 Assignment Expressions

There are **simple** assignments and **compound** assignments, which are also binary operators.

They are easy to understand as given in the following statements.

```
iVar = 5;
iVar = iVar + 9;
iVar += 15;                /* iVar = iVar + 15 */

iVar += iCount + 5;        /* iVar = iVar + (iCount + 5) */
```

2.4 Unary Operators

Unary operators only have one operand. There are several unary operators such as

```
iVar++;          // post-increment operator
--iVar;          // pre-decrement operator
!iVar;           // NOT Boolean operator
```