MASTER OF COMPUTER APPLICATION

SEMESTER 1

PROGRAMMING & PROBLEM-SOLVING USING C

Unit 2

Operators and Expressions

Table of Contents

1. INTRODUCTION

In the previous unit, you learned about the various features and structure of C programs. You also learned how the variables in C are declared. In this unit, you will learn about the operators that are available in C and how the expressions can be formed to get the solutions of any problems.

C supports a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operator can be classified into a number of categories. They include:

1. Arithmetic operators
2. Unary operator
3. Relational operators
4. Logical operators
5. Conditional operator
6. Bitwise operators
7. Increment and Decrement operators

1.1 Objectives

After studying this subject, you should be able to:

- *Explain different categories of operators*
- *Use operators on many operands*
- *Distinguish precedence and associativity of operators*
- *Explain library functions and their use*
- *Write small programs using different types of operators*

2. ARITHMETIC OPERATORS

The basic operators for performing arithmetic are the same in many computer languages:

+ addition

- subtraction

* multiplication

/ division

% modulus (remainder)

The - operator can be used in two ways: to subtract two numbers (as in a - b), or to negate one number (as in -a + b or a + -b).

When applied to integers, the division operator / discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (a real number), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5, and 7.0 / 4.0 is 1.75.

The *modulus* operator % gives you the remainder when two integers are divided: 1 % 2 is 1; 7 % 4 is 3. (The modulus operator can only be applied to integers.)

An additional arithmetic operation you might be wondering about is exponentiation. Some languages have an exponentiation operator (typically ^ or **), but C doesn't. (To square or cube a number, just multiply it by itself.)

Multiplication, division, and modulus all have higher *precedence* than addition and subtraction. The term “precedence'' refers to how “tightly'' operators bind to their operands (that is, to the things they operate on). In mathematics, multiplication has higher precedence than addition, so 1 + 2 * 3 is 7, not 9. In other words, 1 + 2 * 3 is equivalent to 1 + (2 * 3). C is the same way.

All of these operators “group'' from left to right, which means that when two or more of them have the same precedence and participate next to each other in an expression, the evaluation conceptually proceeds from left to right. For example, 1 - 2 - 3 is equivalent to (1 - 2) - 3 and gives -4, not +2. (“Grouping'' is sometimes called *associativity*, although the term is used

somewhat differently in programming than it is in mathematics. Not all C operators group from left to right; a few groups from right to left.)

Whenever the default precedence or associativity doesn't give you the grouping you want, you can always use explicit parentheses. For example, if you want to add 1 to 2 and then multiply the result by 3, you could write (1+2)*3.

Program 2.1: Program that shows the use of integer arithmetic to convert a given number of days into months and days.

```
/* Program to convert days to months and days */
main()
{
int months, days;
printf(“Enter days\n”);
scanf(“%d”,&days);
months=days/30;
days=days%30;
printf(“Months=%d    Days=%d”, months,days);

}
```

SELF-ASSESSMENT QUESTIONS – 1

1. What is the value of the arithmetic expression: 14 % 3 + 7 % 2
   (a) 1 (b) 2 (c) 3 (d) 4
2. _____ operator can be only applied to integers.

3. UNARY OPERATOR

A unary operator acts upon a single operand to produce a new value.

Unary Minus

The most well known unary operator is minus, where a minus sign precedes a constant, variable or expression. In C, all numeric constants are positive. Therefore, a negative number is actually a positive constant preceded by a unary minus, for example:

-3

4. RELATIONAL AND LOGICAL OPERATORS

An if statement like

```
if(x > max)
    max = x;
```

is perhaps deceptively simple. Conceptually, we say that it checks whether the condition x > max is “true'' or “false''. The mechanics underlying C's conception of “true'' and “false,'' however, deserve some explanation. We need to understand how true and false values are represented, and how they are interpreted by statements like if.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: “true'' and “false.'' The study of mathematics involving only two values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, “false'' is represented by a value of 0 (zero), and “true'' is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type int), when we have to pick a specific value for “true,'' we'll pick 1.

The *relational operators* such as <, <=, >, and >= are in fact operators, just like +, -, *, and /. The relational operators take two values, look at them, and “return'' a value of 1 or 0 depending on whether the tested relation was true or false. The complete set of relational operators in C is:

< less than

<= less than or equal

>	greater than
>=	greater than or equal
==	equal
!=	not equal

For example, 1 < 2 is true(1), 3 > 4 is false(0), 5 == 5 is true(1), and 6 != 6 is false(0).

The equality-testing operator is ==, not a single =, which is assignment. If you accidentally write

if(a = 0)

(and you probably will at some point; everybody makes this mistake), it will *not* test whether a is zero, as you probably intended. Instead, it will *assign* 0 to a, and then perform the "true'' branch of the if statement if a is *nonzero*. But a will have just been assigned the value 0, so the "true'' branch will never be taken! (This could drive you crazy while debugging -- you wanted to do something if a was 0, and after the test, a is 0, whether it was supposed to be or not, but the "true'' branch is nevertheless not taken.)

The relational operators work with arbitrary numbers and generate true/false values. You can also combine true/false values by using the *Boolean operators(also called the logical operators),* which take true/false values as operands and compute new true/false values. The three Boolean operators are:

&&	AND
||	OR
!	NOT (takes one operand, "unary'')

The && ("and'') operator takes two true/false values and produces a true

(1) result if both operands are true (that is, if the left-hand side is true **and** the right-hand side is true). The || ("or'') operator takes two true/false values and produces a true (1) result if either operand is true. The ! ("not'') operator takes a single true/false value and negates it, turning false to true and true to false (0 to 1 and nonzero to 0). The logical operators && and || are used when we want to test more than one condition and make decisions.

For example, to test whether the variable i lies between 1 and 10, you might use

```
if(1 < i && i < 10)
...
```

Here we're expressing the relation “i is between 1 and 10'' as “1 is less than i **and** i is less than 10.''

It's important to understand why the more obvious expression

```
if(1 < i < 10)    /* WRONG */
```

would not work. The expression 1 < i < 10 is parsed by the compiler analogously to 1 + i + 10. The expression 1 + i + 10 is parsed as (1 + i) + 10 and means “add 1 to i, and then add the result to 10.'' Similarly, the expression 1 < i < 10 is parsed as (1 < i) < 10 and means “see if 1 is less than i, and then see if the result is less than 10.'' But in this case, “the result'' is 1 or 0, depending on whether i is greater than 1. Since both 0 and 1 are less than 10, the expression 1 < i < 10 would *always* be true in C, regardless of the value of i!

Relational and Boolean expressions are usually used in contexts such as an if statement, where something is to be done or not done depending on some condition. In these cases what's actually checked is whether the expression representing the condition has a zero or nonzero value. As long as the expression is a relational or Boolean expression, the interpretation is just what we want. For example, when we wrote

```
if(x > max)
```

the > operator produced a 1 if x was greater than max, and a 0 otherwise.

The if statement interprets 0 as false and 1 (or any nonzero value) as true.

But what if the expression is not a relational or Boolean expression? As far as C is concerned, the controlling expression (of conditional statements like if) can in fact be *any* expression: it doesn't have to “look like'' a Boolean expression; it doesn't have to contain relational or logical operators. All C looks at (when it's evaluating an if statement, or anywhere else where it needs a true/false value) is whether the expression evaluates to 0 or nonzero. For example, if you have a variable x, and you want to do something if x is nonzero, it's possible to write

```
if(x)
        statement
```

and the statement will be executed if x is nonzero (since nonzero means “true'').

This possibility (that the controlling expression of an if statement doesn't have to “look like'' a Boolean expression) is both useful and potentially confusing. It's useful when you have a variable or a function that is “conceptually Boolean,'' that is, one that you consider to hold a true or false (actually nonzero or zero) value. For example, if you have a variable verbose which contains a nonzero value when your program should run in verbose mode and zero when it should be quiet, you can write things like

```
if(verbose)
    printf("Starting first pass\n");
```

and this code is both legal and readable, besides which it does what you want. The standard library contains a function isupper() which tests whether a character is an upper-case letter, so if c is a character, you might write

```
if(isupper(c))
    ...
```

Both of these examples (verbose and isupper()) are useful and readable.

However, you will eventually come across code like

```
if(n)
   average = sum / n;
```

where n is just a number. Here, the programmer wants to compute the average only if n is nonzero (otherwise, of course, the code would divide by 0), and the code works, because, in the context of the if statement, the trivial expression n is (as always) interpreted as “true'' if it is nonzero, and “false'' if it is zero.

“Coding shortcuts'' like these can seem cryptic, but they're also quite common, so you'll need to be able to recognize them even if you don't choose to write them in your own code. Whenever you see code like

```
if(x)
```

or

```
if(f())
```

where x or f() do not have obvious “Boolean'' names, you can read them as “if x is nonzero'' or “if f() returns nonzero.''

SELF-ASSESSMENT QUESTIONS – 2

3. The logical operators _____ and _____ are used when we want to test more than one condition and make decisions.
4. State whether the following statement is correct or not. (Correct/Incorrect)

```
if(a<4<c)
b=c;
```

5. THE CONDITIONAL OPERATOR

The Conditional operator (ternary operator) pair "?:" is available in C to construct conditional expressions of the form *expr1?expr2:expr3*

where *expr1, expr2* and *expr3* are expressions.

The operator ?: works as follows: expr1 is evaluated first. If it is nonzero(true), then the expression *expr2* is evaluated and becomes the value of the expression. If *expr1* is false, *expr3* is evaluated and its value becomes the value of the expression. For example, consider the following statements:

```
a=100;
b=200;
c=(a>b)?a:b;
```

In this example, c will be assigned the value of b. This can be achieved using the if..else statements as follows: if(a>b)

```
c=a;
else
c=b;
```

6. LIBRARY FUNCTIONS

The C language is accompanied by a number of *library functions or built in functions* that carry out various commonly used operations or calculations. There are library functions that carry out standard input/output operations, functions that perform operations on characters, functions that perform operations on strings and functions that carry out various mathematical calculations.

Functionally similar library functions are usually grouped together as object programs in separate library files.

A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. A function that returns a data item can appear anywhere within an expression in place of a constant or an identifier. A function that carries out operations on data items but does not return anything can be accessed simply by writing the function name.

A typical set of library functions will include a large number of functions that are common to most C compilers, such as those shown in table 2.1.

Function	Purpose
abs(i)	Return the absolute value of (i is integer)
ceil(d)	Round up to the next integer value(the smallest integer that is greater than or equal to d)
cos(d)	Return the cosine of d
exp(d)	Raise e to the power d(e=Naperian constant)
fabs(d)	Return the absolute value of d(d is double)
floor(d)	Round down to the next integer value(the largest integer that

	does not exceed d)
getchar()	Enter a character from the standard input device
log(d)	Return the natural logarithm of d
pow(d1,d2)	Return d1 raised to the power d2
putchar(c)	Send a character to the standard output device
rand()	Return a random positive integer
sin(d)	Return sine of d
sqrt(d)	Return the square root of d
tan(d)	Return the tangent of d
toascii(c)	Convert value of argument to ASCII
tolower(c)	Convert letter to lowercase
toupper(c)	Convert letter to uppercase

Table 2.1

Program 2.2: Program to convert lowercase to uppercase

```
#include <stdio.h> /* Input/Output functions are available in stdio.h */
#include<ctype.h> /* Character functions are available in the file ctype.h */
main()

/* read a lowercase character and print its uppercase equivalent */ {

        int lower, upper;
        lower=getchar();
        upper=toupper(lower);
        putchar(upper);
 }
```

Program 2.3: Program to illustrate the use of library functions

```
#include<stdio.h>
#include<ctype.h>
#include<math.h> /* Mathematical functions are available in math. h*/
main()
{
        int i=-10, e=2, d=10;
        float rad=1.57;
        double d1=2.0, d2=3.0;
        printf("%d\n", abs(i));
        printf("%f\n", sin(rad));
        printf("%f\n", cos(rad));
        printf("%f\n", exp(e));
        printf("%d\n", log(d));
        printf("%f\n", pow(d1,d2));
}
```

Execute the above program and observe the result

SELF-ASSESSMENT QUESTIONS – 3

5. _____ are built-in functions that carry out various commonly used operations or calculations
6. Fill in the blanks as indicated by alphabets. The value of (a) floor(5.8), (b) floor(-5.8), (c) ceil(5.8) and (d) ceil(-5.8) are _____ (a) ____, ___ (b) _____, ___ (c) _____ and ____ (d) _____ respectively.

7. BITWISE OPERATORS

The *bitwise operators* &, |, ^, and ~ operate on integers thought of as binary numbers or strings of bits. The & operator is bitwise AND, the | operator is bitwise OR, the ^ operator is bitwise exclusive-OR (XOR), and the ~ operator is a bitwise negation or complement. (&, |, and ^ are "binary'' in that they take two operands; ~ is unary.) These operators let you work with the individual bits of a variable; one common use is to treat an integer as a set of single-bit *flags*. You might define the 3rd bit as the "verbose'' flag bit by defining

```
#define VERBOSE 4
```

Then you can "turn the verbose bit on'' in an integer variable flags by executing

```
flags = flags | VERBOSE;
```

and turn it off with

```
flags = flags & ~VERBOSE;
```

and test whether it's set with

```
if(flags & VERBOSE)
```

The left-shift and right-shift operators << and >> let you shift an integer left or right by some number of bit positions; for example, value << 2 shifts value left by two bits.

The **comma operator** can be used to link the related expressions together. The expressions are executed one after the other. The most common use for comma operators is when you want multiple variables controlling a for loop, for example:

```
for(i = 0, j = 10; i < j; i++, j--)
```

SELF-ASSESSMENT QUESTIONS – 4

7. _____ let you work with the individual bits of a variable; one common use is to treat an integer as a set of single-bit flags.
8. Fill in the blanks as indicated by alphabets. If flag1=5, flag2=8, then (a) flag1&flag2, (b) flag1|flag2, (c) ~flag1 and (d) flag1^flag2 is computed to be _____ (a) ____, ___ (b) _____, ___ (c)_____ and ____ (d) _____ respectively.

8. INCREMENT AND DECREMENT OPERATORS

When we want to add or subtract constant 1 to a variable, C provides a set of shortcuts: the *autoincrement* and *autodecrement* operators. In their simplest forms, they look like this:

++i *add* 1 *to* i

--j *subtract* 1 *from* j

These correspond to the forms i = i + 1 and j = j - 1. They are also equivalent to the short hand forms i+=1 and j-=1. C has a set of 'shorthand' assignment operators of the form:

v op=exp;

where v is a variable, exp is an expression and op is a C binary arithmetic operator.

The assignment statement

v op=exp;

is equivalent to

v= v op(exp);

Example:

x+=y+1;

This is same as the statement

x=x+(y+1);

The ++ and -- operators apply to one operand (they're *unary* operators). The expression ++i adds 1 to i, and stores the incremented result back in i. This means that these operators don't just compute new values; they also modify the value of some variable. (They share this property -- modifying some variable -- with the assignment operators; we can say that these operators all have *side effects*. That is, they have some effect, on the side, other than just computing a new value.)

The incremented (or decremented) result is also made available to the rest of the expression, so an expression like

k = 2 * ++i

means “add one to i, store the result back in i, multiply it by 2, and store *that* result in k.'' (This is a pretty meaningless expression; our actual uses of ++ later will make more sense.)

Both the ++ and -- operators have an unusual property: they can be used in two ways, depending on whether they are written to the left or the right of the variable they're operating on. In either case, they increment or decrement the variable they're operating on; the difference concerns whether it's the old or the new value that's “returned'' to the surrounding expression. The *prefix* form ++i increments i and returns the incremented value. The *postfix* form i++ increments i, but returns the *prior*, non-incremented value. Rewriting our previous example slightly, the expression

```
k = 2 * i++
```

means “take i's old value and multiply it by 2, increment i, store the result of the multiplication in k.''

The distinction between the prefix and postfix forms of ++ and -- will probably seem strained at first, but it will make more sense once we begin using these operators in more realistic situations. For example,

```
a[i] = c;
i = i + 1;
```

using the ++ operator, we could simply write this

```
as a[i++] = c;
```

We wanted to increment i *after* deciding which element of the array to store into, so the postfix form i++ is appropriate.

Notice that it only makes sense to apply the ++ and -- operators to variables (or to other “containers,'' such as a[i]). It would be meaningless to say something like

```
1++
```

or

```
(2+3)++
```

The ++ operator doesn't just mean “add one''; it means “add one *to a variable''* or “make a variable's value one more than it was before.'' But (1+2) is not a variable, it's an expression; so there's no place for ++ to store the incremented result.

Another unfortunate example is

```
i = i++;
```

which some confused programmers sometimes write, presumably because they want to be extra sure that i is incremented by 1. But i++ all by itself is sufficient to increment i by 1; the extra (explicit) assignment to i is unnecessary and in fact counterproductive, meaningless, and incorrect. If you want to increment i (that is, add one to it, and store the result back in i), either use

```
i = i + 1;
```

or

```
i += 1;
```

or

```
++i;
```

or

```
i++;
```

Did it matter whether we used ++i or i++ in this last example? Remember, the difference between the two forms is what value (either the old or the new) is passed on to the surrounding expression. If there is no surrounding expression, if the ++i or i++ appears all by itself, to increment i and do nothing else, you can use either form; it makes no difference. (Two ways that an expression can appear “all by itself,'' with “no surrounding expression,'' are when it is an expression statement terminated by a semicolon, as above, or when it is one of the controlling expressions of a for loop.) For example, both the loops

```
for(i = 0; i < 10; ++i)
        printf("%d\n", i);
```

and

```
for(i = 0; i < 10; i++)
        printf("%d\n", i);
```

will behave exactly the same way and produce exactly the same results. (In real code, postfix increment is probably more common, though prefix definitely has its uses, too.)

SELF-ASSESSMENT QUESTIONS – 5

9. Increment and Decrement operators are binary operators. (True/False)
10. The prefix form to increment i is _____ whereas the postfix form to increments i is _____.

9. THE SIZE OF OPERATOR

The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

m=sizeof(sum);

n=sizeof(long int);

k=sizeof(235L);

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Program 2.4: Program to illustrate the use of size of operator

```
#include<stdio.h>
main()
{
        int i=10;
        printf("integer: %d\n", sizeof(i);
}
```

The above program might generate the following output:

integer: 2

Thus we see that this version of C allocates 2 bytes to each integer quantity.

Program 2.5: Program to illustrate arithmetic operators

```
#include<stdio.h>
main()
{
        int a, b, c, d;
        a=10;
        b=15;
        c=++a-b;
        printf("a=%d  b=%d c=%d\n", a, b, c);
        d=b++ +a;
        printf("a=%d  b=%d d=%d\n", a, b, d);
        printf("a/b=%d\n", a/b);
        printf("a%%b=%d\n", a%b);
        printf("a*=b=%d\n", a*=b);
        printf("%d\n", (c>d)?1:0);
        printf("%d\n", (c<d)?1:0);

}
```

Execute the above program and observe the result.

10. PRECEDENCE OF OPERATORS

The precedence of C operators dictates the order of evaluation within an expression. The precedence of the operators introduced here is summarised in the Table 2.2. The highest precedence operators are given first.

Operators	Associativity
()->.	left to right
!~+-++--&*	right to left
* / %	left to right
+ -	left to right
<<=>>=	left to right
== !=	left to right
&	left to right
|	left to right
&&	left to right
||	right to left
=*=/=%=+=-=	right to left

Table 2.2

Where the same operator appears twice (for example *) the first one is the unary version.

Program 2.6: A program to illustrate evaluation of expressions

```
#include<stdio.h>
main()
/* Evaluation of expressions */
{
        float a, b, c, x, y, z;
        a=20;
        b=2;
        c=-23;
        x = a + b / ( 3 + c * 4 - 1);
        y = a – b / (3 + c) * ( 4 – 1);
        z= a – ( b / ( 3 + c ) * 2 ) – 1;
        printf( “x=%f\n”, x);
        printf(“y=%f\n”, y);
        printf(“z=%f\n”, z);

}
```

Execute the above program and observe the result.

Program 2.7: Program to convert seconds to minutes and seconds #include <stdio.h>

```
#include <stdio.h>
#define SEC_PER_MIN 60       // seconds in a minute
int main(void)
{
  int sec, min, left;

  printf("Convert seconds to minutes and seconds!\n");
  printf("Enter the number of seconds you wish to convert.\n");
  scanf("%d", &sec);        *number of seconds is read in
  min = sec / SEC_PER_MIN; *truncated number of minutes
  left = sec % SEC_PER_MIN; *number of seconds left over
  printf("%d seconds is %d minutes, %d seconds.\n", sec, min,
        left);
  return 0;
}
```

11. SUMMARY

C supports a rich set of operators. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. A binary operator acts on two operands. A unary operator acts upon a single operand to produce a new value. Multiplication, division, and modulus all have higher precedence than addition and subtraction. Relational and Boolean expressions are usually used in contexts such as an if statement, where something is to be done or not done depending on some condition. The C language is accompanied by a number of library functions or built in functions that carry out various commonly used operations or calculations. The **size of** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program. Associativity is the order in which consecutive operations within the same precedence group are carried out.

12. TERMINAL QUESTIONS

1. If i=10 and j=12, what are the values of c and d after executing the following program segment:

   i++; c=j++ +

   i; d=++i +

   j++;
2. Suppose that x, y and z are integer variables which have been assigned the values 2, 3 and 4, respectively. Evaluate the following expression and determine the value of x.

   x *= -2 * (y + z) / 3
3. Suppose that i is an integer variable whose value is 7 and c is a character variable that represents the character 'w', evaluate the following logical expression:

   (i>=6) && (c=='w')
4. Suppose that i is an integer variable whose value is 7 and f is a floating

   –point variable whose value is 8.5. Evaluate the following expression: (i + f) %4
5. What is meant by associativity?

13. ANSWERS

Self-Assessment Questions

1. 3
2. % (modulus)
3. &&, ||
4. Incorrect
5. Library functions
6. (a) 5, (b) -6, (c) 6, (d) -5
7. Bitwise operator
8. (a) 0, (b) 13, (c) 10 , (d) 13
9. False
10. ++i, i++

Terminal Questions

1. c=23 and d=25
2. -8
3. true
4. Given expression is invalid because a floating point variable can not be used in a modulus operation.
5. Associativity is the order in which consecutive operations within the same precedence group are carried out.

Exercises

1. Suppose a=3, b=5, c=8 and d=4, give the output of the following:
   a) x=a*b-c/d+4 b) z=a-b/d*c+10
2. Suppose i=5, j=8, k=10, then , what is the output of the following:
   a) x=a++ -j b) y=k++ *j—
3. What is the precedence of operators? How expressions are evaluated using the precedences?
4. Suppose a=7, b=11, find the output of the following:
   a) x=(a>b)?b:a b) x=(a<b)?a:b
5. Explain the use of bitwise operators with suitable examples.