Objectives

- To review basic arithmetic operators.
- To be able to understand the different operators and their importance in the programming.
- To understand the expressions and evaluation of expressions.
- To understand control structures like decision making statements, looping statements.
- To understand the decision making statements like if and switch statements.
- To understand the loop control statements like while, for and do while loops, jump statements, break, continue, goto statements.
• An operator is a symbol which helps the user to give instruction to the computer to do a certain mathematical or logical manipulations.

• Operators are used in C language program to operate on data and variables.

• C has a rich set of operators which can be classified as follows:

1. Arithmetic operators.
2. Relational Operators.
3. Logical Operators.
4. Assignment Operators.
5. Increments and Decrement Operators.
8. Special Operators.
Arithmetic Operators

- All the basic arithmetic operations can be carried out in C.
- All the operators have almost the same meaning as in other languages.
- Both unary and binary operations are available in C language.
- Unary operations operate on a single operand,
- Therefore the number 5 when operated by unary – will have the value –5.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition or Unary Plus</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or Unary Minus</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus Operator</td>
</tr>
</tbody>
</table>
1. Arithmetic Operators

Examples of arithmetic operators are: $x + y$, $x - y$, $x * y$, $x / y$, $x \% y$

The modulus operator is a special operator in C language which evaluates the remainder of the operands after division.

**Integer Arithmetic**

When an arithmetic operation is performed on two integers than such an operation is called as integer arithmetic. It always gives an integer as the result.

Let $x = 27$ and $y = 5$

- $x + y = 32$
- $x - y = 22$
- $x * y = 115$
- $x \% y = 2$
- $x / y = 5$

**Floating point arithmetic**

When an arithmetic operation is performed on two real numbers such an operation is called floating point arithmetic.

Let $x = 14.0$ and $y = 4.0$

- $x + y = 18.0$
- $x - y = 10.0$
- $x * y = 56.0$
- $x / y = 3.50$

**Mixed mode arithmetic**

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real type.

$15/10.0 = 1.5$

Program for arithmetic operators u1_ex19.c
Relational and logical operators

2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator comes into picture. C supports the following relational operators.

Ex: It is required to compare the marks of 2 students, salary of 2 persons, we can compare them using relational operators.

\[(6.5 \leq 25), (-65 > 0), (10 < 7 + 5)\]

which result in either TRUE OR FALSE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.
3. Logical Operators

C has the following logical operators; they compare or evaluate logical and relational expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>

**Logical AND (&&)**
If both the expressions are true then the whole compound expression is true.  
**Example:** \( a > b \) && \( x = 10 \)

**Logical OR (||)**
If any one of the 2 expressions is true.
**Example:** \( a < m \) || \( a < n \)

**Logical NOT (!)**
The logical not operator evaluates to true if the expression is false and evaluates to false if the expression is true.
**For example:** \(! (x >= y)\)

Program for logical operators `u1_ex21.c`
4. Assignment Operators

- The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

**Example:** \( x = a + b \)

In addition, C has a set of shorthand assignment operators of the form.

```
var oper = exp;
```

**Example:** \( x += 1 \) is same as \( x = x + 1 \)

The commonly used shorthand assignment operators are as follows:

<table>
<thead>
<tr>
<th>Shorthand Assignment Operators</th>
<th>Simple Assignment Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a += 1 )</td>
<td>( a = a + 1 )</td>
</tr>
<tr>
<td>( a -= 1 )</td>
<td>( a = a - 1 )</td>
</tr>
<tr>
<td>( a *= (n+1) )</td>
<td>( a = a * (n+1) )</td>
</tr>
<tr>
<td>( a /= (n+1) )</td>
<td>( a = a / (n+1) )</td>
</tr>
<tr>
<td>( a %= b )</td>
<td>( a = a % b )</td>
</tr>
</tbody>
</table>

Program for shorthand Assignment operators u1_ex22.c
1. The increment and decrement operators are one of the unary operators which are very useful in C language.

2. They are extensively used in for and while loops. The syntax of the operators is given below

1. `++ variable name`
2. `variable name++`
3. `--variable name`
4. `variable name--`

The increment operator `++` adds the value 1 to the current value of operand. The decrement operator `--` subtracts the value 1 from the current value of operand.
5. Increment and Decrement Operators

1. `++variable name` and `variable name++` mean the same thing when they form statements independently.
2. They behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following:

```c
m = 5;
y = ++m; (prefix)  In this case the value of y and m would be 6
Suppose if we rewrite the above statement as
m = 5;
y = m++; (post fix)  Then the value of y will be 5 and that of m will be 6.
```

Program for increment/ decrement operators u1_ex23.c
6. Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark (?) and colon (:). The syntax for a ternary operator is as follows:

\[ \text{exp1} \ ? \ \text{exp2} \ : \ \text{exp3} \]

The ternary operator works as \( \text{exp1} \) is evaluated first. If the expression is true then \( \text{exp2} \) is evaluated & its value becomes the value of the expression. If \( \text{exp1} \) is false, \( \text{exp3} \) is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example:
\[
\begin{align*}
a &= 10; \\
b &= 15; \\
x &= (a > b) \ ? \ a : b
\end{align*}
\]

Here \( x \) will be assigned to the value of \( b \).
The condition follows that the expression is false therefore \( b \) is assigned to \( x \).

Output
Input 2 integers : 45 34
The largest of two numbers is 45

Program for conditional operators u1_ex24.c
7. Bitwise Operators

- The C language is well suited to system programming because it contains operators that can manipulate data at the bit level.
- Those operators are used for testing, complementing or shifting bits to the right or left.
- C has two categories of bitwise operators that operate on data at the bit level:
  1. Logical bitwise operators and
  2. Shift bitwise operators.

**Logical bitwise operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise Exclusive OR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right</td>
</tr>
</tbody>
</table>
### Bitwise Operator

7. Bitwise Operators Truth table

<table>
<thead>
<tr>
<th>~a</th>
<th>a^b</th>
<th>a</th>
<th>b</th>
<th>a&amp;b</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
7. Bitwise Shift Operators

The shift operators moves bits to the right or left.

**Bitwise shift-right operator:**

- The bitwise shift right (>>) is a binary operator that requires two integral operands.
- The first operand is the value to be shifted.
- The second operand specifies the number of bits to be shifted.

Ex: `a >>= 5; /* shift right 5 bits */`

For unsigned data type, bits positions vacated by shift are filled with zeros.
7. Bitwise Right-Shift Operators

- When we shift a binary numbers, the right-shift operator divides by a power of 2.
- If we shift a binary number two places to the right, we are dividing by 4.
- If we shift it three places, we are dividing by 8.

<table>
<thead>
<tr>
<th>2^ shift value</th>
<th>Divide by</th>
<th>Shift Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>&gt;&gt;1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>&gt;&gt;2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>&gt;&gt;3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>&gt;&gt;4</td>
</tr>
<tr>
<td>.....</td>
<td>......</td>
<td>.....</td>
</tr>
<tr>
<td>n</td>
<td>2n</td>
<td>&gt;&gt;n</td>
</tr>
</tbody>
</table>
The bitwise shift right (<<) is a binary operator that requires two integral operands.

- The first operand is the value to be shifted.
- The second operand specifies the number of bits to be shifted.

Ex: `a <<= 3; /* shift right 5 bits */`

Bits positions vacated by shift are filled with zeros.
7. Bitwise Left-Shift Operators

- When we shift a binary numbers, the Left-shift operator multiplies by a power of 2.
- If we shift a binary number two places to the left, we are multiplying by 4.
- If we shift it three places, we are multiplying by 8.

<table>
<thead>
<tr>
<th>Shift Value</th>
<th>Multiplies by</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>&lt;&lt;1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>&lt;&lt;2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>&lt;&lt;3</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>&lt;&lt;4</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>......</td>
</tr>
<tr>
<td>n</td>
<td>2n</td>
<td>&lt;&lt;n</td>
</tr>
</tbody>
</table>

Program for bitwise shift operators u1_ex26.c
8. Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (, and ->).

The size of and the comma operators are discussed here.

The Comma Operator
The comma operator can be used to link related expressions together. A comma linked list of expressions are evaluated left to right and value of right most expression is the value of the combined expression.

For example the statement
value = (x = 10, y = 5, x + y);

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are In for loops:
for (n=1, m=10, n <=m; n++,m++)
In while loops    While (c=getchar(), c != ‘10’)
Exchanging values  t = x, x = y, y = t;
8. Special Operators

The size of Operator

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.

Example
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 the execution of the program.
Precedence
1. Precedence is used to determine the order in which different operators in a complex expression are evaluated.
2. C extends the concept to 15 levels.
   - Ex: $2 + 3 \times 4 = (2 + (3 \times 4)) = 14$
   - $-b++ = (-(b++))$

Associativity
1. Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression.
2. Precedence is applied before associativity to determine the order in which expressions are evaluated.
3. Associativity can be left-to-right or right-to-left.

Program for precedence of operators u1_ex34.c
**Precedence and Associativity**

**Left to Right Associativity**

All the operators have the same precedence. Their associativity is from left to right. so they are grouped as follows

Ex: 3 * 8 / 4 % 4 * 5

![Left to Right Associativity Diagram](image)

**Right to Left Associativity**

All the operators have the same precedence. Their associativity is from right to left. so they are grouped as follows

Ex: a += b *= c -= 5 which is (a=a+(b=b*(c=c-5)))

![Right to Left Associativity Diagram](image)

Program for Assocoativity of operators u1_ex35.c
## Precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses (function call)</td>
<td>1</td>
<td>left-to-right</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets (array subscript)</td>
<td>1</td>
<td>left-to-right</td>
</tr>
<tr>
<td>.</td>
<td>Member selection via object name</td>
<td>1</td>
<td>left-to-right</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Member selection via pointer</td>
<td>1</td>
<td>left-to-right</td>
</tr>
<tr>
<td>++</td>
<td>Postfix increment/decrement</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>--</td>
<td>Prefix increment/decrement</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+ -</td>
<td>Unary plus/minus</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>! ~</td>
<td>Logical negation/bitwise complement</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>(type)</td>
<td>Cast (convert value to temporary value of type)</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>*</td>
<td>Dereference</td>
<td>3</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address (of operand)</td>
<td>3</td>
<td>left-to-right</td>
</tr>
<tr>
<td>sizeof</td>
<td>Determine size in bytes on this implementation</td>
<td>3</td>
<td>left-to-right</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication/division/modulus</td>
<td>3</td>
<td>left-to-right</td>
</tr>
<tr>
<td>+ -</td>
<td>Addition/subtraction</td>
<td>4</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise shift left, Bitwise shift right</td>
<td>5</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
## Precedence and Associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt; &lt;=</code></td>
<td>Relational less than/less than or equal to</td>
<td>6</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>&gt;</code> <code>&gt;=</code></td>
<td>Relational greater than/greater than or equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>==</code> <code>!=</code></td>
<td>Relational is equal to/is not equal to</td>
<td>7</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Bitwise AND</td>
<td>8</td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Bitwise exclusive OR</td>
<td>9</td>
<td>left-to-right</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>Bitwise inclusive OR</td>
<td>10</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Logical AND</td>
<td>11</td>
<td>left-to-right</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
<td>Logical OR</td>
</tr>
<tr>
<td><code>? :</code></td>
<td>Ternary conditional</td>
<td>13</td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&amp;=</code> <code>^=</code> `</td>
<td>=<code> </code>&lt;&lt;=<code> </code>&gt;&gt;=`</td>
<td>Assignment</td>
<td>14</td>
</tr>
<tr>
<td>,</td>
<td>Comma (separate expressions)</td>
<td>15</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
Expressions & Evaluation of expressions

- An expression is a sequence of operands and operators that reduce to a single value.
- Expressions can be simple or complex.
- An operator is a syntactical token that requires an action be taken.
- An operand is an object on which an operation is performed; it receives an operator’s action.
- A simple expression contains only one operator.
  Ex: 2+5
- A complex expression contains more than one operator.
  Ex: 2+5*7
- An expression always reduces to a single value.
- We can divide simple expressions into six categories based on number of operands, relative position of the operand and operator and the precedence of operator.

expression categories

- Primary
- Postfix
- Prefix
- Unary
- Binary
- Ternary
The most elementary type of expression is a primary expression.
A primary expression consists of only one operand with no operator.
In C, the operand in the primary expression can be a name, a constant, or a parenthesized expression.

**Names:**
A name is any identifier for a variable, a function, or any other object in the language. Examples of some names used as primary expressions.
Ex: a  b12  price  calc  INT_MAX  SIZE

**Literal Constants:**
The second type of primary expression is the literal constant. A literal is a piece of data whose value cannot be changed during execution of the program.
Ex: 5 123.98  ‘A’  “Welcome”

**Parenthetical expressions:**
The third type of primary expression is the parenthetical expression. Any value enclosed in parentheses must be reducible to a single value and is therefore a primary expression.
Ex: (2 * 3+4) (a=23 + b*6).
Postfix Expressions

- The postfix expression consists of operand followed by operator.
- Some of the post fix expressions are function call, postfix increment, and postfix decrement.
- The form of postfix expression is

  ![Operand Operator Diagram]

**Function call:**
All the function calls are postfix expressions. The function is the operand and the operator is the parentheses that follows the name. parentheses may contain arguments or empty.
Ex: `printf("hello world")`  `scanf()`

**Postfix increment/decrement:**
In the post increment/ decrement, the variable value incremented/decremented by 1.

---

Program for postfix expressions u1_ex29.c
In prefix expressions, the operator comes before the operand.

**Prefix Increment/Decrement:**
1. In C, we have only two prefix operators that form prefix expressions:
   - prefix increment
   - prefix decrement
1. Prefix increment and decrement operators are shorthand notations for adding or subtracting 1 from variable.
2. The operand of a prefix expression must be a variable.
3. The value of (++)a is same as (a=a+1).

Program for prefix expressions u1_ex30.c
Note

If ++ is after the operand, as in a++, the increment takes place after the expression is evaluated.

If ++ is before the operand, as in ++a, the increment takes place before the expression is evaluated.
Unary Expressions

A unary expression, like a prefix expression, consists of one operator and one operand and also the operator comes before the operand.

`sizeof`

The `sizeof` operator tells us the size in bytes, of a type or a primary expression.

Ex: `sizeof(int)`, `sizeof(345.23)`,
`sizeof(x)`

### Unary plus/minus

<table>
<thead>
<tr>
<th>Expression</th>
<th>Contents of <code>a</code> Before and After Expression</th>
<th>Expression Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+a</code></td>
<td>3</td>
<td>+3</td>
</tr>
<tr>
<td><code>-a</code></td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td><code>+a</code></td>
<td>-5</td>
<td>-5</td>
</tr>
<tr>
<td><code>-a</code></td>
<td>-5</td>
<td>+5</td>
</tr>
</tbody>
</table>

### Cast operator

The cast operator converts one expression type to another.

Ex: convert an integer to real number `float(x)`
Binary Expressions

Binary expressions are formed by an operand-operator-operand combination.

Operator

Operand

Operand

Multiplicative Expressions

Multiplicative expressions include the operators multiply, divide, modulus operators. These operators have the higher priority among the binary operators and are therefore evaluated first.

Both operands of the modulo operator (%) must be integral type.

Ex:

- $10 \times 3$  // evaluates to 30
- $10 / 3$  // evaluates to 3
- $true \times 4$  // evaluates to 4
- $true / 4$  // evaluates to 0
- ‘A’$\times 2$  // evaluates to 130
- ‘A’$\div 2$  // evaluates to 32
- $22.3 \times 2$  // evaluates to 44.6
- $22.3 / 2$  // evaluates to 11/15
- $10 \% 3$  // evaluates to 1
- $3 / 5$  // evaluates to 0
- ‘A’$\% 2$  // evaluates to 5
- $3 \% 5$  // evaluates to 3
- $22.3 \% 2$  // error

Program for integer binary expressions u1_ex31.c
Additive Expressions

In additive expressions, the second operand is added to or subtracted from the first operand, depending on the operator used. Additive operators are evaluated after multiplicative expressions.

Ex:
3+7  //evaluates to 10
3-7  //evaluates to -4

Assignment Expressions

1. The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left.
2. The value of the total expression is the value of the expression on the right of the assignment operator (=)
3. The left operand in an assignment expression must be a single variable.
4. There are two forms of assignment
   - Simple assignment.
   - Compound assignment.

Program for floating point binary expressions u1_ex32.c
Simple Assignment

1. Simple assignment is found in algebraic expressions.
2. Three examples of simple assignments are shown below:
   \[ a = 5 \quad b = x + 1 \quad i = i + 1 \]

Compound Assignment

1. A compound assignment is a shorthand notation for a simple assignment.
2. It requires that the left operand be repeated as a part of the right expression.

<table>
<thead>
<tr>
<th>Compound Expression</th>
<th>Equivalent Simple Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x *= \text{expression} )</td>
<td>( x = x * \text{expression} )</td>
</tr>
<tr>
<td>( x /= \text{expression} )</td>
<td>( x = x / \text{expression} )</td>
</tr>
<tr>
<td>( x %= \text{expression} )</td>
<td>( x = x % \text{expression} )</td>
</tr>
<tr>
<td>( x += \text{expression} )</td>
<td>( x = x + \text{expression} )</td>
</tr>
<tr>
<td>( x -= \text{expression} )</td>
<td>( x = x - \text{expression} )</td>
</tr>
</tbody>
</table>

Program for compound assignment expressions u1_ex33.c
Now we evaluate expressions as we know precedence, associativity of operators

**Expressions without side effects**

Ex: \( a \times 4 + b / 2 - c \times b \) consider \( a=3 \), \( b=4 \), \( c=5 \)

Replace variables by their values
\( 3 \times 4 + 4 / 2 - 5 \times 4 \)

Apply precedence rules
\( (3 \times 4) + (4 / 2) - (5 \times 4) \)

Apply Associativity rules as they are left to right associative
\( (((3 \times 4) + (4 / 2)) - (5 \times 4)) \)

In evaluation of this expressions there are no side effects
Expressions with side effects

Now look for example with side effects and parenthesized expressions.
Ex: \(-a \times (3 + b) / 2 - c++ \times b\)
Assume \(a=3\) \(b=4\) \(c=5\)

1. Calculate the value of parenthesized expressions
   \(-a \times 7 / 2 - c++ \times b\)

1. Evaluate postfix expression
   \(-a \times 7 / 2 - 5 \times b\)

1. Evaluate prefix expression
   \(2 \times 7 / 2 - 5 \times b\)

1. Now apply multiply and division
   \(14 / 2 - 5 \times 4\)

1. Last step is to evaluate subtraction
   \(7 - 20 = -13\)

After the side effects the variables have the values shown below
\(a=2\) \(b=4\) \(c=6\)

**Warning**

In C, if a single variable is modified more than once in an expression, the result is undefined.
Up to this point, we have assumed that all of our expressions involved the same type. But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number? To perform these evaluations, one of the types must be converted.

Type conversion is of two types:
1. Implicit Type Conversion
2. Explicit Type Conversion (Cast)

Implicit Type Conversion

When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as implicit type conversion.

Conversion rank:
1. We assign a rank to the integral and floating point arithmetic types.
2. These ranks are known as conversion ranks.
3. Conversion ranks are shown below
Conversion Rank

Program for implicit type conversion  u1_ex36.c
A simple assignment involves an assignment operator and two operands. Depending on the difference in the rank, C tries to either promote or demote the right expression to make it the same rank as the left variable.

1. Promotion occurs if the expression has lower rank.
   1. There is normally no problem with promotion.
   2. The rank of right expression is elevated to the rank of the left variable.
   3. The value of the expression is the value of the right expression after the promotion.

Ex:  

```c
bool  b= true
char c= ‘A’
int i=1234
long double d=3458.0004

c=b            //value of c is SOH     (ASCII 1)
i= c            //value of i is 65
d=b            //value of d is is 1.0

d=i            // value of d is 1234.0
```
1. Demotion

Demotion may or may not create problems. If the size of the variable at the left side can accommodate the value of the expression, then there is no problem. If the size of the variable at the left side can not accommodate the value of the expression, then demotion occurs.

Ex:

```c
bool b = false
char c = 'A'
short int s = 78
int j = 32200
int k = 65
b=c  // value of b is 1 (true)
s=j  // value of s is unpredictable
c=k+1  // demotion value of c is 'B'
```
Conversion has a different set of rules for other binary expressions.

1. The operand with higher rank is determined using the ranking table.
2. The lower ranked operand is promoted to the rank defined in step 1. After the promotion both expressions have the same rank.
3. The operation is performed with the expression value having the type of the promoted rank.

Ex:

```c
bool b = true;
char c = 'A'
int i = 3650
short int s = 78
long double d = 3458.0004

b + c       // b promoted: result is 'B'
i * s       // result is an integer
d * c       // result is long double
```
Explicit type conversion (cast)

1. In explicit type conversion we convert the data from one type to another type using explicit type conversion.
2. Explicit type conversion uses the unary type cast operator.
3. To cast the data from one type to another, we specify the new type in parentheses before the value we want to convert.

Ex:  \((\text{float}) \, a\)

One use of the cast is to ensure that the result of a divide is a real number.

Ex:  \((\text{float}) \, \text{totalscores}/ \, \text{numscore}\)

  here totalscores is integer
  numscore is int
  but result is float due type cast operator before totalscores

Program for explicit type conversion  u1_ex37.c
A statement causes an action to be performed by the program. It translates directly into one or more executable computer instructions.

1. You may have noticed that we have used a semicolon at the end of the statements in our programs.
2. Most statements need a semicolon at the end; some do not.
3. C defines eleven types of statements which are shown below.

The role of the semicolon

The semicolon plays an important role in the syntax of the C language. It is used in two different auscultations.
1. Every declaration in C is terminated by a semicolon.
2. Most statements in C are terminated by a semicolon.
3. We must be careful not to use a semicolon when it is not needed.
4. A semicolon should not be used with a preprocessor directive such as the `include` and `define`. 
Null Statement

1. The null statement is just a semicolon (the terminator) as shown below:

   ;  // null statement

1. Although they do not arise often, there are syntactical situations where we must have a statement but no action. In these situations, we use the null statement.

Expression Statement

An expression is turned into a statement by placing a semicolon (;) after it.

**Syntax:** expression;

Ex:

```
a=2;

a=(b=3);
b;
3;
a++;
ioresult=scanf("%d",&x);
um=printf(" x contains %d, y contains %d", x, y);
```
Return Statement

1. A return statement terminates a function.
2. All functions including main, must have a return statement.
3. When there is no return statement at the end of the function, the system inserts one with a void return value.

Syntax: `return expression;`

1. The return statement can return a value to the calling function.
   Ex: `scanf` function returns the number of variables successfully read.

   In the case main, it returns a value to the operating system rather than to another function.

Compound Statement

1. A compound statement is a unit of code consisting of zero or more statements. It also known as a `block`.
2. The compound statement allows a group of statements to become one single entity.
3. A compound statement consists of an opening brace, an optional declaration and definition section, followed by a closing brace.
The compound statement does not need a semicolon.
We have seen that a C program is a set of statements that are normally executed sequentially in the order in which they appear.

However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specific conditions are met.

This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision making capabilities by supporting the following statements:

- if statement
- switch statement
- goto statement

These statements are popularly known as **decision making statements**. These statements control the flow of execution they are also known as **control statements**.
The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression takes the following form

**If (test expression)**

it allows the computer to evaluate the expression first and then depending on whether the value of the expression is true or false. It transfers the control to a particular statement.
1. The if statement may be implemented in different forms depending on the complexity of conditions to be tested.

2. The different forms are:
   - simple if statement
   - if …. else statement
   - nested if… else statement
   - else if ladder.

**Simple if statement**

The general form of simple if statement is:

```c
if (test expression) {
    statement-block;
}
```

**statement-X**

If test expression is `true`, then `statement block` will be executed; otherwise the statement block will be `skipped` and the execution will jump to the `statement-X`.
The if...else statement

The if...else statement is an extension of the simple if statement. It takes the following general form:

```c
if ( test expression)
{
  true-block statement(s)
}
else
{
  false-block statement(s)
}
```

**statement-x**

If the test expression is true, then the true block statements are executed. If the test expression is false, then the false block statements are executed. It never executes both blocks at a time. In both cases the control is transferred to the statement-X.
Statements

1. The expression must be enclosed in parentheses.

2. No semicolon (;) is needed for an if...else statement; statement 1 and statement 2 may have a semicolon as required by their types.

3. The expression can have a side effect.

4. Both the true and the false statements can be any statement (even another if...else statement) or they can be a null statement.

5. Both statement 1 and statement 2 must be one and only one statement. Remember, however, that multiple statements can be combined into a compound statement through the use of braces.

6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.

Syntactical Rules for if...else Statements
The Nesting of if...else statement

When a series of decisions are involved, we may have to use more than one if... else statement in nested form.

```c
if (test condition-1)
{
    if (test condition-2)
    {
        statement-1;
    }
    else
    {
        statement-2;
    }
}
else
{
    statement-3;
}
statement-X
```
The Nesting of if...else statement

Flow chart

Program for nested if else statement u1_ex40.c
1. Once we start nesting if….else statements, we encounter a classical problem known as the dangling else.
2. This problem is created when there is no matching else for every if.  
3. `else` is always paired with the most recent unpaired `if`.

![Diagram of if-else statements with dangling else]

- (a) Code
  ```
  if (expression1)
    if (expression2)
      statement 1
      statement 2
    else
      statement 2
  
  Compiler pairs this if and else!
  ```

- (b) Logic Flow
  - `expression 1`
  - `expression 2`
  - `statement 2` to `statement 1`

**Dangling else**
1. There is another of putting ifs together when multipath decisions are involved.
2. A multipath decision is a chain of ifs in which the statement associated with each else is an if.
3. It takes general form, as shown below.

```plaintext
if(condition-1)
  statement-1;
else if (condition-2)
  statement-2;
else if(condition-3)
  statement-3
else if(condition n)
  statement-n;
else
  default statements;
statement-X
```
The else if Ladder Flow chart

Program for else if ladder statement u1_ex41.c
In addition to two-way selection, most programming languages provide another selection concept known as multiway selection.

Multiway selection chooses among several alternatives.

C has two different ways to implement multiway selection:

The switch statement and

else-if construct. (difficult if no of alternatives are more).

The switch statement can be used only when the selection condition reduces to an integral expression.

Many times, when the selection is based on a range of values, the condition is not an integral. In these cases we use else-if which we already discussed.
Switch statement

```java
switch(expression)
{
    case constant-expression:
        statement(s);
        break; /* optional */

    case constant-expression:
        statement(s);
        break; /* optional */

    default: /* Optional */
        statement(s);
}
```
Switch statement flow chart

- **multiway expression**
  - value 1: value1 action
  - value 2: value2 action
  - value 3: value3 action
  - value 4: value4 action
Switch statement

1. The control expression that follows the keyword `switch` must be an integral type.

2. Each `case` label is the keyword `case` followed by a constant expression.

3. No two `case` labels can have the same constant expression value.

4. But two `case` labels can be associated with the same set of actions.

5. The `default` label is not required. If the value of the expression does not match with any labeled constant expression, the control transfers outside of the `switch` statement. However, we recommend that all `switch` statements have a `default` label.

6. The `switch` statement can include at most one `default` label. The `default` label may be coded anywhere, but it is traditionally coded last.
Switch statement

1. If we want to execute only one of the case label sequences, we must use break statements.
2. The break statement causes the program to jump out of the switch statement—that is, to go to the closing brace and continue with code that follows the switch.
3. If we add a break as the last statement in each case.

Sample program with out break

```c
1  // Program fragment to demonstrate switch
2     switch (printFlag)
3     {
4         case 1:  printf("This is case 1\n");
5
6         case 2:  printf("This is case 2\n");
7
8         default: printf("This is default\n");
9     } // switch
```
Program for switch statement with out break u1_ex42.c
Sample program with break

Program for switch statement with break u1_ex43.c

(a) Logic Flow

(b) Code

switch (printFlag)
{
    case 1:
        printf("This is case 1");
        break;
    case 2:
        printf("This is case 2");
        break;
    default:
        printf("This is default");
        break;
} // switch
1. The real power of computers is in their ability to repeat an operation or a series of operations many times.
2. This repetition, called looping, is one of the basic structured programming concepts.
3. Each loop must have an expression that determines if the loop is done. If it is not done, the loop repeats one more time; if it is done, the loop terminates.

**Concept of a loop**

1. The concept of a loop is shown in the flowchart. Here the action will be repeated forever.
2. We definitely do not want this to happen.
3. We want our loop to end when the work is done.
4. To make sure that it ends, we must have a condition that controls the loop.
5. We must design the loop so that before or after each iteration, it checks to see if the task is done.
6. If not done, the loop repeats one more time;
7. If the task is done, the loop terminates.
8. This test is known as a *loop control expression*. 
1. We need to test for the end of a loop, but where should we check it—before or after each iteration? We can have either a pre- or a post-test terminating condition.
2. Programming languages allows us to check the loop control expression either before or after each iteration of the loop.
3. In a pretest loop, the condition is checked at the beginning of each iteration.
4. In a post-test loop, the condition is checked at the end of each iteration.

**Pretest Loop**

In each iteration, the control expression is tested first. If it is true, the loop continues; otherwise, the loop is terminated.

**Post-test Loop**

In each iteration, the loop action(s) are executed. Then the control expression is tested. If it is true, a new iteration is started; otherwise, the loop terminates.
(a) Pretest Loop

(b) Post-test Loop

Program for pre test loop u1_ex44.c
Two Different Strategies for Doing Exercises

Program for post test loop u1_ex45.c
Minimum Number of Iterations in Two Loops

(a) Pretest

Test 1
false
exit
true
Body 1

In a pretest loop, the body may not be executed.

(b) Post-test

Body 1
true
Test 1
false
exit

In a post-test loop, the body must be executed at least once.
In addition to the loop control expression, two other processes, initialization and updating, are associated with almost all loops.

Loop Initialization

1. Loop initialization must be done before the first execution of the loop body.
2. Initialization may be explicit or implicit.
3. When the initialization is explicit, we include code to set the beginning values of key loop variables.
4. Implicit initialization provided no direct initialization code; rather, it relies on a preexisting situation to control the loop.

Loop Update

How can the condition that controls the loop be true for a while and then change to false? The answer is that something must happen inside the body of the loop to change the condition.

1. The actions that cause these changes are known as loop update.
2. Update is done in each iteration, usually as the last action.
3. If the body of the loop is repeated m times, then updating is also done m times.
Loop Initialization and Updating

(a) Pretest Loop

(b) Post-test Loop
Repetition (1 of 15)

(a) Pretest Loop

(initialization and updating for exercise)

(b) Post-test Loop
All the possible expressions that can be used in a loop limit test can be summarized into two general categories: event-controlled loops and counter-controlled loops.

**Event-Controlled Loops**

1. In an event controlled loop, an event changes the control expression from true to false.
2. In event controlled loops, the updating process can be explicit or implicit.
3. If it is explicit, such as finding a specific piece of information, it is controlled by the loop.
4. If it is implicit, such as the temperature of a batch of chemicals reaching a certain point, it is controlled by some external condition.
Event-controlled Loop Concept

(a) Pretest Loop

(b) Post-test Loop

Program for event controlled loop u1_ex46.c
1. When we know the number of times an action is to be repeated, we use a counter controlled loop.
2. We must initialize, update and test the counter.
3. Although we need to know the number of times we want to execute the loop, the number does not have to be a constant. It can also be a variable or a calculated value.
4. The update can be increment, in which case we are counting up, or a decrement, in which case we are counting down.
Counter-controlled Loop Concept

Program for counter controlled loop u1_ex47.c
1. The number of iterations of a loop is given $n$.
2. In a pretest loop, when we come out of the loop, the limit test has been executed $n+1$ times.
3. In a post test loop, when we come out of the loop, the limit test has been executed only $n$ times.

<table>
<thead>
<tr>
<th>Pretest Loop</th>
<th>Post-test Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization:</strong></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Number of tests:</strong></td>
<td></td>
</tr>
<tr>
<td>$n + 1$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>Action executed:</strong></td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>Updating executed:</strong></td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>Minimum iterations:</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
while, for and do-while loops

There are three loop statements:
- the while
- the for and
- the do...while.

The first two are pretest loops.
and the third is a post-test loop.

1. We can use all of them for event-controlled and counter-controlled loops.
2. All three of these loop constructs continue when the limit test is true and terminate when it is false.

Loops

- while: Pretest Loop
- for: Pretest Loop
- do...while: Post-test Loop
The while loop

1. The while loop is a pretest loop.
2. The while loop uses an expression to control the loop.
3. Because it is a pretest loop, it tests the expression before every iteration of the loop.
4. No semicolon is needed at the end of the while statement.
5. Statements within the while loop ends with semicolon.

The simple while loop (body contains single statement)
The compound while loop (body contains more than one statement)

(a) Flowchart

(b) C Language

while ( expression )
{
    Action
    ... 
    Action
}

// while

Program for while loop u1_ex48.c
The for loop

1. The for statement is a pretest loop that uses three expressions. The first expression contains an initialization statement. The second contains the limit-test expression. The third contains the updating expression.

2. A `for` loop is used when a loop is to be executed a known number of times.

3. We can do the same thing with a `while` loop, but the `for` loop is easier to read and more natural for counting loops.

4. The `for` loop uses three expression in a single line, each expression is separated by a semi-colon.
The simple for loop (body contains single statement)

```
for (expr1; expr2; expr3)  
  statement
```
The compound for loop (body contains more than one statement)
The comparison of for and while loops

Program for for loop u1_ex49.c
The do…while loop

1. The do…while statement is a post test loop.
2. Like the while and for loops, it also uses an expression to control the loop.
3. The do…while loop tests the expression after the execution of the body.
4. The do…while loop is concluded with a semicolon.
5. The body of the do…while loop is executed at least once.
The simple & compound do…while loop

Program for do...while_loop u1_ex50.c
The comma expression

1. A comma expression is a complex expression made up of two expressions separated by a comma.
2. It is most often used in for statements.
3. The expressions are evaluated from left to right.
4. The value and type of the expressions are the value and type of the right expression.
5. The comma expression has the lowest priority of all expressions.

expression, expression, expression

Program for for loop with comma operator u1_ex51.c
Three other C statements are related to loops:

break
continue
goto.

**Break**

1. In a loop, the break statement causes a loop to terminate.
2. If we are in a series of nested loops, break terminates only inner loop—the one we are currently in.
3. Break statements needs a semicolon.
4. The break statements can be used in any of the loops—while, do...while, for and in the selection switch statement.
5. Good structured programming limits its usage.
\begin{verbatim}
while (condition) {
    ...
    for ( ...; ...; ... ) {
        ...
        if (otherCondition) break;
        ...
    } // for
    // more while processing
    ...
} // while
\end{verbatim}

The \texttt{break} statement takes us out of the inner loop (the \texttt{for} loop). The \texttt{while} loop is still active.

\textit{break} and Inner Loops
### Program for break statement u1_ex52.c

<table>
<thead>
<tr>
<th>Statements</th>
<th>Statements</th>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 // A bad loop style</td>
<td>2 // A better loop style</td>
<td>3 // A better loop style</td>
</tr>
<tr>
<td>2 for ( ; ; )</td>
<td>3 for ( ; !condition ; )</td>
<td>4 {</td>
</tr>
<tr>
<td>4 {</td>
<td>5 } // for</td>
<td>6 } // for</td>
</tr>
<tr>
<td>5 ...</td>
<td>7 } // for</td>
<td>8 } // for</td>
</tr>
<tr>
<td>6 if (condition)</td>
<td>9 break;</td>
<td>10 } // for</td>
</tr>
<tr>
<td>7 } // for</td>
<td>11 } // for</td>
<td>12 } // while</td>
</tr>
</tbody>
</table>

```
1 while (x)
2 {
3   ...
4   if (condition)
5     break;
6   else
7     ...
8 } // while
```

```
1 while (x && !condition)
2 {
3   ...
4   if (!condition)
5     ...;
6 } // while
```
The continue statement does not terminate the loop but simply transfers the control to the test expression in while and do...while statements and transfers to the updating expression in a for statement.

Although the transfer is to different positions in pretest and post test loops, both can logically thought of as a jump to the end the loops body.

The use of continue is also considered as unstructured programming.

Program for continue statement u1_ex53.c
The goto

1. C supports the goto statement to branch unconditionally one point to another point in the program.
2. Use of `goto` statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify.
3. Any program that uses a goto can be rewritten so that it doesn't need the goto.

**Syntax:**
The syntax for a `goto` statement in C is as follows:

```plaintext
goto label;
```

Label: statement;

Here `label` can be any plain text except C keyword and it can be set anywhere in the C program above or below to `goto` statement.
Program for goto statement u1_ex54.c, u1_ex55.c