UNIT IV Introduction to Programming

Programming Language

A **programming language** is a vocabulary and set of grammatical rules for instructing a **computer** or computing device to perform specific tasks. The term **programming language** usually refers to high-level **languages**, such as C, C++, COBOL, Java, FORTRAN, A dak and Pascal.

Machine Language: A computer programming language consisting of binary instructions which a computer can respond to directly.

Sometimes it is referred to as **machine** code or object code, **machine language** is a collection of binary digits or bits that the computer reads and interprets. A computer cannot directly understand the **programming languages** used to create computer programs, so the program code must be compiled. Example: 01001000, 01100101, 01101100, 01101100 etc

Advantage:

- This language makes fast and efficient use of the computer.
- It requires no translator to translate the code. It is directly understood by the computer.

Disadvantage:

- All memory addresses have to be remembered.
- All operation codes have to be remembered.

Assembly language:

An assembly language is a low-level programming language in which there is a very strong correspondence between the program's statements and the architecture's machine code instructions. Assembly code is converted into executable machine code by a utility program referred to as an assembler.

A program written in assembly language consists of a series of mnemonic processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters. These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

Example: MOV AL 61h ;(Meaning – Load AL with 61 hex, MOV is abbreviation of Move)

Advantages Of Assembly Language

1. Programs written in machine language are replaceable by mnemonics which are easier to remember.
2. Memory Efficient.
3. It is not required to keep track of memory locations.

4. Faster in speed.
5. Easy to make insertions and deletions.
6. Hardware Oriented.
7. Requires fewer instructions to accomplish the same result.

Disadvantages Of Assembly Language

1. Long programs written in such languages cannot be executed on small sized computers.
2. It takes lot of time to code or write the program, as it is more complex in nature.
3. Difficult to remember the syntax.
4. Lack of portability of program between computers of different makes.

High-level language: A high-level language is any programming language that enables development of a program in a much more user-friendly programming context.

This language is a programming language with strong abstraction about the details of the computer in contrast to low-level programming language (Assembly Language).

Ex: C, C++, Java

High level languages are grouped in two categories based on execution model – **compiled** or **interpreted** languages. Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. **Compiler** read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas **interpreter** read only one line of the source code and convert it to object code.

Advantages of High level language

1. High level languages are programmer friendly. They are easy to write, debug and maintain.
2. It provide higher level of abstraction from machine languages.
3. It is machine independent language.
4. Easy to learn.
5. Less error prone, easy to find and debug errors.
6. High level programming results in better programming productivity.

Disadvantages of High level language

1. It takes additional translation times to translate the source to machine code.
2. High level programs are comparatively slower than low level programs.
3. Compared to low level programs, they are generally less memory efficient.
4. Cannot communicate directly with the hardware.

DATA TYPES

Basic concepts – data types

Let's discuss about a very simple but very important concept available in almost all the programming languages which is called **data types**. As its name indicates, a data type represents a type of the data which you can process using your computer program. It can be numeric, alphanumeric, decimal, etc.

Let’s keep Computer Programming aside for a while and take an easy example of adding two whole numbers 10 & 20, which can be done simply as follows –

100 + 200

Let's take another problem where we want to add two decimal numbers 100.50 & 200.50, which will be written as follows –

100.50 + 200.50

The two examples are straight forward. Now let's take another example where we want to record student information in a notebook. Here we would like to record the information like Name, Class, Section: A and Age

Now, let's put one student record as per the given requirement −

Name: Ram Kumar

Class: 12th

Section: A

Age: 25

The first example dealt with whole numbers, the second example added two decimal numbers, whereas the third example is dealing with a mix of different data. Let's put it as follows −

- Student name "Ram Kumar" is a sequence of characters which is also called a string.
- Student class "12th" has been represented by a mix of whole number and a string of two characters. Such a mix is called alphanumeric.
- Student section has been represented by a single character which is 'A'.
- Student age has been represented by a whole number which is 25.

This way, we realized that in our day-to-day life, we deal with different types of data such as strings, characters, whole numbers (integers), and decimal numbers (floating point numbers).

Similarly, when we write a computer program to process different types of data, we need to specify its type clearly; otherwise the computer does not understand how different operations can be performed on that given data.

Different programming languages use different keywords to specify different data types. For example, **C** and **Java** programming languages use **int** to specify integer data, whereas **char** specifies a character data type.

C and Java Data Types

C and Java support almost the same set of data types, though Java supports additional data types. For now, we are taking a few common data types supported by both the programming languages −

Type	Keyword	Value range which can be represented by this data type
Character	char	-128 to 127 or 0 to 255
Number	int	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
Small Number	short	-32,768 to 32,767
Long Number	long	-2,147,483,648 to 2,147,483,647
Decimal Number	float	1.2E-38 to 3.4E+38 till 6 decimal places

These data types are called primitive data types and you can use these data types to build more complex data types, which are called user-defined data type, for example a string will be a sequence of characters.

Python Data Types

Python has five standard data types but this programming language does not make use of any keyword to specify a particular data type, rather Python is intelligent enough to understand a given data type automatically, like - number, string etc. Here, Number specifies all types of numbers including decimal numbers and string represents a sequence of characters with a length of 1 or more characters.

Data types Representation in programming

Variables are the names you give to computer memory locations which are used to store values in a computer program.

For example, assume you want to store two values 10 and 20 in your program and at a later stage, you want to use these two values. Let's see how you will do it. Here are the following three simple steps −

- Create variables with appropriate names.
- Store your values in those two variables.
- Retrieve and use the stored values from the variables.

1. **Creating variables**

Creating variables is also called **declaring variables** in C programming. Different programming languages have different ways of creating variables inside a program. For example, C programming has the following simple way of creating variables −

```
#include <stdio.h>
int main() {
  int a;
  int b;
}
```

The above program creates two variables to reserve two memory locations with names **a** and **b**. We created these variables using **int** keyword to specify variable **data type** which means we want to store integer values in these two variables. Similarly, you can create variables to store **long**, **float**, **char** or any other data type.

You can create variables of similar type by putting them in a single line but separated by comma as **int a, b;**

Important key points about variables

- A variable name can hold a single type of value. For example, if variable **a** has been defined **int** type, then it can store only integer.
- C programming language requires a variable creation, i.e., declaration before its usage in your program. You cannot use a variable name in your program without creating it, though programming language like Python allows you to use a variable name without creating it.
- You can use a variable name only once inside your program. For example, if a variable **a** has been defined to store an integer value, then you cannot define **a** again to store any other type of value.
- There are programming languages like Python, PHP, Perl, etc., which do not want you to specify data type at the time of creating variables. So you can store integer, float, or long without specifying their data type.

Every programming language provides more rules related to variables and you will learn them when you will go in further detail of that programming language.

2. Store Values in Variables

You have seen how we created variables in the previous section. Now, let's store some values in those variables − Following is a C program, which stored values in variables

```
#include <stdio.h>
int main() {
   int a;
   int b;
   a = 10;
   b = 20;
}
```

The above program has two additional statements where we are storing 10 in variable **a** and 20 is being stored in variable **b**.

Almost all the programming languages have similar way of storing values in variable where we keep variable name in the left hand side of an equal sign = and whatever value we want to store in the variable, we keep that value in the right hand side.

3. Access stored values in variables

If we do not use the stored values in the variables, then there is no point in creating variables and storing values in them. We know that the above program has two variables **a** and **b** and they store the values 10 and 20, respectively. So let's try to print the values stored in these two variables. Following is a C program, which prints the values stored in its variables −

```
#include <stdio.h>
int main() {
   int a;
   int b;
   a = 10;
   b = 20;
   printf( "Value of a = %d\n", a );
   printf( "Value of b = %d\n", b );
}
```

When the above program is executed, it produces the following result −

```
Value of a = 10
Value of b = 20
```

In this program, **printf()** function is used to display the message or value of the variable or both on computer screen, use of **%d**, which will be replaced with the values of the given variable in printf() statements.

Equivalent program written in Python

```
a = 10
b = 20
print "Value of a = ", a
print "Value of b = ", b
```

When the above program is executed, it produces the following result −

```
Value of a = 10
Value of b = 20
```

OPERATORS

An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce final result. In this section you will learn the concept of important arithmetic and relational operators available in C.

Arithmetic Operators

Important arithmetic operators are available in C programming language. Assume variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	This gives remainder of an integer division	B % A will give 0

You can understand the use of above arithmetic operators with following C program.

```
#include <stdio.h>
int main() {
```

```
   int a, b, c;
   a = 10;
   b = 20;

   c = a + b;
   printf( "Value of c = %d\n", c);

   c = a - b;
   printf( "Value of c = %d\n", c);

   c = a * b;
   printf( "Value of c = %d\n", c);

   c = b / a;
   printf( "Value of c = %d\n", c);

   c = b % a;
   printf( "Value of c = %d\n", c);
}
```

When the above program is executed, it produces the following result −

```
Value of c = 30
Value of c = -10
Value of c = 200
Value of c = 2
Value of c = 0
```

Relational Operators

Consider a situation where we create two variables and assign them some values as follows −

```
A = 20
B = 10
```

Here, it is obvious that variable A is greater than B in values. So, we need the help of some symbols to write such expressions which are called relational expressions. If we use C programming language, then it will be written as follows −

```
(A > B)
```

Here, we used a symbol > and it is called a relational operator.

Relational operators available in C programming language. In below table assume A=10 and B=20 then -.

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

You can understand the use of above operators in C programming using following program.

```
#include <stdio.h>
int main() {
   int a, b;

   a = 10;
   b = 20;

   /* Here we check whether a is equal to 10 or not */
   if( a == 10 ) {

      /* if a is equal to 10 then this body will be executed */
      printf( "a is equal to 10\n");
   }
```

```
   /* Here we check whether b is equal to 10 or not */
   if( b == 10 ) {

      /* if b is equal to 10 then this body will be executed */
      printf( "b is equal to 10\n");
   }

   /* Here we check if a is less b than or not */
   if( a < b ) {

      /* if a is less than b then this body will be executed */
      printf( "a is less than b\n");
   }

   /* Here we check whether a and b are not equal */
   if( a != b ) {

      /* if a is not equal to b then this body will be executed */
      printf( "a is not equal to b\n");
   }
}
```

When the above program is executed, it produces the following result −

```
a is equal to 10
a is less than b
a is not equal to b
```

Logical Operators

Logical operators are very important in any programming language and they help us take decisions based on certain conditions. Suppose we want to combine the result of two conditions, then logical AND and OR logical operators help us in producing the final result.

The following table shows all the logical operators supported by the C language. Assume variable **A** holds 1 and variable **B** holds 0, then −

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.

!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

You can understand use of all the logical operators available in C using following program

```
#include <stdio.h>
int main() {
  int a = 1;
  int b = 0;
  if ( a && b ) {
    printf("This will never print because condition is false\n" );
  }
  if ( a || b ) {
    printf("This will be printed print because condition is true\n" );
  }
  if ( !(a && b) ) {
    printf("This will be printed print because condition is true\n" );
  }
}
```

When you compile and execute the above program, it produces the following result −

```
This will be printed print because condition is true
This will be printed print because condition is true
```

CONTROL STATEMENTS

Control Statements: C provides two sytles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching: Branching is so called because the program chooses to follow one branch or another.

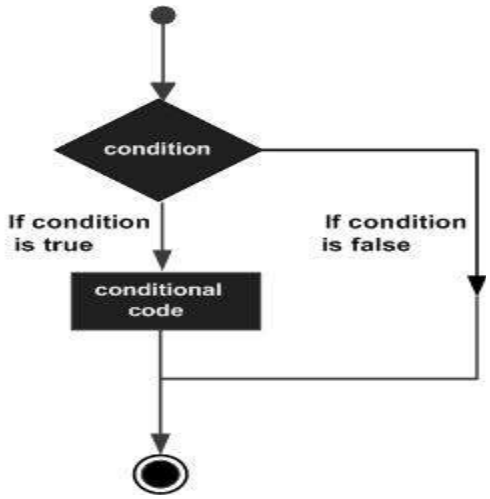
if Statement : This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

Syntax of If Statement

```
if (expression)
  statement;
or
if (expression)
  {
    Block of statements;
  }
```

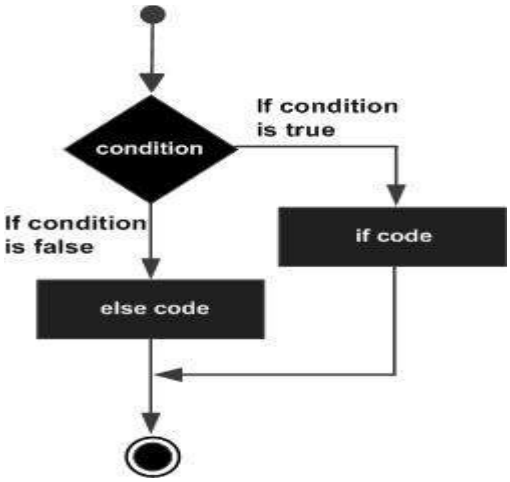
Almost all the programming languages provide this statement that work based on the following flow diagram −



IF … else satatement: if statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. The syntax of an **if...else** statement in C programming language is −

```
if (expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }
```

The above syntax can be represented in the form of a flow diagram as shown below −



You can understand the concept of conditional statements in C programming with following program.

For example, Find the largest of two numbers, if the numbers are a=30 and b=50

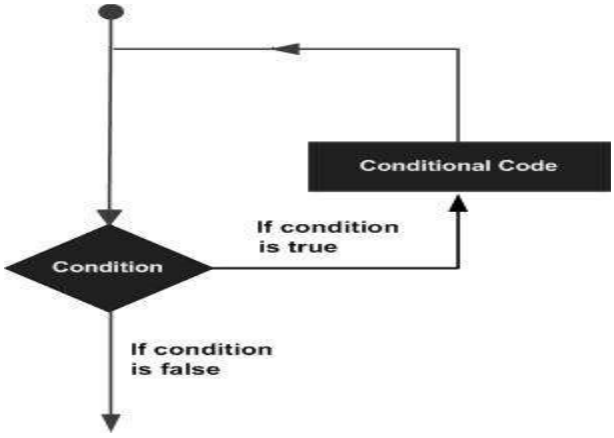
```
#include <stdio.h>
int main() {
  int a = 30, b=50;
   if( a > b ) {
                printf( "Largest number is %d\n", a);
   } else {
     printf( "Largest number is %d\n", b);
   }
}
```

When the above program is executed, it produces the following result −

```
Largest number is 50
```

LOOPING

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C provides a number of looping way.

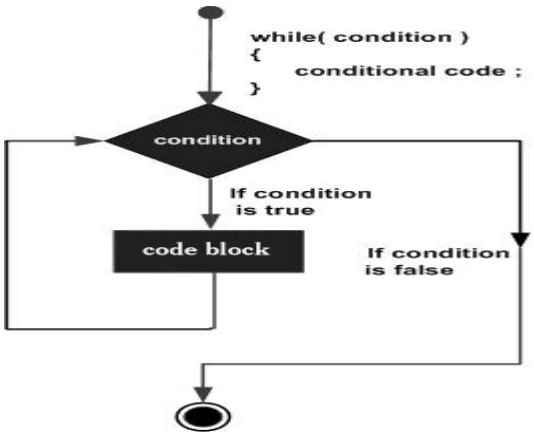
while loop

The most basic loop in C is the while loop. A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statements get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

```
while ( expression )
{
   Single statement
   or
   Block of statements;
}
```

The above code can be represented in the form of a flow diagram as shown below −



for loop

for loop is similar to while, it's just written differently. for statements are often used to proccess lists such a range of numbers:

Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
   Single statement
   or
   Block of statements;
}
```

In the above syntax:

- expression1 - Initializes variables.
- expression2 - Conditional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

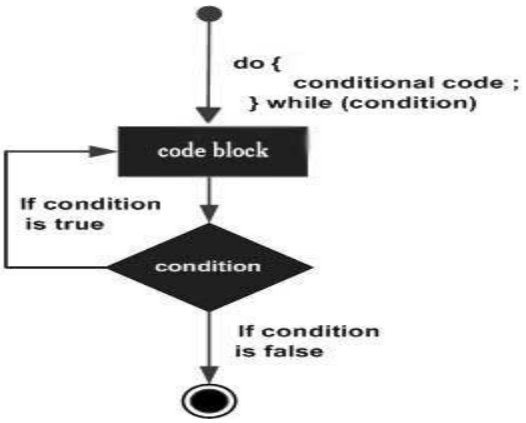
do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

```
do
{
   Single statement
   or
   Block of statements;
}while(expression);
```

The above code can be represented in the form of a flow diagram as shown below −



You can understand the concept of the various loops by execution of the following **C** programming

Using for loop	**Using while loop**	**Using do…while loop**

`#include <stdio.h>`
 	
`int main()`
`{`
`  int i;`
`  for (i=0; i<5; i++)`
`  {`
`  printf( "Hello %d\n", i);`
`  }`
`}`
	`#include <stdio.h>`
	
	`int main()`
	`{`
	`  int i = 0;`
	`  while ( i < 5 ) {`
	`  printf( "Hello %d\n", i);`
	`  i = i + 1;`
	`  }`
	`}`
		`#include <stdio.h>`
		`int main()`
		`{`
		`  int i = 0;`
		`  do{`
		`  printf( "Hello %d\n",i);`
		`  i = i + 1;`
		`  }`
		`  while ( i < 5 );`
		`}`

Output Hello 0 Hello 1 Hello 2 Hello 3 Hello 4	**Output** Hello 0 Hello 1 Hello 2 Hello 3 Hello 4	**Output** Hello 0 Hello 1 Hello 2 Hello 3 Hello 4

The break statement

When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop. The syntax for a **break** statement in C is as follows −

```
break;
```

The continue statement

The **continue** statement in C programming language works somewhat like the **break** statement. Instead of forcing termination, **continue** forces the next iteration of the loop to take place, skipping any code in between. The syntax for a **continue** statement in C is as follows −

```
continue;
```

You can understand the concept of the break and continue statements by execution of the following programs in C programming.

Break statement	**Continue statement**

```c
#include <stdio.h>
int main() {
   int i = 0;
   do {
      printf( "Hello %d\n", i);
      i = i + 1;

      if( i == 2 ) {
         break;
      }
   }
   while ( i < 5 );
}
```

When the above program is executed, it produces the following result −

```
Hello 0
Hello 1
```

```c
#include <stdio.h>
int main() {
   int i = 0;
   do {
      if( i == 2 ) {
         i = i + 1;
         continue;
      }
      printf( "Hello %d\n", i);
      i = i + 1;
   }
   while ( i < 5 );
}
```

When the above program is executed, it produces the following result −

```
Hello 0
Hello 1
Hello 3
Hello 4
```