CCA-101: Fundamentals of IT & Programming

Assignment – 2

Q1. What is the difference between Machine Language and High Level Language?

Ans.

Programming Languages

A program is a set of instructions that tells a computer what to do in order to come up with a solution to a particular problem. Programs are written using a programming language.
A **programming language** is a formal language designed to communicate instructions to a computer. There are two major types of programming languages: low-level languages and high-level languages.

Low-Level Languages

Low-level languages are referred to as 'low' because they are very close to how different hardware elements of a computer actually communicate with each other. **Low-level languages** are machine oriented and require extensive knowledge of computer hardware and its configuration. There are two categories of low-level languages: machine language and assembly language.

Machine language, or **machine code**, is the only language that is directly understood by the computer, and it does not need to be translated. All instructions use binary notation and are written as a string of 1s and 0s. A program instruction in machine language may look something like this:

```
10010101100101001111101010011011100101
```

Technically speaking, this is the only language computer hardware understands. However, binary notation is very difficult for humans to understand. This is where assembly languages come in.

An assembly language is the first step to improve programming structure and make machine language more readable by humans. An **assembly language** consists of a set of symbols and letters. A translator is required to translate the assembly language to machine language. This translator program is called the 'assembler.' It can be called the second generation language since it no longer uses 1s and 0s to write instructions, but terms like `MOVE`, `ADD`, `SUB` and `END`.

Many of the earliest computer programs were written in assembly languages. Most programmers today don't use assembly languages very often, but they are still used for applications like operating systems of electronic devices and technical applications, which use very precise timing or optimization of computer resources. While easier than machine code, assembly languages are still pretty difficult to understand. This is why high-level languages have been developed.

High-Level Languages

A **high-level language** is a programming language that uses English and mathematical symbols, like +, -, % and many others, in its instructions. When using the term 'programming languages,' most people are actually referring to high-level languages. High-level languages are the languages most often used by programmers to write programs. Examples of high-level languages are C++, Fortran, Java and Python.

To get a flavor of what a high-level language actually looks like, consider an ATM machine where someone wants to make a withdrawal of $100. This amount needs to be compared to the account balance to make sure there are enough funds. The instruction in a high-level computer language would look something like this:

```
x = 100
if balance  x:
 print 'Insufficient balance'
else:
 print 'Please take your money'
```

This is not exactly how real people communicate, but it is much easier to follow than a series of 1s and 0s in binary code.

There are a number of advantages to high-level languages. The first advantage is that high-level languages are much closer to the logic of a human language. A high-level language uses a set of rules that dictate how words and symbols can be put together to form a program. Learning a high-level language is not unlike learning another human language - you need to learn vocabulary and grammar so you can make sentences. To learn a programming language, you need to learn commands, syntax and logic, which correspond closely to vocabulary and grammar.

The second advantage is that the code of most high-level languages is portable and the same code can run on different hardware. Both machine code and assembly languages are hardware specific and not portable. This means that the machine code used to run a program on one specific computer needs to be modified to run on another computer. Portable code in a high-level language can run on multiple computer systems without modification. However, modifications to code in high-level languages may be necessary because of the operating system. For example, programs written for Windows typically don't run on a Mac.

A high-level language cannot be understood directly by a computer, and it needs to be translated into machine code. There are two ways to do this, and they are related to how the program is executed: a high-level language can be compiled or interpreted.

Compiler

A **compiler** is a computer program that translates a program written in a high-level language to the machine language of a computer. The high-level program is referred to as 'the source code.' A typical computer program processes some type of input data to produce output data. The compiler is used to translate source code into machine code or compiled code. This does not yet use any of the input data. When the compiled code is executed, referred to as 'running the program,' the program processes the input data to produce the desired output.

When using a compiler, the entire source code needs to be compiled before the program can be executed. The resulting machine code is typically a compiled file, such as a file with an .exe extension. Once you have a compiled file, you can run the program over and over again without having to compile it again. If you have multiple inputs that require processing, you run the compiled code as many times as needed.

Q2. Discuss about different data types of C programming Language.

Ans.

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types:

1. **Primary data types**:

   These are fundamental data types in C namely integer(`int`), floating point(`float`), character(`char`) and `void`.

2. **Derived data types**:

   Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **stucture**, **union** and **pointer**. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable `x` is declared as `int`. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.

Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767

unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

---

void type

`void` type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

Q3. Find the output of the following expressions

a) X=20/5*2+30-5 b) Y=30 – (40/10+6) +10 c) Z= 40*2/10-2+10

Ans.

Q4. Describe the syntax of the following statements

a) If – else statement b) for loop c) while loop d) do-while loop

a) If – else statement

The if/else statement executes a block of code if a specified condition is true. If the condition is false, another block of code can be executed.

The if/else statement is a part of JavaScript's "Conditional" Statements, which are used to perform different actions based on different conditions.

In JavaScript we have the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is true
- Use **else** to specify a block of code to be executed, if the same condition is false
- Use **else if** to specify a new condition to test, if the first condition is false
- Use **switch** to select one of many blocks of code to be executed

Browser Support

Statement				
if/else	Yes	Yes	Yes	Yes

Syntax

The **if** statement specifies a block of code to be executed if a condition is true:

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

The **else** statement specifies a block of code to be executed if the condition is false:

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

The **else if** statement specifies a new condition if the first condition is false:

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and
condition2 is true
} else {
  // block of code to be executed if the condition1 is false and
condition2 is false
}
```

Parameter Values

Parameter	Description
Condition	Required. An expression that evaluates to true or false

Technical Details

JavaScript Version:	ECMAScript 1

More Examples

Example

If the time is less than 20:00, create a "Good day" greeting, otherwise "Good evening":

```
var time = new Date().getHours();
if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

Try it Yourself »

Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
var time = new Date().getHours();
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

Try it Yourself »

Example

If the first element in the document has an id of "myDIV", change its font-size:

```
var x = document.getElementsByTagName("DIV")[0];

if (x.id === "myDIV") {
  x.style.fontSize = "30px";
}
```

Try it Yourself »

Example

Change the value of the source attribute (src) of an <img> element, if the user clicks on the image:

```
<img
id="myImage" onclick="changeImage()" src="pic_bulboff.gif" width="100"
height="180">

<script>
function changeImage() {
  var image = document.getElementById("myImage");
  if (image.src.match("bulbon")) {
    image.src = "pic_bulboff.gif";
  } else {
    image.src = "pic_bulbon.gif";
  }
}
</script>
```

Try it Yourself »

Example

Display a message based on user input:

```
var letter = document.getElementById("myInput").value;
var text;

// If the letter is "c"
if (letter === "c") {
  text = "Spot on! Good job!";

// If the letter is "b" or "d"
} else if (letter === "b" || letter === "d") {
  text = "Close, but not close enough.";

// If the letter is anything else
} else {
  text = "Waaay off..";
}
```

Try it Yourself »

Example

Validate input data:

```
var x, text;

// Get the value of the input field with id="numb"
x = document.getElementById("numb").value;

// If x is Not a Number or less than 1 or greater than 10, output
"input is not valid"
// If x is a number between 1 and 10, output "Input OK"

if (isNaN(x) || x < 1 || x > 10) {
  text = "Input not valid";
} else {
  text = "Input OK";
}
```

b) for loop

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

Instead of writing:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

You can write:

```
var i;
for (i = 0; i < cars.length; i++) {
  text += cars[i] + "<br>";
}
```

Try it Yourself »

Different Kinds of Loops

JavaScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `for/in` - loops through the properties of an object
- `for/of` - loops through the values of an iterable object
- `while` - loops through a block of code while a specified condition is true
- `do/while` - also loops through a block of code while a specified condition is true

The For Loop

The `for` loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example

```
for (i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

Try it Yourself »

From the example above, you can read:

Statement 1 sets a variable before the loop starts (var i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5).

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Statement 1

Normally you will use statement 1 to initialize the variable used in the loop (i = 0).

This is not always the case, JavaScript doesn't care. Statement 1 is optional.

You can initiate many values in statement 1 (separated by comma):

Example

```
for (i = 0, len = cars.length, text = ""; i < len; i++) {
  text += cars[i] + "<br>";
}
```

Try it Yourself »

And you can omit statement 1 (like when your values are set before the loop starts):

Example

```
var i = 2;
var len = cars.length;
var text = "";
for (; i < len; i++) {
  text += cars[i] + "<br>";
}
```

Try it Yourself »

Statement 2

Often statement 2 is used to evaluate the condition of the initial variable.

This is not always the case, JavaScript doesn't care. Statement 2 is also optional.

If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.

If you omit statement 2, you must provide a **break** inside the loop. Otherwise the loop will never end. This will crash your browser. Read about breaks in a later chapter of this tutorial.

Statement 3

Often statement 3 increments the value of the initial variable.

This is not always the case, JavaScript doesn't care, and statement 3 is optional.

Statement 3 can do anything like negative increment (i--), positive increment (i = i + 15), or anything else.

Statement 3 can also be omitted (like when you increment your values inside the loop):

Example

```
var i = 0;
var len = cars.length;
for (; i < len; ) {
  text += cars[i] + "<br>";
  i++;
}
```

Try it Yourself »

The For/In Loop

The JavaScript `for/in` statement loops through the properties of an object:

Example

```
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
```

```
var x;
for (x in person) {
  text += person[x];
}
```

Try it Yourself »

The For/Of Loop

The JavaScript `for/of` statement loops through the values of an iterable objects.

`for/of` lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

The `for/of` loop has the following syntax:

```
for (variable of iterable) {
  // code block to be executed
}
```

variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with `const`, `let`, or `var`.

iterable - An object that has iterable properties.

Looping over an Array

Example

```
var cars = ["BMW", "Volvo", "Mini"];
var x;

for (x of cars) {
  document.write(x + "<br >");
}
```

Try it Yourself »

Looping over a String

Example

```
var txt = "JavaScript";
var x;

for (x of txt) {
  document.write(x + "<br >");
}
```

Try it Yourself »

The While Loop

The `while` loop and the `do/while` loop will be explained in the next chapter.

c) while loop

The `while` loop loops through a block of code as long as a specified condition is true.

Syntax

```
while (condition) {
  // code block to be executed
}
```

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

Example

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

Try it Yourself »

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do/While Loop

The `do/while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

Example

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

Try it Yourself »

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

Comparing For and While

If you have read the previous chapter, about the for loop, you will discover that a while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

The loop in this example uses a `for` loop to collect the car names from the cars array:

Example

```
var cars = ["BMW", "Volvo", "Saab", "Ford"];
var i = 0;
var text = "";

for (;cars[i];) {
  text += cars[i] + "<br>";
  i++;
}
```

Try it Yourself »

The loop in this example uses a `while` loop to collect the car names from the cars array:

Example

```
var cars = ["BMW", "Volvo", "Saab", "Ford"];
var i = 0;
var text = "";

while (cars[i]) {
  text += cars[i] + "<br>";
  i++;
}
```

Try it Yourself »

Exercise:

Create a loop that runs as long as `i` is less than 10.

```
var i = 0;
          (i          10) {
  console.log(i);
  i++
}
```

d) do-while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

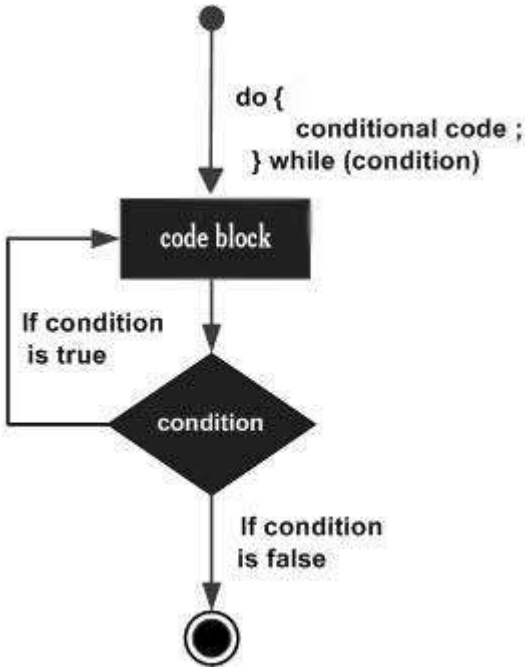
The syntax of a **do...while** loop in C programming language is −

```
do {
   statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
```

```
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Q5. Find the output of the following program segments

a) b) c)

```
#include <stdio.h>
int main()
{
int i;
for (i=1; i<2; i++)
{
printf( "IMS Ghaziabad\n");
}
}
```

b)

```
#include <stdio.h>
int main()
{
int i = 1;
while ( i <= 2 )
{
printf( "IMS Ghaziabad\n");
i = i + 1;
}
```

```
}
c)

#include <stdio.h>
void main()
{
int a = 10, b=100;
if( a > b )
printf( "Largest number is %d\n”, a);
else
printf( "Largest number is %d\n”, b);
```