**Date: 27-04-2021**

**Branch: CSE (III Year) and EE (III Year)**

**Topic: Introduction, Machine Language, Assembly Language, Assembler, difference between Machine Language and Assembly Language, Loops**

**Time: 08:00 AM -09:00 AM**

Computer can understand only the language of Digital Electronics. Digital Electronics deals with presence and absence of voltages. Within the computer there are two logics can play their role. These logics are −

* **Positive Logic** − Here presence of voltage will be denoted by 1 and absence of voltage will be denoted by 0
* **Negative Logic** −Here presence of voltage will be denoted by 0 and absence of voltage will be denoted by 1

But obviously computer can follow anyone of the logics at a time, not both the logics simultaneously. To make the computer understand, a program can be written using only 0s and 1s. The data can also be specified and represented using only 0s and 1s. Such a program is called Machine Language program. Machine language was the first in the evolution of computer programming languages. Computer directly understands a program written in the machine language. So as a result, machine language program does not require any translator to convert from one form to another. In fact, even to this day, basically computers understand only the 0s and 1s.

Disadvantages of machine language program − Writing a program in machine language has the following drawbacks.

* It is very tiresome to work with and highly error prone. While writing the program, a 1 and 0 can get interchanged due to typographical error. But then it is very difficult to locate it for correction. So a machine language program is very difficult to debug.
* Just having a look at the program, it is very difficult to visualize the function of the program. In fact, it is very difficult to make out whether a particular bit sequence is an instruction in the program, or a data value, or the output result. As instructions, data, output and operands, all are represented using 0s and 1s in machine language.
* Machine language programs are platform and architecture-dependent. The same program does not work on another computer by a different manufacturer. This is because machine language is different for different computers. Say the bite pattern 11110000 means addition in one architecture but might be representing subtraction in another architecture as well.
* To develop a program in machine language, the programmer must be highly conversant with the organization and architecture of the computer system being used.

Advantages of machine language program −The only advantages of writing in machine language are

* The Machine language program is executed faster than a program written in a high-level language (high-level language is discussed a little later). The efficiency of the program solely depends on the complexity of the program itself.
* A translator like compiler or interpreter is not needed and so results in a cheaper computer system.

As a conclusion, we can mention that nowadays, machine language is rarely used, except where very high-speed execution is required. Machine language written program is also used in cheap microcomputer systems.

## Machine language example

Below is an example of machine language (binary) for the text "Hello World."

01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100

Below is another example of machine language (non-binary), which prints the letter "A" 1000 times to the computer screen.

169 1 160 0 153 0 128 153 0 129 153 130 153 0 131 200 208 241 96

|  |  |  |
| --- | --- | --- |
| A | 065 | 01000001 |
| B | 066 | 01000010 |
| C | 067 | 01000011 |
| D | 068 | 01000100 |

## What is Assembly Language?

Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.

Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.

A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

## Advantages of Assembly Language

Having an understanding of assembly language makes one aware of −

* How programs interface with OS, processor, and BIOS;
* How data is represented in memory and other external devices;
* How the processor accesses and executes instruction;
* How instructions access and process data;
* How a program accesses external devices.

Other advantages of using assembly language are −

* It requires less memory and execution time;
* It allows hardware-specific complex jobs in an easier way;
* It is suitable for time-critical jobs;
* It is most suitable for writing interrupt service routines and other memory resident programs.

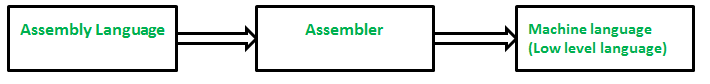
**Machine Language vs Assembly Language**

Machine language and assembly language are both low-level languages, but machine code is below assembly in the hierarchy of computer languages. Assembly language includes human-readable commands, such as mov, add, and sub, while machine language does not contain any words or even letters. Some [developers](https://techterms.com/definition/developer) manually write assembly language to optimize a program, but they do not write machine code. Only developers who write software compilers need to worry about machine language.

Difference between Machine Language and Assembly Language

**Machine language** is the low level programming language. Machine language can only be represented by 0s and 1s. In earlier when we have to create a picture or show data on the screen of the computer then it is very difficult to draw using only binary digits (0s and 1s). For example: To write 120 in the computer system its representation is 1111000. So it is very difficult to learn. To overcome this problem the assembly language is invented.

**Assembly language**is the more than low level and less than high-level language so it is intermediary language. Assembly languages use numbers, symbols, and abbreviations instead of 0s and 1s.For example: For addition, subtraction and multiplications it uses symbols likes Add, sub and Mul, etc.



Below is a table of differences between Machine Language and Assembly Language:

| **Machine Language** | **Assembly Language** |
| --- | --- |
| Machine language is only understand by the computers. | Assembly language is only understand by human beings not by the computers. |
| In machine language data only represented with the help of binary format(0s and 1s), hexadecimal and octadecimal. | In assembly language data can be represented with the help of mnemonics such as Mov, Add, Sub, End etc. |
| Machine language is very difficult to understand by the human beings. | Assembly language is easy to understand by the human being as compare to machine language. |
| Modifications and error fixing cannot be done in machine language. | Modifications and error fixing can be done in assembly language. |
| Machine language is very difficult to memorize so it is not possible to learn the machine language. | Easy to memorize the assembly language because some alphabets and mnemonics are used. |
| Execution is fast in machine language because all data is already present in binary format. | Execution is slow as compared to machine language. |
| There is no need of translator. The machine understandable form is the machine language. | Assembler is used as translator to convert mnemonics into machine understandable form. |
| Machine language is hardware dependent. | Assembly language is the machine dependent and it is not portable. |

Let's consider a situation when you want to print **Hello, World!** five times. Here is a simple C program to do the same −

#include <stdio.h>

int main() {

printf( "Hello, World!\n");

printf( "Hello, World!\n");

printf( "Hello, World!\n");

printf( "Hello, World!\n");

printf( "Hello, World!\n");

}

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

Hello, World!

Hello, World!

Hello, World!

Hello, World!

Hello, World!

It was simple, but again, let's consider another situation when you want to write **Hello, World!** a thousand times. We can certainly not write printf() statements a thousand times. Almost all the programming languages provide a concept called **loop**, which helps in executing one or more statements up to a desired number of times. All high-level programming languages provide various forms of loops, which can be used to execute one or more statements repeatedly.

Let's write the above C program with the help of a **while loop** and later, we will discuss how this loop works

#include <stdio.h>

int main() {

int i = 0;

while ( i < 5 ) {

printf( "Hello, World!\n");

i = i + 1;

}

}

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

Hello, World!

Hello, World!

Hello, World!

Hello, World!

Hello, World!

The above program makes use of a **while loop**, which is being used to execute a set of programming statements enclosed within {....}. Here, the computer first checks whether the given condition, i.e., variable "a" is less than 5 or not and if it finds the condition is true, then the loop body is entered to execute the given statements. Here, we have the following two statements in the loop body −

* First statement is *printf()* function, which prints Hello World!
* Second statement is *i = i + 1*, which is used to increase the value of variable **i**

After executing all the statements given in the loop body, the computer goes back to while( i < 5) and the given condition, (i < 5), is checked again, and the loop is executed again if the condition holds true. This process repeats till the given condition remains true which means variable "a" has a value less than 5.

To conclude, 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 −



This tutorial has been designed to present programming's basic concepts to non-programmers, so let's discuss the two most important loops available in C programming language. Once you are clear about these two loops, then you can pick-up C programming tutorial or a reference book and check other loops available in C and the way they work.

## The while Loop

A **while loop** available in C Programming language has the following syntax −

while ( condition ) {

/\*....while loop body ....\*/

}

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



The following important points are to be noted about a while loop −

* A while loop starts with a keyword **while** followed by a **condition** enclosed in ( ).
* Further to the while() statement, you will have the body of the loop enclosed in curly braces **{...}**.
* A while loop body can have one or more lines of source code to be executed repeatedly.
* If the body of a while loop has just one line, then its optional to use curly braces **{...}**.
* A while loop keeps executing its body till a given **condition** holds true. As soon as the condition becomes false, the while loop comes out and continues executing from the immediate next statement after the while loop body.
* A condition is usually a relational statement, which is evaluated to either true or false. A value equal to zero is treated as false and any non-zero value works like true.

## The do...while Loop

A while loop checks a given condition before it executes any statements given in the body part. C programming provides another form of loop, called **do...while** that allows to execute a loop body before checking a given condition. It has the following syntax −

do {

/\*....do...while loop body ....\*/

}

while ( condition );

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



If you will write the above example using **do...while** loop, then **Hello, World** will produce the same result −

#include <stdio.h>

int main() {

int i = 0;

do {

printf( "Hello, World!\n");

i = i + 1;

}

while ( i < 5 );

}

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

Hello, World!

Hello, World!

Hello, World!

Hello, World!

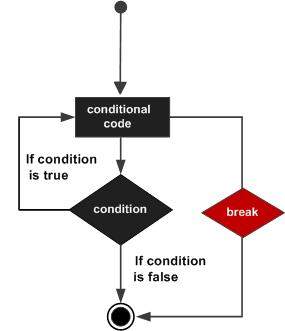
Hello, World!

## 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;

A **break** statement can be represented in the form of a flow diagram as shown below −



Following is a variant of the above program, but it will come out after printing Hello World! only three times −

#include <stdio.h>

int main() {

int i = 0;

do {

printf( "Hello, World!\n");

i = i + 1;

if( i == 3 ) {

break;

}

}

while ( i < 5 );

}

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

Hello, World!

Hello, World!

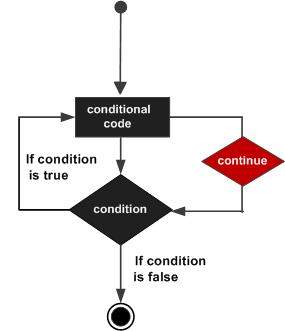
Hello, World!

## 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;

A **continue** statement can be represented in the form of a flow diagram as shown below −



Following is a variant of the above program, but it will skip printing when the variable has a value equal to 3 −

[Live Demo](http://tpcg.io/GmJzRF)

#include <stdio.h>

int main() {

int i = 0;

do {

if( i == 3 ) {

i = i + 1;

continue;

}

printf( "Hello, World!\n");

i = i + 1;

}

while ( i < 5 );

}

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

Hello, World!

Hello, World!

Hello, World!

Hello, World!

## Loops in Java

Following is the equivalent program written in Java that too supports **while** and **do...while** loops. The following program prints **Hello, World!** five times as we did in the case of C Programming −

You can try to execute the following program to see the output, which must be identical to the result generated by the above example.

public class DemoJava {

public static void main(String []args) {

int i = 0;

while ( i < 5 ) {

System.out.println("Hello, World!");

i = i + 1;

}

}

}

The **break** and **continue** statements in Java programming work quite the same way as they work in C programming.

## Loops in Python

Following is the equivalent program written in Python. Python too supports **while** and **do...while** loops. The following program prints **Hello, World!** five times as we did in case of C Programming. Here you must note that Python does not make use of curly braces for the loop body, instead it simply identifies the body of the loop using indentation of the statements.

You can try to execute the following program to see the output. To show the difference, we have used one more print statement, which will be executed when the loop will be over.

i = 0

while (i < 5):

print "Hello, World!"

i = i + 1

print "Loop ends"

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

Hello, World!

Hello, World!

Hello, World!

Hello, World!

Hello, World!

Loop ends

The **break** and **continue** statements in Python work quite the same way as they do in C programming.