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

Ans. 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 cQ2. Discuss about different data types of C programming Language.ompiler 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.each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:
Following are the examples of some very common data types used in C:

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

Data Type	Memory (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d

Data Type	Memory (bytes)	Range	Format Specifier
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

We can use the sizeof() operator to check the size of a variable. See the following C program for the usage of the various data types:

- C

filter_none
edit
play_arrow

brightness_4

```
#include <stdio.h>

int main()

{

    int a = 1;

    char b = 'G';

    double c = 3.14;

    printf("Hello World!\n");



    // printing the variables defined

    // above along with their sizes

    printf("Hello! I am a character. My value is %c and
"

           "my size is %lu byte.\n",

           b, sizeof(char));

    // can use sizeof(b) above as well



    printf("Hello! I am an integer. My value is %d and
"

           "my size is %lu  bytes.\n",

           a, sizeof(int));

    // can use sizeof(a) above as well
```

```
    printf("Hello! I am a double floating point
variable."

           " My value is %lf and my size is %lu
bytes.\n",

           c, sizeof(double));

    // can use sizeof(c) above as well



    printf("Bye! See you soon. :)\n");



    return 0;

}
```

Output:

```
Hello World!

Hello! I am a character. My value is G and my size is 1 byte.

Hello! I am an integer. My value is 1 and my size is 4  bytes.

Hello! I am a double floating point variable. My value is 3.140000
and my size i

s 8 bytes.

Bye! See you soon. :)
```

Quiz on Data Types in C

This article is contributed by Ayush Jaggi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Attention reader! Don't stop learning now. Get hold of all the important **C++ Foundation** and STL concepts with the **C++ Foundation and STL** courses at a student-friendly price and become industry ready.

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.a) x=20/5*2+30-5

X=4*2+30-5
x=8+30-5
x=38-5
x=33
ans. C) Z= 40*2/10-2+10
Z= 40*0.2-2+10
Z=8-2+10
Z=6+10
Z=16
Q4. Describe the syntax of the following statements

C if...else Statement

In this tutorial, you will learn about if statement (including if...else and nested if..else) in C programming with the help of examples.

C if Statement

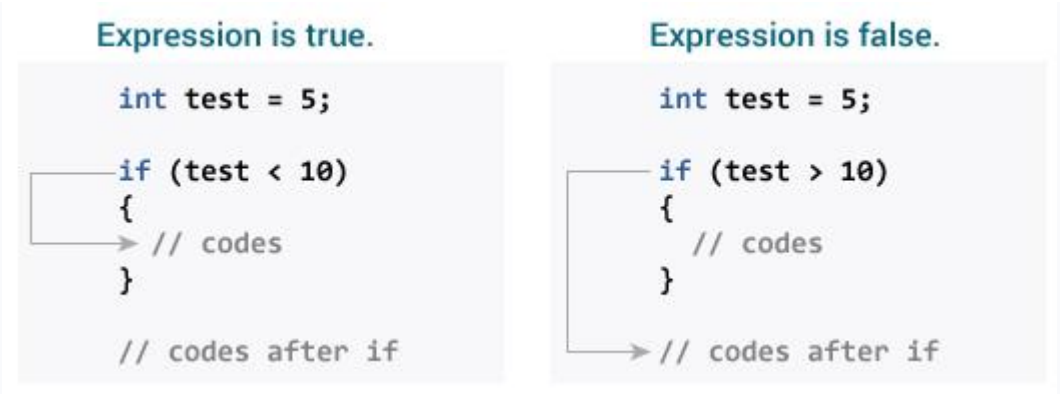
The syntax of the `if` statement in C programming is:

```
if (test expression)
{
   // statements to be executed if the test expression is true
}
```

How if statement works?

The `if` statement evaluates the test expression inside the parenthesis `()`.

- If the test expression is evaluated to true, statements inside the body of `if` are executed.
- If the test expression is evaluated to false, statements inside the body of `if` are not executed.



To learn more about when test expression is evaluated to true (non-zero value) and false (0), check relational and logical operators.

Example 1: if statement

```
// Program to display a number if it is negative

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

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");

    return 0;
}
```

Output 1

```
Enter an integer: -2
You entered -2.
```

```
The if statement is easy.
```

When the user enters -2, the test expression `number<0` is evaluated to true. Hence, `You entered -2` is displayed on the screen.

Output 2

```
Enter an integer: 5
The if statement is easy.
```

When the user enters 5, the test expression `number<0` is evaluated to false and the statement inside the body of `if` is not executed

---

C if...else Statement

The `if` statement may have an optional `else` block. The syntax of the `if..else` statement is:

```
if (test expression) {
    // statements to be executed if the test expression is true
}
else {
    // statements to be executed if the test expression is false
}
```

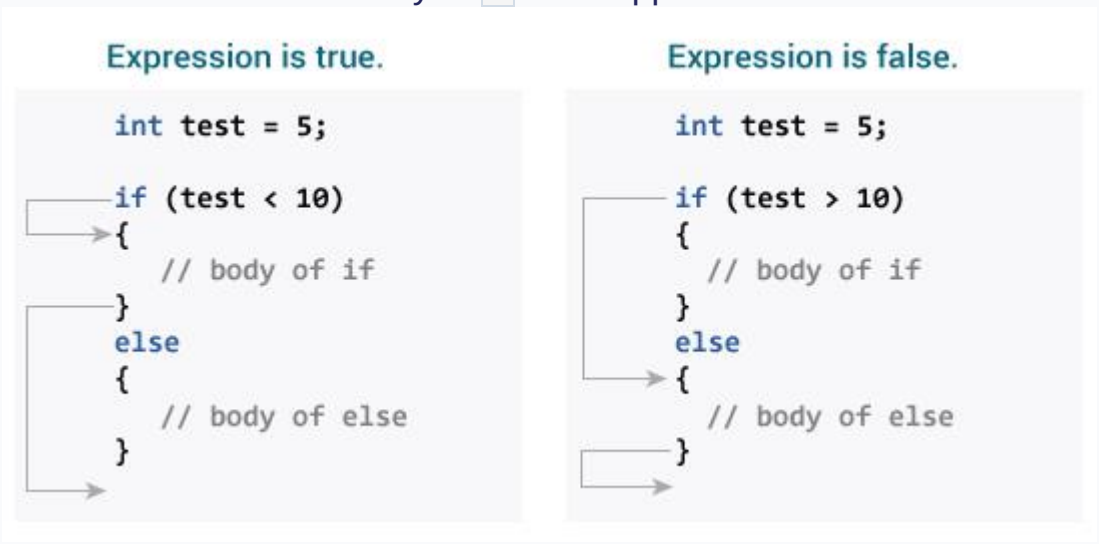
How if...else statement works?

If the test expression is evaluated to true,

- statements inside the body of `if` are executed.
- statements inside the body of `else` are skipped from execution.

If the test expression is evaluated to false,

- statements inside the body of `else` are executed
- statements inside the body of `if` are skipped from execution.



Example 2: if...else statement

```
// Check whether an integer is odd or even

#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if  (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
        printf("%d is an odd integer.",number);
    }

    return 0;
}
```

Output

```
Enter an integer: 7
7 is an odd integer.
```

When the user enters 7, the test expression `number%2==0` is evaluated to false. Hence, the statement inside the body of `else` is executed.

C if...else Ladder

The `if...else` statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.
The if...else ladder allows you to check between multiple test expressions and execute different statements.

Syntax of if...else Ladder

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

Example 3: C if...else Ladder

```c
// Program to relate two integers using =, > or < symbol

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Output

```
Enter two integers: 12
23
Result: 12 < 23
```

Nested if...else

It is possible to include an `if...else` statement inside the body of another `if...else` statement.

Example 4: Nested if...else

This program given below relates two integers using either `<`, `>` and `=` similar to the `if...else` ladder's example. However, we will use a nested `if...else` statement to solve this problem.

```
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2) {
      if (number1 == number2) {
        printf("Result: %d = %d",number1,number2);
      }
      else {
        printf("Result: %d > %d", number1, number2);
      }
    }
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

> If the body of an `if...else` statement has only one statement, you do not need to use brackets `{}`.

For example, this code

```
if (a > b) {
    print("Hello");
}
print("Hi");
```

is equivalent to

```
if (a > b)
    print("Hello");
print("Hi");
```

C for Loop

In this tutorial, you will learn to create for loop in C programming with the help of examples.

In programming, a loop is used to repeat a block of code until the specified condition is met.

C programming has three types of loops:

1. for loop
2. while loop
3. do...while loop

We will learn about `for` loop in this tutorial. In the next tutorial, we will learn about `while` and `do...while` loop.

---

for Loop

The syntax of the `for` loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

How for loop works?

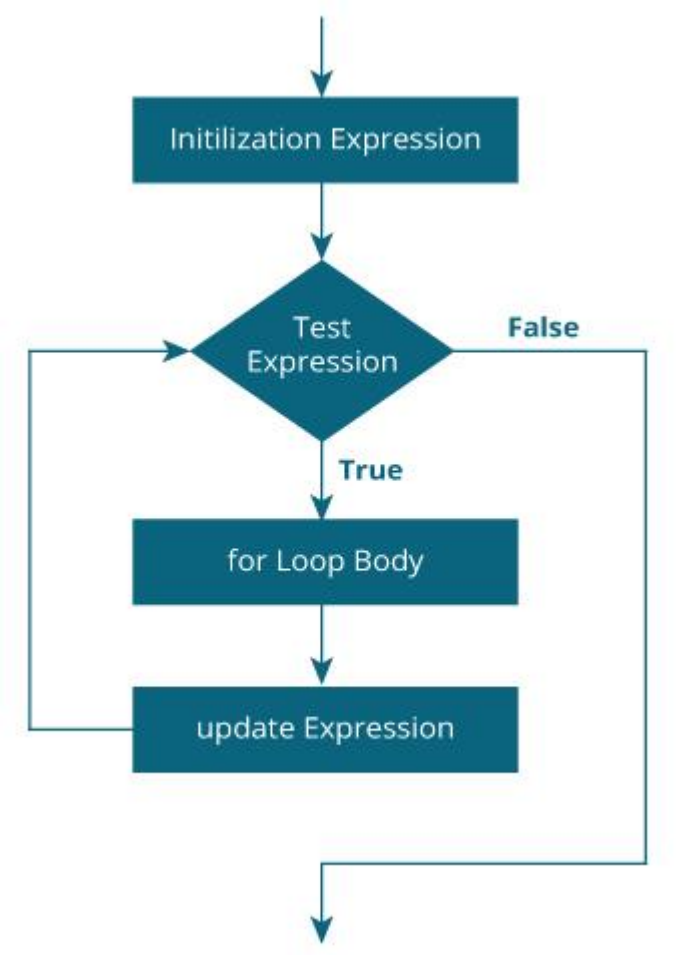
- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the `for` loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of `for` loop are executed, and the update expression is updated.
- Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

---

for loop Flowchart



Example 1: for loop

```c
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
```

```
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

1. `i` is initialized to 1.
2. The test expression `i < 11` is evaluated. Since 1 less than 11 is true, the body of `for` loop is executed. This will print the **1** (value of `i`) on the screen.
3. The update statement `++i` is executed. Now, the value of `i` will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print **2** (value of `i`) on the screen.
4. Again, the update statement `++i` is executed and the test expression `i < 11` is evaluated. This process goes on until `i` becomes 11.
5. When `i` becomes 11, `i < 11` will be false, and the `for` loop terminates.

C while and do...while Loop

In this tutorial, you will learn to create while and do...while loop in C programming with the help of examples.

In programming, loops are used to repeat a block of code until a specified condition is met.

C programming has three types of loops.

1. for loop
2. while loop
3. do...while loop

In the previous tutorial, we learned about `for` loop. In this tutorial, we will learn about `while` and `do..while` loop.

---

while loop

The syntax of the `while` loop is:

```
while (testExpression)
{
    // statements inside the body of the loop
}
```

How while loop works?

- The `while` loop evaluates the test expression inside the parenthesis `()`.
- If the test expression is true, statements inside the body of `while` loop are executed. Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.
- If the test expression is false, the loop terminates (ends).

To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.

---

nlike **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

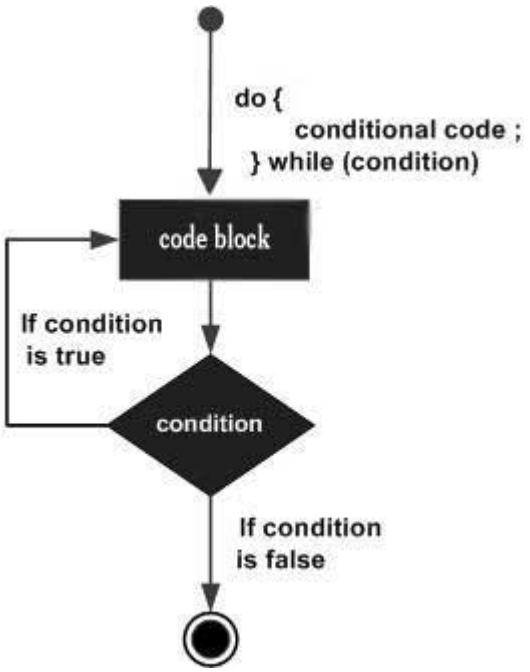
The syntax of a **do...while** loop in C programming language is −

```
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
```