Scientific Programming in High Level Language

Scientific programming is the application of programming techniques to solve scientific problems, such as numerical analysis, data processing, simulation, and visualization. High-level languages are programming languages that have a high level of abstraction from the details of the computer hardware and memory, and focus more on the programming logic and usability.

Some of the advantages of using high-level languages for scientific programming are:

- They are easier to learn, read, write, and debug than low-level languages.
- They have built-in features and libraries that support common scientific tasks, such as matrix operations, statistical functions, plotting, and parallel computing.
- They can interact with other languages and tools, such as calling C or Fortran functions, or using Python or R packages.

Some of the disadvantages of using high-level languages for scientific programming are:

- They may have lower performance and efficiency than low-level languages, especially for computationally intensive tasks.
- They may have less control and flexibility over the low-level aspects of the program, such as memory management, data types, and optimization.
- They may have compatibility and portability issues across different platforms and environments.

Some of the examples of high-level languages that are widely used for scientific programming are:

- Fortran: One of the oldest and most popular languages for scientific computing, especially for numerical calculations and simulations. It has a fast execution speed and a rich set of libraries and standards3.
- Python: A general-purpose and versatile language that has a simple and expressive syntax, and a large and diverse collection of libraries and packages for scientific computing, such as NumPy, SciPy, Matplotlib, Pandas, and TensorFlow.
- Julia: A relatively new and modern language that aims to combine the high-level productivity and ease of use of Python with the high-performance and efficiency of Fortran. It has a powerful and flexible syntax, and a built-in support for multiple dispatch, metaprogramming, and parallel computing.
- R: A language and environment for statistical computing and graphics, that has a comprehensive and coherent set of functions and packages for data analysis, modeling, visualization, and machine learning.
- MATLAB: A proprietary language and platform for numerical computing and engineering, that has a concise and intuitive syntax, and a rich set of built-in functions and toolboxes for various domains, such as linear algebra, optimization, signal processing, and image processing.

We are free to choose any language we want as long as it serves our purpose.

High Level Language and Compiler

A high-level language and a compiler are related concepts in the field of programming. A high-level language is a programming language that is more human-readable and abstract than the low-level machine code that computers can execute.

A compiler is a program that translates the source code written in a high-level language into an equivalent program in machine code or assembly language.

The main advantage of using a high-level language and a compiler is that they make the programming process easier, faster, and more portable across different platforms. However, some drawbacks are that they may introduce some overhead in terms of performance, memory usage, and error detection.

The main steps of compilation are:

- Lexical analysis: The compiler scans the source code and converts it into tokens, which are the basic units of meaning in the language.
- Syntax analysis: The compiler checks the structure and grammar of the source code and builds a parse tree that represents the logical hierarchy of the program.
- Semantic analysis: The compiler verifies the meaning and validity of the source code and performs type checking, scope resolution, and error detection.
- Optimization: The compiler improves the performance and efficiency of the code by eliminating redundant or unnecessary instructions, rearranging the order of execution, and applying various techniques such as loop unrolling, constant folding, and dead code elimination.
- Code generation: The compiler produces the final output code in the target language, which can be machine code, assembly code, or intermediate code.

Compilers can be classified into different types based on various criteria, such as:

- **The source and target languages:** A cross compiler translates the source code for a different platform than the one it runs on, a bootstrap compiler translates the source code of its own language, a source-to-source compiler translates the source code from one high-level language to another, and a decompiler reverses the process of compilation and recovers the source code from the executable code.
- **The number of passes:** A single-pass compiler processes the source code in one pass and generates the output code directly, a multi-pass compiler processes the source code in multiple passes and generates intermediate code in each pass, and an incremental compiler processes the source code in small units and updates the output code accordingly.
- **The time of compilation:** A static compiler performs the compilation before the execution of the program, a dynamic compiler performs the compilation during the execution of the program, and a just-in-time compiler performs the compilation on demand at runtime.

Compilers are important for programming because they enable the use of high-level languages that are more human-readable, portable, and expressive than low-level languages. They also enhance the speed, security, and reliability of the programs by detecting and correcting errors, optimizing the code, and generating efficient output code.

Compiler vs Interpreter

A compiler and an interpreter are two types of programs that convert high-level language code into machine code that can be executed by computers. The main difference between them is how they perform this conversion. Here are some key points to compare them:

- A compiler scans the entire source code and translates it as a whole into machine code, while an interpreter translates one statement of the source code at a time into machine code.
- A compiler generates an executable file that can be run independently, while an interpreter requires the source code every time the program is executed.
- A compiler usually takes more time to analyze the source code, but the execution time is faster, while an interpreter usually takes less time to analyze the source code, but the execution time is slower.
- A compiler can detect syntax and semantic errors before the program runs, while an interpreter can only detect errors during the program execution.
- A compiler is more suitable for performance-oriented and platform-specific programs, while an interpreter is more suitable for interactive and cross-platform programs.

Some examples of compiled languages are C, C++, and Java, and some examples of interpreted languages are Python, Ruby, and JavaScript.

Character Set in C

A character set is a set of alphabets, letters, and some special characters that are valid in C language. The C language supports a total of 256 characters, which are divided into the following categories1:

- **Alphabets**: These include uppercase and lowercase letters from A to Z. C accepts both lowercase and uppercase alphabets as variables and functions2.
- **Digits**: These include numbers from 0 to 9. Digits are used to construct numeric values or expressions in C programs.
- **Special Characters**: These include symbols such as +, -, *, /, =, ;, :, (,), {, }, [,], ., ,, #, &, !, ?, ', ", \, ^, %, |, ~, <, >, _, etc. Special characters are used for various purposes, such as arithmetic operations, assignment, punctuation, comments, escape sequences, preprocessor directives, pointers, etc.
- **White Spaces:** These include blank spaces, tabs, newlines, etc. White spaces are used to separate words and symbols in a C program. They are ignored by the compiler, except when they are part of a string constant or a character constant.

Constants in C

Constants in C are the read-only variables whose values cannot be modified once they are declared in the C program. Constants are also called literals. Constants can be any of the data types. It is considered best practice to define constants using only upper-case names.

Constants are categorized into two basic types, each of which has subtypes/categories. These are:

- Primary Constants: These include integer constants, real or floating-point constants, and character constants.

- Secondary Constants: These include array constants, string constants, structure constants, pointer constants, and enumeration constants.

The difference between constants and literals is that constants are variables that are initialized with a literal value and cannot be changed later, while literals are the fixed values themselves that can be assigned to any variable. For example, in the statement int x = 10;, x is a variable, 10 is a literal, and if we add const before int, then x becomes a constant.

Keywords in C

Keywords in C are predefined, reserved words that have special meanings to the compiler. They are part of the syntax and cannot be used as identifiers (names of variables, functions, etc.) in the program. There are 32 keywords in C, such as int, char, if, else, for, switch, break, continue, return, etc. Each keyword has a specific purpose and function in the C language. For example, the int keyword is used to declare integer variables, the if keyword is used to make conditional statements, the for keyword is used to create loops, and so on. Here is the full list of keywords in C:

Keyword	Description
auto	Declares automatic variables
break	Terminates the current loop or switch statement
case	Labels a branch in a switch statement
char	Declares character type variables
const	Defines constant variables
continue	Skips the current iteration of a loop
default	Labels the default branch in a switch statement
do	Starts a do-while loop
double	Declares double-precision floating type variables
else	Executes a block of code if the condition is falsee
num	Declares an enumeration type
extern	Declares a variable or a function with external linkage
float	Declares single-precision floating type variables
for	Starts a for loop
goto	Jumps to a labeled statement
if	Executes a block of code if the condition is true
int	Declares integer type variables
long	Declares long integer or double-precision floating type variables
register	Declares register variablesreturnReturns a value from a function
short	Declares short integer type variables
signed	Declares signed integer or character type variables
sizeof	Returns the size of a data type or a variable
static	Declares static variables or functions
struct	Declares a structure type
switch	Starts a switch statement
typedef	Defines a new data type name
union	Declares a union type
unsigned	Declares unsigned integer or character type variables
void	Specifies no return type or no arguments
volatile	Declares volatile variables
while	Starts a while loop

Variables in C

A variable in C is a memory location with some name that helps store some form of data and retrieve it when required. We can store different types of data in the variable and reuse the same variable for storing some other data any number of times. They can be viewed as the names given to the memory location so that we can refer to it without having to memorize the memory address.

To use variables in C, we need to follow some rules and syntax. Here are some important points to remember:

- Variables in C must be declared before they can be used. The declaration specifies the type and name of the variable.
- Variables in C can be initialized at the time of declaration or later. The initialization assigns a value to the variable.
- Variables in C can be changed by assigning a new value to them. For example, x = 20; assigns the value 20 to the variable x.

Declaring variables in C is the process of specifying the name and the type of the variable before using it in the program. The syntax for declaring a variable in C is:

```
data_type variable_name = value; // single variable declaration and initialization data_type variable_name1, variable_name2; // multiple variable declaration
```

Here, data_type is one of the C types (such as int, float, char, etc.), variable_name is the name of the variable given by the user, and value is the value assigned to the variable by the user.

For example, to declare a variable of type int with the name x and assign it the value 10, we can write:

```
int x = 10;
```

We can also declare a variable without assigning a value, and assign the value later. For example:

```
int x; // variable declaration
x = 10; // variable initialization
```

To output the value of a variable in C, we need to use the printf() function with format specifiers. Format specifiers are placeholders for the variable values and are preceded by a percentage sign %. For example, %d for int, %f for float, and %c for char. For example, to print the value of the variable x of type int, we can write:

```
printf("%d", x); // Outputs 10
```

Instructions and Programs

The difference between instruction and program in C is that instruction is a single operation that the processor can execute, while program is a collection of instructions that perform a specific task.

- Instruction: An instruction is a set of binary code that tells the processor what to do, such as move data, perform arithmetic, or control the flow of execution. Instructions are the basic building blocks of a program. Each instruction has an opcode (operation code) that specifies the type of operation, and one or more operands that specify the data or addresses involved in the operation. For example, ADD R1, R2 is an instruction that adds the contents of register R1 and R2 and stores the result in R1. Instructions are executed by the processor one by one in a sequential order, unless there is a jump or branch instruction that changes the order. Instructions are also known as machine code, because they are directly understood by the processor.

- Program: A program is a sequence of instructions that perform a specific task or solve a problem. Programs are written by programmers using programming languages, such as C, that are easier to read and write than binary code. Programs are stored in the main memory or secondary storage devices, such as hard disks. Programs need to be translated into machine code before they can be executed by the processor. This translation can be done by compilers, interpreters, or assemblers. Programs are also known as software applications, because they provide some functionality to the user or the system. For example, a web browser, a word processor, or a game are programs that run on a computer.

Operators in C

Operators in C are symbols that represent operations to be performed on one or more operands. Operands are the values or variables on which the operators act. For example, in the expression a + b, a and b are the operands and + is the operator.

There are different types of operators in C, depending on their functionality and the number of operands they require. Some of the common types of operators are:

- Arithmetic operators: These operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, and modulus. They can be either unary (one operand) or binary (two operands). For example, +, -, *, /, %, ++, and --.

- Relational operators: These operators are used to compare the values of two operands and return a Boolean value (true or false) based on the result of the comparison. They can be only binary (two operands). For example, ==, !=, <, >, <=, and >=.

- Logical operators: These operators are used to combine the results of two or more relational expressions and return a Boolean value (true or false) based on the logical rules. They can be only binary (two operands). For example, && (logical AND), || (logical OR), and ! (logical NOT).

- **Bitwise operators:** These operators are used to manipulate the individual bits of an operand and perform bitwise operations such as AND, OR, XOR, NOT, shift, and rotate. They can be either unary (one operand) or binary (two operands). For example, &, |, ^, ~, <<, and >>.

- **Assignment operators:** These operators are used to assign a value to a variable or modify the value of a variable by performing some operation. They can be either unary (one operand) or binary (two operands). For example, =, +=, -=, *=, /=, and %=.

- **Conditional operator:** This operator is used to evaluate a condition and return one of the two values based on whether the condition is true or false. It is also known as the ternary operator because it requires three operands. The syntax is condition ? value_if_true : value_if_false. For example, x = (a > b) ? a : b; assigns the larger of a and b to x.

- **Comma operator:** This operator is used to separate two or more expressions and evaluate them from left to right. The value of the last expression is returned as the result. For example, x = (y = 10, y + 5); assigns 10 to y and 15 to x.

- **Sizeof operator:** This operator is used to return the size of an operand in bytes. It can be either unary (one operand) or binary (two operands). For example, sizeof(int) returns the size of an int type, which is usually 4 bytes.

Expressions in C

An expression in C is a combination of constants, variables, operators, and functions that evaluates to a single value. Expressions can be used to perform calculations, assign values, compare values, or control the flow of the program. For example, x + y * 2 is an expression that adds the value of x to the product of y and 2.

There are different types of expressions in C, depending on the type of operands and operators involved. Some of the common types of expressions are:

- **Arithmetic expressions:** These expressions use arithmetic operators, such as +, -, *, /, and %, to perform mathematical operations on numeric operands. For example, a + b / c is an arithmetic expression that divides b by c and adds the result to a.

- **Relational expressions:** These expressions use relational operators, such as ==, !=, <, >, <=, and >=, to compare the values of two operands and return a Boolean value (0 or 1) based on the result of the comparison. For example, x == y is a relational expression that returns 1 if x and y are equal, and 0 otherwise.

- **Logical expressions:** These expressions use logical operators, such as &&, ||, and !, to combine the results of two or more relational expressions and return a Boolean value (0 or 1) based on the logical rules. For example, x > 0 && y < 0 is a logical expression that returns 1 if x is positive and y is negative, and 0 otherwise.

- **Bitwise expressions:** These expressions use bitwise operators, such as &, |, ^, ~, <<, and >>, to manipulate the individual bits of an operand and perform bitwise operations, such as AND, OR, XOR, NOT, shift, and rotate. For example, x & y is a bitwise expression that performs a bitwise AND operation on x and y.

- **Assignment expressions:** These expressions use assignment operators, such as =, +=, -=, *=, /=, and %=, to assign a value to a variable or modify the value of a variable by performing some operation. For example, x += 5 is an assignment expression that adds 5 to x and assigns the result back to x.

- **Conditional expressions:** These expressions use the conditional operator, also known as the ternary operator, which has the syntax condition ? value_if_true : value_if_false, to evaluate a condition and return one of the two values based on whether the condition is true or false. For example, x > y ? x : y is a conditional expression that returns the larger of x and y.

- **Comma expressions:** These expressions use the comma operator, which has the syntax expression1, expression2, to separate two or more expressions and evaluate them from left to right. The value of the last expression is returned as the result. For example, x = 10, y = 20 is a comma expression that assigns 10 to x and 20 to y, and returns 20 as the result.

Input output statements in C

Input and output statements in C are used to communicate with the user or other devices through the standard input and output streams. The standard input stream is usually the keyboard, and the standard output stream is usually the screen.

There are several functions in C that can perform input and output operations, such as printf(), scanf(), getchar(), putchar(), gets(), and puts(). These functions are defined in the header file stdio.h, which stands for standard input output header.

The **printf() function** is used to display formatted output on the screen. It takes a format string as the first argument, which can contain text, escape sequences, and format specifiers. Format specifiers are placeholders for the values of the variables that are passed as the subsequent arguments. For example, %d is the format specifier for integers, %f is for floats, and %c is for characters. The printf() function replaces the format specifiers with the corresponding values and prints the result on the screen. For example:

```
#include <stdio.h>
int main()
{
 int x = 10;
float y = 3.14;
char z = 'A';
printf("x = %d, y = %f, z = %c\n", x, y, z);
return 0;
}
```

This program will output:

```
x = 10, y = 3.140000, z = A
```

The **scanf() function** is used to take formatted input from the keyboard. It takes a format string as the first argument, which specifies the type and number of values to be read. The subsequent arguments are the addresses of the variables where the input values will be stored. The addresses are obtained by using the & operator, which is also known as the address-of operator. The scanf() function reads the input from the keyboard, converts it to the specified type, and stores it in the corresponding variables. For example:

```
#include <stdio.h>
int main()
{
int x;
float y;
char z;
printf("Enter an integer, a float, and a character: ");
scanf("%d %f %c", &x, &y, &z);
printf("You entered: x = %d, y = %f, z = %c\n", x, y, z);
return 0;
}
```

This program will prompt the user to enter an integer, a float, and a character, separated by spaces. For example, if the user enters:

```
5
2.5
B
```

The program will output:

```
You entered: x = 5, y = 2.500000, z = B
```

The **getchar() function** is used to read a single character from the standard input stream. It returns the ASCII value of the character, or EOF if the end of the file is reached. The **putchar() function** is used to write a single character to the standard output stream. It takes the ASCII value of the character as the argument, and returns the same value, or EOF if an error occurs. For example:

```
#include <stdio.h>
int main() {
char c;
printf("Enter a character: ");
c = getchar();
printf("You entered: ");
putchar(c);
printf("\n");
return 0;
}
```

This program will prompt the user to enter a character, and then display the same character. For example, if the user enters:

X

The program will output:

You entered: X

The **gets() function** is used to read a string from the standard input stream. It takes a character array as the argument, and reads the input until a newline character (\n) or the end of the file is encountered. It stores the input in the character array, and appends a null character (\0) at the end. The **puts() function** is used to write a string to the standard output stream. It takes a character array as the argument, and writes the string until a null character (\0) is encountered. It also appends a newline character (\n) at the end. For example:

```
#include <stdio.h>
int main() {
char str[50];
printf("Enter a word: ");
gets(str);
printf("You entered: ");
puts(str);
return 0;
}
```

This program will prompt the user to enter a word, and then display the same word. For example, if the user enters:

Hello

The program will output:

You entered: Hello

Executable and Non-executable statements in C

Executable and non-executable statements in C are the two categories of statements that can appear in a C program. Executable statements are those that specify the actions to be performed during the execution of the program, such as calculations, assignments, input/output, or control flow. Non-executable statements are those that do not specify any actions to be performed during the execution, but rather provide information to the compiler, such as declarations, definitions, preprocessor directives, or comments.

For example, consider the following C program:

```
#include <stdio.h> // non-executable statement
#define PI 3.14 // non-executable statement
int main() // non-executable statement
{
   int r = 5; // non-executable statement
   float area; // non-executable statement
   area = PI * r * r; // executable statement
   printf("The area of the circle is %f\n", area); // executable statement
   return 0; // executable statement
}
```

In this program, the statements that start with # are preprocessor directives, which are processed by the preprocessor before the compilation. They are non-executable statements, as they do not affect the runtime behavior of the program.

The statements that declare or define variables, such as int r = 5; or float area;, are also non-executable statements, as they only tell the compiler the type and name of the variables, and optionally assign some initial values.

The statements that perform some operations on the variables, such as area = PI * r * r; or printf("The area of the circle is %f\n", area);, are executable statements, as they are translated into machine code and executed by the processor. The statement return 0; is also an executable statement, as it terminates the main function and returns a value to the operating system.

Control Statements

Control statements are used to control the flow of execution of a program. They allow us to make decisions, perform tasks repeatedly, or jump from one section of code to another. There are four types of control statements in C:

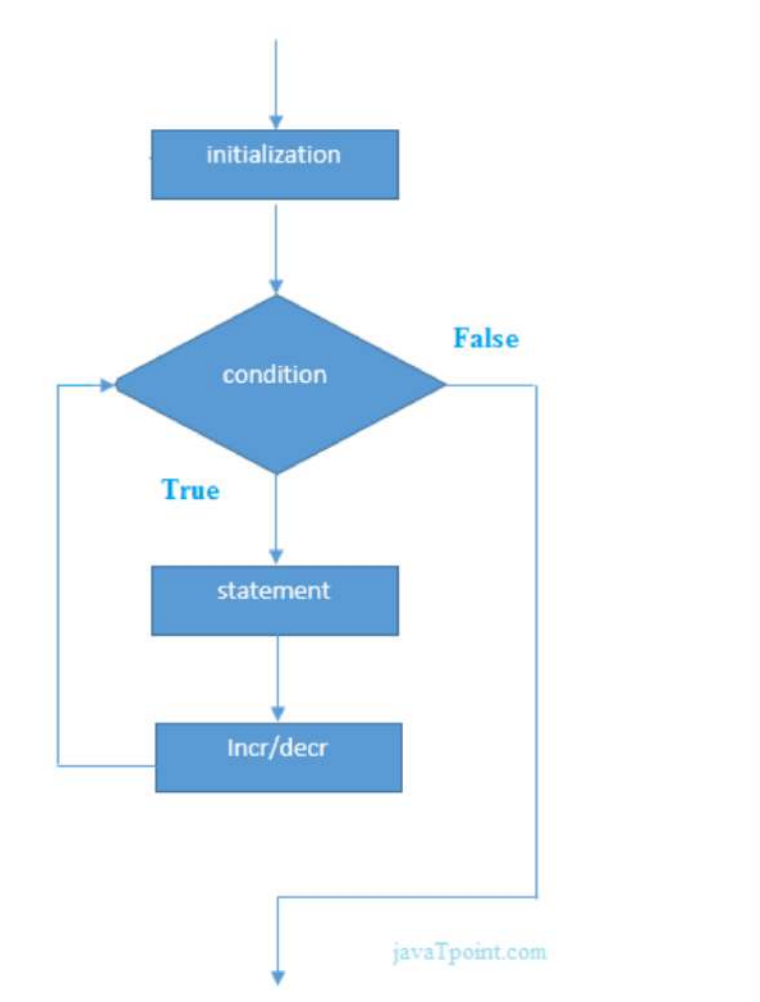
- **Decision making statements**: These statements are used to execute a block of code based on a condition. The condition can be either true or false. The decision making statements in C are if, if-else, and nested if-else.
- **Selection statements**: These statements are used to select one of the multiple choices based on a value or an expression. The selection statements in C are switch and case.
- **Iteration statements**: These statements are used to execute a block of code repeatedly until a condition is satisfied. The iteration statements in C are for, while, and do-while.
- **Jump statements**: These statements are used to transfer the control of the program to another section of code. The jump statements in C are break, continue, and goto.

Loops in C

For Loop

Syntax

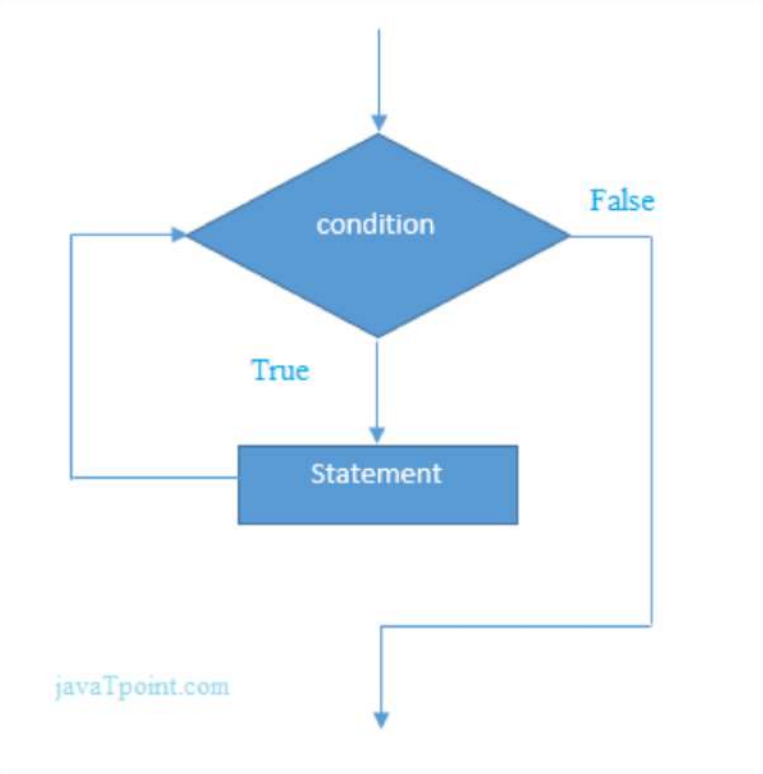
```
for(Expression 1; Expression 2; Expression 3)
   {
    //code to be executed
   }
```



While Loop

Syntax

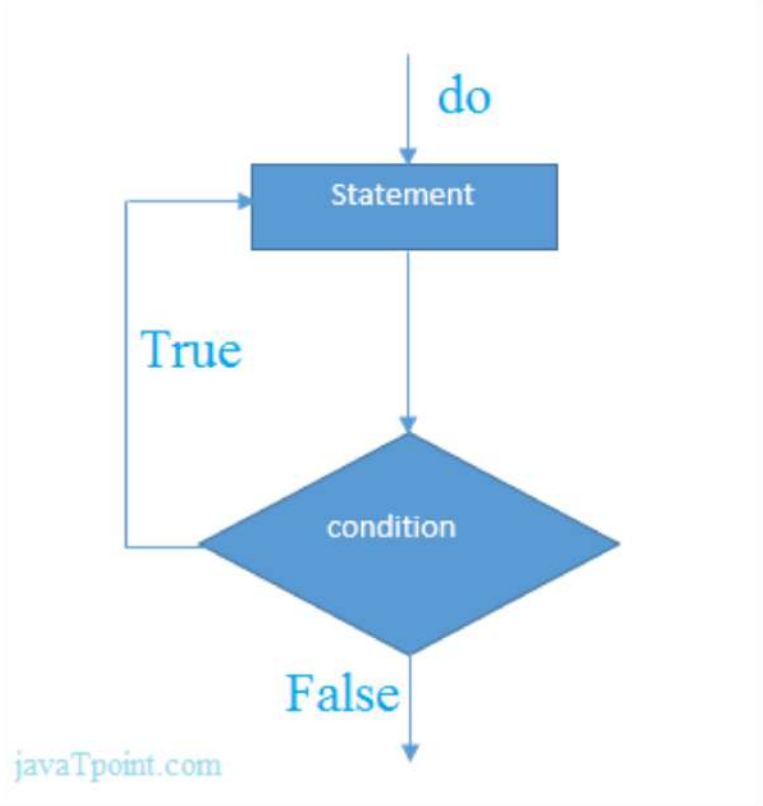
```
while(condition)
    {
    //code to be executed
    }
```



Do While Loop

Syntax

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



Questions:

1. What are high level programming languages?
2. What are the advantages of high level programming language?
3. What are the disadvantages of high level programming languages?
4. Name some high level programming languages used in scientific computing.
5. What are compilers?
6. What are interpreters?
7. Write the differences between compilers and interpreters.
8. Name some compiled languages. Name some interpreted languages.
9. What are C constants?
10. What are C characters?
11. What are C keywords?
12. What are C variables?
13. What is variable declaration?
14. What is variable initialization?
15. Write a note on Operators in C mentioning each type and examples.
16. What are the four types of control statements in C?
17. Write the syntax and flowchart of three types of loops in C?