LANGUAGE SPECIFICATION:-

Print statement:- (print -> cout)

The "print" statement is used to display output to the console. It is equivalent to the "cout" statement in C++. It can display literal strings and the values of variables.

print(“Hello World!”);

print(“The value of a is :”, a);

Code:

#include <iostream>

using namespace std;

int main() {

print("Hello World!"); // Output: Hello World!

print("The value of a is :", 10); // Output: The value of a is : 10

return 0;

}

Variable Input:- (take -> cin)

The "take" statement is used to receive input from the user. It is analogous to the "cin" statement in C++. It prompts the user with a message and stores the user's input into a variable.

take(“Input the value of a”, a);

Data types:-

Data types define the type of data that can be stored in a variable. The language specification includes "string" for sequences of characters, "float" for floating-point numbers, "int" for integers, and "const" for constant (unchangeable) values.

string for both characters, string

float for float

int for integer

const for constant value (unchangeable during program)

e.g:

const int a = 4;

Declaration:

a variable declaration is a statement that defines a variable and allocates memory to store data of a specific type. It tells the compiler or interpreter that a certain name will be used to represent a value of a specific data type, and it reserves the necessary memory space for that variable.

data type variable1;

e.g:-

int a;

data type variable2 = 23;

e.g:-

float b = 5.5;

Comments:-

Comments are non-executable lines used to provide explanations, add notes, or disable code temporarily. In this language, "#" is used for single-line comments, and "###" is used for multi-line comments.

# for single line

### for multi lines ###

Conditional Statements:

Conditional statements allow the program to execute different blocks of code based on specified conditions. The language includes "if," "elif" (else if), and "else" for this purpose.

if(…) for a condition

elif(…) for multi conditions

else() then condition

# Simple program to check if a number is positive, negative, or zero

# Get input from the user

num = float(input("Enter a number: "))

# Check the condition using if statements

if num > 0:

print("The number is positive.")

elif num < 0:

print("The number is negative.")

else:

print("The number is zero.")

Loops:

Loops are used to repeat a block of code multiple times. The language specification includes the "for" loop, which allows code execution for a specified number of iterations.

for() for-loop only

e.g:-

for(i=0; i<3; i++){

print(“Hello World!”)

};

Break and Continue Statement:

"terminate" is used to exit a loop prematurely, while "resume" allows the loop to continue to the next iteration.

terminate for stopping condition

resume for continuing statement

for (initialization; condition; update) {

// Code block

if (some\_condition) {

terminate; // Exit the loop immediately

} else {

// Code to be executed if the condition is not met

resume; // Continue to the next iteration of the loop

}

}

Function:

A function is a self-contained block of code that performs a specific task. Functions can have input arguments and can return values. They are defined using the "data\_type function\_name(...)" syntax and can be called using the function\_name(arguments) syntax.

data type function\_name(…)

{ …

return

}

function\_name(arguments); # function calling

e.g:-

float sum(a, b){

sum = a+b;

print(sum)}

sum(3, 4); # function calling

Punctuators:

Punctuators are special symbols used to structure the code. Examples in this specification include "{}" for code blocks, "[]" for arrays, and "()" for function arguments.

{ }, [ ], ( ) -> brackets

! (NOT), | (OR), & (AND), ^ (power), % (reminder)

Keywords:

Keywords are reserved words that have specific meanings in the programming language. Some of the keywords in this specification include "int," "string," "float," "void," "if," "elif," "else," "for," "continue," "break," "public," "private," "const," "class," "\n," "\t," "virtual," "get," and "set.”

int, string, float, void, if, elif, else, for, continue, break, public, private, const, class, \n, \t, virtual, get, set

Operators:

Operators are symbols used for mathematical and logical operations. The specification includes arithmetic operators (+, -, \*, /), increment and decrement operators (++ and --), assignment operators (=, +=, -=, \*=, /=), comparison operators (==, !=, <, >, <=, >=), and logical operators (!, |, &).

-) Arithmetic operators:

+, -, \*, / add, sub, mul, div

++, -- increment and decrement

-) Assignment operators:

= , +=, -=, \*=, /=

-) Comparison operators:

== (equal to)

!= (not equal to)

< (less than), > (greater than), >= greater than or equal to, <= less than or equal to

-) Logical operators:

! (not), | (or), & (and)

Object Oriented Programming:

OOP is a programming paradigm that focuses on objects and classes. Classes define the blueprint for creating objects, which are instances of classes. The specification shows class and object creation, inheritance, and constructors/destructors.

class Class\_name {

public: # Access specifier

# fields

# functions

# blocks

};

Class\_name ObjectName; -> object creation

Constructors/Destructors:

Class\_name() ---> Class name is the constructor

~ Class\_name() ---> Class name is the destructor

e.g:

class Scaler {

public:

# Constructor

Scaler() {

# Constructor body.

}

# Deconstructor

~Scaler(){

};

};

Inheritance:

Base class (parent): the class being inherited from

Derived class (child): the class that inherits from another class

# Base class

class Vehicle {

public:

string brand = "Ford";

void door\_open() {

print( "Door is opened \n") ;

}

};

# Derived class

class Car: public Vehicle {

public:

string model = "Mustang";

};

int main() {

Car myCar;

myCar.door\_open();

print(myCar.brand + " " + myCar.model);

return 0;

}

Abstract:

An abstract class is a class that cannot be instantiated on its own but serves as a blueprint for other classes. It may contain pure virtual functions, which are meant to be overridden by derived classes.

// An abstract class

class Test {

// Data members of class

public:

// Pure Virtual Function

virtual void show() = 0;

/\* Other members \*/

};

e.g:-

// C++ program to illustrate the abstract class with pure

// virtual functions

**Encapsulation:**

#include <iostream>

using namespace std;

class Test {

// private member variable

int x;

public:

// pure virtual function

virtual void show() = 0;

// getter function to access x

int getX() { return x; }

};

int main(void)

{

// Error: Cannot instantiate an abstract class

Test t;

return 0;

}

Data Structure:

-)Struct:

A struct is a composite data type that groups related variables together. It allows the creation of custom data types containing different types of variables.

struct Books {

string title[50];

string author[50];

string subject[100];

int book\_id;

};

struct Books book1; declare book1 of type Book

struct Books book2; declare book2 of type Book

Arrays:

Arrays are collections of elements of the same data type, accessible through an index. The specification includes 1D, 2D, and 3D arrays.

-) 1D array:

string course = [“OS”, “OR”, “LA”, “OOP”]; #initialization of array

int car[6]; # declaration of array

print(“The 1st course is :”, course[0]) Access the array elements , output will be OS

course[2] = “DS”; Update array element

print(“The 3rd course is :”, course[2]) #output will be DS

-) 2D array:

int x[2][2] = { {2, 4},

{3, 5} }

print(x[0][1]) # output will be 4

# x[0][0]= 9 update

-) 3D arrays:

int y[2][2][3] = { { {1, 2, 3}, {4, 5, 6} },

{ {7, 8, 9}, {10, 11, 12} } };

print(y[1][0][2]) # output will be 9

example code:

int main() {

int numbers[2][3];

print("Enter 6 numbers: ");

# Storing user input in the array

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

take(numbers[i][j]);

};

};

print("The numbers are: “);

# Printing array elements

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

print( "numbers[" i "][" j "]: ",numbers[i][j]);

};

};

return 0;

};

Encapsulation:

Encapsulation is an OOP principle that restricts access to the internal data of a class and exposes methods to interact with that data.

class Employee {

private:

# Private attribute

float salary;

public:

# Setter

void setSalary(int s) {

salary = s;

}

# Getter

float getSalary() {

return salary;

}

};

int main() {

Employee myObj;

myObj.setSalary(50000);

cout << myObj.getSalary();

return 0;

}

Exception Handling:

Exception handling is a mechanism used to deal with errors or exceptional situations that may occur during program execution. The "check" and "catch" blocks handle exceptions.

check {

# code to try

throw exception; # If a problem arises, then throw an exception

}

catch () {

# Block of code to handle errors

}

if (divisor == 0) {

throw runtime\_error("Division by zero is not allowed"); // Throw an exception if the divisor is zero

}

int result = 10 / divisor;

print("10 divided by ", divisor, " is: ", result);

} catch (const exception& ex) {

print("An error occurred: ", ex.what()); // Handle the exception and display the error message

}