**Data types din c++**

In C++, data types are used to specify the type of data that a variable can hold. C++ supports several built-in data types, which can be broadly classified into the following categories:

1. Primitive Data Types:
   * Integer Types: Used to store whole numbers without fractional parts.
     + int: Standard integer type, typically 4 bytes.
     + short: Short integer type, typically 2 bytes.
     + long: Long integer type, typically at least 4 bytes.
     + long long: Long long integer type, typically at least 8 bytes (C++11).
   * Floating-Point Types: Used to store numbers with fractional parts.
     + float: Single-precision floating-point type.
     + double: Double-precision floating-point type.
     + long double: Extended precision floating-point type.
   * Character Types: Used to store individual characters.
     + char: Character type, typically 1 byte.
     + wchar\_t: Wide character type, typically 2 or 4 bytes.
     + char16\_t: 16-bit character type (C++11).
     + char32\_t: 32-bit character type (C++11).
   * Boolean Type:
     + bool: Boolean type, can hold either true or false.
   * Void Type:
     + void: Represents the absence of type.
2. Derived Data Types:
   * Array Types: Used to store a fixed-size collection of elements of the same type.
   * Pointer Types: Used to store memory addresses.
   * Reference Types: Used to provide aliases to existing variables.
   * Enumeration Types: Used to define user-defined enumeration types.
   * User-Defined Types:
     + struct: Used to define a structure containing multiple elements.
     + class: Used to define a class containing data members and member functions.
     + union: Used to define a union, which is a special data structure that can hold multiple data members but only one at a time.

**Print hello world in c++**

#include <iostream>

using namespace std;

int main()

{

    cout << "hello world!";

}

**Printing methods**

1. Multiple cout statement print in the same line. So we can use **endl** and **“/n”** character to print in next line

int main()

{

    cout << "hello world!";

    cout << "\n ram dulari "; << using character “\n”

}

int main()

{

    cout << "hello world!";

    cout << endl; << next line using “endl” statement

    cout << "ram dulari ";

}

1. Print multiple statements using single cout statement

cout << "hello world!"<<endl<<"hello anjali!";

**Variables in C++**

In C++, variables are used to store data that can be manipulated or accessed within a program. Here are some key points about variables in C++:

1. **Declaration**: Before using a variable, you need to declare it. The declaration specifies the variable's data type and name. For example:

int age; // Declares an integer variable named 'age'

double salary; // Declares a double (floating-point) variable named 'salary’

1. **Initialization**: Variables can be initialized at the point of declaration:

int count = 0; // Initializes 'count' to 0

double pi = 3.14159; // Initializes 'pi' to the value of pi

1. Data Types: C++ supports various data types such as **int, double, float, char, bool,** etc., each with its own size and range of values.
2. **Scope**: Variables have a scope, which defines where in the program they can be accessed. Variables declared within a block of code (e.g., within a function) are typically local to that block, while variables declared outside of any function are typically global and can be accessed from anywhere in the program.
3. **Naming Rules**: Variable names must follow certain rules, such as starting with a letter, underscore or $ and can contain letters, digits, and underscores.

Keywords and special symbols are not allowed in identifiers.

1. **Constants**: You can declare constants using the const keyword, which means their value cannot be changed once initialized:

const double PI = 3.14159;

**Operators**

1. **Arithmetic operator:** These operators perform mathematical operations such as addition, subtraction, multiplication, division, and modulus.

int a = 10, b = 3;

int sum = a + b; // Addition

int difference = a - b;// Subtraction

int product = a \* b; // Multiplication

int quotient = a / b; // Division

int remainder = a % b; // Modulus (remainder of division)

1. **Relational Operators**: These operators are used to compare two values and return a boolean result (**true** or **false**).

int x = 5, y = 3;

bool isEqual = (x == y); // Equal to

bool isNotEqual = (x != y); // Not equal to

bool isGreater = (x > y); // Greater than

bool isLess = (x < y); // Less than

bool isGreaterOrEqual = (x >= y); // Greater than or equal to

bool isLessOrEqual = (x <= y); // Less than or equal to

1. **Logical Operators**: These operators perform logical operations on boolean values.

bool p = true, q = false;

bool logicalAnd = p && q; // Logical AND

bool logicalOr = p || q; // Logical OR

bool logicalNotP = !p; // Logical NOT

1. **Assignment Operators**: These operators are used to assign values to variables.

int a = 5;

a += 3; // Equivalent to: a = a + 3

1. **Increment and Decrement Operators**: These operators are used to increase or decrease the value of a variable by one.

int count = 0;

count++; // Increment count by 1

count--; // Decrement count by 1

1. **Conditional Operator (Ternary Operator)**: This operator is used for conditional expressions. It evaluates a condition and returns one of two values depending on whether the condition is true or false.

int x = 5, y = 3;

int maxNum = (x > y) ? x : y; // If x > y, maxNum = x, otherwise maxNum = y

1. **Bitwise Operators**: These operators perform bitwise operations on integer operands at the bit level.

int a = 5, b = 3;

int bitwiseAnd = a & b; // Bitwise AND

int bitwiseOr = a | b; // Bitwise OR

int bitwiseXor = a ^ b; // Bitwise XOR

**operator precedence and associativity**

In C++, operator precedence and associativity dictate the order in which operators are evaluated in an expression. Understanding these concepts is crucial for writing correct and predictable code. Here's a brief overview:

1. \*\*Operator Precedence\*\*: Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence. For example, in the expression `a + b \* c`, the multiplication operator (`\*`) has higher precedence than the addition operator (`+`), so `b \* c` is evaluated first, and then the result is added to `a`.

Here's a summary of operator precedence in C++, from highest to lowest:

- Postfix operators: `() [] -> . ++ --`

- Unary operators: `+ - ! ~ ++ -- \* & sizeof`

- Multiplicative operators: `\* / %`

- Additive operators: `+ -`

- Shift operators: `<< >>`

- Relational operators: `< <= > >=`

- Equality operators: `== !=`

- Bitwise AND: `&`

- Bitwise XOR: `^`

- Bitwise OR: `|`

- Logical AND: `&&`

- Logical OR: `||`

- Conditional operator: `?:`

- Assignment operators: `= += -= \*= /= %= &= ^= |= <<= >>=`

- Comma operator: `,`

2. \*\*Associativity\*\*: Associativity determines the grouping of operators with the same precedence. It specifies whether operators are evaluated from left to right (left-associative) or right to left (right-associative). For example, in the expression `a = b = c`, the assignment operator (`=`) is right-associative, so `b = c` is evaluated first, and then the result is assigned to `a`.

Most operators in C++ are left-associative, meaning they are evaluated from left to right. However, some operators, such as the assignment operators (`=`), are right-associative.

It's important to be aware of operator precedence and associativity to avoid unexpected behavior in your code. Parentheses can be used to explicitly specify the order of evaluation when necessary.

**Taking input in c++**

In C++, you can receive input from the user using the std::cin object, which is part of the standard input/output library (<iostream>). Here's a basic example of how to use std::cin to get input from the user:

int main(){

    int rad ;

    cout <<"enter the radius of the circle";

    cin >> rad; // **taking user input here**

    cout<<"Area of circle is :"<<M\_PI\*pow(rad,2);

    return 0;}

**Type casting**

Type casting in C++ refers to converting a value from one data type to another. There are two main types of type casting in C++: implicit and explicit casting.

1. **Implicit Casting**: Implicit casting, also known as automatic type conversion, occurs when the compiler automatically converts one data type to another without any explicit instruction from the programmer. This typically happens when the conversion is safe and doesn't result in loss of information. For example:

```cpp

int numInt = 10;

double numDouble = numInt; // Implicitly converts int to double

```

In this example, the integer value `10` is implicitly converted to a double when assigned to `numDouble`.

2. **Explicit Casting:** Explicit casting, also known as type casting or type conversion, is done explicitly by the programmer using casting operators. There are several casting operators in C++:

Static Cast: It performs conversions between related types, such as numeric types, and is checked at compile-time.

```cpp

double numDouble = 3.14;

int numInt = static\_cast<int>(numDouble); // Explicitly converts double to int

```

Dynamic Cast: It is used for downcasting in polymorphic class hierarchies (with inheritance) and is checked at runtime.

Const Cast: It is used to add or remove const-ness or volatile-ness of a variable.

Reinterpret Cast: It is used to convert one pointer type to another, even if they are unrelated.

```cpp

double numDouble = 3.14;

int numInt = static\_cast<int>(numDouble); // Explicitly converts double to int

```

In this example, `static\_cast` is used to explicitly convert the double value `3.14` to an integer, resulting in the value `3`.

3. **C-style Casting**: C-style casting is another form of explicit casting that allows you to perform a variety of conversions. It is less safe than the C++ casting operators because it doesn't provide compile-time checks.

```cpp

double numDouble = 3.14;

**int numInt = (int)numDouble;** // Explicitly converts double to int using C-style cast

```

In this example, `(int)` is used to explicitly cast `numDouble` to an integer.

It's important to use explicit casting judiciously and ensure that conversions are safe and appropriate to avoid potential errors and loss of data. Additionally, prefer C++ casting operators over C-style casting for improved type safety and readability.

**Note: float b = 5/2 output is 2 because it consider both operand as int and ans of int is int.**

**Conditional statements**

Conditional statements in C++ allow you to execute different blocks of code based on certain conditions. The primary conditional statements in C++ are if, else if, else, and switch. Here's how each of them works:

1. **if statement**: The **if** statement is used to execute a block of code if a specified condition is true. It can be followed by an optional **else if** and **else** statement.

int x = 10;

if (x > 0) {

std::cout << "x is positive." << std::endl;

} else if (x < 0) {

std::cout << "x is negative." << std::endl;

} else {

std::cout << "x is zero." << std::endl;

}

1. **switch statement**: The **switch** statement is used to select one of many code blocks to be executed based on the value of an expression.

int day = 4;

switch (day) {

case 1:

std::cout << "Monday" << std::endl;

break;

case 2:

std::cout << "Tuesday" << std::endl;

break;

case 3:

std::cout << "Wednesday" << std::endl;

break;

default:

std::cout << "Invalid day" << std::endl;

}

**STL OF C++**

STL is one of the unique abilities of C++ which makes it stand out from every other programming language. STL stands for standard template library which contains a lot of pre-defined templates in terms of containers and classes which makes it very easy for developers or programmers to implement different data structures easily without having to write complete code and worry about space-time complexities.

1. **Pairs :-** In C++, a pair is a template class from the Standard Template Library (STL) that allows you to store two heterogeneous objects as a single entity. It's quite handy when you need to associate two values together. Here's a basic example of how to use pairs in C++:

E.g.

pair<int,int> p ={1,3};

cout<<p.first << “ ” << p.second;

pair<int,pair<int,int>> p = {1,{3,4}};

cout<< p.first << “ ” << p.second.second << “ ” << p.second.first;

pair<int, int> arr[] = {{1,2},{2,5},{5,1}};

cout << arr[1].second;

1. **Vectors :** vectors are a part of the Standard Template Library (STL) and are a dynamic array data structure. You can think of them as arrays that can resize themselves automatically when more elements are added to them.

**Need to include before use #include<vector>**

* **Vector<int> v :** is used to declare the empty vector or dynamic array.
* **V.push\_back(1):** it is used to push the element at the end of array.
* **Vector<pair<int,int>> vec:** this is used to create the vector of pair.

**v.push\_back({1,2});**

**v.emplace\_back(1,2);**

* **Vector<int> v(5,100):** create array {100,100,100,100,100}
* **Vector<int> v(5):** create array {0,0,0,0,0}
* **Vecort<int> v2(v1):** crate vector using vector.

**Printing the vector**

1. **vector<int>::iterator it = v.begin();**

**cout << \*(it) :** iterator will be point the address of the element.

**It++ :** it will move the pointer to point the memory of next element.

**vector<int>::iterator it = v.end();**

**vector<int>::iterator it = v.rend();**

**vector<int>::iterator it = v.rbegin();**

1. **V[index]:**  we can access the element using index.

**Delete elements in vector**

**v.erase(v.begin()+1);**

**v.erase(v.begin()+2,v.begin()+4); give start and end does not include end**

**Insert in vector**

**v.insert(v.begin(),300) :**insert 300 at beginning.

**v.insert(v.begin()+1,2,10):** insert 10 2 times at beg+1 position.

3. **List:** it is similar to vector but it allows operation to insert at begning.

**List<int> ls;**

**Ls.push\_front(5);**

**All other functions are remain same.**

1. **Dequeue:** it is similer to list and vector allows operation like push\_back,push\_front, insert etc.

**Eg.** Deque<int> dq;

Dq.push\_back(1);

1. **Stack:** it is follow filo operation.

**Eg.** Stack<int> st;

**St.**push(1);

**St.**top(); return top element;

**St.**size(); return size;

**St.**empty(); empity stack;