2. Variables, Data Types, and Operators

1. What are the different data types available in C++? Explain with examples.

❖ C++ features several data types categorized into primitive, derived, and user-defined types.

1. Primitive Data Types: These are fundamental built-in types.

   i. int: Stores whole numbers (integers).

   ```
   int age = 30;
   ```

   ii. char: Stores a single character.

   ```
   char grade = 'A';
   ```

   iii. float: Stores single-precision floating-point numbers (decimals).

   ```
   float pi = 3.14f; // 'f' suffix indicates float literal
   ```

   iv. double: Stores double-precision floating-point numbers, offering more precision than float.

   ```
   double temperature = 25.5;
   ```

   v. bool: Stores Boolean values, either true or false.

   ```
   bool isActive = true;
   ```

   vi. void: Represents the absence of a type, typically used for functions that do not return a value or for generic pointers.

   ```
   void printMessage () {
       // Function does not return a value
   }
   ```

2. Derived Data Types: These are built upon primitive data types.

   i. Arrays: Collections of elements of the same data type, stored contiguously.

   ```
   int numbers 5] = {1, 2, 3, 4, 5};
   ```

   ii. Pointers: Variables that store memory addresses.

   ```
   int value = 10;
   int* ptr = &value; // ptr stores the address of 'value'
   ```

iii. References: Aliases for existing variables.

```
int a = 5;
int& b = a; // b is an alias for a
```

iv. Functions: Blocks of code designed to perform a specific task.

```
int add (int x, int y) {
   return x + y;
 }
```

3. User-Defined Data Types: These are created by the programmer.

i. struct (Structures): Groups variables of different data types under a single name.

```
struct Person {
   char name [50];
   int age;
};
Person p1;
```

ii. class (Classes): Blueprints for creating objects, encapsulating data (member variables) and functions (member methods).

```
class Car {
 public:
   void start () {/* ... */ }
 };
Car myCar;
```

iii. union (Unions): Allows different data types to share the same memory location.

```
union Data {
   int i;
   float f;
 };
Data d;
d.i = 10; // Stores 10 as an integer
```

iv. enum (Enumerations): Define a set of named integer constants.

```
enum Day {Monday, Tuesday, Wednesday};
Day today = Monday;
```

v. typedef: Creates an alias for an existing data type.

```
typedef long long int LLI;
LLI bigNumber = 123456789012345LL;
```

2. Explain the difference between implicit and explicit type conversion in C++.

Feature	Implicit Type Conversion	Explicit Type Conversion
Initiation	Automatic by compiler	Manual by programmer
Syntax	No specify syntax required.	Uses cast operator (e.g static_cast) or conversion functions.
When used	In operations involving different data types.	When specific type conversions are needed or data loss is possible.
Safety	Generally safe for widening conversions, can lead to data loss or unexpected results for narrowing conversions.	Programmer has more control, But must be aware of potential issues.
Clarity	Less explicit can be harder to understand code intent.	More explicit improves code readability and intent.
Data Loss	Can occur without programmer's awareness.	Programmer is aware of the potential for data loss.

3. What are the different types of operators in C++? Provide examples of each.

❖ Types of Operators in C++ :-

1. Arithmetic Operators: Used for mathematical operations.

Operator	Description	Example
+	Addiiton	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	Division	a/b
%	Modulus (Reminder)	a%b

2. Relational (Comparison) Operators: Used to compare two values.

Operator	Description	Example
==	Equal To	a == b
!=	Not Equal to	a != b
>	Greater Than	a > b
<	Lesser Than	a < b
>=	Greater Than Equal To	a >= b
<=	Lesser Than Equal to	a <= b

3. Logical Operators: Used to combine multiple conditions.

Operator	Description	Example
&&	Logical AND	(a>0 && b>0)
`		`
!	Logical NOT	1(a>b)

4. Assignment Operators: Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 10

+=	Add and assign	a += 5
-=	Subtract and assign	a -= 2
*=	Multiply and assign	a *= 3
/=	Divide and assign	a /= 2
%=	Modulo and assign	a %= 3

5. Increament and Decrement Operators: Used to increase and decrease a value by 1.

Operator	Description	Example
++	Increament	a++
--	Decrement	a--

6. Bitwise Operators: Used operator at bit level.

Operator	description	Example
&	AND	a & b
`		`
^	XOR	a ^ b
~	NOT	~a
<<	Left Side	a << 2
>>	Right Side	a >> 3

7. Conditional (Ternary) Operator: A shorthand for if-else.

```
condition ? expr1 : expr2
int a =10, b=20;
Int max = (a>b) ? a : b;
```

8. Special Operator
    - sizeof - Returns size of a data type or variable.
    - typeid – Returns type information (used in RTTI)
    - & - Address of operator.
    - * - Pointer dereference.
    - -> - Member access via pointer.

- . – Member access via object.

4. Explain the purpose and use of constants and literals in C++.

- **Constants in C++**

  A **constant** is a variable whose value cannot be altered after its initialization. Declaring constants ensures that certain values remain unchanged throughout the program, enhancing code reliability and readability.

- Ways to Define Constants:
  1. Using the const Keyword:
     const int MAX_USERS = 100;
     Here, MAX_USERS is a constant integer. Any attempt to modify its value will result in a compile-time error.
  2. Using the #define Preprocessor Directives:
     #define PI 3.14159;
     This defines PI as a macro representing the value 3.14159. Note that macros are replaced by their values during preprocessing and do not have type safety.
  3. Using constexpr (Introduced in C++11):
     constexpr int BUFFER_SIZE = 256;
- constexpr defines a constant expression that is evaluated at compile time, ensuring better optimization and type safety

- **Literals in C++**

  A literal is a fixed value directly embedded in the source code. Literals represent constant values and are used to initialize variables or constants.

Types of Literals:

1. Integer Literals:

- Decimal: int dec = 42;
- Octal: int oct = 052;
- Hexadecimal: int hex = 0x2A;
- Binary (C++14 and above): int bin = 0b101010;

2. Floating-Point Literals:
   - Standard Notation: float f = 3.14f;
   - Scientific Notation: double d = 1.22e11;
3. Character Literals:
   - char c = 'A';
   - wchar_t wc = L'Ω'; (wide character)
4. String Literals:
   - const char* s = "Hello, World!";
   - const wchar_t* ws = L"Wide String";
5. Boolean Literals:
   - bool isTrue = true;
   - bool isFalse = false;
6. User-Defined Literals (C++11 and above):
   - Allows creating custom literals by overloading the literal operators.
   - Example: Defining a literal for meters:

```
constexpr long double operator"" _m(long double x) {
   return x * 1000;
}
auto distance = 1.5_m; // 1500
```

- **Relationship Between Constants and Literals**

  While both constants and literals represent fixed values, their roles differ:

  - **Literals** are the actual fixed values written directly in the code (e.g., 42, 'A', "Hello").
  - **Constants** are named entities that hold fixed values, often initialized using literals.

  Example:

  ```
  const int MAX_SCORE = 100; // '100' is a literal assigned to the constant 'MAX_SCORE'
  ```

  - Using constants instead of repeating literals throughout the code enhances maintainability and readability.