U2 and U3 – C++

PROGRAMMING LANGUAGES

- A set of grammatical rules for – instructing a computer / computing device to perform specific tasks.
- Includes **High – level languages** → BASIC, C, C++, COBOL, Java, FORTRAN, Ada, Pascal
- **2 types –**

Procedural Programming		**Object Oriented Programming**
A programming model derived from – **structured programming** – based upon the concept of **calling** procedure.	DEFINITION	A programming model based upon the concept of **objects.** Objects contains – data in the form of **attributes (variables / data)** + Code in the form of **methods (functions).**
Absent	ACCESS SPECIFIER	Have access specifiers like – public, private, protected
DISADVANTAGE – **1.** Adding new data and function – not easy **2. Less secure –** no proper way of hiding data	FEATURES	**ADVANTAGE OF OOP vs PROCEDURAL –** **1.** Adding new data and function – is easy **2. More secure –** provides data hiding
C FORTRAN BASIC Pascal	EXAMPLES	C++ Java Python C#

Basic I/O Programs

INTRODUCTION:

C++ is a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.
One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high – level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications.

ADVANTAGES OF C++

1. **Object – Oriented Programming –** The major principles of the Object – Oriented Programming language, C++ include classes, inheritance, encapsulation, and data abstraction and polymorphism, that enhance the code reusability and strengthen program stability and dependability.
2. **Multi – paradigm language –** Procedural, object – oriented and generic programming are among the several programming paradigms that C++ is compatible with.
3. **Portability –** With the combined features of an intermediate level of programming language, C++ codes require no alterations in order to be compiled and executed on numerous platforms and operating systems.
4. **Memory Management –** In addition to supporting inline functions, exception handling, pointers and references, C++ provides an efficient memory management and, enabling developers to optimize applications for high performance.
5. **Rich Library Support –** The C++ Standard Template Library (STL) provides a large framework that can be used to streamline and accelerate processes including access to database systems.

DISADVANTAGES OF C++

1. **Complexity –** Programmers who are unfamiliar with Object – Oriented Programming, C++ can be a challenging language to master.
2. **Lack of Built – in Support for Web Development –** Certain web development tasks such as AJAX and server – side scripting may require additional efforts to implement since C++ lacks the built – in support for such tasks.
3. **Absence of Integrated Multi – threading Support –** Developing parallel programs may become challenging due to a lack of built – in functionality for multi – threaded applications in C++.
4. **Security – related concerns –** Features such as pointers are vulnerable to null pointer dereferencing attacks, which may lead to program reliability issues.
5. **Potential Memory Leaks –** C++ programming language lacks the potential for garbage collection which causes the developers to manually manage memory allocation and deallocation, leading to an increase in the complexity of the code and memory leaks.

APPLICATIONS OF C++

1. **Web browsers and Operating Systems** – C++ is a fast and strongly-typed programming language which makes it an ideal choice for developing operating systems. A significant portion of Mac OS X, along with several softwares from Microsoft including Windows and IDE Visual Studio, is written in C++. Web browsers such as Mozilla Firefox are completely developed from C++ due to the need for rapid execution, low latency and high efficiency.

2. **Database Software** – C++ is used in developing database software, which requires efficient handling of data and performance.

3. **Banking Applications** – Since banking applications require concurrency and high performance, C++ is the default choice of programming language. Infosys Finacle is a popular banking application developed using C++.

4. **Embedded Systems** – C++ is used in various embedded systems, such as smartwatches and medical devices, where performance and efficiency are crucial.

5. **In-game Programming** – C++ is widely used in game development due to its speed, object-oriented programming capabilities, and ability to manipulate resources and override the complexities of 3D games efficiently.

DATA TYPES IN C++

All variables use data type during declaration to restrict the type of data to be stored.

C++ supports the following data types –

1. **Primitive Data Types –**
   Also termed as primary / fundamental data types, these are built – in data types that can be directly used by the user to declare variables.
   The operator sizeof( ) is used to find the number of bytes occupied by the data type in the computer memory.

Data Type	**Keyword**	**Description**	**Memory (in bytes)**	**Example**
Boolean	bool	It stores only 2 values – (a) true (returns 1) (b) false (returns 0).	1	bool val = true;
Character	char	It stores a single character. The character must be surrounded by single quotes. Alternatively, ASCII values may also be used to display certain characters.	1	char grade = ‘A’; char a = 65;
Integer	int	It stores a whole number without decimals.	2 or 4	int num = 5;
Floating point	float	It stores floating point numbers (with decimals). Precision = 6 – 7 decimal digits.	4	float num = 9.75;
Double Floating Point	double	It stores floating point numbers (with decimals). Precision = 15 decimal digits.	8	double num = 10.345;

2. **Derived Data Types –** Data types derived from the primitive or built – in data types are referred to as Derived Data Types. These data types comprise of –
    (a) functions
    (b) arrays
    (c) pointers
    (d) references

3. **User – defined Data Types –** Also termed as Abstract data types, user – defined data types such as – class, structure, union and enumeration – are defined by the user itself.

HEADER FILE AND FUNCTIONS

C++ uses a convenient abstraction called streams, which are sequences of bytes that allow data to be transferred between devices and memory. They are used to perform input and output operations in sequential media such as the screen, the keyboard, or a file.

The **<iostream>** header file is a part of the C++ standard library and is used for input and output operations. It defines the standard input/output stream objects, such as cin, cout, cerr, and clog. Including this header file in a C++ program allows you to perform input and output operations using these stream objects.

Key features of the <iostream> header file include:

Object	**Description**	**Operator used**	**Syntax**	**Output**
cout	It is a standard output stream object, usually connected to the display screen. It is used to output values/print text.	Extraction (<<)	cout << "Hello World";	Hello World

cin	It is a standard input stream object, usually connected to the keyboard. It is used to accept the input from the standard input device.	Insertion (>>)	int a;
			cout << "Enter a number :";
			cin >> a;
			cout << "a = " << a;
				Enter a number : <u>10</u>
				a = 10

endl	It is a stream manipulator and a pre – defined object used to insert a newline character and flush the output stream, ensuring that any buffered output is displayed immediately.	Extraction (<<)	int a = 10;
			cout << "Num1: " << endl;
			cout << a;
				Num1:
				10

Conditional Statements and Loops

INTRODUCTION:

C++ is an object-oriented, multi-paradigm language that was developed as an extension of the C programming language and was designed for system and application programming. It supports procedural, functional, and generic programming styles.

In C++ programming language, conditional statements and loops are the fundamental concepts essential for controlling the flow of a C++ program and for implementing repetitive tasks. They are crucial for comprehending and developing efficient and effective C++ programs.

CONDITIONAL STATEMENTS

Conditional statements, also referred to as decision control structures, are utilized for decision – making purposes in C++ programs. These conditional statements are employed to assess and evaluate one or more conditions / criteria and determine whether or not to execute a series of statements or a specific block of code. Such decision – making statements in programming languages decide the direction of the flow of program execution.

Types of Conditional Statements

1. **if Statement** –
   The **if** conditional statement is used to decide whether or not a specific statement or block of statements will be executed if a particular condition is true.

   Syntax – if (condition)
   ```
   if (condition)
   {
       // code to be executed if the condition returns true
   }
   ```

   Diagram –

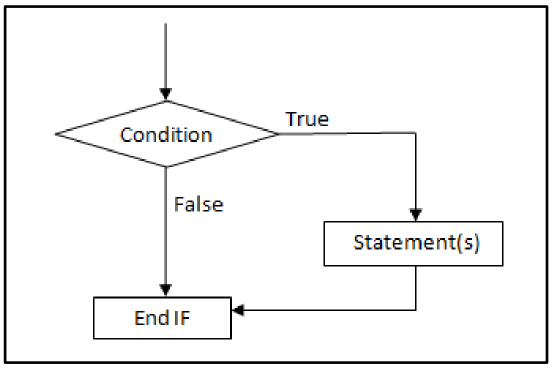

Figure 1: Working of **if** conditional statement

2. **if – else Statement** –
   The **if - else conditional statement** is a two – way branching statement. It consists of two blocks of statements each enclosed inside **if block** and **else block** respectively. If the condition inside **if statement** is true, statements inside **if block** are executed, otherwise statements inside **else block** are executed.

   Syntax – if (condition)
   ```
   {
       // code to be executed if the condition returns true
   }
   else
   ```

```
{
    // code to be executed if the condition returns false
}
```

Diagram –

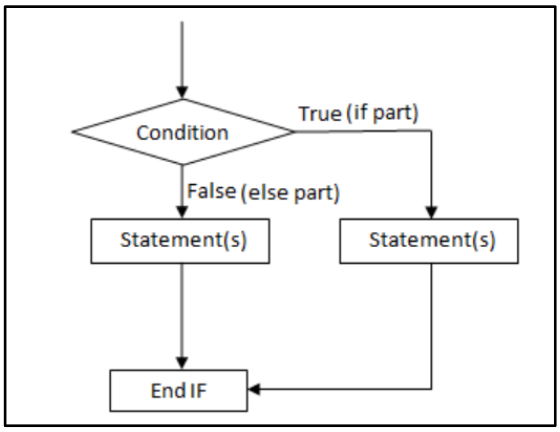

Figure 2: Working of **if – else** conditional statement

3. **if – else – if Ladder –**
The **if – else – if Ladder** is used when more than one condition is to be checked. A block of statement is enclosed inside **if, else if** and **else** part. Conditions are checked in each **if** and **else if** part. If the condition is true, the statements inside that block are executed. If none of the conditions are true, the statements inside else block are executed. Such a statement must have only one **if block** but can have as many **else if block** as required.

Syntax – if (condition1)

```
{
    // code to be executed if condition1 returns true
}
else if (condition2)
{
    // code to be executed if condition1 returns false
    // and condition2 returns true
}

else if (conditionN)
{
    // code to be executed if previous conditions return false
    // and conditionN returns true
}
else
{
    // code to be executed if all the previous conditions return false
}
```

Diagram –

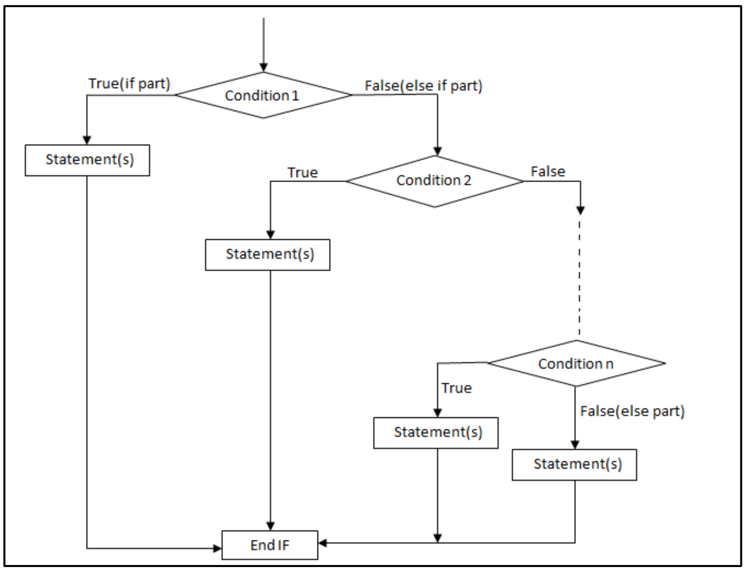

Figure 3: Working of **if – else – if Ladder**

4. **<u>Nested if Statement</u>** –

When an **if statement** is kept inside another **if statement**, it is called nested if statement. **Nested if statements** are used if there is a sub condition to be tested.

Syntax –
```
if (condition1)
{
    // Nested if statement to be executed if condition1 returns true
    if (sub_condition1)
    {
        // code to be executed if sub_condition1 returns true
    }
}
else if (condition2)
{
    // code to be executed if condition1 returns false
    // and condition2 returns true



    // Nested if statement to be executed if condition2 returns true
    if (sub_condition2)
    {
        // code to be executed if sub_condition2 returns true
    }
}
```

Diagram –

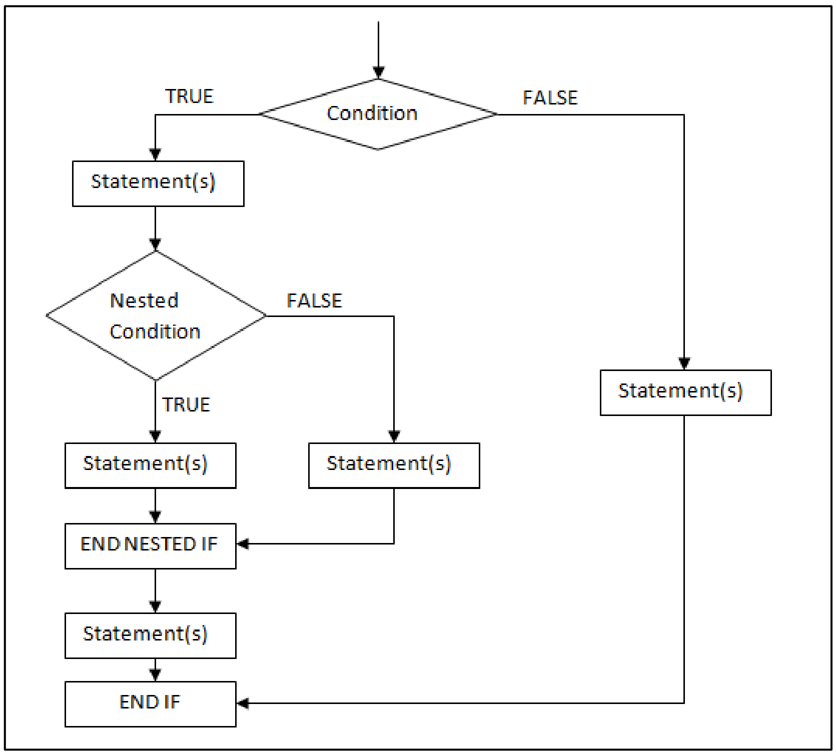

Figure 4: Working of **Nested if** conditional statement

LOOPS

Loops enable the repeated execution of a sequence of instructions until a predetermined condition is met. Following are the parts of a loop –

1. **Initialization Expression** – initializes the loop variables in the beginning of the loop.
2. **Test Expression** – decides whether the loop will be executed (if the test expression is true) or not (if the test expression is false).
3. **Update Expressions** – update (increment / decrement) the values of the loop variables after every iteration of the loop.
4. **Body of the loop** – contains a set of statements to be executed repeatedly.

Types of Loops

1. **Entry – Controlled Loops –**
   In this type of loop, the test expression is checked before entering the body of the loop.

   (a) for loop –
   A **for loop** is a repetition control structure that allows us to write a loop that is executed a specific / pre – determined number of times. The loop enables us to perform 'n' number of steps sequentially in one line.

   Syntax –
```
for (initialization_expression ; test_expression ; update_expression)
{
    // code to be executed
}
```

Diagram –

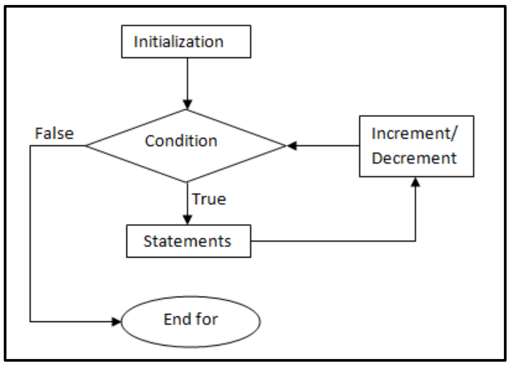

Figure 5: Working of **for loop**

(b) while loop –

A **while loop** repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body. The loop consists of the test expression whereas the initialization expression and update expressions are addressed elsewhere.

Syntax –
```
initialization_expression;
while(test_expression)
{
    // code to be executed
    update_expression;
}
```

Diagram –

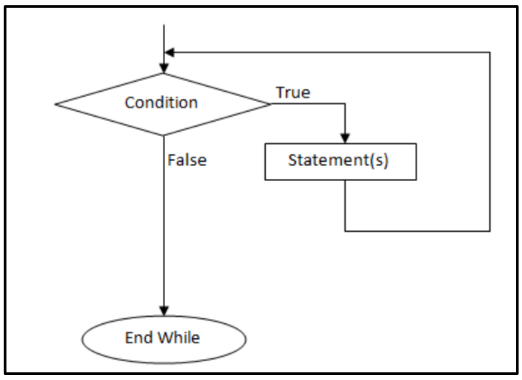

Figure 6: Working of **while loop**

2. **Exit – Controlled Loops –**

In this type of loop, the test condition is checked at the end of the loop body. Therefore, the loop body will execute a set of statements at least once, irrespective of whether the test expression is true or false.

do – while loop is a type of exit – controlled loop where the test expression is tested at the end of the loop body and the loop is terminated on the basis of the test conditions after executing the loop body at least once.

Syntax –
```
initialization_expression;
do
{
    // code to be executed
    update_expression;

} while(test_expression);
```

Diagram –

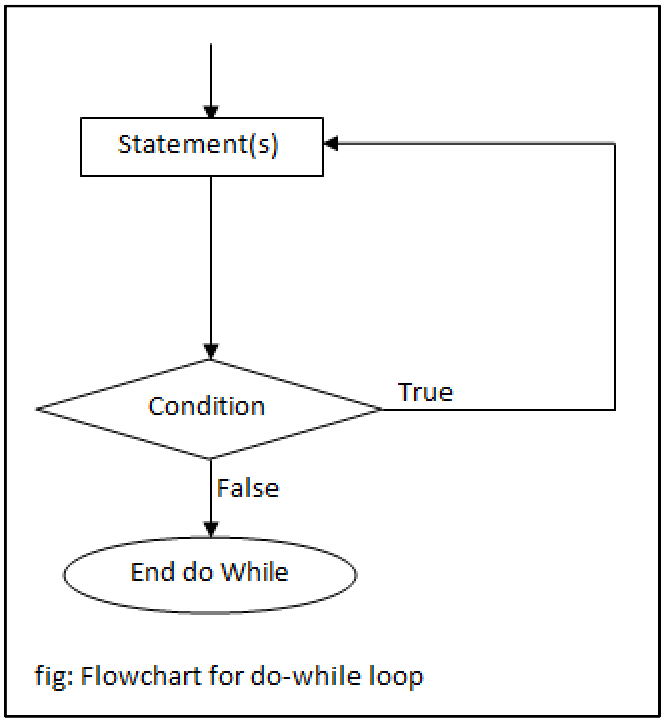

Figure 7: Working of **do – while loop**

Arrays

INTRODUCTION:

In C++ programming language, an **array** is a data structure that is used to store multiple values of similar data types in a contiguous memory location. A number of arithmetic and dynamic programming methods rely on arrays as their efficient means of implementing data structures like lists and vectors. Essentially, arrays are used to store intermediate solutions of a more complex problem as well as to perform sorting and searching operations. Matrix operations, cryptography, data mining and signal processing are few of the major applications where arrays can be employed efficiently.

Following are the properties of an array in the C++ programming language –

1. Indexing of an array commences from 0, hence storing the first element at the 0th index, the second at the 1st, and so on. Hence, Elements of an array can be individually referenced and modified using their indices, which range from 0 to (size_of_the_array -1). For instance, arr[3] retrieves the fourth element of the array 'arr'.
2. The size of an array remains constant throughout the program post declaration of the array. The operator **sizeof** can be used to determine the number of elements in an array.
3. An array can have multiple dimensions.

Property	Syntax	Example
Declaration of an Array		
Declaration of an Array	data_type array_name [size_of_the_array];	int arr[5];
Initialization of an Array		
(a) With Values	data_type array_name [size_of_the_array] = {element1, element2, element3….elementN}; // Number of elements are equal to the size of the array	int arr[5] = {1, 2, 3, 4, 5};
(b) With Values and Without Size	data_type array_name[ ] = {element1, element2, element3….elementN };	int arr[ ] = {1, 2, 3, 4, 5};

(c) After Declaration using loops	for (i = 0 ; i <= size_of_array ; i++)
	{
	arr[i] = element_value;
	}
		int n = 10;
		int value = 1;
		for (int i = 0 ; i <= n ; i++)
		{
		arr[i] = value;
		value++;
		}

(d) Partial Initialization	data_type array_name [size_of_the_array] = {element1, element2, element3….elementN}; // Number of elements are less than the size of the array	int partial_arr[5] = {1, 2};

TYPES OF ARRAYS

1. **One – dimensional Arrays –**
   A one – dimensional array is a group of elements having the same datatype which are stored in a linear arrangement under a single variable name. It requires only one indices specification in order to access a particular element of the array.
   Representation: They represent multiple data items as a list.
   Total Memory (in bytes) = Size of (base type) * Size of the array
   Syntax for Declaration – data_type array_name [size_of_the_array]
   Example for Declaration and Initialization: int arr[3] = {1, 2, 3};

2. **Two – dimensional Arrays –**
   Also termed as a 'matrix', two – dimensional arrays are the simplest form of multi – dimensional arrays in which each array is itself an element. It requires two index specifications (row number and column number) in order to access a particular element of the array.
   Representation: They represent multiple data items in a tabular format consisting of rows and columns.

   Total Memory (in bytes) =
   Size of (base type) * (Number of rows) * (Number of columns)

   Syntax for Declaration –
   data_type array_name [number_of_rows][number_of_columns]

   Example for Declaration and Initialization:
   ```
   int arr[2][5] = { {1, 2, 3, 4, 5},
                     {6, 7, 8, 9, 10}};
   ```

Functions and Structures

INTRODUCTION:

In C++ programming language, a **function** is a segment or a block of code that performs a specific task. It is a fundamental building block of C++ programs, allowing programmers to modularize their code and make it more reusable, maintainable, and understandable. Functions enable programmers to divide complex programs into smaller, more manageable units. This modular approach promotes code reuse, as functions can be easily incorporated into different parts of the program or even reused across multiple projects. The reusability of functions significantly enhances code development efficiency.

By providing a structured approach to code organization and encapsulating a specific task within functions, programmers can avoid rewriting the same code repeatedly, which not only saves time but also reduces the risk of errors and improves code maintainability by making it easier to identify and isolate issues. Functions play a crucial role in enhancing code readability. By clearly defining the purpose and scope of each code segment, functions enhance code readability. Hence, functions are essential tools for C++ programmers, enabling them to create structured, reusable, and maintainable code. By effectively utilizing functions, programmers can write well – organized, efficient, and easy – to – understand C++ programs.

Syntax for Declaration and Definition of a Function –

```
return_type function_name(parameters) // function header
{
        // function body
        // code to be executed
}
```

TYPES OF FUNCTIONS

1. **Built – in functions** –
   Also termed as Standard Library functions, built – in functions are a component of a previously defined compiler package. These functions can be accessed by the compiler from their specific header files and can be implemented in the program directly without defining them.
   Built – in functions can be used for several purposes, including performing arithmetic operations and manipulating string objects.
   Example – sqrt( ), strcat( ).

2. **User – defined functions** –
   Also termed as tailor – made functions, user – defined functions are referred to custom – designed blocks of code created by the programmer to perform specific tasks. These user – defined functions are necessary in order to enhance the reusability of the code and reduce the complexity of a large program.

Following are the types of user – defined functions –

(a) Functions with No Return Value and No Parameter –

A function that does not take any input parameters / arguments and does not return any value. The return type of the function is 'void'.

Syntax	Example
void function_name( )	void sum( )
{	{
//code to be executed	int a = 5, b = 10, c;
}	c = a + b;
	cout << "Sum =" << c;
void main( )	}
{	
// calling the function	void main( )
function_name( );	{
getch( );	sum( );
}	getch( );
	}

(b) Functions with No Return Value and With Parameter –

A function that takes one or more input parameters / arguments and does not return any value.

The return type of the function is 'void'.

These functions can be called in the main( ) function and parameters can be passed either by value or by reference.

Parameters passed during function call are termed as Actual Parameters.

Parameters received by the function are termed as Formal Parameters.

Syntax	Example
void function_name(data_type parameter)	void sum(int a, int b)
{	{
//code to be executed	int c;
}	c = a + b;
	cout << "Sum =" << c;
void main( )	}
{	
// calling the function	void main( )
function_name(parameter);	{
getch( );	sum(5, 10);
}	getch( );
	}

(c) Functions With Return Value and No Parameter –

A function that does not take any input parameters / arguments but returns a value using the keyword **return** after executing certain operations and performing a specific task. The return type of the function is the same as the data type of the value to be returned.

Syntax	Example
return_type function_name( )	int sum( )
{	{
//code to be executed	int a = 5, b = 10, c;
// return statement	c = a + b;
return value;	return c;
}	}
void main( )	void main( )
{	{
// calling the function	int s = sum( );
function_name( );	cout << “Sum =” << s;
getch( );	getch( );
}	}

(d) Functions With Return Value and With Parameter –

A function that takes one or more input parameters / arguments and returns a value using the keyword **return** after executing certain operations and performing a specific task. The return type of the function is the same as the data type of the value to be returned. The functions can be called in the main( ) function and parameters can be passed either by value or by reference.

Syntax	Example
return_type function_name(data_type parameter)	int sum(int a, int b)
{	{
//code to be executed	int c;
// return statement	c = a + b;
return value;	return c;
}	}
void main( )	void main( )
{	{
//calling the function	int s = sum(5, 10);
function_name(parameter);	cout << “Sum =” << s;
getch( );	getch( );
}	}

String Manipulation

INTRODUCTION:

In C++ programming language, **strings** are fundamental data structures used to represent and manipulate sequences of characters. They are employed in a wide range of programming tasks, from simple text processing to complex data analysis and manipulation. C++ offers two primary methods for handling strings: **C – style character arrays** and the **std::string** class.

1. **C-style character arrays** represent strings as arrays of characters terminated by a null character ('\0'). They are the type of strings that C++ inherited from C language.

2. The **std::string** class, introduced in the C++ Standard Library and defined inside <string.h> header file, provides a powerful and versatile tool for managing strings. It offers a rich set of member functions for manipulating strings, including concatenation, substring extraction, character access, and formatting. This provides many advantages over conventional C-style strings such as dynamic size, member functions, etc.

Syntax for Declaring and Defining a String –

1. **Using the keyword 'string' –** string variable_name = "string_to_be_stored";
2. **Storing as an array of characters –**

    char variable_name[size] = {'char1', 'char2'….'charN', '\0'};
    char variable_name[size] = "string_to_be_stored";

Example for Declaring and Defining a String –

string s = "HelloWorld"; // using the **string** keyword

// stored as an array of characters
char s[ ] = {'H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd', '\0'};
char s1[10] = "HelloWorld";

STRING MANIPULATION

String manipulation in C++ involves various operations such as concatenation, finding the length, searching for substrings, extracting substrings, and replacing parts of a string. String manipulation in C++ offers several advantages, including ease of use, compatibility, memory efficiency, performance overhead, and the ability to handle various data types.

Function	Description	Return type	Syntax
strcmp( )	Used to compare two null – terminated strings lexicographically by comparing the ASCII value of each character. (a) If two strings are equal, it returns 0. (b) If first string is greater than the second string, it returns a positive integer.	Integer	strcmp(string1, string2);

	(c) If first string is less than the second string, it returns a negative integer.		
strcpy( )	Used to copy one string to another.	String	strcpy(string1, string2);
strlen( )	Used to find the length of a string, including all the characters and spaces as well.	Integer	strlen(string);
strcat( )	Used to append a copy of one string to the end of another string.	String	strcat(string1, string2);

Class, Objects and Encapsulation

CLASS AND OBJECTS

A **class** in C++ programming language is the building block that leads to Object – Oriented programming. It is a user – defined data type, which holds its own data members and member functions, which can be accessed and used by creating an object (instance) of that class.

Data members are the data variables that represent the characteristics or properties of an object. They are essentially variables that store the unique values of each object.
Member functions are the functions used to manipulate these variables together. They are essentially functions that encapsulate the operations specific to that class. These data members and member functions define the properties and behaviour of the objects in a class.

Objects represent individual instances of a class, embodying the characteristics and behaviours defined within that class. An object is an instantiation of the class blueprint. Objects possess their own unique values for the attributes defined in the class.

C++ class and objects provide a structured way to organize data and behaviour, promoting code reuse, data encapsulation, and modular design. Classes and objects offer several advantages in C++ programming –

1. **Data Encapsulation –** Classes encapsulate data and behaviour, preventing unauthorized access and promoting data integrity.
2. **Code Reusability –** Classes promote code reuse by allowing us to define common characteristics and behaviours once and then create multiple objects with those characteristics and behaviours.
3. **Modular Design –** Classes contribute to modular design by breaking down complex programs into smaller, manageable units.

Access Specifiers / Access Modifiers (specify the access rights for the class members) –

1. **public –** Members are accessible from outside the class by other classes and functions. They can be accessed using the direct member access operator (.) with the object of that class.
2. **protected –** Members cannot be accessed outside the class. Although, class members declared as protected can be accessed by any subclass (derived class) of that class.
3. **private –** Members cannot be accessed / viewed from outside of the class. They can be accessed only by the member functions inside the class.

By default, access to members of a C++ class is private.

Syntax for Declaration and Definition of Class and Objects –

```
class class_name
{
        Access_specifier: // private, public or protected
                data_members;

                member_functions( )
                {
                        // code to be executed
                }
};
```

ENCAPSULATION

One of the core fundamental concepts of object-oriented programming (OOP) in C++ is encapsulation, which is the grouping of methods and data members into a single class. By limiting access to private data members from outside the class, encapsulation aids in data hiding. It provides the following advantages –

1. **Data Protection** – Encapsulation restricts direct access to class members, preventing unauthorized modification or misuse of data.
2. **Improved Code Readability and Maintainability** – By organizing data and functions together, encapsulation makes code more understandable and easier to manage.
3. **Modular Design** – Encapsulation promotes modularity by dividing code into distinct, self-contained units, enhancing code reusability and reducing complexity.
4. **Security Enhancement** – Encapsulation helps protect sensitive data by restricting access to authorized methods, preventing unauthorized modifications or data breaches.

Encapsulation plays a crucial role in OOP by promoting data integrity, improving code organization, and enhancing security. It is an essential concept for writing well-structured, maintainable, and secure C++ code.

Inheritance

In C++, **inheritance** is a fundamental concept in object-oriented programming (OOP) that allows new classes to be created from existing classes. The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The derived class is said to be inherited from the base class. The derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class.

Inheritance promotes code reusability by allowing programmers to leverage the functionality of existing classes when creating new ones. This reduces the need to rewrite code from scratch, saving time and effort. Inheritance also promotes code organization by grouping related classes together based on their shared properties and behaviours. Inheritance enables the creation of hierarchical class structures that makes code more understandable, easier to manage and reflect real – world relationships between entities.

Base Class Visibility	**Derived Class Visibility**		
	Public Derivation	**Private Derivation**	**Protected Derivation**
private	Not inherited	Not inherited	Not inherited
protected	protected	private	protected
public	public	private	protected

Syntax –

```
class base_class_name
{
        // access specifiers
        // data members and member functions
};

class derived_class_name : access_specifier base_class_name
{
        // access specifiers
        // data members and member functions of the derived class
};
void main( )
{
        // create object of derived class to access data members and member functions // of
        both the base class and derived class
}
```

TYPES OF INHERITANCE

1. **Single Inheritance –** In single inheritance, a derived class inherits from only one base class. This ensures a clear parent – child relationship and avoiding potential conflicts.

2. **Multiple Inheritance –** In multiple inheritance, a derived class inherits from multiple base classes. This allows the derived class to combine the features of multiple base classes, creating more complex and specialized classes. However, multiple inheritance can introduce complexity and potential conflicts, making it less commonly used.

3. **Multilevel Inheritance** – Multilevel inheritance is a type of inheritance in C++ where a class inherits from another class, which in turn inherits from another class. This creates a hierarchical structure of classes, where each level inherits the properties and methods of the classes below it.

4. **Hierarchical Inheritance –** Hierarchical inheritance involves multiple derived classes inheriting from a common base class, forming a tree – like structure. This reflects a parent – child relationship between classes, with derived classes becoming increasingly specialized as they inherit from their ancestors.

5. **Hybrid Inheritance –** Hybrid inheritance combines multiple inheritance and hierarchical inheritance, creating complex relationships between classes. While it provides flexibility, it can also introduce significant complexity.

Polymorphism, Virtual Function and Friend Function

AIM:

To explore and demonstrate the concepts of polymorphism, virtual function and friend function in C++ programming language.

INTRODUCTION:

C++ is an object-oriented, multi-paradigm language that was developed as an extension of the C programming language and was designed for system and application programming. It supports procedural, functional, and generic programming styles.

Polymorphism is a key concept in object-oriented programming (OOP) that allows a single entity (function or object) to exhibit different behaviours in different contexts. Polymorphism offers several benefits in C++ programming –

1. **Code Reusability –** Polymorphism allows you to write a single function or class that can handle different data types or perform different operations, reducing code duplication and improving code maintainability.

2. **Flexibility –** Polymorphism makes your code more flexible by allowing you to change the behaviour of functions or objects at runtime, based on the specific context.

TYPES OF POLYMORPHISM

1. **Compile – Time Polymorphism –**
   Compile – Time Polymorphism can be achieved by function overloading, which is the ability to define multiple functions with the same name but different parameter lists. The compiler determines which function to call based on the specific types of arguments passed to it. This allows you to define a single function name that can handle different types of data or perform different operations.

2. **Runtime Polymorphism –**
   Runtime polymorphism, also known as late binding, enables objects of different classes to respond to the same method call in different ways.

   (a) Function Overriding –
   Function overriding allows a derived class to redefine a function from its base class with the same name and signature (return type and parameter list). This allows for different behaviours for the same function call, depending on the object type.

   (b) Virtual Function –
   A virtual function is a member function in a base class that is declared with the **virtual** keyword. This keyword informs the compiler that the implementation of a function should be determined at runtime based on the actual object type, rather than being statically bound at compile time.

   When a virtual function is called through a base class pointer or reference, the compiler determines the actual function implementation to invoke based on the dynamic type of the object being pointed or referenced. This dynamic dispatch mechanism is what enables runtime polymorphism.

Runtime polymorphism allows for more dynamic behaviour but is slower compared to compile – time polymorphism.

FRIEND FUNCTION

A friend function is a function that can access the private and protected members of a class even though it is not a member of that class. Within the class, it is specified using the **friend** keyword. This gives the friend function access to the class's private and protected members, increasing code development efficiency and flexibility while maintaining data encapsulation.

Constructor and Destructor

CONSTRUCTOR

A constructor is a member function of a class that has the same name as the class name and does not have any return type, not even void. It is called whenever an instance of the class is created and can be defined with or without arguments.
A constructor aids to initialize the object of a class. It is used to allocate the memory to an object of the class. It can be defined manually with arguments or without arguments. There can be multiple constructors in a class. It can be overloaded but it cannot be inherited or virtual.

General Syntax of Constructor –

```
class class_name // class
{
        // access specifiers
        // data members and member functions

        class_name( ) //constructor
        {
                // code to be executed
        }
};
```

Types of Constructors –

1. **Default constructor –** This constructor is called automatically when an object of a class is created without any arguments. It is used to initialize the data members of the object to default values. The compiler will automatically generate a default constructor if not user – defined.

2. **Parameterized constructor –** This constructor takes one or more arguments and is used to initialize the data members of an object with specific values. You can define as many overloaded constructors as needed to customize initialization in various ways. Parameterized constructors are typically declared as public so that code outside the class definition or inheritance hierarchy can create objects of the class.

3. **Copy constructor –** This constructor creates a new object by initializing it with an existing object of the same class. It is typically used to pass objects by value as function arguments or return values.

DESTRUCTOR

Like a constructor, a Destructor is also a member function of a class that has the same name as the class name preceded by a tilde (~) operator. It helps to deallocate the memory of an object. It is called while the object of the class is freed or deleted. In a class, there is always a single destructor without any parameters so it cannot be overloaded. If a class is inherited by another class and both the classes have a destructor then the destructor of the child class is called first, followed by the destructor of the parent or base class.

General Syntax of Destructor –

```
class class_name // class
```

```
{
        // access specifiers
        // data members and member functions

        class_name( ) //constructor
        {
                // code to be executed
        }

        ~class_name( ) //destructor
        {
                //code to be executed
        }
};
```