UNIT – 01

INTRODUCTION TO PROGRAMMING

Programming is writing computer code to create a program, to solve a problem. Programs are created to implement algorithms. Algorithms can be represented as pseudocode or a flowchart, and programming is the translation of these into a computer program.

To tell a computer to do something, a program must be written to tell it exactly what to do and how to do it. If an algorithm has been designed, the computer program will follow this algorithm, step-by-step, which will tell the computer exactly what it should do.

PROGRAM DEVELOPMENT LIFE CYCLE

When we want to develop a program using any programming language, we follow a sequence of steps. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language. Generally, program development life cycle contains 6 phases, they are as follows....

- ♣ Problem Definition
- ♣ Problem Analysis
- ♣ Algorithm Development
- ♣ Coding & Documentation
- ♣ Testing & Debugging
- ♣ Maintenance

1. Problem Definition

In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirement, what should be the output of the problem solution? These are defined in this first phase of the program development life cycle.

2. Problem Analysis

In phase 2, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

3. Algorithm Development

During this phase, we develop a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means we write the solution in step by step statements.

4. Coding & Documentation

This phase uses a programming language to write or implement actual programming instructions for the steps defined in the previous phase. In this phase, we construct actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java etc.,

5. Testing & Debugging

During this phase, we check whether the code written in previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test that whether it is providing the desired output or not.

6. Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated again to make the enhancements. That means in this phase, the solution (program) is used by the end user. If the user encounters any problem or wants an enhancement, then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

Algorithm

Step1: Start.

Step 2: Declare : Variables n1,n2,sum.

Step 3: input : n1, n2(READ)

Step 4: Calculate: Sum = n1+n2.

Step 5: Display Sum.

Step 6: End(Stop)

Example :

```
#include<iostream>
using namespace std;
int main()
{
int n1,n2,sum;
cout<<"Enter n1 and n2 values"<<endl;
cin>>n1>>n2>>;
sum=n1+n2;
cout<<"Sum of Numbers is:"<<endl;
return 0;
}
```

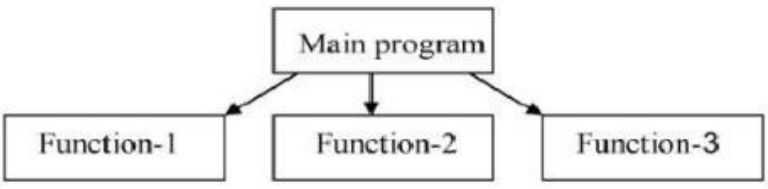
High Level Programming Approaches are broadly classified as

1)Procedure – Oriented Programming (POP) [Examples: COBOL, FORTRAN, C]

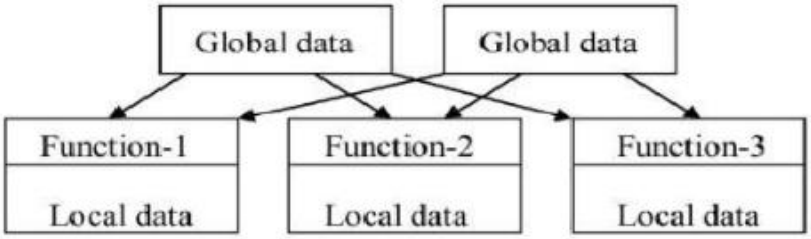
2)Object – Oriented Programming (OOPs) [Examples: JAVA, C++, C#]

Procedure – Oriented Programming (POP)

In this approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. POP basically consists of writing a list of instructions for the computer to follow and organizing these instruction into groups known as functions.



The Disadvantage of the POP are 1) Global data access, 2)It does not model real word problem very well. 3) No data Hiding.



Analysis of Procedure-oriented-Programming

- Concentration is on the development of functions, very little attention is given to the data that are being used by various functions.
- Data move freely between functions where there is chance for the intruders to hake the data.
- Critical applications should be designed in such a way that the data should be protected when they are transferred between functions.

Characteristics of procedure oriented programming

1. Emphasis is on doing things(algorithm)
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data
4. Data move openly around the system from function to function
5. Function transforms data from one form to another.
6. Employs top-down approach in program design

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING PARADIGM

• The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach mainly providing data hiding feature.
• Object oriented programming is the principle of design and development of programs using modular approach. Object oriented programming approach provides advantages in creation and development of software for real life applications.
• Some of the object oriented programming languages are C++, Java, C# and so on.
• The object oriented programming methods use data as the main element in the program. The data is tied to the function that operates on the data and the other functions cannot modify the data tied to a given function. Thus in object oriented programming; a problem is decomposed into a number of components called objects.
• Object is the basic unit of OOP. To design object oriented Model, first a set of classes are defined. A class is a Template from which objects are created. The Template of a class specifies the data, member functions and their attributes.

OOPs: " It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand".

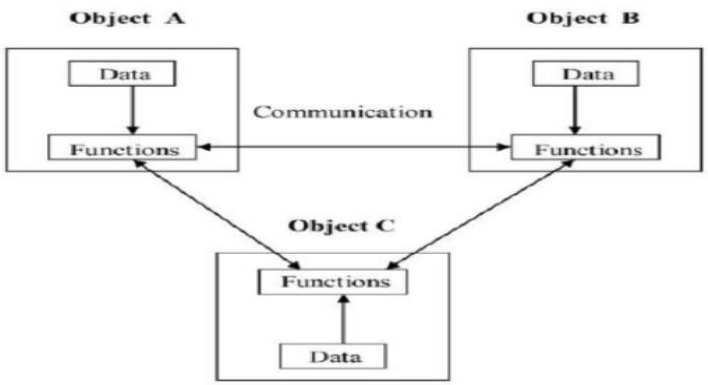
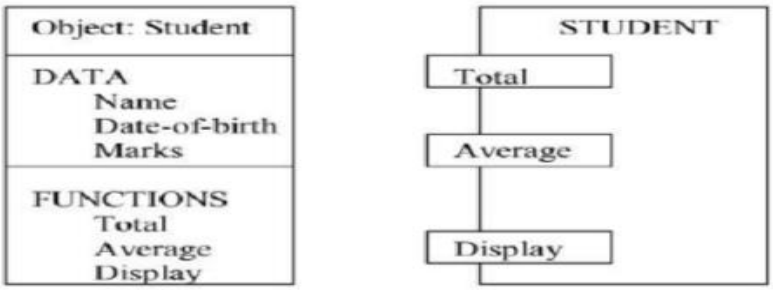


Fig: Organization of data and function in oop

Example:

Object: Student



Features of OOPs

1) Emphasis is on data rather than procedure.
2) Programs are divide into what are known as objects.
3) Data Structures are designed such that they characterize the objects.
4) Functions that operate on the data of an object are ties=d together in the data structure.
5) Data is hidden and cannot be accessed by external functions.
6) Objects may communicate with each other through functions.
7) New data and functions can be easily added whenever necessary.
8) Follows bottom – up approach in program design.

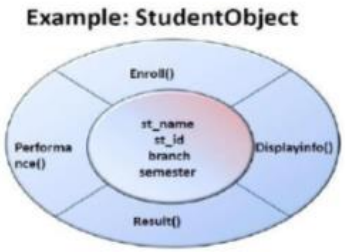
Basic Concepts of OOPs:

1)Objects. 2) Class. 3)Data Abstraction. 4)Data Encapsulation. 5)Inheritance. 6) Polymorphism.

7) Dynamic Binding. 8) Message Passing.

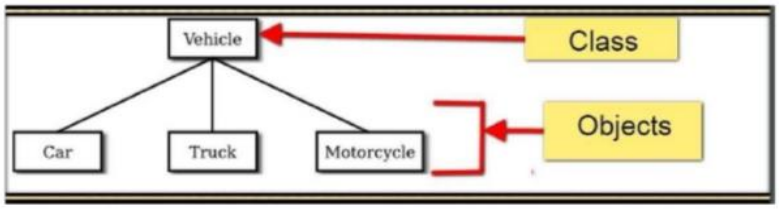
1)OBJECTS: Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.

• The fundamental idea behind object oriented approach is to combine both data and function into a single unit and these units are called objects. The term objects means a combination of data and program that represent some real word entity.



2)CLASS: A group of objects that share common properties for data part and some program part are collectively called as class.

• In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.



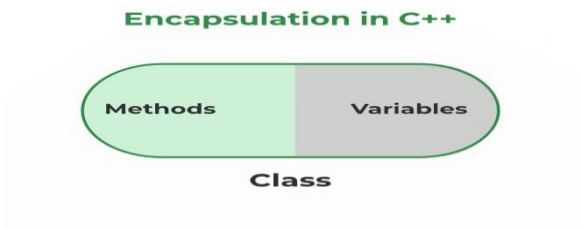
Difference between objects and class.

	Class	Object
1	Class is a container which collection of variables and methods.	object is a instance of class
2	No memory is allocated at the time of declaration	Sufficient memory space will be allocated for all the variables of class at the time of declaration.
3	One class definition should exist only once in the program.	For one class multiple objects can be created.

3)Data ABSTRACTION: Abstraction refers to the act of representing essential features without including the back ground details or explanations.

• Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes. Example of Mobile : The customer needs information only outer working of mobile not the inner working of mobile.

4) DATA ENCAPSULATION: The wrapping up of data and function into a single unit (called class) is known as encapsulation.The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the objects data and the program.



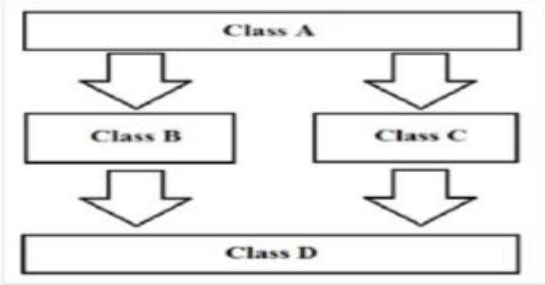
Example: In a company, there are different sections, consider account section , financial section handles all the financial process are done by this section. sales section handles all the sales related activities and keeps records of all sales.

5) INHERITENCE: Inheritance is the process by which objects of one class acquire the properties of another class. In the concept of inheritance provides the idea of reusability.

• This means that we can add additional features to an existing class without modifying it.

• This is possible by designing a new class will have the combined features of both the classes.

Example1 : Class A is Base Class, Class B and C are sub - derived class , Class D are derived class of B and C.



Example2: The Grand Parents are Base class, Parents are sub derived class, Childs are derived class.

6)POLYMORPHISM: Poly means Multiple and Morphing means Actions or forms.

• A language feature that allows a function or operator to be given more than one definition. The types of the arguments with which the function or operator is called determines which definition will be used. Overloading may be operator overloading or function overloading.

Example: A real life Example, a person who are at the same time can have different characteristics. A man at the same time is a Father, Husband, and an Employee. So the same person exhibits different behavior in different situations.(A single function can work differently in different situations.)

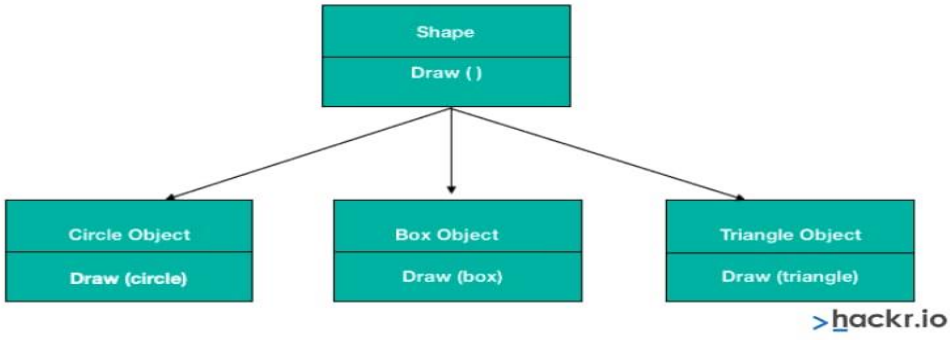
Polymorphism can be divide into 2 ways:

1)Compile -time polymorphism : Function call is bound to its definition.

i)Function Overloading. ii) Operator Overloading

2)Run -time polymorphism : Function call is bound to its definition during runtime.

i) Virtual Functions.



7)Dynamic Binding

Dynamic Binding means the code associated with a given procedure call is not known until the time of the call at run time. It is associated with a polymorphic reference depend upon the dynamic type of that reference.

It is the connection between the function declaration and function call.

8)Message Passing

An object oriented program consists of a set of objects that communicate with each other.

A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result.

Message passing involves specifying the name of the object, the name of the function (message) and information to be sent.

BENEFITS OF OOP:

1. Through inheritance redundant code can be eliminated and extend the use of existing classes.
2. Code reusability leads to saving of development time and higher productivity.
3. The principle of data hiding helps the programmer to build secure programs that can't be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist without any interference.
5. It is easy to partition the work in a project based on objects.
6. Object-oriented systems can be easily upgraded from small to large systems.
7. Message passing techniques for communication between objects makes the interface description with external systems much simpler.
8. Software complexity can be easily managed.

APPLICATIONS OF OOP

The most popular application of oops up to now, has been in the area of user interface design such as windows. There are hundreds of windowing systems developed using oops techniques.

1) **Real-time Systems:**The complexities inherent in real-time systems, such as embedded systems and industrial control, are effectively managed through OOP's structured approach.

2) **Simulation and Modeling:** Complex systems with varying specifications can be effectively modelled and simulated using OOP, allowing for better understanding and analysis.

3) Object oriented databases.: Object – oriented databases store and manage data in the form of objects, providing a more natural way to represent complex relationships.

4)Hyper text and Hyper media:OOPs principle is used in developing systems for managing and navigating hypertext documents and multimedia context.

5)Graphical User Interface (GUI) Development: The design and implementation of interactive and user-friendly GUIs are greatly simplified by using OOP concepts to represent and manage GUI elements.

6)Computer-Aided Design (CAD) and Manufacturing (CAM) Systems:

OOP is widely used in these fields to represent and manipulate design components, facilitating efficient design and production processes.

7)Office Automation Systems: Formal and informal electronic systems for information sharing and communication within and outside organizations benefit from OOP's ability to model and manage interconnected components.

8) AI and Expert Systems:

OOP is crucial in developing Artificial Intelligence applications and expert systems, particularly those based on machine learning, by enabling the modeling of complex problem domains.

9)Software Development: OOP facilitates the creation of large and complex software systems, including accounting systems, inventory management, and banking applications, by promoting organized and reusable code.

10) CIM / CAM / CAD system :
OOPs used in CAD(Computer Aided design) and CAM(Computer Aided manufacturing, CID(Computer integrated manufacturing to model and manipulate design elements.
And some other applications are **Web development and Mobile development.**

Difference between POP and OOPs.

POP	OOP
POP stands for Procedural Oriented Programming	OOP stands for Object Oriented Programming
In procedural programming, program is divided into small parts called functions	In object oriented programming, program is divided into small parts called objects.
Procedural programming follows a top down approach.	Object oriented programming follows a bottom up approach.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected, etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is less secure.	Object oriented programming provides data hiding so it is more secure.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on unreal world.	Object oriented programming is based on real world.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc.

Overview of C++:

C++ is an object oriented programming language. It was developed by **Bjarne Stroustrup in 1979** at Bell Laboratories in Murray Hill, New Jersey.
He initially called the new language **"C with Classes."** However, in 1983 the name was changed to C++. C++ is a superset of C. Stroustrup built C++ on the foundation of C, including all of C's features, attributes, and benefits. Most of the features that Stroustrup added to C were designed to support object- oriented programming.

Why Use C++?

C++ is one of the world's most popular programming languages. C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems. C++ is a cross-platform language that can be used to create high-performance applications.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs. C++ is portable and can be used to develop applications that can be adapted to multiple platforms. C++ is fun and easy to learn.

C++ Comments:

Comments can be used to explain C++ code, and to make it more readable. It can also be used to prevent execution when testing alternative code. Comments can be singled-lined or multi-lined.

1)Single-line Comments: Single-line comments start with two forward slashes (//).Any text between // and the end of the line is ignored by the compiler (will not be executed). This example uses a single-line comment before a line of code:

Example:// This is a comment
cout << "Hello World!";This example uses a single-line comment at the end of a line of code:

Example : cout << "Hello World!"; // This is a comment

2)C++ Multi-line Comments: Multi-line comments start with /* and ends with */. Any text between /* and */ will be ignored by the compiler:

Example

```
/* The code below will print the words Hello World!
to the screen, and it is amazing */
cout << "Hello World!";
```

History of C++:

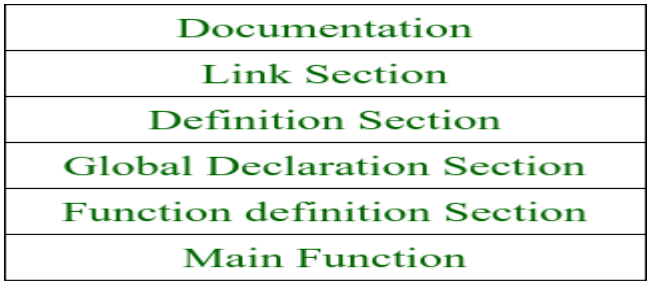
1)Until 1980, C programming was widely popular, and slowly people started realizing the drawbacks of this language, at the same time a new programming approach that was Object Oriented Programming.

2)The C++ programming language was created by Bjarne Stroustrup and his team at Bell Laboratories (AT&T, USA) to help implement simulation projects in an object-oriented and efficient way.

3)C++ is a superset of C because; any valid C program is valid C++ program too but not the vice versa is not true.

4)C++ can make use of existing C software libraries with major addition of “Class Construct”. This language was called “C with classes” and later in 1983, it was named “C++” by Rick Mascitii. As the name C++ implies, C++ was derived from the C programming language: ++ is the increment operator in C.

Structure of C++:



Skeleton of C Program

The C++ program is written using a specific template structure. The structure of the program written in C++ language is as follows:

1)Documentation Section:

- This section comes first and is used to document the logic of the program that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever written in the documentation section is the comment and is not compiled by the compiler.
- Documentation Section is optional since the program can execute without them. Below is the snippet of the same:

```
/*    This is a C++ program to find the area of circle*/
```

2)Linking Section:

The linking section contains two parts:

Header Files:

- Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc. that are already defined in the standard C++ library.
- In order to use such pre-defined elements in a program, an appropriate header must be included in the program.
- Standard headers are specified in a program through the pre-processor directive #include. In Figure, the iostream header is used. When the compiler processes the instruction **#include<iostream>,** it includes the contents of the stream in the program. This enables the programmer to use standard input, output, and error facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

#include<iostream>

Namespaces:

- A namespace permits grouping of various entities like classes, objects, functions, and various C++ tokens, etc. under a single name.
- Any user can create separate namespaces of its own and can use them in any other program.
- In the below snippets, **namespace std** contains declarations for cout, cin, endl, etc. statements.

```
using namespace std;
```

- Namespaces can be accessed in multiple ways:
    - using namespace std;

- using std :: cout;

3)Definition Section:

- It is used to declare some constants and assign them some value.
- In this section, anyone can define your own datatype using primitive data types.
- In **#define pi 3.14** is a compiler directive which tells the compiler the values of pi is defined as constant value.
- **typedef int INTEGER;** this statement tells the compiler that whenever you will encounter INTEGER replace it by int and as you have declared INTEGER as datatype you cannot use it as an identifier.

4)Global Declaration Section:

- Here, the variables and the class definitions which are going to be used in the program are declared to make them global.
- The scope of the variable declared in this section lasts until the entire program terminates.
- These variables are accessible within the user-defined functions also.

5)Function Declaration Section:

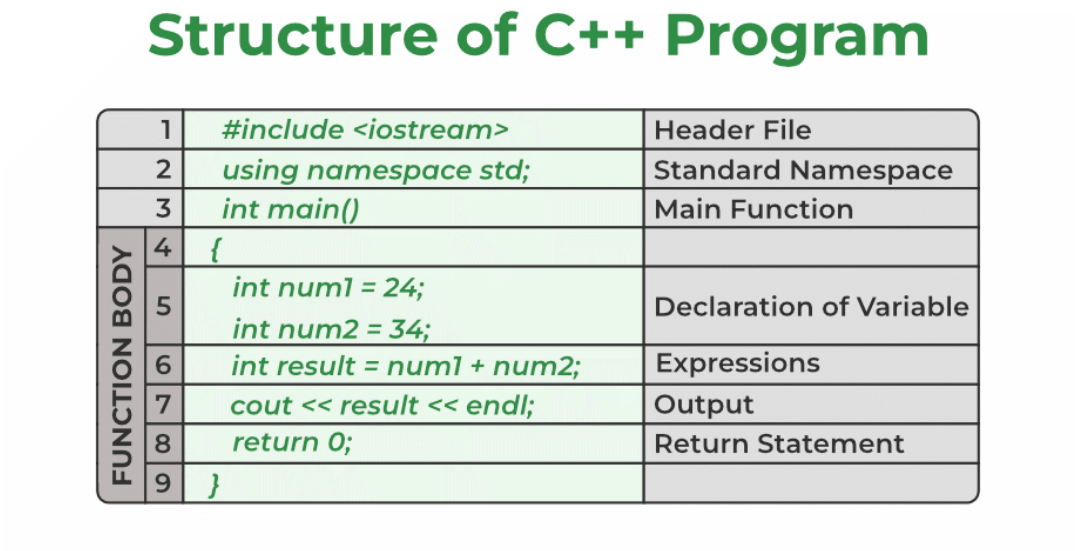
- It contains all the functions which our main functions need.
- Usually, this section contains the User-defined functions.
- This part of the program can be written after the main function but for this, write the function prototype in this section for the function which for you are going to write code after the main function.

6)Main Function:

- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces **{}** which encloses the body of the main function. Once all instructions from the main function are executed, control comes out of the main function and the program terminates and no further execution occur. **Example 1:**

```
#include <iostream.h>
using namespace std;
int main() {
  cout << "Hello World!";
  return 0;
}
```

Example of sum of two numbers program in C++.



Example:

//program to find the area of circle

```
#include<iostream>
#define PI 3.14
Int  main()
{
int r;
float area;
cout<<"Enter the radius:";
cin>>r;
area=PI*r*r;
cout<<"The area of a circle ="<<area;
return 0;
}
```

Input and output statements

The input output operations are done using library functions cin and cout objects of the class iostream the standard input and output library, we will able to interact with the user by printing message on the screen and getting the user's input from the keyboard stream is an object where a program can either insert/extract characters to/from it.

The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared.

Input Operators:

Input Operator ">>": The standard input device is usually the keyboard.

Input in C++ is done by using the "stream extraction" (>>) on the cin stream.

"cin" stands for "console input".

Example: int age;

cin>>age;

Output Operator:

Output Operator "<<": The standard output device is the screen (Monitor).

Outputting in C++ is done by using the object followed by the "stream insertion" (<<).

"cout" stands for console output

Example: cout<<"sum"; //prints sum

cout<<sum; //prints the content of the variable sum;

Tokens:

Smallest individual unit in a program is known as **tokens. Some of the Tokens are:-.**

1. Identifiers
2. Keywords
3. Constants/literals
4. Operators
5. Punctuators / Delimiters.

1) Identifiers: Identifiers is a name given to programming elements such as variables, functions, arrays, objects, classes etc…

The following are some valid identifiers: **Examples: Student , Reg101 , _ale2r3 , dos.**

some Invalid identifiers: Examples **: float, Total points, 1List, T_points, list_1, n_float.**

Rules to be followed while creating identifiers.

1)Identifiers are a sequence of characters which should begin with the alphabet either from A-Z

(Uppercase) or a-z (lowercase) or (underscore).

2)C++ treats uppercase and lowercase characters differently .(It was case sensitive)

No Special character is allowed except underscore(_).

3)Identifier should be single words [i.e blank spaces cannot be included in identifier].

4)Reserved Keywords should not be used as identifiers.[keywords such as if, int, float, break].

5)Identifiers should be of reasonable length.

2)Keywords: They are **predefined word** that gives special meaning to the complier. keywords(also known. as **reserved words**) have special meanings to the C++ compiler and are always written or typed in short(lower) cases. Keywords are words that the language uses for a special purpose, such as void, int, public, etc. It can't be used for a variable name or function name or any other identifiers.

Examples:

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

C++ keywords are case-sensitive and must be written in lowercase. Attempting to use a keyword as an identifier will result in a compilation error.

3)Constants/Literals: A Literals/constant are identifiers whose value does not change during program execution. Constants sometimes referred to as literal. They are essential for representing fixed or unchanging values, improving code readability, and ensuring data integrity.

A constant may be divide into following:

1)Integer Constant,2)Floating Constant, 3)Character Constant,4)String Constant

1)Integer Constant: An integer constant is a whole number, which can be either positive or negative,

They do not have fractional part exponents. We can specify integer constants in Decimal Integer Constant .Example: int a 120;//integer Constant

Examples Octal Integer Constant: int b=0374;//Octal Constant

Examples Hexadecimal Integer Constant: int c= OXABF;//Hexadecimal Constant

Examples Unsigned Constant: int d= 4328u://Unsigned value.

2)Floating Point Constant: Floating point constants are also called **as "real constants".** These values contain decimal points (.) and can contain exponents. They are used to represent values that will have a fractional part and can be represented in two forms (i. e fractional form and exponent form) We can specify Floating Point constants in:

float a= 23.46//equal to 23.46 x 10^0 =23.46 x1=23.46

float b=26.126.

3)Character constants: are specified as single character enclosed in pair of single quotation marks. For example : char ch='P';

There are certain characters used in C++ which represents character constants called as **escape sequence** which starts with **a back slash (\)** followed by a character.

Escape Sequence is a special string used to control output on the monitor and they are represented by a single character and hence occupy one byte.

Escape Sequence	Meaning	Escape Sequence	Meaning
\'	Single Quote	\"	Double Quote
\?	Question Mark	\\	Back Slash
\0	Null Character	\a	Audible Bell
\b	Backspace	\f	New Page
\n	New Line	\r	Carriage Return
\t	Horizontal Tab	\v	Vertical Tab
\nnn	Arbitrary octal value	\xnn	Arbitrary Hexa Value

4)String Constants: A string constant consists of zero none character enclosed by double marks (" "). Multiple character constants are called **string constants** and they are treated as an array of char. By default compiler adds a special character called the "Null Character" (10) at the end of the string to mark the end of the string.

Example: char str[25]= "Hello Adisesha";

This is actually represented as char str[25] ="Hello Adisesha\0" in the memory.

Example Boolean literals: true and false.

4) Operators: An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations Operators are used to perform operations on variables and values .C++ is rich in built-in operators and there are almost 45 different operators.

There are three group of operators:1)Unary Operator.2)Binary Operator, 3)Ternary Operator.

1)Unary operator: The unary operators operate on one operands. Example: increment operator(++) and decrement operator(- -).

2)Binary operator:

The binary operators operate on two operands. They are: Arithmetic operators, relational, logical operators, bitwise, assignment operators ,Logical operators.(+, -, *, /, %, &&, ||).

In C++, operators are symbols that instruct the compiler to perform specific operations on operands (variables or values). These operations can be mathematical, logical, relational, or bitwise, among others.

3)Ternary operator: The binary operators operate on three operands. Examples ?: (Conditional Operator).

5)Punctuators/Special Symbols: These are symbols that have specific syntactic or structural meaning in the language, us ed for grouping, separation, or specific operations (e.g., ; for statement termination, {} for code blocks, () for function calls).

Punctuators in C++ are special characters or symbols that have syntactic and semantic meaning to the compiler, helping to structure the code and define its meaning. While some punctuators can also function as operators, their primary role is to separate or delimit different parts of a program. Common punctuators in C++ include:

- Brackets [ ] : Used for array indexing and defining array sizes.
- Parentheses ( ): Used for function parameter lists, grouping expressions, and type casting.
- **Braces { }** : Used to define code blocks (e.g., function bodies, loops, conditional statements), and initializer lists.
- **Comma ,** Used as a separator in lists (e.g., function arguments, variable declarations) and in the comma operator.
- **Semicolon ;** Used to terminate statements and declarations.
- **Colon ::** Used in various contexts, such as in switch statements, base class specifiers, and bit-field declarations.
- **Asterisk *** Used as a pointer dereference operator and to declare pointer types.
- **Equal sign =** Used as the assignment operator and for initialization.
- **Pound sign #** Used for preprocessor directives (e.g., #include, #define).
- **Dot .** Used for member access in objects and structures.
- **Arrow ->** Used for member access through pointers to objects or structures.
- **Scope Resolution Operator ::** Used to specify a member of a namespace or class, or to access static members.

These punctuators are fundamental to C++ syntax, enabling the compiler to correctly interpret the structure and flow of the program.

Variables in C++:

Variables are containers for storing data values. In C++, a variable serves as a symbolic name assigned to a specific memory location. It acts as the fundamental unit of storage within a program. The value stored in a variable can be modified during the program's execution.

Operations performed on a variable directly affect the data stored at its corresponding memory location. A crucial rule in C++ is that all variables must be declared before they can be used.

Variables are fundamental to programming as they allow the storage and manipulation of data. Each variable has a type that determines the kind of data it can hold, such as integers, floating-point numbers, characters, etc.

In C++, there are different types of variables (defined with different keywords), for example:

1)int - stores integers (whole numbers). 2) double - stores floating point numbers, with decimals.
3) char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
4)string - stores text, such as "Hello World". String values are surrounded by double quotes .
5)bool - stores values with two states: true or false

Declaring Variables: The general syntax for declaring a single variable is:

Syntax : datatype variable_Name = value;

Where type is one of C++ types (such as int), and variable _Name is the name of the variable (such as x or myName). The equal sign is used to assign values to the variable.

Declaring Multiple Variables: The **general syntax** for declaring a multiple variable is:

datatype variable1, variable2, ..., variable N;

example: int x=4,y=7, z=9;

cout<<x<<y<<z;

In C++, variables can be declared anywhere in the program where they are required, unlike ANSI C where all variables had to be declared at the beginning. This flexibility enhances program readability by allowing declarations in the context of their use.

Reference Variables: A reference variable provides an alternative name, or alias, for an existing variable.

Syntax for declaring reference variable: datatype & reference_name = variable_name;

This syntax creates reference_name as an alias for variable_name, both referring to the same memory location. The provided text describes variables and their declaration in C++, along with the concept of reference variables.

Variable Naming Rules:

1)Allowed Characters:Variable names can consist of alphabets (both uppercase and lowercase), numbers, and the underscore character (_).

2)Starting Character: A variable name must start with either a letter of the alphabet or an underscore. It cannot begin with a number.

3)Case Sensitivity: Variable names are case-sensitive, meaning Arr and arr are treated as distinct variables. Whitespace and Special Characters: Variable names cannot contain whitespace or other special characters (e.g., #, $, %).

4)Keywords: C++ keywords (e.g., float, double, class) cannot be used as variable names.

Create a variable called myNum of type int and assign it the value 15: Example: int mynum=15;

```
cout<<mynum;
```

You can also declare a variable without assigning the value, and assign the value later:

```
int mynum;

mynum=15;

cout << mynum;
```

Note :that if you assign a new value to an existing variable, it will overwrite the previous value:

```
int age=20;

age=25;

cout << "I am " << age << " years old.";
```

Examples:

```
int myNum = 5;               // Integer (whole number without decimals)
double myFloatNum = 5.99;    // Floating point number(with decimals)
char myLetter = 'D';         // Character
string myText = "Hello";     // String (text)
bool myBoolean = true;  // Boolean(true or False).
```

Variables must be declared before they can be used. Declaration involves specifying the data type and the variable name. Initialization involves assigning an initial value to the variable during its declaration.

int age = 25; *// Declares an integer variable 'age' and initializes it to 25*
double salary; // Declares a double variable 'salary'

Purpose of Variables:

Data Storage: They provide a way to store and manage data used by the program.

- **Data Manipulation:** They allow for operations and calculations to be performed on stored data.
- **Code Readability:** Using descriptive variable names makes the code more understandable.
- **Memory Management:** They enable the efficient allocation and use of memory.

Key characteristics of C++ variables:

- **Name (Identifier):**Each variable must have a unique name (identifier) to distinguish it from other variables. This name is used to refer to the variable and its stored value.
- **Data Type:**Every variable in C++ must be declared with a specific data type (e.g., int, float, char, bool). The data type determines the kind of values the variable can store (e.g., whole numbers, decimal numbers, characters, true/false values) and the amount of memory allocated for it.

- **Value:**A variable holds a value of its declared data type. This value can be assigned during declaration (initialization) or changed later in the program.**Lifetime:** The lifetime of a variable refers to the period during which it exists in memory. This is typically tied to its scope

Scope of variables:

The scope of a variable defines the region of the program where it can be accessed. Variables can **have**

1)local scope **(accessible only within a specific block or function): Declared inside a function or block.**

Example :

```
include <iostream>
using namespace std;
int main()
{
   int i=10;
   if(i<20)       // if condition scope starts
   {
      int n=100;   // Local variable declared and initialized
   }            // if condition scope ends
   cout << n;      // Compile time error, n not available here
}
```

2)Global scope **(accessible from anywhere in the program): Declared outside any function, accessible throughout the entire program.**

For example: Only declared, not initialized

```
include <iostream>
using namespace std;
int x;              // Global variable declared
int main()
{
   x=10;              // Initialized once
   cout <<"first value of x = "<< x;
   x=20;              // Initialized again
   cout <<"Initialized again with value = "<< x;
}
```

3)Static local variable**: Can be local or member variables, retaining their value across function calls or class instances.** The scope Limited to the function where they are defined.

Lifetime: Retains its value between function calls and exists until the program ends.

Example:

```
#include<iostream>
using namespace std;
int main()
{
void example() {
    static int staticVar = 0; // Static local variable
    staticVar++;
    std::cout << staticVar;
}
}
```

Static Global Variables

Definition: Declared outside all functions with the static keyword.

Scope: Limited to the file where they are defined (file scope).

Lifetime: Exists throughout the program's execution.

Example:

```
static int staticGlobalVar = 30; // Static global variable
void example() {
    std::cout << staticGlobalVar;
}
```

Omitting Namespace: You might see some C++ programs that runs without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

Example

```
#include <iostream>

int main() {
  std::cout << "Hello World!";
  return 0;
}
```

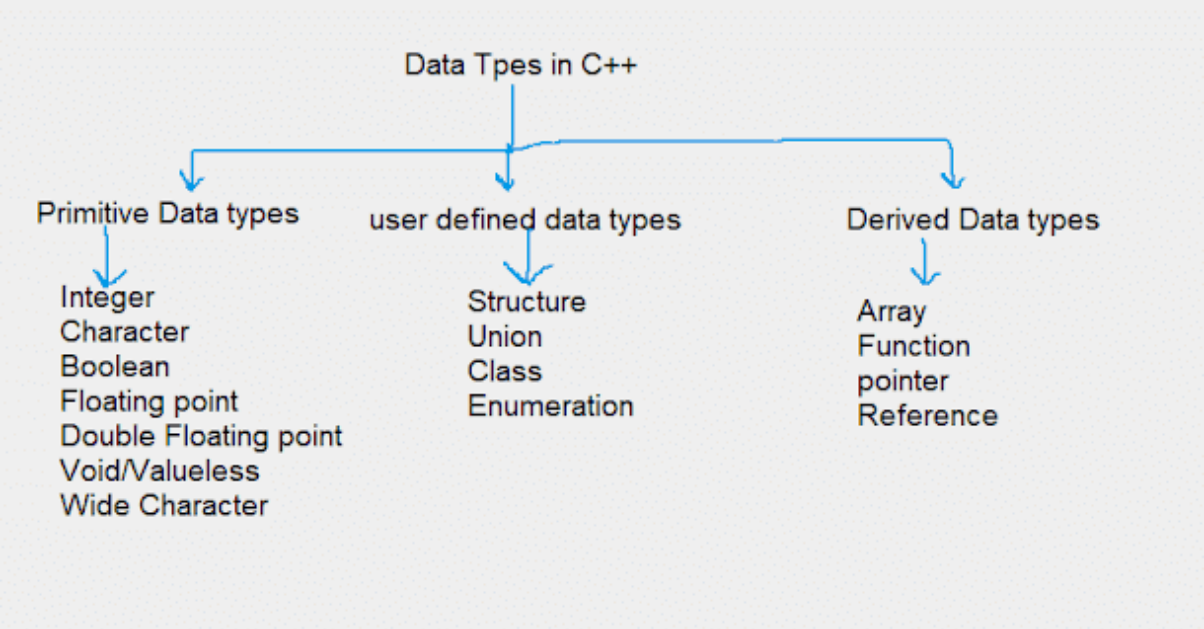
Data types in C++.

The data type specifies the size and type of information the variable will store. C++ data types categorize the kind of values a variable can hold and the operations that can be performed on them. They are broadly classified into three main categories:

1)Built -in data type(Primitive Data type).

2)Derived data type.

3)User – defined data type.



1)**Primitive (Built-in) Data Types:** These are fundamental data types predefined by the C++ language.

- **Integer Types:** Store whole numbers (positive, negative, or zero). Examples: int, short, long, long int..
- **Character Types:** Store single characters. Examples: char, wchar_t (for wide characters).
- **Boolean Type:** Stores logical values true or false. Example: bool.
- **Floating-Point Types:** Store numbers with decimal points. Examples: float (single precision), double (double precision), long double.
- **Long double:** Stores extends precision floating point numbers.(8bytes).
- **Void Type:** Represents the absence of a type, typically used for functions that do not return a value or for generic pointers. Example: void.
- **Short int:** Typically smaller than int.
- **Long int:** Typically larger than int.
- **Wchar_t:** Stores wide characters, used for extended characters sets.

Built-in Data types in C++ using sign & size qualifiers. **Size & range of C++ basic data types:**

TYPE	BYTES	RANGE
Char	1	-128 to127
unsigned char	1	0 to 255
signed char	1	-128 to 127
Int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E -308 to 1.7E +308
long double	10	3.4E-4932 to 1.1E+ 4932

2) Derived Data Types: These are constructed from or derived from the primitive data types.

- **Arrays:** Collections of elements of the same data type stored in contiguous memory locations.

  Example: int num[5]={1,2,3,4,5};

- **Pointers:** Variables that store memory addresses of other variables.(It was denoted by * and &(Address)).

  Example: int *ptr=&age;

- **References:** Aliases or alternative names for existing variables.
- **Syntax:** datatype &reference _name=variable_name;

  Example1: int &ref=age;

  Example2:

  float total=1500;

  float &sum=total;

- **Functions:** Blocks of code designed to perform a specific task.
  Example: int sum(int, int); // Function type returing int and taking 2 int parameters.

3) User-Defined Data Types: These are defined by the programmer to create custom data structures.

1)Structures (struct): A Collections of variables of different data types under a single name**.**
Example:

Structures uses struct Keyword. A structure groups multiple variables (of different types) under one name.

Example:

```
struct Student {
   int id;
   char name[50];
   float marks;
   double height;
};
```

2)Classes (class): Blueprints for creating objects, encapsulating data (member variables) and functions (member methods). Classes are an advanced version of structures with added features like access specifiers (public, private, protected) and member functions.

Example:

```
class Car {
public:
   string brand;
   int year;
   void display() {
      cout << "Brand: " << brand << ", Year: " << year << endl;
   }
};
```

3)Unions (union): It was a special class type allow different data types to share the same memory location. A union is similar to a structure but shares memory among its members, meaning only one member can hold a value at a time.

Example:

```
union Data {
   int intvalue;
   float floatvalue;
   char charvaluestr[20];
};
```

4)Enumerations: define a set of named integral constants, improving code readability**.**

Example: enum Color { RED, GREEN, BLUE };

5)Typedef (typedef): These allow creates new names (aliases) for existing data types.
.Example:

```
typedef unsigned int uint;
```

using uint = unsigned int;

In C++, user-defined data types allow programmers to create custom types tailored to their specific needs. These types extend the basic data types provided by the language. These user-defined types enhance flexibility and make code more modular and easier to maintain. Special Modifiers[Modifiers like signed, unsigned, short, and long can adjust the size or range of data types].

Operators in C++:

Operators: An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations Operators are used to perform operations on variables and values .C++ is rich in built-in operators and there are almost 45 different operators. Operators in C++ are can be divided into the following classes: There are three group of operators: **1)Unary operator**: The unary operators operate on one operands.

Unary operators:

Unary operations have only one operand i.e they use only one variable.
The following are the list of unary operators:

Operator	Name
!	Logical NOT. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
&	Address-of. Used to give the address of the operand.
~	One's complement. Converts 1 to 0 and 0 to 1.
*	Pointer dereference. Used along with the operand to represent the pointer data type.
+	Unary plus. Used to represent a signed positive operand.
++	Increment. Used to increment an operand by 1.
–	Unary negation. Used to represent a signed negative operand.
– –	Decrement. Used to decrement an operand by 1.

Increment:

Increment and decrement operators:

++ is used to increment the value of the variable by 1.
-- is used to decrement the value of the operator by 1.

If the ++/-- operator is to the left of the variable then, it is called pre-increment/decrement operator and if ++/-- operator is to the right of the variable then, it is called post-increment/decrement operator.

Eg for increment and decrement:
Consider the following statements in a program:

```
Int a= 5;
  b =++a;
```

b =a+1=6

```
Int a= 5;
  b =a--;
```

b =a-1 =4

Example: int a=5;

b=a++; (Post increment) // b=a+1;

b=5+1=6;

Example of pre increment:
int a=5;

b=++a; (Pre increment) // b=a+1;

b=5+1=6;

Decrement Operator: It was used to decrement the value of the variable by 1.

Example of post increment:
int a=5;

b= a - - ; (Post decrement) // b=a-1;

b=5-1=4;

Example of pre increment:
int a=5;

b= - -a; (Pre decrement) // b=a-1;

b=5-1=4;

2)Binary Operator: The binary operators operate on two operands. They are: Arithmetic operators, relational, logical operators, bitwise, assignment operators Special operator.(+, -, *, /, %, &&, ||, sizeof , comma).

1)Arithmetic Operator: used for performing simple arithmetic operations such as addition, subtraction, multiplication, division, modulus.

❖ **Arithmetic operators**: used for performing simple arithmetic operations.

Operator	Description	Example(a=10, b=20)
+	Adds two operand	a + b = 30
-	Subtracts second operand from the first	a - b = -10
*	Multiply both operand	a * b = 200
/	Divide numerator by denominators	b / a = 2
%	Modulus operators and remainder of after an integer division	b % a = 0

2)Relational Operator: used for comparing the values of operands. The return value of a comparison is either true (1) or false (0).

Let us assume that variable a=5and variable b= I0 then:

Operator	Description	Example
==	Checks if the value of two operands is equal or not,then condition becomes true.	(a == b) is false.
==	Checks if the value of two operands is equal or not,then condition becomes true.(a == b) is false.	(a == b) is false.
>	Checks if the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is false.
<	Checks if the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is false.
<=	Checks if the value of left operand is less than or equal to the value of right operand, then condition becomes true.	a <= b) is true.

3)Logical Operators: It was used for performing logical operations,Logical AND(&&),Logical not(!),Logical OR(||).It is used to combine conditional experssions results in Boolean.

❖ **Logical operators:** used for performing logical operations

Let us assume that variable a=0 and variable b=1 then:

Operator	Description	Example
&&	Called as Logical AND operator. If both the operands are non-zero then condition becomes true.	(a&& b) is false.
||	Called as Logical OR Operator. If any of the two operands is non-zero then condition becomes true.	(a || b) is true.
!	Called as Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(a && b) is true.!(a) is true.!(b) is false.

4)Bitwise operator :It works on bits and performs bit by bit operation.It performs operations on individuals bits of integer operands.

- **Bitwise operators**: Bitwise operator works on bits and performs bit by bit operation. Truth Table:

&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement

a	b	a&b (Bitwise and)	a | b (Bitwise or)	a ^ b (Bitwise xor)
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Example:
a = 00111100
b = 00001101

a&b = 0000 1100 (bitwise and).
a|b=00111101 (bitwise or).
a^b = 0011 0001 (bitwise xor).
--a = 1100 0011 (bitwise not)

C++ Shorthands.

C++ offers special shorthands that simplify the coding of a certain type of assignment statements.

Ex: a=a+10; can be written as a+=10;

The general form of C++ shorthand is
Variable operator=expression.

Following are some example of C++ shorthands:

x- =10	Equivalent to	x=x-10;
x*=5	Equivalent to	x=x*5;

x/=5	Equivalent to	x=x/5;
x%=z	Equivalent to	x=x%z;

5)Assignment operator: It was used to assign values to variables denoted by symbol (=).

Compound Assignment are +=,-=,*=,/=,%=,&=,|=,^=,<<=,>>=.

❖ **Assignment operators:** Assignment operators are used to assign values to variables

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	c = a + b will assign value o a + b into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
- =	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	c - = a is equivalent to c = c – a
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a

Special operators: Some of the special operator are sizeof, comma, Adressof, reference operator.

Special operators:

Operator	Description
sizeof	sizeof operator returns the size of a variable. For example sizeof(a), where a is integer, will return 2.
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
&	Pointer operator & returns the address of an variable. For example &a; will give actual address of the variable a.
*	Pointer operator* is pointer to a variable. For example *a; will pointer to a variable a.

3)Ternary operator(Conditional) Operator:

The operator acts on three operands is called ternary operator. It was also called as **conditional operator.It was denoted by ?:**

Syntax: is:(condition)? experssion1:expression2;

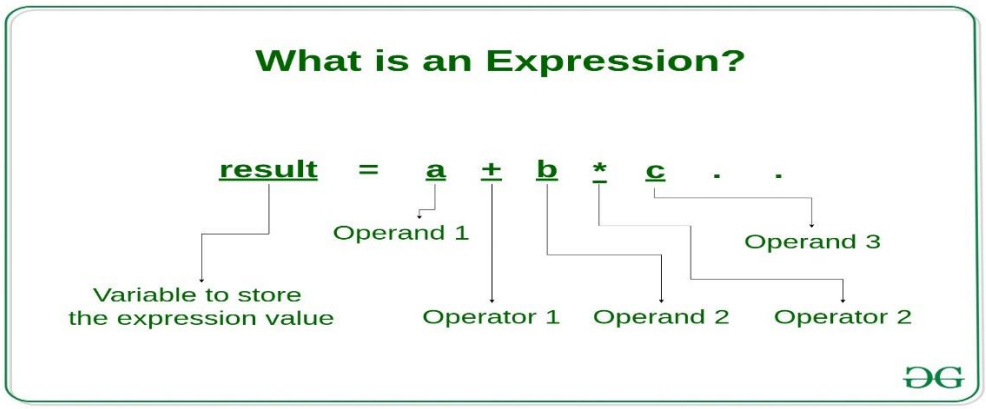
If the condition is TRUE, then expression 1 is evaluated. If the condition is false then the expression 2 is evaluated.

int a=5,b=4;

result=(a<b)? b:a;

Expression:-

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value



Expressions can also have side effects such as modifying a value of a variable. Some of the expression are:-

Constant expressions: Constant Expressions consists of only constant values. A constant value is one that doesn't change . It can be evaluated at compile time. Example :const int x=5+3.

- **Integral expressions**: Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. **Examples**:
  int a=10,b=3;
  int result=a/b; //result will be 3.
  where a and b are integer variables.
- **Floating expressions**: Float Expressions are which produce floating point results after implementing all the automatic and explicit type conversions. **Examples**:
  ```
  float x=10.05, y=3.0;

  float result=x/y; // result in 3.3333
  ```
  where x and y are floating point variables.
- **Relational expressions**: Relational Expressions yield results of type bool which takes a value true or false. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions. **Examples**:
  ```
  int a=5,b=10;

  bool is_greater=(a>b);       //is_greater will be false.
  ```
- **Logical expressions(Boolean)**: Logical Expressions combine two or more relational expressions and produces bool type results. **Examples**:
  ```
  x > y && x == 10, x == 10 || y == 5
  ```
- **Pointer expressions**: Pointer Expressions produce address values.
  These involve pointers and operators like * (dereference) and & (address-of).**Examples**:
  ```
  &x, ptr, ptr++
  ```

where x is a variable and ptr is a pointer.

- **Bitwise expressions**: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. **Examples:**

```
int x=5;

int shifted=x<<1;              //answer will be 10[1010 in binary form].
```

shifts one bit position to left.

```
y >> 1
```

shifts one bit position to right. Shift operators are often used for multiplication and division by powers of two.

Bitwise operator are used to manipulated data at bit level. They are basically shifting of bits.

- Arithmetic Expressions: This expressions use arithmetic operators(+,-,*,/,%) to perform operations such as mathematical computations.Example: int sum=10+5;
- Function call Expressions:These are expression involve calling a function which typically returns a value.

  Example: int result=myfunction(arg1,arg2);

  Example: sqrt(16), max(a, b)

Example

```
#include <iostream>
using namespace std;

int main() {
    int a = 10, b = 20;
    int result = (a + b) * 2;  // Arithmetic Expression
    bool isGreater = a > b;   // Relational Expression
    int maxVal = (a > b) ? a : b;  // Conditional Expression

    cout << "Result: " << result << endl;
    cout << "Is a greater than b? " << isGreater << endl;
    cout << "Maximum value: " << maxVal << endl;

    return 0;
}
```

Expressions are versatile and essential for implementing logic and computations in C++.

Operator precedence in C++.

- An expression is a combination of operator and operand.
- The operators would be arithmetic, relational, and logical operators.
- If the expression contains multiple operators, the order in which operations carried out is called the precedence of operators. It is also called as priority or hierarchy.
- The Operators with highest precedence appear at the top of the table and those with the lowest appear at the bottom.
- **Scope resolution operator has highest precedence [First].denoted by ::**

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to or not equal to	left to right
&&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	comma operator	left to right

Expression Evaluation

The evaluation of expressions in C++ follows specific rules:

1)Operator Precedence: Determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence.

2)Associativity: Defines the order of evaluation for operators of the same precedence. Operators can be left-associative (evaluated left-to-right) or right-associative (evaluated right-to-left).

3)Type Conversion: During the evaluation of expressions, implicit type conversions may occur to match operand types

4)Notes on Operator Precedence and Associativity .Example: c*a+b; multiplication (*) has higher precedence than addition (+), so in the expression c is evaluated first, then the result is added to a.

Associativity: When operators of the same precedence level appear in an expression, associativity determines the order of evaluation. For most operators, associativity is left-to-right, meaning expressions are evaluated from left to right. For example, in a - b - c, the subtraction is performed left-to-right. Some operators, like the assignment and conditional operators, have right-to-left associativity
Parentheses: Parentheses () can be used to explicitly specify the order of evaluation, overriding the default precedence and associativity rules. For example, in the expression (3 + 5) * 2, the addition is performed first due to the parentheses. Example:

```
int a = 5, b = 10, c = 15;
 int result = a + b * c; //
result1=5 + (10 * 15) = 155 ;
int result2 = (a + b) * c; //
 result2 = (5 + 10) * 15 = 225 ;
```

Right to left Associativity : Same operators like assignment operators (=,+=,-=,*= Etc..) and conditional operator(?:), are right associative.
Example: In a=b=c, the assignment b=c is evaluated first(assigning c value to b) and then the result of that assignment(The value assigned to b) is assigned to a.
() has highest precedence. Parenthesis can be explicitly override the default operator precedence and enforces a specific order of evaluation. Operations within parenthesis are evaluated.
[REFER MORE EXAMPLES of PRECEDENCE in notes]

Type Conversion:

In C++, type conversion refers to converting a value from one data type to another. It can be categorized into two types:

1)Implicit Type Conversion (Type Coercion): Performed automatically by the compiler. Happens when operations involve mixed data types, and the compiler converts one type to another to prevent data loss or ensure compatibility.

Implicit conversion is safer but limited to compatible types.[OR]

They are automatically performed when a value is copied to a compatible type. Implicit Type Conversion Also known as **'automatic type conversion'**. Done by the compiler on its own, without any external trigger from the user. Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data. All the data types of the variables are upgraded to the data type of the variable with largest

Example1:

```
short a=2000;
int b; b=a;  // 'a' is implicitly converted to int.
```

Example2:

```
int a = 5;
double b = a;  // 'a' is implicitly converted to double.
```

I)If you pass an argument to a function that expects a different data type.

II)Operations performed on values of different datatypes.

III)Assigning a value of one datatype to a variable of another datatype.

2. Explicit Type Conversion (Type Casting): Done manually by the programmer.

Used when you want to forcefully convert a value from one type to another.

Explicit Conversion: Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion. This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

Methods of explicit type casting:

1)C-style Cast:This method is inherited by C++ from C. The conversion is done by explicitly defining the required type in front of the expression in parenthesis. This can be also known as forceful conversion.

Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax: (type) expression; where type indicates the data type to which the final result is converted.

Example

```
short a 2000, int b;
b=(int) a;          //c-like cast notation data type.
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
```

```
double x=1.2;
int sum=(int)x+1; ;  // Explicitly cast double to int

cout<<sum;
return 0;
}
```

Output: 2

2)Function-style Cast:

```
double x = 5.5;
int y = int(x);  // Function-style cast
```

3)C++ Named Casts:C++ introduces its own typecasting method using cast operators. Cast operator is unary operator which one datatype to be converted into another datatype.

Conversion using Cast operator: A Cast operator is an unary operator which forces one data type to be converted into another data type.This is done to take advantage of certain features of type hierarchies or type representations.

Four types of casting are:

i)static_cast: For standard conversions.

ii)dynamic_cast: For polymorphic types.

iii)const_cast: To add/remove const qualifier.

iv)reinterpret_cast: For low-level reinterpretation of data.

```
#include<iostream>
using namespace std;
int main()
{
double x = 5.5;
int y = static_cast<int>(x);  // Safer and more readable
cout<<sum;
return 0;
}
```

Key Points:

Explicit conversion gives more control but should be used cautiously to avoid undefined behaviour or data loss.

Risks of type conversions:

1)Data loss that occurs when converting from a larger type to smaller type(Ex: int to char).

2)Undefined behaviour: That happens when casting pointers between unrelated types and reference them.

3)Memory misalignment: Casting pointers to types with stricter alignment can be crashes.

Explicit conversion Example:

```
#include<iostream>
using namespace std;
int main()
```

```
{
int i=10;
char c='A';              // value of A is 65.
cout<<(int ) c<<endl;  //printing c after manually converting it.
int sum=i+c;             // adding i and c.
cout<<sum;               // Printing sum.
return 0;
}
```

Output :

65

75

Implicit conversion Example:

```
#include<iostream>
using namespace std;
int main()
{
int i=10;
char c='a';              // value of a is 97.
i=i+c;                    //Adding i and c.
float f=i+1.0            // x is implicit converted to float.
cout<< "i="<<i<<endl;          //printing i after manually converting it.
cout<< "c="<<c<<endl;          //printing c after manually converting it.
cout<< "f="<<f<<endl; //printing c after manually converting it.
return 0;
}
```

OUTPUT:

i=107

c=a

f=108

Storage Classes. : In C++, storage classes define the scope (visibility), lifetime, and linkage of variables or functions within a program. They determine how and where variables are stored, their default initial values, and how they are linked when declared in different translation units. Understanding storage classes is crucial for managing resources efficiently and ensuring proper program behaviour. Types of Storage Classes

1. auto
2. register
3. static
4. extern
5. mutable

1)auto : The auto keyword in modern C++ is primarily used for type inference, allowing the compiler to automatically deduce the type of a variable from its initializer. However, historically in C and early C++, auto was a storage class specifier indicating automatic storage duration. By default, local variables are auto. It was a default storage class for local variables declared within block or function. They are created when the block or function is entered and destroyed when it's exited. They are typically stored on the stack, the "auto" keyword is rarely used directly because local variables are considered "auto" by default. Eg: auto int month;

2)register: The "register" storage class is used to instruct the compiler to store a variable in a register, which is a tiny, fast-access storage space within the CPU. However, the compiler is free to ignore this request, and modern compilers often optimize register usage automatically. Variables with the "register" storage class also have automatic storage duration. Eg: register int miles;

3)static: Variables declared with the "static" storage class have static storage duration, meaning they are allocated memory for the entire program's duration. They are created when the program starts and destroyed when it terminates. The "static" variables have local scope within a block or function, but their values persist across function calls. They are typically stored in a separate data segment of memory. Eg: static int count = 10;

4)extern: The "extern" storage class is used to declare variables that are defined in other source files. It extends the visibility of a variable to the entire program. "extern" variables have global scope, and they are often used to share variables across multiple source files. Eg: extern int x;

5)mutable: The mutable specifier applies only to class objects. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function. Eg: mutable int y;

These storage classes help programmers to control the memory usage and lifecycle of variables, which is crucial for managing resources efficiently and writing maintainable code.

Properties of storage classes:

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

Why storage classes are used in C++?

In C++, storage classes are used to regulate how variables are saved in memory, how long they exist, and where they may be accessed inside a program. C++ is essential for managing memory resources, improving efficiency, and assuring variable scope and lifespan. Here are some key reasons why storage classes are used in C++

Memory Management: Different storage classes allow programmers to allocate memory for variables in various ways. For example, "auto" variables are typically allocated on the stack with automatic memory management, while "static" variables are allocated in a separate data segment with

static memory management. This flexibility enables efficient utilization of memory resources based on the variable's usage and scope.

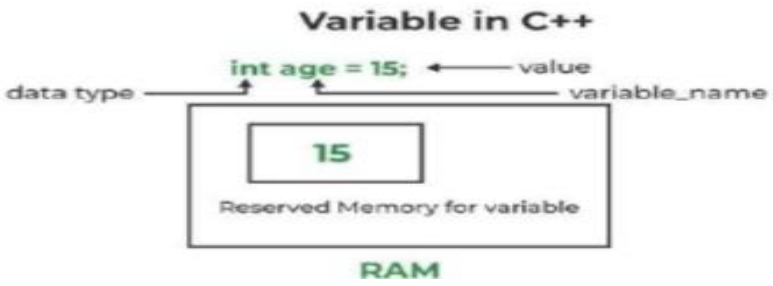
Lifetime Control: Storage classes determine the lifetime of variables. The "Auto" variables have a limited scope and are destroyed automatically when they go out of scope. The "Static" variables persist across function calls, retaining their values throughout the program's execution. This control over variable lifetimes ensures memory is allocated and deallocated appropriately, preventing memory leaks.

Performance Optimization: The "register" storage class allows programmers to suggest to the compiler that a variable should be stored in a CPU register, which can result in faster access. While modern compilers often make their own decisions about register allocation, using the "register" keyword can help improve performance in critical sections of code.

Sharing Data: The "extern" storage class enables variables to be shared across different translation units (source files). It is essential for creating global variables that can be accessed from multiple parts of a program, facilitating communication between different parts of the codebase.

Code Organization: Proper use of storage classes contributes to well-structured and organized code. By choosing the appropriate storage class for each variable, programmers can indicate the variable's intended scope, visibility, and lifecycle, making the code easier to understand and maintain.

Example for declaring a variables.[Continue]



Here, int age=15; // single variable.

```
Example:
main( )
{
float x;
float sum=0.0;
for(int i=1;i<5;i++)
{
cin>>x;
sum=sum+x
}
float average;  average=sum/4;  cout<<average; return 0;
}
```

Here, the program is declaring for multiple variables.

Giving Multiple inputs from the user.

```
#include<iostream>
using namespace std;
int main()
{
string name;
int age;
cin>>name>>age;
cout<<"NAME="<<name<<endl;
cout<<"Age="<<age<<endl;
 return 0;
}
```

Output

NAME= ANU.

AGE= 20.