AMBITION GURU

C++ Programming (HCAC-154)

II semester, BCA

Compiled by Ankit Bhattarai
Ambition Guru College

Compiled by ab

Syllabus

AMBITION GURU

Unit	Contents	Hours	Remarks
1.	**OOP Basics**	**9**	
2.	Classes & Objects	10	
3.	Overloading and Inheritance	10	
4.	Pointers	8	
5.	Virtual function and polymorphism	8	
6.	Templates and Exception Handling	7	
7.	I/O stream	8	

Practical Works

Credit hours : 3

Practical Works

Laboratory work:

1. Functions & Recursion.
2. Inline Functions.
3. Programs to Understand Different Function Call Mechanism.
4. Call by reference & Call by Value
5. Programs to Understand Storage Specifiers.
6. Constructors & Destructors.
7. Use of “this” Pointer. Using class
8. Programs to Implement Inheritance and Function Overriding.
9. Multiple inheritance –Access Specifiers
10. Hierarchical inheritance – Function Overriding /Virtual Function
11. Programs to Overload Unary & Binary Operators as Member Function & Non Member Function.
12. Programs to Understand Friend Function & Friend Class.
13. Programs on Class Templates
14. Use Exception handling example
15. Program to use Stream Console and File Input / Output

Compiled by ab

Resources

Resources

1. Some chapters in pdf form.
2. Question Collection & Assignments (50-80 Questions)

Text books:

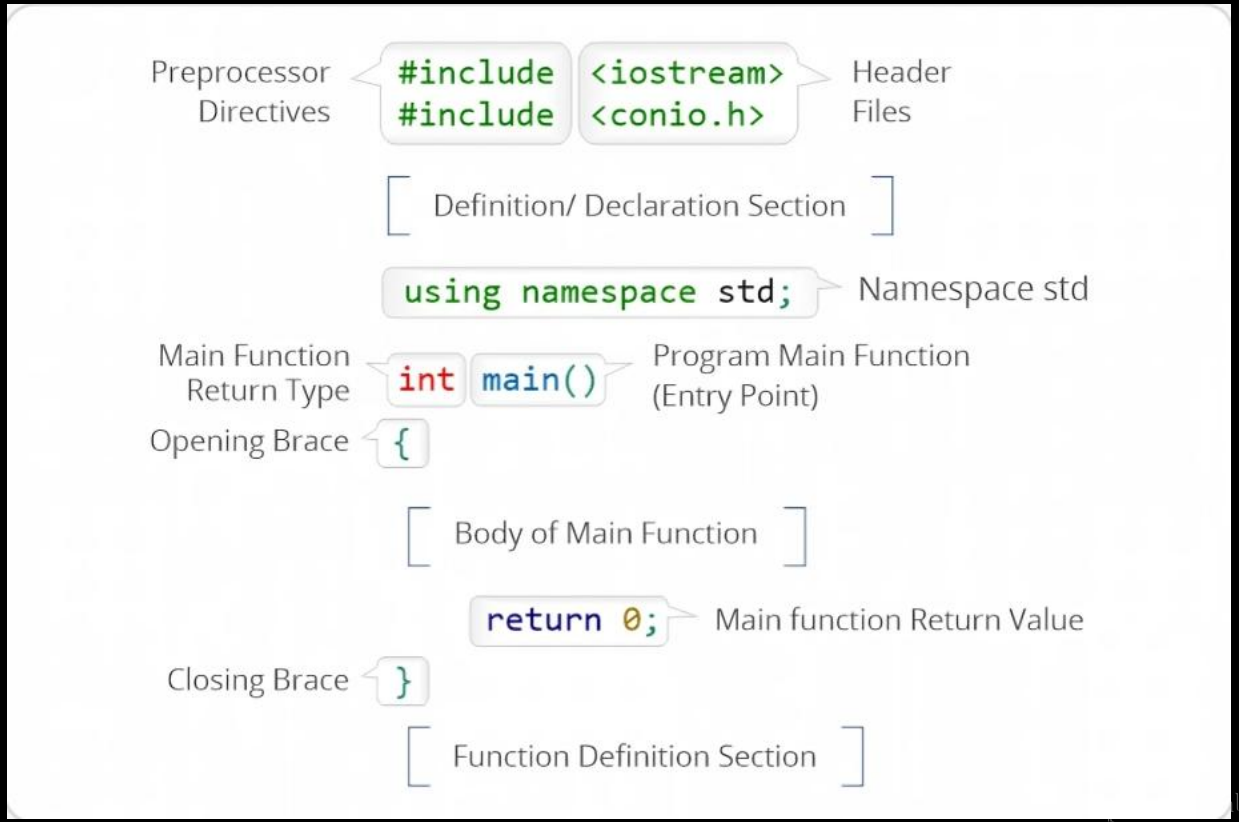
1. Lafore Robert, “Object Oriented Programming in Turbo C++”, Galgotia
2. E. Balaguruswamy: Object Oriented Programming with C++, Tata McGraw Hill

Reference:

1. Lippman, “C++ Primer”, 3rd Edition, Pearson Education, 2010.
2. Farrell, “Object Oriented Programming Using C++”, 1st Edition 2008, Cengage Learning India.
3. NavajyotiBarkakati, "Object-Oriented Programming in C++" Prentice Hall of India
4. Venugopal, Rajkumar & Ravishankar, "Mastering C++" Tata Mc Graw Hill Publication, India

Format for C++ program

AMBITION GURU



AMBITION GURU

Structure of a C++ program

// my first program in C++

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

Output : Hello World!

Unit 1

OOP Basics

(9 hrs.)

- **Introduction:** Basic Concept of OOP, differentiate between Procedural Programming and OOP, Benefits of OOP, Object &Class, Data Abstraction, Data Encapsulation, Data Hiding member functions, Reusability, Inheritance, creating new Data Types, Polymorphism, Overloading, Dynamic binding and Message passing.

- **C++ Features:** the iostream class, C++ Comments, Tokens, Keywords and Identifiers, Variable declaration, the const Qualifier. The endl, Set Waste precision, Manipulators, Operators in C++,The scope resolution operator, The new & delete Operations., Implicit Conversion, Control Structure in C++.

- **Functions:** Simple Functions, Function declaration, calling the function, function definition, passing argument to, returning value from function, passing constants, Variables, pass by value, passing structure variables, pass by reference, Default arguments, return statements, return by reference, overloaded functions; Different number of arguments, Different Kinds of argument, Static Data Member and Static Function, inline function.

Introduction: Basic Concept of OOP, differentiate between Procedural Programming and OOP, Benefits of OOP, Object & Class, Data Abstraction, Data Encapsulation, Data Hiding member functions, Reusability, Inheritance, creating new Data Types, Polymorphism, Overloading, Dynamic binding and Message passing.

Object Oriented Programming

Definition: A programming paradigm based on the concept of "objects," which contain data (attributes) and code (methods).

Key Characteristics:

- **Encapsulation:** Bundles data and methods that operate on the data.
- Inheritance: Enables a class to derive properties and behavior from another class.
- Polymorphism: Allows methods to perform different tasks based on the object that invokes them.
- Abstraction: Hides implementation details from the user.

Object Oriented Programming

Advantages:

- Code reusability: Classes can be reused across programs.
- Scalability: Facilitates large-scale application development.
- Easier maintenance: Modular structure makes it easier to manage changes.

Disadvantages:

- Complexity: Can be harder to learn and implement for beginners.
- Overhead: May introduce performance and resource consumption issues.

Examples: Languages: Python, Java, C++, C#.

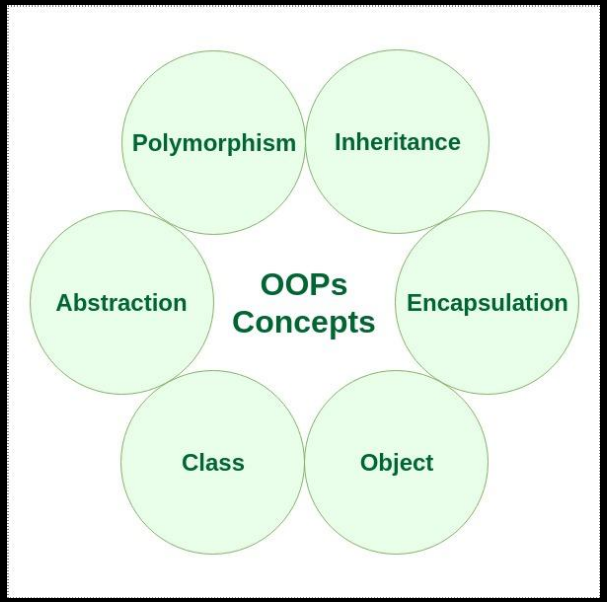
Differences between Procedural and Object Oriented Programming

AMBITION GURU

Procedural Programming	Object Oriented Programming
In Procedural programming, a program is divided into small programs that are referred to as functions.	In OOP, a program is divided into small parts that are referred to as objects.
It is less secure than OOPs	Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming.
It follows a top-down approach.	It follows a bottom-up approach.
There are no access modifiers in procedural programming.	The access modifiers in OOP are named as private, public, and protected.
Procedural programming does not have the concept of inheritance.	There is a feature of inheritance in object-oriented programming.
It is not appropriate for complex problems.	It is appropriate for complex problems.
Examples of Procedural programming include C, Fortran, Pascal, and VB.	The examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++.

Object Oriented Programming

- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.



Class & Objects

What is Class ?

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 instance of that class.

A class is like a **blueprint for an object**.

Data members => data variables and member functions => functions

Functions are used to manipulate these variables and together these data members and member functions define the **properties and behavior** of the objects in a Class.

What is Class ?

Example: Class of Cars

There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. will act as data members.

Applying brakes, increasing speed may lead to member function of that class.

AMBITION GURU

What is Class ?

A class can be thought of as a template used to create a set of objects.

A class is a static definition; a piece of code written in a programming language.

One or more objects described by the class are ***instantiated*** at runtime.

The objects are called ***instances*** of the class.

Each instance will have its own distinct set of attributes.

Every instance of the same class will have the same set of attributes;

every object has the same attributes but,

each instance will have its own distinct values for those attributes.

Data Members and Member Functions

Data members

The variables which are declared in any class by using any fundamental data types (like int, char, float etc.) or derived data type (like class, structure, pointer etc.) are known as Data Members.

Member Functions

And the functions which are declared in class are known as Member functions.

What do you mean by object ?

- Objects are the runtime entities, existing in the real world. They can be represented as person, a place, a bank account.
- Real-world objects have attributes and behaviors.
    - Examples:

    Dog =>

        - Attributes: breed, color, hungry, tired, etc.
        - Behaviors: eating, sleeping, etc.

    Bank Account =>

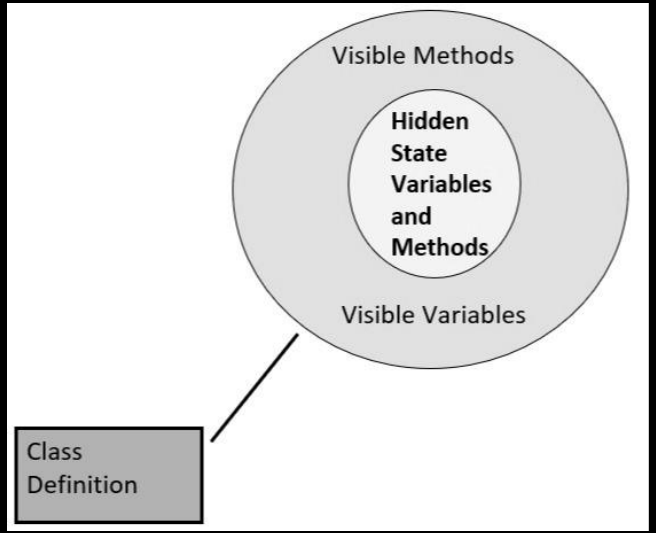
    - Attributes: account number, owner, balance
    - Behaviors: withdraw, deposit

Note: When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. One class can have multiple objects, which can interact with each other.

Encapsulation

Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation.
- When classes are defined, programmers can specify that certain methods or state variables remain hidden inside the class.
- These variables and methods are accessible from within the class, but not accessible outside it.
- The insulation of the data from direct access by the program is called data hiding or information hiding.



Encapsulation

- In OOP, hiding member functions means making methods inaccessible from outside the class or limiting their visibility.
- This is done to: Prevent misuse of internal logic, control access to critical functionality and improve modularity and maintainability.

How Member Functions Can Be Hidden

Using Access Modifiers: Most OOP languages (like C++, Java, Python, etc.) support access modifiers:

Modifier	Description
private	Accessible only within the same class.
protected	Accessible within the class and its subclasses.
Public	Accessible from anywhere.

Encapsulation

Why Hide Member Functions?

- **Encapsulation**: Keep internal details hidden to reduce complexity.
- **Security**: Prevent unauthorized access or changes.
- **Abstraction**: Expose only relevant methods to users.
- **Maintenance**: Easier to update internal logic without affecting external code.

Abstraction

Abstraction

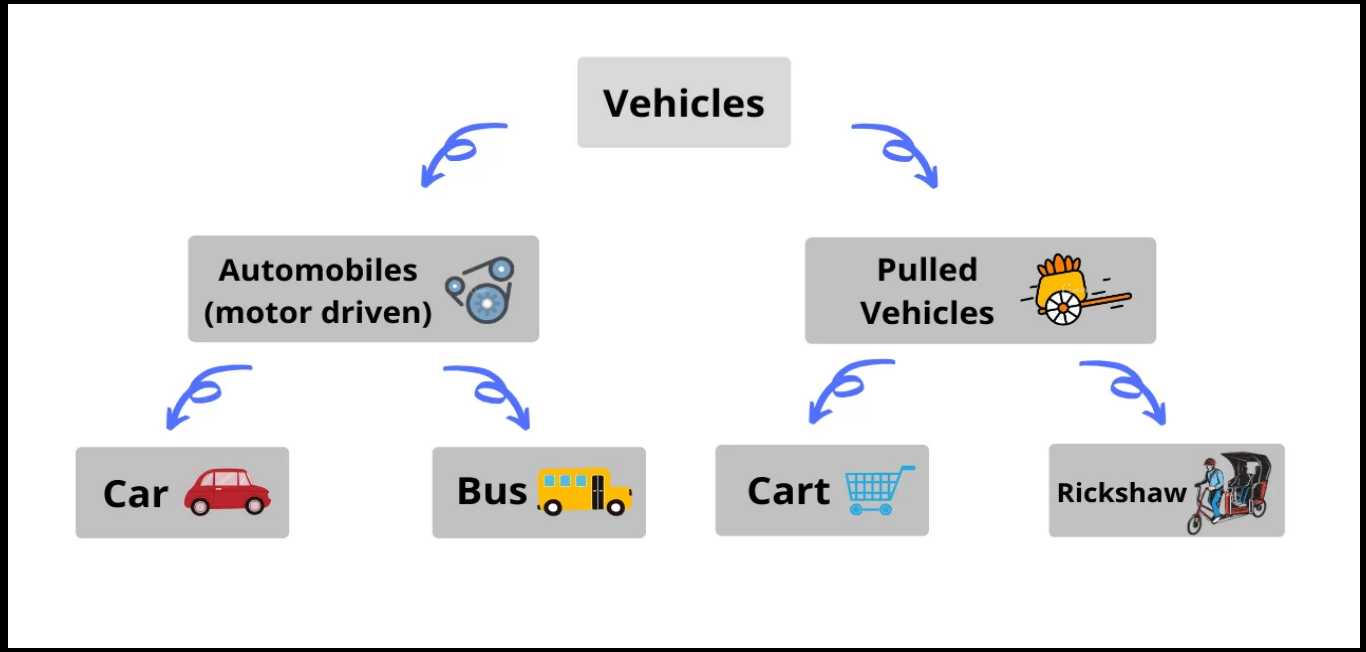
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.
- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc. in the car. This is what abstraction is.

Inheritance

What is inheritance ?

- Inheritance is the process by which objects of one class acquire the properties of objects of another class.
- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- Inheritance is one of the most important features of Object-Oriented Programming.
  - **Sub Class**: The class that inherits properties from another class is called Sub class or Derived Class.
  - **Super Class**: The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

What is inheritance ?



Polymorphism

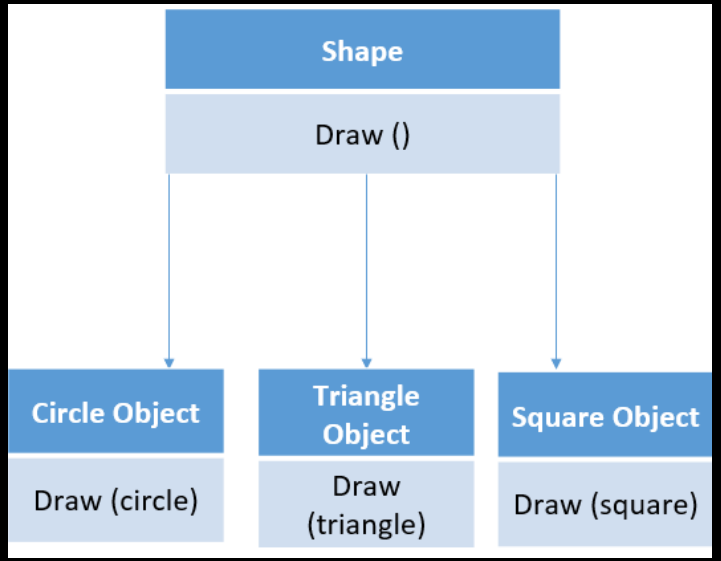
What is polymorphism ?

- Polymorphism is one of the essential features of an object- oriented language. It means ability to take more than one form.
- An operation may exhibit different behaviors in different instances. The behavior depends upon the type of data used in the operations.
- **Example:** Addition operation, with integer datatype, it will generate sum, but if operands are strings, it will generate third string by concatenation.
- The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading.**

AMBITION GURU

What is polymorphism ?

- Single function name can be used to handle different number and different types of arguments.
- This is something similar to a particular word having several different meaning depending on the context. So, using *a single function name to perform different types of task*s is known as function overloading.
- Polymorphism is extensively used in implementing inheritance.



Dynamic Binding

Dynamic Binding

- Dynamic binding refers to linking a procedure call to code that will execute only once.
- It means that the program waits until runtime to decide which version of a function to call.
- The code associated with the procedure is not known until the program is executed, which is also known as late binding.
- Dynamic binding happens when all information needed for a function call cannot be determined at compile-time.
- Dynamic binding can be achieved using the virtual functions.
- **Advantage of using dynamic binding** is that it is flexible since a single function can handle different type of objects at runtime

Message Passing

Message Passing

- An object-oriented program consists of a set of objects that communicate with each other by sending and receiving information same as people pass message to one another.
- Message passing means sending a message from one object to another to request an action.

The process of programming in an object-oriented language involves following steps:

- Creating classes that define objects and their behavior.
- Creating objects from class definition &
- Establishing communication among objects.
- Objects have a life cycle, hence can be created and destroyed Message passing involves specifying the name of the object, name of the function (message) & the information to be sent.

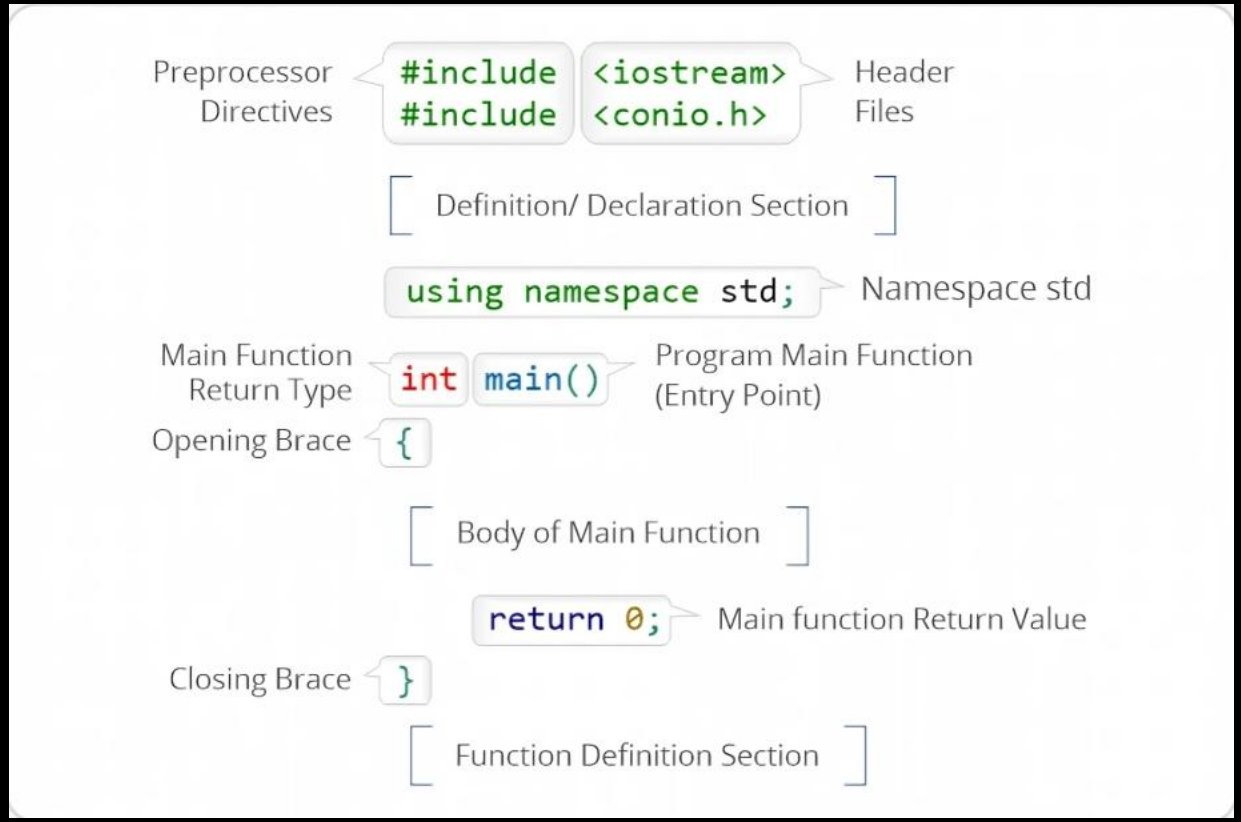
Message Passing

- A message is always given to some object called receiver. The action performed in response to message is not fixed but differ depending upon the class of the receiver .i.e. different object may accept the same message yet perform different action.
- There are three identifiable parts to any message passing expression. These are receiver (object to which message is being sent); the message selector (the text that indicates the particular message being sent) and the argument (used in responding to the message). It is a dynamic process of asking an object to perform action.

  a.getdata(100); where 'a' is declared as an instance of class.

C++ Features: the iostream class, C++ Comments, Tokens, Keywords and Identifiers, Variable declaration, the const Qualifier. The endl, Set Waste precision, Manipulators, Operators in C++,The scope resolution operator, The new & delete Operations., Implicit Conversion, Control Structure in C++.

Format for C++ program



Structure of a C++ program

// my first program in C++

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

Output : Hello World!

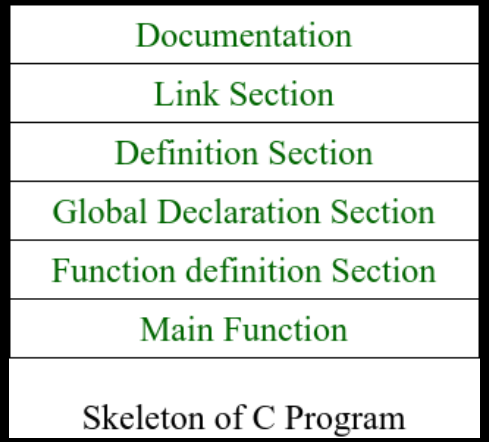
Structure of a C++ program

The structure of the program written in C++ language is as follows:

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.

Linking Section:
The linking section contains two parts:

1. Header Files:
Standard headers are specified in a program through the preprocessor directive #include.

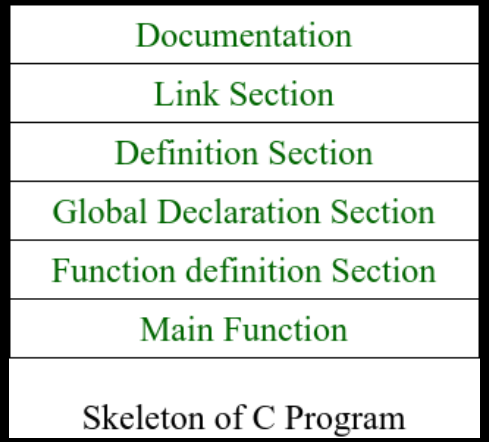


Structure of a C++ program

2. Namespaces: A namespace permits grouping of various entities like classes, objects, functions, and various C++ tokens, etc. under a single name.

Namespaces can be accessed in multiple ways:

```
using namespace std;
using std :: cout;
```



Skeleton of C Program

Definition Section:

It is used to declare some constants and assign them some value.

Structure of a C++ program

Global Declaration Section:

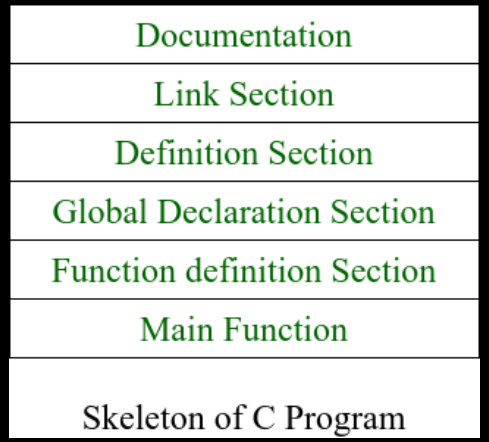
Here, the variables and the class definitions which are going to be used in the program are declared to make them global.

Function Definition Section:

It contains all the functions which our main functions need. Usually, this section contains the User-defined functions.

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.



Structure of a C++ program

```
// my first program in C++

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

Output : Hello World!

Structure of a C++ program

// my first program in C++

- This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program.
- The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

Structure of a C++ program

```
#include <iostream>
```

- They are not regular code lines with expressions but indications for the compiler's preprocessor.
- In this case the directive #include <iostream> tells the preprocessor to include the iostream standard file.
- This specific file (iostream) includes the *declarations of the basic standard input-output library in C++,* and it is included because its functionality is going to be used later in the program.

Structure of a C++ program

```
using namespace std;
```

- In C++, most of the features from the standard library (like cout, cin, string, etc.) are stored inside something called a namespace.
- Think of a namespace as a folder that keeps things organized so that names don’t conflict with each other.
- The standard C++ library uses a namespace called std, which stands for standard.

Structure of a C++ program

```
int main ()
```

- This line marks the start of the main function, which is where every C++ program begins execution. No matter where it appears in the code, the program always starts running from here.
  - main is the function name.
  - () means it's a function (can include parameters).
  - {} contains the code that runs when the program starts.

Every C++ program must have a main() function.

Structure of a C++ program

cout()

- cout is used to display output on the screen in C++.
- It stands for "character output".
- It is part of the standard library .
- The << operator is called the insertion operator — it sends the data to the output stream (usually the screen).

NOTE:

`cin()`

- cin is used to take input from the user in C++.
- It stands for "character input".
- It is part of the standard library.
- The >> operator is called the extraction operator — it takes data from the user and stores it in a variable.

Tokens, Keywords and Identifiers, Variable declaration, the const Qualifier, endl, datatypes

Tokens

Tokens are the smallest units in a C++ program. The compiler breaks a program into these tokens while parsing.

Types include:

- **Keywords**
- **Identifiers**
- **Constants**
- **Operators**
- **Punctuation/Symbols**

Example C++ program

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

int main() {
      int age = 25;          // 'int' is a keyword, 'age' is an identifier
     float height = 5.9;      // 'float' is a keyword, 'height' is an identifier

    const float PI = 3.14159; // 'const' is a keyword, PI is a constant identifier

     cout << "Age: " << age << endl;
     cout << "Height: " << height << endl;
     cout << "PI (constant): " << PI << endl;

     return 0; // 'return' is a keyword
}
```

Output:
Age: 25
Height: 5.9
PI (constant): 3.14159

Keywords

Keywords are reserved words in C++ with special meaning, like int, return, if, while, etc. You cannot use them as variable names.

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

Identifiers

identifiers are the names you assign to variables, functions, arrays, classes, etc. To ensure the compiler recognizes them correctly, you must follow specific **rules**:

Rules for Defining Identifiers in C++

- Can only contain : Letters (A–Z or a–z), Digits (0–9), Underscore (_)
- Must begin with a letter or underscore (_). *Example: total, _value, sum1 are valid, 1sum is invalid*
- Cannot be a C++ keyword. *Example: int, while, class cannot be used as identifiers.*
- Case-sensitive. *Example: score, Score, and SCORE are treated as different identifiers*
- No special characters or spaces. *Example: Invalid: user-name, total$, first name*

Variable Declaration

Declaring a variable means specifying its name and type so the compiler can reserve memory for it.

Syntax: `data_type variable_name;`

Example: `int count;`

const Qualifier

The const qualifier is used to declare a constant value that cannot be changed after initialization.

Example:
```
const float PI = 3.14;
```

endl

In C++, endl is a stream manipulator used with cout to:

- Insert a newline character (\n)
- Flush the output buffer

Example:

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

int main() {
    cout << "Line 1" << endl;
    cout << "Line 2" << endl;
    return 0;
}
```

Output:
Line 1
Line 2

Note:

Datatype in C++

Datatype in C++

AMBITION GURU

1. int (Integer)

Use: Stores whole numbers (no decimal point).

Size: Typically 4 bytes.

Range: -2,147,483,648 to 2,147,483,647 (on 32-bit systems)

Example: `int age = 25;`

2. char (Character)

Use: Stores a single character.

Size: 1 byte.

Stored as: ASCII value internally.

Example: `char grade = 'A';`

Datatype in C++

AMBITION GURU

3. float (Floating-point)

Use: Stores decimal numbers (single precision).

Size: 4 bytes.

Precision: Up to 6-7 digits after the decimal.

Example: `float temperature = 36.6;`

4. double (Double floating-point)

Use: Stores decimal numbers with more precision than float.

Size: 8 bytes.

Precision: Up to 15 digits.

Example: `double pi = 3.14159265359;`

Datatype in C++

AMBITION GURU

5. string

Use: Stores a sequence of characters (text).

Note: It's part of the std namespace, so you need to #include <string>.

Example: `string name = "Ambition";`

6. bool (Boolean)

Use: Stores truth values: true or false.

Size: Typically 1 byte.

Used in: Conditions, loops, flags, etc.

Example: `bool isPassed = true;`

Datatype in C++ Example:

```
#include <iostream>
#include <string> // Required for using string type
using namespace std;
int main() {
    int age = 25;                // Integer data type
    float height = 5.9;          // Float data type
    char grade = 'A';            // Char data type
    string name = "Ambition";       // String data type
    bool isStudent = true;       // Bool data type: true or false

    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Height: " << height << endl;
    cout << "Grade: " << grade << endl;
    cout << "Is Student: " << isStudent << endl;
    return 0;
}
```

Set Precision & Manipulators in C++

In C++, manipulators from the <iomanip> header help control the formatting of output, especially for floating-point numbers. One of the most used manipulators is setprecision.

Set Precision & Manipulators in C++

1. setprecision(n):

Sets the total number of significant digits (not just after the decimal point).

```
#include <iostream>
#include <iomanip> // required for setprecision
using namespace std;

int main() {
    float pi = 3.14159265;

    cout << "Default: " << pi << endl;
    cout << "setprecision(2): " << setprecision(2) << pi << endl;
    cout << "setprecision(5): " << setprecision(5) << pi << endl;

    return 0;
}
```

Set Precision & Manipulators in C++

2. fixed: Forces the number to be displayed in fixed-point notation — it makes setprecision(n) count digits after the decimal point.

Code:

```
cout << fixed << setprecision(2) << pi;
// Output: 3.14
```

3. scientific: Displays the number in scientific (exponential) format.

Code:

```
cout << scientific << pi;
// Output: 3.141593e+00
```

Set Precision & Manipulators in C++

4. setw(n): Sets the width (in characters) of the next output field.

Code:

```
cout << setw(10) << 123;
// Output: '       123' (padded with spaces)
```

5. left, right: Used to justify output.

Code:

```
cout << left << setw(10) << 123;
cout << right << setw(10) << 123;
```

Operators in C++,The scope resolution operator, The new & delete Operations., Implicit Conversion, Control Structure in C++.

Operators in C++

- **Operators** in C++ are special symbols used to perform operations on variables and values. C++ supports a wide range of operators grouped into different types.

1. Arithmetic Operators: These operators perform mathematical calculations: + (Addition), - (Subtraction), * (Multiplication), / (Division), and % (Modulus - remainder of division).

2. Assignment Operators: These operators assign values to variables:

- = (Assignment)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

Operators in C++

AMBITION GURU

3. **Comparison Operators:** These operators compare two values and return a boolean result:

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

4. Logical Operators: These operators combine or modify boolean expressions:

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

Operators in C++

AMBITION GURU

5. Bitwise Operators: These operators perform operations on the individual bits of integers:

& (Bitwise AND), | (Bitwise OR), ^ (Bitwise XOR), ~ (Bitwise NOT), << (Left shift), and >> (Right shift).

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

int main() {
    int a = 6, b = 4;

    // Binary AND operator
    cout << "a & b is " << (a & b) << endl;

    // Binary OR operator
    cout << "a | b is " << (a | b) << endl;

    // Binary XOR operator
    cout << "a ^ b is " << (a ^ b) << endl;

    // Left Shift operator
    cout << "a<<1 is " << (a << 1) << endl;

    // Right Shift operator
    cout << "a>>1 is " << (a >> 1) << endl;

    // One's Complement operator
    cout << "~(a) is " << ~(a);

    return 0;
}
```

Output:
a & b is 4
a | b is 6
a ^ b is 2
a<<1 is 12
a>>1 is 3
~(a) is -7

Operators in C++

AMBITION GURU

6. Unary Operators: These operators act on a single operand:

- ++ (Increment)
- -- (Decrement)
- + (Unary plus)
- - (Unary minus)
- ! (Logical NOT)
- ~ (Bitwise NOT)

7. Other Operators:

- sizeof (Returns the size of a variable or type)
- ? : (Ternary operator - conditional expression)
- :: (Scope resolution operator)
- . (Member access operator)
- -> (Pointer member access operator)

Scope resolution operator

The scope resolution operator :: is used in C++ to access global variables, class members, namespaces, or to define functions outside a class.

Scope resolution operator :: is used for following purposes:

- To access a global variable when there is a local variable with same name
- To define a function outside a class.
- Refer to a class inside another class
- For namespace
- To access a class's static variables.
- In case of multiple Inheritance

Scope resolution operator

AMBITION GURU

Use-1: To access global variable:

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

int m=30; //global m

int main()
{
    int m=20;       //m redeclared, local to main
    {
        int m=10;   //m declared again,local to inner block
        cout<<"we are in inner block"<<endl;
        cout<<"value of m="<<m<<"\n";
        cout<<"value of ::m="<<::m<<"\n";
    }
    cout<<endl<<"we are in outer block"<<endl;
    cout<<"value of m="<<m<<"\n";
    cout<<"value of ::m="<<::m<<"\n";
return 0;
}
```

Output:

```
we are in inner block
value of m=10
value of ::m=30

we are in outer block
value of m=20
value of ::m=30
```

Scope resolution operator

AMBITION GURU

Use-2: To define a function outside a class:

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

class A
{
    public:
        void fun();  // Only declaration
};

// Definition outside class using ::
void A::fun()
{
   cout << "fun() called";
}

int main()
{
   A a;
   a.fun();
   return 0;
}
```

Output:

```
fun() called
```

Scope resolution operator

AMBITION GURU

Use-3: Refer to a class inside another class

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

class outside
{
    public:
        int x;
        class inside
        {
            public:
                int x;
        };
};

int main()
{
    outside A;
    outside::inside B;
    A.x=5;
    cout<<A.x<<endl;
    B.x=50;
    cout<<B.x;

}
```

Output:

```
5
50
```

Scope resolution operator

AMBITION GURU

Use-4: Namespace

```cpp
#include<iostream>

int main()
{
    std::cout << "Hello" << std::endl;
}
```

Output:

```
Hello
```

New & delete Operations

New & delete Operations

In C++, new and delete are used for dynamic memory allocation and deallocation at runtime. They replace traditional malloc() and free() from C with safer and type-aware mechanisms.

1. new Operator:

- Allocates memory on the heap.
- Returns a pointer to the allocated memory.
- Automatically calls constructor (for objects).

New & delete Operations

Syntax:

```
pointer = new dataType;
pointer = new dataType(value);      // with initialization
pointer = new dataType[size];       // array
```

Example:

```
int* ptr = new int;         // allocates memory for 1 int
*ptr = 10;
cout << *ptr << endl;       // output: 10
```

New & delete Operations

2. delete Operator

- Frees memory allocated by new.
- Prevents memory leaks.
- Automatically calls destructor (for objects).

Syntax:

```
delete pointer;
delete[] pointer;
```

Example:

```
delete ptr;
delete[] arr;
```

New & delete Operations

Example:

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

int main() {

    int *num = new int(25);

    cout << "Value: " << *num << endl;

    // Freeing memory
    delete num;
    return 0;
}
```

Output:
Value: 25

Type Conversion

Type Conversion

- Type conversion is the process that converts the predefined data type of one variable into an appropriate data type.
- The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.
- For example, we are adding two numbers, where one variable is of int type and another of float type; we need to convert or typecast the int variable into a float to make them both float data types to add them.

There are two types of type conversion:

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion

- The type conversion that is done automatically by the compiler is known as implicit type conversion.
- This type of conversion is also known as automatic conversion.
- When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically.
- This is also called type promotion.

- The order of types is given below:

Data Type	Order
long double	(highest)
double	
float	
long	
int	
char	(lowest)

// An example of implicit conversion

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    char y = 'a';
    float z;
    // y implicitly converted to int.
    //ASCII value of 'a' is 97
    x = x + y;
    // x is implicitly converted to float
    z = x + 1.5;
```

```
cout << "x = " << x << endl
        << "y = " << y << endl
        << "z = " << z << endl;
    return 0;
}
```

Output:

x = 107

y = a

Z = 108.5

Explicit Type Conversion

- Conversions that require ***user intervention*** to change the data type of one variable to another, is called the **explicit type conversion**.
- In other words, an explicit conversion allows the programmer to manually change or typecast the data type from one variable to another type. Hence, it is also known as typecasting.
- Generally, we force the explicit type conversion to convert data from one type to another because it does not follow the implicit conversion rule.
- C++ permits explicit type conversion of variables or expressions as follows:

(type-name) expression //C notation

type-name (expression) //C++ notation

Example:

```
#include <iostream>
using namespace std;
int main() {
    double num_double = 5.63;
    int num_int1,num_int2;
    cout << "num_double = " << num_double << endl;

    // C-style conversion from double to int
    num_int1 = (int)num_double;
    cout << "num_int1   = " << num_int1 << endl;
```

```
    // function-style conversion from double to int
    num_int2 = int(num_double);
    cout << "num_int2   = " << num_int2 << endl;
    return 0;
}
```

Output:

num_double = 5.63

num_int1 = 5

num_int2 = 5

Control Structure in C++

Control Statements

1. Sequential structure (straight line)
2. Selection structure (branching or decision)
3. Loop structure (iteration or repetition)

Sequence structure (straight line)

- It is the default mode i.e., sequential execution of code statements (one line after another from top to bottom).

```
statement 1;
statement 2;
statement 3;
……..
statement n;
```

AMBITION GURU

```
#include<iostream>
using namespace std;
int main()
{
    int num1,num2,sum;
    cout<<"Enter two numbers:\n";
    cin>>num1>>num2;
    sum=num1+num2;
    cout<<"Sum is:"<<sum;
    return 0;
}
```

Output:

```
Enter two numbers:
3
4
Sum is:7
```

Conditional Structure

- Conditional statements help us to make a decision based on certain conditions.
- These conditions are specified by a set of conditional statements having Boolean expressions which are evaluated to a Boolean value true or false.
- There are following types of conditional statements.

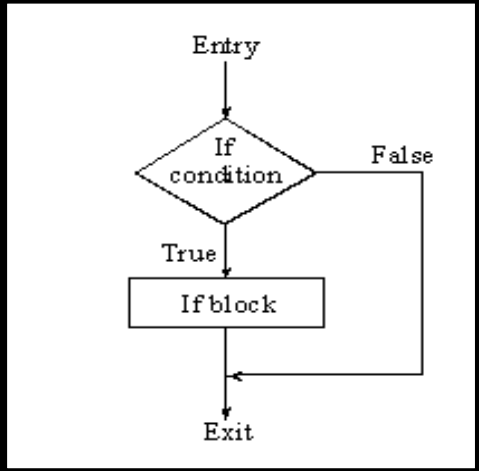
1. If statement
2. If-else statement
3. Nested if-else statement
4. If-else-if ladder
5. Switch statement

If statement

- The single if statement in C++ language is used to execute the code if a condition is true. It is also called one-way selection statement.

Syntax:

```
if(condition)
{
        statement(s);
}
```



```
#include<iostream>
using namespace std;
int main()
{
    int num;
    cout<<"Enter a number:\n";
    cin>>num;
    if(num%2==0)
    {
        cout<<num<<" is even";
    }
    return 0;
}
```

Output:

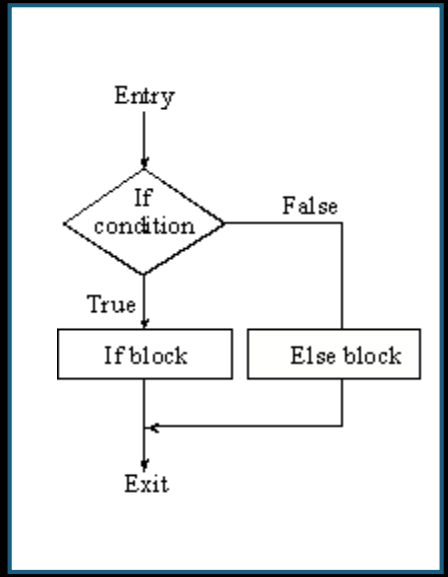
```
Enter a number:
10
10 is even
```

If...else statement

- The if...else statement in C++ language is used to execute the code if condition is true or false.
- It is also called two-way selection statement.

Syntax:

```
if(condition)
 {
    statements;
 }
else
 {
   statements;
 }
```



```
#include<iostream>
using namespace std;
int main(){
    int num;
    cout<<"Enter a number:\n";
    cin>>num;
    if(num%2==0){
        cout<<num<<" is even";
    }
    else{
        cout<<num<<" is odd";
    }
    return 0;
}
```

Output:

```
Enter a number:
5
5 is odd
```

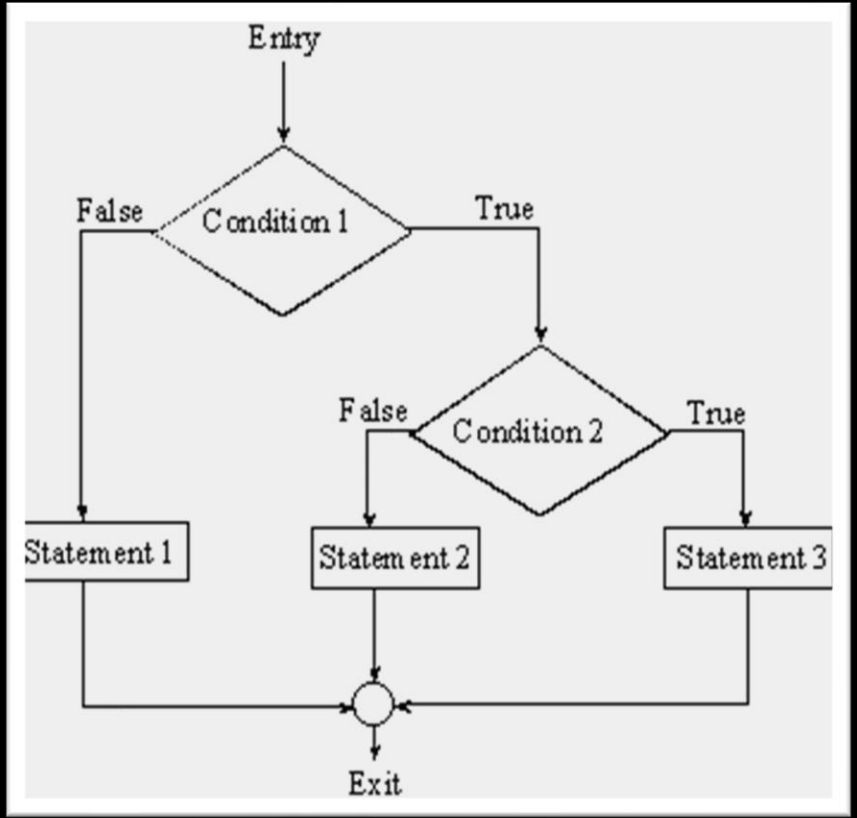
Nested if...else statement

- The nested if...else statement is used when a program requires more than one test expression.
- It is also called a multi-way selection statement. When a series of the decision are involved in a statement, we use if else statement in nested form.

AMBITION GURU

Syntax:

```
if(condition1 )
{
   if(condition2)
    {
       statement1;
    }
   else
    {
       statement 2;
    }
}
else
{
   statement 3;
}
```



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

int main() {
    int a, b, c;
    cout << "Enter 3 numbers:\n";
    cin >> a >> b >> c;

    if (a > b) {
        if (a > c) {
            cout << a << " is greatest";
        } else {
            cout << c << " is greatest";
        }
    } else {
        if (b > c) {
            cout << b << " is greatest";
        } else {
            cout << c << " is greatest";
        }
    }

    return 0;
}
```

Output:

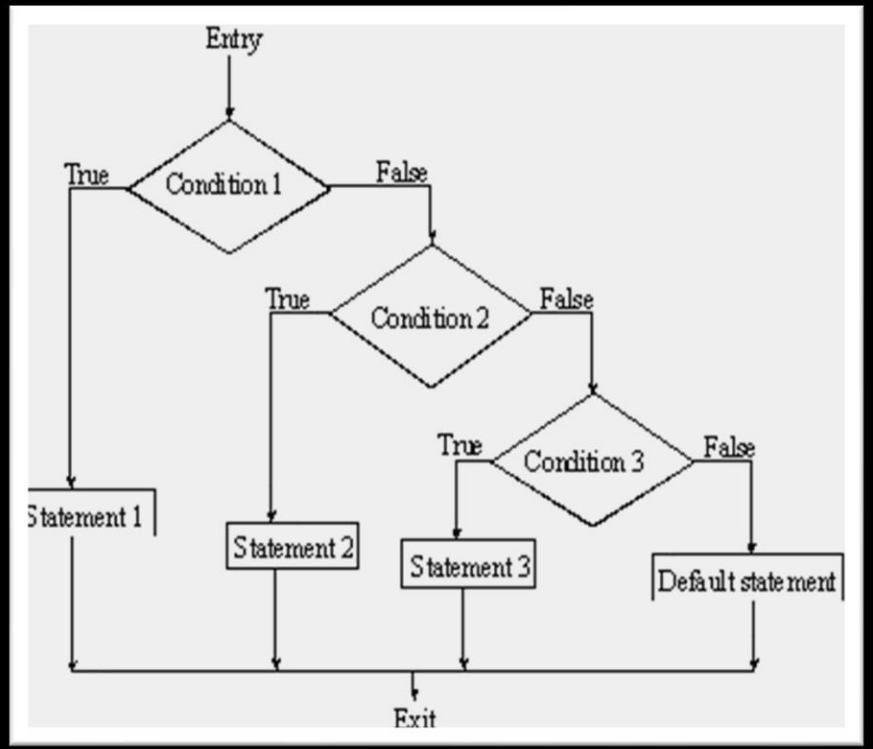
```
Enter 3 number:
3 5 2
5 is greatest
```

if...else if ladder

- The if...else if statement is used to execute one code from multiple conditions.
- It is also called multipath decision statement.
- It is a chain of if..else statements in which each if statement is associated with else if statement and last would be an else statement.

Syntax:


```
if(condition1)
        statement1;
    else if (condition2)
        statement2;
           else if (condition3)
               statement3;
              .
              .
              .
           else

default_statement;
```



Switch statement

- C++ has a built-in multiway decision statement known as "switch" which tests the value of a given variable (or expression) against a list of case values, and when a match is found, a block of statements associated with that case is executed.

Syntax:

```
switch (expression){
            case value-1:
             block-1;
            break;
             case value-2:
             block-2;
             break;
          .
          .
          .
               default:
            default-block;
    }
```

```
#include<iostream>
using namespace std;
int main( )
{
    int n;
    cout<<"Enter any no. from 1 to 7:\n";
    cin>>n;
    switch (n)
    {
        case 1:
       cout<<"Sunday";
       break;
      case 2:
       cout<<"Monday";
       break;
       case 3:
       cout<<"Tuesday";
       break;
       case 4:
       cout<<"Wednesday";
       break;
        case 5:
       cout<<"Thursday";
       break;
```

```
    case 6:
          cout<<"Friday";
          break;
    case 7:
                cout<<"Saturday";
        break;
    default:
                cout<<"Enter valid number from 1 to 7";
}
return 0;
}
```

Output:

```
Enter any no. from 1 to 7:
5
Thursday
```

Repetitive Structure

- *Repetitive structures*, or *loops* are used when a program needs to repeatedly process one or more instructions until some condition is met, at which time the loop ends.
- The process of performing the same task over and over again is called *iteration*, and C++ provides built-in iteration functionality.
- A loop executes the same section of program code over and over again, as long as a loop condition of some sort is met with each iteration.
- This section of code can be a single statement or a block of statements (a compound statement).

- There are three looping structures in C++:

1. for loop
2. do.... While
3. while Loop

for loop

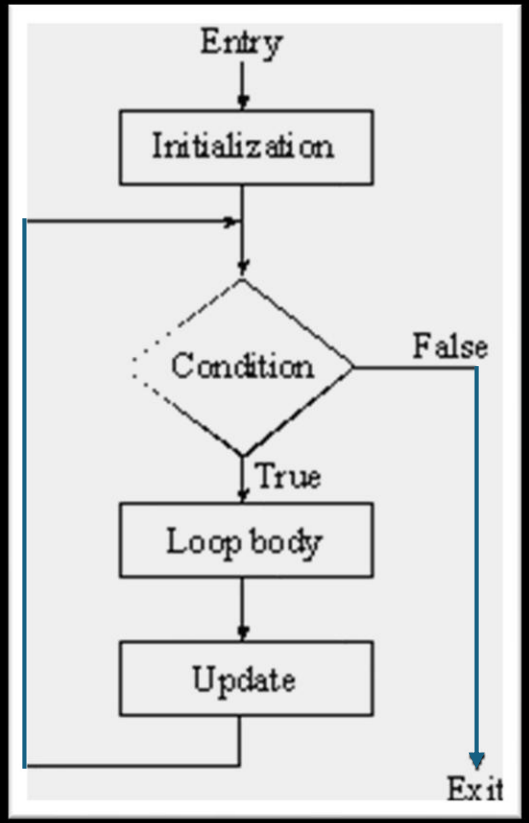
- It is a count controlled loop which is used when some statement(s) is to be repeated certain number of times.

Syntax:

```
for (initialization ; condition ; update)
{
    body of the loop; //statement(s) to be executed repeatedly
}
```

How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
- Again the test expression is evaluated.
- This process goes on until the test expression is false. When the test expression is false, the loop terminates.



```
#include<iostream>
using namespace std;
int main()
{
    int i;
    for (i = 1; i <=10; i++)
    {
        cout<<i<<"\t";
    }
    return 0;
}
```

Output:

```
1    2    3    4    5    6    7    8    9    10
```

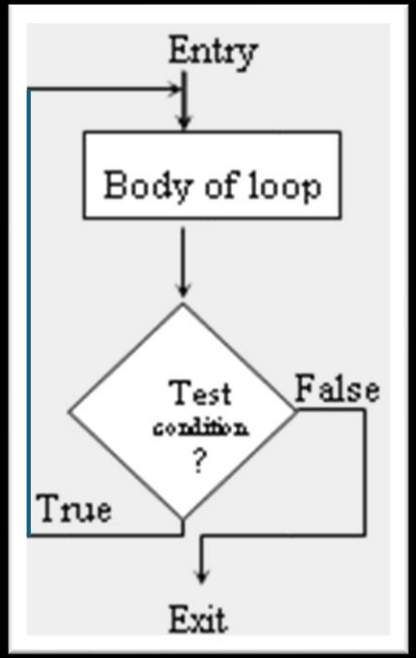
do ..... while loop

- “do-while loop” is exit-controlled loop(condition controlled) in which test condition is evaluated at the end of loop.
- The body of the loop executes at least once without depending on the test condition and continues until the condition is true.

Syntax:

```
do
{
   body of the loop;
} while (test condition);
```

Flowchart



```
#include<iostream>
using namespace std;
int main()
{
    int i;
    i = 1;
    do
    {
        cout<<i<<"\t";
        i++;
    } while (i <=10);
    return 0;
}
```

Output:

```
1    2    3    4    5    6    7    8    9    10
```

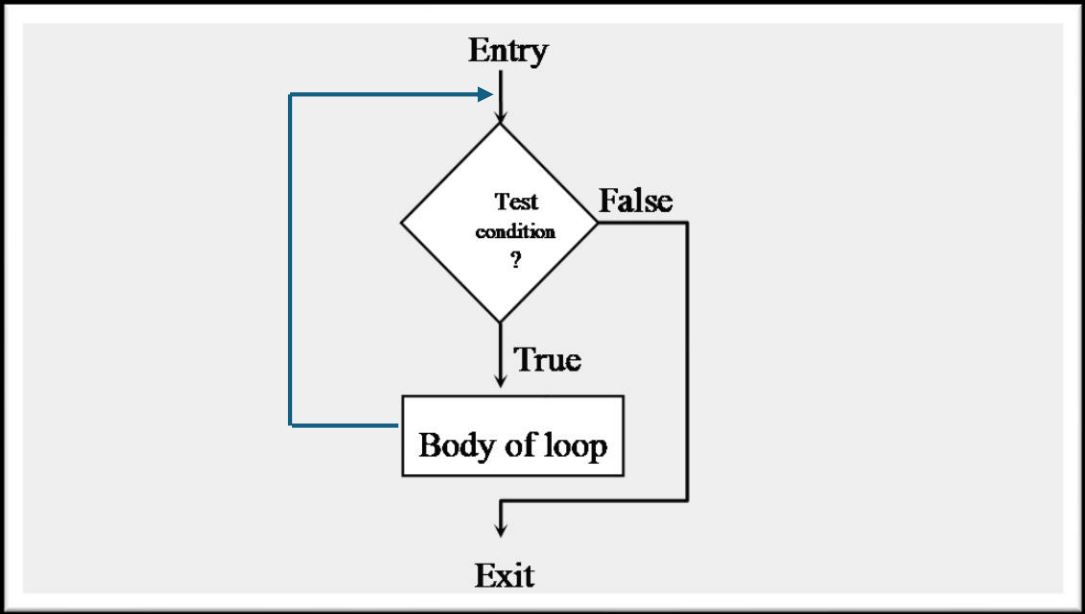
while loop

- “while loop” is entry-controlled loop (condition controlled) in which test condition is evaluated at the beginning of the loop execution.
- If the test condition is true, only then the body of the loop executes, or else it does not.

Syntax:

```
while (test condition)
{
    body of the loop;
}
```

Flowchart:



```
#include<iostream>
using namespace std;
int main()
{
    int i;
    i = 1;
    while(i<=10)
    {
        cout<<i<<"\t";
        i++;
    }
    return 0;
}
```

Output:

```
1    2    3    4    5    6    7    8    9    10
```

Functions: Simple Functions, Function declaration, calling the function, function definition, passing argument to, returning value from function, passing constants, Variables, pass by value, passing structure variables, pass by reference, Default arguments, return statements, return by reference, overloaded functions; Different number of arguments, Different Kinds of argument, Static Data Member and Static Function, inline function

Function

- A **function** is a group of statements that together perform a task.
- Functions are made for code **reusability** and for saving **time** and **space**.

Library Function	User defined function
Predefined	Created by user
Declarations inside header files	Reduced complexity of program
Body inside .dll files	

Difference between library and user defined function.

Categories of User-defined Functions

- Function with no argument and no return value.
- Function with no argument but return value.
- Function with argument but no return value.
- Function with argument and return value.

The main function

- main() function is the **entry point** at which execution of program is started.
- C allows main() function type to be void. **C++ does not all main() function to be void.**
- In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a **return(0)** statement.
- The explicit use of a return(O) statement will indicate that the **program was successfully executed.**

```
int main()
{
    .........
    .........
    return 0;
}
```

Function Prototyping / Declaration

- Describes number and type of arguments and the type of return values.
- When a function is called, the compiler ensure that proper arguments are passed, and the return value is treated correctly.
- Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.
- These checks and controls did not exist in the conventional C functions.

```
void f();

int main()
{
     f(5);
}

void f()
{

}
```

Simple Function

Function Prototyping / declaration

Syntax:
```
return-type function-name (argument-list);
```

Example:
```
float volume(int,float,float);  OR  float volume(int x,float y,float z);
```

Function Calling:
```
float volume(int x,float y,z); //illegal
```

Function Definition:
```
float cube=volume(b1,w1,h1); //Actual Parameter
```

```
float volume(int a,float b,float c)
{
    float v=a*b*c;
    .....
    .....
}
```

Simple Function

AMBITION GURU

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

int add(int, int); // Function Declaration

int main()
{
      int a, b, sum;
      cout << "Enter two numbers:"<< endl;
      cin>>a>>b;
      sum = add(a, b); //Function call
      cout<<"The sum is: "<< sum<<endl;
      return 0;
}

int add(int x, int y) //Function definition
{
      int s;
      s = x + y;
      return s;
}
```

Output:

```
Enters two numbers to add:
8
-4
Sum = 4
```

Function Overloading

- More than one functions having same name but differs either in number of arguments or type of arguments or both is said to be function overloading.
- Function overloading shows the compile time polymorphism because the particular function call and its associated function definition is known by the compiler before the running of the program i.e.at the compile time only.

Advantages of function overloading:

- Remembering names is easier
- Lesser complexity
- Increases readability

WAP to find the volume of a cube, cuboid and cylinder using the concept of functionoverloading.


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

void volume(float l)
{
   cout<<"The volume of the cube is:"<<l*l*l<<endl;
}

void volume(float l,float b,float h)
{
   cout<<"The volume of the cuboid is:"<<l*b*h<<endl;
}

void volume(float r, float h)
{
   cout<<"The volume of the cylinder is:"<<3.14*r*r*h;
}
```

```
int main( )
{
    volume(3.5);
    volume(3.5,6.5,9.5);
    volume(3.5,5.5);
    return 0;
}
```

Output:

```
The volume of the cube is:42.875
The volume of the cuboid is:216.125
The volume of the cylinder is:211.558
```

WAP to find the cube of a integer , float and double number using the concept of function overloading(passing single argument to the function)

WAP to find the cube of a integer , float and double number using the concept of function overloading(passing single argument to the function)


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

void findcube(int a){
   cout<<"The cube of the integer number is:"<<a*a*a<<endl;
}

void findcube(float b)
{
   cout<<"The cube of the float number is:"<<b*b*b<<endl;
}

void findcube(double c)
{
   cout<<"The cube of the double number is:"<<c*c*c;
}
```

```
int main( )
{
   int x;
   float y;
   double z;
   cout<<"Enter values for int, float double type variables respectively:";
   cin>>x>>y>>z;
   findcube(x);
   findcube(y);
   findcube(z);
   return 0;
}
```

Output:

```
Enter values for int, float double type variables respectively:2
3.5
4.5e+7
The cube of the integer number is:8
The cube of the float number is:42.875
The cube of the double number is:9.1125e+22
```

WAP to create a function findarea( ) to find area of rectangle, square and cube on the basis of parameters passed to it.

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

void findarea(float l,float b)
{
   cout<<"The area of the rectangle="<<l*b<<endl;
}

void findarea(float l)
{
   cout<<"The area of the square="<<l*l<<endl;
}

void findarea(double l)
{
   cout<<"The area of the cube="<<6*l*l<<endl;
}
```

```
int main( )
{
   findarea(5.5,6.5);
   findarea(7.5f);
   findarea(8.5); //findarea(9); error
   return 0;
}
```

Consider the case :


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

void findsum(int a, float b)
{
   cout<<"The sum of first integer and second float is:"<<a+b<<endl;
}

void findsum(float c, int d)
{
   cout<<"The sum of first float and second integer is:"<<c+d<<endl;
}
```

```
int main( )
{
      findsum(2,6.5); //calls first function
      findsum(7.5,3); //calls second function
      //findsum(2.5,6.5); //Ambiguity or error
      //findsum(7.5,3.5); //Ambiguity or error
      return 0;
}
```

Output:

```
The sum of first integer and second float is:8.5
The sum of first float and second integer is:10.5
```

Inline functions

AMBITION GURU

- To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.
- An **inline function** is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion).
- **Syntax:**

```
inline return-type function-name(parameters)

{
    //function code
}
```

Inline functions

AMBITION GURU

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

inline double cube(double s)
{
    return s * s * s;
}

int main()
{
    cout << "The cube of 3 is: " << cube(3.0) << "\n";
    cout << "The cube of 4 is: " << cube(2.5 + 1.5) << "\n";
    return 0;
}
```

Output:

```
The cube of 3 is: 27
The cube of 4 is: 64
```

Inline functions

AMBITION GURU

NOTE:

- All inline functions must be **defined** before they are called.
- To make a function inline, prefix the keyword **inline** to the function definition.
- Usually, functions are made inline when they are **small** enough to be defined in one or two lines.
- If the function grows in size, the speed benefit of inline function diminish.

Inline functions

- Remember, inline is only a **request** to the compiler, not a command.
- Compiler can **ignore the request** for inline if function definition is too long or too complicated and compile the function as a normal function.

Some situations where inline expansion may not work:

- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function return type is other than void, and the return statement doesn't exist in function body.
- If a function contains switch or goto statement.

Inline functions

AMBITION GURU

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

inline float mul(float x, float y)
{
    return (x*y);
}

inline double div(double p, double q)
{
    return (p/q);
}

int main()
{
    float a=12.345;
    float b=9.82;

    cout<<mul(a,b)<<"\n";
    cout<<div(a,b)<<"\n";

    return 0;
}
```

Output:

```
121.228
1.25713
```

Default Arguments

Default Arguments

- C++ allows us to call a function without specifying all its arguments.
- In such cases, the function assigns a **default value** to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is **declared**.
- The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Default Arguments

AMBITION GURU

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

int sum(int a, int b=10, int c=20);

int main()
{
    cout<<sum(1)<<endl;
    cout<<sum(1, 2)<<endl;
    cout<<sum(1, 2, 3)<<endl;
    return 0;
}

int sum(int a, int b, int c)
{
    int z;
    z = a+b+c;
    return z;
}
```

Output:

```
31
23
6
```

Default Arguments

NOTE: Only the trailing arguments can have default values and therefore we must add defaults from **right to left**.

```
int mul(int i,int j=5, int k=10);    //legal
int mul(int i=5,int j);              //illegal
int mul(int i=0,int j, int k=10);    //illegal
int mul(int i=2,int j=5, int k=10); //legal
```

Default arguments are useful in situations where some arguments always have the same value.

- **Example:** bank interest may remain the same for all customers for a particular period of deposit.

Practice Program : Default Arguments

AMBITION GURU

Write a function called power () that takes two arguments: a double value for n and an int for p, and returns the result as double value. Use **default argument** of 2 for p, so that if this argument is omitted, the number will be squared. Write a main () function that gets values from the user to test this function.

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

double power(double,int=2);

int main()
{
    int p;
    double n,r;
    cout << "Enter number : ";
    cin >> n;
    cout << "Enter exponent : ";
    cin >> p;
    r = power(n,p);
    cout << "Result is " << r;
    r = power(n);
    cout << "\nResult without passing exponent is " << r;
    return 0;
}

double power(double a, int b)
{
    double x = 1;
    for(int i = 1; i <= b; i++)
        x = x * a;
    return x;
}
```

Output:

```
Enter number : 5
Enter exponent : 4
Result is 625
Result without passing exponent is 25
```

Default Arguments

Volume of Ellipsoid = (4/3) × pi × radius1 × radius2 × radius3. Write a program having function volume () which takes three float arguments: radius1, radius 2 and radius3 and returns the volume of an Ellipsoid. Use default argument of 2 for radius1, 3 for radius2 and 4 for radius3 so that if arguments are omitted then the volume of Ellipsoid is always 100.48. Write a main ( ) function that gets values from the user to test this function.

Pass by Reference

- In pass by reference we pass the arguments in the function definition by reference and not as pass by value.
- In pass by reference during function call, the formal arguments in the called function become alias (alternative name) to the actual arguments in the calling function i.e. When we are working with formal arguments we are actually working with the actual arguments.

Reference variable:

- C++ supports one more type of variable called reference variable in addition to the value variable and pointer variables in C.
- Value variables are used to hold some numeric values and pointer variables are used to hold address of some other value variables. Reference variable behaves similar to both a value variable and a pointer variable.
- In a program code it is used similar to that of a value variable but has an action of pointer variable. Thus it provides an alias(alternative name) of the variable that is previously defined. It helps the function for returning multiple values from the function.

Syntax: data_type &reference name=variable name;

- Reference variable should be immediately initialized after its declaration with already defined variable.

Let us consider the following two programs

1)Program without use of reference variable

```
#include<iostream>

using namespace std;

 int main( )
{
    int b=10;
    int a=b;
    a++;
    cout<< "a="<<a<<endl;
    cout<<"b="<<b;
return 0;
}
```

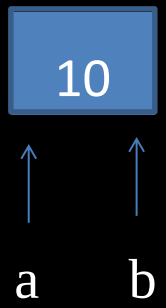
Output

a=11

b=10

Program with the use of reference variable.


```
#include<iostream>
using namespace std;
int main( )
{
      int b=10;
      int &a=b;
      a++;
      cout<<"a="<<a<<endl;
      cout<<"b="<<b;
      return 0;
}
```



Output

a=11
b=11

In above program a is an reference variable to a previously defined variable b that means a and b denote same variable and no separate memory is allocated for variable a and b that means change in one variable is affected upon the other.

Program to swap two numbers.


```
#include<iostream>
using namespace std;
void swap(int &x, int &y)
{
     int temp;
     temp=x;
     x=y;
     y=temp;
}
```

```
int main( )
{
    int a=5,b=9;
    cout<<"Before swapping"<<endl;
    cout<<"Value of a="<<a;
    cout<<"\tValue of b="<<b<<endl;
    swap(a,b);
    cout<<"After swapping"<<endl;
    cout<<"Value of a="<<a;
    cout<<"\tValue of b="<<b;
return 0;
}
```

Output

```
Before swapping
Value of a=5    Value of b=9
After swapping
Value of a=9    Value of b=5
```

Return by Reference

- In C++, function can return a variable by reference.
- Return by reference means a function is returning an alias of the variable in return statement.
- As a return by reference returns the alias of the variable so the functions call can be placed at the left hand side of the assignment operator(=).

Example:


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

int &max(int &x, int &y)
{
     if(x>y)
          return x;
     else
          return y;
}
```

```
int main( )
{
  int a=5,b=9;
  cout<<"Before calling the function a="<<a<<" and b="<<b<<endl;
  max(a,b)=-1;
  cout<<"After calling the function a="<<a<<" and b="<<b;
  return 0;
}
```

Output

Before calling the function the a=5 and b=9
After calling the function a=5 and b=-1

In above function the return type of the function is int & so the function returns alias of the variable.

Static Data Member and Static Function, Inline function

Static Data Members

- A data member of a class can be qualified as **static**.

Static data member has certain special characteristics:

- It is ***initialized to zero*** when the first object of its class is created. No other initialization is permitted.
- ***Only one copy*** of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its ***lifetime is the entire program*****.**

Static Data Members

AMBITION GURU

- A static variable is normally used to maintain value common to the entire class.
  - ***For e.g., to hold the count of objects created*.**
- Note that the type and scope of each static member variable **must be declared outside the class definition.**
- This is necessary because the static data members are stored separately rather than as a part of an object.
- While defining a static variable, some initial value can also be assigned

```
int item::count=10;
```

- Since they are associated with the class itself rather than with any class objects, they are also known as **class variables.**

Static Data Members

AMBITION GURU

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

class item
{
     int number;
     static int count;
public:
     void getdata(int a)
     {
          number =a;
          count++;
     }

     void getcount()
     {
      cout<<"count:";
      cout<<count<<"\n";
     }

};
```

Static Data Members

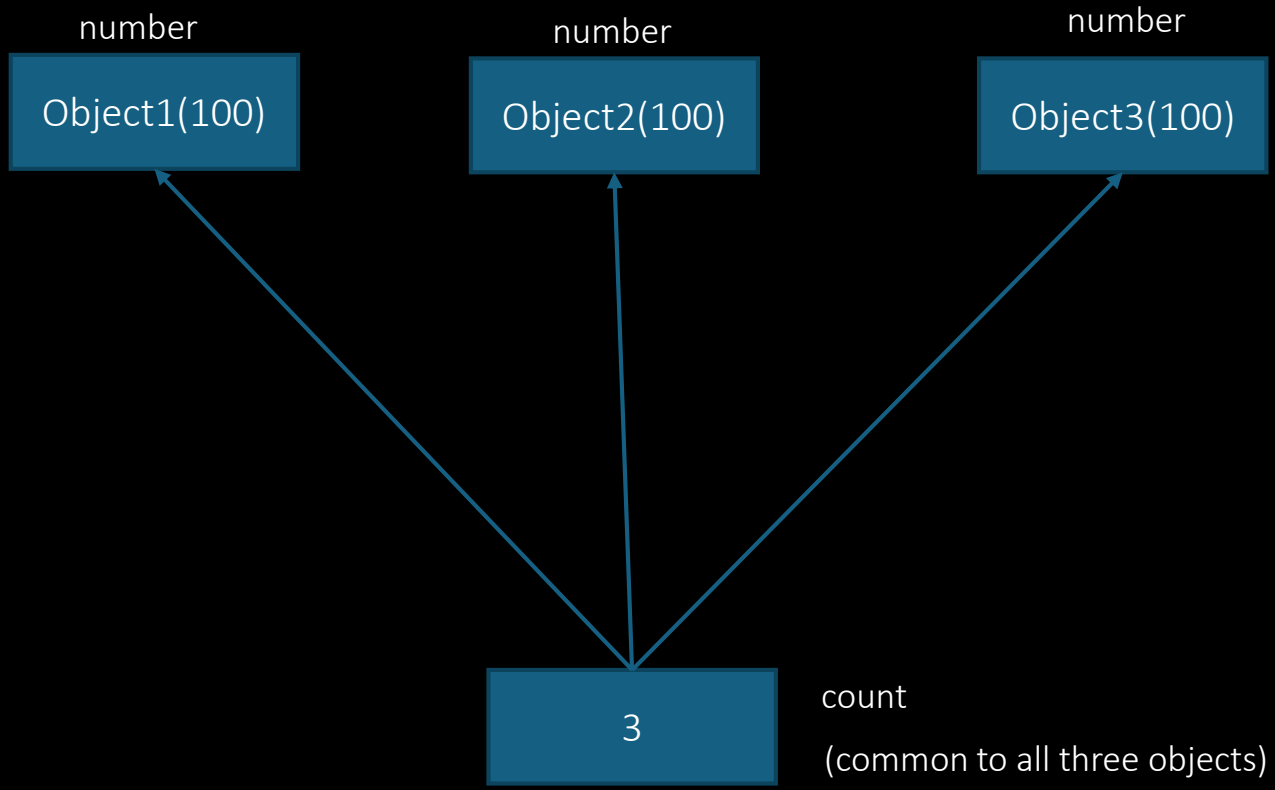
AMBITION GURU

```
int item::count;

int main()
{
     item a,b,c;

     a.getcount();
     b.getcount();
     c.getcount();

     a.getdata(100);
     b.getdata(200);
     c.getdata(300);

     cout<<"After reading data"<<"\n";

     a.getcount();
     b.getcount();
     c.getcount();
     return 0;
}
```

Output:

```
count:0
count:0
count:0
After reading data
count:3
count:3
count:3
```

Static Data Members



Static Data Members

Write a C++ program to display the number of objects created using static member.

Static Member Functions

- Like a static member variable, we can also have static member functions.

Static member function has certain special characteristics:

- A static function can have access to only **other static members (function or variable)** declared in the same class.
- A static member function can be called using the **class name (instead of its object)** as follows:

```
class_name:: function_name();
```

Static Member Functions

AMBITION GURU

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

class test {
    int code;
    static int count;  // static data member

public:
    void setcode() {
        code = ++count;
    }

    void showcode() {
        cout << "object number: " << code << "\n";
    }

    static void showcount() {
        cout << "count: " << count << "\n";
        // cout << code; // ERROR: code is not static
    }
};
```

Static Member Functions

AMBITION GURU

```
// Definition of static member outside the class
int test::count;

int main() {
    test t1, t2;

    t1.setcode();  // count = 1
    t2.setcode();  // count = 2

    test::showcount();  // Output: count: 2

    test t3;
    t3.setcode();       // count = 3

    test::showcount();  // Output: count: 3

    t1.showcode();      // Output: object number: 1
    t2.showcode();      // Output: object number: 2
    t3.showcode();      // Output: object number: 3

    return 0;
}
```

Output

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

Questions to ask:

1. Explain the significance of type conversion. How do we achieve dynamic memory allocation in C++? Explain with an example.
2. When inline functions may not work? Define and write syntax for default arguments. Write a program to display N number of characters by using default arguments for both parameters. Assume that the function takes two arguments, one character to be printed and other how many times the character to be printed respectively.
3. What is the principle reason for using default arguments in the function? Explain how missing arguments and default arguments are handled by the function simultaneously?
4. How object oriented programming differs from object based programming language? Discuss benefits of OOP.

Questions to ask:

AMBITION GURU

5. What is meant by pass by reference? How can we pass arguments by reference by using reference variable? Illustrate with example.
6. Write short notes on the following features of C++: iostream class, C++ Comments, Tokens, Keywords and Identifiers, The const Qualifier, The endl Manipulator
7. **Write a recursive function** to calculate the factorial of a number.
8. **WAP to find whether the number is prime or not using C++**
9. **List down the pros and cons of OOP. Difference between OOP and procedural language.**
10. Explain in brief about the features of OOP.
11. Explain about the structure of a C++ program.

THANK YOU

Any Queries ?