Unit Code & Title	**COM 210 Structured Programming**
Lecturer Name	**DR. Harriet Tsinale**
Telephone	**0725218728**
Course Description	This course introduces students to structured programming concepts using C++. It emphasizes problem-solving techniques, algorithm development, and structured approaches to program design and implementation
Course Objectives	• By the end of this course, students should be able to: • Understand the fundamentals of structured programming and C++ syntax. • Develop algorithms to solve computational problems. • Apply control structures, functions, arrays, and pointers in programming. • Write, compile, debug, and execute C++ programs. • Develop modular programs using structured design principles.

Course Content

Week	Topics	Subtopics & Details	Learning Activities
1–3	Introduction to Programming & C++	- Programming concepts and languages - History and features of C++ - Structure of a C++ program - Data types, variables, constants - Input/Output (cin, cout)	Lectures, simple input/output exercises
4–6	Control Structures	- Operators (arithmetic, relational, logical) - Decision-making (if, if-else, switch) - Loops (for, while, do-while) - Break & continue	Practice with decision and looping structures
7	CAT		
8–9	Functions	- Function declaration/definition - Parameters (value & reference) - Inline functions - Variable scope - Recursion basics	Writing modular programs using functions
10–11	Arrays & Strings	- 1D & 2D arrays - Character arrays - String manipulation functions	Array-based problems, string operations
12	Pointers & References	- Pointer basics & arithmetic - Pointers with arrays & functions	Memory allocation exercises, pointer debugging

13	Structures & User-defined Types	- Structures - Typedef & Enumerations - Intro to classes/objects	Struct-based programs
14	Review & Project	- Course revision - Mini-project using structured programming concepts	Final project & presentation

Assessment Breakdown

Continuous Assessment (Quizzes, Assignments, Mid-Sem Tests): 30%

Practical/Lab Work: 20%

Final Exam: 50%

References

C++ Programming: From Problem Analysis to Program Design – D.S. Malik

Programming in C++ – John Hubbard

C++ Primer – Lippman, Lajoie & Moo

Online C++ documentation (cplusplus.com, cppreference.com)

Teaching Methods

Lectures and tutorials

Hands-on lab sessions

Group assignments and individual exercises

Assessment Methods

Continuous Assessment Tests (CATs): 30%

Quizzes, assignments, tests and practical/lab work:

Final Examination: 70%

Reference Materials

C++ Programming: From Problem Analysis to Program Design by D.S. Malik

Programming in C++ by John Hubbard

C++ Primer by Stanley B. Lippman, José Lajoie, Barbara Moo

Online C++ documentation (cplusplus.com, cppreference.com)

Prepared by Dr. Harriet Tsinale

PROGRAMMING CONCEPTS AND LANGUAGES

Programming is the process of designing and creating a set of instructions (a program) that tells a computer how to perform specific tasks.

These instructions are written using a programming language that a computer can interpret and execute.

Key Concepts

Algorithm:
A step-by-step procedure or formula for solving a problem.
Example: To make tea: Boil water → Add tea leaves → Add milk and sugar → Serve.

Program:
A collection of instructions written in a programming language that tells the computer what to do.

Programming Language:
A formal language consisting of syntax and semantics used to communicate instructions to a computer.

Types of Programming Languages

Machine Language

The lowest-level language.

Uses binary code (0s and 1s).

Directly executed by the CPU.

Difficult to read and maintain.

Assembly Language

Uses mnemonics (e.g., MOV, ADD) instead of binary.

Requires an assembler to convert to machine code.

Easier than machine language but still hardware-specific.

High-Level Languages

Close to human language (e.g., C++, Java, Python).

Portable and easier to learn.

Require a compiler or interpreter.

Fourth-Generation Languages (4GLs)

Higher level of abstraction.

Designed to reduce programming effort.

Examples: SQL, MATLAB.

Programming Paradigms

Procedural (Structured) Programming

Programs are divided into functions/procedures.

Focuses on step-by-step instructions.

Examples: C, Pascal.

Object-Oriented Programming (OOP)

Organizes programs into objects (data + functions).

Supports concepts like inheritance, polymorphism, and encapsulation.

Examples: C++, Java.

Functional Programming

Based on mathematical functions.

Emphasizes immutability and avoids changing states.

Examples: Haskell, Lisp.

Event-Driven Programming

Responds to events such as user clicks, keystrokes, or messages.

Common in GUI and mobile app development.

History and Features of C++

History

1970s:

The C programming language was developed by Dennis Ritchie at Bell Labs.

Provided powerful features for system and application programming.

1979:

Bjarne Stroustrup began working on "C with Classes" to include object-oriented concepts.

1983:

The language was officially named **C++**, symbolizing an increment of C.

1998:

ISO standardized C++ (C++98).

2003–2020:

Modern versions introduced (C++03, C++11, C++14, C++17, C++20) with new features like smart pointers, lambda expressions, auto keyword, concurrency, and modules.

Key Features of C++

Compiled Language: Code must be compiled before execution.

Supports Both Procedural and Object-Oriented Programming: Makes it versatile for different types of applications.

Portable: Runs on various platforms with minimal modification.

Efficient: Provides low-level memory management with high performance.

Rich Standard Library: Includes tools for I/O, algorithms, and data structures.

Modular and Extensible: Supports functions, classes, and templates.

- Compatible with C – Can run most C programs with minimal changes.

STRUCTURE OF A C++ PROGRAM

C++ Program Structure

Below is a simple code that would print the words ***Hello World.***

```
#include <iostream>
#include <conio.h>
using namespace std;
// main() is where program execution begins.
int main()
{
   cout << "Hello World"; // prints Hello World
   getch();
}
```

parts of the above program

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
- **#include <conio.h>** is a header file that is used to pause the output screen.
- The line **using namespace std;** tells the compiler to use the standard namespace. Namespaces are a relatively recent addition to C++. cout, cin and a lot of other things are defined in it. (This means that one way to call them is by **using std**::cout and **std**::cin.)
- The next line '// **main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.
- The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.
- The next line **getch() -** write a character to screen.

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item.

An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers.

C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers −

mohd	zara	abc	move_name	a_123
myname50	_temp	j	a23b9	retVal

C++ Keywords/ Reserved words

The following list shows the reserved words in C++.

These reserved words may not be used as constant or variable or any other identifier names.

Asm	else	new	This
Auto	enum	operator	Throw
Bool	explicit	private	True
Break	export	protected	Try
Case	extern	public	typedef
Catch	false	register	typeid
Char	float	reinterpret_cast	typename
Class	for	return	union
Const	friend	short	unsigned
const_cast	goto	signed	using
Continue	if	sizeof	virtual
Default	inline	static	Void
Delete	int	static_cast	volatile
Do	long	struct	wchar_t
Double	mutable	switch	while
dynamic_cast	namespace	template	

Trigraphs

Trigraph	Replacement

- A few characters have an alternative representation, called a **trigraph sequence**.
- A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.
- Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.
- Following are most frequently used trigraph sequences −
- All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Whitespace in C++

1. A line containing only whitespace, possibly with a comment, is known as a **blank line**, and C++ compiler totally ignores it.

```
2.   int age;
```

the statement above must have at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

3. fruit = apples + oranges;   // Get the total fruit

whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Comments in C++

Program comments are explanatory statements that you can include in the C++ code.

These comments help anyone reading the source code.

All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments.

All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */.

For example −

/* This is a comment */

A comment can also start with //, extending to the end of the line.

Eg

// prints Hello World

C++ Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types .

basic C++ data types −

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Several of the basic types can be modified using one or more of these type modifiers −

signed

unsigned

short

long

```
// program that will show the size of a data type
#include <iostream>
#include <conio.h>
using namespace std;
int main() {
  cout << "Size of char : " << sizeof(char) << endl;
  cout << "Size of int : " << sizeof(int) << endl;
  cout << "Size of short int : " << sizeof(short int) << endl;
  cout << "Size of long int : " << sizeof(long int) << endl;
  cout << "Size of float : " << sizeof(float) << endl;
  cout << "Size of double : " << sizeof(double) << endl;
  getch();
}
```

- **endl** - is used to inserts a new-line character after every line
- **<<** operator - is used to pass multiple values out to the screen.
- **sizeof()** operator - is used to get size of various data types.

C++ Variable Types

- A variable provides us with named storage that our programs can manipulate.
- In C++ variable is used to store data in a memory location, which can be modified or used in the program during program execution.
- Variables are used in C++ where you will need to store any type of values within a program and whose value can be changed during the program execution.
- These variables can be declared in various ways each having different memory requirements and storing capability.
- Variables are the name of memory locations that are allocated by compilers and the allocation is done based on the data type used for declaring the variable.
- Following section will cover how to define, declare and use various types of variables.

❖ A **variable definition** means that the programmer writes some instructions to tell the compiler to create the storage in a memory location syntax

```
data_type variable_name;

data_type variable_name1, variable_name2;
```

```
eg

int a, b, c;
char   letter;
float  area;
double d;
```

- **Variables initialization** - Variables can be initialized and the initial value can be assigned along with their declaration.

syntax

```
data_type variable_name = value;

eg

char   letter='A';
float  area = 26.5;
```

<u>Rules for declaring variables</u>

A variable name can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.

The first character must be a letter or underscore.

Blank spaces cannot be used in variable names.

Special characters like #, $ are not allowed.

C++ keywords cannot be used as variable names.

Variable names are case-sensitive.

A variable name can be consisting of 31 characters only if we declare a variable more than 1 characters compiler will ignore after 31 characters.

Variable type can be bool, char, int, float, double, void or wchar_t.

```
//local variable declaration
#include <iostream>
#include <conio.h>
using namespace std;

int main() {

    int x = 5;
    int y = 2;
    int Result;
    Result = x * y;
    cout << Result;
   getch();
}
```

Variable Scope in C++

Local Variables

They are variables that are declared inside a function or block.

They can be used only by statements that are inside that function or block of code.

Local variables are not known to functions outside their own.

```
#include <iostream>
#include <conio.h>
using namespace std;
// Global Variable declaration:
int x, y;
```

```
float f;
main()
{
  // Local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;
getch();
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program.

The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

```
#include <iostream>
#include <conio.h>
using namespace std;

// Global Variable
declaration
int x, y;
float f;

main()
{
  {
  // Local variable
   int tot;
       x = 10;
       y = 20;
       tot = x + y;
       cout << tot;
     cout << endl;
       }
     f = 70.0 / x ;
     cout << f;
     cout << endl;
   getch();
   }
```

Note: It is a good programming practice to initialize variables properly; otherwise sometimes program would produce unexpected result.

C++ Modifier Types

C++ allows the **char, int,** and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here −

- signed
- unsigned
- long
- short

Constants

Defining Constants

There are two simple ways in C++ to define constants −

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

syntax

```
#define CONTANTNAME value

EG
#define PI 3.142
#define PRINCIPLE 10000
```

Example

```
#include <iostream>
#include <conio.h>
using namespace std;
#define LENGTH 20
#define WIDTH  5
main()
```

```
{
int area;
   area = LENGTH * WIDTH;
   cout << area;
getch();
}
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows

```
const datatype CONSTANTNAME = value;
```

```
const int  LENGTH = 10;
const int  WIDTH  = 5;
```

Example

```
#include <iostream>
#include <conio.h>
using namespace std;

main()
{
      const int  LENGTH = 10;
      const int  WIDTH  = 5;
      int area;
   area = LENGTH * WIDTH;
   cout << area;
getch();
}
```

Note: it is good programming practice to define constants in CAPITALS.

OPERATORS IN C++

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- Operators are used to perform various operations on variables and constants.

Types of operators

1. Assignment Operator
2. Mathematical Operators

3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

– There are following arithmetic operators supported by C++ language −
– Assume variable **A holds 10** and variable **B holds 20**

Operator	Description	Example
+	Addition	A + B will give 30
-	Subtracts	A - B will give -10
*	Multiplication	A * B will give 200
/	Division	B / A will give 2
%	**Modulus** Operator and remainder of after an integer division	B % A will give 0
++	**Increment operator**, increases integer value by one	A++ will give 11
--	**Decrement operator,** decreases integer value by one	A-- will give 9

Relational Operators

– These operators establish a relationship between operands.
– Assume variable **A holds 10** and variable **B holds 20**

Relational operators	description	Example
==	equivalent	(A == B) is not true.
!=	Not equivalent	(A != B) is true.
>	greater than	(A > B) is not true.
<	less than	(A < B) is true.
>=	greater than equal to	(A >= B) is not true.
<=	less than or equal to	(A <= B) is true.

Logical Operators

They are used to combine two different expressions together.

Assume variable **A holds 1** and variable **B holds 0**

Logical Operator	Description	Example
&&	Logical **AND** operator. If both the operands are non-zero, then condition becomes true.	**(A && B)** is false.
||	Logical **OR** Operator. If any of the two operands is non-zero, then condition becomes true.	**(A || B)** is true.
!	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.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation.

They work with only integral data types like `char`, `int` and `long` and not with floating point values.

- Bitwise AND operators **&**
- Bitwise OR operator **|**
- And bitwise XOR operator **^**
- And, bitwise NOT operator **~**

The truth tables for **~ , &**, |, and ^ are as follows −

p	Q	~p	p & q	p | q	p ^ q
0	0	1	0	0	0
0	1	1	0	1	1
1	1	0	1	1	0
1	0	0	0	1	1

Assignment Operators

Operates '=' is used for assignment

It takes the right-hand side (called **rvalue**) and copy it into the left-hand side (called **lvalue**).

Assignment operator is the only operator which can be overloaded but cannot be inherited.

Operator	Description	Example
=	Simple assignment operator	C = A + B will assign value of A + B into C
+=	Add AND assignment operator	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator	
/=	Divide AND assignment operator	
%=	Modulus AND assignment operator	
&=	Bitwise AND assignment operator.	
^=	Bitwise exclusive OR and assignment operator.	
|=	Bitwise inclusive OR and assignment operator.	

Operators Precedence in C++

- Operator precedence determines the grouping of terms in an expression.
- This affects how an expression is evaluated. Certain operators have higher precedence than others
- Example, the multiplication operator has higher precedence than the addition operator

Eg

x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7

C++ Basic Input/Output

C++ I/O occurs in streams, which are sequences of bytes.

If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

I/O Library Header Files

1. #include<iostream>
   Include this file whenever using C++ I/O
2. #include<iomanip>

This file must be included for most C++ manipulators. If you don't know what a manipulator is, don't worry. Just include this file along with `iostream` and you can't go wrong

3. #include<fstream>
   Include this file whenever working with files.

The main header files important to C++ programs is <iostream>

<iostream> - This file defines the following objects

- **cin -** standard input stream
- **cout -** standard output stream
- **cerr -** un-buffered standard error stream
- **clog -** buffered standard error stream The Standard Output Stream (cout)

1. **The Standard Output Stream (cout)**

The cout object is said to be connected to the standard output device, which usually is the display screen.

cout <<

Eg

```
#include <iostream>

using namespace std;

int main()

{

  char str[] = "Hello C++";

  cout << "Value of str is : " << str << endl;

}
```

Output

```
Value of str is : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement and **endl** is used to add a new-line at the end of the line.

2. **The Standard Input Stream (cin)**

The cin object is said to be attached to the standard input device, which usually is the keyboard.

cin >>

```
#include <iostream>
using namespace std;
int main()
{
  char name[50];
  cout << "Please enter your name: ";
  cin >> name;
  cout << "Your name is: " << name << endl;
}
```

Output

```
Please enter your name: Harriet
Your name is: Harriet
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement.

eg

```
cin >> name >> age;
```

This is the same as the following two statements −

```
cin >> name;
cin >> age;
```

The Standard Error Stream (cerr)

The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

cerr <<

```
#include <iostream>

using namespace std;

int main() {

   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;

}
```

Output

```
Error message : Unable to read....
```

3. **The Standard Log Stream (clog)**

The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered.

This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

clog <<

```
#include <iostream>

using namespace std;

int main() {

   char str[] = "Unable to read....";


   clog << "Error message : " << str << endl;

}
```

Output

Error message: Unable to read....

NOTE

You would not be able to see any difference in **cout, cerr and clog** with these small examples, but while writing and executing big programs the difference becomes obvious.

So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

Example 1

```
//Print Number Entered by User
#include <iostream>
using namespace std;
int main()
{
   int number;
   cout << "Enter an integer: ";
   cin >> number;
   cout << "You entered " << number;
   return 0;
}
```

Example 2

//Use cin and cout to Display Number and Text Input by User

```
#include <iostream>
#include <string>
```

```
using namespace std;
int main()
{
int InputNumber;                              // Declare a variable to store
an integer
cout << "Enter an integer: ";
cin >> InputNumber;                   // store integer given user input

cout << "Enter your name: ";          // The same with text i.e. string data

string InputName;
cin >> InputName;
cout << InputName << " entered " << InputNumber << endl;
return 0;
}
```

Example 3

```
//program to find area and circumference
#include <iostream>
using namespace std;
int main()
{
   float radius, circumference, area;                // Declare 3 floating-point
variables
   const float PI = 3.14;                                    // Declare and
define PI

   cout << "Enter the radius: ";                     // Prompting message
   cin >> radius;                                            // Read input into
variable radius

  // Compute area and circumference
   area = PI * radius * radius;
   circumference = PI * 2 * radius;

  // Print the results
   cout << "The radius is: " << radius << endl;
   cout << "The area is: " << area << endl;
   cout << "The circumference is: " << circumference << endl;

   return 0;
}
```

Example 4

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

```
int main(int argc, char *argv[])
{
   cout<<"*****\n";
   cout<<"*****\n";
   cout<<"*****\n";
   cout<<"*****\n";
   cout<<"*****\n";
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

CLRSCR() AND GETCH() IN C++

clrscr() and getch() both are predefined function in "conio.h" (console input output header file).

Clrscr()

- It is a predefined function in "conio.h" (console input output header file) used to clear the console screen.
- It is a predefined function, by using this function we can clear the data from console (Monitor).
- Using of clrscr() is always optional but it should be place after variable or function declaration only.

Example of clrscr()

```
#include<iostream.h>
#include<conio.h>

void main()
{
        int a=10, b=20;
        int sum=0;
        clrscr();                // use clrscr() after variable declaration
        sum=a+b;
        cout<<"Sum: "<<sum;
getch();
}
```

Output

```
Sum: 30
```

Getch()

- It is a predefined function in "conio.h" (console input output header file) will tell to the console wait for some time until a key is hit given after running of program.

- By using this function we can read a character directly from the keyboard.
- Generally getch() are placing at end of the program after printing the output on screen.

Example of getch()

```
#include<iostream.h>
#include<conio.h>

void main()
{
        int a=10, b=20;
        int sum=0;
        clrscr();
        sum=a+b;
        cout<<"Sum: "<<sum;
getch();                        // use getch() befor end of main()
}
```

Output

Sum: 30

DECISION MAKING

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

C++ handles decision-making by supporting the following statements,

- *if* statement
- *switch* statement
- conditional operator statement
- *goto* statement

Decision making with *if* statement

The *if* statement may be implemented in different forms depending on the complexity of conditions to be tested. V

The different forms are,

- Simple *if* statement
- *If....else* statement
- Nested *if....else* statement
- *else if* statement

1. Simple *if* statement

```
if( expression )

{

 statement-inside;

}

 statement-outside;
```

- The `if` statement evaluates the test expression inside parenthesis.
- If test expression is evaluated to true, statements inside the body of `if` is executed.
- If test expression is evaluated to false, statements inside the body of `if` is skipped.

Example 1

```
#include< iostream.h>

#include< conio.h>

using namespace std;

int main( )

{

 int x,y;

 x=15;

 y=13;

 if (x > y )

 {

  cout << "x is greater than y";

 }

getch();

}
```

Output :

```
x is greater than y
```

Example 2

```
#include <iostream>
using namespace std;
int main()
{
    int number;
    cout << "Enter an integer: ";
    cin >> number;

    if ( number > 0)
    {
   cout << "You entered a positive integer: " << number << endl;
    }
    cout << " You've  entered an integer";

    return 0;
}
```

2. *if...else* statement

```
if( expression )
{
 statement-block1;
}
else
```

```
{
 statement-block2;
}
```

If the 'expression' is true, the 'statement-block1' is executed, else 'statement-block1' is skipped and 'statement-block2' is executed.

Example 1

```
#include <iostream>
#include<conio.h>
using namespace std;
int main( )
{
 int x,y;
 x=15;
 y=18;
 if (x > y )
 {
  cout << "x is greater than y";
 }
 else
 {
  cout << "y is greater than x";
 }
getch();
```

}

Output :

y is greater than x

Example 2

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

int main()
{
int number;
cout << "Enter an integer: ";
cin >> number;

if ( number >= 0)
   {
    cout << "You entered a positive integer: " << number << endl;
   }

 else
   {
    cout << "You entered a negative integer: " << number << endl;
   }
```

```
  cout << "This line is always printed.";
  return 0;
}
```

Example 3

//check whether a number is ood or even

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

Example 4

```
#include <iostream>
int main()
{
        using namespace std;
        int age;
        cout << "Enter your age" << endl;
        cin >> age;
        if(age >= 18)
        {
                cout << "You are 18+" << endl;
                cout << "Eligible to vote" << endl;
        }
        else
        {
                cout << "You are not yet 18" << endl;
                cout << "Not eligible to vote" << endl;
        }
        return 0;
}
```

3. Nested *if....else* statement

```
if( expression1 )
{
  if( expression2 )
        {
                statement-block1;
        }
 else
        {
                statement-block2;
        }
}
else
{
  statement-block3;
}
```

if 'expression1' is **false** the 'statement-block3' will be executed

if 'expression1' is **true** then it continues to perform the test for 'expression 2' .

If the 'expression 2' is **true** the 'statement-block1' is executed otherwise 'statement-block2' is executed.

Example 1

```
#include <iostream>
#include <conio.h>
using namespace std;
void main()
{

int a,b,c;

cout << "enter 3 number";

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";

        }

  }

getch();

}
```

Example 2

```
#include <iostream>
int main()
{
        using namespace std;
        int x = 4, y = 3, z = 5;
        if(z > x)
        {
                if(z > y)
                {
                        cout << "z is the greatest number" << endl;
                }
        }
        return 0;
}
```

Example 3

```
#include <iostream>
int main()
{
        using namespace std;
        int x = 4, y = 3, z = 5;
        if(x > y)
        {
                if(x > z)
                        cout << "x is the greatest integer";
                else
                        cout << "x is not the greatest integer";
        }
        else
                cout << "x is not the greatest integer";
        return 0;
}
```

Example 4

```
#include <iostream>
using namespace std;
int main()
{
      int x = 4, y = 3, z = 5;
      if( (x > y) && (x > z))
            cout << "x is the greatest integer";
      else
            cout << "x is not the greatest integer";
      return 0;
}
```

else-if ladder

```
if(expression 1)
        {
        statement-block1;
        }
else if(expression 2)
        {
        statement-block2;
        }
else if(expression 3 )
        {
        statement-block3;
        }
else
        default-statement;
```

The expression is tested from the top(of the ladder) downwards.

As soon as the true condition is found, the statement associated with it is executed.

Example :

```
#include <iostream>
#include <conio.h>
using namespace std;
void main( )

{

int a;

cout << "enter a number";

cin >> a;

if( a%5==0 && a%8==0)

        {

          cout << "divisible by both 5 and 8";

        }

else if( a%8==0 )

        {

          cout << "divisible by 8";

        }

else if(a%5==0)

        {

          cout << "divisible by 5";

        }

else

        {

          cout << "divisible by none";

        }

getch();

}
```

Other examples

Example 1

//Find Largest Number Using if Statement

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

int main()
{
    float n1, n2, n3;

    cout << "Enter three numbers: ";
    cin >> n1 >> n2 >> n3;

    if(n1 >= n2 && n1 >= n3)
    {
        cout << "Largest number: " << n1;
    }

    if(n2 >= n1 && n2 >= n3)
    {
        cout << "Largest number: " << n2;
    }

    if(n3 >= n1 && n3 >= n2) {
        cout << "Largest number: " << n3;
    }

    return 0;
}
```

Example 2

// Find Largest Number Using if...else Statement

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

int main()
{
    float n1, n2, n3;

    cout << "Enter three numbers: ";
    cin >> n1 >> n2 >> n3;

    if((n1 >= n2) && (n1 >= n3))
        cout << "Largest number: " << n1;
    else if ((n2 >= n1) && (n2 >= n3))
        cout << "Largest number: " << n2;
    else
        cout << "Largest number: " << n3;

    return 0;
}
```

Example 3:

//Find Largest Number Using Nested if...else statement

```
#include <iostream>
```

```
using namespace std;

int main()
{
    float n1, n2, n3;

    cout << "Enter three numbers: ";
    cin >> n1 >> n2 >> n3;

    if (n1 >= n2)
    {
        if (n1 >= n3)
            cout << "Largest number: " << n1;
        else
            cout << "Largest number: " << n3;
    }
    else
    {
        if (n2 >= n3)
            cout << "Largest number: " << n2;
        else
            cout << "Largest number: " << n3;
    }

    return 0;
}
```

Example 4

```
//Find Smallest of three Numbers in C++
#include <iostream>
using namespace std;

int main() {

  int a, b, c;

  cout << "Enter three numbers \n";
  cin >> a >> b >> c;

  if (a < b && a < c)
        {
        cout << "Smallest number is " <<
a;
}
        }
  else if (b < a && b < c)
        {
         cout << "Smallest number is " <<
b;

         }
Else
         {
        cout << "Smallest number is "<< c;

         }
    return 0;
```

C++ switch statement

– A **switch** statement allows a variable to be tested for equality against a list of values.

– Each value is called a **case**, and the variable being switched on is checked for each case.

Syntax

```
switch(expression)
{
   case constant-expression  :
      statement(s);
      break;
   case constant-expression  :
      statement(s);
      break;

   // you can have any number of case
statements.
   default :       //Optional
      statement(s);
}
```

Rules for apply switch

- The switch expression must be of integer or character type.
- With switch statement use only byte, short, int, char data type.
- With switch statement not use float data type.
- You can use any number of case statements within a switch.
- The case value can be used only inside the switch statement.

- Value for a case must be same as the variable in switch.

Example 1

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

int main ()
{
  // local variable declaration:
  char grade = 'D';

  switch(grade)
          {
            case 'A' :
          cout << "Excellent!" << endl;
          break;
            case 'B' :
            case 'C' :
          cout << "Well done" << endl;
           break;
           case 'D' :
          cout << "You passed" << endl;
          break;
           case 'F' :
          cout << "Better try again" << endl;
          break;
           default :
          cout << "Invalid grade" << endl;
          }
  cout << "Your grade is " << grade <<
endl;

  return 0;
}
```

This would produce the following result −

```
You passed
Your grade is D
```

Example 2

```
// Program to built a simple calculator using switch Statement

#include <iostream>
using namespace std;

int main()
{
    char a;
    float num1, num2;

    cout << "Enter an operator (+, -, *, /): ";
    cin >> a;

    cout << "Enter two operands: ";
    cin >> num1 >> num2;

    switch (a)
    {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1+num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1-num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1*num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1/num2;
            break;
        default:
            // operator doesn't match any case constant (+, -, *, /)
            cout << "Error! operator is not correct";
            break;
    }

    return 0;
}
```

Example 3

//Simple Calculator using switch statement

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

int main()
{
   char a;
   float num1, num2;

   cout << "Enter operator either + or - or * or /: ";
   cin >> a;
   cout << "Enter two operands: ";
   cin >> num1 >> num2;

   switch(a)
   {
      case '+':
         cout << num1+num2;
         break;
      case '-':
         cout << num1-num2;
         break;
      case '*':
         cout << num1*num2;
         break;
      case '/':
         cout << num1/num2;
         break;
```

```
    default:
      cout << "Error! operator is not correct";
      break;
  }
```

Example 3

```
#include <iostream>
using namespace std;
int main(){
  int num=5;
  switch(num+2) {
    case 1:
      cout<<"Case1: Value is: "<<num<<endl;
    case 2:
      cout<<"Case2: Value is: "<<num<<endl;
    case 3:
      cout<<"Case3: Value is: "<<num<<endl;
    default:
      cout<<"Default: Value is: "<<num<<endl;
  }
  return 0;
}
```

Example 4

```
#include<iostream.h>
#include<conio.h>

void main()
{
int ch;
clrscr();
cout<<"Enter any number (1 to 7)";
cin>>ch;
switch(ch)
{
case  1:
cout<<"Today is Monday";
break;

case  2:
cout<<"Today is Tuesday";
break;
case  3:
cout<<"Today is Wednesday";
break;

case  4:
cout<<"Today is Thursday";
break;

case  5:
cout<<"Today is Friday";
break;

case  6:
cout<<"Today is Saturday";
break;

case  7:
cout<<"Today is Sunday";
break;

default:
cout<<"Only enter value 1 to 7";
}
getch();
}
```

LOOPING

loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

A sequence of statement is executed until a specified condition is true.

This sequence of statement to be executed is kept inside the curly braces `{ }` known as **loop body**.

After every execution of loop body, condition is checked, and if it is found to be **true** the loop body is executed again. When condition check comes out to be **false**, the loop body will not be executed.

There are 3 type of loops in C++ language

1. *while* loop
2. *for* loop
3. do-while loop

while loop

while loop can be address as an **entry control** loop.

It is completed in 3 steps.

- Variable initialization.(e.g int x=0;)
- condition(e.g while(x<=10))
- Variable increment or decrement (x++ or x-- or x=x+2)

Syntax :

```
variable initialization ;
while (condition)
{
 statements ;
 variable increment or decrement ;
}
```

How while loop works?

- The while loop evaluates the test expression.
- If the test expression is true, codes inside the body of while loop is evaluated.
- Then, the test expression is evaluated again. This process goes on until the test expression is false.
- When the test expression is false, while loop is terminated.

Example 1

```
#include <iostream>
using namespace std;
int main()
{
   int i=1;
   while(i<=6)
    {
          cout<<"Value of variable i 
is: "<<i<<endl;
           i++;
    }
}
```

Example 2

```
#include <iostream>
using namespace std;
int main(){
   int i=1;
while(i<=6)
{
      cout<<"Value of variable i is: "<<i<<endl;
         i--;
   }
}
```

Example 3

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

int main ()
{

   int a = 10;
   while( a < 20 )
       {
        cout << "value of a: " << a << endl;
        a++;
       }

   return 0;
}
```

Example 3

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

```
int a;
cout << "Enter the Number :";
cin>>a;

int counter = 1;
while (counter <= a)
        {
        cout << "Execute While " << counter << " time" << endl;
        counter++;
        }
getch();
return 0; }
```

Example 4

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i;
clrscr();
i=1;
while(i<5)
{
cout<<endl<<i;
i++;
}
getch();
}
```

For loop

For loop is used to execute a set of statement repeatedly until a particular condition is satisfied.

We can say it an **open ended loop.**

For loop contains 3 parts.

- Initialization
- Condition
- Iteration

syntax

```
for(initialization; condition ; increment/decrement)
{
  statement-block;
}
```

Example 1

```
#include<iostream.h>
#include<conio.h>

void main()
{
     int i;
     for(i=1;i<5;i++)
{
cout<<endl<<i;
}
getch();
}
```

Example 2

`

```
#include <iostream>
```

```
using namespace std;
int main(){

    int n;
    for( n = 1; n <= 10; n++ ){
        cout << n << endl;
    }
    return 0;
}
```

Example 3

```
#include <iostream>
int main(){

        using namespace std;
        int sum = 0, i, n;

        for(i = 0; i < 10; i++){

                cout << "Enter number" << endl;
                cin >> n;

                sum = sum + n;

        }
        cout << "Sum is " << sum << endl;

        return 0;

}
```

Nested for loop

We can also have nested **for** loop, i.e one **for** loop inside another **for** loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
  for(initialization; condition; increment/decrement)
     {
        statement;
        }
}
```

Example 1

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

        int i;
        int j;

        for(i = 12; i <= 14; i++){
                cout << "Table of " << i << endl;

                for(j = 1; j <= 10; j++)
                {
                cout << i << "*" << j << "=" << (i*j) << endl;
                }
        }
        return 0;
}
```

Example 2

// Program to print half pyramid using *

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << "\n";
    }
    return 0;
}
```

Example 3

//Program to print half pyramid a using numbers

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << j << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

Example 4

//Program to print half pyramid using alphabets

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

int main()
{
    char input, alphabet = 'A';

    cout << "Enter the uppercase character you want to print in the last row: 
";
    cin >> input;

    for(int i = 1; i <= (input-'A'+1); ++i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << alphabet << " ";
        }
        ++alphabet;

        cout << endl;
    }
    return 0;
}
```

Example 5

Programs to print inverted half pyramid using * and numbers

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = rows; i >= 1; --i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

Example 6

//Inverted half pyramid using numbers

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

int main()
{
    int rows;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = rows; i >= 1; --i)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << j << " ";
        }
        cout << endl;
    }
```

```
     return 0;
}
```

Example7

//Program to print full pyramid using

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

int main()
{
    int space, rows;

    cout <<"Enter number of rows: ";
    cin >> rows;

    for(int i = 1, k = 0; i <= rows; ++i, k = 0)
    {
        for(space = 1; space <= rows-i; ++space)
        {
            cout <<"  ";
        }

        while(k != 2*i-1)
        {
            cout << "* ";
            ++k;
        }
        cout << endl;
    }
    return 0;
}
```

Example 8

Program to print pyramid using numbers

```
using namespace std;

int main()
{
    int rows, count = 0, count1 = 0, k = 0;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; ++i)
    {
        for(int space = 1; space <= rows-i; ++space)
        {
            cout << "  ";
            ++count;
        }

        while(k != 2*i-1)
        {
            if (count <= rows-1)
            {
                cout << i+k << " ";
                ++count;
            }
            else
            {
                ++count1;
                cout << i+k-2*count1 << " ";
            }
            ++k;
        }
        count1 = count = k = 0;

        cout << endl;
    }
    return 0;
}
```

Example 9

//Print Floyd's Triangle.

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

#include <iostream>
using namespace std;

int main()
{
    int rows, number = 1;

    cout << "Enter number of rows: ";
    cin >> rows;

    for(int i = 1; i <= rows; i++)
    {
        for(int j = 1; j <= i; ++j)
        {
            cout << number << " ";
            ++number;
        }

        cout << endl;
    }

    return 0;
}
```

Do while loop

In some situations it is necessary to execute body of the loop before testing the condition.

Such situations can be handled with the help of **do-while** loop.

Do statement evaluates the body of the loop first and at the end, the condition is checked using **while** statement.

```
Variable initialization;
do{
   statement(s)
}
while( condition );
```

Example 1

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

Jumping out of loop

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes true, that is jump out of loop. C++ language allows jumping from one statement to another within a loop as well as jumping out of the loop.

1) break statement

When **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

```
#include <iostream>

using namespace std;

int main()

{

    for (int i = 1; i <= 10; i++)

       {

          if (i == 5)

          {

             break;

          }

     cout<<i<<"\n";

       }

}
```

Output:

```
1

2

3

4
```

```
#include <iostream>
using namespace std;
int main()
{
   for(int i=1;i<=3;i++){
         for(int j=1;j<=3;j++){
            if(i==2&&j==2){
               break;
                  }
               cout<<i<<" "<<j<<"\n";
               }
        }
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

2) Continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.

```
#include <iostream>

using namespace std;

int main()

{

    for(int i=1;i<=10;i++){

            if(i==5){

                continue;

            }

            cout<<i<<"\n";

        }

}
```

Output:

```
1

2

3

4

6

7

8

9

10
```

Example

```
#include <iostream>

using namespace std;

int main()
```

```
{
 for(int i=1;i<=4;i++){
            for(int j=1;j<=3;j++){
             if(i==2&&j==3){
                continue;
                        }
                cout<<i<<" "<<j<<"\n";
                    }
            }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

Let's see the simple example of goto statement in C++.

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

```
{
ineligible:
        cout<<"You are not eligible to vote!\n";
    cout<<"Enter your age:\n";
    int age;
    cin>>age;
    if (age < 18){
        goto ineligible;
    }
    else
    {
        cout<<"You are eligible to vote!";
    }
}
```

Output:

```
You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
22
You are eligible to vote!
```

FIBONACCCI SERIES IN C++ WITHOUT RECURSION

Let's see the fibonacci series program in C++ without recursion.

```
#include <iostream>
```

```
using namespace std;
int main() {
int n1=0, n2=1, n3, i, number;
cout<<"Enter the number of elements: ";
cin>>number;
cout<<n1<<" "<<n2<<" "; //printing 0 and 1
for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed
{
n3=n1+n2;
cout<<n3<<" ";
n1=n2;
n2=n3;
}
return 0;
}
```

```
0 1 1 2 3 5 8 13 21 34
```

FACTORIAL PROGRAM IN C++

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

```
int main()
{
   int i,fact=1,number;
  cout<<"Enter any Number: ";
 cin>>number;
  for(i=1;i<=number;i++){
      fact=fact*i;
  }
  cout<<"Factorial of " <<number<<" is: "<<fact<<endl;
  return 0;
}
```

Output:

```
Enter any Number: 5
Factorial of 5 is: 120
```

C++ PROGRAM TO REVERSE NUMBER

We can reverse a number in C++ using loop and arithmetic operators. In this program, we are getting number as input from the user and reversing that number.

```
#include <iostream>

using namespace std;

int main()

{

int n, reverse=0, rem;

cout<<"Enter a number: ";

cin>>n;

  while(n!=0)

  {

     rem=n%10;

     reverse=reverse*10+rem;

     n/=10;

  }

 cout<<"Reversed Number: "<<reverse<<endl;

return 0;

}
```

Output:

```
Enter a number: 234
Reversed Number: 432
```

MATRIX MULTIPLICATION IN C++

We can add, subtract, multiply and divide 2 matrices.

To do so, we are taking input from the user for row number, column number, first matrix elements and second matrix elements. Then we are performing multiplication on the matrices entered by the user.

In matrix multiplication first matrix one row element is multiplied by second matrix all column elements.

Let's try to understand the matrix multiplication of **3*3 and 3*3** matrices by the figure given

below:

Let's see the program of matrix multiplication in C++.

```
#include <iostream>

using namespace std;

int main()

{

int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;

cout<<"enter the number of row=";

cin>>r;

cout<<"enter the number of column=";

cin>>c;

cout<<"enter the first matrix element=\n";

for(i=0;i<r;i++)

{

for(j=0;j<c;j++)

{

cin>>a[i][j];

}

}

cout<<"enter the second matrix element=\n";
```

```
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
cin>>b[i][j];
}
}
cout<<"multiply of the matrix=\n";
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
cout<<mul[i][j]<<" ";
}
cout<<"\n";
}
```

return 0;

}

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 2 3
1 2 3
1 2 3
enter the second matrix element=
1 1 1
2 1 2
3 2 1
 multiply of the matrix=
14 9 8
14 9 8
14 9 8
```

PALINDROME PROGRAM IN C++

A **palindrome number** is a number that is same after reverse.

For example 121, 34543, 343, 131, 48984 are the palindrome numbers.

Palindrome number algorithm

- Get the number from user
- Hold the number in temporary variable
- Reverse the number
- Compare the temporary number with reversed number
- If both numbers are same, print palindrome number
- Else print not palindrome number

Let's see the palindrome program in C++. In this program, we will get an input from the user and check whether number is palindrome or not.

#include <iostream>

```
using namespace std;
int main()
{
  int n,r,sum=0,temp;
  cout<<"Enter the Number=";
  cin>>n;
 temp=n;
 while(n>0)
{
 r=n%10;
 sum=(sum*10)+r;
 n=n/10;
}
if(temp==sum)
cout<<"Number is Palindrome.";
else
cout<<"Number is not Palindrome.";
  return 0;
}
```

Output:

```
Enter the Number=121
Number is Palindrome.
Enter the number=113
Number is not Palindrome.
```

C++ PROGRAM TO PRINT ALPHABET TRIANGLE

There are different triangles that can be printed. Triangles can be generated by alphabets or numbers. In this C++ program, we are going to print alphabet triangles.

Let's see the C++ example to print alphabet triangle.

```
#include <iostream>
using namespace std;
int main()
{
 char ch='A';
    int i, j, k, m;
    for(i=1;i<=5;i++)
    {
        for(j=5;j>=i;j--)
            cout<<" ";
        for(k=1;k<=i;k++)
            cout<<ch++;
            ch--;
        for(m=1;m<i;m++)
            cout<<--ch;
        cout<<"\n";
        ch='A';
    }
return 0;
}
```

Output:

```
     A
    ABA
   ABCBA
  ABCDCBA
 ABCDEDCBA
```

C++ PROGRAM TO PRINT NUMBER TRIANGLE

Like alphabet triangle, we can write the C++ program to print the number triangle. The number triangle can be printed in different ways.

Let's see the C++ example to print number triangle.

```
#include <iostream>

using namespace std;

int main()

{

int i,j,k,l,n;

cout<<"Enter the Range=";

cin>>n;

for(i=1;i<=n;i++)

{

for(j=1;j<=n-i;j++)

{

cout<<" ";

}

for(k=1;k<=i;k++)

{

cout<<k;

}

for(l=i-1;l>=1;l--)

{

cout<<l;

}

cout<<"\n";
}
return 0;
}
```

Output:

```
Enter the Range=5
     1
    121
   12321
  1234321
 123454321
```

C++ FUNCTIONS

- The function in C++ language is also known as **procedure or subroutine** in other programming languages.
- To perform any task, we can create function.
- A function can be **called many times**.
- It provides modularity and code reusability.

Advantage of functions in C++

1) Code Reusability

By creating functions in C++, you can **call it many times**. So we don't need to **write the same code again and again.**

2) Code optimization

It makes the code optimized, we **don't need to write much code**.

Types of Functions

There are two types of functions in C++ programming:

1. Library Functions: are the functions which are **declared in the C++ header files** such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

Declaration of a function

The syntax of creating function in C++ language is given below:

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

Example

```
#include <iostream>
using namespace std;
void func() {
  static int i=0; //static variable
  int j=0; //local variable
  i++;
  j++;
  cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

CALL BY VALUE AND CALL BY REFERENCE IN C++

– There are two ways to pass value or data to function in C++ language

1. call by value
2. call by reference.

– In call by value the Original value is not modified but in call by reference it is modified

Call by value in C++

– In call by value, **original value is not modified.**
– In call by value, value being passed to the function is locally stored by the function parameter in stack memory location.
– If you change the value of function parameter, it is changed for the current function only.
– It will not change the value of variable inside the caller method such as main().

```
#include <iostream>

using namespace std;

void change(int data);

int main()

{

int data = 3;

change(data);
cout << "Value of the data is: " << data<< endl;

return 0;

}

void change(int data)

{

data = 5;

}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

- In call by reference, original value is modified because we pass reference (address).
- address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ RECURSION

- Is when function is called within the same function
- Is a function which calls the same function
- A function that calls itself, and doesn't perform any task after function call, is known as tail recursion.
- In tail recursion, we generally call the same function with return statement.

Syntax

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

C++ Recursion Example

```
// print factorial number using recursion
#include<iostream>
using namespace std;
int main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

C++ STORAGE CLASSES

- Storage class is **used to define the lifetime and visibility of a variable and/or function within a C++ program.**
- **Lifetime** refers to **the period during which the variable remains active**
- **visibility** refers to **the module of a program in which the variable is accessible**.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

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

Automatic Storage Class

- It is the **default storage class for all local variables**.
- The **auto keyword is applied to all local variables automatically**.

```
{
auto int y;
float y = 3.45;
}
```

- The above example defines **two variables with a same storage class**, **auto can only be used within functions.**

Register Storage Class

- The register variable **allocates memory in register than RAM**.
- Its **size is same of register size**.
- It has a **faster access than other variables**.
- It is **recommended to use register variable only for quick access such as in counter.**

Note: We can't get the address of register variable.

```
register int counter=0;
```

Static Storage Class

- The **static variable is initialized only once and exists till the end of a program**.
- It **retains its value between multiple functions call**.
- The static variable has the **default value 0 which is provided by compiler**.

```
#include <iostream>

using namespace std;

void func() {

  static int i=0; //static variable

  int j=0; //local variable

  i++;

  j++;

  cout<<"i=" << i<<" and j=" <<j<<endl;

}

int main()

{

 func();

 func();

 func();

}
```

Output:

```
i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1
```

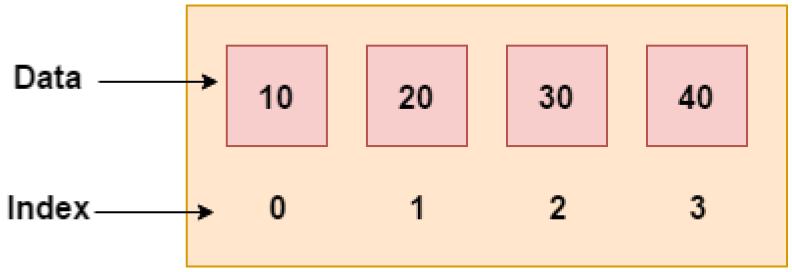
External Storage Class

- The extern variable **is visible to all the programs.**
- It is used **if two or more files are sharing same variable or function**.

extern int counter=0;

C++ ARRAYS

- An array in C++ is a **group of similar types of elements that have contiguous memory location.**
- In C++ **std::array** is a **container that encapsulates fixed size arrays**.
- In C++, **array index starts from 0**.
- We can **store only fixed set of elements in C++ array**.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

– There are 2 types of arrays in C++ programming:

1.Single Dimensional Array
2.Multidimensional Array

C++ Single Dimensional Array

// simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>

using namespace std;

int main()

{

 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array

        //traversing array

        for (int i = 0; i < 5; i++)

        {

            cout<<arr[i]<<"\n";

        }

}
```

Output:

```
10
0
20
0
30
```

C++ Array Example: Traversal using foreach loop

// traverse the array elements using foreach loop. It returns array element one by one.

```
#include <iostream>

using namespace std;

int main()

{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
        //traversing array
       for (int i: arr)
        {
            cout<<i<<"\n";
        }
}
```

Output:

```
10
20
30
40
50
```

C++ Multidimensional Arrays

- The multidimensional array is also known as **rectangular arrays in C++**.
- It **can be two dimensional or three dimensional**.
- The **data is stored in tabular form (row ∗ column)** which is also known as **matrix.**

C++ Multidimensional Array Example

// simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

a_{00}	b_{01}	c_{02}
d_{10}	e_{11}	f_{12}
g_{20}	h_{21}	i_{22}

```
include <iostream>
using namespace std;
int main()
{
int test[3][3];  //declaration of 2D array
test[0][0]=5;  //initialization
test[0][1]=10;
test[1][1]=15;
test[1][2]=20;
test[2][0]=30;
test[2][2]=10;
```

```
//traversal
for(int i = 0; i < 3; ++i)
{
for(int j = 0; j < 3; ++j)
{
cout<< test[i][j]<<" ";
}
cout<<"\n"; //new line at each row
}
return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Multidimensional Array Example: Declaration and initialization at same time

// simple example of multidimensional array which initializes array at the time of declaration.

```
#include <iostream>
using namespace std;
int main()
{
 int test[3][3] =
   {
      {2, 5, 5},
      {4, 0, 3},
      {9, 1, 8}  };  //declaration and initialization
   //traversal
   for(int i = 0; i < 3; ++i)
   {
      for(int j = 0; j < 3; ++j)
      {
         cout<< test[i][j]<<" ";
      }
```

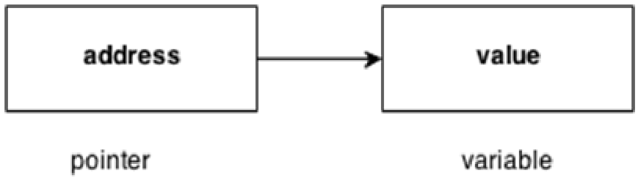
```
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:

```
2 5 5
4 0 3
9 1 8
```

C++ POINTERS

- The pointer in C++ language **is a variable**,
- It is also known as **locator or indicator that points to an address of a value**.



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to **retrieving strings, trees** etc. and **used with arrays, structures and functions**.

2) We can **return multiple values from function** using pointer.

3) It makes **you able to access any memory location in the computer's memory**.

Usage of pointer

1. **Dynamic memory allocation**

- We can dynamically **allocate memory using malloc() and calloc() functions where pointer is used.**

2. **Arrays, Functions and Structures**

– used in arrays, functions and structures. It **reduces the code and improves the performance.**

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
∗ (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

– The pointer in C++ language can be **declared using ∗ (asterisk symbol).**

```
int ∗  a; //pointer to int
char ∗  c; //pointer to char
```

Example 1

```
// example of using pointers printing the address and value.
#include <iostream>
using namespace std;
int main()
{
int number=30;
int ∗   p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Output

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Example 2

```
//Pointer Program to swap 2 numbers without using 3rd variable

#include <iostream>

using namespace std;

int main()

{

int a=20,b=10,*p1=&a,*p2=&b;

cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;

*p1=*p1+*p2;

*p2=*p1-*p2;

*p1=*p1-*p2;

cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;

  return 0;

}
```

Output

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

Structures in C++

A **structure** is a user-defined data type that groups related variables of different types under one name.

Syntax:

```
struct Student {

   int id;

   char name[50];

   float marks;

};
```

Key Points:

Members can be of different data types.

Memory is allocated for each member individually.

Access members using the dot operator (`.`).

Example:

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

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

int main() {
   Student s1 = {101, "Alice", 89.5};
   cout << "ID: " << s1.id << "\nName: " << s1.name << "\nMarks: " << s1.marks;
   return 0;
}
```

Typedef and Enumerations

Typedef

Used to create an alias (new name) for an existing data type.

Improves readability of the code.

Syntax:

```
typedef unsigned int uint;
uint age = 25;
```

Enumerations (enum)

Used to define a set of named integral constants.

Makes the program more readable and manageable.

Syntax:

enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

Day today = Monday;

By default:

Sunday = 0, Monday = 1, … Saturday = 6.

Values can be manually assigned.

enum Month { Jan = 1, Feb, Mar, Apr }; // Feb = 2, Mar = 3, Apr = 4

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING WITH C++

- The purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming language.
- The core of OOP is to create an object, in code, that has certain **properties/attributes and methods**.
- C++ modules try to see whole world in the form of objects.
  Eg
  A **car is an object** which has certain **properties** such as color, number of doors, and the like. It also has certain **methods** such as accelerate, brake, and so on.

Principle concepts that form the foundation of object-oriented programming

Object

- It is the basic unit of object oriented programming.
- It is an instant of a class.
- Objects have states and behaviors

Class

- Is a **blueprint for an object.**
- is a template/**blueprint that describes the behaviors**/states that an object of its type support.

- This doesn't actually define any data, but it does define
  i. What the class name means, that is, what an object of the class will consist of
  ii. What operations can be performed on such an object.

Methods - A method is basically a behavior.

- A class can contain many methods.
- It is in methods where the logics are written, data is manipulated and all the actions are executed.

Abstraction

- Data abstraction - is **providing only essential information to the outside world and hiding their background details**, i.e., to represent the needed information in program without presenting the details.
- For example, a database system hides certain details of how data is stored and created and maintained.
- C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

- Encapsulation is placing the **data and the functions that work on that data in the same place.**
- Object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

- One of the most useful aspects of object-oriented programming is **code reusability**.
- Inheritance is the process of forming a new class from an existing class that is from the existing class called as **base class**, new class is formed called as **derived class**.
- This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

- Poly refers to many.
- The **ability to use an operator or function in different ways**
- Giving different meaning or functions to the operators or functions is called **polymorphism.**
- That is a single function or an operator functioning in many ways different upon the usage is called **polymorphism**.

Overloading

- The concept of overloading is also a branch of polymorphism.
- When the **exiting operator or function is made to operate on new data type**, it is said to be **overloaded**.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

C++ OBJECT AND CLASS

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

C++ Object

- In C++, Object is a **real world entity**, for example, chair, car, pen, mobile, laptop etc.
- Object is an entity that has **state and behavior**. Here, **state means data** and **behavior means functionality.**
- Object is a **runtime entity, it is created at runtime**.
- Object is an **instance of a class**.
- All the **members of the class can be accessed through object**.

An example to create object of student class using s1 as the reference variable.

```
Employee e1;  //creating an object of Employee
```

In this example, Employee **is the type and e1 is the reference variable** that refers to the **instance of** Employee **class.**

C++ Class

- In C++, **object is a group of similar objects**.
- It is a **template from which objects are created**.
- It can have **fields, methods, constructors etc.**

example of C++ class that has three fields only.

```
class Employee
 {
   public:
     int id;  //field or data member
     float salary; //field or data member
     String name;//field or data member
 }
```

C++ Object and Class Example

– An example of class that has **two fields: id and name**. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
   Student s1; //creating an object of Student
   s1.id = 2000;
   s1.name = "harriet";
   cout<<s1.id<<endl;
   cout<<s1.name<<endl;
```

```
    return 0;
}
```

Output:

```
201
Sonoo Jaiswal
```

Differences: Structure vs Class

Default Access Modifier:

Structure: Public

Class: Private

Classes support **member functions**, **constructors**, **destructors**, and **OOP features** (inheritance, polymorphism, encapsulation).

Structures are mainly for grouping related data.