INDEX

Practice + Revision
13 to 24 Aug

- lect + Notes
- Practice

25 - 31 Aug

C++ PROGRAMMING

* Basic features of C++

- low level programming
- Speed of execution
- Derived from C.
- Richer library than C
- Support OOP
- Generic Programming
- Application written in C++ are: Chrome, firefox, OS, Device drivers, compiler.

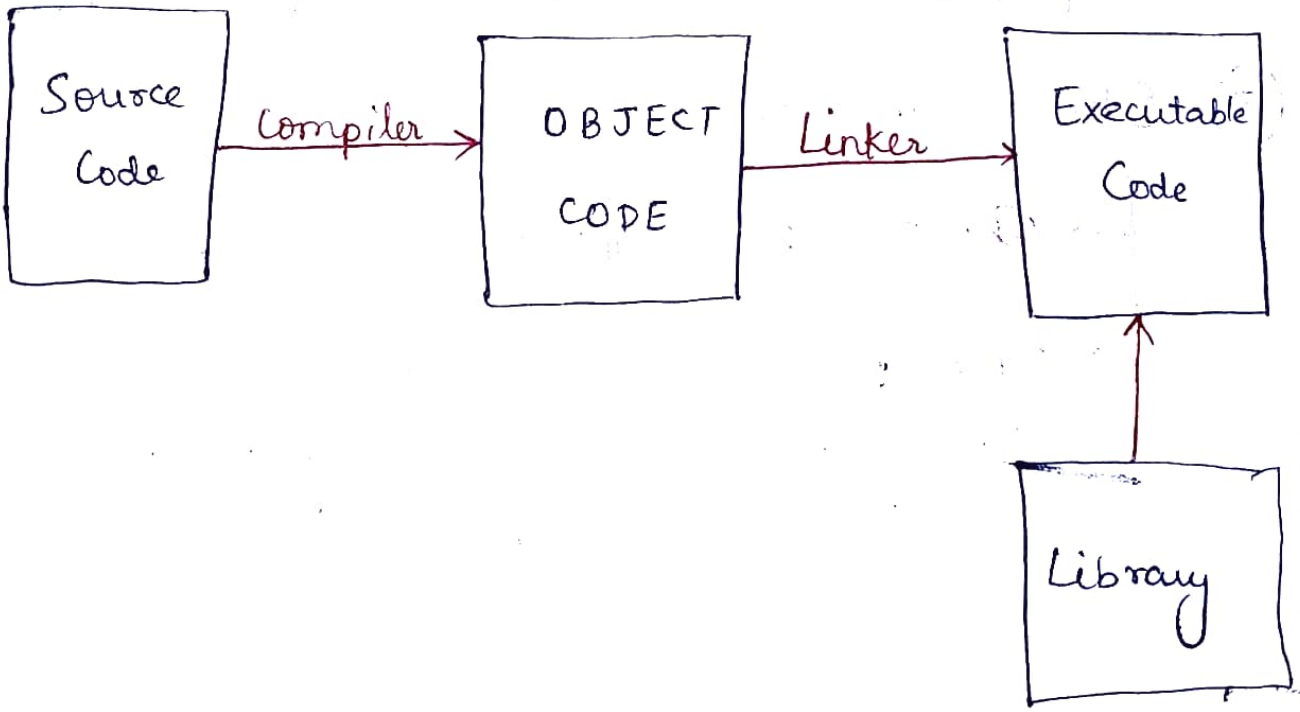
Note

(i) first standardized in 1988 (C++98)

(ii) Later standard (C++03, C++11, C++14, C++17, C++20)

(iii) Compiler - GCC, IBM, VS Code.

* How do C++ Program Run ?



→ Basic Programming Terminology

① Keyboard and Variable

↳ (Reserve words)
eg :- for, while, true, false, if, else etc.

↳ (user defined)
- used to store and acess values.

② Function

- A sequence of instruction that does a specific task.
- c = max (a, b)
  printf ("GeeksforGeeks") {String}

③ Object and classes

→ Class: A type like student, Teacher

→ Object: Instance of a class
(when it is created memory is allocated to the class).

④ statically Typed & Dynamically Typed

↳ (C++, Java)

- Faster, but we have to declare every variable before using it.

↳ Python

- we do not declare variable before using it.
- slower.

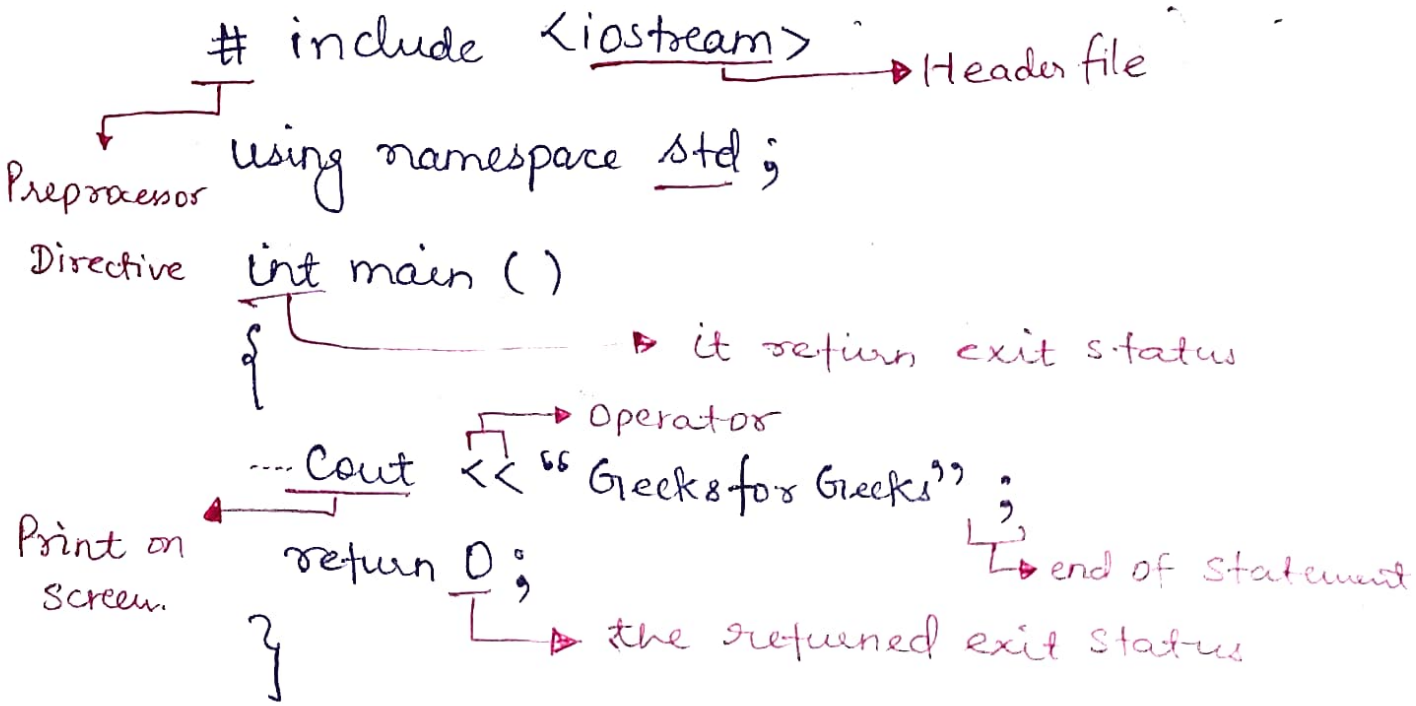
⑤ Header files

- Store declaration of variable, function, classes and structure etc.
- Eg :- # include <iostream>
  ↳ (Decleration related to input & output)

⑥ Namespace

- Divides code into different logical group.
- std namespace contain standard library function and types.
  - ↳ Divide Codes into different logical group.
  - ↳ Name collision Avoiding.

FIRST C++ PROGRAM



OUTPUT

Geeks for Geeks

(i) Preprocessor Directive
↳ Program that process the source code before compilation (add libraries to code & execute).

(ii) Inside curly braces : block of code

(iii) int main() :- fn from where code execution starts / begin.

→ Comment in C++

(i) Multiline

/* ---
--- */

(ii) Single line

// ---

→ VARIABLE

(containers for storing data values, like number and character)

variable naming rules

(i) Allowed characters
- → Lower case ('a' to 'z')
- → Upper case ('A' to 'Z')
- → Digit ('0' to '9')
- → Underscore ('_')

(ii) Cannot begin with digit

(iii) Cannot be a keyword.

Conventions
- → Camel Case (name, currentYear, playerName)
- → Snake Case (name, current_year, player_name)
- → For Constant : MAX_AGE, LIMIT, PI, ...

→ Data Types in C++

① Primitive Data Type: (fundamental | Basic Data Types)

* Integer Types: short, int, long, long long, unsigned short, unsigned int, unsigned long, unsigned long long.
* Character Types: char, unsigned char, wchar_t, char8_t, char16_t, char32_t
* Floating Type: float, double, long double.
* Others: bool, void

② Non-Primitive Data Type

Array, Pointer, User Defined (class & structure), string, vector, ...

→ Operator size in C++

- → An operator in c++ (evaluated at Compile time)
- → Return number of bytes required for data type.
- → Can be used with variable and literals also.

→ Global variable & Local variable

```
# include <iostream>
using namespace std;
int main ()
{
  int x = 10;
  cout << x;
  return ;
}
```

output :- 10

LOCAL variable

Note :-

int x ;

→ output :- undefined (any value)

```
# include <iostream>
using namespace std;
int x = 10
int main()
{
  cout <<x;
  return
}
```

output :- 10

Global variable

Note :-

int x

→ output :- 0

(a) What happend when we have local and global variable with same name ?

↳ print local variable (select innermost value)

NOTE :- (i) Functions, loops and Conditional statement create a new inner scope.

(ii) The outer scope variable are accessible in inner scope but vice versa not true.

(iii) we can create a new scope by using a pair of curly brackets.

→ Static variable in C++

```
# include <iostream>
using namespace std;
int main ()
{
  static int x;
 cout << x;
 return 0;
}
```

OUTPUT :-

0

* Static keyboard in C++ & Uses are :-

→ Static local variable are created once and stay for lifetime of program. (call static any time → same value come)

→ Static data members of a class are shared by all object.

→ Static method of a class can access only Static data.

→ Static Global variable have

* Static is variable declare using keyword Static.

→ Const in C++

(when we use const before a variable it becomes Constant)

```
#include <iostream>
using namespace std;
int main()
{
  const int x = 10;
  x = x + 1;   ← X
  cout << x;
  return 0;
}
```

OUTPUT

Compiled Error

10.

→ Auto Keyword in C++

(used for variable declaration)

eg :-

```
# include <typeinfo>
# include <iostream>
using namespace std;
int main()
{
    auto a = 10;
    auto b = 15.5;

    cout << a << "\n" << b;
    cout << typeid(a).name() << "\n"
         << typeid(b).name();

    return 0;
}
```

literals (const are called)

to check what data type is assigned.

OUTPUT

```
10
15.5
i (integer)
d (double)
```

Advantages of Auto

(1) No Conversion happens when auto is used

Conversion happens → float x = 3.4; (Double literal converted to float)

No → auto x = 3.4; (Type of y is double)

(2) If a functions return auto, its return type can be changed without worrying about prototype change.

(3) can be very helpful for lengthy types, especially STL (standard template library) iterators.

* vector <int>:: iterator i;
  can be replaced with auto i;

(4) Laambda Expression (Anonymous function)

→ Literals in C++

(2)
```
#include <iostream>
using namespace std;
int main()
{
  int a = 20;
  int b = 0x1A;
  int c = 016;
  int d = 0b11;

  cout << a << "\n"
       << b << "\n"
       << c << "\n"
       << d;

  return 0;
}
```

Using Prefixes in Integer literals

- No prefix → Decimal
- 0x → Hexadecimal
- 0 → Octal
- 0b → Binary

OUTPUT

```
20
26   (1A = 10x16^0 + 1x16^1)
14   (16 = 6x8^0 + 1x8^1)
3    (11 = 1x2^0 + 1x1^1)
```

(b)

```
#include <iostream>
using namespace std;
int main ()
{
 int a = 124;
 unsigned int b = 124u;
 long int = 124L;
 long long int = 124 u;
 return 0;
}
```

Using Suffixes in integer literals

Floating Point literal
- Allowed Prefix : 0x
- Allowed suffix :
  - f → float
  - l → long double
- No suffix → double

(c) Floating Point literals
(value has decimal point → floating point)

```
#include <iostream>
using namespace std;
int main ()
{
    float a = 10.5f ; // Float
    double b = 10.515 ; // Double
    float c = 2.1e15f ; // 2.1 x 10^15
    double d = 200.1e-80 ; // 200.1 x 10^-80
    double f = 0x1A.1p2 ; // 26.0625 x 2^2
    return 0;
}
```

(d) Character and string literal in C++

```
#include <iostream>
using namespace std;
int main()
{
  char c = 'g';
  const char *s1 = "Gfg";
  String s2 = "courses";
  return 0;
}
```

Character & String literal in C++

↳ Prefixes :

No Prefix : ASCII

u8 : UTF-8

u = UTF-16

U = UTF-32

L = wchar_t

* NOTE : When we create string literal, it is first stored in static area of memory in a constant area that cannot be changed. And you get addres of this String in variable. (* → address operator)

→ Type Conversion in C++

(*) Most of the primitive types are allowed to convert to each other without any explicit type casting.

(*) Some of the conversion cause loss of information.

eg :- float x = 10.5 ;
int y = x ; // y = 10
bool z = y ; // z = true

(*) If we have an expression of multiple types below rules are followed :

bool → char → short int → int → unsigned int → long → long long → float → double → long double

```
int main ()
{
int x = 10
char y = 'A'          → (Ascii value is 65)
cout << (x+y) << "\n";
float z = 5.5 ;
cout << (x+z);
return 0;
}
```

OUTPUT

75
15.5

(*) Explicit Conversion

Without Explicit

```
#include <iostream>
using namespace std;
int main ()
{
  int x=15, y=2;
  double z = x/y
  cout <<z;
  return 0;
}
```

OUTPUT

7

Here, first division happen
↓
then double value assign.

With Explicit

```
#include <iostream>
using namespace std;
int main ()
{
int x=15; y=2;
double z = double (x) /y;
   // Explicit
  cout <<z;
  return 0;
}
```

OUTPUT

7.5

with the help of explicit double value assign & then
↓
division performed.

- It is C-Style Conversion.

(2) C++ style Explicit Conversion

(Static cast do validatory ~~to~~ check that whether we can convert or not)

↓

throw Compiler error.

```
# include <iostream>
using namespace std;
int main ()
{
 int x = 15 , y = 2 ;
 double z = static_const <double > (x)/y ;
 Const << z ;
  return 0 ;
}
```

OUTPUT

7.5

---

* STRING

↳ string variable contains a collection of character surrounded by double quotes.

→ Input and Output in C++

(a) Input

- Cin : object of istream
- >> : Extraction operator.

* stream is an abstraction.

(b) Output

- Cout : object of ostream
- << : Insertion operator

* Abstraction (displaying only essential information and hiding the details)

NOTE :- getline() fo Read string with spaces

```
# include <iostream>
using namespace std;
int main()
{
  string name;
  Cout << "Enter your name : \n";
  getline (cin , name);
  Cout << "Welcome" << name;
  return 0;
}
```

OUTPUT

>> Enter your name
>> Sandeep Jain
>> Welcome Sandeep Jain

Escape Sequence in C++

Q. How to print the word enclosed in double quotes?

Welcome to "GeeksforGeeks"

A. The below line will not print the output:

```
cout << "welcome to "GeeksforGeeks"";
```

```
cout << "welcome to \"GeeksforGeeks\"";
```

Q. How to print a backslash?

↳ We type two backslash instead of one.

IO Manipulation

(How we change default behavior of printing things like integer, float, bool etc.)

(1) Boolean value

```
# include <iostream>
using namespace std;
int main()
{
    bool a = true;
    cout << a << "\n"; // 1
    cout << std::boolalpha;
    cout << a << "\n"; // true
    cout << std::noboolalpha;
    cout << ; // 1
    return 0;
}
```

(2) Integer ⟷ Hexadecimal

(3) Integer ⟷ Octa

→ Default Printing Format

⊛ Floating Point

- No trailing zeroes.
- Precision means total digits (excluding the digit used after e)
- Default precision value is 6.
- When value before decimal point does not fit in 6 digit, power format is used.

  Eg:- 1234563.3 is printed as 1.23457e+06.

```
# include <iostream>
using namespace std;
int main ()
{
  double x = 1.2300;
  cout << x << "\n"; // 1.23
  double y = 567.56732;
  cout << x << "\n"; // 567.57
  double z = 1244567.45;
  cout << z << "\n"; // 1.24457e+06
  double w = 124456.67;
  cout << w << "\n"; // 124457
  double u = 123e+2;
  cout << u << "\n"; // 12300
  return 0;
}
```

→ Manipulating Default Format

⊛ Floating Point

- → Setprecision (n) : change the default precision.
- → Showpoint : shows trailing zeroes, noshowpoint reverts it.
- → showpos : show + sign for positive values, noshowpos reverse it.
- → Uppercase : Print 'e' as 'E'. nouppercase reverse it.

```
#include <iostream>
# include <iomanip>
using namespace std;
 int main()
 {
  cout << std :: setprecision (4);
  double x = 15.5683, y = 34267.1;
  cout << x << "  " << y << "\n"; // 15.57  3.427e+04
  double z = 1.23;
  cout << std :: showpoint << z << "\n"; // 1.230
  cout << std :: showpos << z << "\n"; // +1.230
  cout << std :: uppercase << y << "\n"; // +3.427E+04
  return 0;
 }
```

Floating Point

(Fixed and Scientific)

(Fixed)	(Scientific)
No power (or e)	Use (E)
Use . (dot)	1.2e+04
10.5 , 1001.5	1.45e+06

- In both, precision mean digit after the decimal point.
- If there are not enough digit, then trailing zeroes are shown in both.
- We can set back to default using 'defaultfloat'.

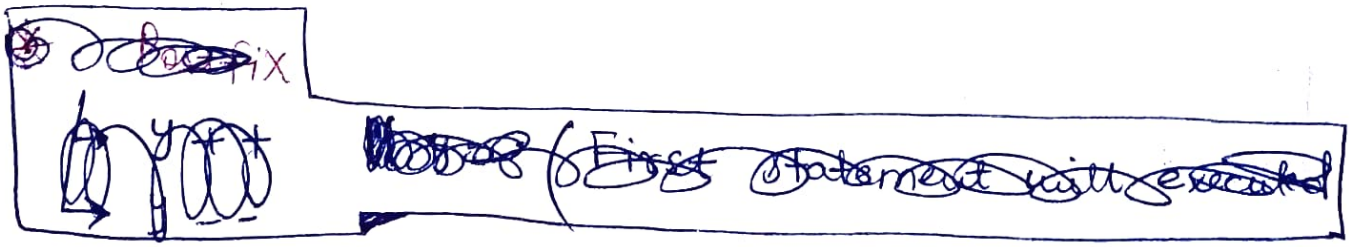
```
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
  double x = 1.23, y = 11122456.453;
  cout << std::fixed;
  cout << x << "\n";  // 1.230000
       << y << "\n"; // 11122456.453000
  cout << std::setprecision (2);
  cout << x << "\n"; // 1.23
       << y << "\n"; // 11122456.45
  double z = 1.2e+7;
  cout << z; // 12000000.00
  return;
}
```

Arithmetic Operation

- +, -, *, and / work for both integer type (int, long long, ..) and floating point types (float, double and long long) but % work for integer types only.
- The sign of a x b is same as sign of a

```
Cout << (10%-3); // 1
Cout << (-10%3); // -1
```

- In case of integer, the result of a/b and a % b is undefined when b is zero.



* POSTFIX
  - (x++, x--)
  - First statement will executed with the current value of variable f then value get incremented.

* PREFIX
  - (++x, --y)
  - first value of x get incremented and then statement will get executed.

✳ Assignment Operator

(=, +=, -=, *=, /=, %=, $=, !=,
<<=, >>=, ^=)

- x += 10 (x = x+10)
- x -= 10 (x = x-10)
- x *= 10 (x = x*10)
- x /= 10 (x = x/10)
- x %= 10 (x = x%10)

✳ Comparision Operator

(<, >, ==, >=, <=, !=, <=>)

<=> added in C++20

eg:-

```
int main
{
int x=10, y=20;
cout << (x<y) << " "
     << (x>y) << " "
     << (x==y) << " "
     << (x>=y) << " "
     << (x<=y) << " "
     << (x!=y) << " "
```

OUTPUT

1
0
0
0
1
1

Logical operators

Binary { && (and)
         || (or)

Unary { ! (not)

} we can also use and, or, not instead of &&, ||, !

NOTE:- Short circuiting in logical operator

(a) (if the result of first expression is false then there is no need to evalute the second expression
↓
Simply print result 0)

(b) if first expression is true then check second expression.

Operator Procedure & Associativity

++ , -- (suffix)	left to Right
! , ++ , -- (Prefix)	Right to left → Start simplification from right side.
* , / , %	Left to right
+ , -	"
<=>	"
< , > , <= , >=	"
== , !=	"
&&	"
!!	"
+= , -= , *= , /= , %= , >>= , <<= , &= , != , ^=	Right to left
,	left to right

eg:-

```
# include <iostream>
using namespace std ;
int main ()
{
 int x = 10 , y = 20 ;
 int z = x + x * y ;
 Cout << x << "\n" ;  // 20
 z = y / x * x ;
 Cout << z << "\n"  // 20

 return 0 ;

}
```

→ Binary Representation of Negative Numbers

- Negative no. are represented in 2's compliment form.
- Range of numbers : $[-2^{n-1}$ to $2^{n-1}-1]$
  (n = no. of bits)
- Steps to get 2's Compliment
  (a) Invert all bits
  (b) Add 1

(Direct formula $= 2^n - x$)

eg :- $n = 4$

Range = $[-2^3$ to $2^3-1]$

Binary representation of $x = -3$

3 : 0011

2's Comp. = (1100+1)

= 1101.

(*) Why 2's Compliment form ?

(1) We have only one representation of zero.

(2) The arthematic operation are easier to perform. Actually 2's compliment form is derived from the idea of 0-X.

(3) The leading bit is always 1.

→ BITWISE OPERATOR

(1) Bitwise AND : &

```
int main ()
{
 int x = 3;
 int y = 6;
 cout << (x & y);
 return 0;
}
```

OUTPUT : 2

3 : 00.....0011
6 : 00.....0110 &
00.....0010

1st	IInd	&
0	0	0
0	1	0
1	0	0
1	1	1

(2) Bitwise OR : !

```
int main ()
{
 int x = 3;
 int y = 6;
 Cout << (x ! y);
```

OUTPUT : 7

3 : 00.....0011
6 : 00.....0110 !
00.....0111

1st	2nd	!
0	0	0
0	1	1
1	0	1
1	1	1

③ Bitwise XOR : ^

```
int main()
{
 int x = 3;
 int y = 6;
 cout << (x ^ y);
 return 0;
}
```

OUTPUT : 5

3 : 00.... 0011
6 : 00.... 0110 ^
00.... 0101

1st	2nd	^
0	0	0
0	1	1
1	0	1
1	1	0

④ Left shift : <<

```
int main()
{
 int x = 3;
 cout << (x<<1) <<endl;
 cout << (x<<2) <<endl;
 int y = 4;
 int z = (x<<y);
 cout << 3;
 return 0;
}
```

OUTPUT :

6
1 2
48

x : 0000..... 00001 1

x<<1 : 0000..... 0001 10

left ← right

(i) last 1 (or n) digit move towards left side

(ii) Put 0 at last or infill area.

x<< 2 : 0000..... 0001 100

(5) Right Shift : >>

```
int main()
{
 int x = 33;
 cout << (x>>1) << endl;
 cout << (x>>2) << endl;
 int y = 4;
 int z = (x >>y);
 cout << z;
 return 0;
}
```

x : 0000 100001

x>>1 : 0000 010000

(1) eliminate last digit moving towards right

(2) add 0's at begning

x>>2 ; 0000....001000

OUTPUT:

16
8
2.

(*) x << y is equivalent to $x * 2^y$ it leading y bits are 0 in x.

x	y	x << y	
3	1	6	(3×2^1)
3	2	12	(3×2^2)
3	4	18	(3×2^4)

(*) x >> y is equivalent to $[x/2^y]$:-

x	y	x >> y	
33	1	16	$(33/2^1)$
33	2	8	$(33/2^2)$
33	4	2	$(33/2^4)$

(6) Bitwise Not : ~

```
int main()
{
unsigned int x = 1;
cout << (~x) << endl;
x = 5;
cout << (~x) << endl;
return 0;
}
```

x = 1

00..... 01

~x

11..... 10

x = 5

00..... 101

~x

11..... 010

OUTPUT :

4294967294

4294967290

NOTE:- $2^{32} - 1 = 4294967296 - 1$

$= 4294967295$.

(*) 2's Compliment of x $= 2^{32} - x$

(For signed Number).

Questions

① Day before N Days.

n = no. of days

d = current day

d	Day
0	Sun
1	Mon
2	Tues
3	Wed
4	Thurs
5	Fri
6	Sat

- input : d = 1, n = 1 { d = 0, d = 9
- output : 0 { 5

input	n = 123	n = -352
output	3	2

- Simply divide these ally by 10 (remainder will be our output).

```
# include <iostream>
using namespace std;
int main ()
{
   int x = -235
   int ans = x % 10;
   cout << ans ;
   return 0;
}
```

OUTPUT

-5

```
#include <iostream>
using namespace std;
int main()
{
int x = -235 ;
int ans = abs(x) % 10;
cout << ans ;
return 0;
}
```

OUTPUT

5

So, to neglect these − sign we use

Convert negative → +ve

Convert positive → remains same.

(3Q) Sum of all natural (N) number.

Soln 1 use for loop & add all the number.

Soln 2 using formula : $S_n = \frac{n(n+1)}{2}$

```
int main()
{
  int n = 6;
  int ans = n*(n+1)/2;
  cout << ans;
}
```

(4Q) Nth term of an AP

- 2, 5, 8, 11, 14

(N) $t_n = a + (n-1)d$

(t_n = nth term, a = first term, d = common diff.)

```
int main() {
int a = 2;
int d = 1;
int N = 5;
int ans = a + (n-1)d;
cout << ans;
return 0;
}
```

FLOW CONTROL

→ If else in C++

```
if (Condition)
{
    // Statements to Execute
    // when condition is true
}
else {   // optional
    // statement to execute
    // when condition is false
```

Q-1 Even odd Program

```
#include <iostream>
using namespace std;
int main () {
int n;      // before using variable first declare.
cout << " Enter a number:" ;
cin >> n;
if (n%2 == 0)
cout << " Even" ;
else
 cout << " odd" ;
return 0;
```

OUTPUT

```
>> Enter a number : 5
>> odd.
```

(8-2) Positive Negative Zero Programm

I/P : 10	I/P : -5	I/P : 0
O/P : Positive	O/P : Negative	O/P : Zero.

- Condition :
  - $n > 0 \Rightarrow$ positive
    $n < 0 \Rightarrow$ negative
    $n = 0 \Rightarrow$ zero

```
#include <iostream>
using namespace std;
int main()
{
  int n;
  cout<<" Enter a number : ";
  cin >> n ;
  if (n>0)
  cout <<"Positive" ;
  else if (n<0)
   cout <<"Negative" ;
  else
   cout <<" zero";
  return 0;
  }
```

```
if (condition 1)
{
  // when con 1 is true.
}
else if (condition 2)
{
  // when cond 2 is true
}
else
  {
   // when both true
  }
```

OUTPUT

<< Enter a number : 10
<< Positive .

(Q-3) Number Sign and Even Odd Program.

I/P : 10	I/P : -12	I/P : 0
O/P : Positive Even	O/P : Negative Even	O/P : Zero

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

SWITCH IN C++

- The control variable must evaluate to integral type.
- The expression used in case should be constant

```
# include <iostream>
using namespace std;
 int main ( )
 {
  int x=0 , y = 0 ;
  cout << " Enter a choice \n" ;
  Char move ;
  cin >> move ;
  Switch (move)
  {
    case 'L' : x-- ;
               break ;
    case 'R' : x++ ;
               break ;
    Case 'U' : y++ ;
               break ;
    case 'D' : y-- ;
               break ;
   default : cout << " Invalid choice" ;
  }
  cout << x << " " << y ; // Depend on move.
 }
```

FUNCTIONS

- block of code which only runs when it is called.

```
#include <iostream>
using namespace std;
void fun()
{
  cout << "fun() called \n";
}
int main()
{
 cout << "Before calling fun() \n";
 fun(); // Function called
 cout << "After calling fun()";
 return 0;
}
```

→ cannot return any value

Function

OUTPUT

```
Before calling fun()
fun() called
After calling fun()
```

- we can call a function many times.

① function can return a value to the Caller.

```
#include <iostream>
using namespace std;
int getMax (int x , int y)
{
  if (x>y)
    return x;
  else
    return y;
}
int main()
{
  int a = 10, b = 20
  Cout<< getMax (a, b);
  return 0;
}
```

return ← Keyword

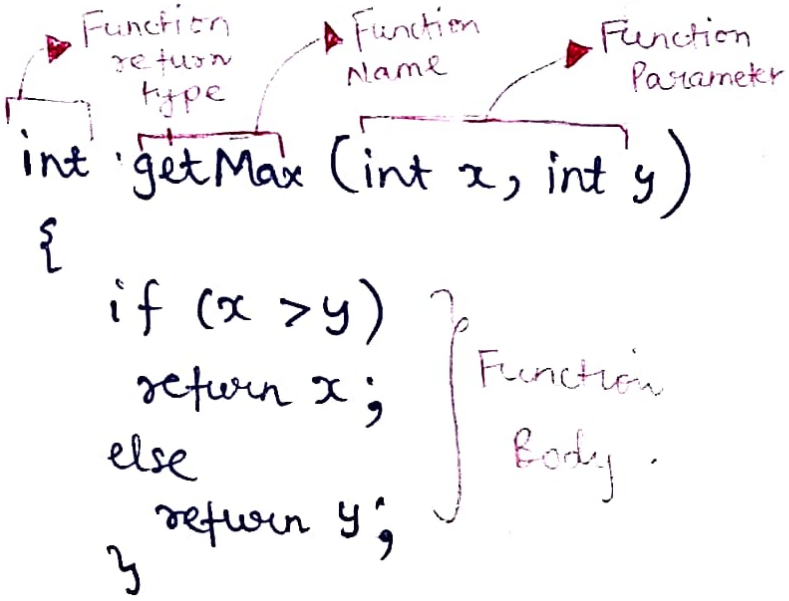
Caller

OUTPUT

20

If the function returned is encounter
↓
it gives the Comand to Caller.

⊛ Function Call Stack



(*) Applications of Function (or Method)

1. Avoid Code Redundancy & Ease maintainance.
2. Make Code Modular
   - takeInput()
   - ProcessData()
   - ProduceOutput()
3. Abstraction: (For example, in library functions we do not have to worry about internal working)
4. Avoid variable name collisions.

(*) Function Declaration and Defination

→ Two ways to use functions:

1. Define first, then use
2. Declare first, then use, then define.

→ If we declare first and does not define function its shows → Linker error.

Default Arguments in C++

```
Void printDetail (int id, string name = "NA",
                  string address = "NA")
{
    cout << "Id is " << id << "\n";
    Cout << "Name is" << name << "\n";
    Cout << "Address is" << address << "\n";
}
int main()
{
    printDetails (101, "Sandeep", Noida");
    Cout << "\n";
    printDetails (201, "Shivam");
    Cout << "\n";
    Print Details (301);
     Cout << "\n";
    return 0;
}
```

Output

```
Id is 101.
Name is Sandeep
Address is Noida

Id is 201
Name is shivam.
Address is NA.

Id is 301.
Name is NA.
Address is NA.
```

Properties of Default Argument

(1) All Default argument must appear at end.

```
void printDetail (int id , string name = "NA",
                                    string Address ).
{
   cout << "Id is " << id << "\n";
   Cout << "Name is" << name << "\n";
   Cout << "Address is" << address << "\n" ;
}
 int main ()
{
    printDetail (101, "Sandeep", "Nodia")
    cout << "\n";
    printDetail (201, "Delhi") ;
    return 0;
 }
```

OUTPUT
Compiler Error

(2) Default argument values can be provided either in declaration or in defination.

```
int sum (int a, int b, int c=0, int d=0);
int main()
{
  cout << sum (10, 20, 30) << "\n";
  cout << sum (10, 5) << "\n";
  return 0;
}
int sum (int a, int b, int c, int d)
{
  return (a+b+c+d);
}
```

OUTPUT

```
60
15
```

NOTE:- If we provide default argument values at both

(1) Declaration
(2) Defination

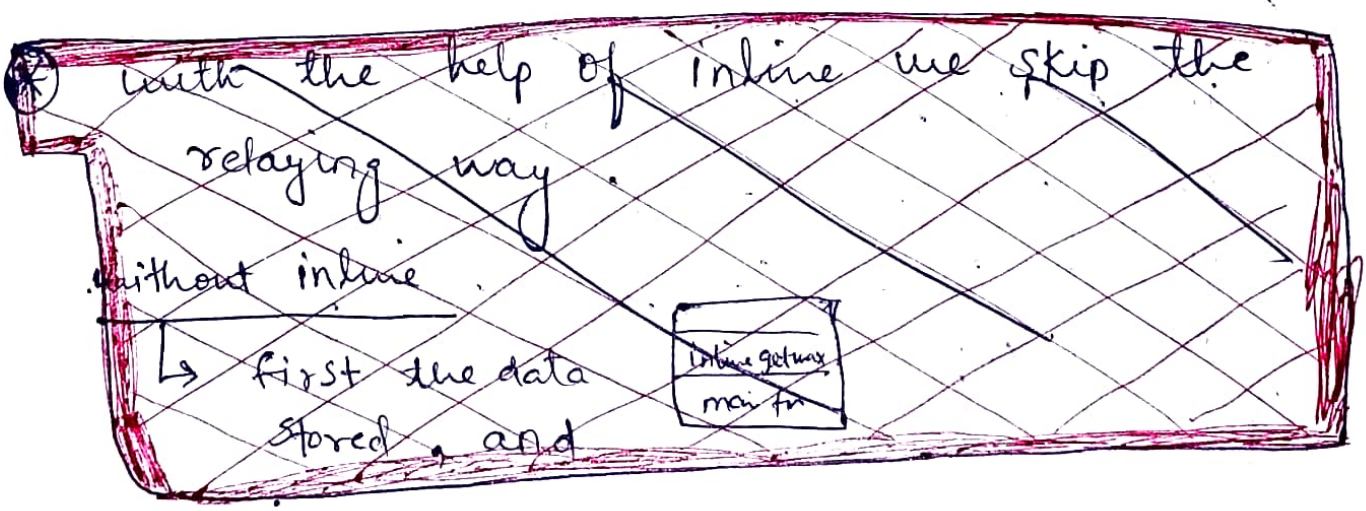
↓

Compiler Error

→ Inline Function

```
inline int getMax (int x, int y)
{
  return (x>y) ? x : y;
}
int main ()
{
   cout << getMax (10, 20);
   return 0;
}
```

OUTPUT
20



- Inline fn reduce the function call Overhead.
- Inline is just suggestion to the compiler.
- Modern compiler may make a fn inline even if we do not use inline.
- Using inline too much may increase the binary file size
- when we define a method inside a class, it is automatically treated as inline suggestion to the compiler.

→ Function Overloading

```
int myMax (int x, int y)
{
  return (x>y)? x:y;
}
int myMax (int x, int y, int z)
{
  return myMax(myMax(x,y),z);
}
int main()
{
  int a=10, b=30, c=5;
  cout << myMax(a,b) << "\n";
  cout << myMax(a,b,c) << "\n";
  return 0;
}
```

OUTPUT
30
30

(*) Same fn name but with different input parameter.

(*)

```
void print (string s)
{
  cout << s << "\n";
}

void print (int x)
{
  cout << x << "\n";
}

void print (char c)
{
  cout << c << "\n";
}

int main ()
{
  print ('a');
  print ("10");
  print ("GfG");
  return 0;
}
```

OUTPUT
a
10
GfG

(Automatically Conversion of Data types)

if we cut this fn

then the print('a')

char value get conveted into ASCII values of (a = 97).

OUTPUT
97
10
GfG

(*) Function Overloading Carefully used with default argument

```
void print (int x=0)
{
   cout << x << "\n";
}
void print ()
 {
 }
int main()
{
 print();
 return 0;
}
```

OUTPUT

Compiler Error

Practice Session

```
#include <iostream>
using namespace std;
void fun()
{
    static int x = 1;
    int y = 1;
    x++;
    y++;
    cout << x << " " << y << endl;
}
int main()
{
    fun();
    fun();
    fun();
    return 0;
}
```

it is local to this function

When we have static local variable. so this variable is allocated & defined only once.

OUTPUT	
2	2
3	2
4	2

→ LOOPS IN C++

(*) Applications of loop

- → Doing Some Work Repeatedly
- → Traversing through containers like array, string, map etc.
- → Running Service in System.

(1) While Loop in C++

```
while (Condition)
{
    Statement 1;
    Statement 2;
    :
}
```

SYNTAX

```
# include <iostream>
using namespace std;
int main()
{
  int i = 0;
  while (i<5)
  {
    cout << "GfG \n";
    i++;
  }
  return 0;
}
```

→ i = 0
condition is True
GfG
i = 1
condition is True
GfG
i = 2
condition is True
GfG
i = 3
condition is True
GfG
i = 4
condition is True
GfG
i = 5
Condition is False.

OUTPUT

GfG
GfG
GfG
GfG
GfG

(*) Infinite loop using WHILE

```
while (1)
{
  ..... .....
  ..... .....
}
```

OR

```
while (true)
{
  ..... .....
  ..... .....
}
```

(*)

```
#include <iostream>
using namespace std;
int main()
{
  int i = 0;
  while (i < 5);
  {
    cout << "GfG \n";
    i++
  }
  return 0;
}
```

Considered as single line statement

another block of code

∴ It print infinite GfG.

Σ

2 For loop in C++

```
for (initialization ; condition ; change loop variable )
{
    // Statements to be executed inside the
    // loop
}
```

SYNTAX

eg :-

```
int main ()
{
    for (int i = 0 ; i < 3 ; i++)
    {
        cout << " GfG \n" ;
    }
    return 0;
}
```

initialization — condition — change loop variable

OUTPUT

```
GfG
GfG
GfG
```

i = 0
Condition is true — GfG
i = 1
Condition is true — GfG
i = 2
Condition is true — GfG
i = 3
Condition is False

eg :-

```
int main ()
{
int i;
for (int i = 0; i < 3; i++)
    cout << "GfG" << i << endl;
cout << i;
return 0;
}
```

declare inside the loop.

Compiler error because variable is not declare.
(we have to declare variable outside the loop)

OUTPUT
GfG 0
GfG 1
GfG 2
3

eg :- Not print anything

```
int main()
{
for (int i=3; i<3; i++);
    cout << "GfG \n";
return 0;
}
```

if we use ; in for loop statement

Print GfG one time.

(*) int main ()

{

for (; ;)

{

cout << "GfG \n" ;

}

return 0 ;

}

Infinite Loop

OUTPUT

GfG
GfG
.
.
.

[3] Do while Loop in C++

```
do {
    // Statement to be
    // executed inside the
    // loop
} while (condition);
```

→ Statement written here going to executed at least 1 time.

-
```
int main ()
{
    int i = 0;
    do {
        cout << "GfG \n";
        i++;
    } while (i<3);
    return 0;
}
```

→ what if int i = 3

↓

It print GfG once.

OUTPUT

GfG
GfG
GfG

BREAK in C++

(8) Find the smallest divisor greater than 1.

I/P : n = 6	I/P : n = 13	I/P : n = 49
O/P : 2	O/P : 13	O/P : 7

Soln:

- Run a loop from 2 to n.
- If $n \% x == 0$, then print x & come out of the loop.
  - Here we use BREAK.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Enter n: ";
    cin >> n;
    for (int x = 2; x <= n; x++)
    {
        if (n % x == 0)
        {
            cout << "Smallest divisor is " << x;
            break;
        }
    }
    return 0;
}
```

OUTPUT

```
>> Enter n:
>> 6
>> Smallest Divisor is 2.
```

(*)

```
#include <iostream>
using namespace std;
int main()
{
   int i=1;
   while (i<=5)
   {
      if (i==3)
         break;
      cout<<" i<<" ";
      i++;
   }
   cout<<i<<" ";
   return 0;
}
```

OUTPUT

1 2 3

NOTE

(i) 1 is print

(ii) 2 is print

(iii) 3 is print & executation of code is stop.

(∵ at there there is break).

CONTINUE in C++

(Q) Print all numbers smaller than or equal to n that are not multiple of x.

- I/P : n = 10, x = 3

  O/P : 1 2 4 5 7 8 10

- I/P : n = 7, x = 5

  O/P : 1 2 3 4 6 7

When we have a loop & we want some parts to be not skip

↓

use of Continue

- ```
  int main()
  {
    int n, x;
    cout << "Enter n : ";
    cin >> n;
    cout << "Enter x : ";
    cin >> x;
    for (int i = 1, i <= n, i++)
    {
        if (i % x == 0)
            continue;
      cout << i << " ";
    }
    return 0;
  }
  ```

without continue

```
{
if (i % x != 0)
cout << i << " ";
}
```
```

NESTED LOOP

(Q) Print tables of first n positive integers

I/P : n = 4

O/P :

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40

```
# include <iostream>
using namespace std;
int main()
{
 int n;
 cout << " Enter n : " ;
 cin >> n ;
  for (x = 1; x <= n ; x++)
   {
      for (i = 1; i <= 10, i++)
      {
       cout << (i*x) << " ";
      }
      cout << "\n" ;
   }
   return 0;
}
```

OUTPUT

```
>> Enter n :
>> 4
>>
```

(Q)

```
int main()
{
  for (int i=1; i<3; i++)
  {
    int j=1;
    while (j++<3)
    {
      cout << i << " " << j << "\n";
      j++
    }
    cout << "GfG\n";
  }
  return 0;
}
```

OUTPUT

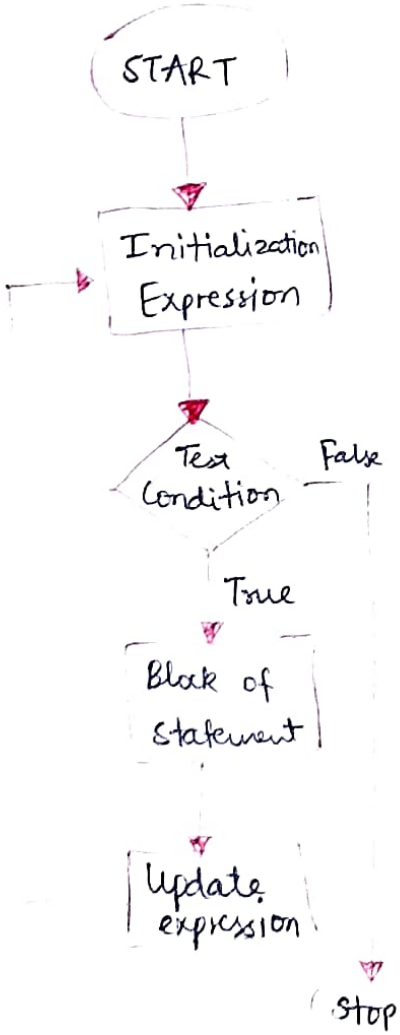
Loop

Entry Control

For

```
for(ini ; cond ; updat)
{

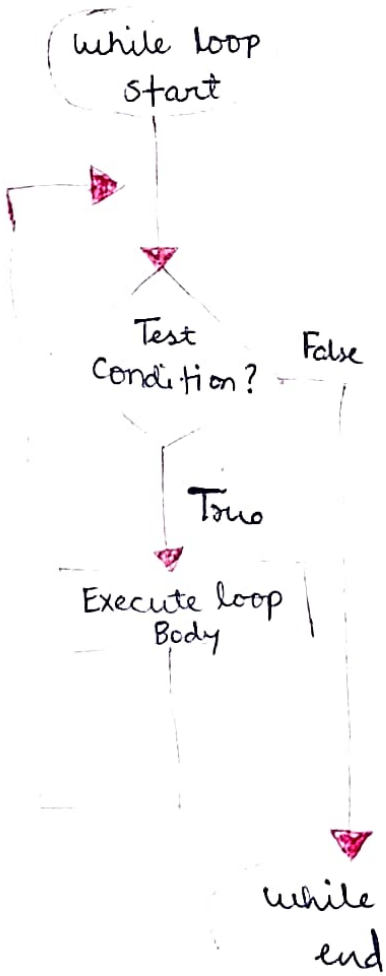
}
```



while

```
while (cond)
{

}
```

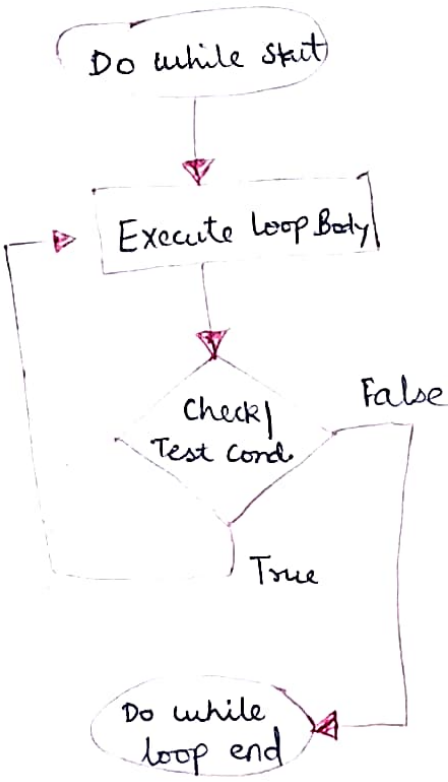


Exit Control

do while

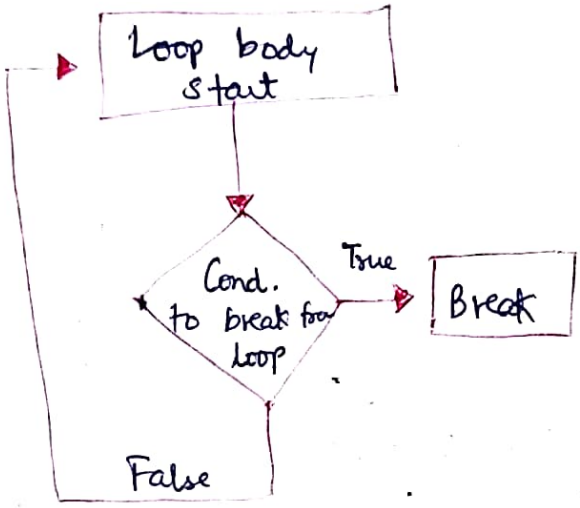
```
do
{

}while (cond)
```



BREAK

- terminate the loop

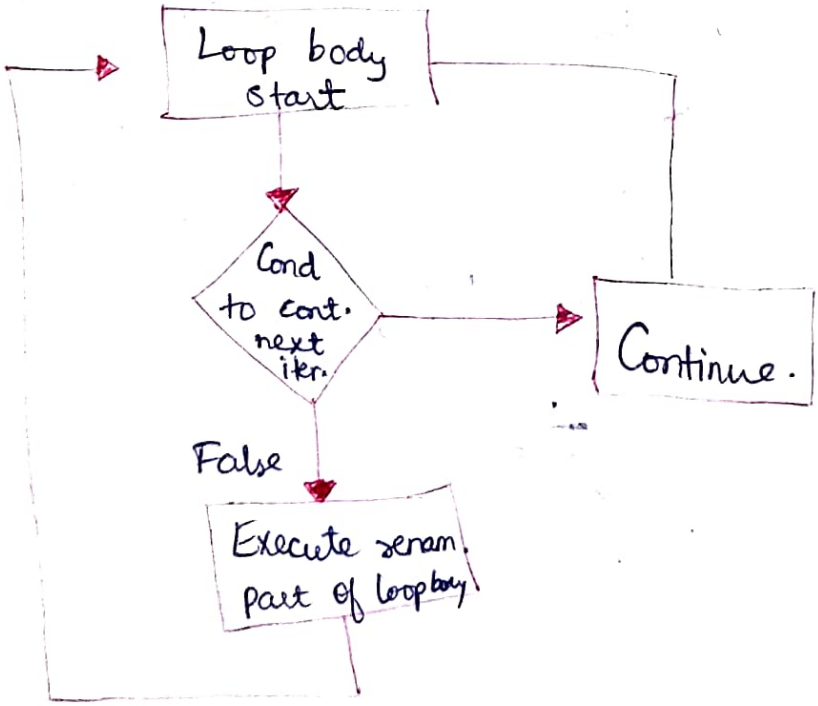


- usage of break stat :-
  - (i) Simple loop
  - (ii) Nested loop
  - (iii) Infinite loop

CONTINUE

- it forces to execute the next ikration of the loop.

ikration → repeatition of process.



[VVI] Basic Approach towards solving a pattern questions.

- Understand the question
- Analyze the pattern.
- Design the solution
- write Code → Test → Modify it (if Possible)

Practice

8) Count digit of a number.

I/P : 1234	I/P : 12
O/P : 4	O/P : 2

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

int main()
{
    int n;
    int count = 0;
    cout << " Enter the number : ";
    cin >> n;

    while (n>0) {
    n = n \ 10;
      count ++;
    }
     Cout << count;
}
```

OUTPUT

>> Enter the number : 124

>> 3.

(8-2) All divisor of a number ?

I/P: n = 6

O/P : 1, 2, 3, 6

I/P : n = 3

O/P : 1, 3

→ (i) give them a range using (for)

→ (ii) Condition using (if) under (for).

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

int main ()
{
   int n;
   cout<<" Enter the number:" ;
   cin >> n;

   for (i=1 ; i<=n ; i++)
    {     if (n%i==0)
            {
             cout <<"The divisors are:"<<i<<" ";
            }
    }

   return 0;
}
```

OUTPUT

>> Enter the number: 7

>> The divisors are: 1, 7

0-3) factorial of a number.

I/P : n = 3

O/P : (3 x 2 x 1) 6

I/P : n = 5

O/P : 120 (5 x 4 x 3 x 2 x 1)

Q4 GCD of two number.

Q-5 LCM of two number.

(Q-6) Binary to Decimal.

(5-7) Decimal to Binary.

(8) Check for Prime.

(8-9) Next Prime Number.

(10) Table of a number.

I/P :- Enter the number : 5

O/P :- 5 10 15 20 25 30 35 40 45 50

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

   for (int i=1 , i<=10 ; i++)
    {
      cout << n*i << " " ;
    }

 return 0;
}.
```

OUTPUT

>> 5 10 15 20 25 30 - - 50

Also use while

```
int n;
cout<<" Enter the number : ";
 cin >> n;
 int i=1;
 while (i<=10)
  {
   cout << n*i <<" ";
     i++
  }
 return 0;
}
```

Q-11 Fibonacci Number.

→ Patterns.

Q-12 Triangle Pattern.

I/P :- Enter no. of rows = 5

O/P :

```
*
* *
* * *
* * * *
* * * * *
```

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout<<"Enter no. of rows:";
    cin>>n;

    for (int i=0; i<=n ; i++) // for Row
    {
        for (int j=0; j<=i ; j++) // for Column.
        {
            cout<<" * ";
        }
        cout<<endl;
    }
    return 0;
}
```

Q-13) Inverted ∨ Pattern.

Triangle

I/P: Enter no of rows = 5

O/P:

```
* * * * *
 * * * *
  * * *
   * *
    *
```

```
#include <iostream>
using namespace std;
Int main()
{
  int n;
  Cout<<" Enter no. of rows:";
  cin >>n;

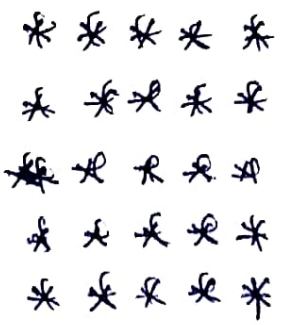
    for (int i=0; i< n ; i++)
      {
          for (int j= 1 ; j<=n-i ; j++)
            {
              cout<<" * ";
            }
        cout << endl;
      }

    return 0;
    }
```

14 Square / Solid Pattern.

I/P :- Enter no of rows : 5

O/P :-



```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout<<" Enter no. of rows :";
    cin >> n;

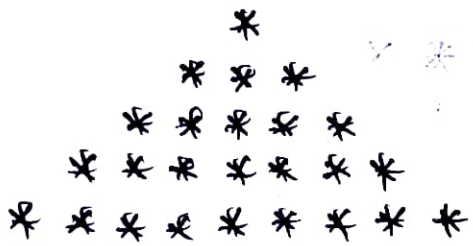
    for (int i=0 ; i<n ; i++)
    {
        for (int j=1 ; j<=n ; j++)
        {
            cout <<" * ";
        }
        cout<< endl;
    }

    return 0;
}
```

Q-15 Pyramid Pattern -

I/P : Enter no of rows : 5

O/P :



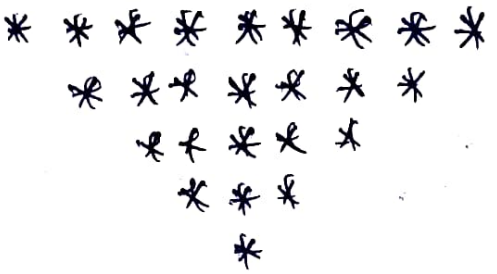
spaces	*
4	1
3	3
2	5
1	7
0	9

```
#include <iostream>
using namespace std;
int main()
{
  int n;
  cout<<"Enter no of rows:";
  cin>>n;
  for(int i=0; i<n; i++)  // Row
    {
      for(int i=1; i<=n; i++)
        {
          for(int j=1; j<=n-i; j++)
            {
              cout<<" ";  // SPACE
            }
          for(int j=1; j<=2*i-1; j++)
            {
              cout<<"*";  // * Print
            }
          cout<<endl;
        }
    return 0;
}
```

Q-16) Inverted Pyramid Pattern.

I/P : Enter no. of rows : 5

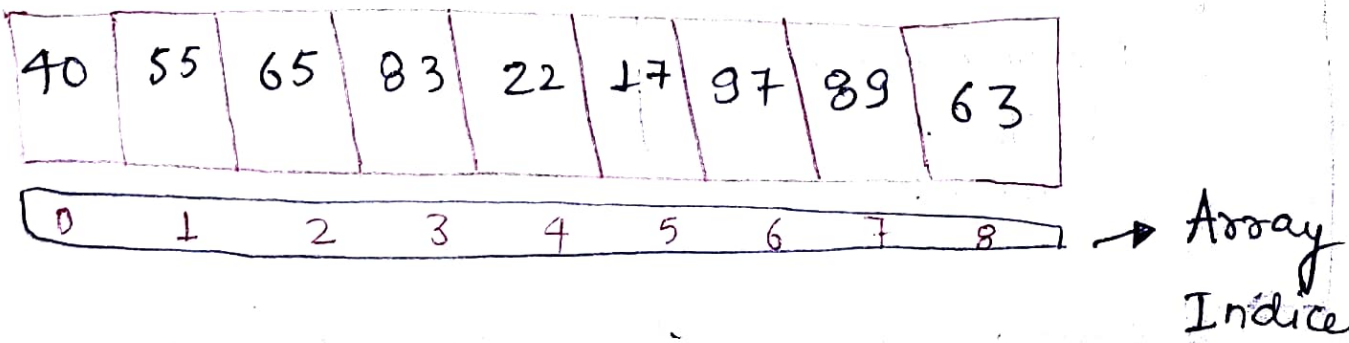
O/P :



```
# include <iostream>
using namespace std;
int main()
{
   int n;
   cout<<" Enter no of rows : ";
   cin >> n;
   for (int i = 0; i<n; i++)
     {
        for (int j = 0; j<= i ; j++)
          {
            cout<<" ";
          }
        for (int j =1 ; j<= 2*n-2* i-1 ; j++ )
            {
              cout << " * ";
            }
        cout << endl;
     }
   return 0;
}
```

→ ARRAY

- Group of variable of similar data types refered to by a single element.
- Used to store collection of primitive data types such as int, float, double, char etc. or any particular type.
- stored in contiguous memory location.
- Size of array should be mentioned while declearing it.
- array element are always count from zero(0) onward
- accessed using the position of the element in the array.
- Array can have one or more dimensions.



Array length = 9

First index = 0

Last index = 8

(*) Why we need arrays ?

→ we can use normal variable ($V_1, V_2, V_3, \ldots$) when we have small number of object,

→ but if we want to store large number of instances, it becomes difficult to manage them with normal variable.

→ Array :- represent many instances in one variable

→
```
int V1 = 10;
int V2 = 20;
int V3 = 30;
int V4 = 40;
int V5 = 50;
```

Multiple variable to store each value.

10	20	30	40	50	.

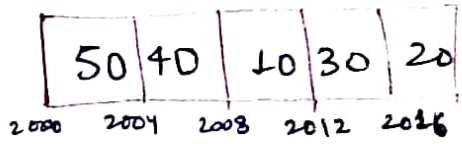
Single array to store all values.

→ Disadvantages :- store only the fixed size of element in array.
It does not grow its size at runtime.

Declaring & Initializing Array

(a)

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5] = {50,40,10,30,20}
return 0;
}
```

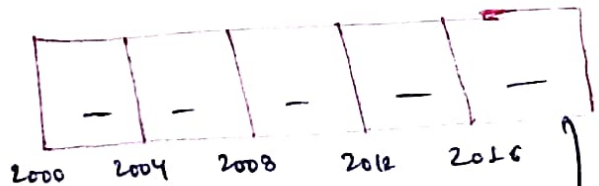


- 2000, 2004, 2008, ... are example address value
- Assumption :- int takes 4 bytes.

if we remove size part → compiler automatically compute array size of 5.

(b) Only array declaration, not initialize.

```
# include <iostream>
using namespace std;
int main()
{
 int arr [5];
return 0;
}
```



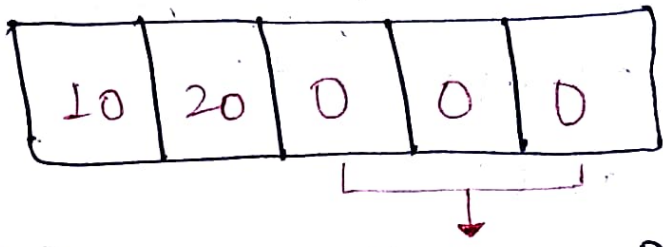
Gets some Random Values.

(*) Size of array can be variable.

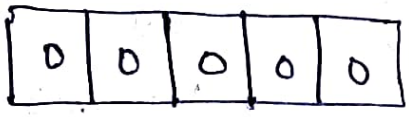
We can take User input also for array size.

```
{
 int n = 5;
 int arr [n];
```

(c) int arr[5] = {10, 20};



(d) int arr[5] = {0}



they got 0.
(because these values are not initialized)

(*) int arr[5] = {3}

(only first member is 3
other will 0)

(e) Compiler Error [array size ≠ member element]

```
int main()
{
    int arr[2] = {10,20,30}
    return 0;
}
```

OUTPUT

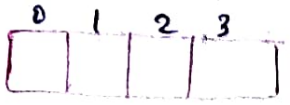
Compiler Error

(f) Alternate ways to initialize Array

```
int main()
{
  int arr1 [ ] {10, 20, 30};
  int arr2 [ ] {10, 20, 30, 40, 50};
  int arr3 [ ] {10, 20};
 return 0;
}
```

→ Accessing Array Element

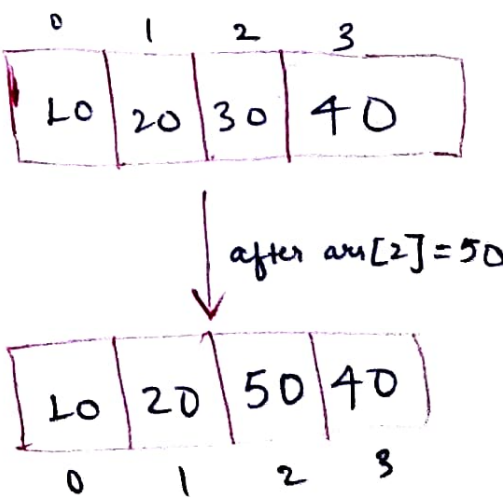
↳ using INDEXES (start from 0)



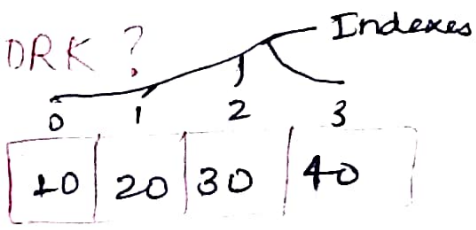

```
int main()
{
  int arr[ ] = {10,20,30,40};
  cout << arr[2]<<" "<<arr[0]<<"\n";
  arr[2] = 50;
  cout << arr[2] <<"\n";
  return 0;
}
```

OUTPUT

```
>> 30 10
>> 50
```



HOW DOES INDEXING WORK?




```
int main()
{
  int arr[] = {10,20,30,40};
  cout << arr << "\n"; // Print address of First element
  cout << arr[2]; // Print element at address
                  // arr + 2 * size of (int),
                  // i.e., 30.
  return 0;
}
```

(*) NO OUT OF BOUND CHECKING IN C++

```
int main()
{
  int arr[] = {10, 20, 30, 40};
  cout << arr[5];
  return 0;
}
```

OUTPUT

- Random values
- segmentation fault
- program may crash.

Size of array

```
# include <iostream>
using namespace std;
int main()
{
  int arr[ ] = {10,20,30,40,50};
  cout << sizeof (arr);
  return 0;
}
```

Assumption
↳ Size of an int is 4 bytes

OUTPUT

20

(*) Count number of element in array

```
# include <iostream>
using namespace std;
int main()
{
  int arr[ ] = {10,20,30,40,50};
  int n = sizeof (arr) / size of (arr[0]);
  cout << n;
  return 0;
}
```

▶ what if
int arr[6] = {10,20}
(so, the other value also get memory so no of element = 6.)

OUTPUT

5

→ ARRAY TRAVERSAL

① Using Normal loop

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

int arr[] = {10,40, 30, 45};
int n = sizeof (arr)/size of(arr[0]);

for (int i=0; i<n; i++)
  {
    cout << arr[i] << " ";
  }

return 0;
}
```

OUTPUT

10 40 30 45

② Using Range Based for loop

```
#include <iostream>
using namespace std;
int main ()
{
  int arr[ ] = {10, 40,30, 45};

 for ( int x : arr )
  {
    cout <<x << " ";
  }
return 0;
}
```

OUTPUT

10 40 30 45

(*) Modification During Traversal

```
# include <iostream>
using namespace std;
int main()
{
int arr[] = {10,40,30};
int n = sizeof(arr)/sizeof(arr[0]);
for (int i=0; i<n; i++)
   arr[i] = arr[i]*2;
for (int i=0; i<n; i++)
   cout << arr[i] << " ";
return 0;
}
```

OUTPUT

20 80 60

```
# include <iostream>
using namespace std;
int main()
{
int arr[] = {10,40,30};
for (int &x : arr)
   x = x*2;
for (int x : arr)
   cout << x << " ";
return 0;
}
```

OUTPUT

20 80 60

Refcences :- main value also modefied.

→ Different Types of Arrays

(1) Fixed Sized Arrays

[a] Allocated in Function call Stack

- int arr[100];
- int arr[n];
- int arr[] = {10, 20, 30, 40};

[b] Allocated in Heap

- int * arr = new int[n];

(2) Dynamic sized Arrays

vector in C++ STL.

Practice

1. Check if Array is Sorted or not.

I/P : [10, 20, 30] O/P : Yes	I/P : [10, 20, 20, 30] O/P : Yes	I/P : [ ] O/P : Yes
* Next element is greater than previous ones.	I/P : [10, 5, 2] O/P : NO	I/P : [10] O/P : Yes

Q-2 Count Distinct Element

I/P : [10, 20, 10, 20, 30]	I/P : [1, 2, 3, 3]	I/P : [1, 1, 1, 1]
O/P : 3	O/P : 3	O/P : 1

Q-3) Sum of an Array & Average.

I|P : [1, 2, 3, 4, 5]

O|P : 15.

Q-4) Min & Max of an Array.

I/P : [1, 2, 3, 4, 5]

O/P : Max = 5

I/P : [1, 2, 3, 4, 5]

O/P : Min = 1.

REFERENCES

- use ampersand (or &) before reference variable declaration.
- must be initialized.
- cannot be Null
- cannot be changed

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    int &y = x;
    cout << y << " ";      // 10
    x = x+5;
    cout << y << " ";      // 15
    y = y+5;
    cout << x << " ";      // 20
    return 0;
}
```

same memory are alloted to them.

↓

if we change x y will also chang

↓

if we change y x will also chan

Functions Parameters & References (Cons L value refer.)

- Problem with normal parameter passing
  - → Changes are not reflected.
  - → The whole object is copied
    (Consider passing a large object, like a large string.)

for better understanding we use ↓ Const

```
#include <iostream>
using namespace std;
void fun(int x)
{
    x = x+5;
}

int main()
{
    int x = 10;
    fun(x);
    cout << x;
    return 0;
}
```

Now, if we print x ↓ 10

(Here Global variable call)

```
#include <iostream>
using namespace std;
void fun(const string &s)
{
    cout << s;
}

int main()
{
    string s = "geeksforgeek Coiurses";
    fun(s);
    return 0;
}
```

Both are different & when we print the values are copied

* Performance issue.

* After making references this issue resolved.

Range Based for loop & References (Const L value)

- Problem with normal loop variable :-
  - → Cannot change element
  - → Performance issue.

→ TRANSVERSAL

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

int main()
{
 int arr[] = {10, 20, 30, 40};
   for (int & x : arr)
      x = x * 2;
   for (int x : arr)
      cout<<x<<" ";
 return 0;
}
```

every variable is copied to this new x.
↓
∴ use references.

OUTPUT

>> 10 20 30 40

>> 20 40 60 80

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

int main()
{
string arr[] = {"geeksforgeek",
                "cpp course",
                "learning"};
```

we use it for better readability

```
for (const string & s : arr)
   cout<< s <<"\n";
return 0;
}
```

every variable is copied to new s.
↓
∴ we use references.
↓
after &
↓
location for both are same.

OUTPUT

>> geeksforgeek
>> cpp course
>> learning

Const & R value References

- The references discussed so far cannot refer to R value e.g., literals and result of expression.
- Const references and R values references (&&) are used for this purpose.

```
#include <iostream>
using namespace std;
void fun (string && s)
{
    s = "Hi" + s;
    cout << s;
}
int main()
{
   fun ("user");
    return 0;
}
```

after this we can now modify the values

// with R value references
// we can change also

```
int & x = 3;          // Invalid
const int & x = 3;    // valid
int && x = 3;         // valid

string & s = "gfg";   // Invalid
const string & s = "gfg";   // valid
string && s = "gfg";  // valid
```

NOTE: if we use Const or &

↓

we can print the values

↓

But we can't modify them.

✱ References Practice Questions.

Q-1

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10, y = 20;
    int &z = x;
    z = y;
    z = z + 5;
    cout << x << " "
         << y << " "
         << z;
    return 0;
}
```

References are only bound to one ↓ ∴ value of y is equal to z. ↓ Since both z & x are of same location ↓ value of both z and x will change.

OUTPUT

>> 25
>> 20
>> 25

Q-2

```
#include <iostream>
using namespace std;
int &fun()
{
    static int x = 10;
    return x;
}
int main()
{
    int &y = fun();
    y = 20;
    cout << fun();
    return 0;
}
```

OUTPUT

>> 20

Q-3

```
# include <iostream>
using namespace std;
int main ()
{
   String s1 = "GfG" , s2 = "Course";
   String ff s3 = s1 + s2;
   s3 = " Welcome to" + s3;
   Cout << s3; // Welcome to GfG Course.
   return 0;
}
```

* Reversing a digit of variable & Swap the value b/w two variable.

I/P :-	O/P :
a = 12345	a = 97887
b = 78790	b = 54321

→ POINTERS

(*) Address and Dereference Operator in C++

[i] & : when we use emphersand before a variable name (except during declaration), we get address of variable.

[ii] * : when we use star before an address (except during declaration), we get value at the address.

eg:-

```
# include <iostream>
using namespace std;
int main ()
{
  int x = 10;
  cout << &x << "\n";   // address (2Dx...)
  cout << *(&x) << "\n"; // 10
  return 0;
}
```

Introduction to Pointers in C++

- Symbolic representation of addresses.

```
#include <iostream>
using namespace std;
int main()
{
  int x = 10; // declare normal variable
  int * p ; // declare Pointer variable
  p = &x; // p stores address of x.
          address of x.
cout << x << "\n";
cout << *p << "\n";
cout << p << "\n";
return 0;
}
```

OUTPUT:

```
>> 10
>> 10
>> 0x1234
```

0x123	x=10
⋮	
0x5658	P= 0x1234

* When we print *p.

↓

it print the value stored at a address which is stored in p.

```
# include <iostream>
using namespace std;
int main ( )
{
   int x = 10;   // local variable
   int *ptr = &x; // Ptr = &x (address of x), *Ptr = value of x.
   Cout << * ptr << "\n"; // 10
   x = x+30;
   Cout << * ptr << "\n"; // 40
   * ptr = *ptr + 40;
   Cout << x << "\n";  // 80
   return 0;
}
```

- Go to their address.
- update the value.

0x1234	x = 10 ~~40~~ 80
0x5658	P = 0x1234

OUTPUT

>> 10
>> 40
>> 80

(*) Pointer stores addresses and all the addresses are of same size (wheather the address of an integer, float, double, char etc).

```
# include <iostream>
using namespace std;
int main()
{
    int *p1;
    double *p2;
    String *p3;
    cout << sizeof(p1) << "\n"; // 8
    cout << sizeof(p2) << "\n"; // 8
    cout << sizeof(p3) << "\n"; // 8
    return 0;
}
```

NOTE: Whenever we create multiple pointer variable in a single declaration we must use * before it.

```
int *p1, x, *p2;
```

Applications of Pointers in C++

1. Changing Passed Parameter.
2. Passing large object.
3. Dynamic Memory Allocation
4. Implementing data structure like linked list, Tree, BST etc.
5. To do System Level Programming.
6. To return multiple value.
7. Used for accessing array element.
8. To pass array argument.

11 Function Parameter and Pointers

- Problem with normal parameter passing :
  - → Changes are not reflect.
  - → The whole object is copied

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

void fun (int *x)
{
  *x = *x + 5;
}

int main ()
{
  int x = 10;
  fun (&x);

  cout << x ; // 10
  return 0;
}
```

// *p = &x (p = &x)

p has acces to the value of x.

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

void fun (int *p)
{        use here for better readibil
  *p = *p + 5;
}

int main ()
{
  int x = 10;
  fun (&x);
  cout << x ; // 15.
  return 0;
}
```

// fun (&p) = fun(&x)

p has access of x's location.

```
#include <iostream>
using namespace std;
void fun (string *s)
{
   cout << *s;
}
int main()
{
   string s = "geeksforgeek Courses";
   fun (&s);   // both share same address
   return 0;   // after using pointer no
               // whole object is copied.
}
```

(5) arr[] = *arr = pointer.

Why C/C++ treat array as pointer.

Ans: When we passing & receiving array we don't want to receive it by value, where we copy one array to another. So, to avoid this we treat it as pointer.

[8] Array Parameter and Pointers

```
#include<iostream>
using namespace std;
void fun(int arr[])
{
  int n = sizeof(arr) / sizeof(arr[i]);
  for (int i=0; i<n; i++)
    cout << arr[i] << " ";
}

int main()
{
  int arr[] = {10, 20, 30, 40};
  int n = sizeof(arr) / sizeof(arr[0]);
  for (int i=0; i<n; i++)
    cout << arr[i] << " ";
  cout << "\n";
  fun(arr);
  return 0;
}
```

Output on a compiler with 8 byte pointer size & 4 byte int size

int *arr { Pointer }

size of pointer = 8.

(*(arr + 0))

* Don't pass array as parameter it is pointer.
* So, always pass size as a seperate parameter like n.

OUTPUT

>> 10 20 30 40

>> 10 20

Compiler might throw a warning also.

→ Pointers Vs Array in C++

```
#include <iostream>
using namespace std;
int main()
{
    int arr[] = {10, 20, 30};
    int *ptr = arr;
    cout << sizeof(arr); // 12 (4x3)
    cout << sizeof(ptr); // 8
    cout << *(arr+2); // 30
    cout << arr[2];
    cout << ptr[2]; // 30
    cout << *(ptr+2);
    return 0;
}
```

address of 1st element

arr[i]
*(arr+i)

→ Recommanded Practice

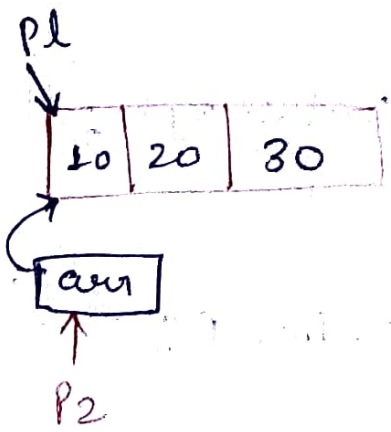
↓

Better readability

```
#include <iostream>
using namespace std;
int main()
{
  int arr[] = {10,20,30};
  int *p1 = arr;
  int (*p2)[3] = &arr;

  cout << *p1 << "\n";
  cout << **p2 << "\n";
  return 0;
}
```

address of 1st element



NOTE:

(*) int *p2 = &arr; would not compile.

(*) int *p2[3]; would mean an array of pointer of size 3.

```
int main ()
{
  int *ptr ;
    :
    :
  cout<< *ptr; || segmentation fault
    :
    :
  return 0;
}
```

// Code without NULL

```
int main ()
{
  int *ptr = NULL;
      |
      |
  if (ptr != NULL)
    cout << *ptr ;
      |
  return 0 ;
}
```

// Code with NULL

special keyword
it does not hold any address

② HOW IS NULL DEFINED ?

↳ typically defined as 0 in c++.

```
int main ()
{
  int *ptr = 0;
  return 0;
}
```

special integer value

// Compile time

```
int main ()
{
  int * ptr =10 ;
  return 0;
}
```

// Compilation Error

* Applications of NULL

1. For pointer with no valid memory address.
2. Function use NULL to return invalid output
3. In data structures like linked list, Tree etc.

---

* Some Important points

1. A NULL pointer converts to bool value false.
   And all other pointer values convert to true.

```
int *ptr = NULL;
if (ptr)
{          ↳ evaluate to false
  cout << "Not NULL";
}
else
{
}
```

2. NULL can be used for any type.

```
double * p1 = NULL;
Char *p2 = NULL;
```

Uninitialized Pointer
↓
called as Wild Pointer
↓
because it can have any memory address
↓
Segmentation fault

→ nullptr in C++

- replacement of null

(*)

```
void fun(int x) { }
void fun (int *ptr) { }
int main()
{
   fun(NULL);
   return 0;
}
```

→ it apporaches both (1) int x (2) pointer.

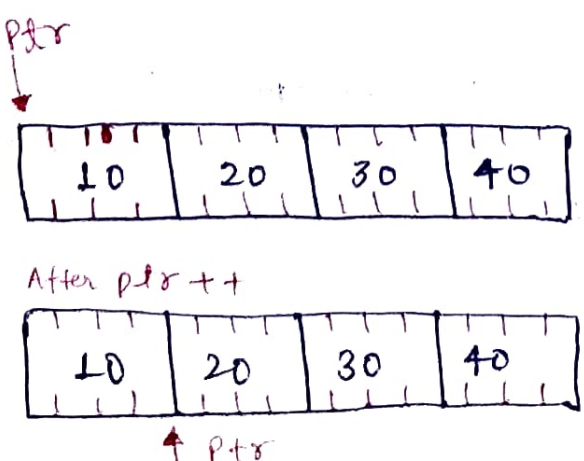
OUTPUT
Compiler Error : Ambiguous Call

- Added in C++ 11 as replacement of Null
- int x = NULL; is allowed.
- int x = nullptr; is not allowed.
- The type of nullptr is nullptr_t.

➡ POINTER ARITHMETIC

- (++), (--), (ptr + int), (ptr - int), (ptr1 - ptr2)

```
#include <iostream>
using namespace std;
int main()
{
  int arr[] = {10,20,30,40};
  int *ptr = arr;
  cout << *ptr << " " << ptr << "\n";
  ptr++;
  cout << *ptr << " " << ptr << "\n";
  ptr--;
  cout << *ptr << " " << ptr << "\n";
  return 0;
}
```



OUTPUT

```
10  0x7ffe7bca40
20  0x7ffe7bca44
10  0x7ffe7bca40
```

(b) Ptr ± int

```
#include <iostream>
using namespace std;
int main()
{
 int arr[] = {10, 20, 30, 40};
 int *ptr = arr;
 cout << *ptr << " " << ptr << "\n";
 ptr = ptr + 3;
 cout << *ptr << " " << ptr << "\n";
 ptr = ptr - 2;
 cout << *ptr << " " << ptr << "\n";
 return 0;
}
```

OUTPUT

```
10   0x7ffc7bcfa40
40   0x7ffc7bcfa4c
20   0x7ffc7bcfa44
```

(c) Ptr2 - Ptr1

```
#include <iostream>
using namespace std;
int main()
{
 int arr[] = {10, 20, 30, 40};
 int *ptr1 = arr;
 cout << *ptr1 << " " << ptr1 << "\n";
 int *ptr2 = ptr1 + 3;
 cout << *ptr2 << " " << ptr2 << "\n";
 cout << (ptr2 - ptr1) << "\n";
 return 0;
}
```

OUTPUT

```
10   0x7ffc7bcfa40
40   0x7ffc7bcfa4c
3
```

Practice Questions-

①
```
int main()
{
  int arr[]={10,20};
  int *ptr = arr;
  ++*ptr;
  Cout << arr[0]<<" "
       <<arr[1]<<" "
       <<*ptr;
  return 0;
}
```

OUTPUT
11 20 11

(*) Precedence of

++ & * = Same.

(right associative)

(*) ++*ptr ⇔ ++(*ptr)

first incr. then use.

②
```
int main()
{
  int arr[]={10,20};
  int *ptr = arr;
  *ptr++;
  Cout << arr[0]<<" "
       <<arr[1]<<" "
       <<*ptr;
  return 0;
}
```

Postfix (++) has higher precedence than *

first use then incr.

OUTPUT
10 20 20

(*) *ptr++ ⇔ *(ptr++)

③
```
int main()
{
  int arr[]={10,20};
  int *ptr = arr;
  *++ptr;
  Cout << arr[0]<<" "
       <<arr[1]<<" "
       <<*ptr;
  return 0;
}
```

OUTPUT
10 20 20

DYNAMICALLY ALLOCATED MEMORY

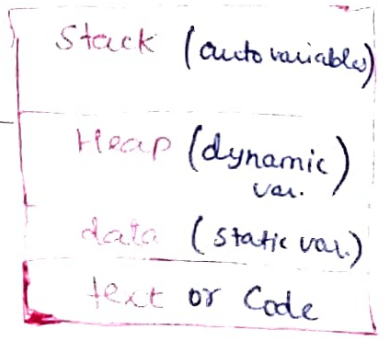
(*) Types of memory allocation

Programmer's allocate / Compiler

① Automatic (happens on fn call & go out of scope when fn is over)

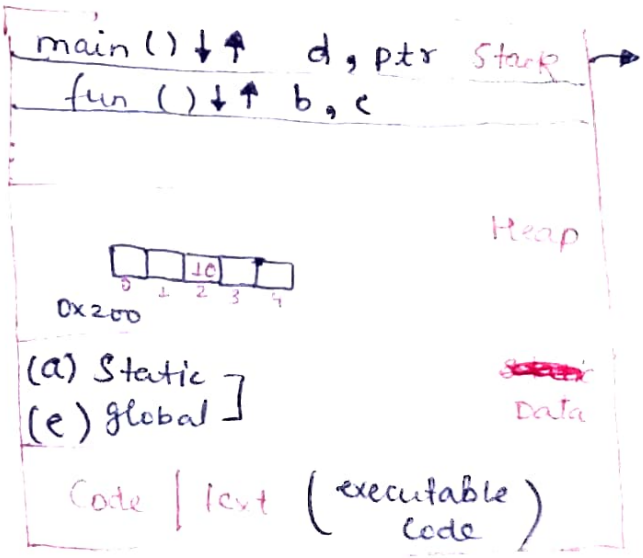
② Static (stays throughout the lifetime of program)

③ Dynamic




```
int e;  → global variable
void fun()
{
  static int a;  (static)
  int b, c;  (auto)
  ---
}

int main()
{
  int d;  (auto)
  fun();
  int *ptr = new int [5];  → [] array of 5 integers
  *(ptr+2) = 10;
  delete [] ptr;
  ptr = NULL;
  return 0;
}
```



↑↓ (when fn called activation record created & when its over the fun. goes out of scope.)

Dynamic mem. allocation

- keyword = new
- (ptr = new int / delete ptr;) → for 1 array

free the memory when we dont need.

NULL is special value for pointer → says, ptr has no access to any memory.

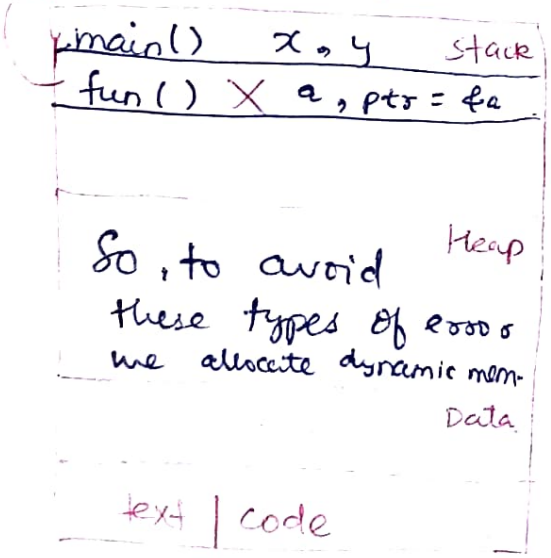
(*) int * fun ( )

```
{
    int a = 10 ;
    int * ptr = &a ;
    return ptr ;
}

int main ( )
{
int x, y ;
cout << * fun ( ) ;
return 0 ;
}
```

store in stack

When fun() return then this fn go out of scope.



OUTPUT

>> Segmentation fault

>> Garbage Value

>> Error

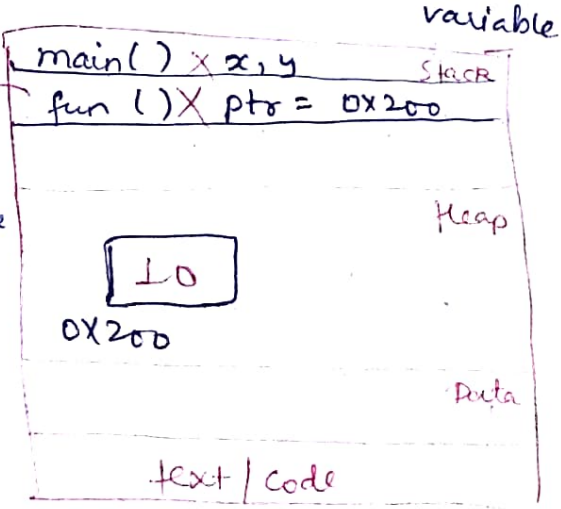
→ by allocating dynamic memory.

```
int * fun ( )
{
    int * ptr = new int ;
    * ptr = 10 ;
    return ptr ;
}
int main ( )
{
  int x, y ;
  cout << * fun ( ) ;
  return 0 ;
}
```

Store in Heap

int *ptr = fun(); delete ptr;

To delete the dynamically allocated variable ↓ Get address of it in another pointer variable



when fun() go out of scope our progr. run. Because we store out variable in Heap. it present here until we delete it.

OUTPUT

10

MEMORY LEAK

- When we allocating memory & do not free up.
- Suppose, our program was in loop & dynamically allocated & when we do not delete it our memory gets fulls and our program may crash.
- eg:-

```
void releaseConn (int *ptr) { delete ptr; };
int * CrackConn ()
{
  int *ptr = new int;
  return ptr;
}
int main ()
{
  while (1)
  { int *ptr = CreateConn ()
    //do some work
  }
}
```

→ releaseConn (ptr); → No memory leak.

More On `new`

- an operator
- return the pointer to the memory allocated.
- always used for dynamic memory allocation.
- Call Constructs for object of class / struct.
- Can initialize value also.
- `int *ptr = new int (5);` // 5 (0X200.)

STRING

- Sequences of character, eg. "gfg", "courses", etc.
- Small set (typically) and no separator required.
- 'A' to 'Z' have values from 65 to 90
- 'a' to 'z' have values from 97 to 122

(Q) Why are we having seperate name string ? why do we not simply called an array of characters?

Reason is string has small sets (256) and thir is no seperator required.

NOTE:- String are stored as integer values.

(*) string

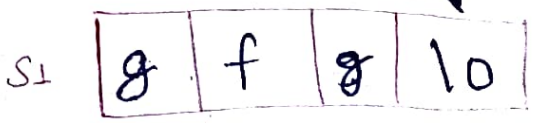
- c style string in c++.
- String class in c++

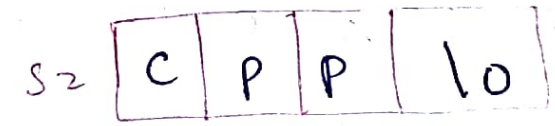
(1) C style string in C++

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

int main()
{
  char s1[] = "gfg";
  char s2[] = {'C','P','P','\0'};
  cout << strlen(s1) << "\n";
  cout << strlen(s2) << "\n";
  return 0;
}
```

here program stop.





backslash 0 (\0) is automatically abended.

because of " "

if we remove → compiler error

that fn work on the effect of their last element must be \0

(*) C style string is basically character array & their last element is '\0'.

- String Terminator
- ASCII value = 0.

OUTPUT
3
3

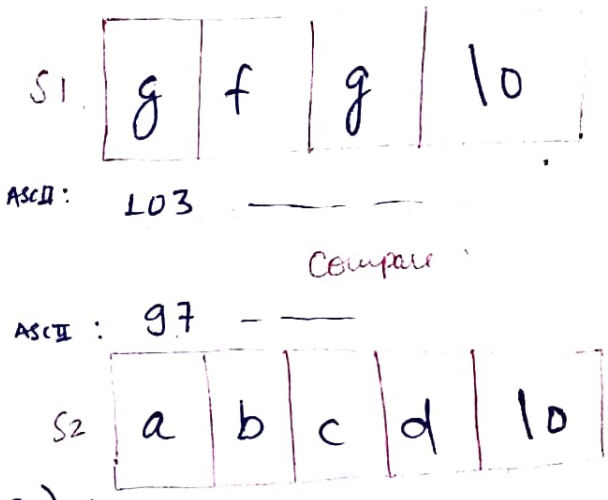
(*) Strlen(s1)

↳ fn which give length of string.

(b) checking lexicographically (res)

→ Compare ASCII value b/w two element.

```
# include <cstring>
# include <iostream>
using namespace std.
int main()
{
  char s1[] = "gfg";
  char s2[] = "abcd";
  int res = strcmp(s1, s2);
  if (res == 0)
    {
     cout << "same";
    }
  else if (res < 0)
    {
     cout << "smaller";
    }
  else
    {
     cout << "Greater";
    }
  return 0;
}
```



(*) if ASCII value = same
res == 0 (s1 = s2)

(*) if s1 > s2 (in ASCII)
res > 0

(*) if s1 < s2 (in ASCII)
res < 0.

OUTPUT

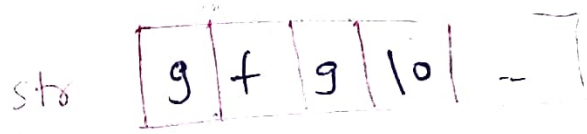
>> Greater

© Note :- we can not do :

```
Char str[5];
str = "gfg";
```

So, we use strcpy( ) in C style string.

```
# include <iostream>
# include <cstring>
using namespace std;
int main()
{
  char str[5];
  strcpy (str, "gfg");
  cout << str;
  return 0;
}
```



it take gfg and copy in str.

OUTPUT

gfg

→ String class in C++

```
# include <iostream>
using namespace std;
int main()
{
    string str = "geeksforgeeks";
    cout<< str ;
    return 0;
}
```

OUTPUT

geeksforgeek.

(*) Advantages Over C-style string

- → we do not have to worry about size.
- → can assign a string later.
- → Support of operator like +, <, >, ==, <= and >=.
- → Richer library
- → Can be converted to C-style string if needed.

g	e	e	k	s	f	o	r	g	e	e	k	s	\0

→ String [illegible] Operation

— length ( )
— substr (1, 3)
— find ("—")

```
#include <iostream>
using namespace std;
int main()
{
    String str = "gfg";
    cout << str.length() << endl;
    str = str + "xyz";
    Cout << str << endl;
    Cout << str.substr(1, 3) << endl;
    cout << str.find("fg");
    return 0;
}
```

OUTPUT

```
>> 3
>> gfgxyz
>> gfg
>> 1
```

String Comparision

```
#include <iostream>
using namespace std;
int main()
{
    string str = "geeksforgeek";
    int res = str.find("eek");
    if (res == string::npos)
        cout << "Not Present \n";
    else
        cout << "First occurence"
             << "at index" << res;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    string s1 = "abc";
    string s2 = "bcd";
    if (s1 == s2)
        cout << "Same";
    else if (s1 < s2)
        cout << "Smaller";
    else
        cout << "Greater";
    return 0;
}
```

getline in string

```
# include <iostream>
using namespace std;
int main ()
{
   string name ;
   Cout << "Enter your name";
   getline (cin , str);
   Cout << "Your name is"
                << str ;
   return 0;
}
```

OUTPUT

Your name is Vikas Kumar.

```
getline (cin , str, 'i' );
```

When we inter this character in input.

input is stop at this point.

String Iterator

```
# include <iostream >
using namespace std;
int main ()
{
   string str = "gfg";
for (int i=0; i<str.length (); i++)
      Cout << str [i] ;
      Cout << endl ;
for (char x :. str)
      cout << x ;
      Cout << endl ;
for (auto it = str. begin ();
     it. != end (); it ++ )
   Cout << (* it) ;
return 0;
}
```

OUTPUT

Practice

1-a Reverse a String

Q-2 Check for Palindrome

Q-4 String : Binary to Decimal

[R-5] String : Decimal to Binary

Q-6 Find an extra character in string.

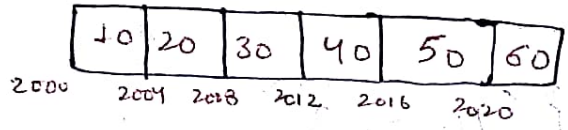
Q-7 String : Pangram checking

Q-3 Check wheather two string are anagram of each other.

→ Multidimensional Array

(*) Imp Points

[1] Elements are stored in row-major order.



[2] Internal Curly Brackets are optional.

```
int arr[3][2] = {10,20,30,40,50,60};

                (OR)

{ {10,20},
  {30,40},
  {50,60} }
```

[3] Only the first dimensions can be Omitted when we initialize.

```
int arr[ ][2] = { {1,2} {3,4} }

int arr[ ][2][2] = { { {1,2}, {3,4} },
                     { {5,6}, {7,8} } }
```

```
#include <iostream>
using namespace std;
int main()
{
  int arr[3][2] = {{10,20},
                   {30,40},
                   {50,60}};

  for (int i=0; i<3; i++)  // Row
    for (int j=0; j<2; j++)  // Coulumn.
      {
      }
       cout << arr[i][j] << " ";
  return 0;
}
```

Recommanded Practice.

OUTPUT
10 20 30 40 50 60

* arr [i] [j]

$$\begin{pmatrix} [00] & [01] \\ [10] & [11] \\ [20] & [21] \end{pmatrix} = \begin{pmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{pmatrix}$$

(*) Variable Sized array

(from C++ 14 standard it allows user to write some variables inside the array dimensions)

```
# include <iostream>
using namespace std;
int main ()
{
   int m = 3 , n = 2 ;
   int arr [m] [n] ;
   for (int i = 0 ; i < m ; i++)
      for (int j = 0 ; j < n ; j++)
         arr [i] [j] = i+j ;

   for (int i = 0 ; i < m ; i++ ) // Row
      for (int j = 0 ; j < n ; j++ ) // Coulmn
         Cout << arr [i] [j] << " " ;
   return 0 ;
}
```

for values of matrix

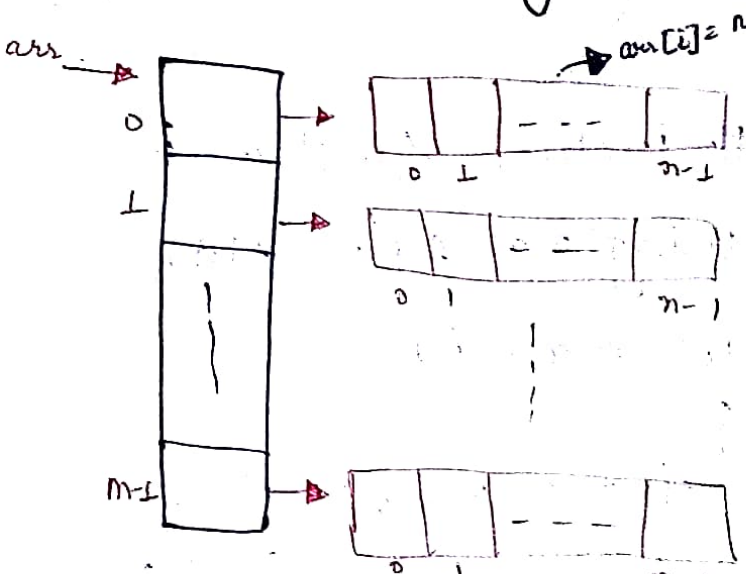
OUTPUT

0 1 1 2 2 3

→ OTHER WAYS TO CREATE

1 Double Pointer

** arr = array of pointers




```
# include <iostream>
using namespace std;
int main()
{
  int m = 3, n = 2;
  int ** arr;
  arr = new int * [m]; // arr store rows.
  for (int i=0 ; i<m ; i++)
    arr [i] = new int [n] ; // vertical row array store columns in it.
  for (int i=0; i<m ; i++)
    for (int j=0 ; j<n ; j++)
    {
      arr [i][j] = 10;
      Cout << arr [i] [j] <<" ";
    }
}
```

m x n

value + output

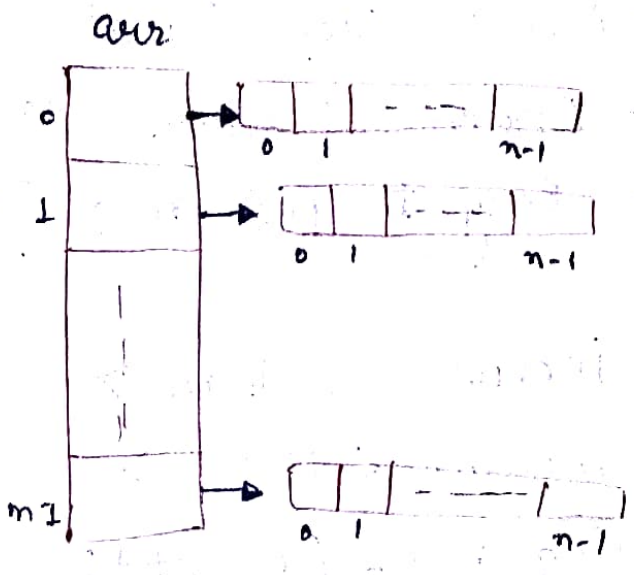
Adv → for every row we can write diff. values.

Disadv → Not cache friendly. Not be in Contingous location

OUTPUT

10 10 10 10 10 10

[2] Array of Pointer




```
#include <iostream>
using namespace std;
int main()
{
    int m = 3, n = 2;
    int * arr [m];
    for (int i = 0; i<m ; i++)
        arr [i] = new int [n];

    for (int i = 0; i<m; i++)
        for (int j = 0 ;j<n ; j++)
            {
                arr [i] [j] = 10;
                cout << arr[i][j]<< " ";
            }

    return 0;
}
```

m x n

Value + Output

OUTPUT
10 10 10 10 10 10

3 Arrays of Vectors

- Not as cache friendly as simple 2-D arrays.
- Individual rows are of dynamic sizes.
- Easy to pass to a function.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  int m=3, n=2;
  vector<int> arr[m];
  for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
      arr[i].push_back(10);
  for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
      cout<<arr[i][j]<<" ";
}
```

OUTPUT

10 10 10 10 10 10

[4] Vector to Vector

↳ Numbers of rows can also be dynamic.

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

int main()
{
   int m=3, n=3;
   vector<vector<int>> arr;
   for (int i=0; i<m; i++)
      {
         vector<int> v;
       for (int j=0; j<n; j++)
          v.push_back(10);
       }
   arr.push_back(v);

  for (int i=0; i<arr.size(); i++)
    for (int j=0; j<arr[i].size(); j++)
       cout<<arr[i][j]<<" ";
}
```

OUTPUT
10 10 10 10 10 10

→ Passing 2D array as argument in C++

(*) Method 1

```
void print (int mat[3][2])
{
  for (int i=0 ; i<3 ; i++)
    for (int j=0 ; j<2 ; j++)
    {
      cout << mat[3][2] << " ";
    }
}

int main()
{
  int mat[3][2] = {{10,20},
                   {30,40},
                   {50,60}};
  print (mat);
  return 0;
}
```

OUTPUT

```
10 20 30 40 50 60
```

But this code is only valid for 3×2 matrix, so, we use default argument to taking inputs.

(+) Taking argument for row m.

```
void print (int mat[][2], int m)
{
  for (int i=0; i<m ; i++)
    for (int j=0; j<2 ; j++)
    {
      cout << mat[i][j] << " ";
    }
}

int main ( )
{
  int mat[][] = {{10, 20},
                 {30, 40},
                 {50, 60}};
  print (mat, 3)
  return 0;
}
```

Here, 1st dimension is Omitted.