UNIT 2
BASICS OF C++ PROGRAMMING

LH – 5 HRS

PRESENTED BY:

ER. SHARAT MAHARJAN

OBJECT ORIENTED PROGRAMMING (OOP)

CONTENTS (LH – 5HRS)

2.1 C++ Program Structure

```
//My first program in C++              -> comments.
#include<iostream>                     -> not regular code lines but are directives for the preprocessor
#include<conio.h>                         tells preprocessor to include the iostream standard file.

using namespace std;                   -> namespaces allow to group entities under a name and tells the
                                          compiler to use the std namespace.

int main(){                            -> point by where all C++ programs start their execution.
        cout<<"Hello World!";          -> causes the message "Hello World" to be displayed on the screen.
        getch();                       -> holds the program until a key is pressed.
        return 0;                      -> terminates main( )function and causes it to return the value 0 to the
                                          calling process.
}
```

Output

```
Hello World!
```

2.2 Character Set and Tokens

1. <u>Character Set:</u>

- It is a set of **valid characters that a language can recognize**. A character represents any **letter, digits or any other sign**.
- C++ has the following character set:
  - **<u>Letters</u>**: Letters refer to the 26 alphabets that used in the English language. Both uppercase (**A-Z**) and lowercase (**a-z**) alphabets can be used in C++.
  - **<u>Digits</u>**: Digits from **0-9** or **any combination** of these can be used.
  - **<u>Special Characters/Symbols</u>**: Apart from the usual letters and digits, we have a set of **special characters** that can be used in C++. These are:
    - + − ∗ ∕ ^ \ ( ) [ ] { } = != < > . ' " $, ; : % ! & _ # <= >= @
  - **<u>White Spaces</u>**: These are a special set of characters that are mainly used for the purpose of formatting the text in the C++ program. They are also called formatting characters. Some common examples are **blank space, horizontal tab, newline etc.**

2. Tokens:

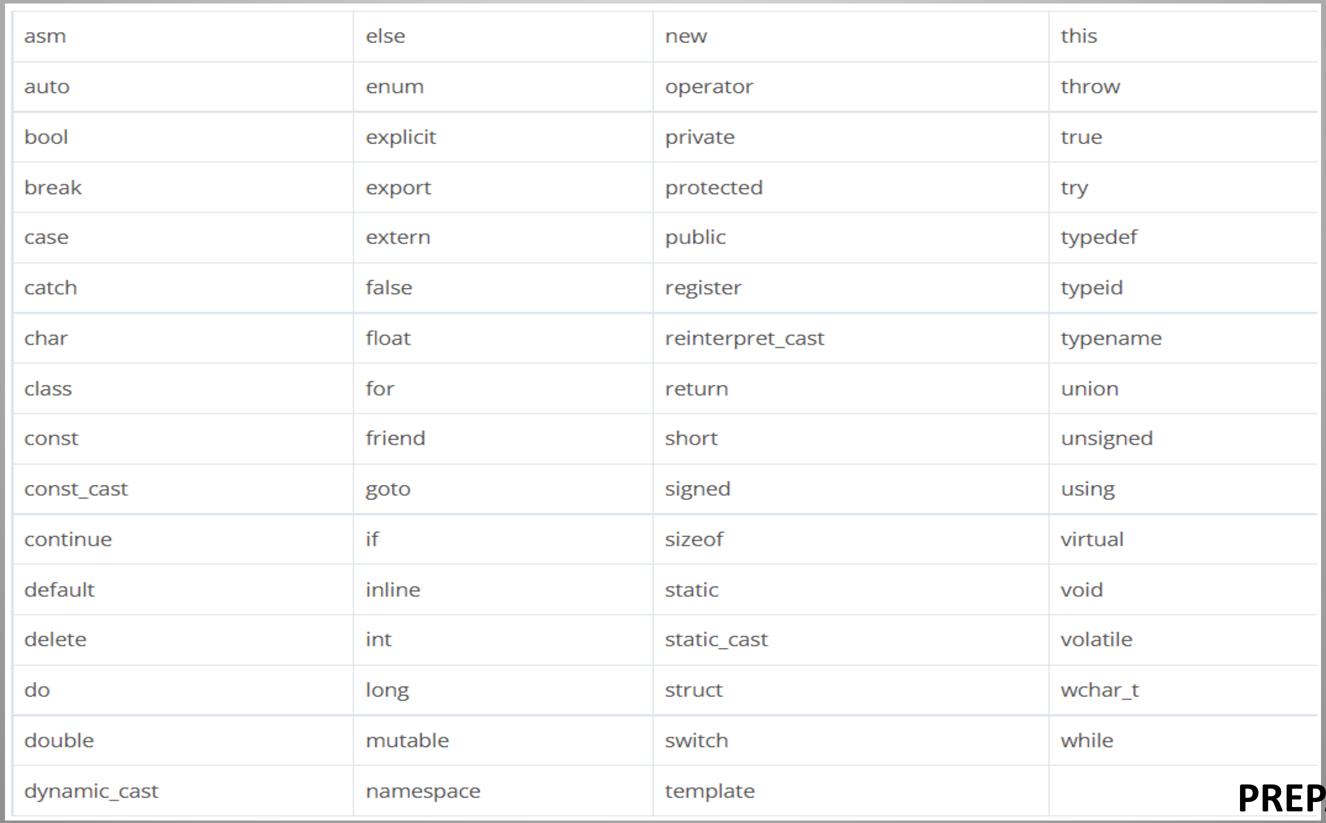
The **smallest individual units in a program** are known as tokens.

C++ has the following tokens:

a. Keywords
b. Identifiers
c. Constants
d. Operators

a. Keywords:

- They are set of reserved words that **one cannot use as an identifier**. These words are known as "**reserved words**" or "**Keywords**". We **cannot use keywords as variable names, class names, or method names, or as any other identifier.**
- Below is list of available keywords in C++ programming Language –



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	

b. Identifiers:

- An identifier is a name given to **program elements such as variables, array, class and functions etc**. An identifier is a **sequence of letters, digits, and underscores, the first character of which can not be a digit**.

Following **rules** must be kept in mind while naming an identifier.

- The **first character** must be an **alphabet** (uppercase or lowercase) or can be an **underscore.**
- An identifier **can not start with a digit**.
- All succeeding characters must be alphabets or digits.
- **No special characters** like !, @, #, $, % or punctuation symbols is allowed **except the underscore**"_".
- **No two successive underscores** are allowed.
- **Keywords can not be used** as identifiers.

c. Constants:

- A constant is any expression that has a **fixed value**. They can be divided in **Integer** numbers, **Floating-Point** numbers, **Characters and Strings.**
- Integer numbers: 1776, 707, -273 etc.
- Floating-Point numbers: 3.14, 6.02e23=$6.02*10^{23}$ etc.
- Non-Numeric constants: ‘z’, “Hello World” etc.
- Escape characters: \n, \t etc.

d. Operators:

An operator is a **special symbol that is used to carry out some specific operation on its operand**. In C++, we have rich set of built in operators to carry out different type of operations. There are **operators for assignment, arithmetic operations, logical operations and comparison operations etc**. Operators can be used with many types of variables or constants, but some of the operators are restricted to work on specific data types. Most operators are **binary,** meaning they take two operands, but a few are **unary** and only take one operand.

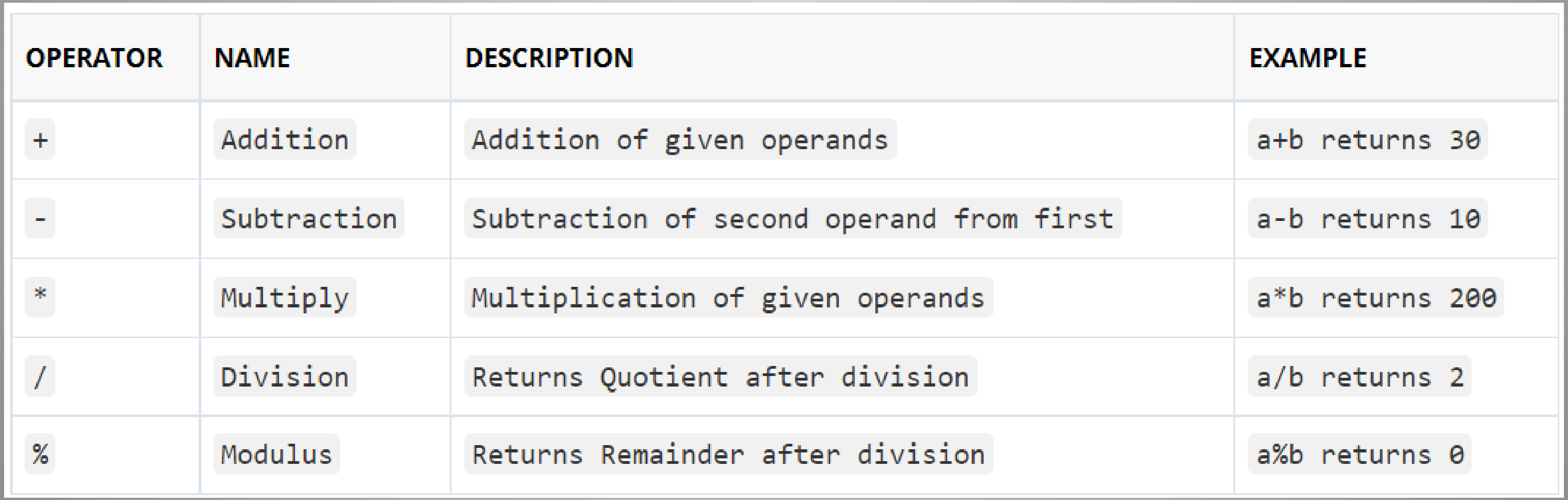
They can be classified as:

1. Arithmetic operators
2. Assignment operators
3. Unary operators
4. Comparison operators
5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

1. Arithmetic operators:

Arithmetic Operators are used to **perform arithmetic operations** like **addition, subtraction, multiplication, division and modulus**.

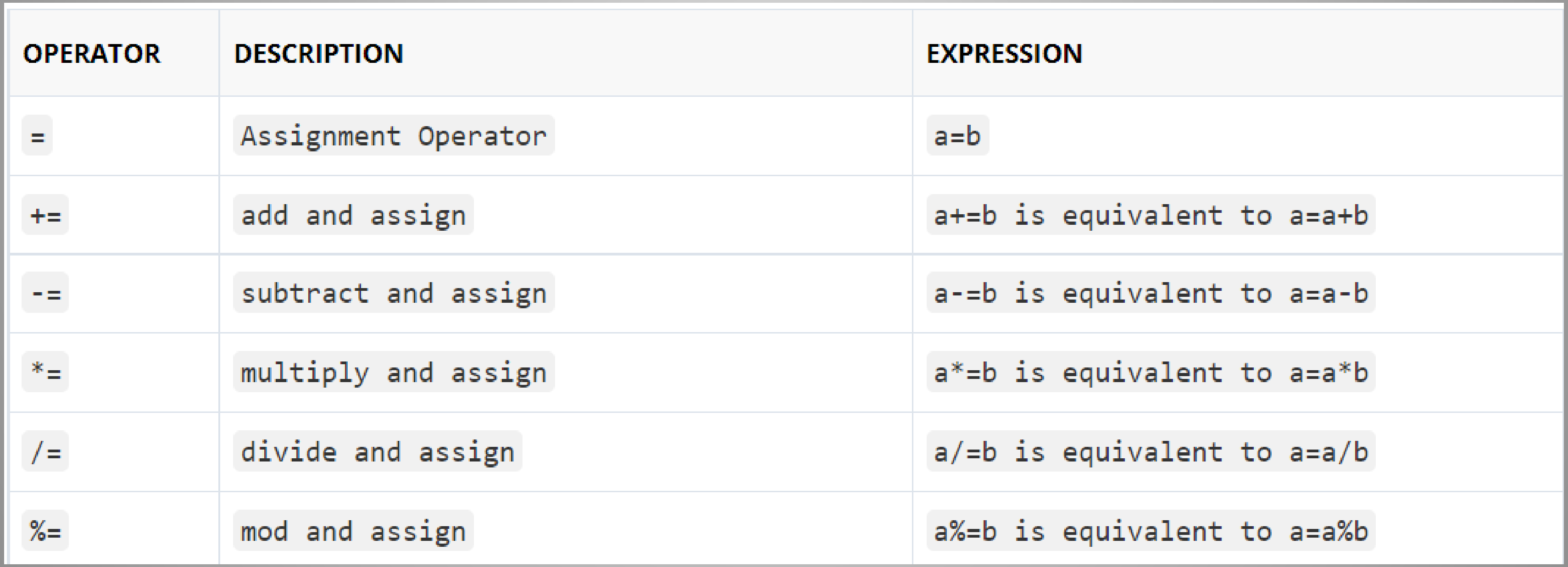
Let variable **a** holds 20 and variable **b** holds 10, then –



OPERATOR	NAME	DESCRIPTION	EXAMPLE
`+`	`Addition`	`Addition of given operands`	`a+b returns 30`
`-`	`Subtraction`	`Subtraction of second operand from first`	`a-b returns 10`
`*`	`Multiply`	`Multiplication of given operands`	`a*b returns 200`
`/`	`Division`	`Returns Quotient after division`	`a/b returns 2`
`%`	`Modulus`	`Returns Remainder after division`	`a%b returns 0`

2. Assignment operators:

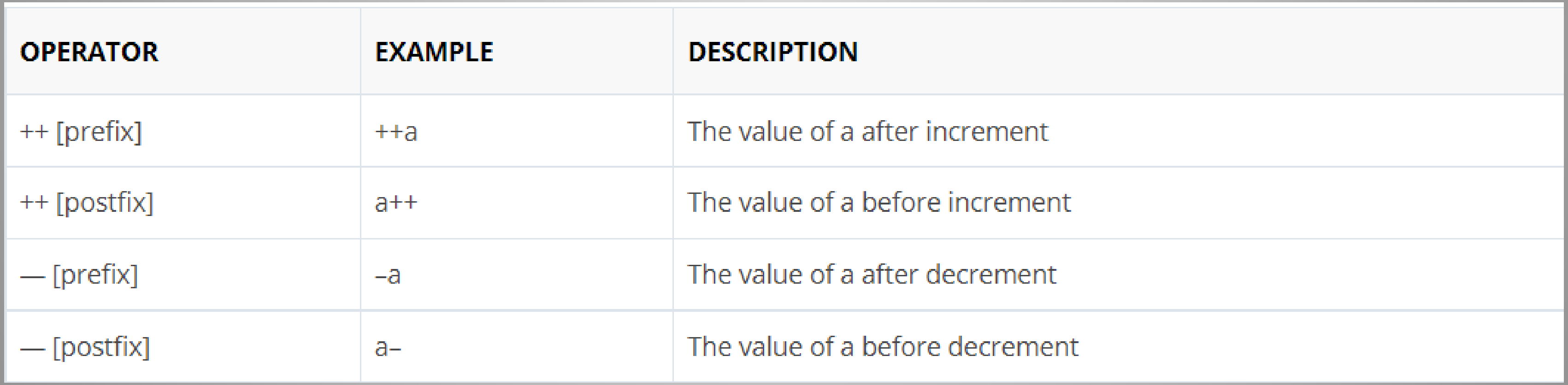
Assignment operators are used to assign value to a variable, one can assign a variable value or the result of an arithmetical expression.



OPERATOR	DESCRIPTION	EXPRESSION
`=`	`Assignment Operator`	`a=b`
`+=`	`add and assign`	`a+=b is equivalent to a=a+b`
`-=`	`subtract and assign`	`a-=b is equivalent to a=a-b`
`*=`	`multiply and assign`	`a*=b is equivalent to a=a*b`
`/=`	`divide and assign`	`a/=b is equivalent to a=a/b`
`%=`	`mod and assign`	`a%=b is equivalent to a=a%b`

3. Unary operators:

In C ++, ++ and -- are know as increment and decrement operators respectively. These are unary operators it means they works on **single operand**. ++ adds 1 to operand and -- subtracts 1 to operand respectively.

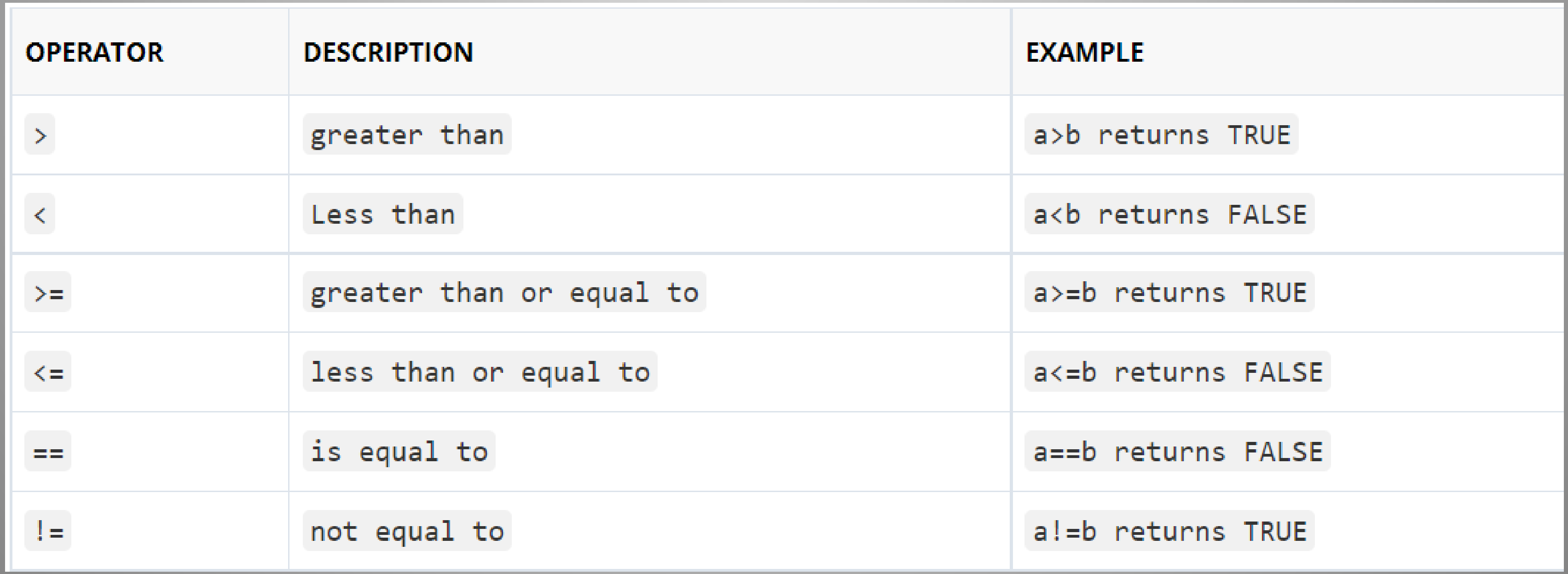


OPERATOR	EXAMPLE	DESCRIPTION
++ [prefix]	++a	The value of a after increment
++ [postfix]	a++	The value of a before increment
— [prefix]	–a	The value of a after decrement
— [postfix]	a–	The value of a before decrement

4. Comparison operators:

Comparison Operators are used evaluate a **comparison between two operands**. The **result** of a comparison operation is a **Boolean value that can only be true or false**. Comparison Operators are also referred as **relational operators**.

Let variable **a** holds 20 and variable **b** holds 10, then –



OPERATOR	DESCRIPTION	EXAMPLE
`>`	`greater than`	`a>b returns TRUE`
`<`	`Less than`	`a<b returns FALSE`
`>=`	`greater than or equal to`	`a>=b returns TRUE`
`<=`	`less than or equal to`	`a<=b returns FALSE`
`==`	`is equal to`	`a==b returns FALSE`
`!=`	`not equal to`	`a!=b returns TRUE`

5. Shift operators:

It works on individual bits in a byte and only on integer data type.

Operator	Description	Example	Explanation
<<	Shifts bits to left, filling zeroes at right	a=10<<3	$10*2^3=80$
>>	Shifts bits to right, filling zeroes at left	a=10>>3	$10/2^3=1$

10=00001010

When doing a **left shift by 3(10<<3),** the result is “**01010000**”, which is equivalent to **80**.

When doing a **right shift by 3(10>>3),** the result is “**00000001**”, which is equivalent to **1**.

6. Bit-wise operators:

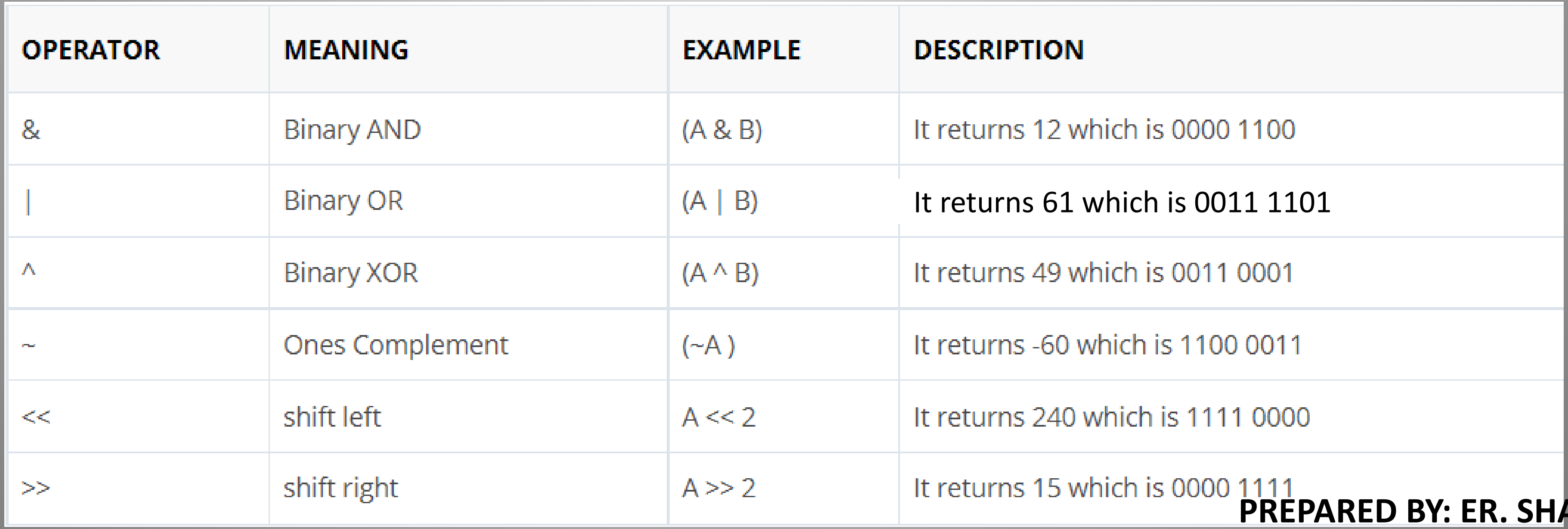
Bitwise operator are used to perform **bit level operation over its operand.**

Let A = 60; and B = 13;

Binary equivalent

A = 0011 1100

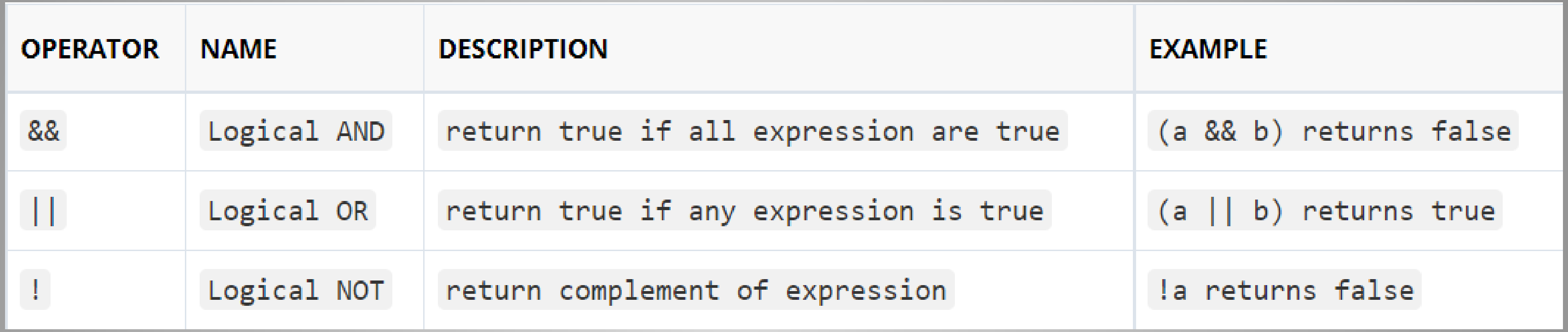
B = 0000 1101



OPERATOR	MEANING	EXAMPLE	DESCRIPTION
&	Binary AND	(A & B)	It returns 12 which is 0000 1100
|	Binary OR	(A | B)	It returns 61 which is 0011 1101
^	Binary XOR	(A ^ B)	It returns 49 which is 0011 0001
~	Ones Complement	(~A)	It returns -60 which is 1100 0011
<<	shift left	A << 2	It returns 240 which is 1111 0000
>>	shift right	A >> 2	It returns 15 which is 0000 1111

7. Logical operators:

Logical operators are used to **combine expressions with conditional statements using logical (AND,OR,NOT) operators**, which results in **true or false**. Let variable **a** holds **true or 1** and variable **b** holds **false or 0,** then −



OPERATOR	NAME	DESCRIPTION	EXAMPLE
`&&`	`Logical AND`	`return true if all expression are true`	`(a && b) returns false`
`||`	`Logical OR`	`return true if any expression is true`	`(a || b) returns true`
`!`	`Logical NOT`	`return complement of expression`	`!a returns false`

8. Conditional operators:

In C++, conditional operator is considered as short hand for **if-else** statement. Conditional operator is also called as **"Ternary Operator"**.

Syntax: condition ? result1 : result2

Example: result = (10 > 15) ? "True" : "False";

cout<<result;

Output: False

2.3 Data Type:

The various data types provided by C++ are: Basic Data Types, Derived Data Types and User-Defined Data Types.

1. Basic Data Types:

C++ has following basic data types:

a. Numbers

b. Boolean

c. Characters

d. Void

a. **Numbers:-** The Number data type is used to **hold the numeric values**. C++ supports following numerical data types:

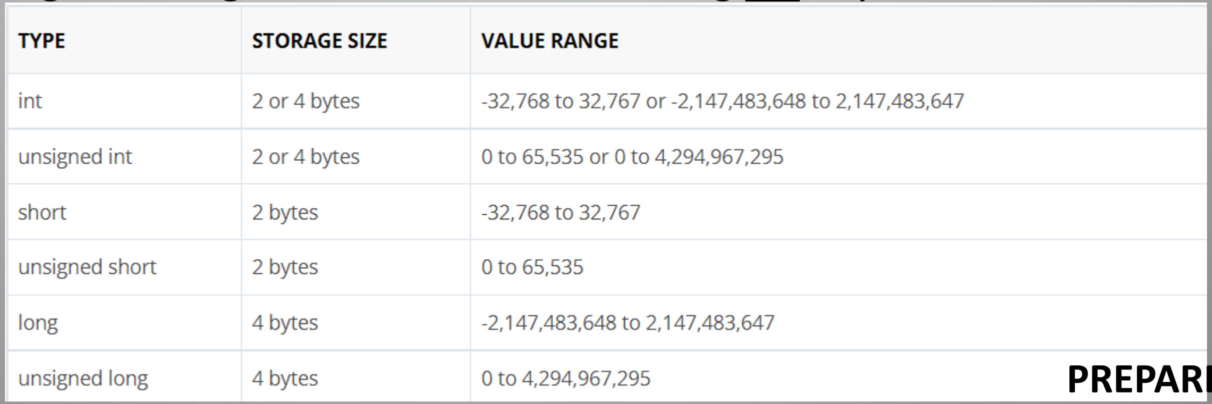
i. int:- Integers are used to store whole numbers. C++ supports several integer types, varying internal sizes for storing signed and unsigned integers. Integers can be declared using **int** keyword.

Syntax:- int <variable name>;

Example:- int num1;

short int num2;

long int num3;



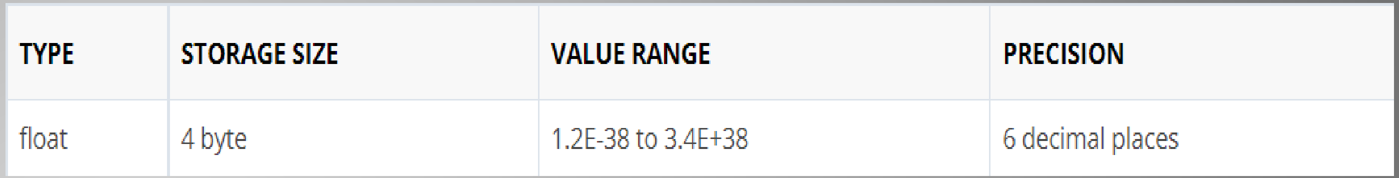
TYPE	STORAGE SIZE	VALUE RANGE
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

ii. float:-

A Float type is used to store numbers that contain a **decimal component** (real numbers). Float can be declared using **float** keyword.

Syntax:- float <variable name>;

Example:- float num1;



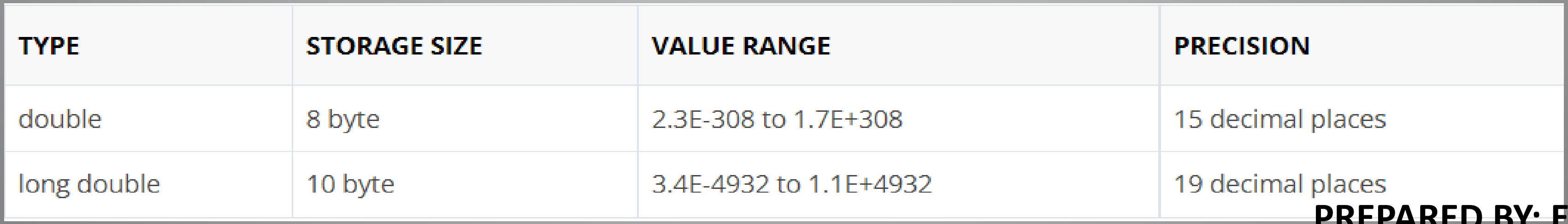
TYPE	STORAGE SIZE	VALUE RANGE	PRECISION
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places

iii. double:-

In C++, double is used to represent a floating-point number and numbers with larger decimal points. Double can be declared using **double** keyword.

Syntax:- double <variable name>;

Example:- double num1;



TYPE	STORAGE SIZE	VALUE RANGE	PRECISION
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

b. <u>Boolean:</u>

The Boolean data type is used to represent the **truth values**, which can be either **True or False**. Boolean are commonly used in decision making statements. Boolean can be declared using **<u>bool</u>** keyword.

Syntax:- bool <variable name>;

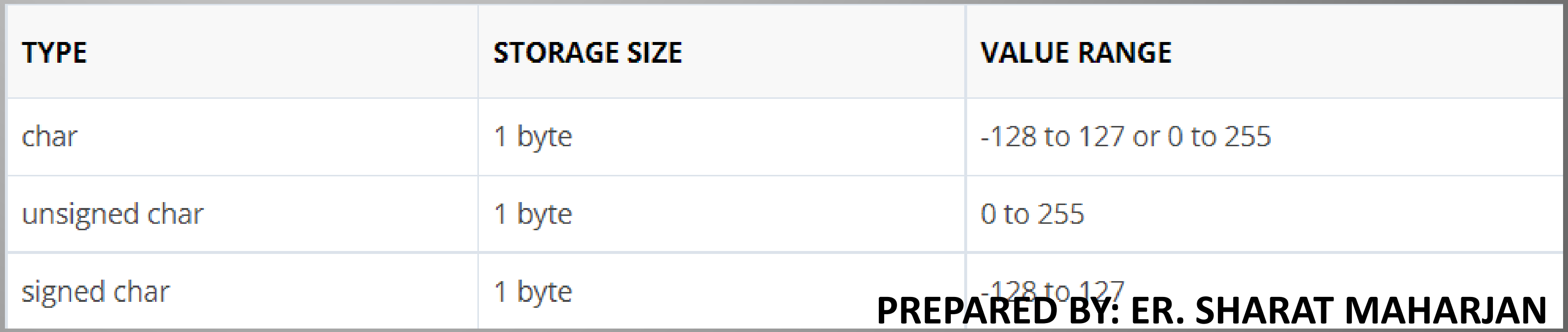
Example:- bool b1 = true;   // declaring a boolean with true value

c. <u>Characters:</u>

The character data type is used to hold the **single literal**. Character are declared **<u>char</u>** keyword.

Syntax:- char <variable name>;

Example:- char ch = 'a’;



TYPE	STORAGE SIZE	VALUE RANGE
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127

d. Void:

The void type means **no values**. It is usually used with function to specify its return type or its arguments.

Example:

```
void echo_square(int Number)
{
  cout<<Number*Number;
}
```

2. Derived Data Types:

Data types that are **derived from the built-in data types** are known as derived data types. The various derived data types provided by C++ are **arrays, references and pointers**.

3. User-Defined Data Types:

Various user-defined data types provided by C++ are **structures, unions, enumerations and classes.**

2.4 Type Conversion

Type conversion is basically way to **convert an expression of a given type into another type**. In C++, there are two types of type conversion –

1. Implicit conversion
2. Explicit conversion

1. <u>Implicit conversion:</u>

In implicit or **automatic conversion** compiler will automatically change one type of data into another. Typecasting should always be used in right order (low to higher datatype). Below is the right order for numerical conversion.

int -> long -> float -> double -> long double

```
int x=3;
float y=2.5,z;
z=x+y;          //z=5.5
```

2. Explicit conversion (Type Casting):

It is the forceful conversion from one type to another type. C++ permits explicit type conversion of variables or expressions as follows:

Syntax 1: **type (expression)** //functional casting

Example:

```
double x = 10.3;
int y;
y = int (x);    // y=10
```

Syntax 2: (type) expression //c-like casting

Example:

```
double x = 10.3;
int y;
y = (int) x;    // y=10
```

2.5 Preprocessor Directives

- The preprocessors are the directives, **which give instructions to the compiler to preprocess the information before actual compilation starts.**
- All preprocessor directives **begin with #,** and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do **not end in a semicolon (;).**

1. <u>The #define Preprocessor Directive:</u>

- The #define preprocessor directive **creates symbolic constants**. The symbolic constant is called a **macro** and the general form of the directive is −

  #define macro-name replacement-text

- When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −

```
#include<iostream>
#include<conio.h>
#define PI 3.14
using namespace std;
int main(){
        cout<<"Value of PI="<<PI<<endl;
        getch();
        return 0;  }            //output: Value of PI=3.14
```

2. The #include Preprocessor Directive:

- When the preprocessor finds an **#include directive it replaces it by the entire content of the specified header or file.**
- There are two ways to use #include:

#include <header>

#include "file"

- In the **first case**, a header is specified between angle-brackets <>. This is used to include headers provided by the implementation, such as the headers that compose the **standard library (iostream, string etc).**
- The syntax used in the **second #include uses quotes and include a file**.

2.6 Namespace

- Namespaces allow us to **group named entities that otherwise would have *global scope* into narrower scopes**, giving them ***namespace scope***. This allows organizing the elements of programs into different logical scopes referred to by names. **Namespace is a feature added in C++ and not present in C.**
- A namespace is a declarative region that **provides a scope to the identifiers (names of the types, function, variables etc) inside it**.
- A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name{
int x,y;
}
```

Suyash

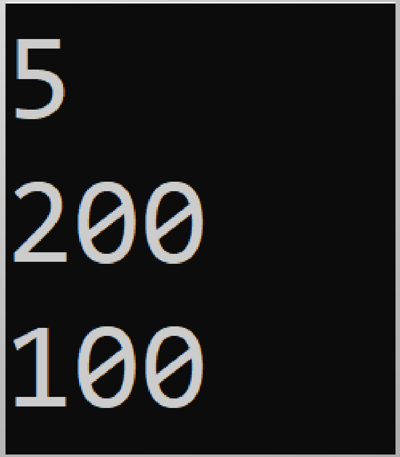
```
// Creating namespaces
#include <iostream>
using namespace std;
namespace ns1
{
        int value() { return 5; }
}
namespace ns2
{
        const double x = 100;
        double value() { return 2*x; }
}
int main()
{
        // Access value function within ns1
        cout << ns1::value() << '\n';

        // Access value function within ns2
        cout << ns2::value() << '\n';

        // Access variable x directly
        cout << ns2::x << '\n';

        return 0;
}
```

OUTPUT:

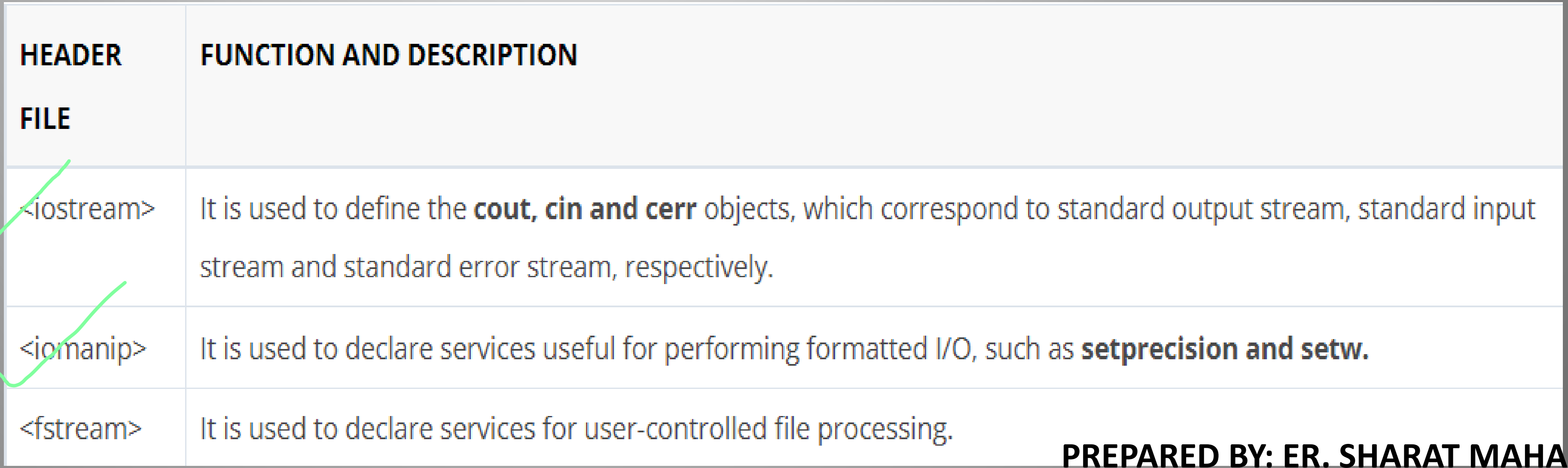


```
5
200
100
```

2.7 Input/Output Streams and Manipulators

1. Input/Output Streams:

- 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**.
- In C++, we have following header files important for basic I/O operation:



HEADER FILE	FUNCTION AND DESCRIPTION
<iostream>	It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as **setprecision and setw.**
<fstream>	It is used to declare services for user-controlled file processing.

There are following standard streams are available to all C++ programs –

1. cin (standard input)
2. cout (standard output)
3. cerr (standard error)
4. clog (standard log)

1. C++ Standard Input Stream (cin):

- The cin is basically a predefined object of **istream** class. The cin object in conjunction with stream extraction operator (>>) allows us to accept/extract data from standard input device(keyboard).

Example:-

```
#include <iostream>
using namespace std;
int main( ) {
 int age;
  cout << "Enter your age: ";
  cin >> age;            //scanf() in C
  cout << "Your age is: " << age << endl;
}
```

Output:-

```
Enter your age: 35
Your age is: 35
```

2. C++ Standard Output Stream (cout):

- The cout is basically a predefined object of **ostream** class. The cout object in conjunction with stream insertion operator (<<) allows us to send data to standard output device(screen).

Example:-

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

Output:- Hello World!

3. C++ Standard Error Stream (cerr):

- The **cerr** is also a predefined object of **ostream** class. The cerr object in conjunction with stream insertion operator (<<) allows us to send un-buffered error messages to standard error device(screen).

Example:-

```
#include <iostream>
using namespace std;
int main( ) {
   char str[] ="Unable to read...";
   cerr<<"Error message: "<<str<<endl;
   return 0;
}
```

Output:-

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

4. C++ Standard Log Stream (clog):

- The **clog** is also a predefined object of **ostream** class. The clog object in conjunction with stream insertion operator (<<) allows us to send buffered error messages to standard error device(screen) until the buffer(temporary placeholder) is filled or until the buffer is flushed.

Example:-

```
#include <iostream>
using namespace std;
int main( ) {
  char str[] ="Unable to read...";
  clog<<"Error message: "<<str<<endl;
  return 0;
}
```

Output:- Error message: Unable to read...

2. Manipulators:

They are the operators used with the insertion operator (<<) **to modify or manipulate the way data is displayed**. The most commonly used manipulators are **endl, setw and setprecision**. They are defined in the header file "**iomanip.h**" therefore we need to include it before using manipulators. Manipulator **"endl" is also defined in iostream.h therefore if we want to use only "endl", we do not need to include "iomanip.h".**

a. The endl Manipulator:

It has the same effect as using the newline character '\n'.

Cout<<"Hello World."<<endl;   //display Hello World. on screen.

b. The setw Manipulator:

This manipulator causes the output stream that follows it to be printed within a **field of n characters wide, where n is the argument to setw. Right justified.**

For example,

```
cout<<“Hello”<<endl<<“World”;
```

Output:

```
Hello
World
```

Effect of using setw:

```
cout<<setw(11)<<“Hello”<<endl<<setw(11)<<“World”;
```

Output:

```
      Hello
      World
```

c. The setprecision Manipulator:

This manipulator sets the **n digits of precision to the right** of the decimal point to the floating point output, where **n is the argument to setprecision(n).** For example,

```
    float a=2.123456;
    cout <<setprecision(2)<<a;        //output= 2.12
```

The **header file for setw and setprecision manipulators is iomanip.h.**

2.8 Dynamic Memory Allocation

Suyash

C++ defines **two unary operators new and delete** that perform the task of **allocating and freeing the memory**. They are also known as free store operators. A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete.

Syntax:

```
new data-type;
delete variable-name;
```

```
//Program to demonstrate DMA
#include <iostream>
using namespace std;
int main () {
   double *pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable
   *pvalue = 29494;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;
   delete pvalue;         // free up the memory.
   return 0;
}
```

2.9 Control Statements

C++ supports the following three control structures:

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

1. Sequential structure:

In this structure, the sequence statements are executed one after another from top to bottom in the order in which they are written.

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

In this case statement 1 is executed before statement 2, statement 2 is executed before statement 3 and so on.

2. Selection structure:

This structure makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression. The two structures of this type are: **if** and **switch**.

a. The if statements:

It is used to execute an instruction or block of instructions only if a condition is fulfilled. Its form is:

```
        if(condition)
        statement;
```

Where, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues on the next instruction.

For example,

```
if(x==100)
cout<<"x is 100";
```

b. The if…else statement:

We can additionally specify what we want that happens if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

```
    if(condition)
    statement1;
    else
    statement2;
```

For example:

```
    if(x==100)
    cout<<“x is 100”;
    else
    cout<<“x is not 100”;
```

c. The else…if Ladder:

The if+else structures can be concatenated with the intention of verifying a range of values.

The following example shows its use telling if the present value stored in x is positive, negative or none of the previous, that is to say, equal to zero.

```
if(x>0)
cout<<“x is positive.”;
else if(x<0)
cout<<“x is negative.”;
else
cout<<“x is 0”;
```

d. The switch statement:

It is also called multi way decision statement.

The syntax is of the form:

```
          switch(expression)
          {
                    case value1:
                              statement(s);
                              break;
                    case value2:
                              statement(s);
                              break;
                    ……………..
                    case valuen:
                              statement(s);
                              break;
                    default:
                              statement(s);
          }
```

In this case, the value of the expression is compared with each of the constant values in the case statements. If a match is found, the program executes the statement(s) following the matched case statement. If none of the constants matches the value of the expression, then the program executes the statement(s) following default statement. The value of the expression must be of type integer or character. Each case value must be a unique literal. The break statement is used to terminate the entire switch statement.

3. Looping structure:

These structures repeatedly execute a section of program a certain number of times. The repetition continues until a condition is true. When the condition becomes false, the loop ends and control passes to the statement following the loop. C++ provides three kinds of loops: the for loop, the while loop and the do… while loop.

a. The for loop:

The general form is:

```
for(initialization; test expression; increment)
{
        statement(s);
}
```

Before loop starts, the initialization statement is executed that initializes the loop control variable or variables in the loop. Second the test expression is evaluated, if it is true, the program executes the body of loop. Finally, the iteration statement will be executed and the test expression will be evaluated again, this continues until the test expression is false.

b. The while loop:

The general form is:

```
initialization
while(test expression)
{
        statement(s);
        increment/decrement of control variable;
}
```

Here the program executes the body of loop as long as the test expression is true. When the test expression becomes false, the while loop stops executing its body.

c. The do-while loop:

The general form is:

```
initialization
do
{
        statement(s);
        increment/decrement of control variable;
}while(test expression);
```

2.10 Functions

- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most programs can define additional functions.
- A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
- A function is known with various names like a method or a sub-routine or a procedure etc.

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

1. Default Arguments:

- When declaring a function we can specify a default value for each of the parameters which are called default arguments. This value will be used if the corresponding argument is left blank when calling to the function.
- If value for the parameter is not passed when a function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

For example:

```
//Default values in functions
#include<iostream>
#include<conio.h>
using namespace std;
int divide(int a, int b=2){
        return a/b;
}
int main(){
        cout<<divide(20)<<endl;
        cout<<divide(20,5)<<endl;
        getch();
        return 0;
}                                   //output: 10     4
```

2. Inline Functions:

Pardaina

- A function which is expanded inline by the compiler each time its call is appeared instead of jumping to the called function as usual is called inline function.
- It is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once.

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
inline void sum(int a, int b){
        cout<<"Sum= "<<a+b;
}
int main(){
        sum(10,20);
        getch();
        return 0;
}
```

3. Function Overloading:

- In C++ two different functions can have the same name if their parameter types or number are different. That means one can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. This is called function overloading.
- It allows a single function name to be used for different functions.

a. Overloading Functions with different type of arguments

```
#include<iostream>
#include<conio.h>
using namespace std;
int mul(int a, int b){
        return a*b;
}
float mul(float a, float b){
        return a*b;
}
int main(){
        int p=2,q=3;
        float x=1.0,y=5.5;
        cout<<"Product of integers= "<<mul(p,q)<<endl;
        cout<<"Product of reals= "<<mul(x,y);
        getch();
        return 0;
}
```

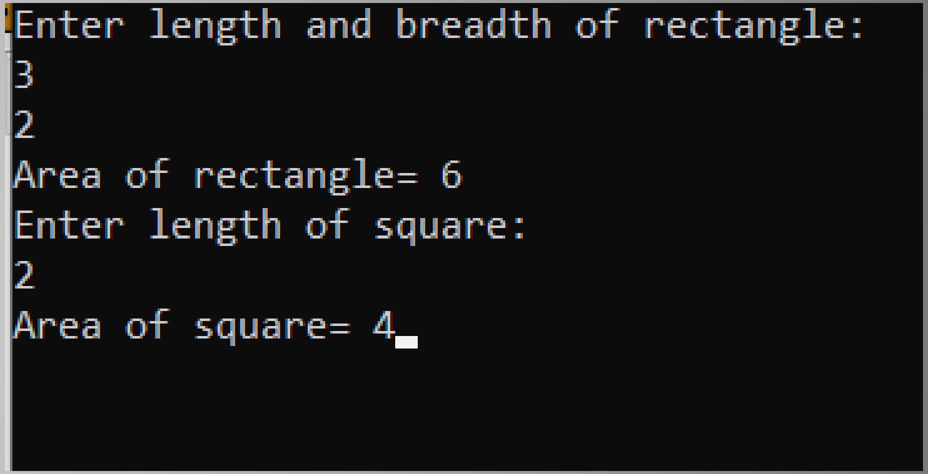
Output:

```
6
5.5
```

b. Overloading Functions with different number of arguments

```
#include<iostream>
#include<conio.h>
using namespace std;
int area(int l, int b){
        return l*b;
}
int area(int l){
        return l*l;
}
int main(){
        int l,b;
        cout<<"Enter length and breadth of rectangle: "<<endl;
        cin>>l>>b;
        cout<<"Area of rectangle= "<<area(l,b)<<endl;
        cout<<"Enter length of square: "<<endl;
        cin>>l;
        cout<<"Area of square= "<<area(l);
        getch();
        return 0;
}
```

OUTPUT:



```
Enter length and breadth of rectangle:
3
2
Area of rectangle= 6
Enter length of square:
2
Area of square= 4
```

PREPARED BY: ER. SHARAT MAHARJAN

IMP

4. Passing Arguments to the Function:

We can pass arguments to the function in three ways:

a. Pass by Value a = x

b. Pass by Reference &a = x

c. Pass by Pointer *a = &x

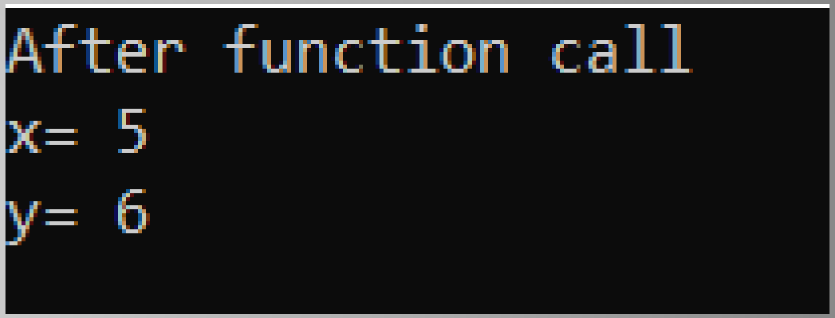
a. Pass by Value:

In case of pass by value, copies of the arguments are passed to the function not the variables themselves.

For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
void exchange(int a, int b){
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main(){
        int x=5,y=6;
        exchange(x,y);
        cout<<"After function call "<<endl;
        cout<<"x= "<<x<<endl<<"y= "<<y;
        getch();
        return 0;
}
```

OUTPUT:



```
After function call
x= 5
y= 6
```

PREPARED BY: ER. SHARAT MAHARJAN

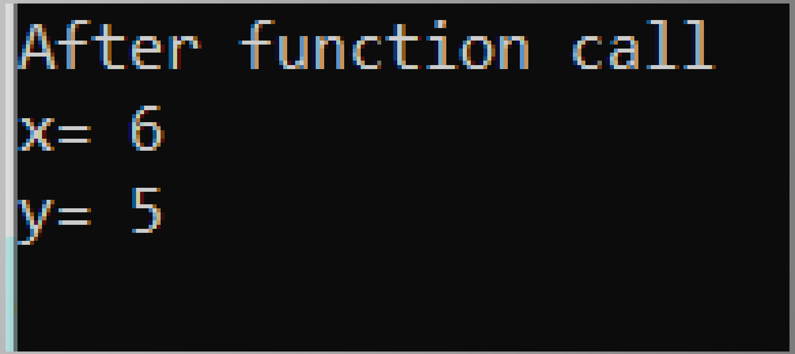
b. Pass by Reference:

In case of pass by reference, address of the variables (variable itself) are passed to the function but not the copies of the arguments.

For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
void exchange(int &a, int &b){
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main(){
        int x=5,y=6;
        exchange(x,y);
        cout<<"After function call "<<endl;
        cout<<"x= "<<x<<endl<<"y= "<<y;
        getch();
        return 0; }
```

OUTPUT:



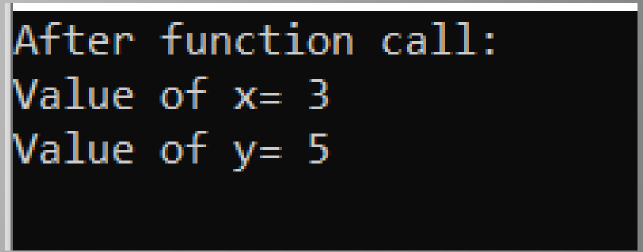
```
After function call
x= 6
y= 5
```

c. Pass by Pointer:

Working principle of pass by pointer is same as the pass by reference. Only difference is that instead of using the reference variable we use pointer variable in function definition and pass address of the variable from function call statement as below:

```
#include<iostream>
#include<conio.h>
using namespace std;
void exchange(int *a, int *b){
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
int main(){
        int x=5,y=3;
        exchange(&x,&y);
        cout<<"After function call: "<<endl;
        cout<<"Value of x= "<<x<<endl<<"Value of y= "<<y;
        getch();
        return 0; }
```

OUTPUT:



```
After function call:
Value of x= 3
Value of y= 5
```

5. Returning from Function:

We can return values from function in three ways:

a. Return by Value
b. Return by Reference
c. Return by Pointer

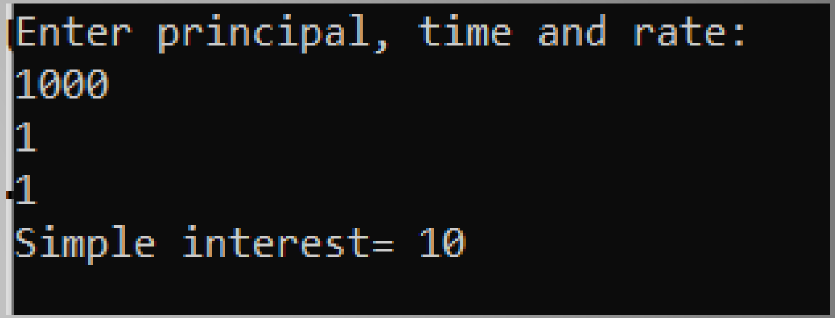
a. Return by Value: (returning normal value)

In case of return by value, copy of the value is returned to the user.

For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
float simpleInterest(float p, float t, float r){
        return (p*t*r)/100;
}
int main(){
        float p,t,r;
        cout<<"Enter principal, time and rate: "<<endl;
        cin>>p>>t>>r;
        cout<<"Simple interest= "<<simpleInterest(p,t,r);
        getch();
        return 0;
}
```

OUTPUT:



```
Enter principal, time and rate:
1000
1
1
Simple interest= 10
```

b. Return by Reference: (return reference variable by function)

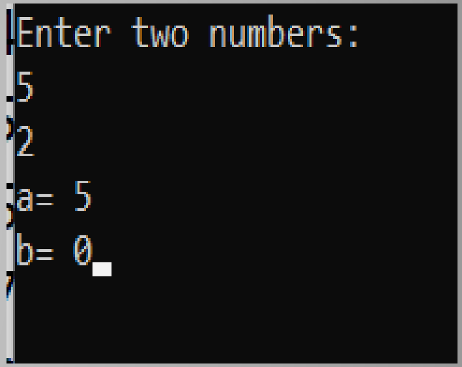
In case of return by reference, a reference to the variable (variable itself) is passed back to the user.

For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int &min(int &x, int &y){ //&min to return reference variable
        if(x<y)
        return x;
        else
        return y;
}
int main(){
        int a,b;
        cout<<"Enter two numbers: "<<endl;
        cin>>a>>b;
        min(a,b)=0;
        cout<<"a= "<<a<<endl<<"b= "<<b;
        getch();
        return 0;}
```

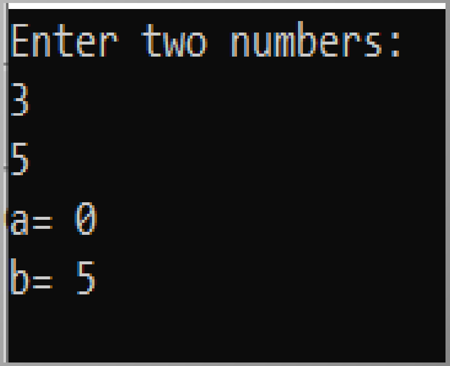
OUTPUT:

1st execution:



```
Enter two numbers:
5
2
a= 5
b= 0
```

2nd execution:



```
Enter two numbers:
3
5
a= 0
b= 5
```

PREPARED BY: ER. SHARAT MAHARJAN

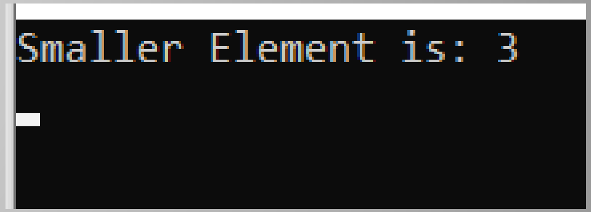
c. Return by Pointer: (return pointer variable by function)

Returning by pointer involves returning the address of a variable to the user. Just like pass by pointer, return by pointer can only return the address of a variable.

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int *min(int *a, int *b){ //*min to return pointer variable
        if(*a<*b)
        return a;
        else
        return b;
}
int main(){
        int x=5,y=3;
        int *m=min(&x,&y); //*m to store pointer variable
        cout<<"Smaller Element is: "<<*m<<endl; //*m to display content in memory
address m
        getch();
        return 0; }
```

OUTPUT:



```
Smaller Element is: 3
```

2.11 Scope/Visibility and Storage Class

1. Scope of Variables:

All the **variables have their area of functioning and out of that boundary they don't hold their value**, this boundary is called **scope of the variable**. There are three types of variable scope:

a. **Local Scope**

b. **Global Scope (File Scope)**

c. **Class Scope**

__a. Local Scope:__

A **variable declared within a block of code enclosed by braces ({}) is accessible only within that block** and only after the point of declaration. **Outside that they are unavailable and leads to compile time error.**

__b. Global Scope:__

Any variable declared outside all blocks or classes has global scope. It is **accessible anywhere in the file after its declaration**.

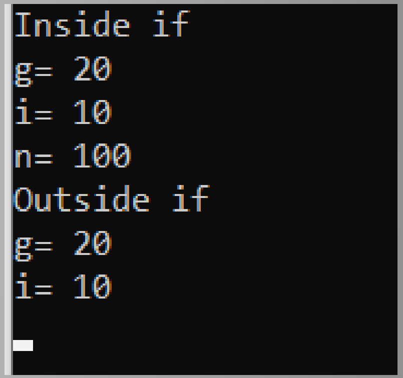
__c. Class Scope:__

Members of classes have class scope. **Variables with class scope are accessible in all of the methods of the class.**

For example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int g=20;               //global variable
int main(){
        int i=10; //local variable
        if(i<20){ //if condition scope starts
                int n=100;          //local variable
                cout<<"Inside if"<<endl;
                cout<<"g= "<<g<<endl;
                cout<<"i= "<<i<<endl;
                cout<<"n= "<<n<<endl;
        }       //if condition scope ends
        cout<<"Outside if"<<endl;
                cout<<"g= "<<g<<endl;
                cout<<"i= "<<i<<endl;
                //cout<<"n= "<<n<<endl;   ------ error can't be accessed here
        getch();
        return 0;}
```

OUTPUT:



```
Inside if
g= 20
i= 10
n= 100
Outside if
g= 20
i= 10
```

PREPARED BY: ER. SHARAT MAHARJAN

<u>2. Storage Classes:</u>

Pardaina

Storage class of a variable **defines the lifetime and scope(visibility) of a variable**. **Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible.**

There are five types of storage classes in C++. They are:

a. Automatic
b. External
c. Static
d. Register
e. Mutable

a. Automatic Storage Class:

The keyword auto is used to declare automatic variables. **If a variable is declared without any keyword inside a function, it is automatic by default.** This variable is **visible only within the function it is declared and its lifetime is same as the lifetime of the function as well**. Once the execution of function is finished, the variable is destroyed.

b. External Storage Class:

It **assigns variable a reference to a global**. The **keyword extern** is used to **declare external variables**. They are **visible throughout the program and its lifetime is same as the lifetime of the program** where it is declared. **This is visible to all the functions present in the program.**

c. Static Storage Class:

Static storage class ensures a variable has the **visibility mode of a local variable but lifetime of an external variable**. **When a function is called, the variable defined as static inside the function retains its previous value and operates on it.**

d. Register Storage Class:

It **assigns a variable's storage in the CPU** rather than primary memory. It has its **lifetime and visibility same as automatic variable**. The purpose of creating register variable is **to increase access speed and makes program run faster.**

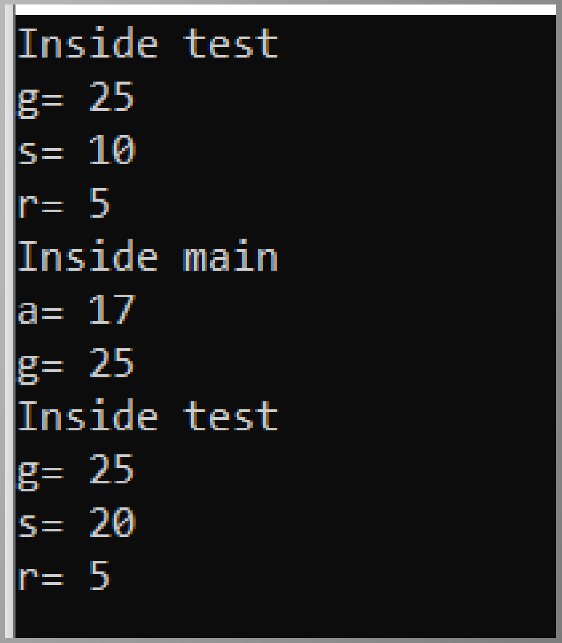
e. Mutable Storage Class:

It **applies only to objects**. It allows a member of an object to override const member function.

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int g;		//global variable, initially holds 0
void test(){
		static int s;		//static variable, initially holds 0
		register int r;		//register variable
		r=5;
		s=s+r*2;
		cout<<"Inside test"<<endl;
		cout<<"g= "<<g<<endl;
		cout<<"s= "<<s<<endl;
		cout<<"r= "<<r<<endl;
}
int main(){
		int a;		//automatic variable
		g=25;
		a=17;
		test();		//calling test() function from main
		cout<<"Inside main"<<endl;
		cout<<"a= "<<a<<endl;
		cout<<"g= "<<g<<endl;
		test();
		getch();
		return 0; }
```

OUTPUT:



```
Inside test
g= 25
s= 10
r= 5
Inside main
a= 17
g= 25
Inside test
g= 25
s= 20
r= 5
```

Program to demonstrate external variable

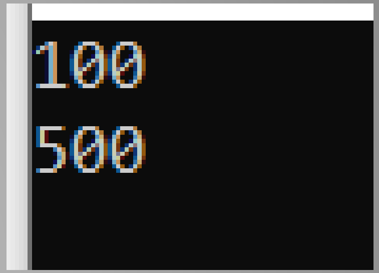
File: mul.cpp

```
int test=100; //assigning value to test
void multiply(int n){
        test=test*n;
}
```

File: main.cpp

```
#include<iostream>
#include "mul.cpp"     //includes the content of mul.cpp
extern int test;
using namespace std;
int main(){
        cout<<test<<endl;
        multiply(5);
        cout<<test<<endl;
        return 0;
}
```

OUTPUT:



```
100
500
```

PREPARED BY: ER. SHARAT MAHARJAN

2.12 Pointers

Suyash

A pointer is a **variable whose value is the address of another variable**. One **must declare a pointer before working with it**. The general form of a pointer variable declaration is:

type *var-name;

Following are the valid pointer declaration-

```
int *ip;            //pointer to an integer
double *dp;         //pointer to a double
float *fp;          //pointer to a float
char *ch;           //pointer to character
```

1. Operators in pointers:

- The **address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&)** known as **address-of operator**.

For example:

```
int a=10;
int *p=&a; //pointer declaration and initialization, can be in two  line also
```

- This would **assign the address of variable 'a' to pointer variable p.** We are **no longer assigning the content of the variable itself to p, but its address.**

An interesting property of pointers is that **they can be used to access the variable they point to directly.** This is done by **preceding the pointer name with the dereference operator (*)**.

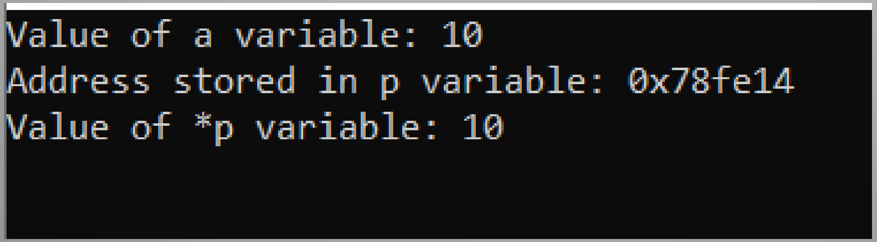
Consider the statement given below:

```
int x=*p;    //this would actually assign the value 10 to x
```

<u>Example:</u>

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(){
        int a=10;               //actual variable declaration
        int *p;                 //pointer variable
        p=&a;                   //store address of variable 'a' in pointer variable
        cout<<"Value of a variable: "<<a<<endl;
        cout<<"Address stored in p variable: "<<p<<endl;
        cout<<"Value of *p variable: "<<*p<<endl;
        getch();
        return 0;
}
```

<u>OUTPUT:</u>



```
Value of a variable: 10
Address stored in p variable: 0x78fe14
Value of *p variable: 10
```

2. Pointer and Arrays:

Arrays work very much like pointers to their first elements and an array can always be implicitly converted to the pointer of the proper type. **For example**, consider two declarations:

```
int x[20];
int *p;
p=x;   //valid
x=p;   //not valid since x represents the same block of 20
       //elements while value of p may be different
```

In arrays, brackets ([]) are used for specifying the index of an element of the array known as **dereferencing operator just as * does in pointer.**

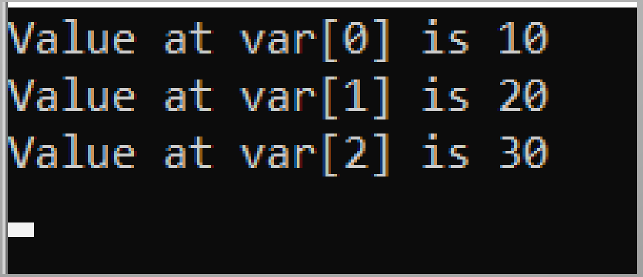
```
a[5]=0;       //a[offset of 5]=0, gives content
*(a+5)=0;     //pointed to by (a+5)=0, gives content
```

These two expressions are **equivalent and valid.**

Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(){
	int var[3]={10,20,30};
	int *p;
	p=var;
	for(int i=0;i<3;i++){
		cout<<"Value at var["<<i<<"] is "<<*p<<endl;
		p++;	//point to next location
	}
	getch();
	return 0;
}
```

OUTPUT:



```
Value at var[0] is 10
Value at var[1] is 20
Value at var[2] is 30
```

THANK YOU FOR YOUR ATTENTION

PREPARED BY: ER. SHARAT MAHARJAN