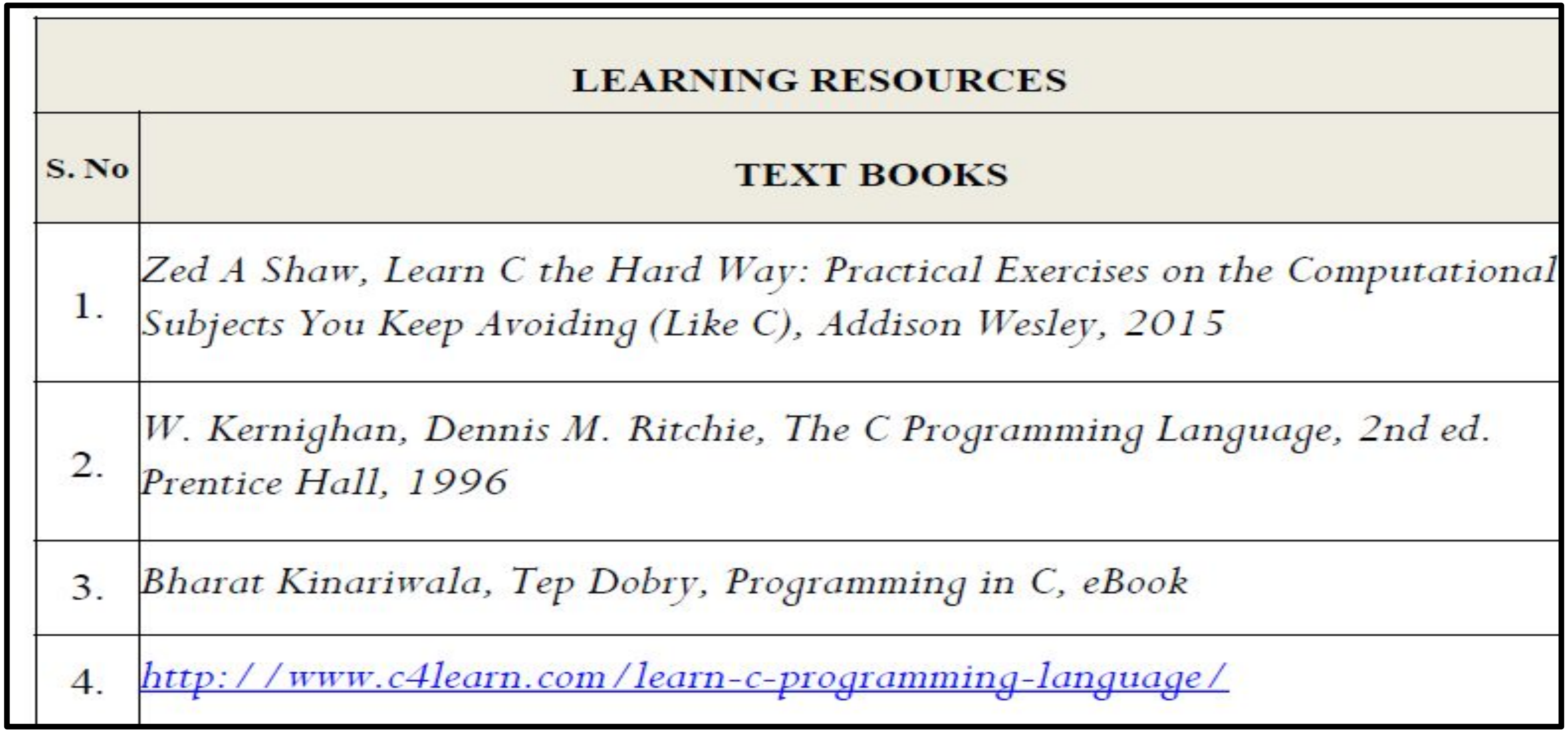
SRM Institute of Science and Technology, Chennai

21CSS101J –Programming for Problem Solving
Unit-I

SRM Institute of Science and Technology, Chennai

Prepared by:
Dr. P. Robert
Assistant Professor
Department of Computing Technologies
SRMIST-KTR

SRM Institute of Science and Technology, Chennai



LEARNING RESOURCES	
S. No	TEXT BOOKS
1.	*Zed A Shaw, Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C), Addison Wesley, 2015*
2.	*W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2nd ed. Prentice Hall, 1996*
3.	*Bharat Kinariwala, Tep Dobry, Programming in C, eBook*
4.	*http://www.c4learn.com/learn-c-programming-language/*

Unit-I

Evolution of Programming & Languages - Problem solving through programming - Writing algorithms & Pseudo code - Single line and multiline comments - Introduction to C: Structure of the C program - Input and output statements. Variables and identifiers, Constants, Keywords - Values, Names, Scope, Binding, Storage Classes - Numeric Data types: integer, floating point Non-Numeric Data types: char and string - L value and R value in expression, Increment and decrement operator - Comma, Arrow and Assignment operator, Bitwise and Size-of operator - Arithmetic, Relational and logical Operators - Condition Operators, Operator Precedence - Expressions with pre / post increment operator

Evolution of Programming Languages

Programming Language is considered as the **set of commands and instructions** that we give to the machines to perform a particular task. For example, if you give some set of instructions to add two numbers then the machine will do it for you and tell you the correct answer accordingly.

But a good programming language----

1. Portability
2. Maintainability
3. Efficient
4. Reliable
5. Machine Independence
6. Cost Effectiveness
7. Flexible

Evolution of Programming Languages

In the global, we have about 500+ programming languages with having their own syntax and features.

1883	In the early days, Charles Babbage had made the **device**, but he was confused about **how to give instructions** to the machine, and then Ada Lovelace wrote the instructions for the analytical engine.
1949 Assembly Language	• It is a type of low-level language • AL can be easily understandable by machine • It is also used to create computer viruses • Real time programs such as simulation of flight navigation system and medical equipment.

Evolution of Programming Languages

1952 Autocode	• Developed by Alick Glennie • It is the first compiled computer programming language • Ex: FORTRAN, COBOL
1957 FORTRAN	• Developed by IBM • Numeric computation and scientific computing • Software for NASA was written in FORTRAN
1958 ALGOL	• ALGOrithmic language • It is the fundamental for C, C++ and Java • The first PL to have a code block like "**Begin**" that indicates that your program has started and "**End**" means you have ended your code
1959 COBOL	• It stands for Common Business Oriented Language • In 1997, 80% of the world business ran on COBOL

Evolution of Programming Languages

1972 C	▪ It is a general-purpose, **procedural programming** language and the most popular programming language till now. ▪ All the code that was previously written in assembly language gets replaced by the C language like operating system, kernel, and many other applications. ▪ It can be used in implementing an **operating system**, **embedded system**, and also on the website using the Common Gateway Interface (CGI). ▪ C is the mother of almost all higher-level programming languages like C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, and Unix's C shell.

Evolution of Programming Languages

1972 SQL	• SQL was developed at IBM by Donald D. Chamberlin and Raymond F. Boyce. The earlier name was SEQUEL (Structured English Query Language).
1978 MATLAB	• It stands for **MATrix LABoratory**. It is used for matrix anipulation, implementation of an algorithm, and creation of a user interface.
1983 C, C++	• C++ is the fastest high-level programming language. Earlier, Apple Inc uses Objective-C to make applications.
1991 Python	• The language is very easy to understand. Famous language among data scientists and analysts.

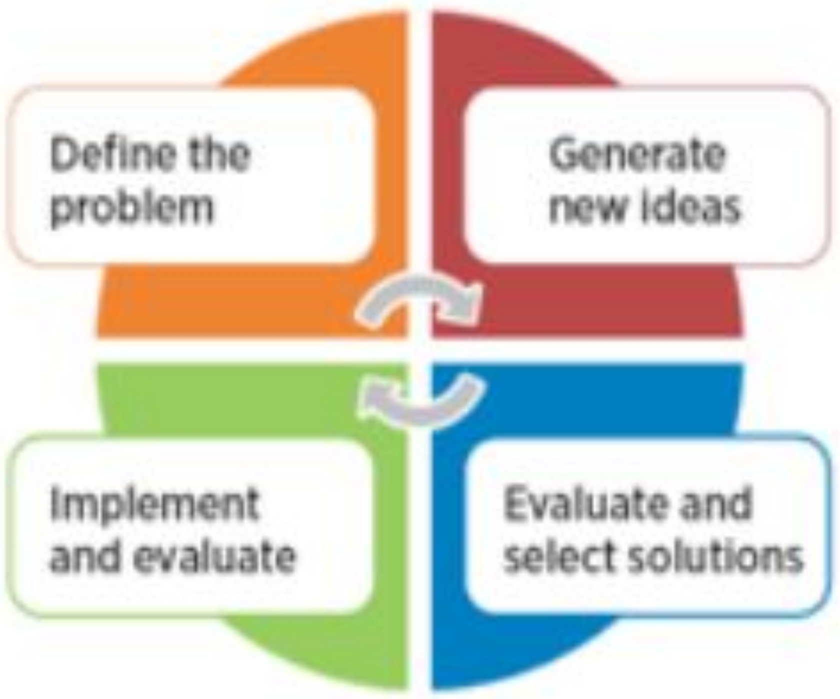
Evolution of Programming Languages

1995 Java, PHP, Java Scipt	• JAVA is everywhere. JAVA is the platform-independent language. • PHP is a scripting language mainly used in web programming for connecting databases. • JavaScript enables interactive web pages. JS is the most popular programming language. JS is famous for building a web application. It makes our page interactive.
2000 C#	• C#(C-sharp) is mainly used for making games. Unity engine uses C# for making amazing games for all platforms

Problem Solving through programming

What is problem solving?

Problem solving is the act of defining a problem; determining the cause of the problem; identifying, prioritizing, and selecting alternatives for a solution; and implementing a solution.



Problem Solving through programming

Stages of Problem solving

1. Understand the problem
2. Define the problem
3. Define boundaries
4. Plan solution
5. Check solution

Divide and Conquer Approach

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

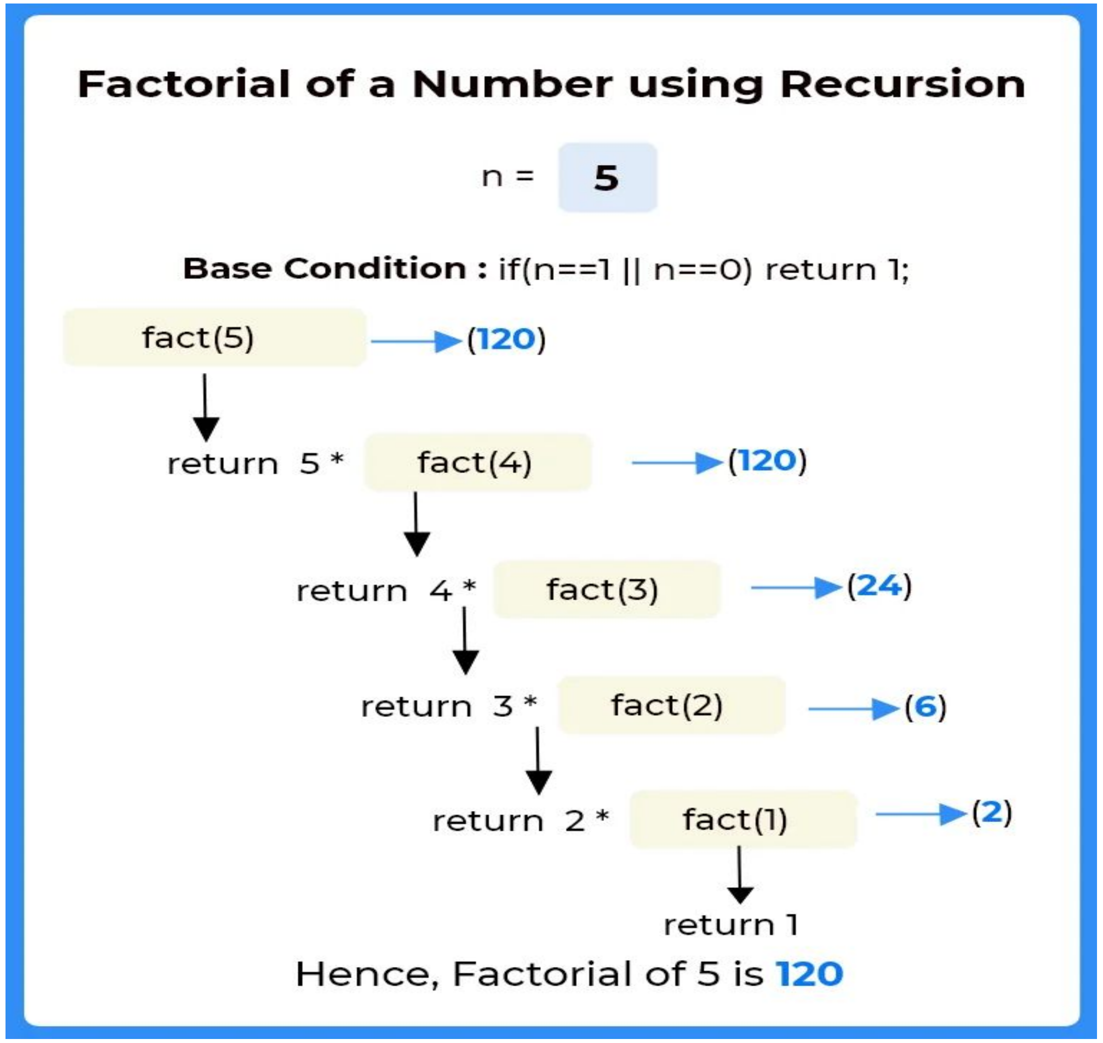
Problem Solving through programming

```
#include <stdio.h>
int factorial(int);
int main() {
    int n, result;
   printf("Enter a non-negative number: ");
   scanf("%d",&n);
    result = factorial(n);
    printf("The factorial of of %d is %d",n,result);
    return 0;
}
int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

To use divide and conquer algorithm recursion is used.

Recursive call

Problem Solving through programming



Writing Algorithms and Pseudo Code

An algorithm is a **set of steps** designed to solve a problem or accomplish a task. Algorithms are usually written in pseudocode, or a combination of your speaking language and one or more programming languages, in advance of writing a program.

Step-1: Obtain **detailed information** on the problem.

Step-2: Analyze the Problem

Step-3: Think of a **problem solving** approach

Step-4: Review the problem solving approach and try to think of a better alternative

Step-5: Develop a **basic structure** of the algorithm

Step-6: Optimize, improve and refine.

Writing Algorithms and Pseudo Code

Characteristics of a good algorithm

1: Input and output must be specified

2: All important steps must be mentioned

3: Instructions must be perfectly ordered

4: Short and effective descriptions

5: The algorithm must contain finite number of steps

Writing Algorithms and Pseudo Code

Examples of Algorithm

1. **Addition of two numbers**

```
Step-1: Start
Step-2: Declare variables num1, num2, and sum
Step-3: Read values of num1 and num2
Step-4: Add the values of num1 and num2 and assign the result
Step-5: Display the sum
Step-6: End
```

Writing Algorithms and Pseudo Code

Examples of Algorithm

2. Comparison of 3 numbers to find the largest number

```
1: Start
2: Declare variables num1, num2, and num3
3: Read values of num1 and num2 and num3
4: Compare num1, num2 and num3
5: If num1>num2 and num1>num3
        Display num1 is the largest number
Else
        If num2>num1 and num2>num3
        Display num2 is the largest number
Else
        Display num3 is the largest number
6: End
```

Writing Pseudo Code

1. Always capitalize the initial word (often one of the main six constructs).
2. Make only one statement per line.
3. Indent to show hierarchy, improve readability, and show nested constructs.
4. Always end multi-line sections using any of the END keywords (ENDIF, ENDWHILE, etc.).
5. Keep your statements programming language independent.
6. Use the naming domain of the problem, not that of the implementation. For instance: “Append the last name to the first name” instead of “name = first+ last.”
7. Keep it simple, concise and readable.

Writing Pseudo Code

Main Constructs of Pseudocode

Sequence: Sequentially completed linear tasks are represented by it.

WHILE: It is a loop that starts with a condition.

REPEAT-UNTIL: A loop containing a condition present at the bottom called REPEAT-UNTIL.

FOR: an additional looping method.

IF-THEN-ELSE: A conditional statement that alters the algorithm’s flow is known as IF-THEN-ELSE.

CASE: It is the IF-THEN-ELSE generalization form.

Single line and Multi line comments

In general, there are two types of comments in programming languages.

1. **Single line comments**

Single line comments is accomplished by double-slash (//). Everthing that is followed by double-slash till the end of line is ignored by the compiler.

2. **Multi line comments**

Multi-line comments starts by using forward slash followed by asterisk (/*) and ends by using asterisk followed by forward slash (*/). Everthing between (/*) and (*/) are ignored by compiler whether it is one or more than one line.

Single line and Multi line comments

Example:

```
/* This program takes age input from the user It stores it in the age variable And, print the value using printf() */
#include <stdio.h>
int main() {
 //declare integer variable
 int age;
 printf("Enter the age: ");
 scanf("%d", &age);
 printf("Age = %d", age);
 return 0;
}
```

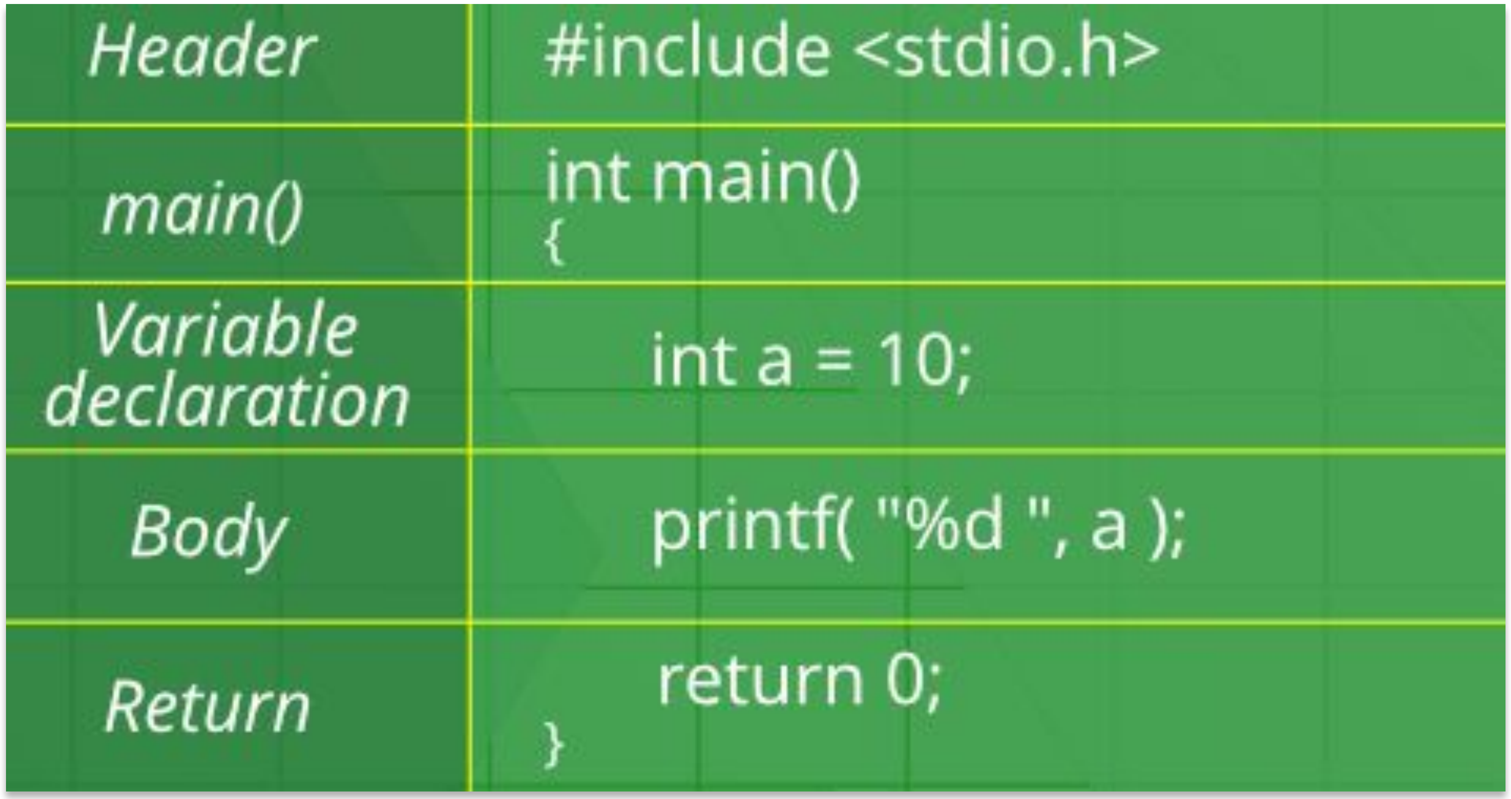
Introduction to C

Basic Structure of C Program

- ❖ C is an imperative programming language. Dennis Ritchie invented it in the year 1972. It was created primarily as a system programming language for creating an operating system.
- ❖ The main features of the C language include low-level memory access, a **simple set of keywords**, and a clean style, these features make C language suitable for system programming like an operating system or compiler development.
- ❖ Many advanced programming languages have borrowed syntax/features directly or indirectly from the C language. The languages like Java, PHP, Java script and many other programming languages are mainly based on the C language.

Basic Structure of C Program

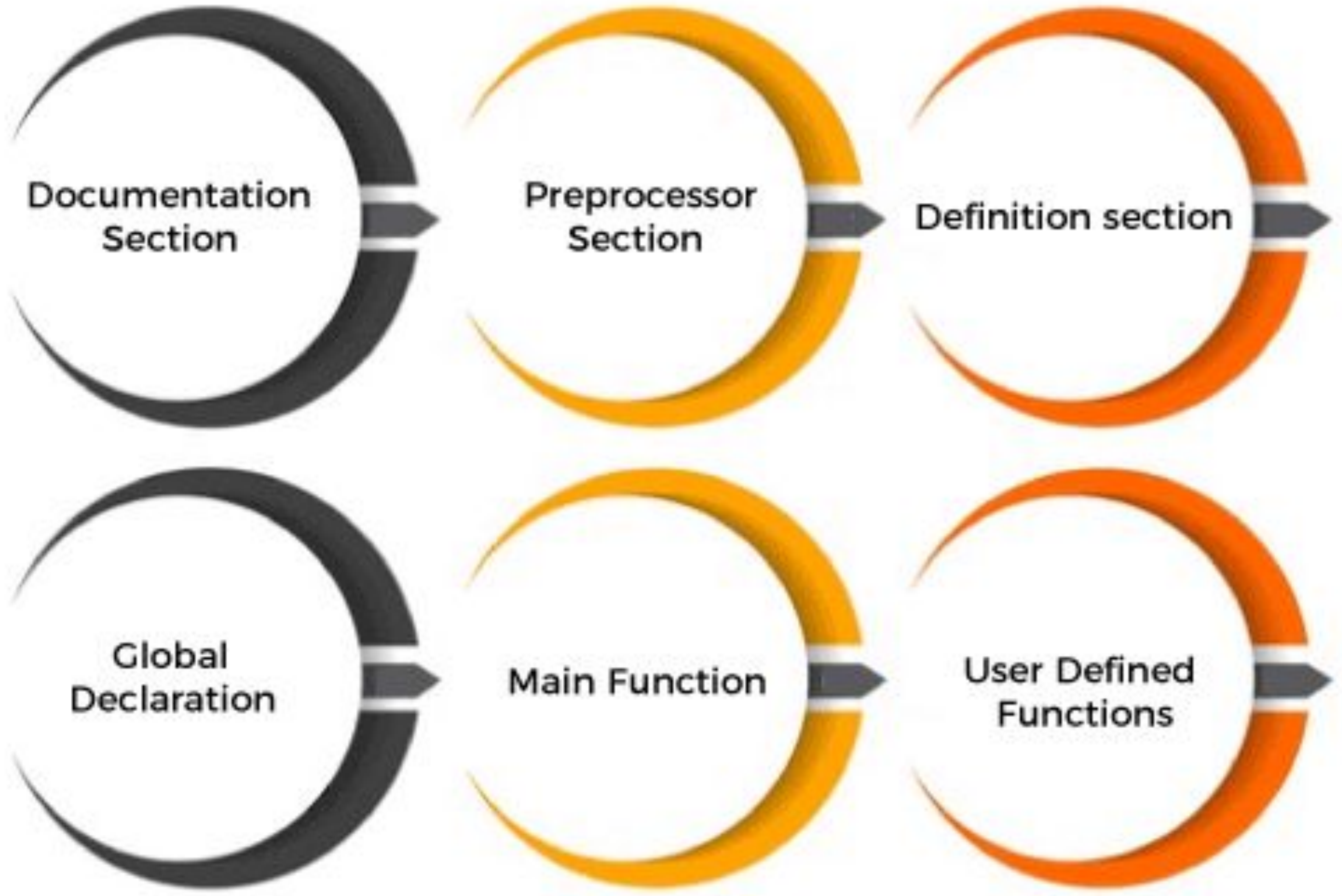
The C programming language contains a set of protocols that describe its operations.



Header	#include <stdio.h>
main()	int main() {
Variable declaration	int a = 10;
Body	printf("%d ", a);

Return	return 0;
	}

Basic Structure of C Program



Basic Structure of C Program

The sections of a C program are listed below:

- Documentation section
- Preprocessor section
- Definition section
- Global declaration
- Main function
- User defined functions

Basic Structure of C Program

Documentation Section:

It includes statements such as a program's name, date, description, and title that are specified at the start of the program.

Example:

//name of the program

(or)

/*

Specify the title and date

***/**

It provides overview of the program. *The Documentation section consists of a set of comment lines.*

Basic Structure of C Program

Preprocessor Section:

All of the header files used in a program are found in the preprocessor section. It informs the system that the header files should be linked to the system libraries. It is provided by:

#include<stdio.h> // Defines core input and output functions

#include<string.h> // Defines string handling functions

#include<math.h> // Defines common mathematical functions

This section provides instruction to the compiler to link the header files or functions from the system library.

Basic Structure of C Program

Definition Section:

The definition section defines all symbolic constants such by using the **#define** directive.

```
#define a=5
```

Global Declaration Section:

There are some variables that are used in more than one function, such variables are called global variables.

In C there are two types of variable declaration,

Local variable declaration: Variables that are declared **inside the main function**.

Global variable declaration: Variables that are declared **outside the main function**.

```
float num = 2.54;
int a = 5;
char ch ='z';
```

Basic Structure of C Program

Main Function Section:

Every C-program should have **one main()** function. **main()** is the first function to be executed by the computer. It is necessary for a code to include the main().

The main function is declared as:

main()

We can also use int or main with the main (). The void main() specifies that the program will not return any value. The int main() specifies that the program can return integer type data.

int main()

(or)

void main()

Basic Structure of C Program

Main Function Section:

Main function is further categorized into **local declarations, statements,** and **expressions.**

Local Declarations

The variable that is declared inside a given function or block refers to as local declarations.

```
main()
{
int i = 2; //local variable
i++;
}
```

Basic Structure of C Program

Main Function Section:

Statements:

The statements refers to **if, else, while, do, for**, etc. used in a program within the main function.

Expressions:

An expression is a type of formula where operands are linked with each other by the use of operators. It is given by:

```
a - b;
a +b;
```

Basic Structure of C Program

User Defined Function:

It includes number of functions implemented in the program. For example, color(), sum(), division(), etc.

Return statement is generally the last section of a code. But, it is not necessary to include. It is used when we want to return a value. The return function returns a value when the return type other than the void is specified with the function.

return;

(or)

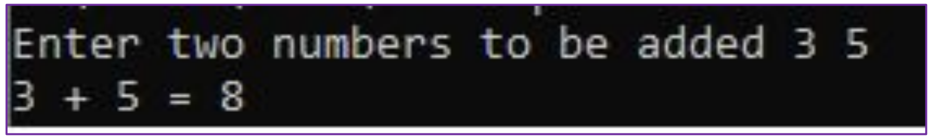
return expression;

For example

return 0;

Basic Structure of C Program

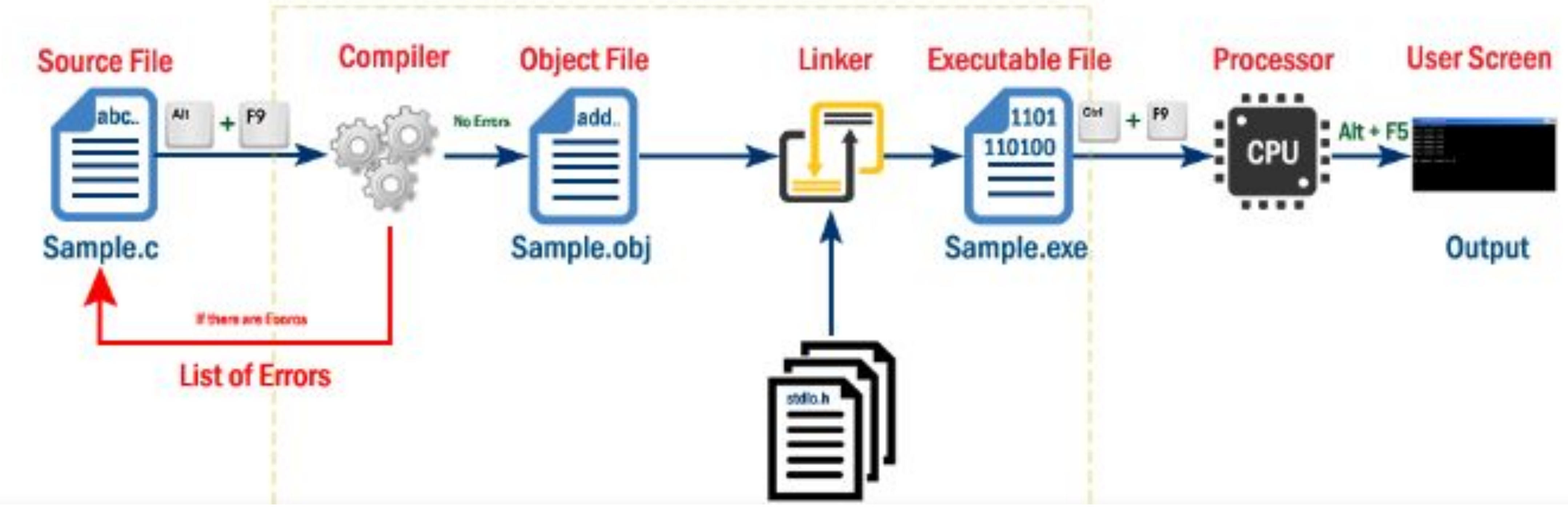
Example:



```
Enter two numbers to be added 3 5
3 + 5 = 8
```

```
/* Sum of two numbers */
#include<stdio.h>
int main()
{
int a, b, sum;
printf("Enter two numbers to be added ");
    scanf("%d %d", &a, &b);
    // calculating sum
    sum = a + b;
    printf("%d + %d = %d", a, b, sum);
    return 0;  // return the integer value in the sum
}
```

Executing a C Program



Executing a C Program

Creating Source Code:

1. Click on the **Start** button
2. Select **Run**
3. Type **cmd** and press Enter
4. Type **cd c:\TC\bin** in the command prompt and press **Enter**
5. Type **TC** press **Enter**
6. Click on **File -> New** in C Editor window
7. Type the **program**
8. Save it as **FileName.c** (Use shortcut key **F2** to save)

Executing a C Program

Compile Source Code:

The compilation is the process of converting high-level language instructions into low-level language instructions. We use the shortcut key **Alt + F9** to compile a C program in **Turbo C**.

Whenever we press **Alt + F9**, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into **object code** and stores it as a file with **.obj** extension.

Then the object code is given to the **Linker**. The Linker combines both the **object code** and specified **header file** code and generates an **Executable file** with a **.exe** extension.

Executing a C Program

Executing/Running Executable File(Ctrl+F9):

After completing **compilation** successfully, an executable file is created with a **.exe** extension. The processor can understand this **.exe** file content so that it can perform the task specified in the source file. We use a shortcut key **Ctrl + F9** to run a C program. Whenever we press **Ctrl + F9**, the **.exe** file is submitted to the **CPU**. On receiving **.exe** file, **CPU** performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called **User Screen**.

Check Result(Alt+F5):

After running the program, the result is placed into **User Screen**. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key **Alt + F5** to open the User Screen and check the result.

VARIABLES

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Syntax:

datatype variable_name;

int a;
float b;
char c;
Here, a, b, c are variables. The int, float, char are the data types. We can also provide values while declaring the variables as given below:

int a=10,b=20;//declaring 2 variable of integer type

float f=20.8;

char c='A';

VARIABLES

Rules for defining variables:

- A variable can have alphabets, digits, and underscore.
- A variable name can **start with the alphabet**, and underscore only. It can't start with a digit.
- **No whitespace** is allowed within the variable name.
- A variable name must not be any **reserved word** or **keyword**, e.g. int, float, etc.

Valid variable names:

```
int a;
int _ab;

int a30;
```

Invalid variable names:

```
int 2;

int a b;

int long;
```

VARIABLES

Types of variables in C:

There are many types of variables in c:

1) **local variable**
2) **global variable**
3) **static variable**
4) **automatic variable**
5) **external variable**

VARIABLES

local variable:

A variable that is declared inside the function or block is called a local variable.

It must be declared at the start of the block.

```
void add(){
int x=12;//local variable
}
```

VARIABLES

global variable:

A variable that is declared **outside the function** or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

```
int value=20;//global variable
void mul(){
int x=10;//local variable
}
```

VARIABLES

static variable:

A variable that is declared with the static keyword is called static variable.

It retains its value between multiple function calls.

```
void function1(){
int x=10;//local variable
static int y=10;//static
variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

VARIABLES

automatic variable:

All variables in C that are declared inside the block, are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

```
void main(){
int x=10;//local variable (also automatic)
auto int y=20;//automatic variable
}
```

IDENTIFIER

"Identifiers" or "symbols" are the names you supply for variables, types, functions, and labels in your program. Identifier names must differ in spelling and case from any keywords. You can't use keywords (either C or Microsoft) as identifiers; they're reserved for special use. Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```
int money;
double accountBalance;
```

Here, money and accountBalance are identifiers

Also remember, identifier names must be different from keywords. You cannot use int as an identifier because int is a keyword.

IDENTIFIER

Rules for Naming Identifiers

- A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
- The first letter of an identifier should be either a letter or an underscore.
- You cannot use keywords like int, while etc. as identifiers.
- There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

Constant

A constant is a value or variable that can't be changed in the program. C Constants is the most fundamental and essential part of the C programming language. Constants in C are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

- **Constants are also called literals.**
- **Constants can be any of the data types.**
- **It is considered best practice to define constants using only upper-case names.**

Two ways to define constant in C

1. const keyword
2. #define preprocessor

Constant

C const Keyword

The const keyword is used to define constant in C programming.

Syntax:

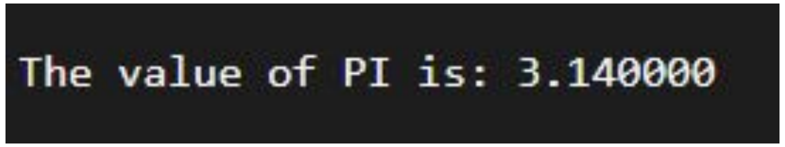
const data_type constant_name;

Example:

const float PI=3.14;

```c
#include<stdio.h>
int main(){
const float PI=3.14;
printf("The value of PI is: %f",PI);
return 0;
}
```

Output:



```
The value of PI is: 3.140000
```

Constant

C const Keyword

The const keyword is used to define constant in C programming.

Syntax:

const data_type constant_name;

Example:

const float PI=3.14;

```
#include<stdio.h>
int main(){
const float PI=3.14;
PI=4.5;
printf("The value of PI is: %f",PI);
return 0;
}
```

Note: If we try to change the value of PI, it will throw compile time error.
Compile Time Error: Cannot modify a const object

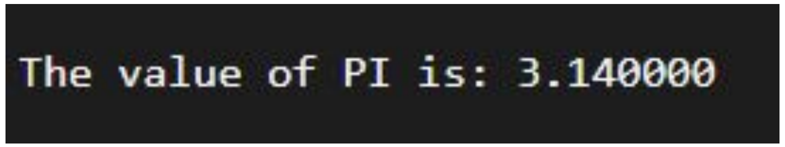
Constant

C #define preprocessor

The #define preprocessor is also used to define constant.

```
#include <stdio.h>
#define PI 3.14
void main() {
   printf("%f",PI);
}
```

Output:



```
The value of PI is: 3.140000
```

Constant

Constant Types in C

- ❑ Numeric Constant
  - ✔ Integer Constant
  - ✔ Real Constant
- ❑ Character Constant
  - ✔ Single Character Constant
  - ✔ String Constant
  - ✔ Backslash Character Constant

Constant

Integer Constant

It's referring to a sequence of digits. Integers are of three types:

- ✔ Decimal Integer
- ✔ Octal Integer
- ✔ Hexadecimal Integer

Example:

15, -265, 0, 99818, +25, 045, 0X6

Real Constant

The numbers containing fractional parts like 99.25 are called real or floating points constant.

Constant

Single Character Constant

It simply contains a single character enclosed within ' and ' (a pair of single quote). It is to be noted that the character '**8**' is not the same as **8**.

Example:

'X', '5', ';’

String Constant

These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blank spaces. It is again to be noted that "**G**" and '**G**' are different - because "G" represents a string as it is enclosed within a pair of double quotes whereas 'G' represents a single character.

Example:

"Hello!", "2015", "2+1"

Constant

Backslash Character Constant

C supports some character constants having a backslash in front of it. The lists of backslash characters have a specific meaning which is known to the compiler. They are also termed as "Escape Sequence".

For Example:

\t is used to give a tab

\n is used to give a new line

KEYWORDS

A keyword is a reserved word. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

auto	**break**	**case**	**char**	**const**	**continue**	**default**	**Do**
double	**else**	**enum**	**extern**	**float**	**for**	**goto**	**If**
int	**long**	**register**	**return**	**short**	**signed**	**sizeof**	**Static**
struct	**switch**	**typedef**	**union**	**unsigned**	**void**	**Volatile**	**while**

Input and Output Statements

Input and Output statement are used to read and write the data in C programming. These are embedded in stdio.h (standard Input/Output header file).

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

Input and Output Statements

scanf() and printf() functions

1. The printf() function

The printf() function is the most used function in the C language. This function is defined in <stdio.h> header file.

1. Print a sentence

printf(“Welcome to SRMIST”);

2. Print an integer value

We can use the printf() function to print integer value using the %d format specifier.

Example:

```
int x=10;
printf(“X=%d”,x);
```

Input and Output Statements

Format specifier
Float-%f
Integer-%d
Double-%lf
Character-%c

2. The scanf() function

The scanf() is used to store the input value into a variable.

Syntax:

Scanf("%f",&var);

Input integer value

scanf("%d",&var1);

Input float value

scanf("%f",&var1);

Input and Output Statements

getchar() & putchar() functions

The **getchar()** function reads a character from the terminal and returns it as an integer.

Syntax:

```
int getchar(void)
```

The **putchar()** function is used to display only one character at a time.

```
int putchar(int character)
```

gets() & puts() functions

The gets() function reads a line from standard input into the buffer pointed to by str pointer.

Syntax:

```
char* gets(char* str)
```

The puts() function writes the string str with a newline character at the end to stdout. On success, non-negative value is returned.

Syntax:

```
int puts(const char* str)
```

Input and Output Statements

The main difference between scanf() and gets() functions is

gets() function reads space as character.

scanf() stop reading characters when it encounters a space.

If we eneter SRM University using scanf() it will only read and store SRM and will leave the part of the string after space. But gets() function will read it completely.

Input and Output Statements

The main difference between scanf() and gets() functions is

gets() function reads space as character.

scanf() stop reading characters when it encounters a space.

If we eneter SRM University using scanf() it will only read and store SRM and will leave the part of the string after space. But gets() function will read it completely.

Scope

A **block** or a **region** where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed.

```
#include <stdio.h>
 int main()
{
      int var = 34;     // Scope of this variable is within main() function only.
                        // Therefore, called LOCAL to main() function.
      printf("%d", var);
      return 0;
}
```

Scope

Local Variables:

Variables that are declared **within the function block** and can be used only within the function are called local variables.

Local Scope or Block Scope

A local scope or block is a collective program statement placed and declared within a function or block (a specific area surrounded by curly braces). C also has a provision for nested blocks, which means that a block or function can occur within another block or function. So it means that variables declared within a block can be accessed within that specific block and all other internal blocks of that block but cannot be accessed outside the block.

Scope

Example (Local Variable):

```
#include <stdio.h>
 int main ()
{
    //local variable definition and initialization
    int x,y,z;
     //actual initialization
    x = 20;
    y = 30;
    z = x + y;
   printf ("value of x = %d, y = %d and z = %d\n", x, y, z);
   return 0;
}
```

Scope

Example (Global Variable):

```
#include <stdio.h>
int z; //global variable
 int main ()
{
    //local variable definition and initialization
    int x,y;
     //actual initialization
    x = 20;
    y = 30;
    z = x + y;
   printf ("value of x = %d, y = %d and z = %d\n", x, y, z);
   return 0;
}
```

Binding

In C, binding refers to the association of a name with a particular entity, such as a variable or a function.

Binding is typically done through the use of declarations or definitions in the code.

Example:

```
#include <stdio.h>
  int main() {
   int x = 5;  // Binding the name 'x' to the value 5
   printf("The value of x is: %d\n", x);
   {
      int x = 10;  // Binding a new 'x' in a nested block
      printf("The value of nested x is: %d\n", x);
   }
   printf("The value of x is still: %d\n", x);  // Accessing the outer 'x'
   return 0;
}
```

Storage classes

Each variable in C programming language has two properties.

1. type
2. storage class

Type refers to the data type of a variable. And, a storage class determines or specify the scope or lifetime of the variable.

There are four types of storage classes

1. automatic
2. external
3. static
4. register

Storage classes

Storage Class	Purpose
auto	It is a default storage class
extern	It is a global variable
static	It is a local variable which is capable of returning value even when control is transferred to the function call
register	It is a variable which is stored inside a register.

Storage classes

Automatic variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined. The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Storage classes

Automatic variable

Example:

auto int age;

```
  int add(void)
{
  int a=13;
  auto int b=48;
  return a+b;
}
```

Storage classes

Automatic variable

```
#include <stdio.h>
int main( )
{
  auto int j = 1;
  {
    auto int j= 2;
    {
      auto int j = 3;
      printf ( " %d ", j);
    }
    printf ( "\t %d ",j);
  }
  printf( "%d\n", j);
}
```

Storage classes

Extern

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

- The default initial value of global variable is 0 otherwise null.
- The external variable can be initialized outside the function only.
- An external variable can be declared many times but can be initialized at only once.

Storage classes

Global variable

```
#include <stdio.h>
void display();
int n = 5;  // global variable
int main()
{
    ++n;
    display();
    return 0;
}
void display()
{
    ++n;
    printf("n = %d", n);
}
```

Output:
n=7

Storage classes

static

A static variable is declared by using the keyword static. For example

static int i;

The value of the static variable persists until the end of the program.

- A static variable can be declared many times but can be initialized at only once.
- The default value of the static variable is 0 otherwise null.

Storage classes

static variable

```
#include <stdio.h>
void display();
int main()
{
    display();
    display();
}
void display()
{
    static int c = 1;
    c += 5;
    printf("%d  ",c);
}
```

Output:

```
6  11
```

Storage classes

register

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

- The variables defined as register is allocated the memory into the CPU registers.
- We can not dereference the register variable.
- The access time of register variable is faster than the local variables.
- The initial default value of the register variable is 0.
- Static variables cannot be stored to the register.

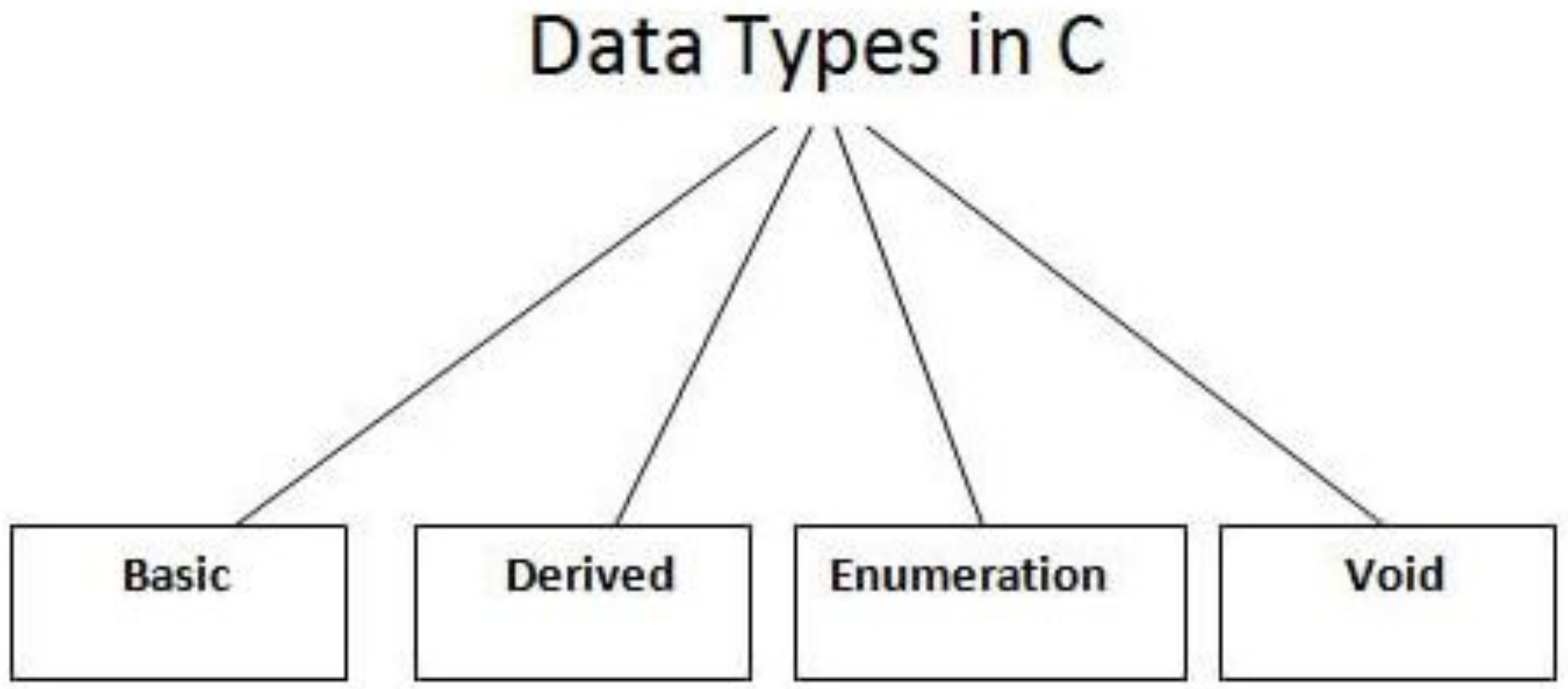
Storage classes

register variable

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default
 value of a is 0.
printf("%d",a);
}
```

DATA TYPES

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.

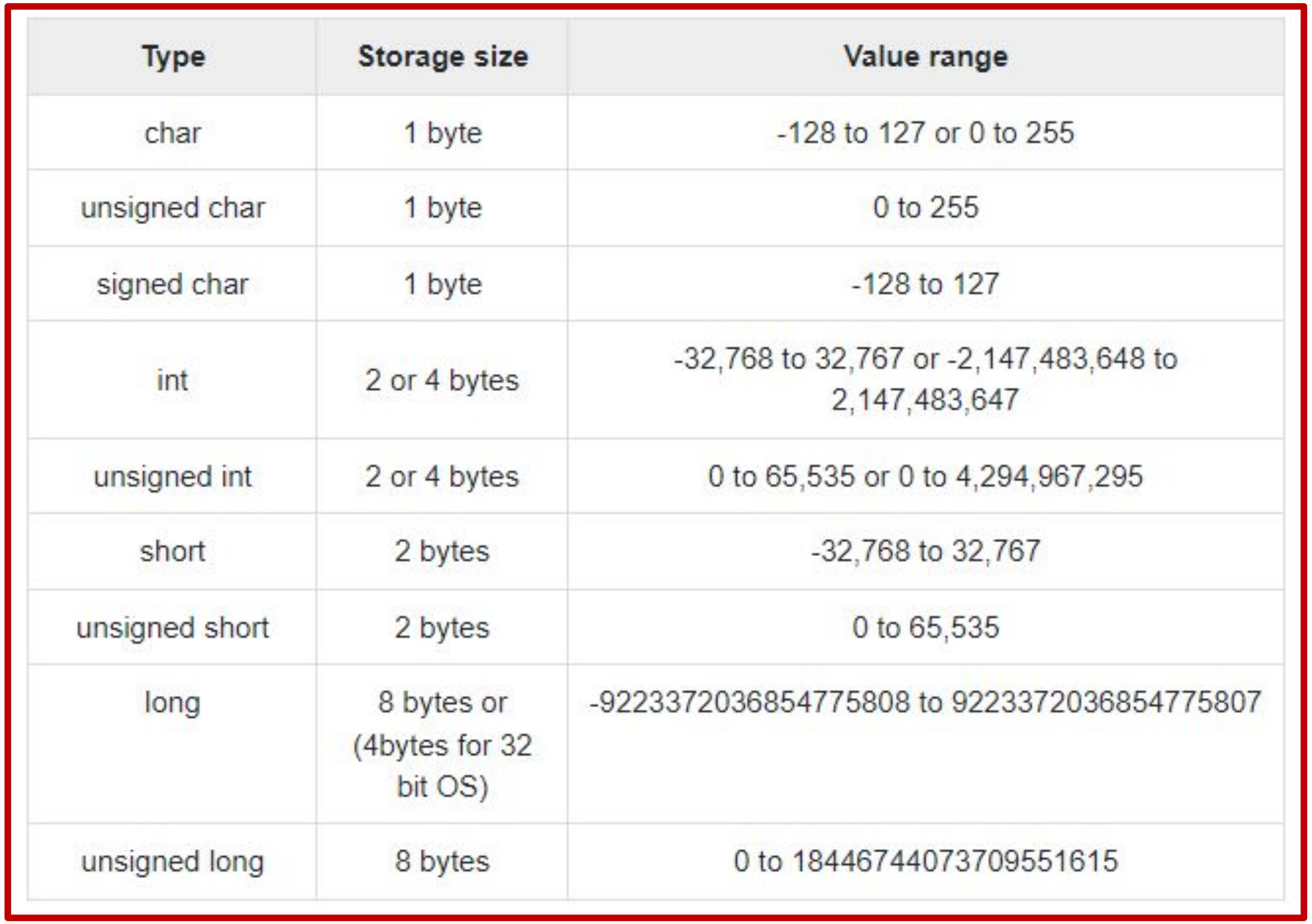


DATA TYPES

Types	Description
Basic Types	They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
Enumerated Types	They are again arithmetic types, and they are used to define variables that can only assign certain discrete integer values throughout the program.
void	The type specifier void indicates that no value is available.
Derived Types	They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

DATA TYPES

Integer Types



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
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	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

DATA TYPES

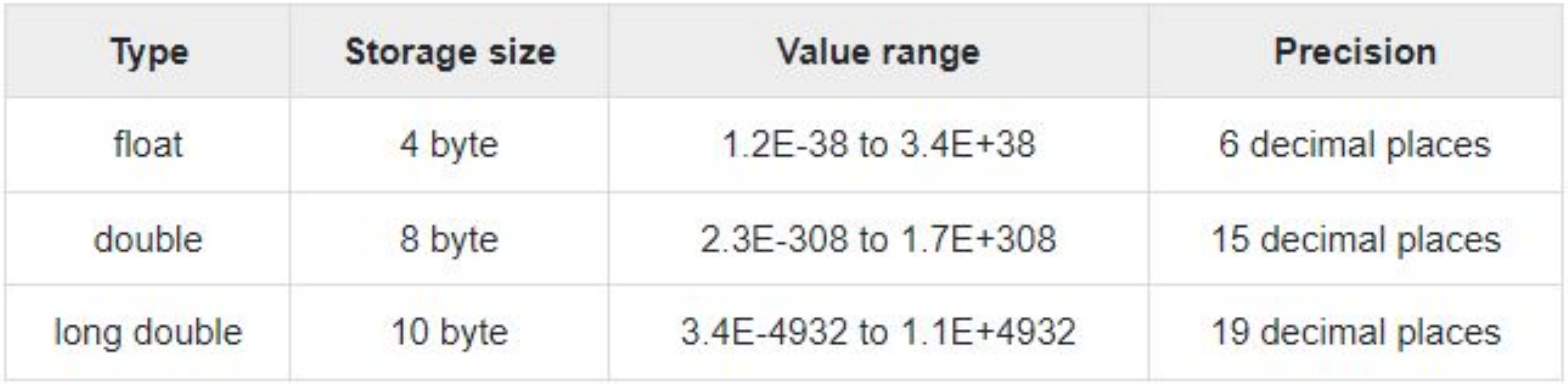
Integer Types

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator.

The expressions *sizeof(type)* yields the storage size of the object or type in bytes.

DATA TYPES

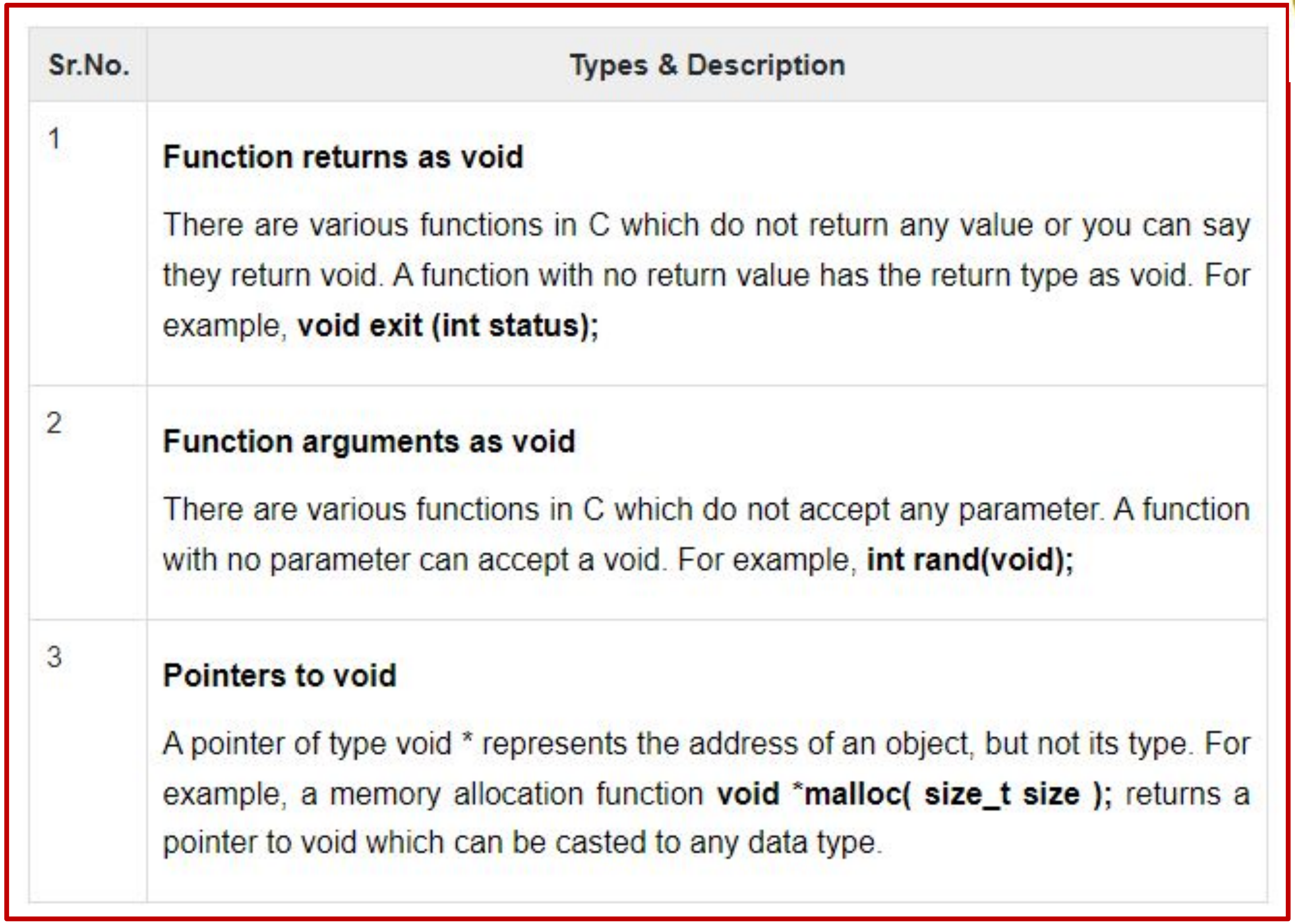
Floating-Point Types



Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
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

DATA TYPES

The void type



Sr.No.	Types & Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

L value and R value in expression

L value and R value refer to the left and right sides of the assignment operator.

An **L value** refers to an expression that represents a memory location. An **L value** means expression which can be placed on the left-hand side of the assignment operator. An expression which has memory location.

An examples of **L values** include variables, array elements, and deferential pointers.

An **R value** refers to an expression that represents a value that is stored at some location.

It can appear on the right-hand side of the assignment operator.

An examples of **R value** include literals (e.g numbers and characters), the results of arithmetic operations, and function return values.

L value and R value in expression

Note:

L value can be used as an R value , but an R-value cannot be used as an L value.

The unary operator ‘&’ can be used to get the address of L value.

The address of operator ‘&’ cannot be applied to R value.

```
int main() {
    int x = 10; // 'x' is an l-value
    int y = x; // 'x' is an r-value on the right side of the assignment, and 'y' is an l-value

    int* ptr = &x; // '&' operator gets the address of 'x', which is an l-value
    // The following lines would result in compilation errors:
    // int* ptr2 = &10; // Error: Cannot take the address of an r-value
    // &x = 20; // Error: 'x' is an l-value, but the left side of the assignment operator expects an l-value
    return 0;
                                                      }
```

Operator Precedence

- The precedence of operators in C indicates the order in which the operators will be evaluated in the expression.
- Associativity, on the other hand, defines the order in which the operators of the same precedence will be evaluated in an expression. Also, associativity can occur from either right to left or left to right.
- The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

OPERATORS

An operator is a symbol that operates on a value or a variable.

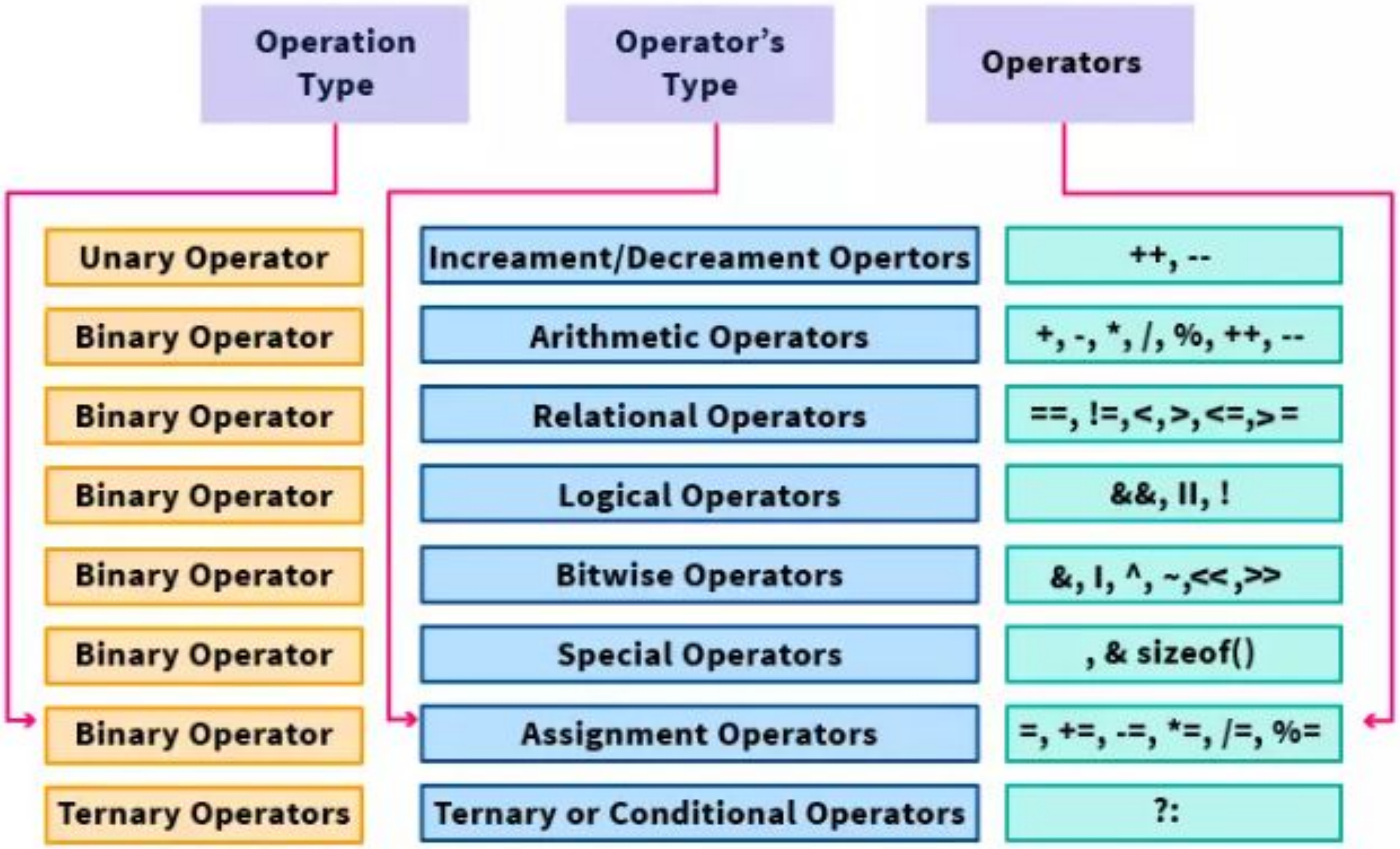
For example : **a+b** (or) **4+5**

A symbol that instructs the compiler to perform specific mathematical or logical functions is known as an operator.

Types of Operators

1. Unary Operator
2. Arithmetic Operator
3. Relational Operator
4. Logical Operator
5. The ternary or conditional operator
6. Increment and Decrement operator
7. Bitwise Operator
8. Comma Operator
9. Sizeof Operator

OPERATORS



Operation Type	Operator's Type	Operators
Unary Operator	Increament/Decreament Opertors	++, --
Binary Operator	Arithmetic Operators	+, -, *, /, %, ++, --
Binary Operator	Relational Operators	==, !=,<,>,<=,>=
Binary Operator	Logical Operators	&&, II, !
Binary Operator	Bitwise Operators	&, I, ^, ~,<<,>>
Binary Operator	Special Operators	, & sizeof()
Binary Operator	Assignment Operators	=, +=, -=, *=, /=, %=
Ternary Operators	Ternary or Conditional Operators	?:

OPERATORS

Arithmetic Operator

An arithmetic operator performs mathematical operations on numerical values such as **addition**, **subtraction**, **multiplication**, and **division** (constants and variables).

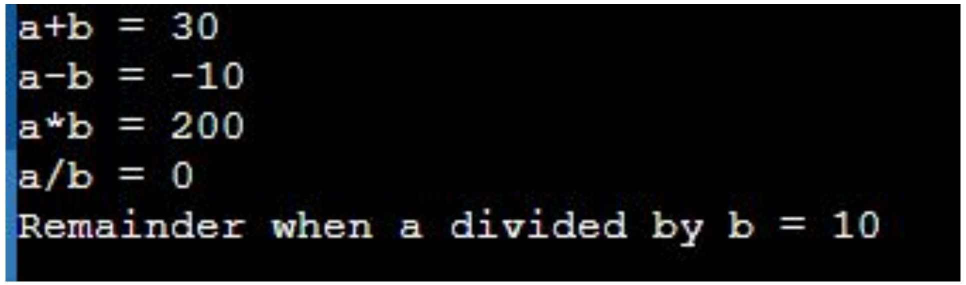
Operator	Meaning of Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

OPERATORS

Arithmetic Operator

```
#include <stdio.h>
int main()
{
    int a = 10,b = 20, c;
    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);
    return 0;
}
```

Output



```
a+b = 30
a-b = -10
a*b = 200
a/b = 0
Remainder when a divided by b = 10
```

OPERATORS

Relational Operators

A relational operator checks the **relationship** between **two operands**. Relational operators are specifically used to **compare two quantities** or values in a program. If the relation is true, it returns 1; if the relation is false, it returns value 0.

OPERATORS

Relational Operators

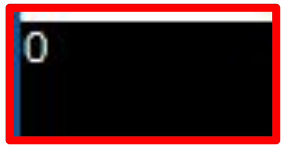
Operator	Meaning of Operator
==	Equal to
>	Greater than
<	Less than
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

OPERATORS

Relational Operators

```
#include <stdio.h>
int main() {
  int x = 5;
  int y = 3;
  printf("%d", x == y); // returns 0 (false) because 5
is not equal to 3
  return 0;
}
```

Output



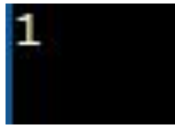
```
0
```

OPERATORS

Relational Operators

```
#include <stdio.h>
int main() {
  int x = 5;
  int y = 5;
  printf("%d", x == y); // returns 1 (true) because 5
is  equal to 5
  return 0;
}
```

Output



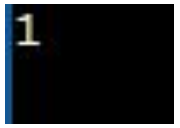
```
1
```

OPERATORS

Relational Operators

```
#include <stdio.h>
int main() {
  int x = 5;
  int y = 3;
  printf("%d", x != y); // returns 1 (true) because 5
is not  equal to 3
  return 0;
}
```

Output



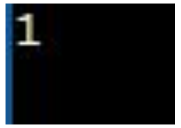
```
1
```

OPERATORS

Relational Operators

```
#include <stdio.h>
int main() {
  int x = 5;
  int y = 3;
  printf("%d", x>y); // returns 1 (true) because 5 is
greater than 3
  return 0;
}
```

Output



```
1
```

OPERATORS

Logical Operators

An expression containing logical operator **returns either 0 or 1** depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

OPERATORS

Logical Operators

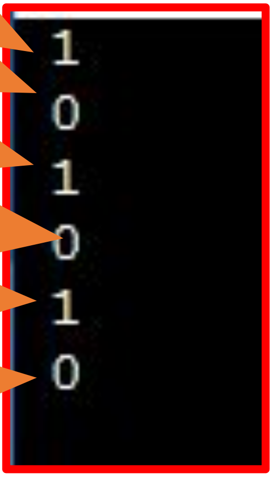
Operator	Meaning of Operator
&&	Logical AND. True only if all operands are true
||	Logical OR. True only if either one operand is true
!	Logical NOT. True only if the operand is 0

OPERATORS

Logical Operators

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;
    result = (a == b) && (c > b);
    printf(" %d \n", result);
    result = (a == b) && (c < b);
    printf(" %d \n", result);
    result = (a == b) || (c < b);
    printf(" %d \n", result);
    result = (a != b) || (c < b);
    printf(" %d \n", result);
    result = !(a != b);
    printf(" %d \n", result);
    result = !(a == b);
    printf(" %d \n", result);
    return 0;
}
```

Output

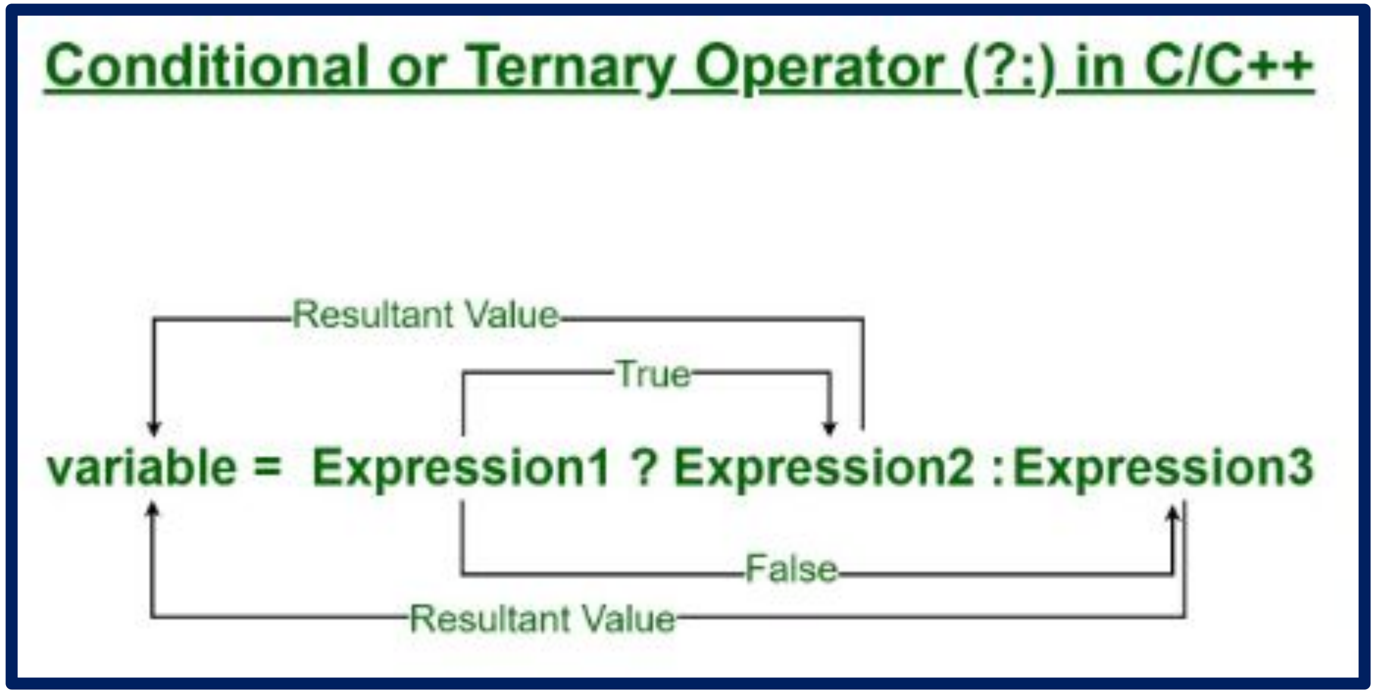


```
1
0
1
0
1
0
```

OPERATORS

The ternary or conditional operator

The **conditional operator** is similar to the **if-else** statement in that it follows the same algorithm, but the conditional operator takes **less space** and helps to write the if-else statements in the shortest possible way.


Conditional or Ternary Operator (?:) in C/C++
Resultant Value
True
variable = Expression1 ? Expression2 : Expression3
False
Resultant Value

OPERATORS

The ternary or conditional operator

Syntax:

The conditional operator is of the form

variable = Expression1 ? Expression2 : Expression3

It can be visualized into if-else statement as:

```
if(Expression1)
{
    variable = Expression2;
}
else
{
    variable = Expression3;
}
```

Since the Conditional Operator ‘?:’ takes three operands to work, hence they are also called ternary operators.

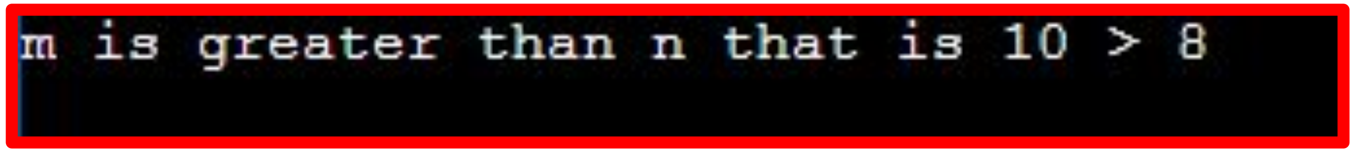
OPERATORS

The ternary or conditional operator

```
#include <stdio.h>
int main()
{
   int m = 10, n = 8;

   (m > n) ? printf("m is greater than n that is %d > %d",m, n)
           : printf("n is greater than m that is %d > %d",n, m);
   return 0;
}
```

Output



```
m is greater than n that is 10 > 8
```

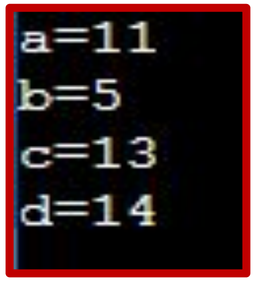
OPERATORS

Increment and Decrement operator

C programming has two operators **increment ++ and decrement --** to change the value of an operand (constant or variable) **by 1**. Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are **unary operators**, meaning they only operate on a single operand.

```
#include <stdio.h>
int main()
{
    int a = 10,b=6,c=12,d=15;
    printf("a=%d \n", ++a);
    printf("b=%d \n", --b);
    printf("c=%d \n", ++c);
    printf("d=%d \n", --d);
    return 0;
}
```

Output



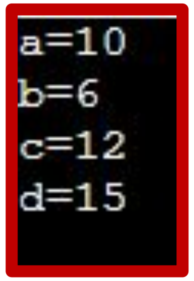
```
a=11
b=5
c=13
d=14
```

OPERATORS

Increment and Decrement operator

```
#include <stdio.h>
int main()
{
    int a = 10,b=6,c=12,d=15;
    printf("a=%d \n", a++);
    printf("b=%d \n", b--);
    printf("c=%d \n", c++);
    printf("d=%d \n", d--);
    return 0;
}
```

Output



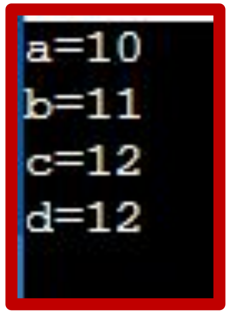
```
a=10
b=6
c=12
d=15
```

OPERATORS

Increment and Decrement operator

```
#include <stdio.h>
int main()
{
    int a = 10;
    printf("a=%d \n", a++);
    printf("b=%d \n", a);
    printf("c=%d \n", ++a);
    printf("d=%d \n", a);
    return 0;
}
```

Output



```
a=10
b=11
c=12
d=12
```

Bitwise Operators in C

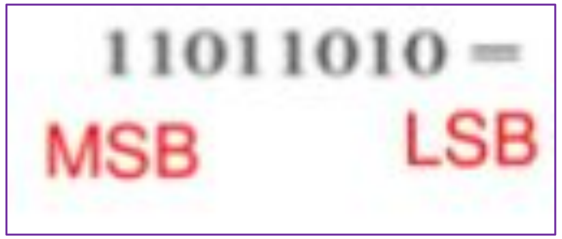
- Bitwise operators are used to **manipulate one or more bits** from integral operands like char, int, short, long.
- Bitwise operators operate on individual bits of integer (int and long) values.
- If an operand is shorter than int, it is promoted to int before doing the operations.
- In the calculation, just the individual bits of a number are considered, not the complete number.
- Negative integers are stored or represented in two’s complement form. For example, -4 is 1111 1111 1111 1111 1111 1111 1111 1100.

Bitwise Operators in C

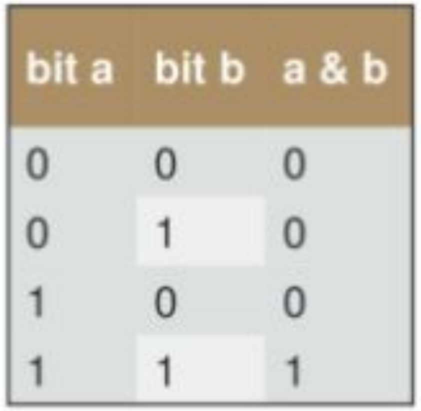
Operator	Description	Example
& (AND)	Returns AND of input values	a & b
| (OR)	Returns OR of input values	a | b
^ (XOR)	Returns XOR of input values	a ^ b
~ (Complement)	Returns the one’s complement	~ a

Bitwise Operators in C

Bitwise AND ‘&’



Truth Table



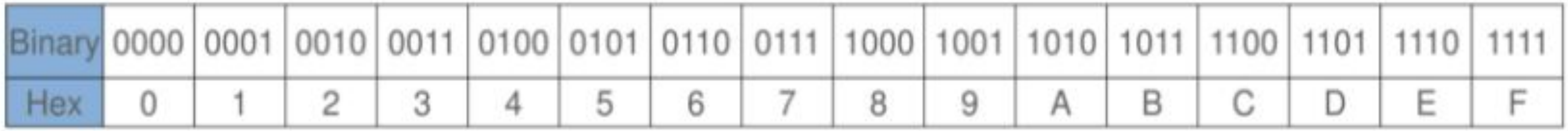
bit a	bit b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
int  a=2; // 0010
int b=3; // 0011
```

```
int  c=a & b;
```

```
0 0 1 0
0 0 1 1
---------
0 0 1 0
```

Output:
2

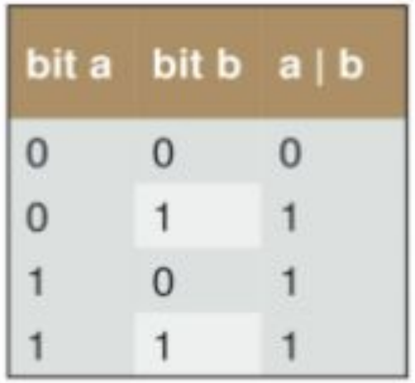


Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Bitwise OR ‘|’

Truth Table



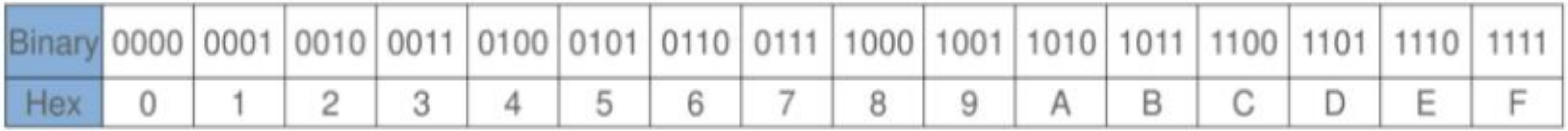
bit a	bit b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

```
int  a=2; // 0010
int b=3; // 0011
```

```
int  c=a | b;
```

```
0 0 1 0
0 0 1 1
---------
0 0 1 1
```

Output:
3



Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Bitwise XOR ‘^’

Truth Table

bit a	bit b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

```
int  a=2; // 0010
int b=3; // 0011
```

```
int  c=a ^ b;
```

```
0 0 1 0
0 0 1 1
---------
0 0  0 1
```

Output:
1

Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Bitwise Negation '~'

Truth Table

bit a	~ a (1's complement of a)
0	1
1	0

```
int  a=2; // 0010
```

```
int  c=~a;
```

```
0 0 1 0
---------
1 1  0 1
```

Output:
-3

Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Bitwise Negation '~'

```
#include <stdio.h>
int main()
{
 printf("%d",~2);
   return 0;
}
```

Input	2=> 0 0 1 0
Apply negation	1 1 0 1
Take 2's complement(1's complement+1)	1 1 0 1 0 0 1 0 1 ---------- 0 0 1 1
Equivalent decimal value is	-3

Output:
-3

Bitwise Operators in C

Shift operators are used to shift a number's bits left or right. Basically two types of shift operators,

1. Left Shift Operator (<<)
2. Right Shift Operator (>>)

Left Shift Operator

It moves the first operand's bits to the left by the number of positions specified by the second operand. Simultaneously, the empty spaces left by the shifted bits are filled with zeroes.

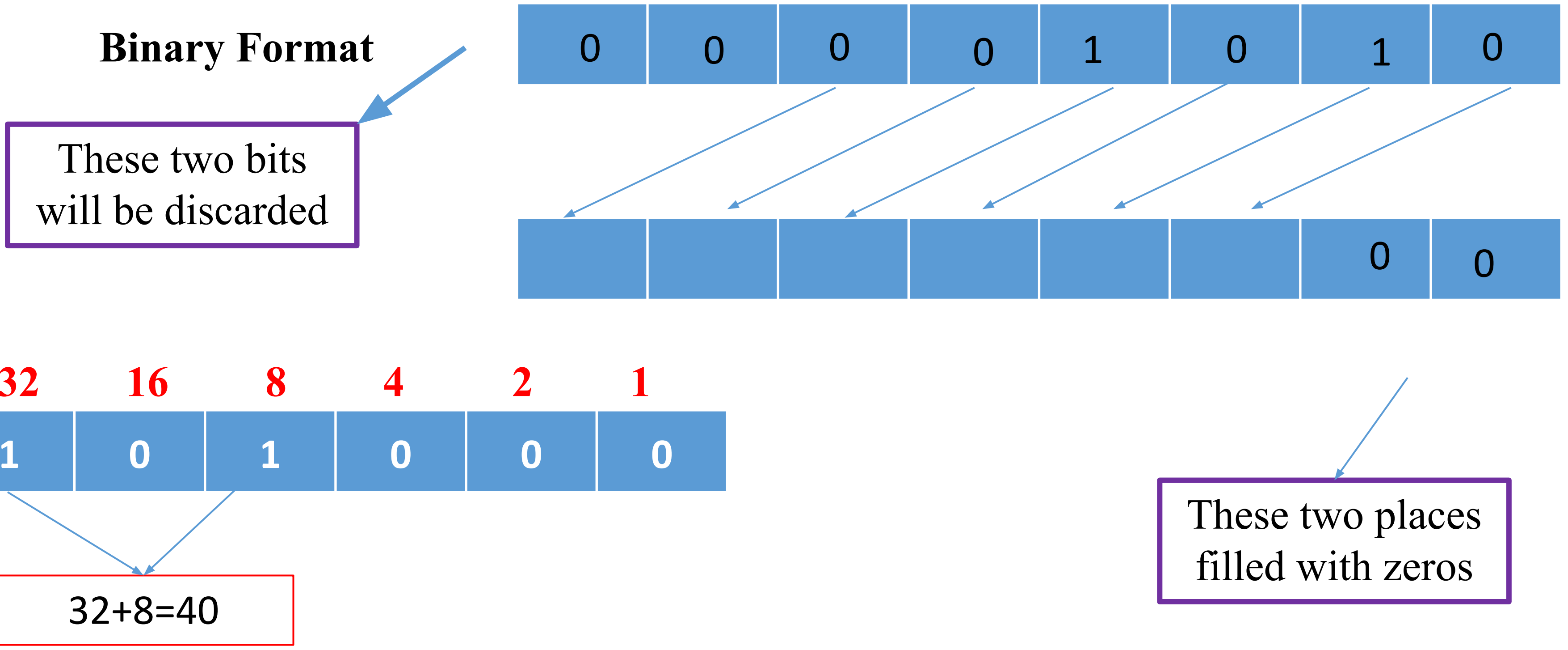
Right Shift Operator

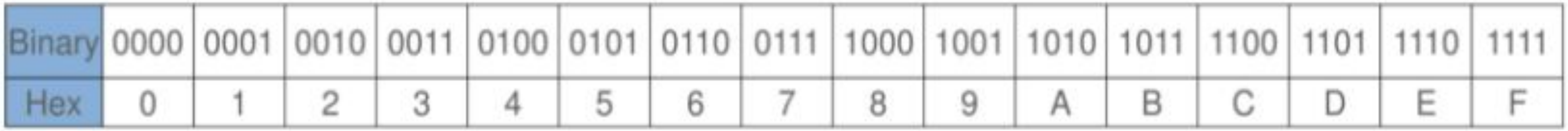
The right shift operator (>>) shifts the first operand to the right by the specified number of bits. Excess bits that are shifted to the right are discarded.

Bitwise Operators in C (Left Shift)

Example:

```
int a=10;
c=a<<2;
```





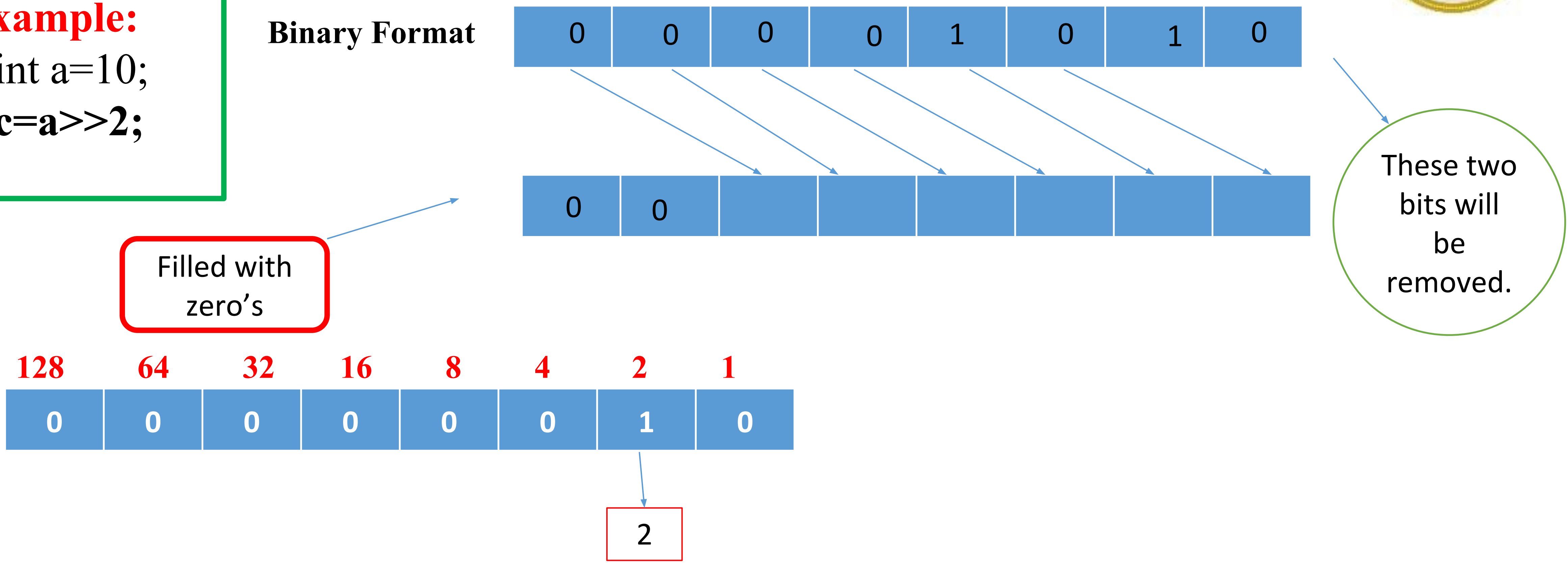
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

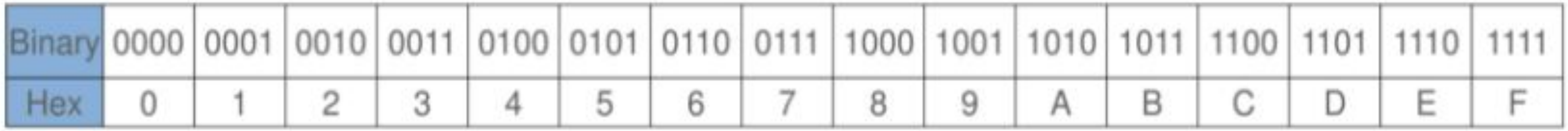
Bitwise Operators in C Right Shift)

Example:

int a=10;

c=a>>2;





Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Program

```
#include <stdio.h>
int main()
{
    printf("%d",10<<2);
    return 0;
}
```

```
Output: 40
```

Bitwise Operators in C

Program

```
#include <stdio.h>
int main()
{
    printf("%d",10>>2);
    return 0;
}
```

```
Output: 2
```

Bitwise Operators in C (Signed Integer)

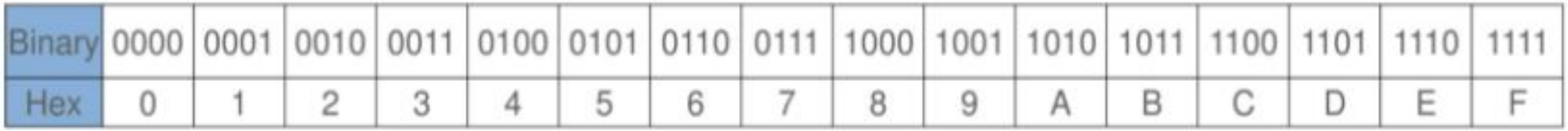
Signed integers are numbers with a “+” or “-“ sign. If n bits are used to represent a signed binary integer number, then out of n bits,1 bit will be used to represent a sign of the number and rest (n - 1)bits will be utilized to represent magnitude part of the number itself.

There are various ways of representing signed numbers in a computer −

- **Sign and magnitude**
- **One's complement**
- **Two's complement**

Bitwise Operators in C (Signed Integer)

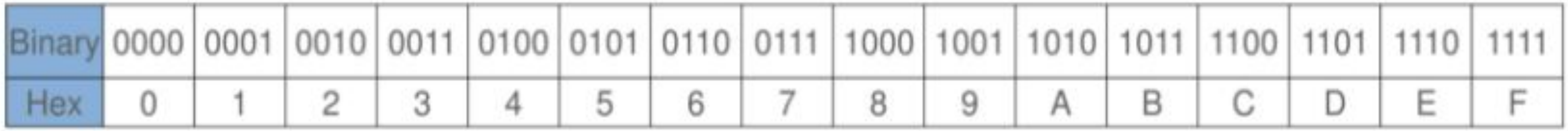
-8>>0	1 0 0 0	No change	-8
-8>>1	1 0 0 0	1 1 0 0	-4
-4>>1	1 1 0 0	1 1 1 0	-2
-2>>1	1 1 1 0	1 1 1 0	-1
-1>>1	1 1 1 0	1 1 1 1	0



Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C (Signed Integer)

-8<<0	1 0 0 0	No change	-8
-8>>1	1 0 0 0	1 1 0 0	-16
-4>>1	1 1 0 0	1 1 1 0	-32
-2>>1	1 1 1 0	1 1 1 0	-64
-1>>1	1 1 1 0	1 1 1 1	-128



Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

Example: **-13>>2**

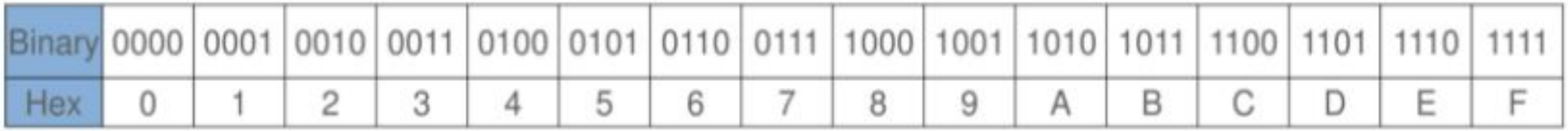
Perform right shift for two times.

How?

```
#include <stdio.h>
int main()
{
    printf("%d",-13>>2);
    return 0;
}
```

Output:

```
-4
```



Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

The signed integer can not be processed directly as it is. It should be represented in 2's complement format. Now the computer will perform shifting of bits. To find the equivalent decimal value , again take 2's complement. The **MSB** will be used to represent a sign of the number.

Input	Output	Task
-13	1 1 0 1	Binary representation of 13
1 1 0 1	0 0 1 0	One's complement
0 0 1 0	0 0 1 1	Add 1
0 0 1 1	1 1 0 0	Right shift
To find the equal decimal value take 2's complement		
Output=(-4)		

Bitwise Operators in C

Example: **-13<<2**

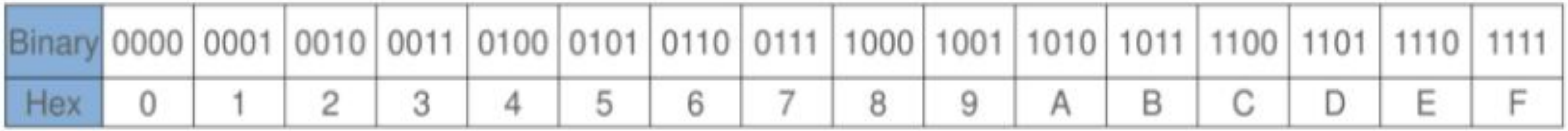
Perform right shift for two times.

```
#include <stdio.h>
int main()
{
    printf("%d",-13<<2);
    return 0;
}
```

Output:

```
-52
```

How?

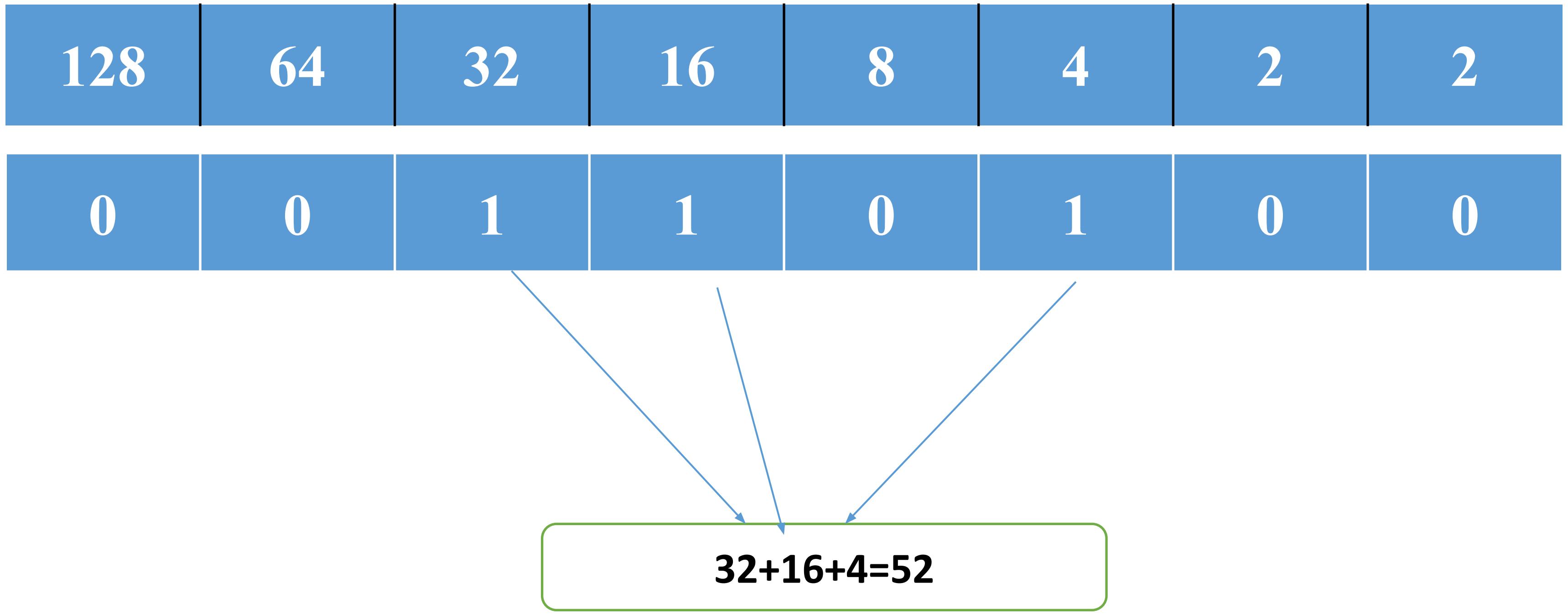


Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Bitwise Operators in C

-13>>2	Perform shift operation for two times	
Binary form of 13	Take 2’s complement of 13	
13	0 0 0 0 1 1 0 1	Binary format
	1 1 1 1 0 0 1 0	One’s complement
	1 (+)	Add 1
	1 1 1 1 0 0 1 1	Perform left shift(two bits will be removed)
After shifting	**1 1 0 0 1 1 0 0**	**Actual output of signed left shift**
To get equivalent decimal value	Take 2’s complement again	
	1 1 0 0 1 1 0 0	
	0 0 1 1 0 0 1 1	One’s complement
	1(+)	Add 1
	0 0 1 1 0 1 0 0	Find the equivalent decimal value

Bitwise Operators in C



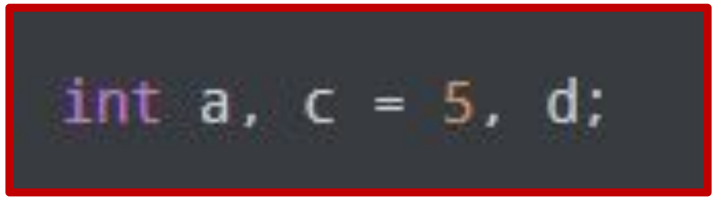
128	64	32	16	8	4	2	2
0	0	1	1	0	1	0	0

32+16+4=52

OPERATORS

Comma operator

Comma operators are used to link related expressions together.



```
int a, c = 5, d;
```

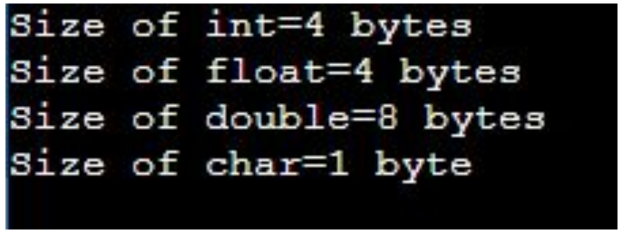
OPERATORS

sizeof operator

The sizeof is a unary operator that returns the size of data (constants, variables, array, structure, etc).

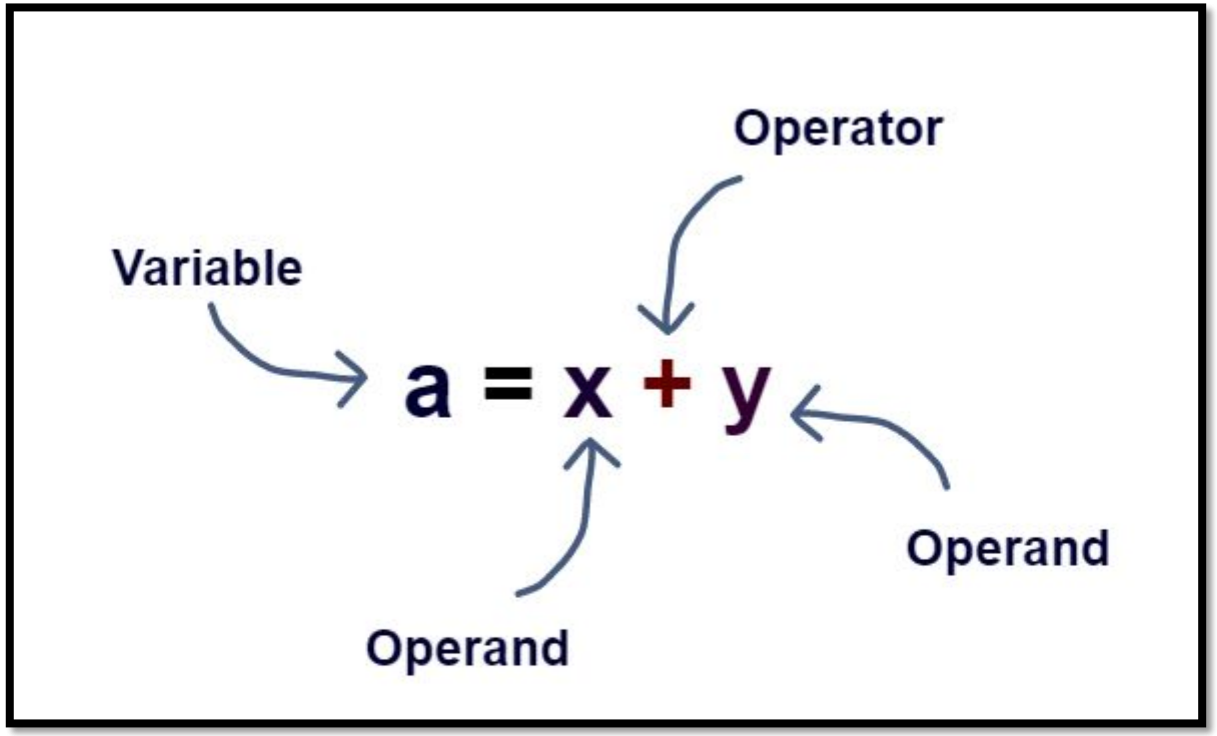
```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));
    return 0;
}
```

Output


```
Size of int=4 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte
```

EXPRESSIONS

In C, an expression is a set of operands and operators that computes a single value stored in a variable. The operator represents the action or operation to be carried out. The operands are the items to which the operation is applied.



EXPRESSIONS

An expression can be defined depending on the position and number of its operator and operands.

Infix Expression (operator is used between the operands)

a = x + y

Postfix Expression (operator is used after the operands)

xy+

Prefix Expression (operator is used before the operands)

+xy

Unary Expression (one operator and one operand)

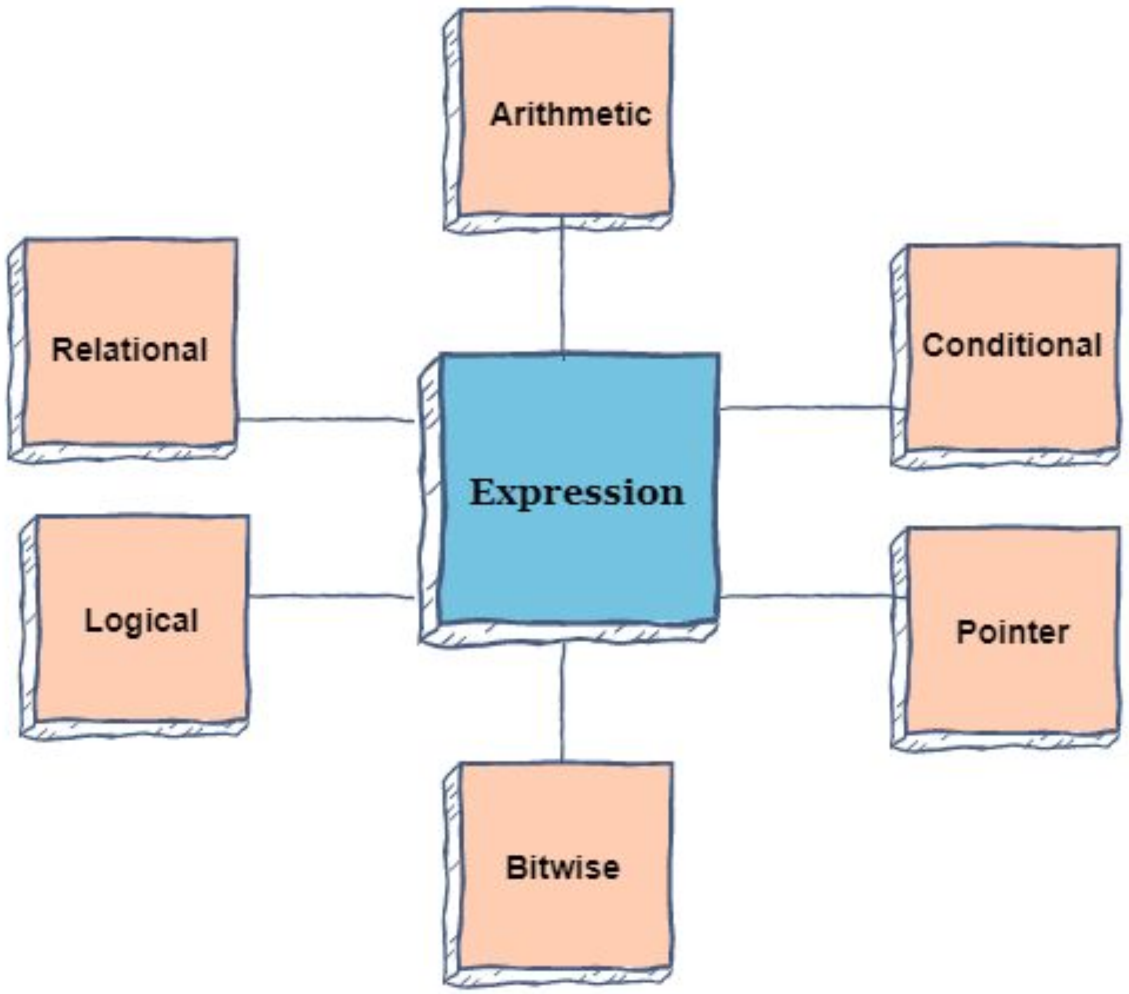
x++

Binary Expression (one operator and two operands)

x+y

EXPRESSIONS

Types of Expression



EXPRESSIONS

Types of Expression

Arithmetic Expression

It consists of arithmetic operators (+ , - , * , and /) and computes values of int, float, or double type.

Relational Expression

It usually consists of comparison operators (> , < , >= , <= , === , and !==) and computes the answer in the bool type, i.e., true (1) or false (0).

Logical Expression

It consists of logical operators (&&, ||, and !) and combines relational expressions to compute answers in the bool type.

EXPRESSIONS

Types of Expression

Conditional Expression

It consists of conditional statements that return true if the condition is met and false otherwise.

Pointer Expression

It may consist of an ampersand (&) operator and returns address values.

Bitwise Expression

It consists of bitwise operators (>>, <<, ~, &, |, and ^) and performs operations at the bit level.

EXPRESSIONS

```
#include <stdio.h>
int main(){
  int a = (6 * 2) + 7 - 9; //Arithmetic Expression
  printf("The arithmetic expression returns: %d\n", a);
   int b = 10;
  printf("The relational expression returns: %d\n", b % 2 == 0); //Relational Expression
  int c = (7 > 9) && ( 5 <= 9); //Logical Expression
  printf("The logical expression returns: %d\n", c);
  int d = (34 > 7) ? 1 : 0; //Conditional Expression
  printf("The conditional expression returns: %d\n", d);   //Pointer Expression
  int e = 20;
  int *addr = &e;
  printf("The pointer expression returns: %p\n", addr); //Bitwise Expression
  int f = 10;
  int shift = 10 >> 1;
  printf("The bitwise expression returns: %d\n", shift);
  return 0;
}`
```

EXPRESSIONS

Output

```
The arithmetic expression returns: 10
The relational expression returns: 1
The logical expression returns: 0
The conditional expression returns: 1
The pointer expression returns: 0x7ffdb0430704
The bitwise expression returns: 5
```

Operator Precedence

Let us consider an example:

int x=5-17*9;

In C, the precedence of * is higher than – and =. Hence, 17*9 is evaluated first. Then – is evaluated and result is assigned to the variable x.

Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Operator Precedence

Parentheses:

The expressions within parentheses are evaluated first. This allows you to control the order of evaluation.

int result = (2 + 3) * 4; // result = 20

(2+3) is evaluated first and then it is multiplied with 4.

1) 2+3=5

2) 5*4=20

Postfix operators:

Postfix operators like function calls and array subscripting have higher precedence than most other operators.

int arr[5] = {1, 2, 3, 4, 5};

int value = arr[2] * 3; // value = 9

arr[2]=3

3*3=9

Operator Precedence

Parentheses:

The expressions within parentheses are evaluated first. This allows you to control the order of evaluation.

int result = (2 + 3) * 4; // result = 20

(2+3) is evaluated first and then it is multiplied with 4.

1) 2+3=5

2) 5*4=20

Postfix operators:

Postfix operators like function calls and array subscripting have higher precedence than most other operators.

int arr[5] = {1, 2, 3, 4, 5};

int value = arr[2] * 3; // value = 9

arr[2]=3

3*3=9

Operator Precedence

Unary operators:

Unary operators, such as the increment (++) and decrement (--) operators, are applied next.

int a = 5;

int result = ++a; // a becomes 6, (Right to Left)

result = 6

Multiplicative operators:

Multiplication (*), division (/), and modulo division (%) operators have higher precedence than additive operators.

int result = 10 + 2 * 5; // result = 20 (multiplication is done first)

Additive operators:

Addition (+) and subtraction (-) operators are applied after the multiplicative operators.

int result = 10 - 2 + 5; // result = 13 (subtraction is done first) (Left to Right)

Operator Precedence

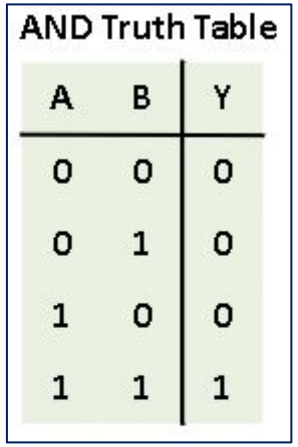
Relational and equality operators:

These operators compare values and have higher precedence than logical operators.

int result = 5 < 10 && 2 == 2; // result = 1 (true)

Order of Evaluation

1. 5<10 =>1(Hence 5 is less than 10)
2. 2==2=>1
3. Finally && is evaluated (1&&1)=1



AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Operator Precedence

Logical operators:

Logical AND (&&) and logical OR (||) operators are applied after relational and equality operators.

int result = 5 < 10 || 2 > 5; // result = 1 (true)

Assignment operators:

Assignment operators (=, +=, -=, *=, /=, %=) are applied after all other operators.

int a = 5;

a += 3; // a becomes 8

a=a+3;

a=5+3=>8

Conditional operator (ternary operator):

The conditional operator (?:) is evaluated after all other operators. It is used for conditional expressions.

int a = 5; int result = (a > 10) ? a : 10; // result = 10

Thank
You