



Programming for Problem Solving (ESCG101)



Course Details

- ❑ Program Name: B.Tech (CSE)
- ❑ Course Name: Programming for Problem Solving
- ❑ Course Code: ESCG101
- ❑ Contact: 3L
- ❑ Credit: 3
- ❑ Pre-requisite
 - basic concept of algorithm
 - basic concept of flowchart



Complete Syllabus

- ❑ **Module 1** - Introduction to Programming: Introduction to components of a computer system (disks, memory, processor, operating system, compilers). Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm. Flowchart/Pseudocode with examples. Algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and Logical Errors in compilation, object and executable code.
- ❑ **Module 2** - Arithmetic expressions and precedence ,Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching ,Iteration and loops, Arrays: Arrays (1-D, 2-D), Character arrays and Strings
- ❑ **Module 3** - Function: Built in libraries, Parameter passing in functions, call by value, Passing arrays to functions: call by reference, Recursion: Example programs, Finding Factorial, Fibonacci series, Ackerman function, Quick sort or Merge sort.
- ❑ **Module 4** - Structure: Structures, Defining structures and Array of Structures, Pointers: Idea of pointers, Defining pointers, Use of Pointers in self-referential structures, notion of linked list
- ❑ **Module 5** - File handling, File concepts, parameters, modes of operation



Textbooks/References

- Byron Gottfried, Schaum's Outline of Programming with C, McGraw-Hill
- E. Balaguruswamy, Programming in ANSI C, Tata McGraw-Hill
- Gary J. Bronson, A First Book of ANSI C, 4th Edition, ACM
- Kenneth A. Reek, Pointers on C, Pearson
- Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice Hall of India



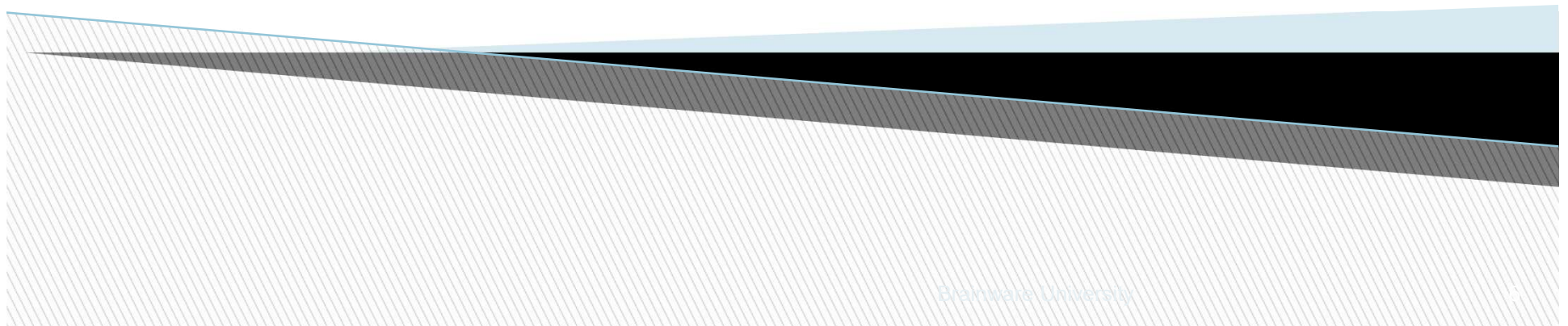
Course Objectives

From this course, the students will be able to:

- Define the basic concepts of computer system and recall the ideas of algorithm and flowchart
- Discuss the basic concepts and ideas of C programming language
- Explain the concepts of loops and functions and apply in solving a problem
- Identify the use of structure and pointer and compare the importance of using file in programming language



Module 1





Contents of Module 1

- Introduction to Programming: Introduction to components of a computer system (disks, memory, processor, operating system, compilers).
- Idea of Algorithm: steps to solve logical and numerical problems. Representation of Algorithm.
- Flowchart/Pseudocode with examples. Algorithms to programs; source code, variables (with data types) variables and memory locations, Syntax and Logical Errors in compilation, object and executable code



Objectives of Module 1

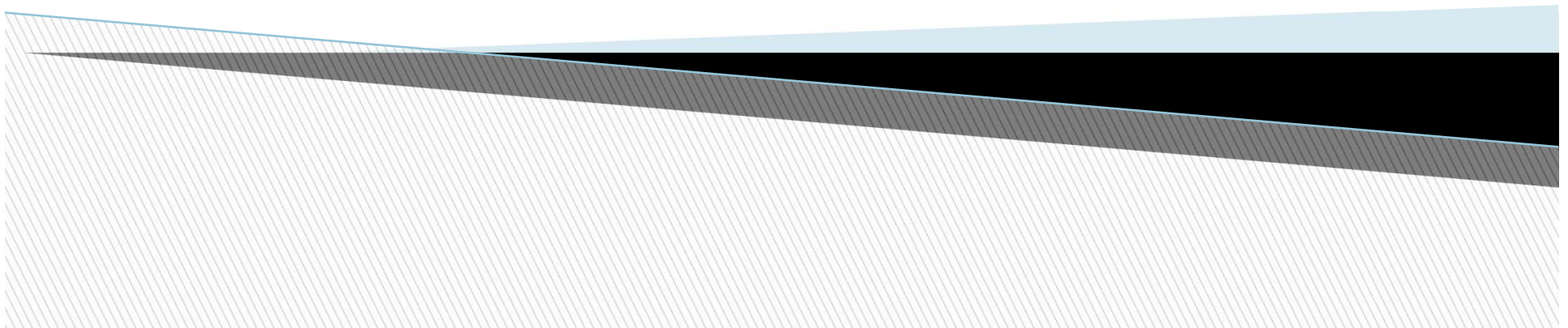
- ❑ Learn components of a computer system like disks, memory, processor, operating system, compilers etc.
- ❑ Get the idea of algorithm to solve problems.
- ❑ Draw flowchart for a problem.
- ❑ Learn different data types, variables and memory locations.
- ❑ Identify errors like syntax and logical errors.
- ❑ Learn different types of operators.



Introduction to Programming

Programming Languages: Programming languages are used to communicate instructions to computers.

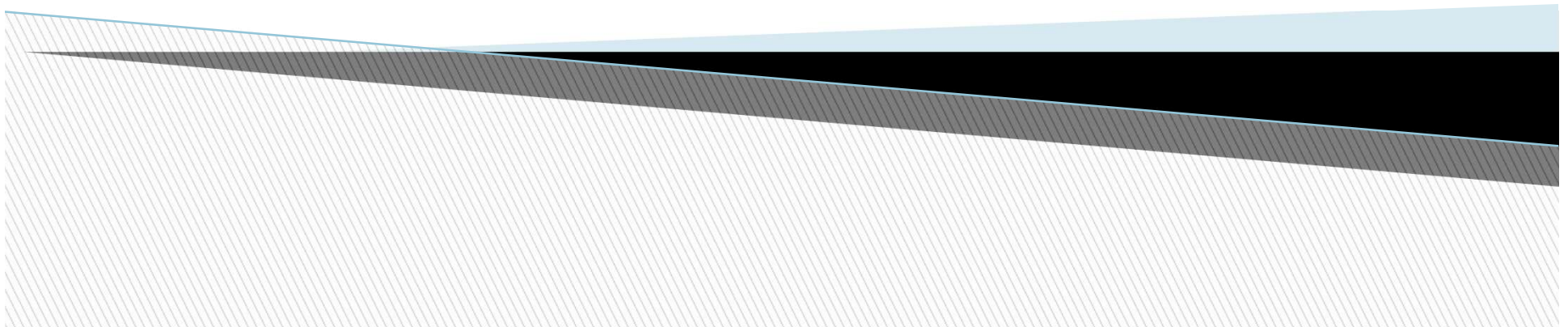
Examples include C, Python, Java and many more. Each language has its syntax, rules, and purposes.





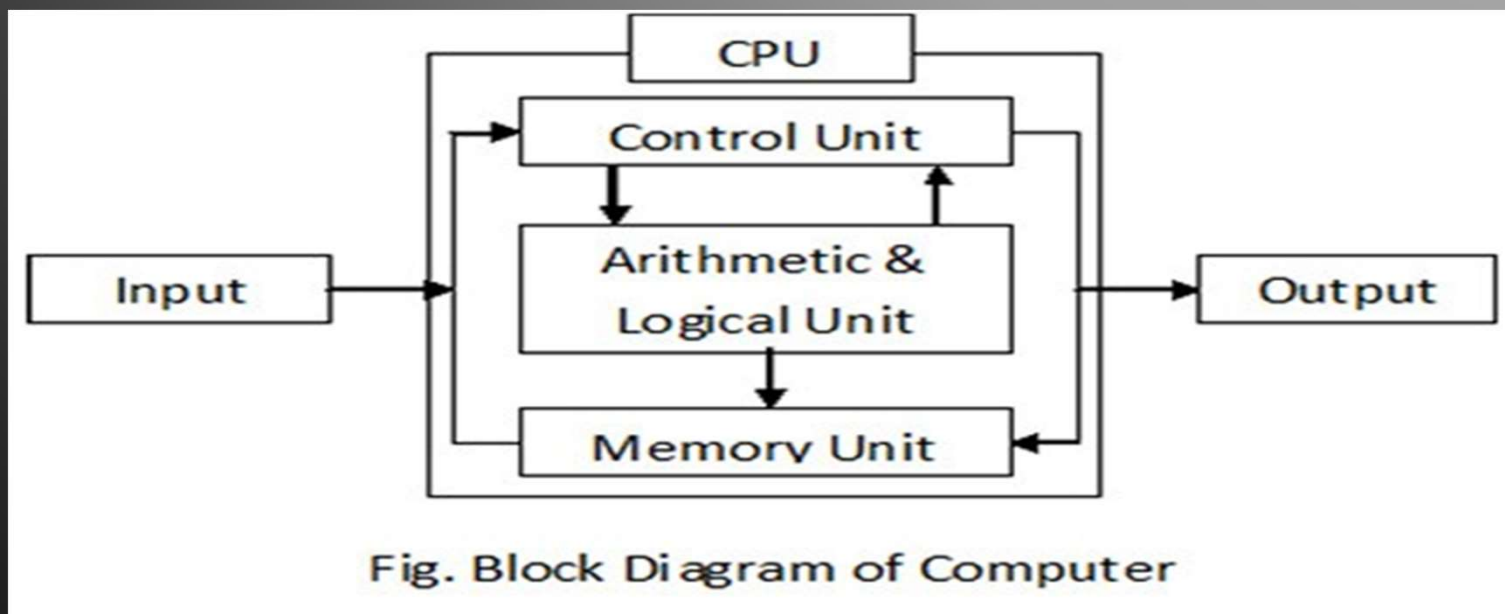
CONTD..

- ❑ **Syntax:** refers to the rules governing the structure of code in a programming language. Correct syntax is essential for the code to run without errors.
- ❑ **Debugging:** is the process of identifying and fixing errors (bugs) in code.



Computer System

- Input devices
- CPU (CU+ALU+MU)
- Output devices





Computer System

- Example of input device – keyboard, mouse etc
- Example of output device – monitor, printer etc
- Control unit (CU) is the brain of computer
- Arithmetic Logic unit (ALU) is used for performing arithmetic and logical operations
- Memory unit (MU) is the memory used for storing data
 - Primary memory, secondary memory



Algorithm

- It is a step by step method of solving a problem
- Example: to add two numbers

1. start
2. take first number
3. take second number
4. add first and second number
5. stop

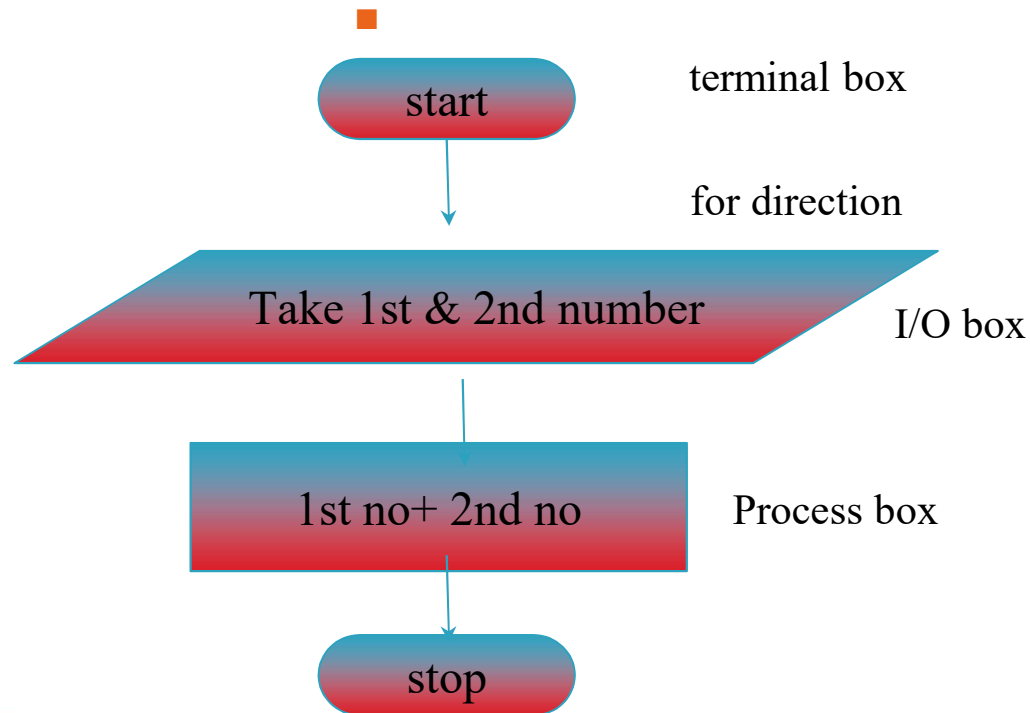


Features of an algorithm

- Precision
 - steps precisely defined
- Uniqueness
 - no ambiguity
- Finiteness
 - end to some solution
- Input/Output

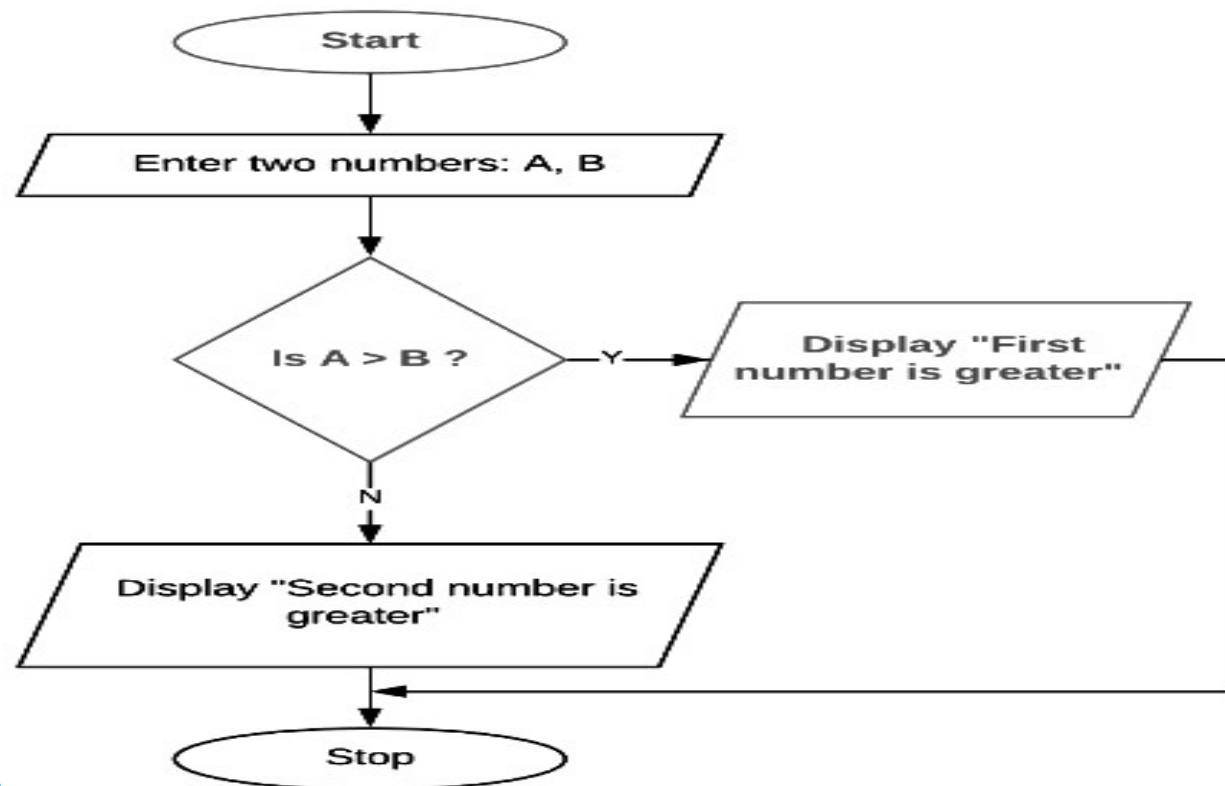
Flowchart

- The pictorial representation of the step by step solution of a given problem
- Pictorial representation of an algorithm





Flowchart for greatest of two numbers





Error and Types

- Unexpected happening or illegal operation performed by the user which results in abnormal working of the program
- Types:
 - Syntax-- violates the rules (compile time)
 - Logical-- unexpected due to logic
 - writing $a+b$ rather than $a-b$
 - Run-time-- say (a/b) , it works except $b=0$

C Programming Language



- ❑ Developed by Dennis Ritchie
- ❑ In 1972 at AT&T Bell Laboratory
- ❑ Keyword- word with predefined meaning
 - Ex-- int, if, void etc
- ❑ Identifier- any name given to a variable, function, array etc
- ❑ Variable- a name to store data in memory location



Classifying Data types

□ Basic (primitive/primary) types include:

- Integers
- Characters
- Floating points (real numbers)
- Boolean

□ Secondary/derived types include:

- Array
- Pointer
- Structure
- String



Uses

- 'int' (Integers) is used for storing integers values (-5, 0, 5 etc.)
- 'float' is used to for storing decimal numbers (5.5, -5.7 etc.)
- 'char' (character) is used for storing single letter or special symbols (a,C,\$ etc.)



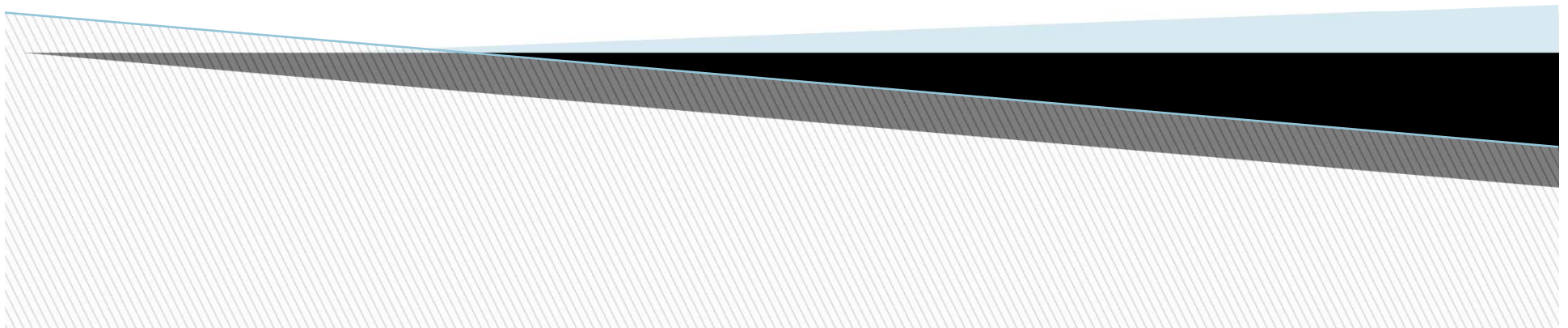
Storage Capacity

- 'int' = 2 bytes (16 bit compiler)
= 4 bytes (32 bit compiler)
- 'float' = 4 bytes
- 'char' = 1 byte



Calculate the total memory required (both bytes and kilobytes) to store an array of 156 integers and 52 floating-point numbers where the machine has 32bit compiler?

Ans: 832 bytes (0.8125KB)





Format specifier

- Format specifiers are needed to access different data types
- int - %d
- float - %f
- char - %c
- String - %s
- Pointer- %p



Keywords

- They are the fixed defined words with some meaning
- Example :
 - **Void:** is a keyword in C used to indicate that a function does not return any value. It is typically used in function declarations and definitions to specify that the function does not produce a result that can be used by other parts of the program.
 - **main:** is required in every C program, and it must return an integer value (usually 0 to indicate successful execution) to the operating system when the program terminates.



Identifier

- Any name assigned to represent something
- Example :
 - Say roll number 1 represents a student
 - Similarly any name used to represent in C language like variable, function etc



Variable

- ❑ Used to hold values
- ❑ It is a named-storage location that holds a value.
- ❑ Example : `int var;`
 - Here var is a variable that will hold integer data type values
 - There can be any name except keywords



Rules for Identifier Name

- ❑ Any name except keywords like int, float, if etc.
- ❑ It can contains alphabets, numbers etc.
- ❑ Must start with a letter
- ❑ Contains special character
 - Underscore (_) only
- ❑ First character of name cannot be a number
- ❑ It is case-sensitive



Operators

- ❑ An operator is a symbol that represents a specific operation to be performed on one or more operands.
- ❑ Operator is used to manipulate data and variables
- ❑ Example: $a + b$, where a & b are operands and $+$ is an operator



Types of Operators

- ❑ Arithmetic Operators
- ❑ Relational Operators
- ❑ Assignment Operator
- ❑ Logical Operators
- ❑ Conditional Operators
- ❑ Bitwise Operators
- ❑ Increment/Decrement Operators



Arithmetic Operators

- Arithmetic operators used to perform arithmetic operations
- The arithmetic operators in C language:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo



Relational Operators

- Also known as comparison operators, are symbols used in programming languages to compare two values or expressions. These operators allow you to determine the relationship between the values and make decisions based on their comparisons. The relational operators are:

>	Greater than
<	Less than
>=	greater than Equal to
<=	Less than Equal to
==	Equal to
!=	Not Equal to



Assignment Operators

- Assignment operators are used to assign value to a variable
- '=' sign is used to assign a value to variable
- Example: `int a = 8;`



Logical Operators

- Logical operators are used in programming to perform logical operations on boolean values (true or false).

Examples:

- Logical AND (&&): It returns 'true' if both of its operands are true, otherwise it returns false.
- Logical OR (||): logical OR operator returns true if at least one of its operands is true, and it returns false if both operands are false.
- Logical NOT (!): The logical NOT operator negates the value of its operand. If the operand is true, it returns false, and if the operand is false, it returns true.



Conditional Operators

The conditional operator, often referred to as the ternary operator, is a concise way to create conditional expressions. It's used to make decisions between two values or expressions based on a condition.

Syntax: `condition ? value_if_true : value_if_false`

- ❑ `value_if_true`: This is the value or expression that is returned if the condition is true.
- ❑ `value_if_false`: This is the value or expression that is returned if the condition is false.

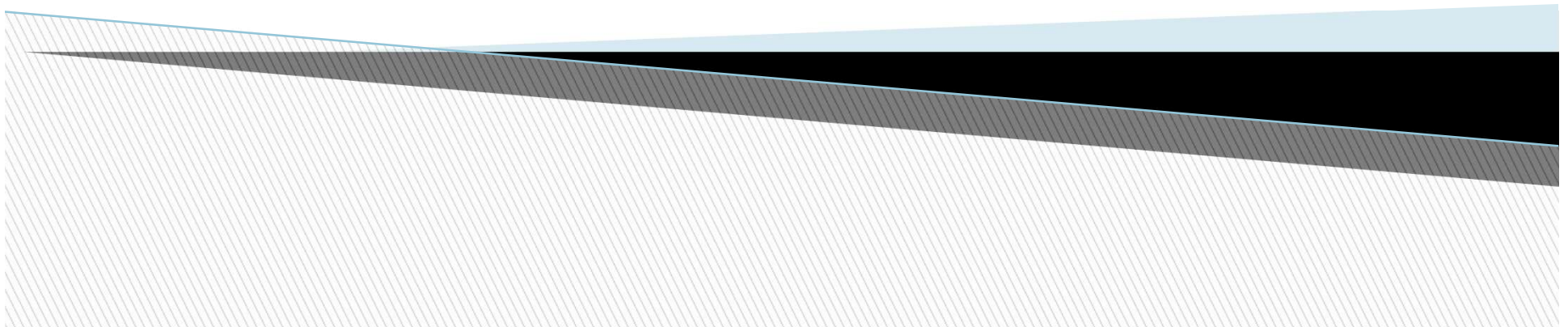


CONTD..

Example:

```
#include <stdio.h>
```

```
int main() {  
    int x = 10;  
    int y = 20;  
    int max_value = (x > y) ? x : y;  
    printf("The maximum value is: %d\n",max_value);  
    return 0;  
}
```





Bitwise Operators

- Bitwise operators in C are used to perform operations at the bit level of integers. They manipulate individual bits of an integer value rather than the entire value itself.
- There are various bitwise operators in C as following:

& Bitwise AND

| Bitwise OR

<< Left Shift

>> Right Shift



CONTD..

Bitwise AND (&):

Performs a bitwise AND operation between the corresponding bits of two operands.

Eg: A=5

B=3 A & B=?

ANS. 1

Truth Table of Bitwise AND (&) Operator

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1



CONTD..

Bitwise OR (|):

Performs a bitwise OR operation between the corresponding bits of two operands.

Eg: A=5

B=3 A | B=?

ANS. 7

Truth table

x	y	x y
0	0	0
0	1	1
1	0	1
1	1	1



CONTD..

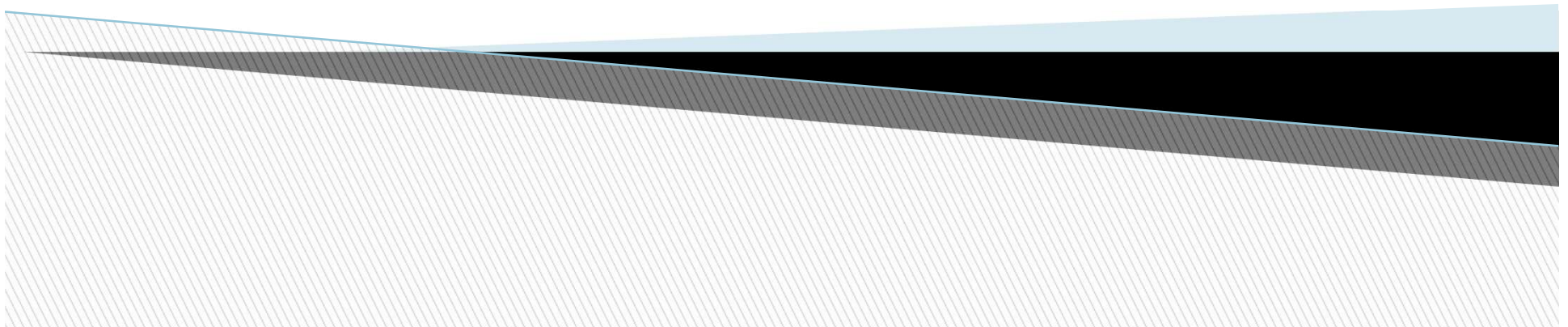
Left Shift (<<):

Shifts the bits of the left operand to the left by a specified number of positions.

Eg: $A=5$

$A \ll 2 = ?$

ANS. 20





CONTD..

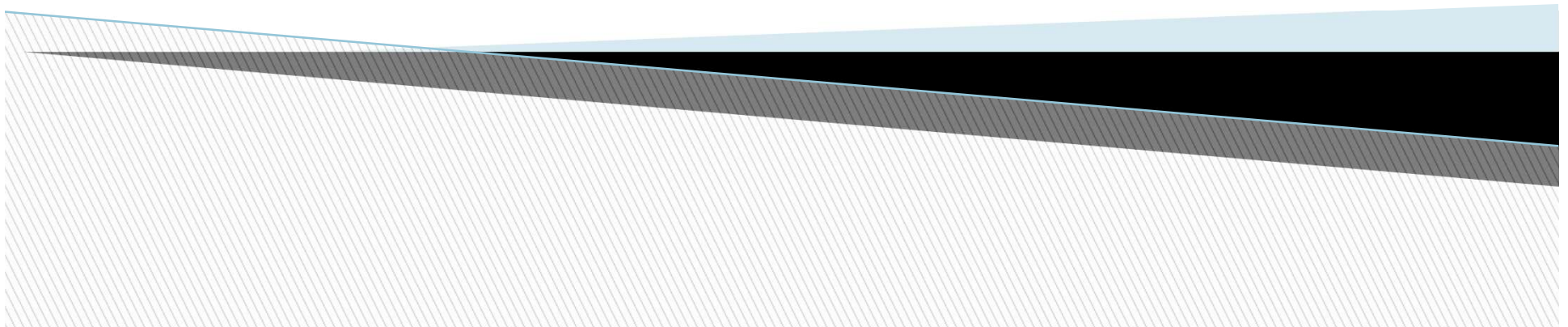
Right Shift (>>):

Shifts the bits of the left operand to the right by a specified number of positions.

Eg: $A=15$

$A \gg 2 = ?$

ANS. 3





Questions:

1) $A=12$

$B=25$ $A \& B=?$

Ans. 8

2) $A=25$

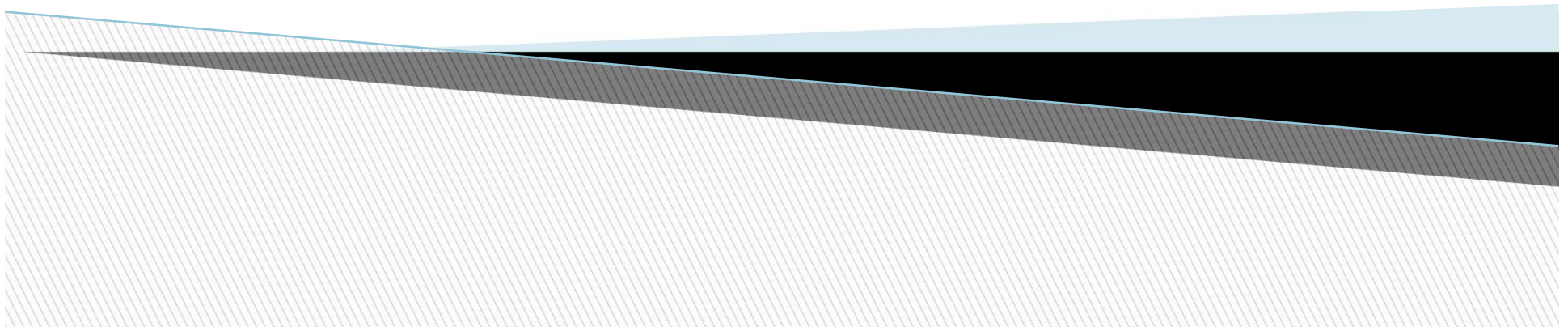
$B=18$ $A | B=?$

Ans. 27

3) $A=13$ $A >> 2=?$ Ans. 3

4) $A=25$ $25 << 2=?$

Ans. 100





Increment/Decrement Operators

- Increment operator are used to increase the value by 1
- Decrement operator decrease the value by 1
- Increment operator is denoted by ++
- Decrement operator is denoted by --



Increment/Decrement Operators

Increment:

int x=5;

x++; //the value of x become $5+1=6$

Decrement:

int x=5;

**x-- ; //the value of x
become $5-1=4$**



Precedence of Operators

- Operator precedence is a crucial concept in programming that determines the order in which operators are evaluated in expressions.
- Like out of three operators like $*$, $/$ and $+$, which is executed first?
- Precedence in decreasing order: $(++, --)$, $(*, /, \%)$, $(+, -)$, $(<<, >>)$, $(<, <=, >, >=)$, $(==, !=)$, $\&$, $|$, $\&\&$, $||$.
- Operators in bracket are having same priority



CONTD..

Example:

1) Int a=5, b=3, c=7;
 $a+b*c/a - c\%b = ?$ 8

2) x=5, y=10, z=15;
 $x+y*z-y/x = ?$ 153

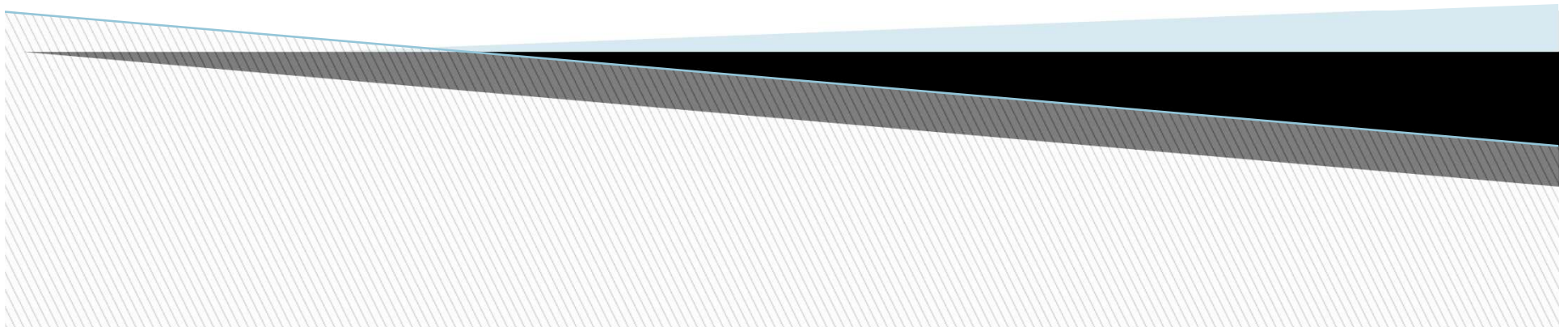


CONTD..

3) $X=5, Y=2, Z=8;$
 $X>Y \&\& Y<Z = ?$ 1(True)

4) $a=5, b=2, c=8;$
 $++a * b-- + c = ?$ 20

5) $a=3, b=4, c=5;$
 $++a || b-- \&\& c++ = ?$ 1 (True)





Simple C program

```
void main()  
{  
    printf("Hello World");  
}
```

// here main is a library function where execution starts

//printf is a function to display on screen (here, Hello World)

// semi colon is used to terminate a statement



Simple C program

```
void main()
{
    int num1=5, num2 =2;
    int add;
    add = num1 + num2;
    printf("Addition is %d",add);
}
//num1, num2 and add are integer variables
```




Storage Class

- ❑ In the C programming language, a storage class defines the scope (visibility) and lifetime of a variable or a function within a program. C provides several storage classes that determine how variables and functions are stored in memory and how they can be accessed by different parts of a program.



Types of Storage class

- Automatic (auto) -- local variable & default garbage value
- Register -- local variable stored in register and default garbage value
- Static -- local variable & with 0 default value
- External (extern) -- global variable & 0 default value



Module 2



Contents of Module 2

- Arithmetic expressions and precedence, Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching, Iteration and loops
- Arrays: Arrays (1-D, 2-D), Character arrays and Strings



Objectives of Module 2

- ❑ Learn arithmetic expressions and precedence.
- ❑ Identify need and use of conditional branching.
- ❑ Learn loops, their types and syntax.
- ❑ Use arrays for storing homogenous data.



Decision Control

- Sometimes situation arises where some decision should be taken based on certain condition
- Ex- if age is ≥ 18 , then we can vote
- In C, the following are used for condition checking
 - if
 - if-else
 - switch-case



if statement

□ Syntax:

```
if (condition)
{
    statements; // if condition is true
}
```

□ Ex-

```
if (marks >= 40)
{
    printf("You are passed");
}
```



Program using if

```
int marks = 50;  
if (marks >= 40)  
printf("You have passed");
```

//Here, output will be ' You have passed' because
condition is satisfied



if–else statement

- Syntax:

```
if (condition)
{
    statements; // if
                condition is true
}
else
{
    statements; // if
                condition is false
}
```

- Example:

```
if (marks >= 40)
{
    printf(" You are passed
");
}
else
{
    printf(" You are failed
");
}
```



Program using if-else

```
int marks = 50;  
if (marks >= 40)  
    printf("You have passed");  
else  
    printf("You have failed");
```



Switch Case

- A switch statement tests the value of a variable and compares it with multiple cases
- Once the case match is found, a block of statements associated with that particular case is executed
- The value provided by the user is compared with all the cases inside the switch block until the match is found
- It provides a more efficient and concise way to handle multiple conditional branches compared to using multiple if statements.



Switch Case

- If a case match is not found, then the default statement is executed

- Syntax :

```
switch (n)
```

```
{
```

```
case 1: // code to be executed if n = 1
```

```
break;
```

```
case 2: // code to be executed if n = 2
```

```
break;
```

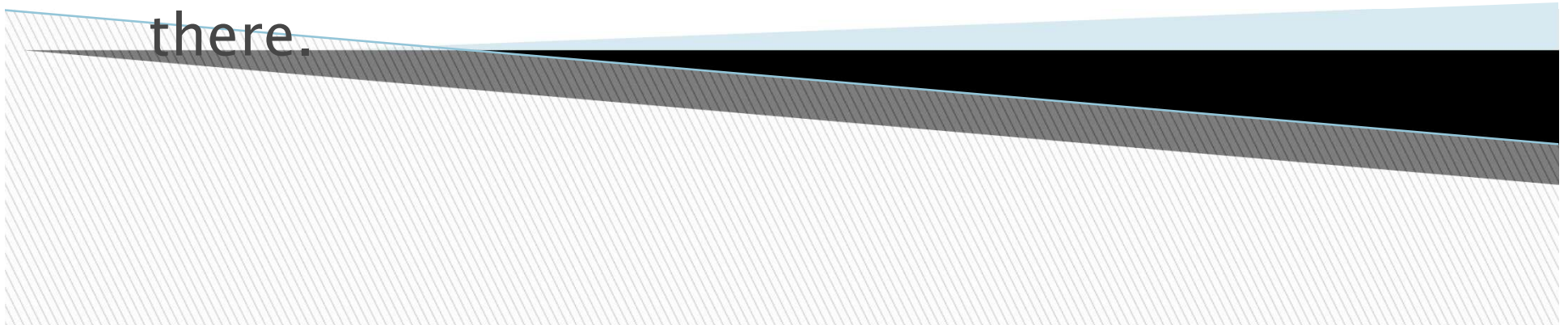
```
default: // code to be executed if n  
         doesn't match any cases
```

```
}
```



Break

The break statement is used primarily within loop and switch statements. Its purpose is to control the flow of execution by "breaking out" of a loop. Break Statement is a loop control statement that is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there.





Points to be noted in Switch - Case

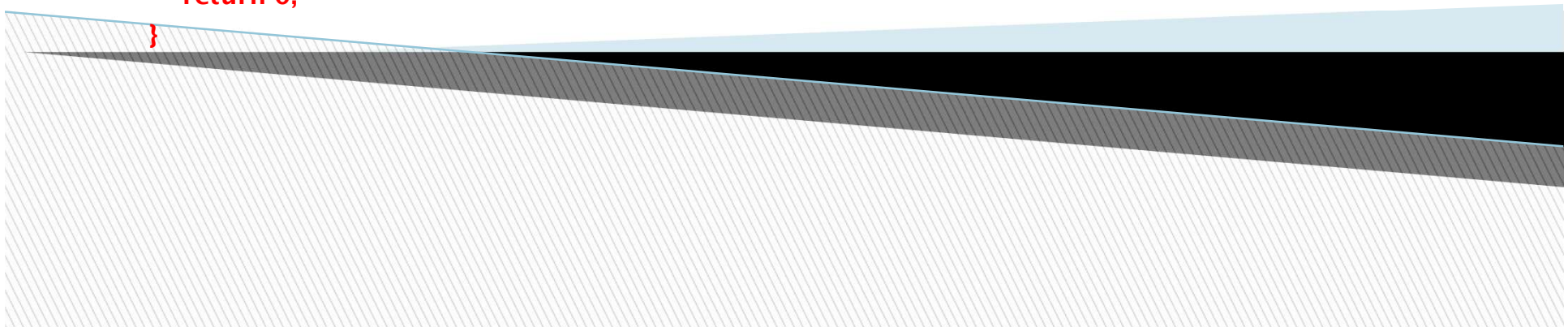
- Duplicate case values are not allowed
- The default statement is optional
 - Even if the switch case statement do not have a default statement, it would run without any problem

Program Using Switch Case



```
#include <stdio.h>
int main() {
int choice;
printf("Enter a number (1-2): ");
scanf("%d", &choice);
switch (choice) {
case 1:
printf("You selected option 1.\n");
break;
case 2:
printf("You selected option 2.\n");
break;
default:
printf("Invalid choice. Please enter a number between 1 and 2.\n");
break;
}

return 0;
}
```





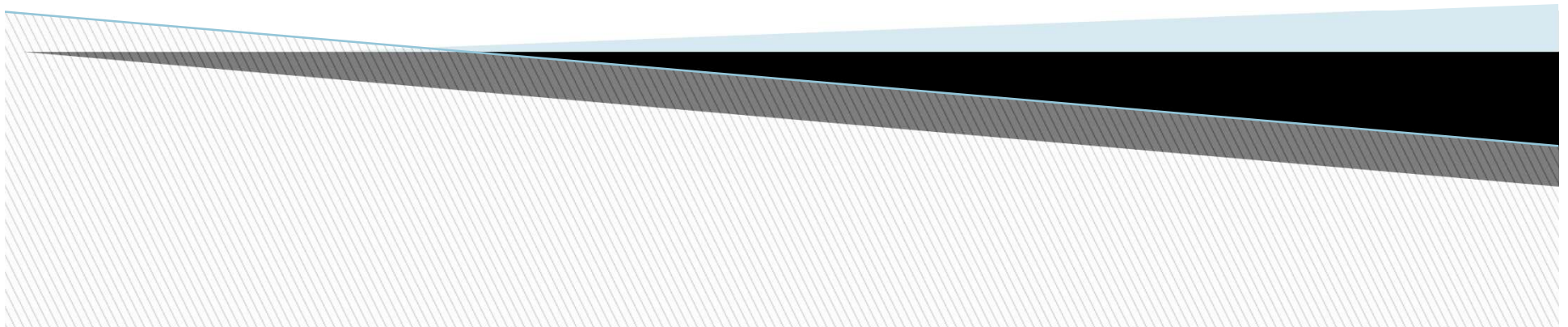
Output

Enter a number (1-2): 2

You selected option 2.

Enter a number (1-2): 4

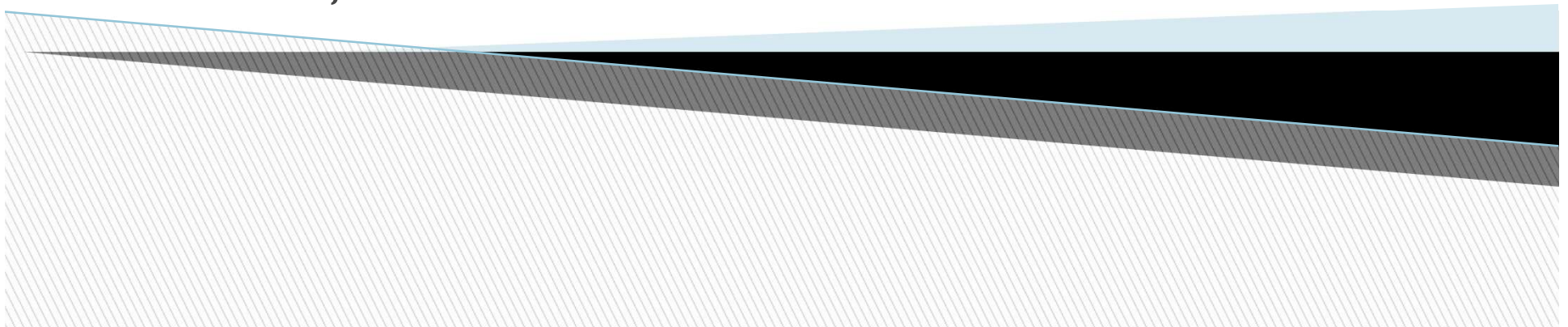
Invalid choice. Please enter a number between 1 and 2.





Conditional Branching

Conditional branching in C refers to the ability to execute different code blocks or statements based on certain conditions or expressions. It allows you to control the flow of your program by making decisions at runtime. Conditional branching is typically achieved using conditional statements, such as if, else etc.





Loop

Generally, statements are executed one after other, but (say) if same task needed for certain numbers of times?

- Loops in C is used to execute the block of code several times according to the condition given in the loop.
- Using loop, set of statements are executed repeatedly (iteration)
- Iteration - number of times the body of loop (statements defined in loop) is executed



Types of Loop

- There are three types:
 - while loop
 - for loop
 - do-while loop



While Loop

- A *while* loop in C program repeatedly execute statements defined in the body of loop as long as given condition is true

- Syntax:

```
while (condition test)
```

```
{
```

```
    //statement to be executed repeatedly
```

```
    //increment (++) Or decrement (--)
```

```
}
```



Program using while loop

```
#include <stdio.h>
int main() {
    int count = 1;
    while (count <= 5) {
        printf("This is iteration %d\n", count);
        count++;
    }
    return 0;
}
```



Output

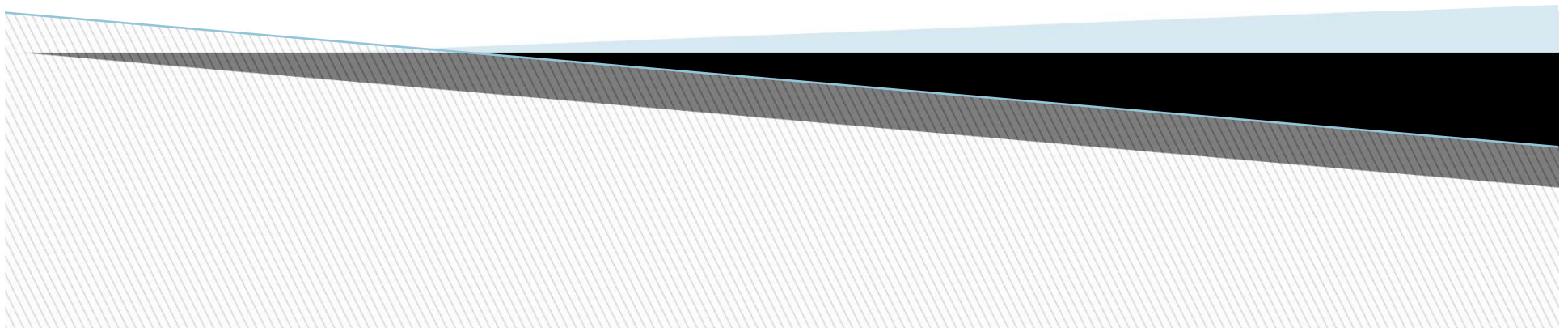
This is iteration 1

This is iteration 2

This is iteration 3

This is iteration 4

This is iteration 5





For Loop

- This is used when you know the number of iterations in advance and typically consists of three parts: initialization, condition, and increment (or decrement).
- Syntax:
for (initialization; condition; modification)
{
 body of loop
}
- here, modification mean increment or decrement



do-while Loop

- There is a minor difference between the working of while and do-while loops
- This difference lie in the place where the the condition is tested
- The *while* loop test the condition **before** execution of any of the statement within the while loop whereas, the do-while test the condition **after** having executed the statement within the loop

Syntax



```
do  
{  
statements;  
}while (condition);
```

Program using do-while loop



TABLE OF 1

```
#include<stdio.h>
```

```
int main(){
```

```
int i=1;
```

```
do{
```

```
printf("%d \n",i);
```

```
i++;
```

```
}while(i<=10);
```

```
return 0;
```

```
}
```

Output



1

2

3

4

5

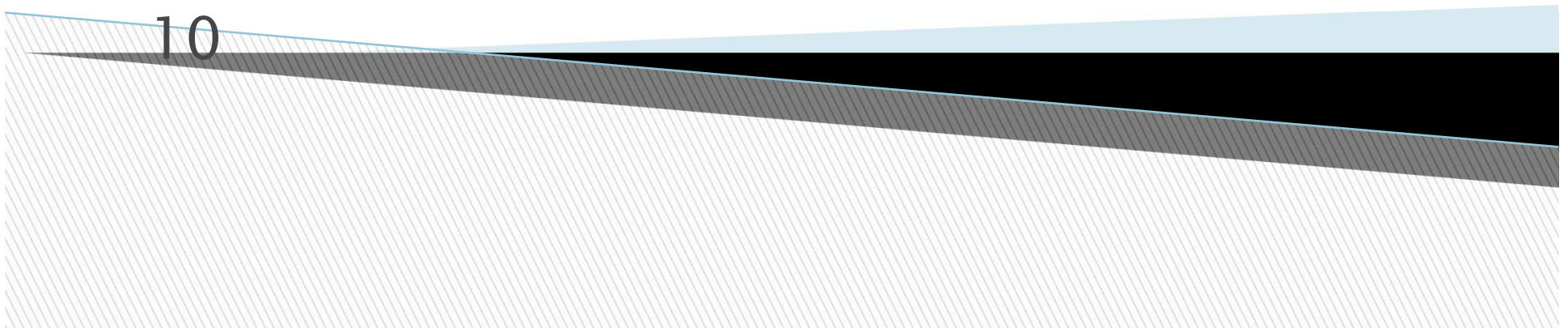
6

7

8

9

10





Comparison Between For Loop, While Loop & Do-While Loop

For loop	While loop	Do while loop
Syntax: For(initialization; condition;updating), { . Statements; }	Syntax: While(condition), { . Statements; . }	Syntax: Do { . Statements; } While(condition);
It is known as entry controlled loop	It is known as entry controlled loop.	It is known as exit controlled loop.
If the condition is not true first time than control will never enter in a loop	If the condition is not true first time than control will never enter in a loop.	Even if the condition is not true for the first time the control will enter in a loop.
There is no semicolon; after the condition in the syntax of the for loop.	There is no semicolon; after the condition in the syntax of the while loop.	There is semicolon; after the condition in the syntax of the do while loop.



Array

- Collection of elements of similar (homogenous) datatypes (int/float/char etc)
- Syntax: data type array_name [size];

Example:

Declaration: To declare an array in C, you specify the data type of its elements and the number of elements it can hold. For example:

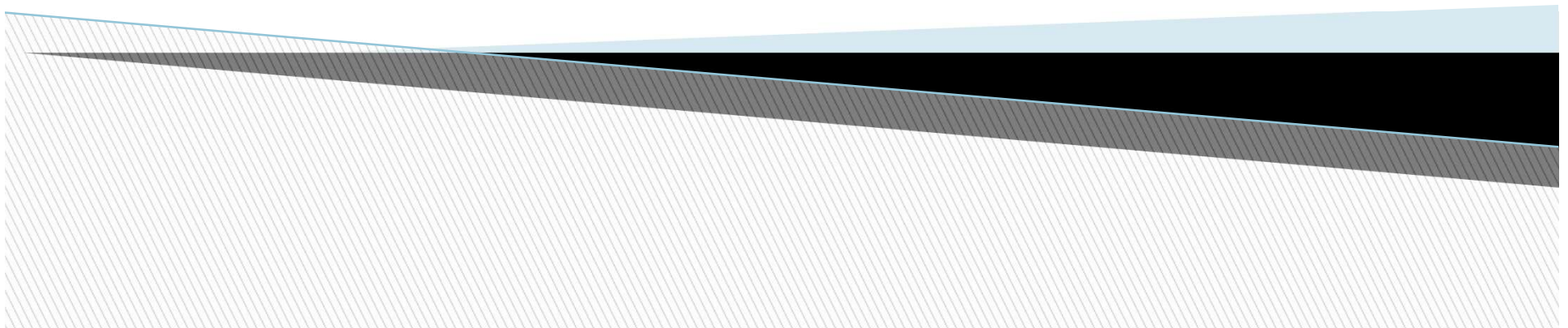
```
int myArray[5]; // Declares an integer array with 5 elements
```



CONTD..

Initialization: One can initialize an array at the time of declaration.

```
int myArray[5] = {1, 2, 3, 4, 5}; // Initializing  
at declaration
```





CONTD..

Accessing Elements: Array elements are accessed using an **index**, starting from 0 for the first element.

```
int x = myArray[2]; // Accesses the third element (with index 2)
```

- **Index:** An index in an array is a numeric value used to identify and access individual elements within the array.



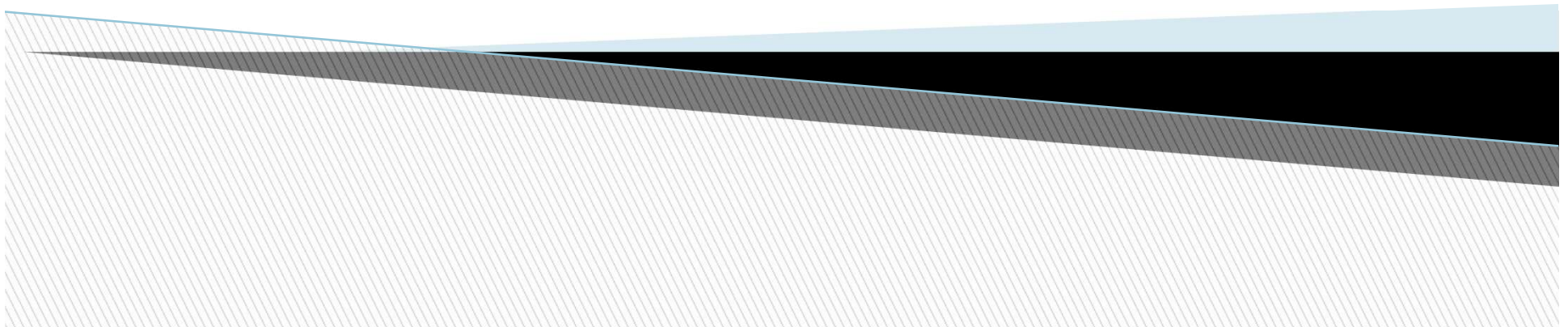
CONTD..

Zero-Based Indexing: C uses 0 to (n-1) as the array bounds.

Example: `int arr[5] = {6,5,1,9,2};` // 'arr' goes from 0
→ 4

Note:

Bounds Checking: Array indexes should be within the valid range of the array. Accessing an element with an index outside this range can lead to errors, such as "index out of bounds" or undefined behavior. It's essential to ensure that your index values are valid.



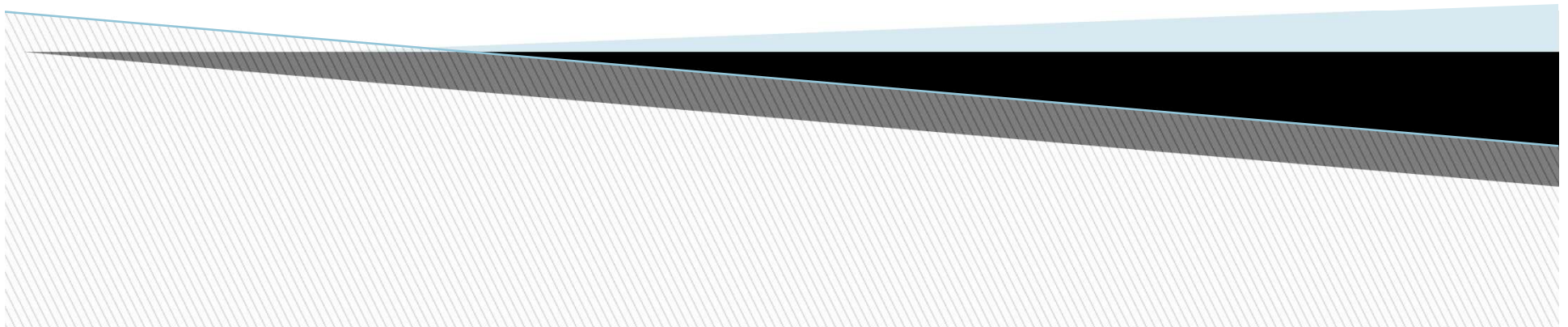


CONTD..

Size: You can find the number of elements in an array by dividing the total size of the array by the size of one element.

```
int size = sizeof(myArray) / sizeof(myArray[0]);
```

sizeof(myArray): This part of the expression calculates the total size (in bytes) of the entire array.





CONTD..

sizeof(myArray[0]): This part calculates the size (in bytes) of a single element within the array "myArray". (myArray[0]) represents the first element of the array.

sizeof(myArray) / sizeof(myArray[0]):
divides the total size of the array by the size of a single element. This division yields the number of elements in the array.



Array declaration and representation

- C uses 0 to (n-1) as the array bounds
 - `int arr[5] = {6,5,1,9,2};` **// 'arr' goes from 0 → 4**

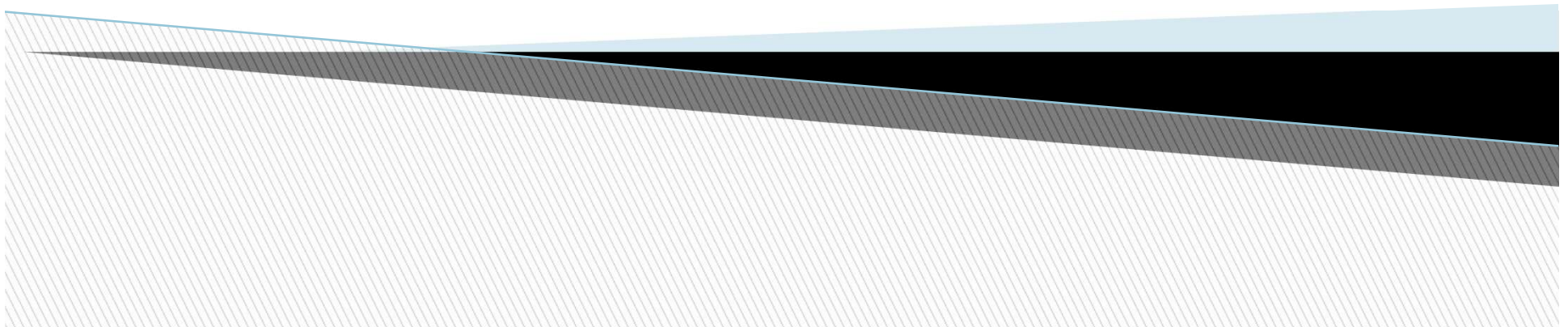
6	5	1	9	2
<code>arr[0]</code>	<code>arr[1]</code>	<code>arr[2]</code>	<code>arr[3]</code>	<code>arr[4]</code>



EXAMPLE

```
int myNumbers[] = {25, 50, 75, 100};  
printf("%d", myNumbers[0]);
```

// Outputs 25





Change an Array Element

In C, one can change an element of an array by directly assigning a new value to the desired element using the array's index.

Example: `int myNumbers[4] = {25, 50, 75, 100};
myNumbers[0] = 33;
printf("%d", myNumbers[0]);`

// Now output is 33 instead of 25

Types of Array

- **One dimensional array:** The One-dimensional arrays, also known as 1-D arrays in C are those arrays that have only one dimension.
- **Example:**

1D Array

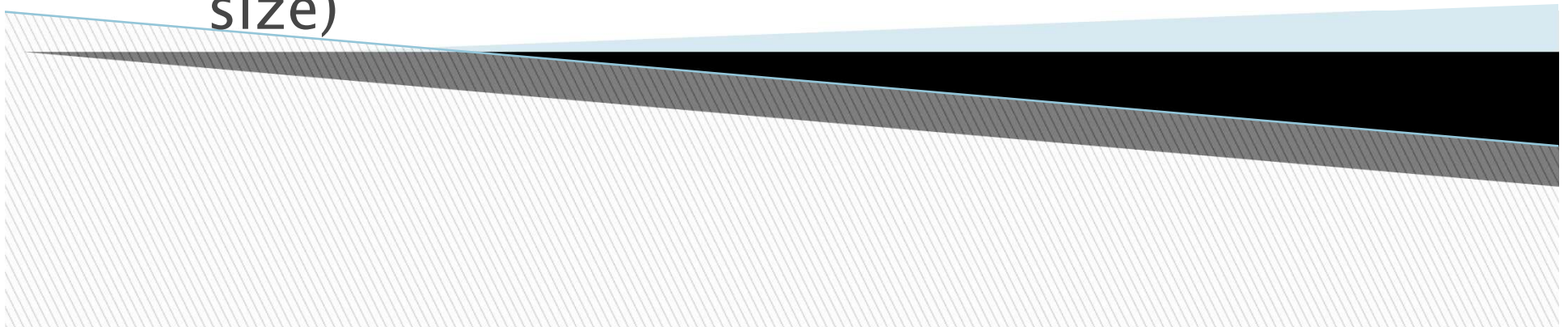
1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---



CONTD..

Multidimensional Array: Multi-dimensional Arrays in C are those arrays that have more than one dimension. Generally we study 2D array.

Syntax: data-type array_name[size1] [size2];
size1: Size of the first dimension.(row size)
size2: Size of the second dimension.(column size)





CONTD..

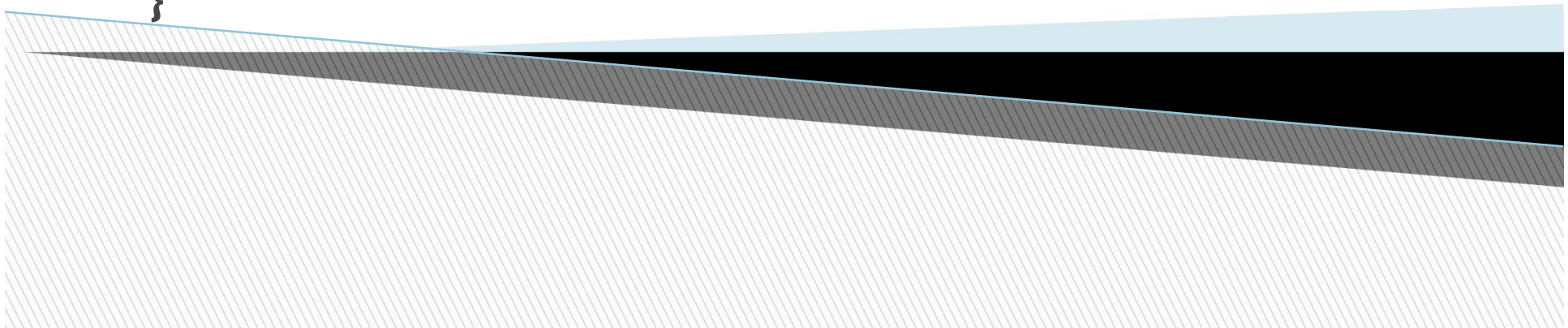
2D Array

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4



Program using 1-D Array

```
#include <stdio.h>
int main() {
    // Declaration and initialization of a 1D integer array
    int numbers[5] = {10, 20, 30, 40, 50};
    // Accessing and printing elements of the array
    printf("Elements of the array:\n");
    for (int i = 0; i < 5; i++) {
        printf("numbers[%d] = %d\n", i, numbers[i]);
    }
    return 0;
}
```





Output

Elements of the array:

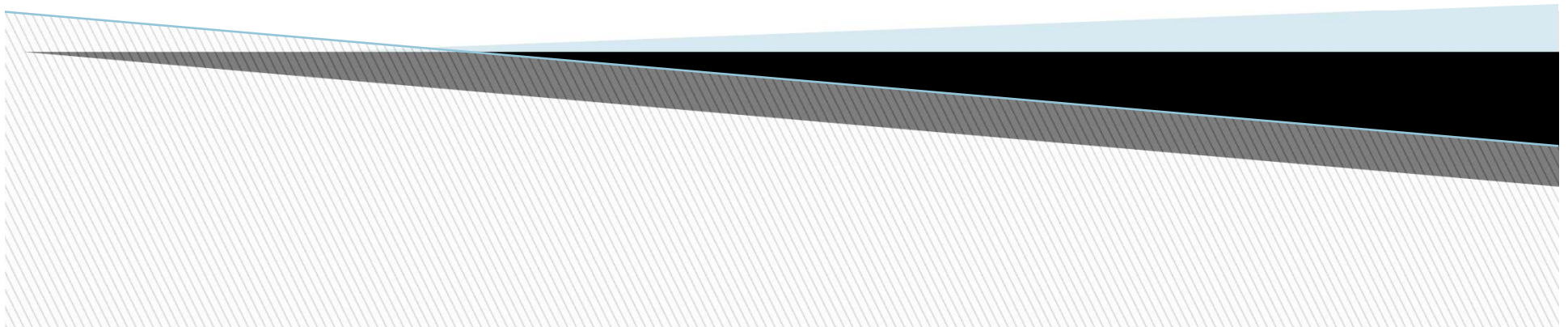
`numbers[0] = 10`

`numbers[1] = 20`

`numbers[2] = 30`

`numbers[3] = 40`

`numbers[4] = 50`



Program using 2-D Array

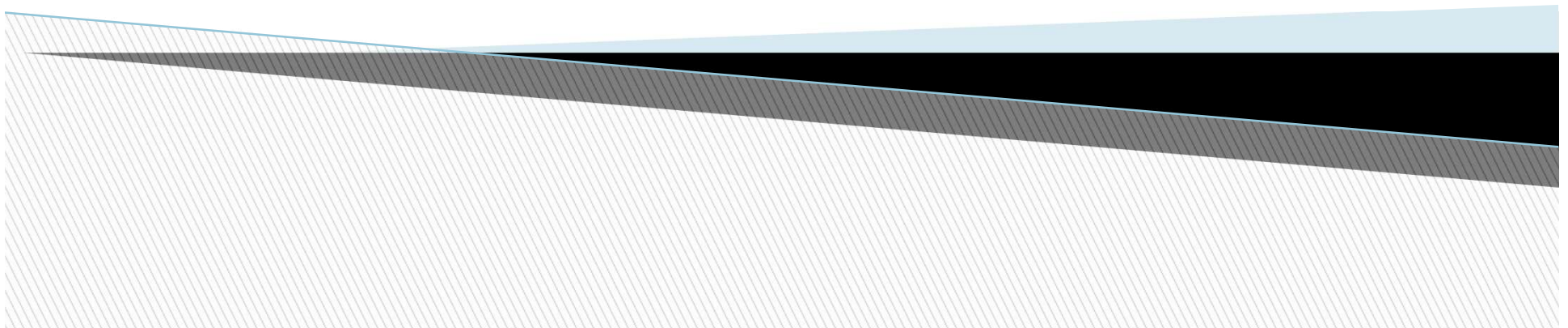


```
#include <stdio.h>
int main() {
    // Declare and initialize a 2D array
    int arr[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    // Print the 2D array using a loop
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```



Output

1 2 3
4 5 6
7 8 9





String

In C programming, a string is a sequence of characters stored as an array of characters.

Declaration

String Declaration: To declare a string in C, you can use the “char” data type followed by array name and square brackets to specify the size of the character array.

Example: `char myString[20];`

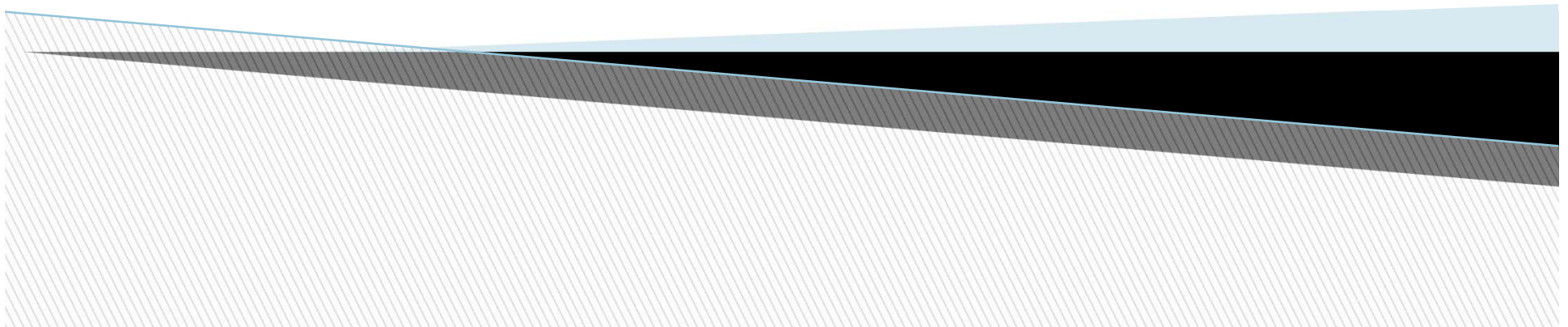
- A null character (`\0`) is used to denote the end of a string. This is placed at the end of the string.



CONTD..

String Initialization: You can initialize a string at the time of declaration.

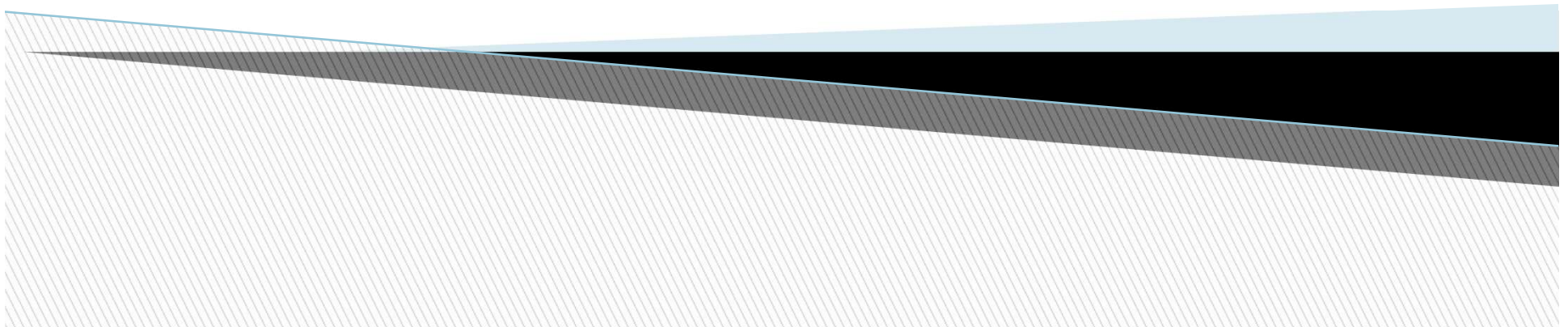
Example: `char greeting[] = "Hello, World!";` // **String initialized at declaration**





CONTD..

Size of a String: The size of a C string (i.e., the number of characters it can hold) is determined by the size of the character array plus one additional byte for the null character. For example, if you have a character array of size 10, it can hold a string of up to 9 characters plus the null character.

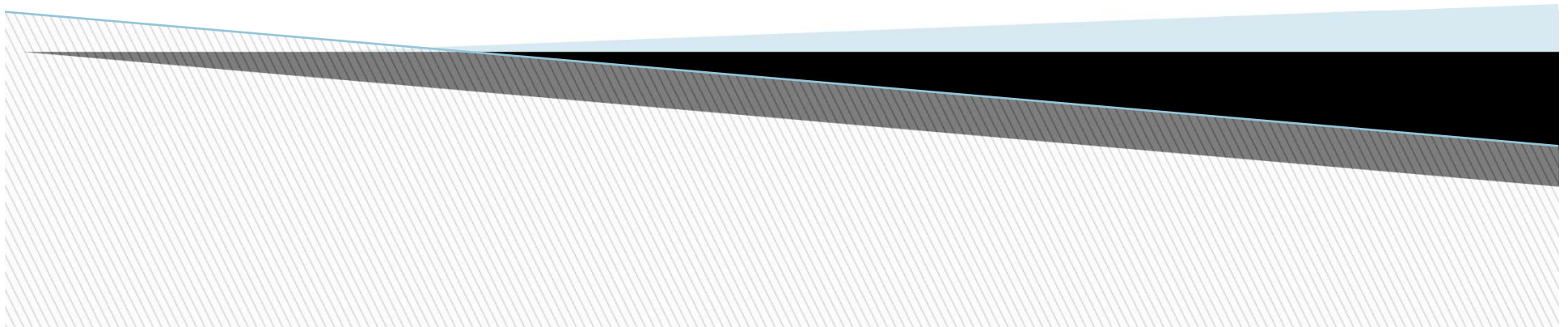




CONTD..

```
char greeting[] = "Hello, World!";  
sizeof(greeting)
```

Assuming that each character in the string occupies 1 byte (which is typically the case in most systems), the size of the array would be 14 bytes (13 characters + 1 null character).





CONTD..

- ASCII value of null character (`\0`) is 0
- `char arr [] = {'h','e','l','l','o'};`
 - In this case, size of arr is 5 because each element is explicitly assigned using single quotation mark. So, This initialization does not include a null terminator.
 - `char arr [] = "hello";`
 - In this case, size of arr is 6 because it is stored as a string in memory with a `\0` character.



Simple String program

```
#include <stdio.h>
int main() {
int i;
char arr[5] = {'a', 'p', 'p', 'l', 'e'};
for (i = 0; i < 5; i++) {
printf("Value at %d position is %c\n", i, arr[i]);
}
return 0;
}
// here i represents position and arr[i] represents
character at position
```



Output

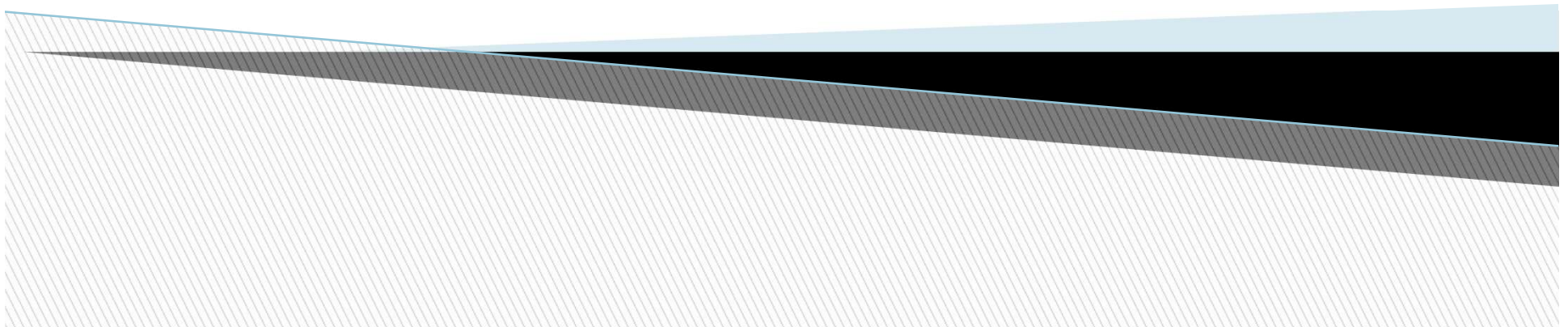
Value at 0 position is a

Value at 1 position is p

Value at 2 position is p

Value at 3 position is l

Value at 4 position is e





String Functions

- To manipulate strings effectively, Inbuilt functions are there for doing operations on string, such as `strlen()` - used to find the length of a string.
- These functions can be used by including standard library like `<string.h>`

Depending on your specific task, you may use these functions.



CONTD..

- ❑ **strlen():** Returns the length (number of characters) of a string.

Syntax: `strlen("Brainware")` returns 9.

- ❑ **strcpy():** used to copy the content from source string to target string.

Syntax: `strcpy(destination, source)`

- ❑ **strcat():** This function is used to concatenate (join) two strings.

Syntax: `strcat(string1,string2)` appends the contents of **string2** to the end of **string1**.

String program to find length

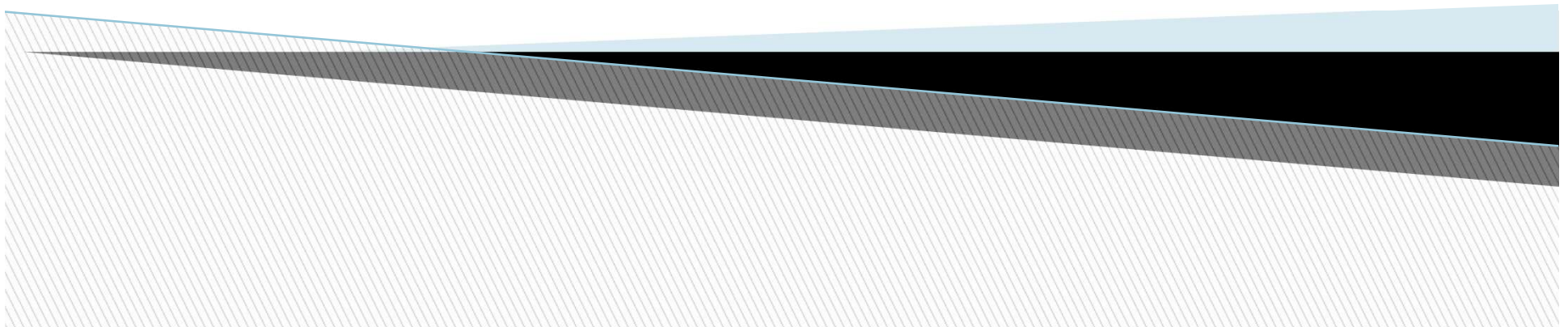


```
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
    printf("The length of the string is: %d\n",
length);
    return 0;
}
```



Output..

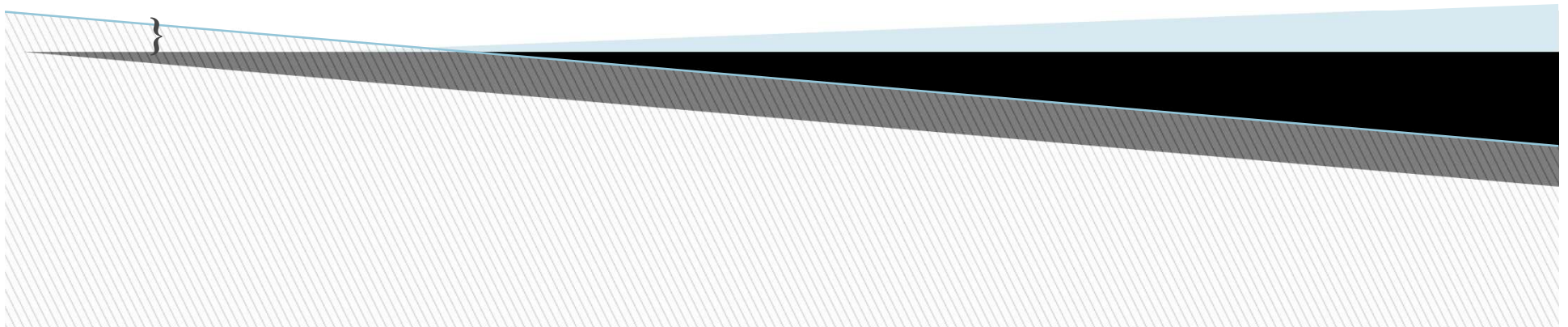
The length of the string is: 13





String program to copy the contents of a source string to a destination string:

```
#include <stdio.h>
#include <string.h>
int main() {
    char source[] = "Hello, World!";
    char destination[20]; // Make sure the target string has enough
                           // space
    strcpy(destination,source);
    printf("Source string: %s\n",source);
    printf("Destination string: %s\n",destination);
    return 0;
}
```

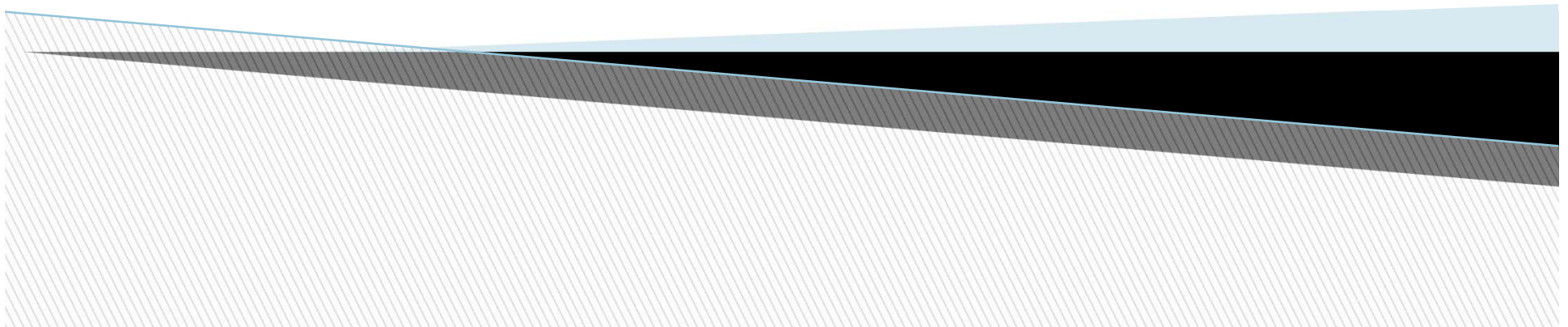




Output..

Source string: Hello, World!

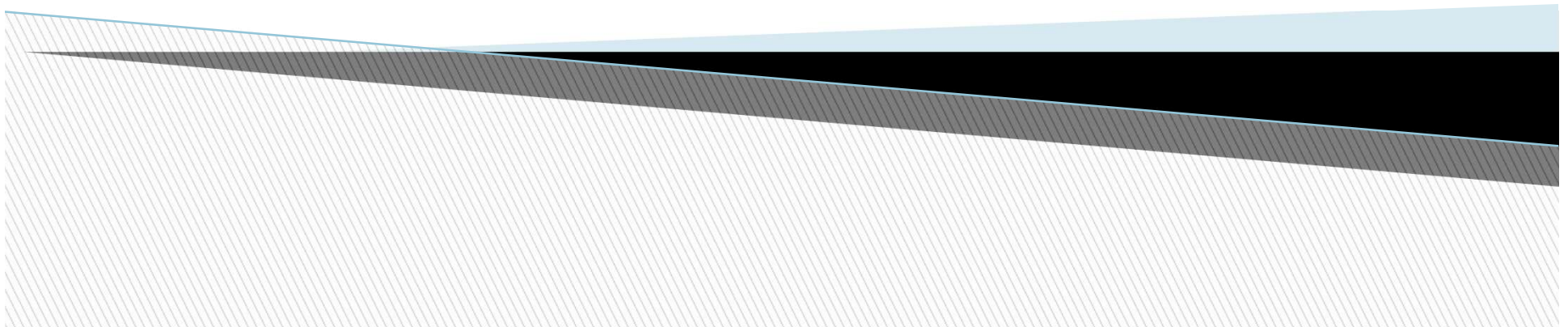
Destination string: Hello, World!





String program to concatenate (join) two strings

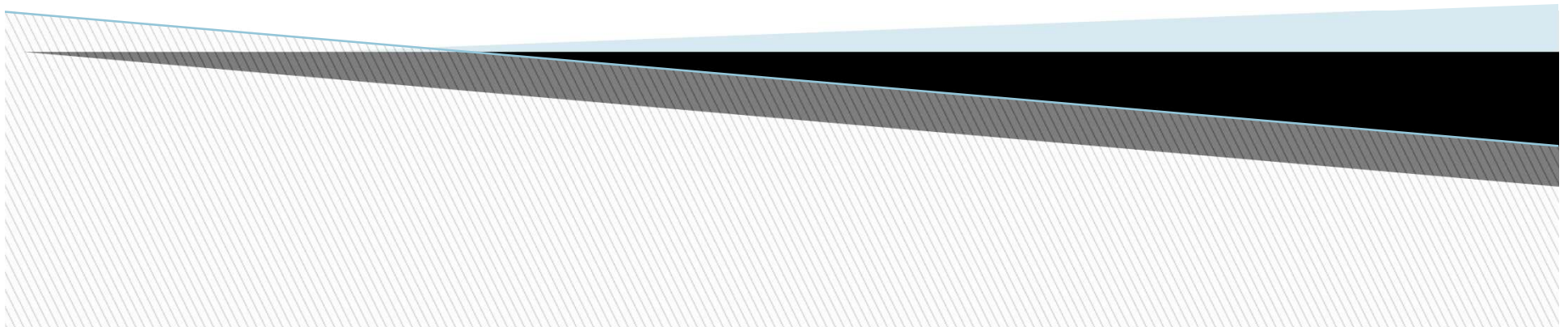
```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[50] = "Hello, ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```





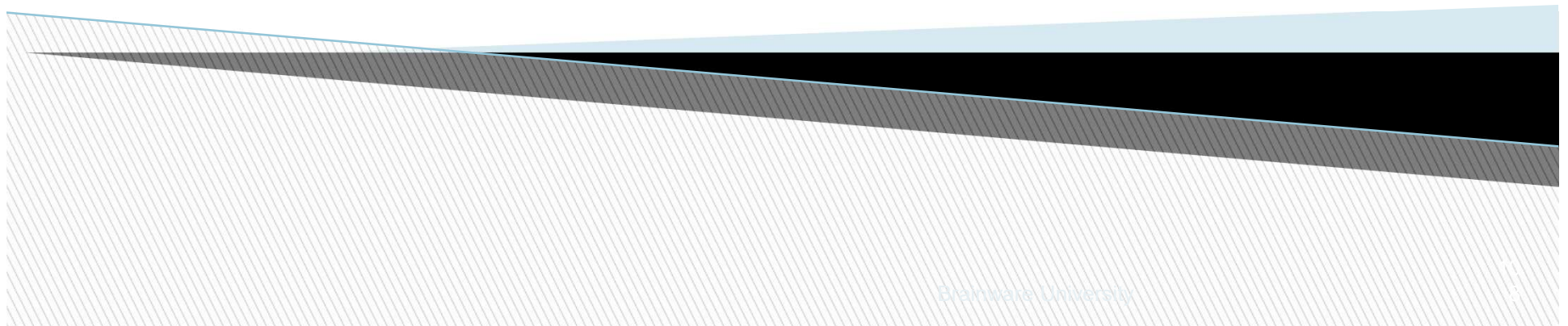
Output..

Concatenated string: Hello, World!





Module 3





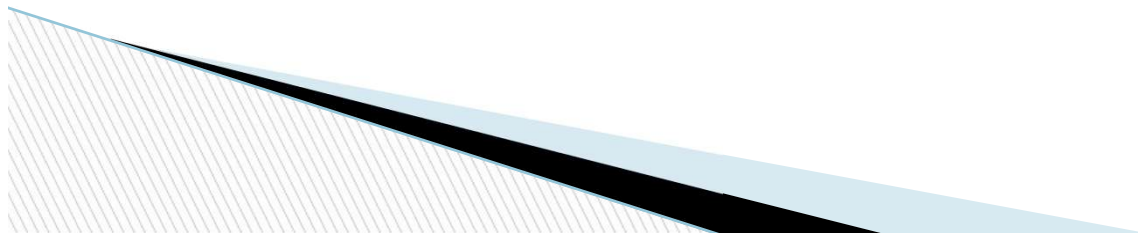
Contents of Module 3

- Function: Built in libraries, Parameter passing in functions, call by value, Passing arrays to functions: call by reference
- Recursion: Example programs, Finding Factorial, Fibonacci series, Ackerman function
- Sorting: Quick sort or Merge sort.



Objectives of Module 3

- ❑ Learn function and their types.
- ❑ Compare inbuilt and user defined functions.
- ❑ Use recursion technique to solve problems.
- ❑ Learn basic idea of sorting.





Function

- ❑ In C programming, a function is a self-contained block of code that performs a specific task or set of tasks. Functions allow you to break down a complex program into smaller, manageable pieces or modules. Each function can focus on a specific task or subtask, making the code easier to understand, test, and maintain. Every C program has at least one function, which is `main()`.
- ❑ Once you define a function, you can reuse it in multiple places within your program without duplicating code. This reduces redundancy and helps ensure consistency in your code.



Syntax

```
return_type function_name(arguments)
{
    statements;
}
```

(Here, return type can be void, int, float, char etc.)

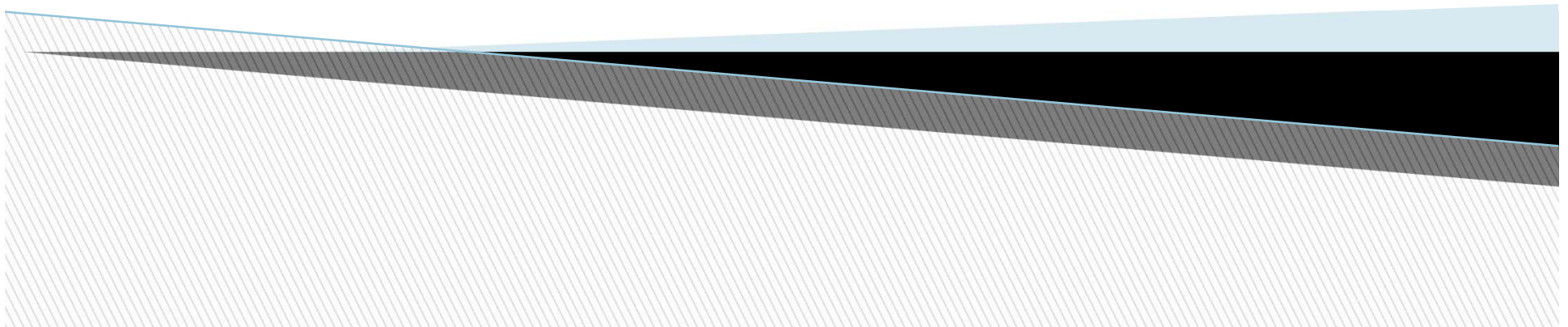
Arguments: In C programming, "arguments" refer to the values or expressions that we pass to a function when we call it. These arguments are used by the function to perform its task or calculations. Arguments provide a way to pass data into a function, allowing us to work with different data each time we call the same function.



CONTD..

Function Declaration: A function must be declared before it is used in the program. The declaration typically includes the function's return type, name, and a list of parameters (if any).

Example: `int add(int a, int b);` // Function declaration



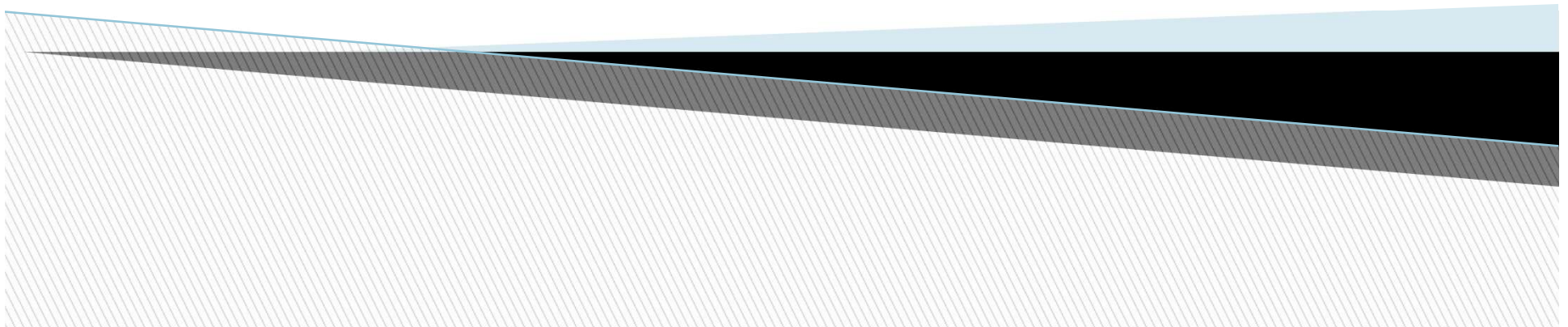


CONTD..

Function Definition: The actual implementation of a function is called its definition. It includes the function's return type, name, parameters, and the code block (function body) enclosed in curly braces.

Example:

```
int add(int a, int b) {  
    return a + b; // Function definition  
}
```



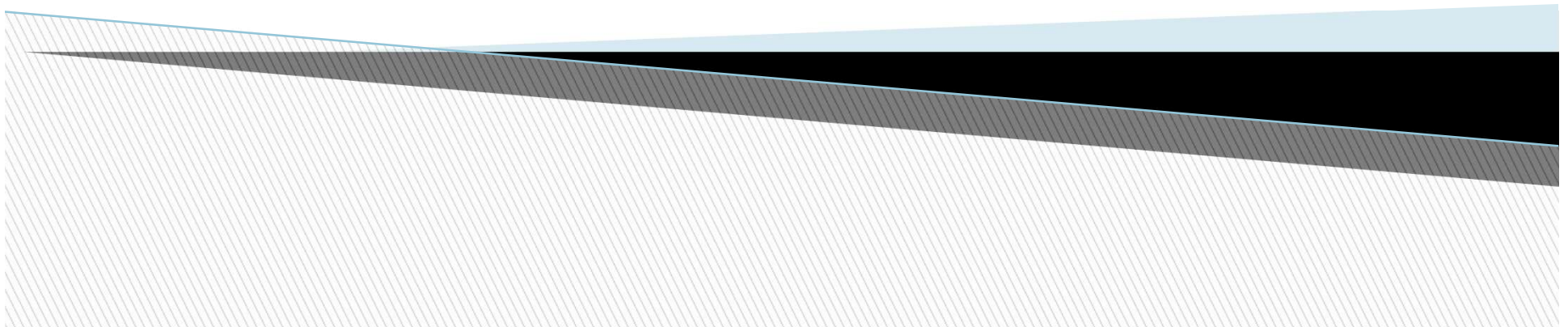


CONTD..

Function Call: To execute a function, you call it by its name.

Example:

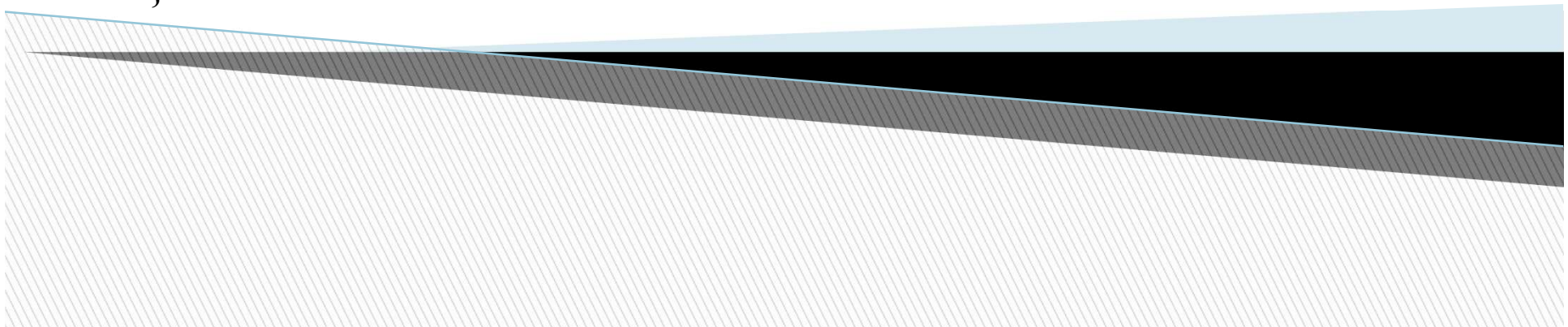
```
int result = add(5, 3); // Function call
```



Addition of two numbers using Function



```
#include <stdio.h>
int add(int a, int b);
int main() {
    int a, b, result;
    int add(int a, int b) {           // Function to add two numbers
        return a + b;
    }
    printf("Enter the two numbers: ");
    scanf("%d%d", &a,&b);
    result = add(a, b); // Call the add function and store the result
    printf("The sum of %d and %d is %d\n", a, b, result);
    return 0;
}
```

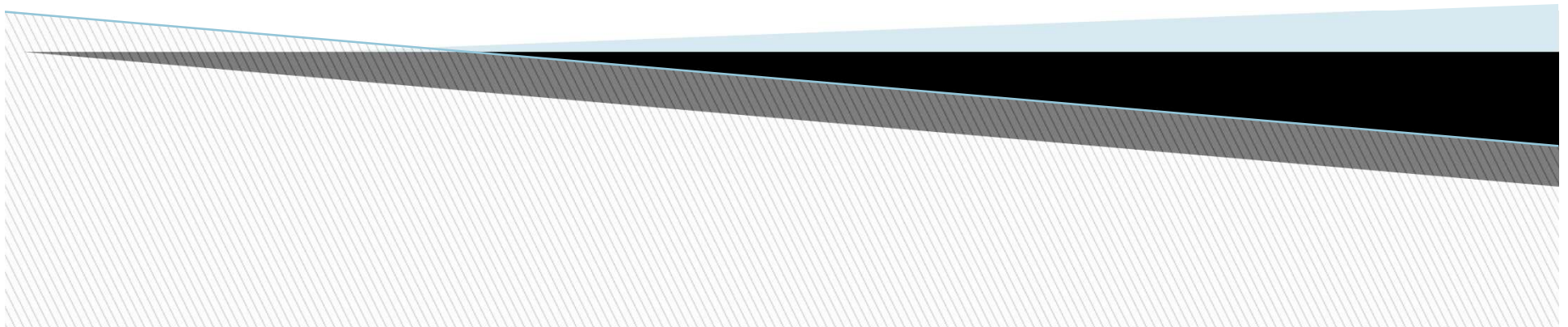




Output..

Enter the two numbers: 70 40

The sum of 70 and 40 is 110





Types of Function

Mainly there are two types of function-

- **Pre-defined/ Standard Library Functions:** These are built-in functions provided by the C Standard Library. They perform various tasks and are available for use without the need for explicit declaration or definition. Like- **printf()**, **scanf()** etc.
- **User-Defined Functions:** These are functions created by the user to perform specific tasks. They are defined by the user and can be called from other parts of the program. User-defined functions help modularize code and improve code readability.



Function Prototype

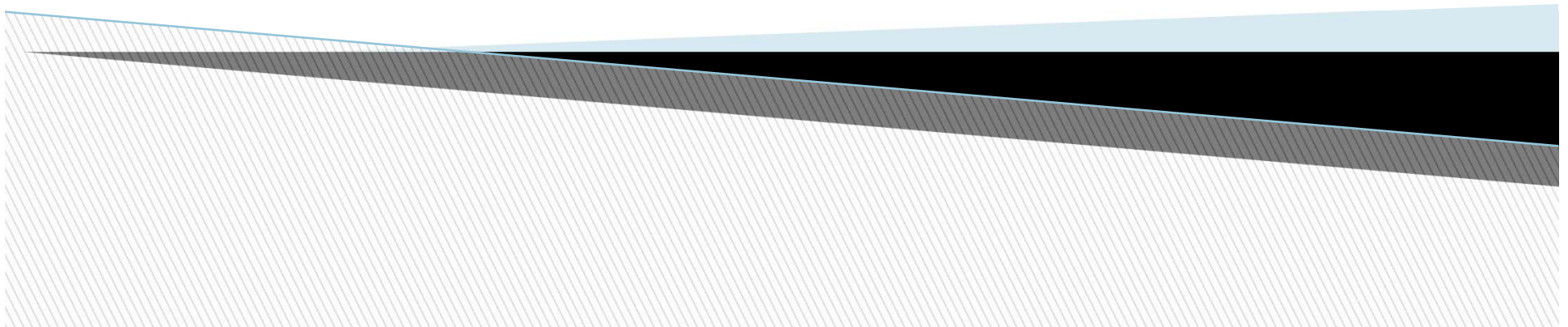
A function prototype is a special type of function declaration that also specifies the function's name, return type, and parameter types. In addition to this, a function prototype typically appears at the beginning of a program and it provides information to the compiler about functions that are defined elsewhere in the program. It is just the declaration of the function without any body.



CONTD..

Purpose:

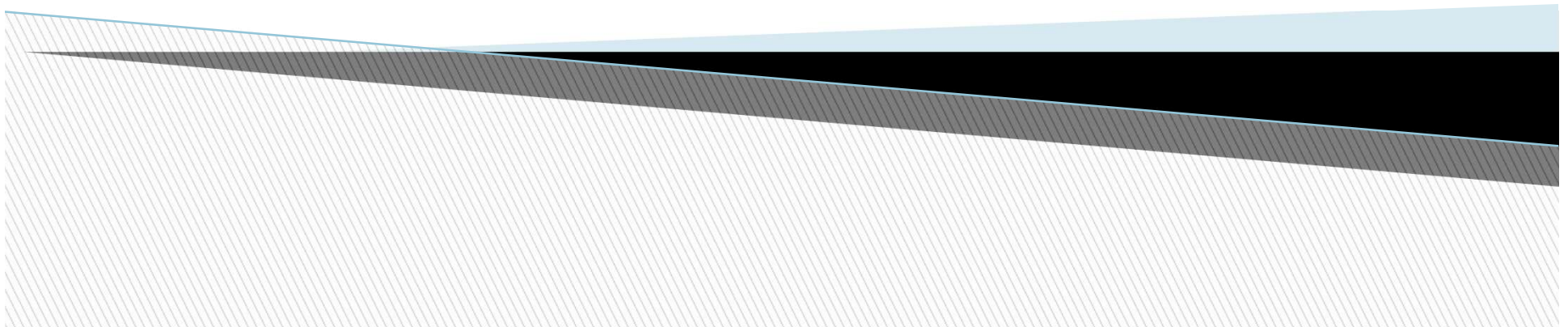
- **Forward Declaration:** Function prototypes are often used for forward declaration. They allow you to declare functions at the beginning of a program before the actual function definition. This is useful when you need to use a function before it's defined in the code.





CONTD..

- **Type Checking:** Function prototypes help in type checking. They ensure that you use the correct number and types of arguments when calling a function.





Example:

// Function prototype

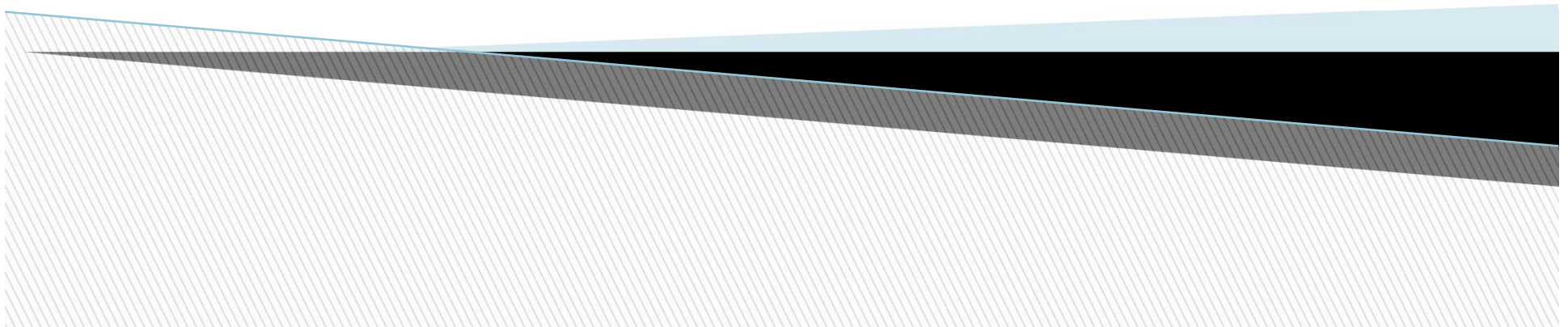
```
int add(int a, int b);
```

// Function definition

```
int add(int a, int b) {  
    return a + b;  
}
```

// Function call

```
int result = add(5, 3);
```





Recursion

- Recursion is a programming technique where a function **calls itself** to solve a problem. In C, a recursive function is a function that invokes itself either directly or indirectly to solve a problem.
- A termination condition is required to make it finite.
- Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting problems.

Factorial of a number using recursion



```
#include <stdio.h>
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
int main() {
    int num, result;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        int result = factorial(num);
        printf("Factorial of %d is %d\n", num, result);
    }
    return 0;
}
```



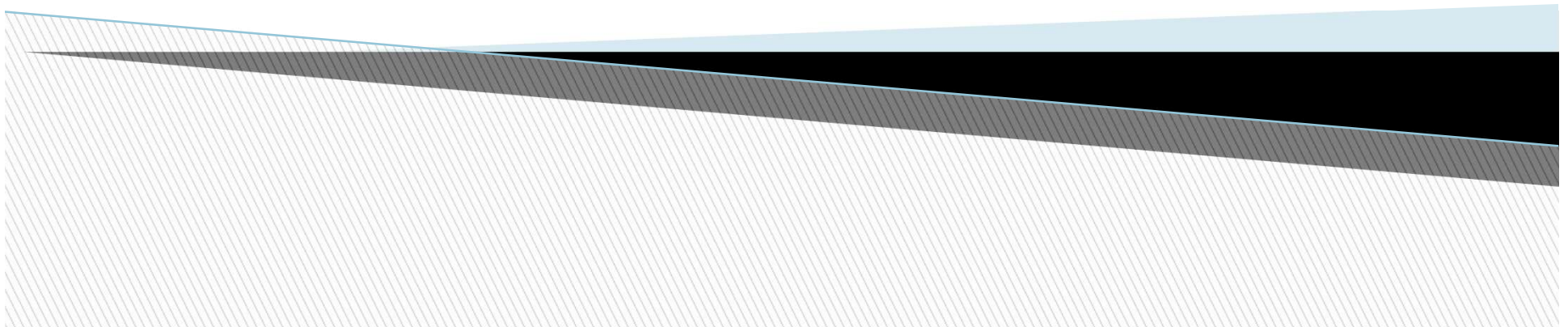

CONTD..

Enter a positive integer: 6

Factorial of 6 is 720

Enter a positive integer: -1

Factorial is not defined for negative numbers.



Fibonacci Series using recursion in C



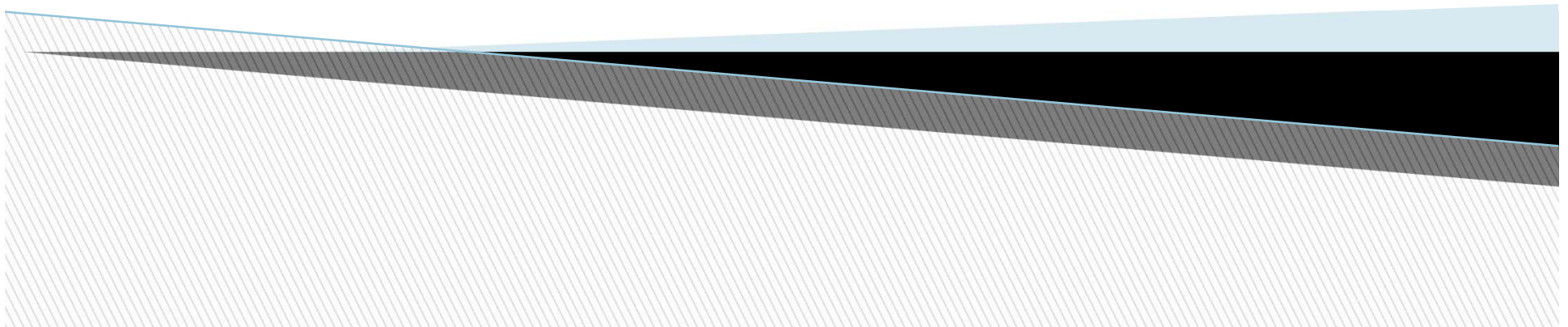
```
#include <stdio.h>
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
int main() {
    int num;
    printf("Enter the number of Fibonacci terms to generate: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Please enter a non-negative integer.\n");
    } else {
        printf("Fibonacci Series: ");
        for (int i = 0; i < num; i++) {
            printf("%d ", fibonacci(i));
        }
    }
    return 0;
}
```




Output

Enter the number of Fibonacci terms to generate: 10

Fibonacci Series: 0 1 1 2 3 5 8 13 21 34



Calculate the sum of the first n natural numbers using recursion



```
#include <stdio.h>
int sumOfNaturalNumbers(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + sumOfNaturalNumbers(n - 1);
    }
}
int main() {
    int num; Result
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    if (num < 0) {
        printf("Please enter a positive integer.\n");
    } else {
        Result = sumOfNaturalNumbers(num);
        printf("Sum of the first %d natural numbers is %d\n", num, Result);
    }
    return 0;
}
```



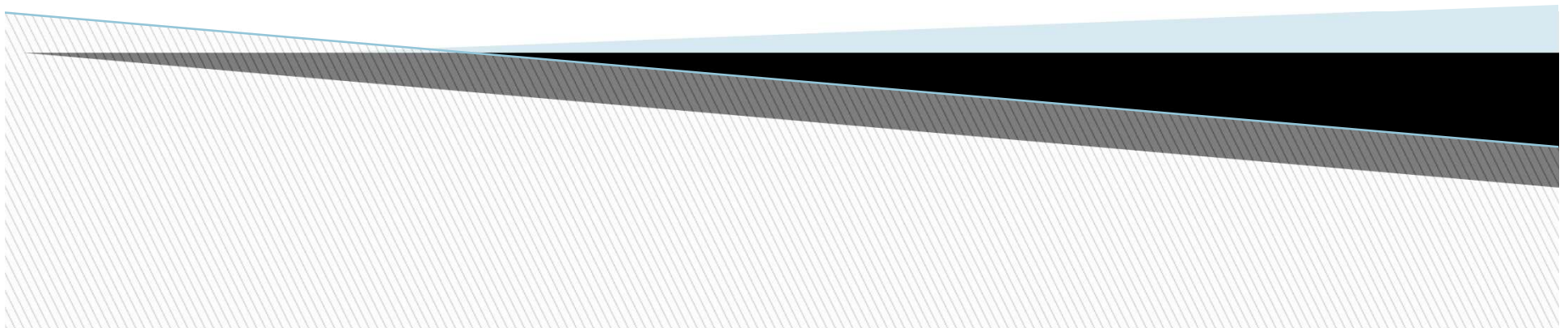
CONTD..

Enter a positive integer: 10

Sum of the first 10 natural numbers is 55

Enter a positive integer: -10

Please enter a positive integer.



Sum of Array Elements



```
#include <stdio.h>
int main() {
    int n, sum;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid input. Please enter a positive integer.\n");
        return 1; }
    int arr[n];
    printf("Enter %d elements of the array:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
        sum += arr[i];
    }
    printf("Sum of array elements: %d\n", sum);
    return 0;
}
```

OUTPUT



Enter the number of elements in the array: 3

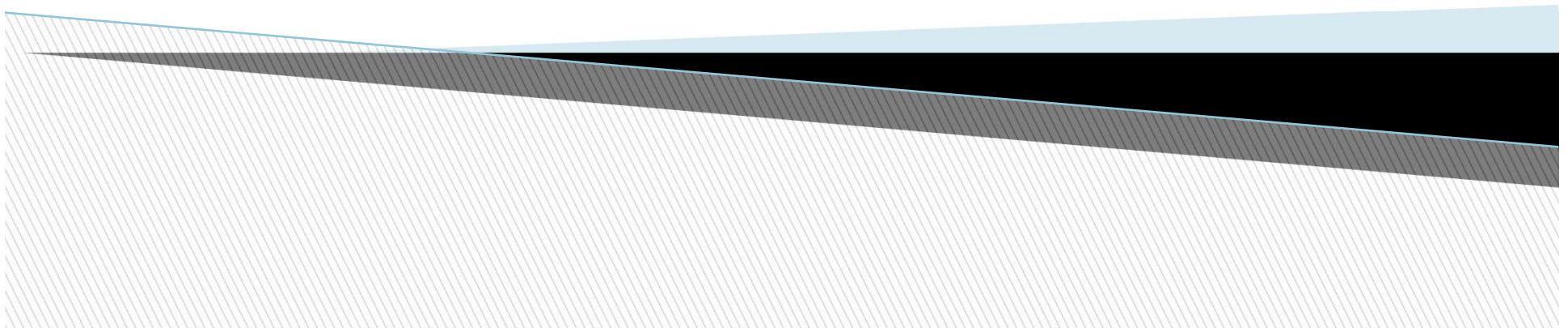
Enter 3 elements of the array:

Element 1: 2

Element 2: 4

Element 3: 0

Sum of array elements: 6



Write a C program to find out the average height of persons



```
#include <stdio.h>
int main() {
    int n;
    double sum;
    double average;
    printf("Enter the number of persons: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Invalid input. Number of persons must be greater than 0.\n");
        return 1;
    }
    double heights[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the height of person %d (in centimeters): ", i + 1);
        scanf("%lf", &heights[i]);
        sum += heights[i];
    }
    average = sum / n;
    printf("Average height of %d persons is: %.2lf centimeters\n", n, average);
    return 0;
}
```



Output

Enter the number of persons: 5

Enter the height of person 1 (in centimeters): 100

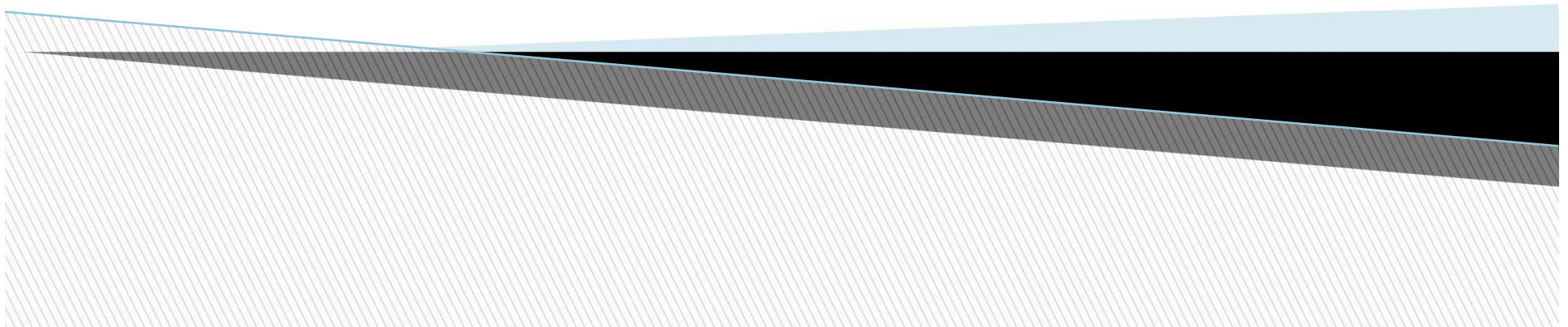
Enter the height of person 2 (in centimeters): 200

Enter the height of person 3 (in centimeters): 300

Enter the height of person 4 (in centimeters): 400

Enter the height of person 5 (in centimeters): 500

Average height of 5 persons is: 300.00 centimeters



A C program to get the largest element of an array using the function



```
#include <stdio.h>
int findLargest(int arr[], int size) {
    int largest = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }
    return largest;
}
int main() {
    int n;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    int largest = findLargest(arr, n);
    printf("The largest element in the array is: %d\n", largest);
    return 0;
}
```



Output

Enter the size of the array: 4

Enter 4 elements:

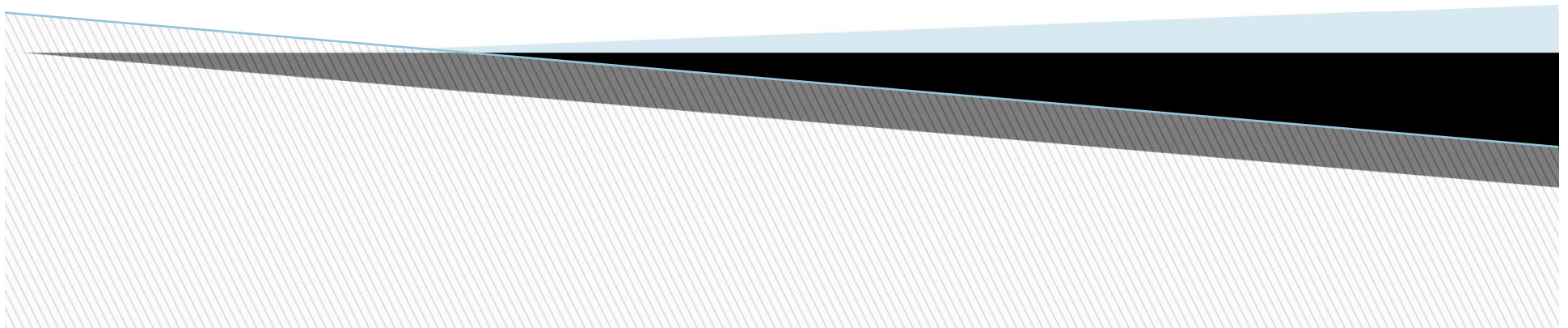
1

2

3

4

The largest element in the array is: 4





Sorting

Sorting is a fundamental operation in computer science and is used to arrange elements in a specific order.

Example:

```
int A[] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 }
```

The Array sorted in ascending order will be given as:

```
int A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }
```

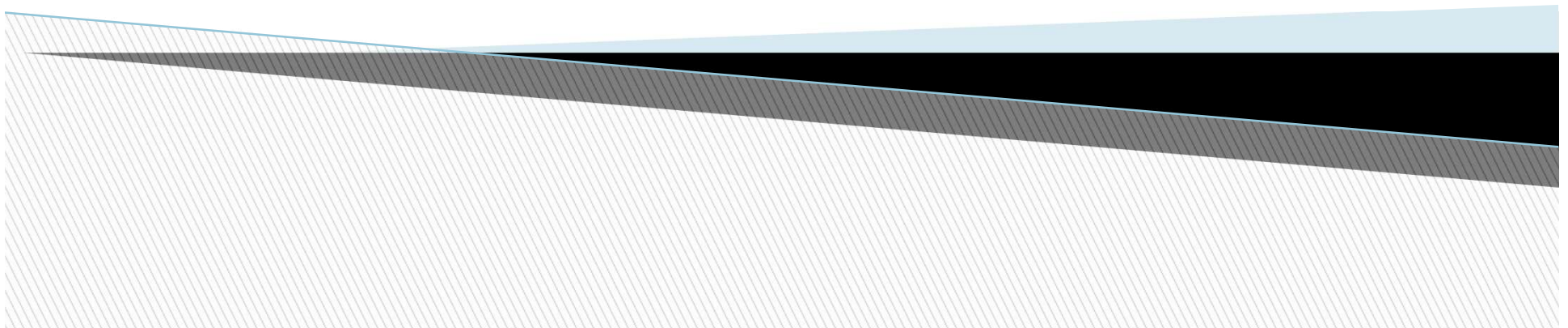
Types:

- Bubble Sort
- Merge Sort
- Quick Sort

Bubble Sort



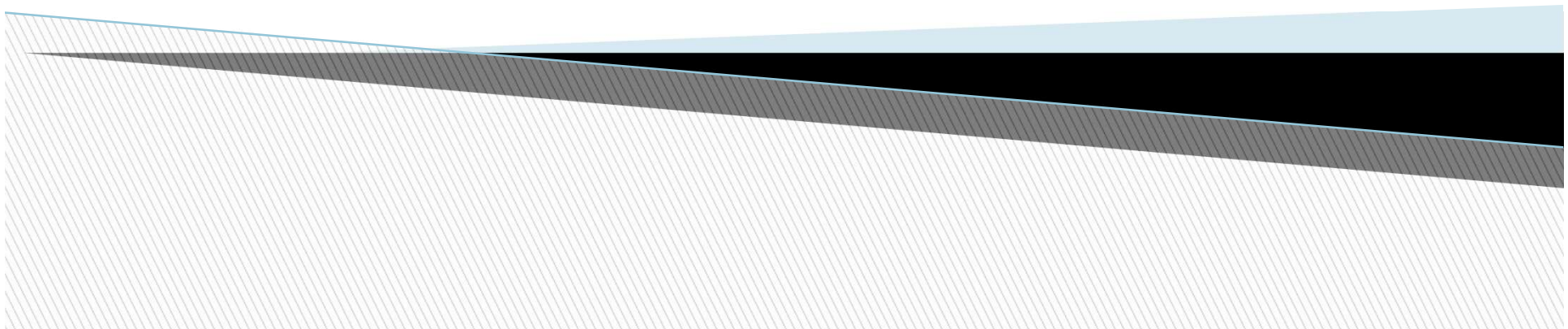
Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets.



Algorithm:



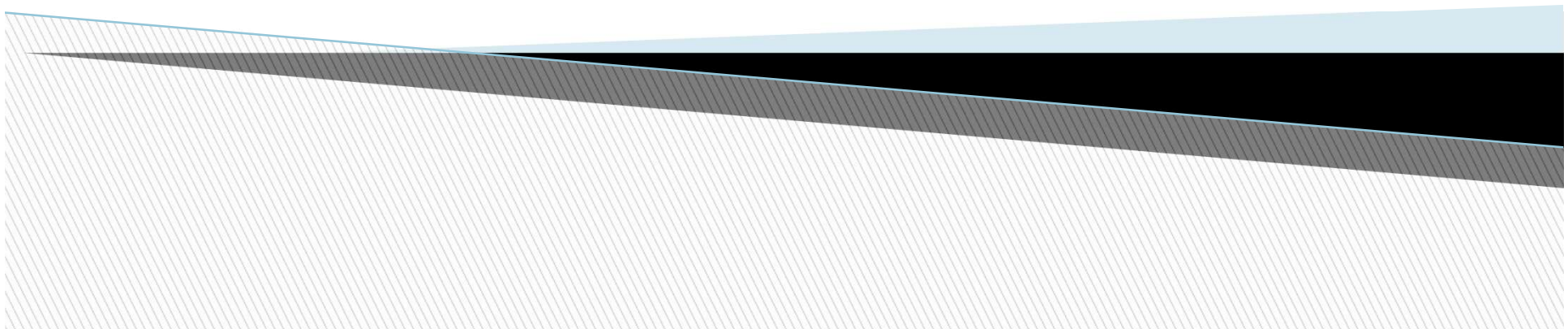
- Start with an unsorted list of elements.
- Begin a pass through the list by comparing the first two elements (the 0th and 1st elements).
- If the first element is greater than the second element, swap them.
- Move to the next pair of elements (the 1st and 2nd elements) and repeat the comparison and swap if necessary.
- Continue this process, comparing and swapping adjacent elements as you move through the list. After the first pass, the largest element will have "bubbled up" to the end of the list.
- Repeat steps 2-5 for the entire list, excluding the last element (which is already in its correct position after the first pass).



CONTD..



- After the first pass, the largest element is in its correct position at the end of the list. On the second pass, focus on the remaining unsorted portion of the list (excluding the last element), and repeat steps 2-5 to move the second-largest element to its correct position.
- Continue this process for all remaining elements, one at a time, until the entire list is sorted.
- The algorithm terminates when no more swaps are needed during a pass, which indicates that the list is fully sorted.



Working of Bubble Sort Algorithm



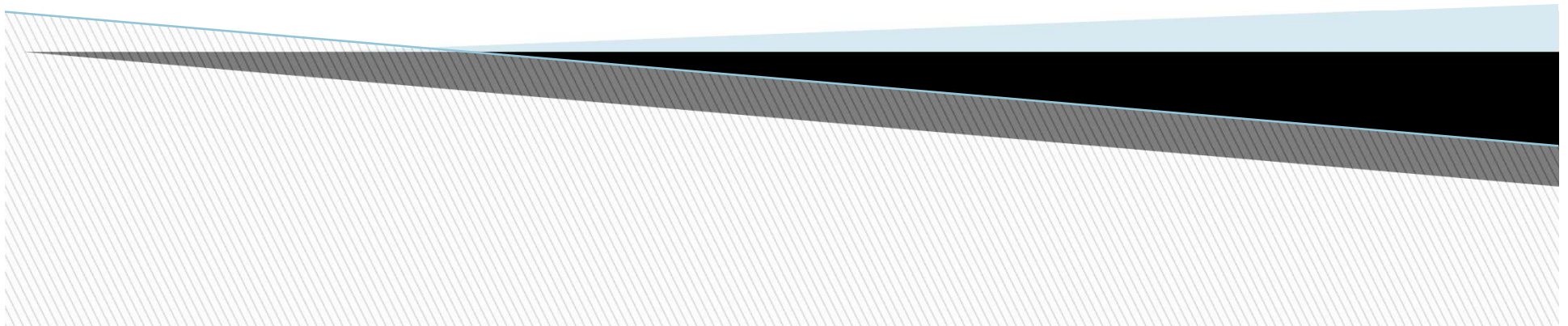
Let's take an unsorted array.

13	32	26	35	10
----	----	----	----	----

First Pass

Sorting will start from the initial two elements. Let compare them to check which one is greater.

13	32	26	35	10
----	----	----	----	----



CONTD..

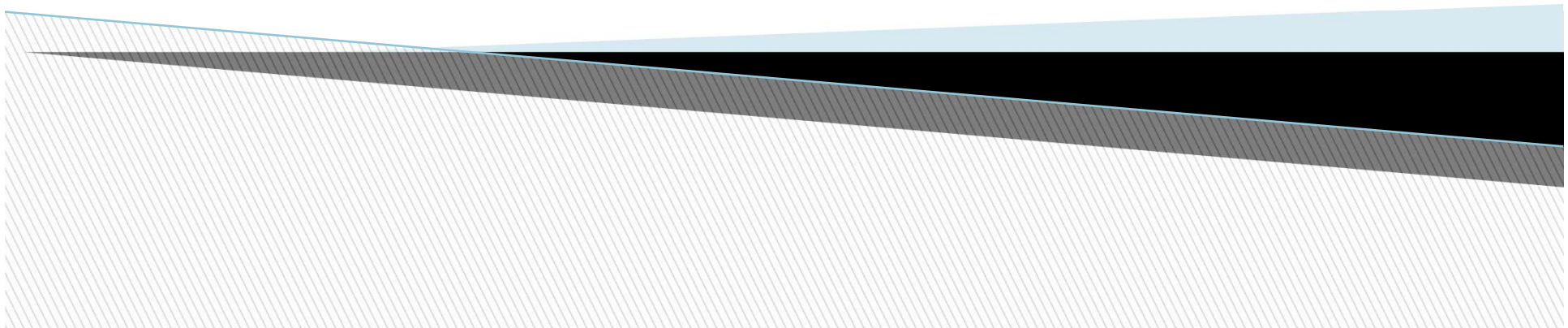


Here, 32 is greater than 13 ($32 > 13$), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10
----	----	----	----	----

Here, 26 is smaller than 32. So, swapping is required. After swapping new array will look like -

13	26	32	35	10
----	----	----	----	----





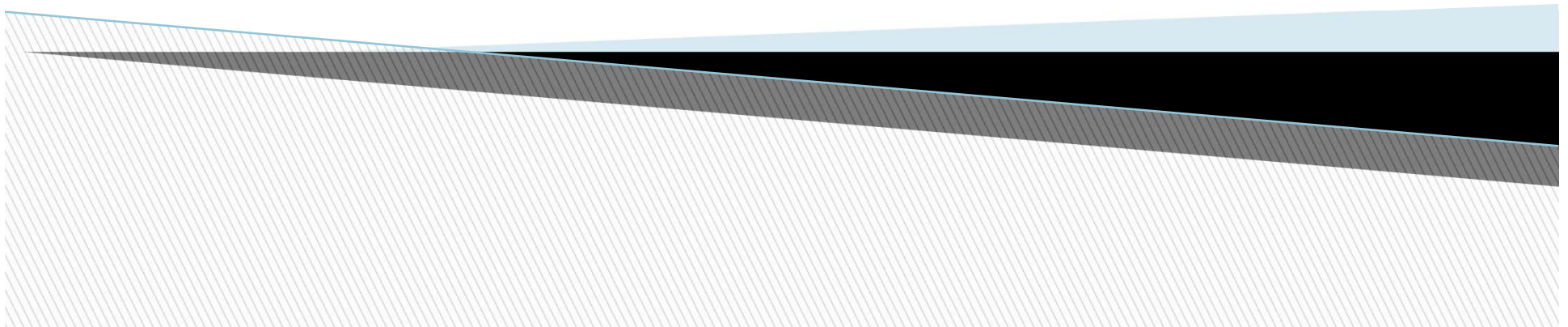
CONTD..

Now, compare 32 and 35.

13	26	32	35	10
----	----	----	----	----

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



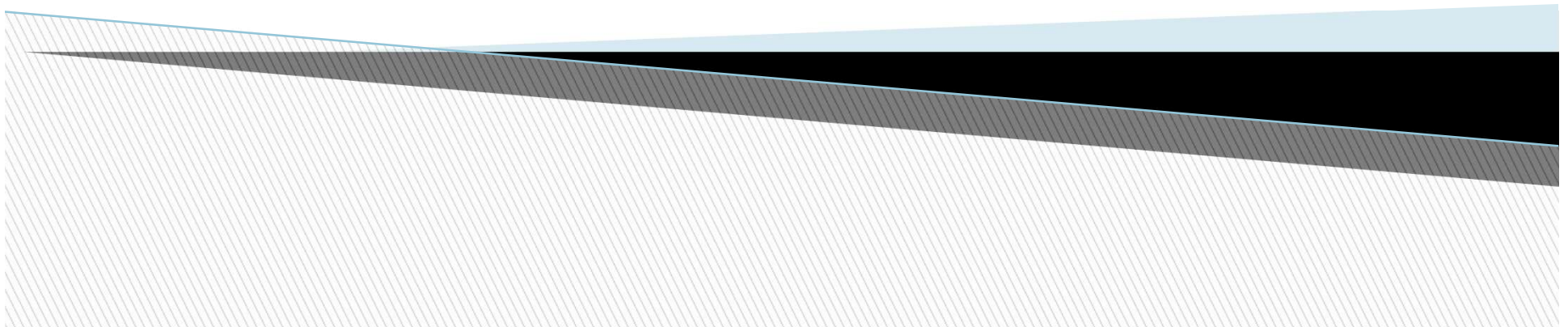


CONTD

13	26	32	35	10
----	----	----	----	----

Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

13	26	32	10	35
----	----	----	----	----





CONTD...

The same process will be followed for second iteration.

13	26	32	10	35
----	----	----	----	----

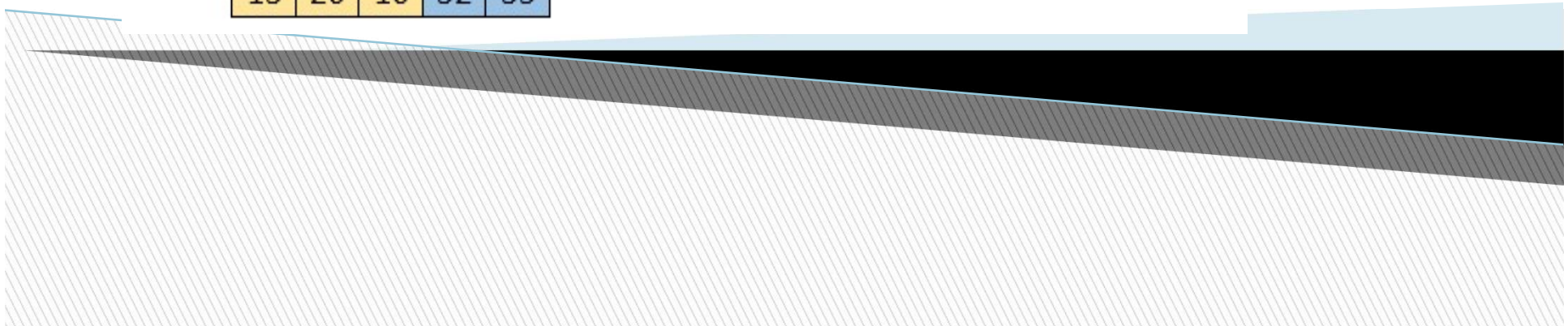
13	26	32	10	35
----	----	----	----	----

13	26	32	10	35
----	----	----	----	----

Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

13	26	10	32	35
----	----	----	----	----

13	26	10	32	35
----	----	----	----	----





CONTD..

Now, move to the third iteration.

Third Pass:

The same process will be followed for third iteration.

13	26	10	32	35
----	----	----	----	----

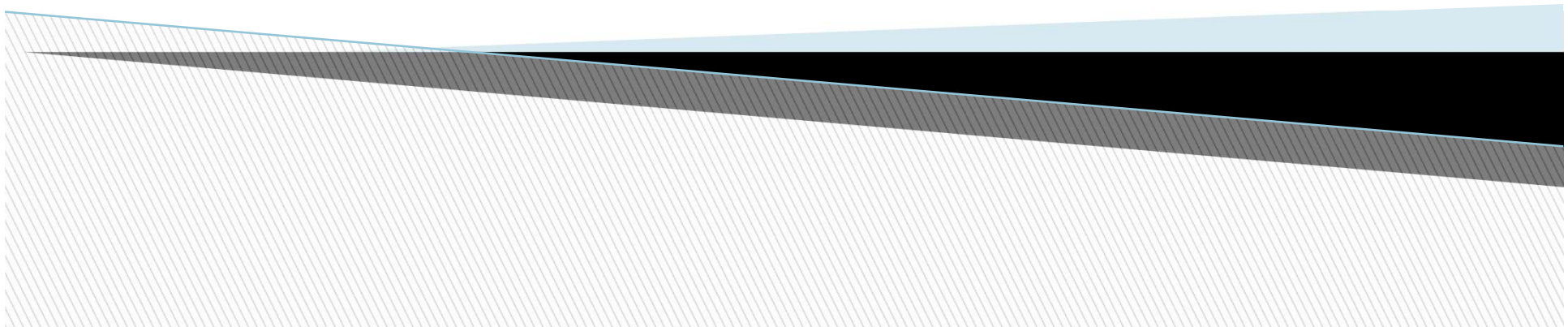
13	26	10	32	35
----	----	----	----	----

Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -

13	10	26	32	35
----	----	----	----	----

13	10	26	32	35
----	----	----	----	----

13	10	26	32	35
----	----	----	----	----



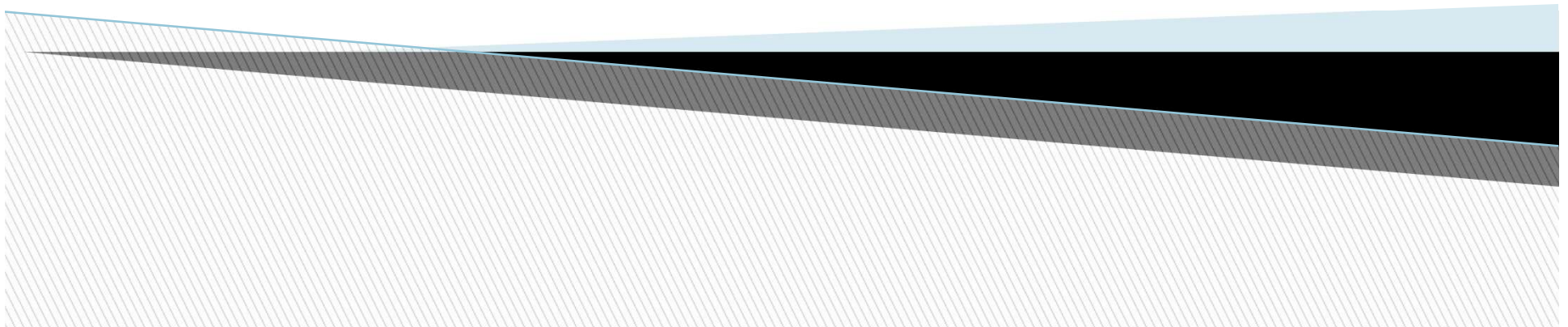


CONTD..

Now, move to the fourth iteration.

10	13	26	32	35
----	----	----	----	----

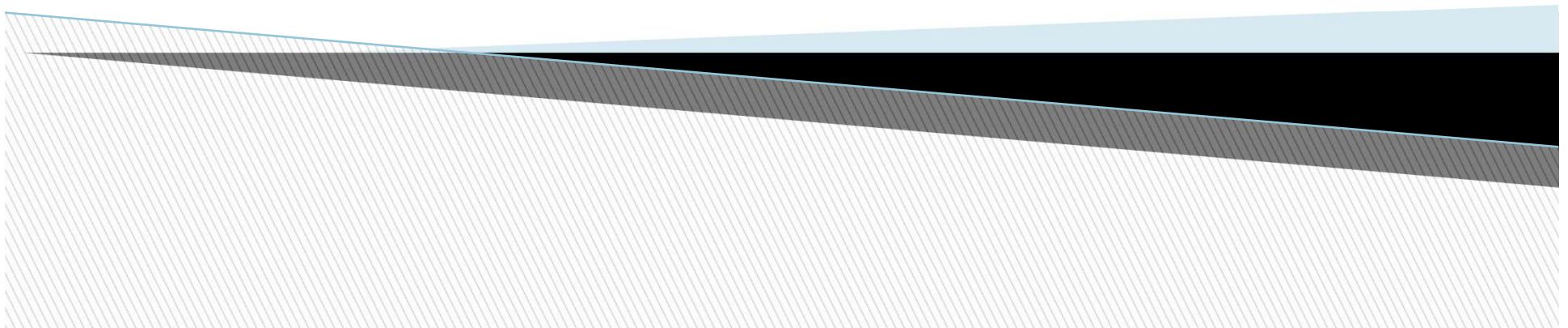
Hence, there is no swapping required, so the array is completely sorted.





Advantages

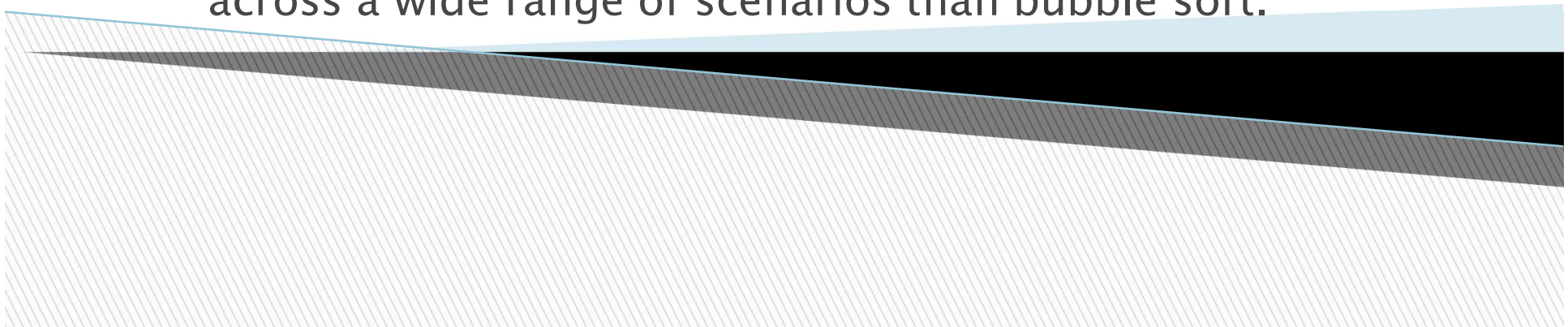
- **Ease of Implementation:** Bubble sort is one of the simplest sorting algorithms to understand and implement.
- **No Extra Memory Requirement:** It doesn't require additional memory or storage to sort the elements. This can be beneficial in situations where memory usage is a big concern.
- **Small Dataset Efficiency:** Bubble sort can be efficient for sorting small datasets or lists with only a few elements. In such cases, its simplicity and low overhead may make it competitive with more complex sorting algorithms.



Disadvantages:



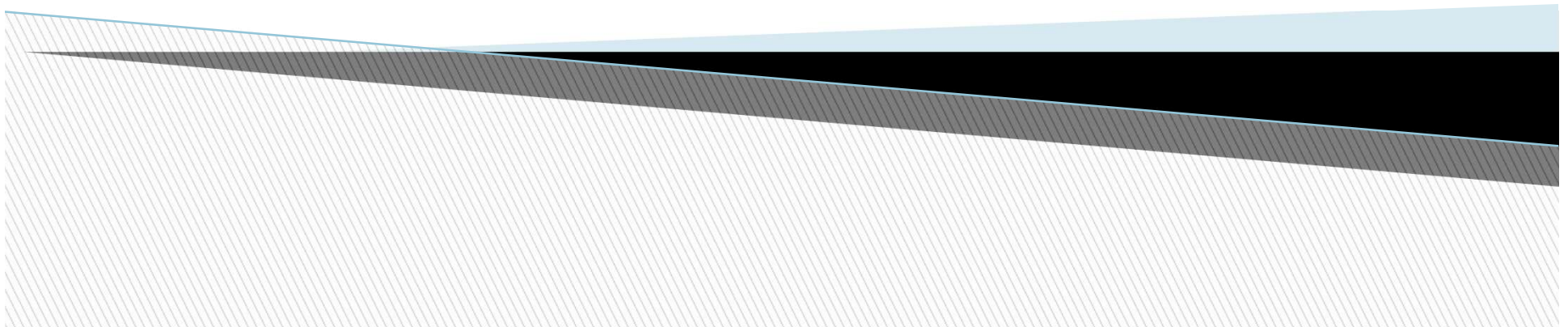
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.
- **Inefficient for Large Datasets:** Bubble sort becomes extremely slow for sorting large datasets, making it impractical for real-world applications where sorting efficiency is a critical factor.
- **Obsolete in Most Real-World Applications:** Bubble sort has largely been replaced by more efficient sorting algorithms like quicksort, merge sort in real-world applications. These algorithms offer better performance across a wide range of scenarios than bubble sort.



Merge Sort



Merge sort is using the **divide and conquer** approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal(or nearby equal) halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.



Algorithm

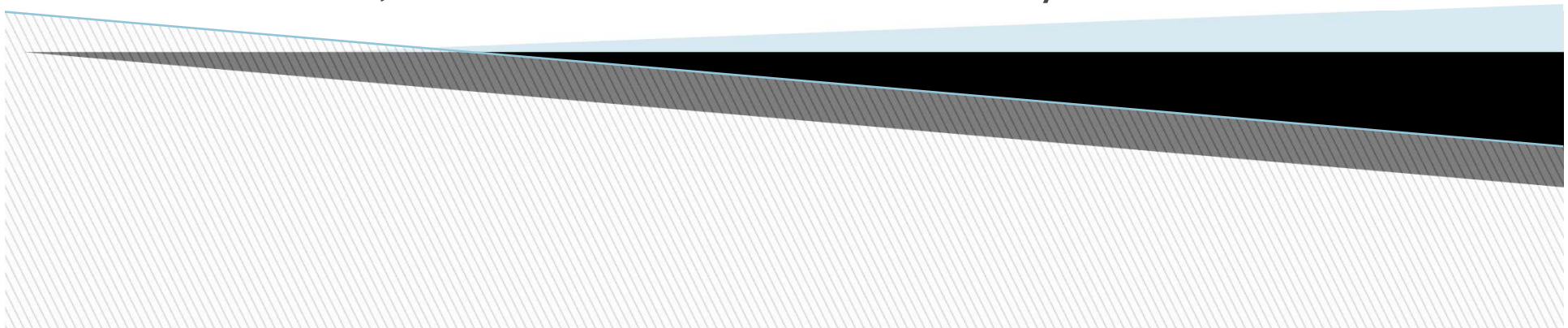


Step 1: Divide

- 1.1. Begin with an unsorted array we want to sort.
- 1.2. Divide the array into two equal (or nearly equal) halves.
- 1.3. Recursively apply the merge sort algorithm to each of the two halves. This division and recursion continue until we have subarrays with one element, as these are considered sorted.

Step 2: Conquer

- 2.1. The conquer step happens implicitly as part of the recursion. As the algorithm divides the array into smaller subarrays, it eventually reaches subarrays with one element, which are considered sorted by definition.

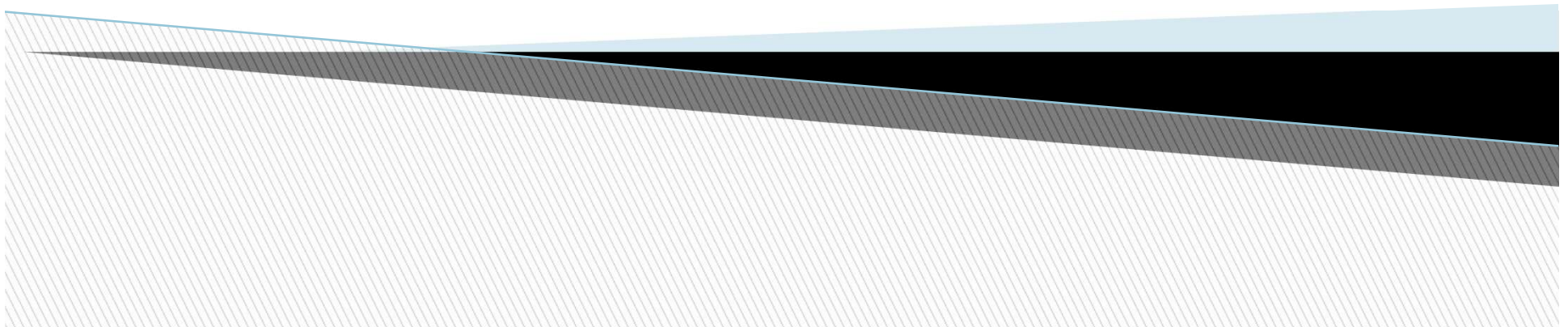


CONTD..



Step 3: Merge

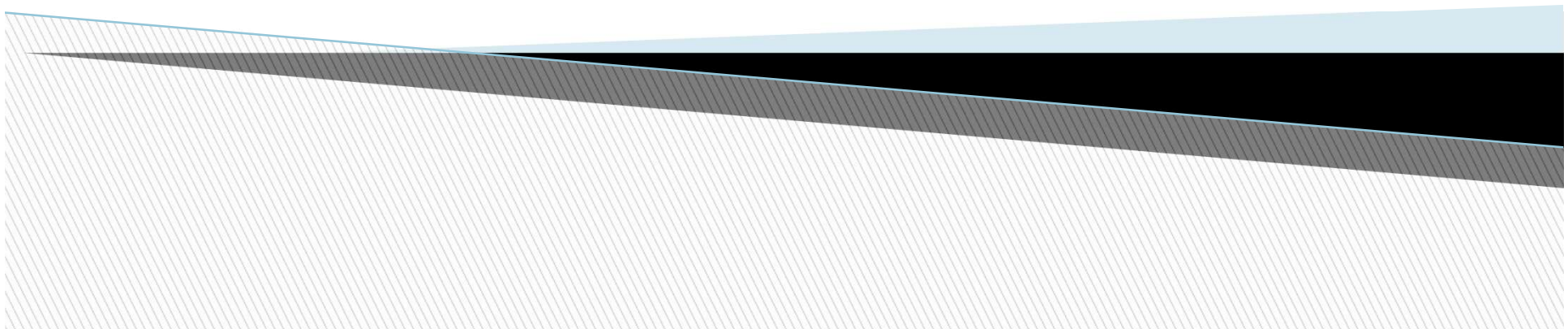
- 3.1. Merge the sorted subarrays back together into a single, sorted array. This is done by comparing elements from the two subarrays and placing them in the correct order in a temporary storage (usually a new array).
- 3.2. Start with two pointers, one for each subarray, initially pointing to the first element of each subarray.
- 3.3. Compare the elements at the current positions of the two pointers. Take the smaller element and place it in the temporary storage. Move the pointer one position forward in the subarray from which we took the element.



CONTD..



- 3.4. Repeat the comparison and placement process (3.3) until you have processed all elements from both subarrays.
- 3.5. If one of the subarrays has remaining elements, simply copy them into the temporary storage, as they are already sorted.
- 3.6. Copy the merged elements from the temporary storage back into the original array, overwriting the unsorted elements with the sorted ones.





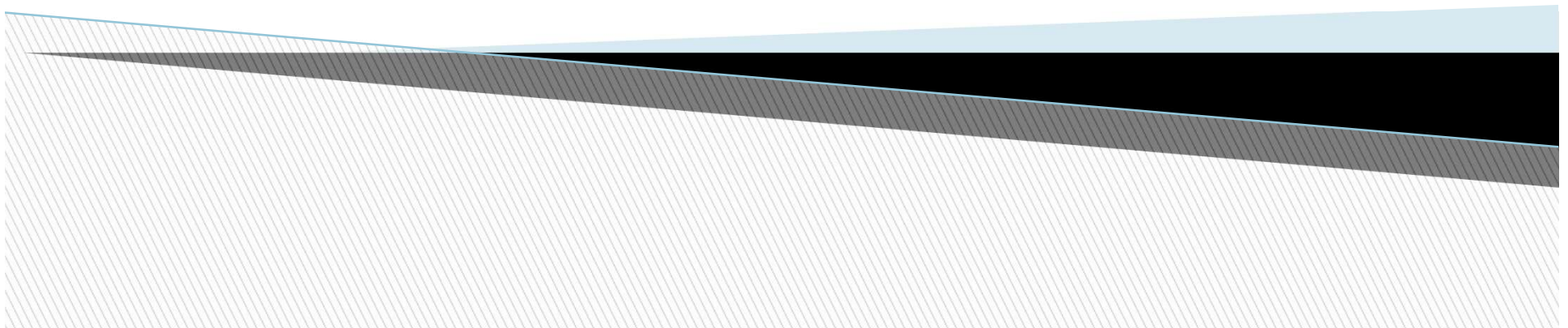
CONTD..

Step 5: Repeat (if necessary)

5.1. If there are more divisions to be made; repeat the process from Step 1 on each of the subarrays generated in Step 1.

NOTE:

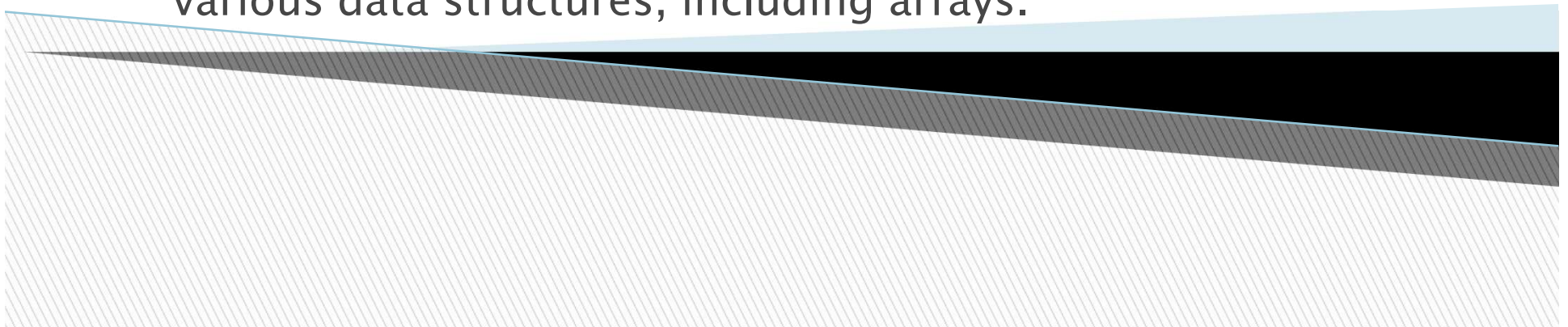
The merge sort algorithm continues recursively dividing, conquering, merging, and copying back until the entire array is sorted.





Advantages

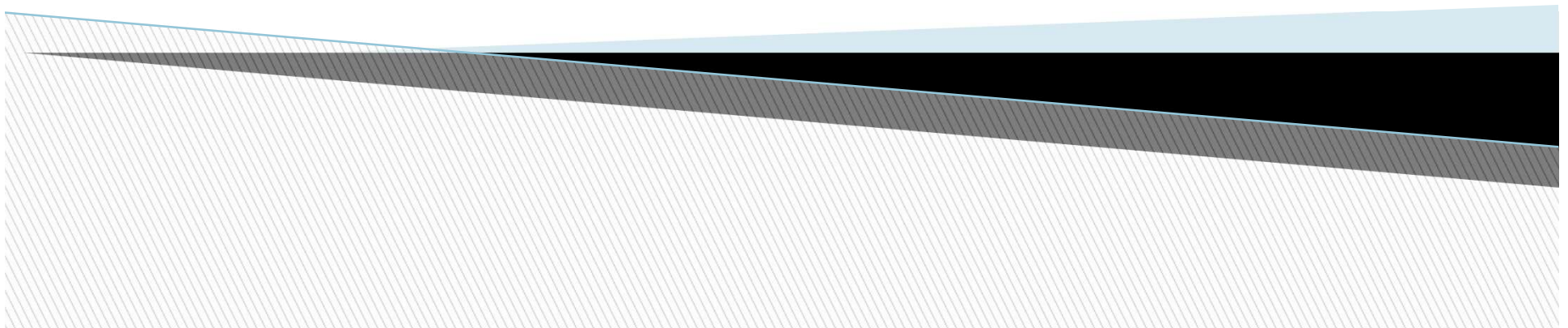
- **Efficiency for Large Datasets:** Merge sort is particularly efficient for sorting large datasets. Its divide-and-conquer approach allows it to efficiently sort large arrays or lists by breaking them into smaller, manageable subproblems. This is in contrast to some other sorting algorithms, like quicksort, which can perform poorly on large datasets in the worst case.
- **Parallelization:** Merge sort is well-suited for parallelization, as you can easily divide the sorting task into subproblems that can be processed concurrently. This improving sorting speed even further.
- **Adaptability:** Merge sort can be easily adapted to handle various data structures, including arrays.





Disadvantages:

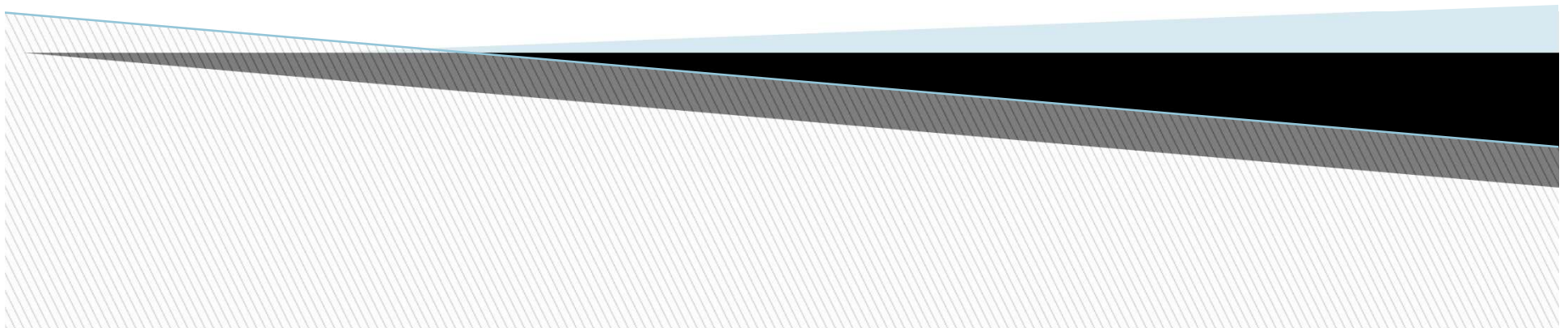
- **Space Complexity:** Merge sort requires additional memory for the temporary storage of subarrays during the merge phase. This space requirement can be a drawback, especially when working with limited memory resources.
- **Slower on Small Lists:** Merge sort can be less efficient than some other sorting algorithms, like insertion sort or quicksort, when sorting very small lists.





CONTD..

- **Slower on Some Hardware:** Merge sort's performance can be impacted by the architecture of the computer it runs on. On machines with limited memory, the additional memory allocation and data copying steps during merging can make the process slower.
- **Complex Implementation:** Implementing merge sort can be more complex compared to some other sorting algorithms. It involves recursive calls and careful management of subarrays and temporary storage.



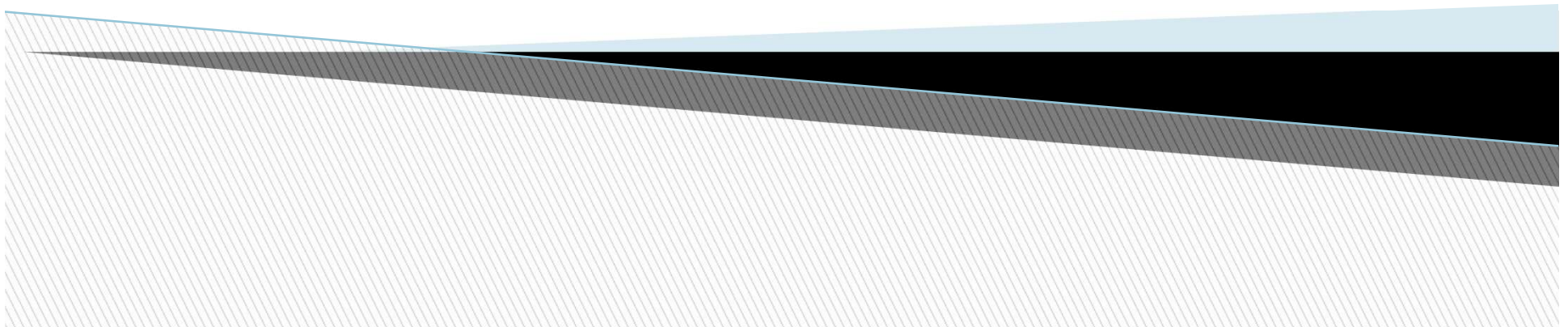
Working of Merge Sort Algorithm



Let the elements of array are -

12	31	25	8	32	17	40	42
----	----	----	---	----	----	----	----

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.



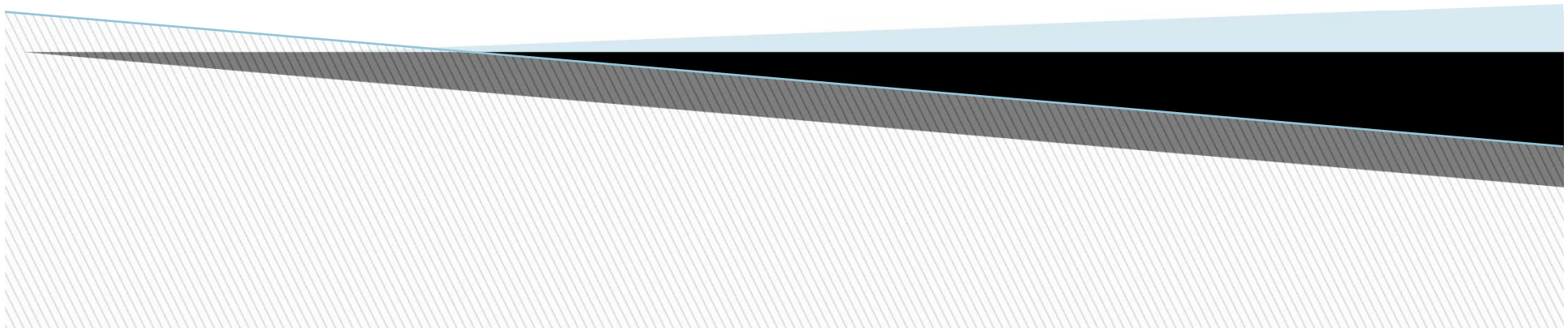
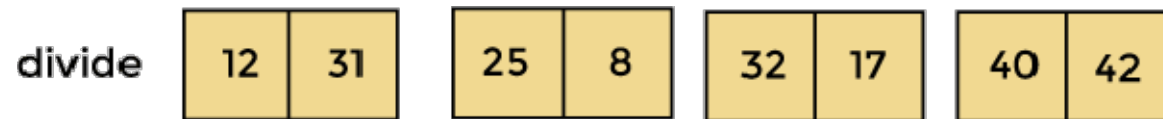
CONTD..



As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



CONTD..



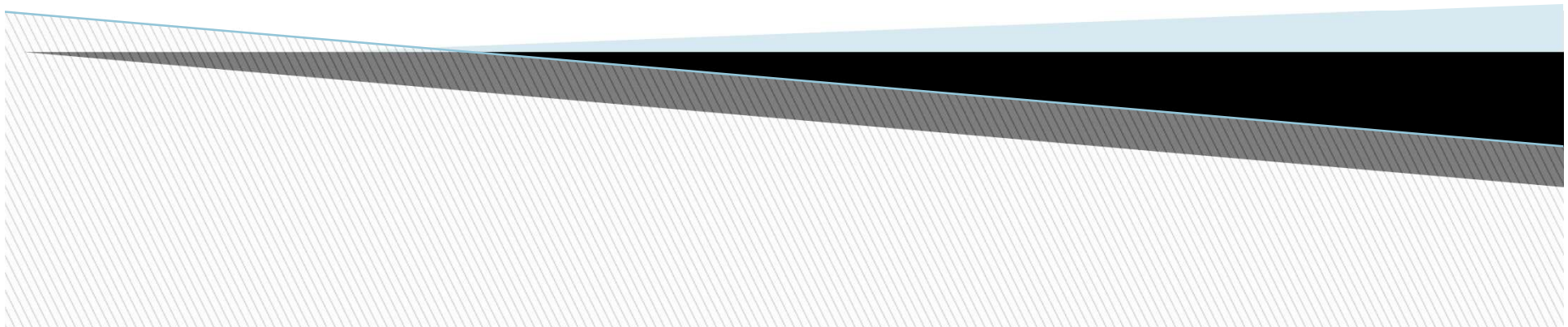
Now, again divide these arrays to get the atomic value that cannot be further divided.



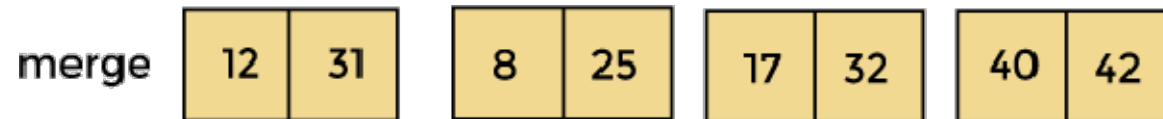
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

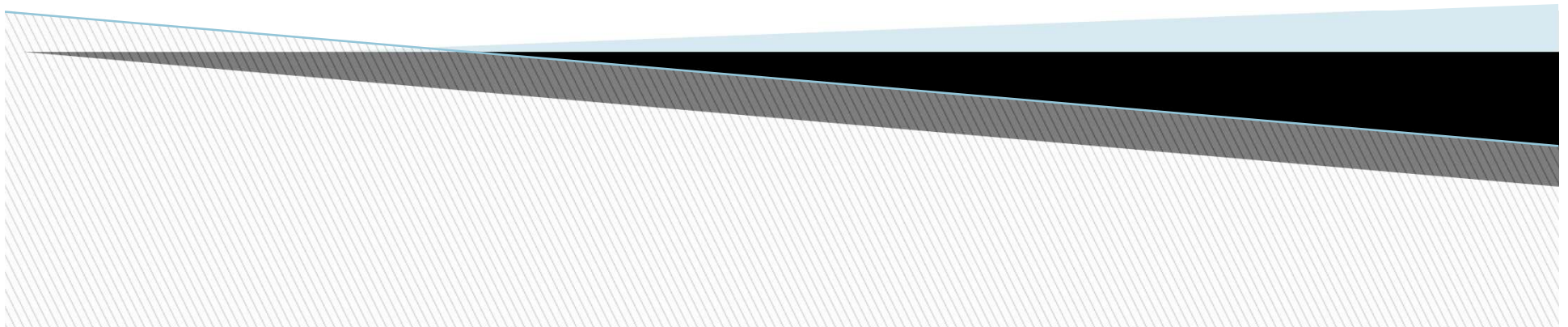
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



CONTD..



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of four values in sorted order.



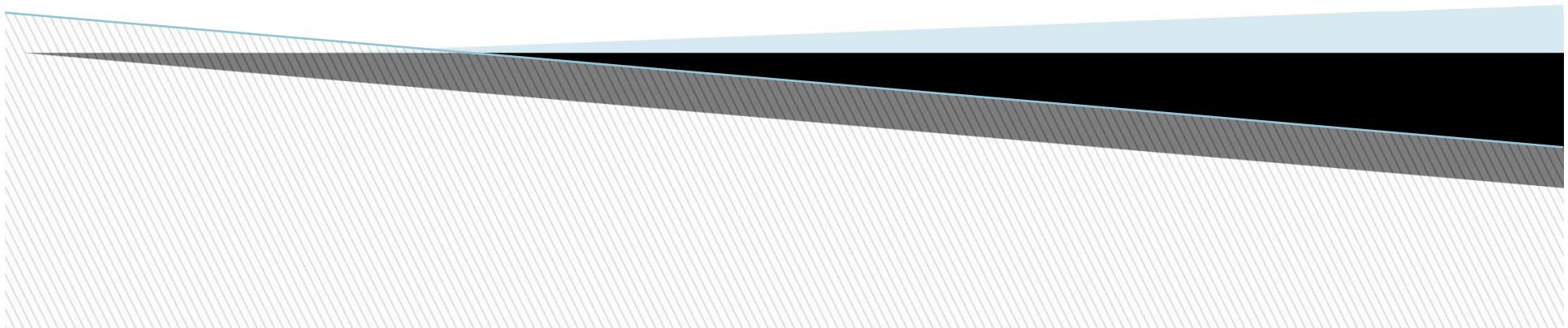
CONTD..



merge	8	12	25	31	17	32	40	42
-------	---	----	----	----	----	----	----	----

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like:

8	12	17	25	31	32	40	42
---	----	----	----	----	----	----	----



Programming for Problem Solving

(ESCG101)

Module 4

Module IV: Function

- Functions (including using built-in libraries)
- Parameter passing in functions, call by value.
- Passing arrays to functions: the idea of call.

Function

- ❑ A function is a group of statements that together performs a task
- ❑ Every C program has at least one functions, which is main()
- ❑ Additional functions can be defined as per the requirement

Importance of function in C programming language:

Function refers to a group of statements to perform a specific task. It is useful in any programming language, especially in C programming because it not only provides large sets of built-in functions but also user-defined functions for executing many tasks. Here are other reasons why it is important:

- It allows it to remove big programs\' complexity by breaking them down into small functions.
- It enhances the readability of programs so it can easily understand the codes.
- The top-down execution in C lets it learn the program flow and control every line.
- It can also create its header file and reuse it in other programs.
- It can test individual functions and make codes prone to fewer errors.

Types of Function

□ Two types

- Pre-defined -- these are the inbuilt functions of C library like `main()`, `printf()`, `scanf()` etc
- User-defined -- these are defined by the user

Library functions or built-in functions are functions : are pre-defined by the programming language. It can use those functions for different purposes. C language is enriched with hundreds of library functions. The `main()` is also a built-in function responsible for the code program execution. Some popular library functions of header files `"math.h"` `pow()` - It calculates the power of a number, `sin()` - It calculates the sine of a number, `cos()` - It calculates the cosine of a number, `exp()` - It calculates the exponent of a number, `sqrt()` - It calculates the square root of a number.

➤ **Can a C program be compiled or executed in the absence of a `main()`?**

The program will be compiled but will not be executed. To execute any C program, `main()` is required.

Objective of the main () function in C:

The main () function in C is the inlet to the C program. It is the entry point where the process of execution of the program starts. When the execution of the C program initiates, the control of the program is directed towards the main () function.

It is mandatory that every C program has a main () function. Although it is the function that indicates the programming process, it is not the first function to be executed.

In C, both constants and variables are widely used while designing a program. The major difference between variables and constants is that variables can alter their assigned value at any point of the program. In contrast, the value of the constant remains unaltered during the entire program. The value of the constant is locked during the execution of the program.

How does a function operate?

- ❑ Codes can be divided into separate parts
- ❑ Those parts can be used to create functions
- ❑ Example:

```
int main() // main is pre defined
{
    function
    //statements
}
void show() // show is user defined function
{
    // statements
}
```

Syntax

```
return_type function_name(arguments)
{
    statements;
}
```

-here, return type can be void, int, float, char.

Function

- ❑ **Calling function-** the function who calls other function
- ❑ **Called function-** the function who get called by other function
- ❑ **Actual parameters-** arguments which are passed to called function
- ❑ **Formal parameters-** arguments of the called function

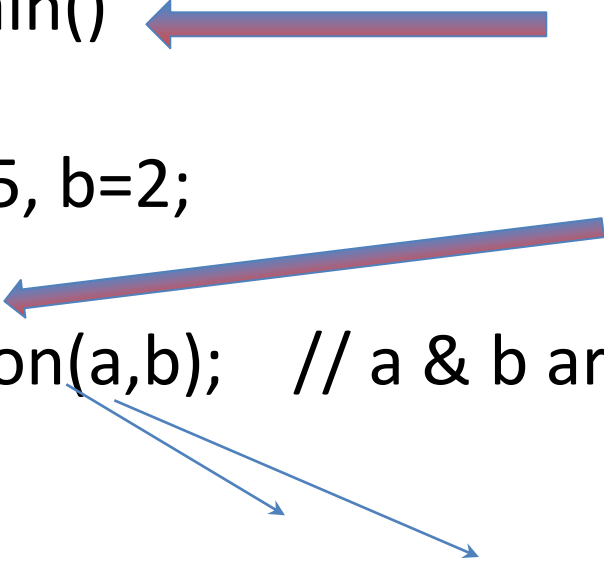
Example

```
int main()
{
    int a=5, b=2;
    addition(a,b); // a & b are actual parameters
}
```

calling function

called function

```
void addition(int x, int y) // x & y are formal
{
    int sum=x+y;
}
```



Function prototype

- Function prototype is needed when user defined function is defined after main function
- It is just the declaration of the function with any body

- Example :

```
void show()
```

```
{
```

```
    // statements
```

```
}
```

```
// here prototype will be void show(); defined before main  
function
```


Local Variable: A local variable is a type of variable that we declare inside a block or a function.

Global Variable: A global variable is one that is defined outside the scope of all functions. Because global variables have a global scope, they can be accessed and modified by any function, structure, or scope in C.

Static Variable: A static variable possesses the property of preserving its actual value even after it is out of its scope.

- In C it is always the most local variable that gets preference.
- Same variable name can be declared to the variables with different variable scopes as the following example.

```
int var;  
void function()  
{  
    int variable;  
}  
int main()  
{  
    int variable;  
}
```

Difference between the local and global variables in C :

Local variables are declared inside a block or function but global variables are declared outside the block or function to be accessed globally.

Local Variables	Global Variables
Declared inside a block or a function.	Variables that are declared outside the block or a function.
By default, variables store a garbage value.	By default value of the global value is zero.
The life of the local variables is destroyed after the block or a function.	The life of the global variable exists until the program is executed.
Variables are stored inside the stack unless they are specified by the programmer.	The storage location of the global variable is decided by the compiler.
To access the local variables in other functions parameter passing is required.	No parameter passing is required. They are globally visible throughout the program.

Differentiate between getch() and getche():

Both the functions are designed to read characters from the keyboard and the only difference is that getch(): reads characters from the keyboard but it does not use any buffers. Hence, data is not displayed on the screen. getche(): reads characters from the keyboard and it uses a buffer. Hence, data is displayed on the screen.

toupper() function with an example: toupper() is a function designed to convert lowercase words/characters into upper case.

```
#include<stdio.h>
#include<ctype.h>
int main()
{
    char c;
    c=a;
    printf("\n%c after conversions %c", c, toupper(c));
    return 0;
}
```

Output: A

rand() function: Random numbers in C Language can be generated as follows:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a,b;
    for(a=1;a<=10;a++)
    {
        b=rand();
        printf("%d\n",b);
    }
    return 0;
}
```

Output:

```
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421
```

Limitations of scanf() and how can it be avoided: The Limitations of scanf() are as follows:

- scanf() cannot work with the string of characters.
- It is not possible to enter a multiword string into a single variable using scanf().

To avoid this the gets() function is used. It gets a string from the keyboard and is terminated when enter key is pressed. Here the spaces and tabs are acceptable as part of the input string.

Difference between macros and functions C programming language:

Macro : Macro is a small chunk of code which gets replaced whenever respective macros have been called. It is called a preprocessor directive, and it requires pre-processing. It does not require any type-checking. The increased number of macros increases the size of the whole program. Macro execution is faster than function execution. Use macro when it need to execute small codes multiple times.

Function : A function is a group of one or more statements to perform a specific job. It could be predefined or user-defined, and it needs compilation. It requires type-checking. The function length does not impact the size of the program. The function execution is slower than the macro execution. Use functions when executing large codes and different tasks in a program.

String functions in the C language:

C provides many string functions in its standard library header file- 'string.h'. The most commonly used string functions are:

strlen(string_source): Returns length of the string_source

strcpy(string_destination, string_source): Copies string_source into string_destination.

strcat(string_destination, string_source): Joins both the strings and stores the output in string_destination.

strcmp(string_destination, string_source): Compares both strings, if found equal returns 0(zero).

strrev(string_source): Reverses the string

strlwr(string_source): Returns the string into lowercase

strupr(string_source) Returns the string into uppercase

Function prototype with an example:

Function prototype of prototype function refers to the declaration of any function with the following information:

- Name of function
- Function return type
- Number of arguments and their data type.

The function prototype provides the assurance that the function call matches the return type and returns the correct number of arguments of the called function. Simply put, it helps prevent the user from making mistakes while using multiple functions in the program. Syntax: `return_type function_name(datatype argument1, datatype argument 2, ...);`

Example: `int sum(int x, int y);` This line is declaring the function `"sum"` which is taking two integer arguments. The function will also return the integer.

Difference between malloc() and calloc() in the C programming language:

Parameter	Malloc()	Calloc()
Definition	It is a function that creates one block of memory of a fixed size.	It is a function that assigns more than one block of memory to a single variable.
Number of arguments	It only takes one argument.	It takes two arguments.
Speed	malloc() function is faster than calloc().	calloc() is slower than malloc().
Efficiency	It has high time efficiency.	It has low time efficiency.
Usage	It is used to indicate memory allocation.	It is used to indicate contiguous memory allocation.

C program to convert number to string using sprintf():

```
#include <stdio.h>
#include <string.h>
// Driver code
int main()
{
    char res[20];
    float a = 32.23;
    sprintf(res, "%f", a);
    printf("\nThe string for the num is %s", res);
    return 0;
}
```

Output:

The string for the num is 32.230000

C program to check whether a string is palindrome or not:

```
#include <stdio.h>
#include <string.h>
void Palindrome(char s[]);
int main()
{
    Palindrome("abba");
    return 0;
}
void Palindrome(char s[])
{
    /* start will start from 0th index and end will start from length-1 */
    int start = 0;
    int end = strlen(s) - 1;
    /* Comparing characters until they are same */
    while (end > start)
    {
        if (s[start++] != s[end--])
        {
            printf("%s is not a Palindrome \n", s);
        }
    }
    printf("%s is a Palindrome \n", s);
}
```

A string is said to be palindrome if it remains the same on reading from both ends. It means that when you reverse a given string, it should be the same as the original string.

Output:

abba is a Palindrome

C program to find out the factorial of a number using function:

```
#include<stdio.h>
long int factorial(int n);
int main()
{
    int num;
    printf("Enter a number :");
    scanf("%d", &num);
    if (num<0)
        printf("No factorial of negative number\n");
    else
        printf("Factorial of %d is %ld\n", num, factorial(num));
    return 0;
}
long int factorial(int n)
{
    int i;
    long int fact=1;
    if (n==0)
        return 1;
    for (i=n; i>1; i--)
        fact*=i;
    return fact;
}
```

Output:

Enter a number: 5
Factorial of 5 is 120

Thank You

Programming for Problem Solving

(ESCG101)

Module V

Recursion:

Recursion is the process of making the function call itself directly or indirectly. A recursive function solves a particular problem by calling a copy of itself and solving smaller sub-problems that sum up the original problems.

Recursion helps to reduce the length of code and make it more understandable. The recursive function uses a LIFO (Last In First Out) structure like a stack. Every recursive call in the program requires extra space in the stack memory.

While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

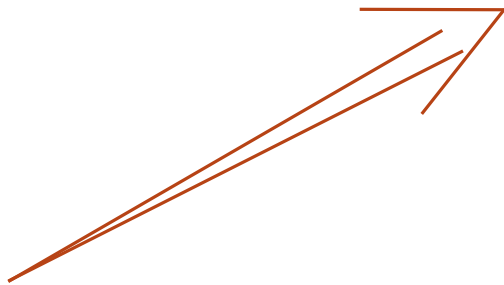
Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example:

```
int main ()
{
    .....
    rec ();
    .....
} /* End of main () */
void rec ()
{
    .....
    rec ();    /* recursive call */
    .....
} /* End of rec () */
```

□ Example: (finding factorial using recursion)

```
int fact(int x)
{
    if(x==1)
        return 1;
    else
        f=x * fact (x-1); // fact
                           is defined in term of fact itself
}
```



Here the function rec () is calling itself inside its own function body, so rec () is a recursive function.

Factorial of a given number using a recursive function:

```
#include <stdio.h>
long int fact(int n);
int main()
{
    int num;
    printf ("Enter a number: ");
    scanf("%d", &num);
    if (num<0)
        printf ("No factorial for negative number\n");
    else
        printf ("Factorial of %d is %ld\n", num, fact(num));
    return 0;
}
long int fact(int n)
{
    if(n <= 1)
        return 1;
    return (n*fact(n - 1));
}
```

Output:

Enter a number: 5
Factorial of 5 is 120

Fibonacci series:

Fibonacci series is a sequence of integers that starts with 0 followed by 1, in this sequence the first two terms i.e. 0 and 1 are fixed, and we get the successive terms by summing up their previous last two terms. i.e, the series follows a pattern that each number is equal to the sum of its preceding two numbers.

Mathematically it can be written as : $F(n)=F(n-1)+F(n-2)$

where F is the Fibonacci function having base values $F(0)=0$ and $F(1)=1$

0 1 1 2 3 5 8 13 21 34

In this series each number is a sum of the previous two numbers.

Program to generate fibonacci series:

```
# include <stdio.h>
int fib(int n);
int main()
{
    int nterms, i;
    printf ("Enter number of terms: ");
    scanf ("%d", &nterms);
    for (i=0; i<nterms; i++)
    {
        printf ("%d\t", fib(i));
    }
    printf("\n");
    return 0;
}
int fib (int n)
{
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return (fib(n-1) + fib(n-2));
}
```

Output:

Enter number of terms: 8

0 1 1 2 3 5 8 13

Program to find GCD (Greatest Common Divisor) using recursion:

```
# include <stdio.h>
int GCD (int a, int b);
int main ()
{
    int x, y, z;
    printf ("Enter two numbers : ");
    scanf ("%d%d", &x,&y);
    z = GCD (x,y);
    printf ("GCD of two numbers is %d", z);
    return 0;
}
int GCD (int a, int b)
{
    if (b==0)
        return a;
    return GCD (b, a%b);
}
```

Output:

```
Enter two numbers : 35 21
GCD of two numbers is 7
```

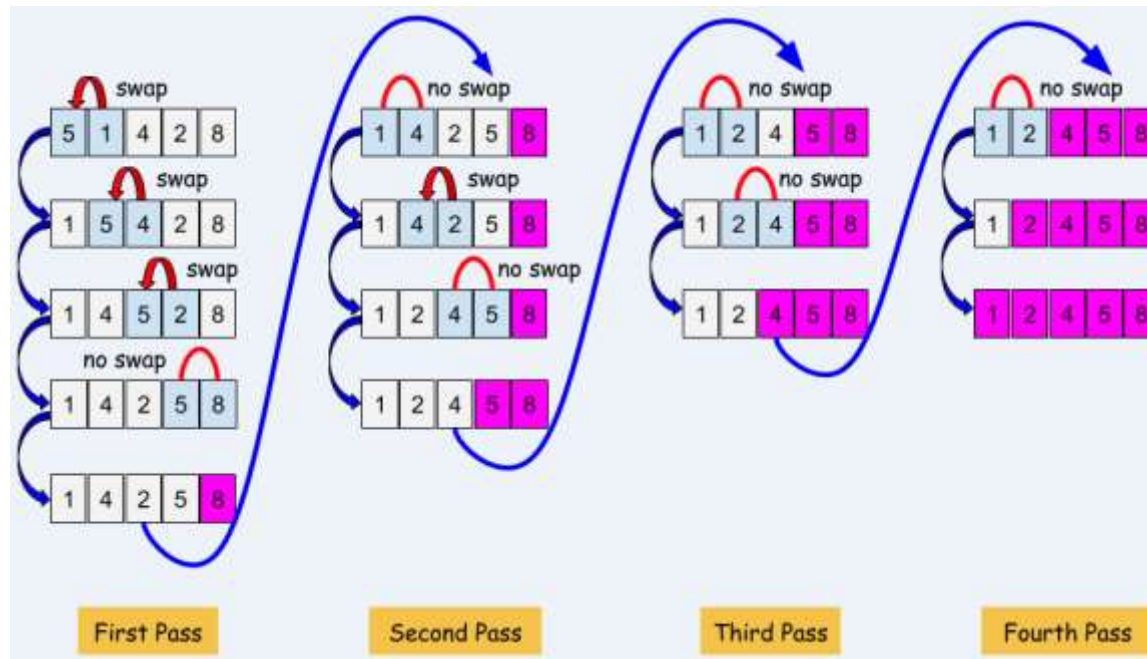
Sorting

- Process of arranging the data in some order (ascending/descending)

- Types:
 - bubble
 - merge
 - quick

Bubble Sort Algorithm, Explain with a program:

Bubble Sort is a simple sorting algorithm commonly used to sort elements in a list or array. It works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. The algorithm iterates through the list multiple times until no more swaps are needed, resulting in a sorted sequence.



https://www.google.com/search?q=bubble+sort+in+c&sca_esv=576600514&rlz=1C1FKPE_en&hl=en&tbm=isch&sxsrf=AM9HkKnyXT2mf_2lQ0G8LvfnY1ZwhRELJA:1698263952770&source=lnms&sa=X&ved=2ahUKEwiH346d_pGCAXWna2wGHfWC4wQ_AUoAXoECAEQAw&biw=1024&bih=608&dpr=1.25#imgrc=wmo2mm5TBzOckM&imgdii=7Wz2xhFDUC5bxM

Program to sort an array using Bubble sort algorithm:

```
#include <stdio.h>
void sort (int m, int x[]);
int main()
{
    int i, marks[5] = {40, 90, 73, 81, 35};
    printf ("Marks before sorting\n");
    for (i=0; i<5; i++)
    {
        printf ("%d\t", marks[i]);
    }
    printf ("\n");
    sort (5, marks);
    return 0;
}
void sort (int m, int x[])
{
    int i, j, t;
    for (i=0; i<=m-1; i++)
    {
        for (j=0; j<=m-i; j++)
        {
            if (x[j-1]>=x[j])
            {
                t=x[j-1];
                x[j-1]=x[j];
                x[j]=t;
            }
        }
    }
    printf ("Marks after sorting\n");
    for (i=0; i<5; i++)
    {
        printf ("%d\t", x[i]);
    }
}
```

Output:

Marks before sorting				
40	90	73	81	35
Marks after sorting				
35	40	73	81	90

Thank You