MODULE 3
Functions

3.12 Introduction

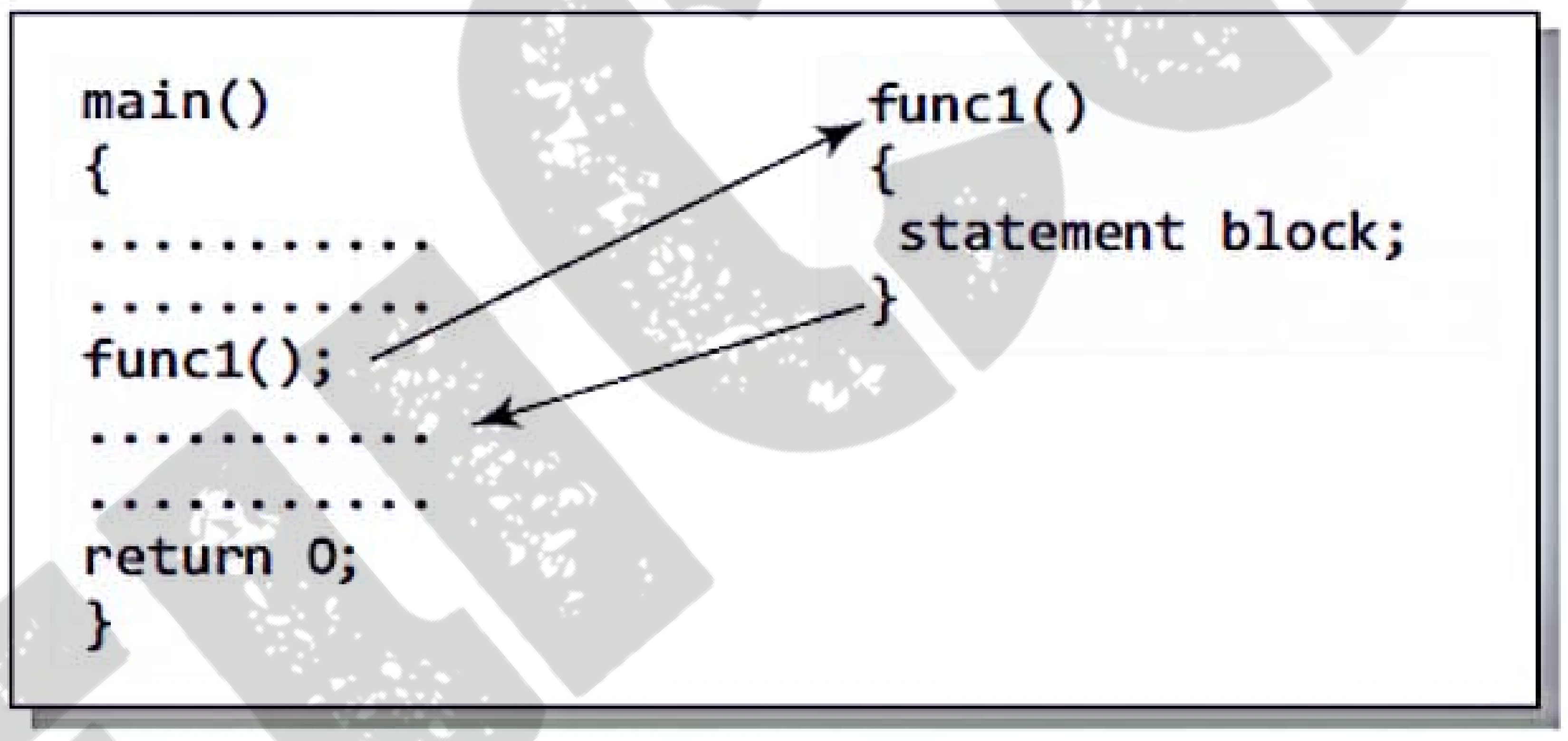
- ✓ C enables its programmers to **break up a program into segments** commonly known as **functions**, each of which can be written **more or less independently of the others**.
- ✓ **Every function in the program is supposed to perform a well-defined task.**
- ✓ Therefore, the program code of one function is completely insulated from the other functions.

Definition

- ✓ **"The set of instructions that performs some specific, well-defined task is called as a Function."**

Or

- ✓ **"Function is a small program or program segment that carryout some specific well-defined tasks".**
- ✓ Fig. 1.9 explains how the main() function calls another function to perform a well-defined task.
- ✓ In the figure, we can see that main() calls a function named func1(). Therefore, main() is known as the **calling function** and func1() is known as the **called function**.
- ✓ The moment the compiler encounters a function call, the control jumps to the statements that are a part of the called function.
- ✓ After the called function is executed, the control is returned to the calling program.



```
main()                      func1()
{                           {
...........                  statement block;
...........                 }
func1();
..........
..........
return 0;
}
```

Figure 1.9 `main()` calls `func1()`

3.12.1 Why are Functions Needed?

- ✓ **Dividing the program** into separate **well-defined functions** facilitates each function to be **written and tested separately.** This **simplifies the process of getting the total program to work.**
- ✓ **Understanding, coding, and testing** multiple separate functions is easier than doing the same for one big function.
- ✓ If a **big program** has to be developed without using any function other than main(), then there will be **countless lines** in the main() function and **maintaining that program** will be a **difficult task.**
- ✓ All the **libraries in C contain a set of functions**, which have been **pre-written and pre-tested**, so the **programmers can use them without worrying about their code details.** This **speeds up program development**, by allowing the programmer to concentrate only on the code that he has to write.

- ✓ Like C libraries, **programmers can also write their own functions and use them** from different points in the main program or any other program that needs its functionalities.
- ✓ When a big program is broken into comparatively smaller functions, then different programmers working on that project can **divide the workload by writing different functions.**

3.13 Using Functions

While using functions, we will be using the following terminologies:

- ✓ A function **f** that **uses another function g** is known as the **calling function**, and **g** is known as the **called function**.
- ✓ The **inputs that a function takes** are known as **arguments**.
- ✓ When a **called function returns some result back** to the calling function, it is said to **return** that **result.**
- ✓ The **calling function may or may not pass parameters** to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- ✓ **Function declaration** is a declaration statement that identifies **a function's name, a list of arguments that it accepts, and the type of data it returns.**
- ✓ **Function definition** consists of a **function header** that identifies the function, followed by the **body of the function** containing the executable code for that function.

3.14 Types of Functions

i. Library Functions/Pre-Defined/ Built-in Functions

- ✓ C Library of C Compiler has a collection of various functions which perform some standard and pre-defined tasks.
- ✓ These functions written by designers of C Compilers are called as Library functions/Pre-Defined/Built-in functions.

Ex: sqrt(n)- computes square root of n.
pow(x,y)- computes x^y.
printf()- used to print the data on the screen.
scanf()- used to read the data from the keyboard.
abs(x)- computes absolute value of x.

ii. User-Defined/ Programmer Defined Functions

- ✓ The functions written by the programmer/user to do the specific tasks is called User-Defined/ Programmer Defined Functions.
- ✓ main( ) is the user defined function.

3.15 Elements of User-Defined Functions

- ✓ The three elements of user-defined functions are shown below:
  i. Function Prototype/Declaration
  ii. Function Definition
  iii. Function Call

3.15.1 Function Declaration/Function Prototype

- ✓ Before using a function, the compiler must know the **number of parameters and the type of parameters** that the function expects to receive and the **data type of value** that it will return to the calling program.
- ✓ Placing the function declaration statement **prior to its use** enables the compiler to make a **check on the arguments used** while calling that function.
- ✓ The general format for declaring a function that accepts arguments and returns a value as result can be given as:

return_data_type function_name(data_type variable1, data_type variable2,..);

Here, **function_name** is a **valid name for the function**. A function should have a meaningful name that must specify the task that the function will perform.

return_data_type specifies the **data type of the value that will be returned** to the calling function as a result of the processing performed by the called function.

(data_type variable1, data_type variable2, ...) is a **list of variables** of specified data types. These variables are passed from the calling function to the called function. They are also known as **arguments or parameters** that the called function accepts to perform its task.

Ex: int add(int a,int b);

- ✓ Things to remember about function declaration:
  - After the declaration of every function, there should be a **semicolon.** If the semicolon is missing, the compiler will generate an error message.
  - The function declaration is **global.**
  - **Use of argument name** in the function declaration is **optional.**
    int func(int, char, floa t);
    or
    int func(int num, char ch, float fnum);
  - A function **cannot be declared** within the body of another function.
  - A function having **void as its return type cannot return any value.**
  - A function having void as its parameter list **cannot accept any value.** So the function declared as
    void print();
    does not accept any input/arguments from the calling function .
  - If the function declaration does not specify any return type, then by default, the function **returns an integer value.** Therefore, when a function is declared as
    sum(int a, int b);
    Then the function sum accepts two integer values from the calling function and in sum returns an integer value to the caller.
  - Some compilers make it compulsory to declare the function before its usage while other compilers make it optional.

3.15.2 Function Definition

- ✓ When a **function is defined, space is allocated** for that function in the memory.
- ✓ A function definition comprises of two parts:
  - **Function header**
  - **Function body**

✓ The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
        ............
        statements
        ............
        return(variable);
}
```

✓ The **number of arguments and the order of arguments** in the function header must be the **same as that given in the function declaration** statement.

✓ While **return_data_type function_name(data_type variable1, data_type variable2,...)** is known as the **function header**, the rest of the portion comprising of **program statements within the curly brackets { } is the function body** which contains the code to perform the specific task.

✓ Note that the function header is same as the function declaration. The only difference between the two is that a **function header is not followed by a semi-colon.**

Ex:
```
int add(int a,int b)
    {
            int sum;
            sum=a+b;
            return sum;
     }
```

3.15.3 Function Call

✓ The function call statement **invokes the function.**

✓ When a function is invoked, the **compiler jumps to the called function** to **execute the statements** that are a part of that function.

✓ Once the called function is executed, the program control passes back to the calling function. A function call statement has the following syntax:

function_name(variable1, variable2, ...);

✓ The following points are to be noted while calling a function:

- **Function name and the number and the type of arguments** in the **function call** must be **same** as that given in the **function declaration and the function header** of the function definition.
- **Names** (and not the types) of variables in **function declaration, function call, and header of function definition may vary.**
- **Arguments** may be passed in the **form of expressions** to the called function. In such a case, **arguments are first evaluated** and converted to the type of formal parameter and then the body of the function gets executed.
- If the **return type** of the function is **not void**, then the value returned by the called function may be assigned to some variable as given below.

variable_name = function_name(variable1, variable2, ...);

Ex: add(a,b);

3.16 return STATEMENT

- ✓ **The return statement terminates the execution of the called function and returns control to the calling function.**
- ✓ When the return statement is encountered, the program execution resumes in the calling function at the point immediately following the function call.
- ✓ A return statement **may or may not return a value to the calling function.**
- ✓ The syntax of return state can be given as

return <expression>;

Here expression is placed in between angular brackets because specifying an expression is optional.

- ✓ A function that has **void return type cannot return any value** to the calling function.

3.17 Passing Parameters to Functions

- ✓ There are **two ways** in which arguments or parameters can be passed to the called function.

Call by value: The **values of the variables** are passed by the calling function to the called function.

Call by reference The **addresses of the variables** are passed by the calling function to the called function.

1. Call by Value

- ✓ In this method, the **called function creates new variables to store the value of the arguments passed to it.** Therefore, the called function uses a **copy of the actual arguments** to perform its intended task.
- ✓ **If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function, no change will be made to the value of the variables. This is because all the changes are made to the copy of the variables and not to the actual variables.**

Example: Write a C program to add two numbers using call by value.

```
#include<stdio.h>
int add (int a,int b)
{
        int  sum;
        sum = a + b;
        return sum;
}

void main()
{
        int a,b, res;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        res = add(a,b);
        printf("result =%d\n", res);
}
```

Output:

```
Enter the values of a and b: 4 5
result =9
```

- ✓ Following are the points to remember while passing arguments to a function using the call-by value method:
  - **When arguments are passed by value, the called function creates new variables of the same data type as the arguments passed to it.**
  - **The values of the arguments passed by the calling function are copied into the newly created variables.**
  - **Values of the variables in the calling functions remain unaffected when the arguments are passed using the call-by-value technique.**

Pros and cons

- ✓ The biggest advantage of using the call-by-value technique is that **arguments** can be passed as **variables, literals, or expressions**, while its main drawback is that **copying data consumes additional storage space.**
- ✓ In addition, it can take a **lot of time to copy**, thereby resulting in **performance penalty**, especially if the function is called many times.

2. Call by Reference

- ✓ **When the calling function passes arguments to the called function using the call-by-value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement.** A better option is to pass arguments using the call-by-reference technique.
- ✓ In this method, we declare the **function parameters as references** rather than normal variables.
- ✓ **When this is done, any changes made by the function to the arguments it received are also visible in the calling function.**
- ✓ To indicate that an argument is passed using call by reference, an **asterisk (*)** is placed after the type in the parameter list. Hence, in the call-by-reference method, a function **receives an implicit reference to the argument**, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well.

Example:

1. Write a C program to add two numbers using call by reference.

```
#include<stdio.h>

int add (int *a,int *b)
{
        int  sum;
        sum = *a + *b;
        return sum;
}

void main()
{
        int a,b, res;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        res = add(&a,&b);
        printf("result =%d\n", res);
}
```

Output:

```
Enter the values of a and b: 4 5
result =9
```

2. Write a C program to swap two numbers using call by reference.

```
#include<stdio.h>
void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
void main()
{
        int a,b;
        printf("Enter the values of a and b:");
        scanf("%d%d",&a,&b);
        printf("Before swapping: a=%d\tb=%d", a, b);
        swap(&a,&b);
        printf("After swapping: a=%d\tb=%d", a, b)
}
```

```
Enter the values of a and b: 10 20
Before swapping: a=10        b=20
After swapping: a=20         b=10
```

Output:

```
Enter the values of a and b: 10 20
Before swapping: a=10        b=20
After swapping: a=20         b=10
```

Advantages

- ✓ Since arguments are not copied into the new variables, it provides **greater time and space efficiency.**
- ✓ The **function can change the value** of the argument and the change is reflected in the calling function.
- ✓ A function can **return only one value**. In case we need **to return multiple values**, we can pass those arguments by reference, so that the modified values are visible in the calling function.

Disadvantages

- ✓ However, the drawback of using this technique is that if **inadvertent changes** are caused to variables in called function then these **changes would be reflected in calling function** as **original values would have been overwritten.**

3.18 Scope of Variables

- ✓ In C, all **constants and variables** have a **defined scope**.
- ✓ By scope we mean the **accessibility and visibility of the variables at different points in the program.**
- ✓ A variable or a constant in C has **four types of scope: block, function, program, and file.**

3.18.1 Block Scope

- ✓ We have studied that a **statement block is a group of statements enclosed within opening and closing curly brackets { }.**
- ✓ **If a variable is declared within a statement block then as soon as the control exits that block, the variable will cease to exist.**
- ✓ Such a variable also known as a **local variable is said to have a block scope.**
- ✓ So far we had been using local variables.
- ✓ For example, if we declare an integer x inside a function, then that variable is unknown to the rest of the program (i.e., outside that function).
- ✓ Variables declared with **same names** as those in outer block **mask the outer block variables** while executing the inner block.
- ✓ In nested blocks, **variables declared outside the inner blocks are accessible to the nested blocks**, provided these variables are not re-declared within the inner block.

3.18.2 Function Scope

- ✓ Function scope indicates that a **variable is active and visible from the beginning to the end of a function.**
- ✓ In C, only the **goto label** has function scope.
- ✓ In other words function scope is applicable only with goto label names. This means that the programmer cannot have the same label names inside a function.

Ex:

```
void main()
{
        …
        …
        loop: /*A goto label has function scope */
        …
        …
        goto loop /* the goto statement */
        …
        …
}
```

- ✓ In this example, the **label loop is visible from the beginning to the end of the main () function.** Therefore, there **should not be more than one label having the same name** within the main() function.

3.18.3 Program Scope

- ✓ Till now we have studied that variables declared within a function are **local variables**. These local variables (also known as internal variables) are **automatically created** when they are **declared in the function** and are **usable only within that function**. The local variables are **unknown to other functions** in the program. Such variables **cease to exist after the function** in which they are declared is exited and are re-created each time the function is called.
- ✓ However, if you want a **function to access some variables which are not passed to it as arguments**, then declare those variables outside any function blocks. Such variables are commonly known as **global variables** and can be accessed from any point in the program.

Lifetime: Global variables are created at the **beginning of program execution** and remain in existence throughout the **period of execution of the program**. These variables are **known to all the functions** in the program and **are accessible** to them for usage. Global variables are not limited to a particular function so they exist even when a function calls another function. These variables retain their value so that they can be used from every function in the program.

Place of Declaration: The Global variables are **declared outside all the functions including main().** It is always recommended to declare them on **top of the program code.**

Name conflict: If we have a variable declared in a function that has **same name as that of the global variable**, then the function will use the **local variable declared within** it and **ignore** the global variable. Consider the following program,

```
#include <stdio.h >
int x = 10;
void print();

void main()
{
        printf("\n The value of x in the main() = %d", x);
        int x = 2;
        printf("\n The value of local variable x in the main() = %d", x);
        print();
}

void print()
{
        printf("\n The value of x in the print() = %d", x);
}
```

Output:

```
The value of x in the main() = 10
The value of local variable x in the main()= 2
The value of x in the print () = 10
```

3.18.4 File Scope

✓ When a global variable is **accessible until the end of the file**, the Variable is said to have **file scope**. To allow a variable to have file scope, declare that variable with the static keyword before specifying its data type:

```
static int x;
```

✓ A global static variable can be used **anywhere from the file** in which it is declared but it is **not accessible by any other file.**

3.19 Storage Classes

✓ **Storage class defines the scope (visibility) and lifetime of variables and/or functions declared within a C program.**

✓ In addition to this, the storage class gives the following information about the variable or the function.

- The storage class of a function or a variable determines the **part of memory where storage space will be allocated for that variable or function** (whether the variable function will be stored in a **register or in RAM**).

- It specifies **how long the storage allocation will continue to exist** for that function or variable.
- It specifies the **scope of the variable or function**.
- It specifies whether the variable or function has **internal, external, or no linkage.**
- It specifies whether the variable will be **automatically initialized to zero or to any indeterminate value.**

✓ C supports four storage classes: **automatic, register, external, and static.**

✓ The general syntax for specifying the storage class of a variable can be given as:

<storage_class_specifier> <data type > <variable name>

3.19.1 auto Storage Class

✓ The auto storage class specifier is used to explicitly declare a variable with **automatic storage**.

✓ It is the **default storage class** for variables declared inside a block.

✓ For example, if we write

```
auto int x;
```

then x is an integer that has automatic storage. It is deleted when the block in which x is declared is exited.

✓ The auto storage doss can be used to declare **variables in a block or the names of function parameters.**

✓ Important things to remember about the variables declared with **auto** storage class are as follows :

- **All the variables declared within a function** belong to automatic storage class by default.
- They should be declared at the **start of the program block**, right after the opening curly brackets {.
- **Memory** for the variable is **automatically allocated upon entry to a block** and freed automatically upon exit from that block.
- The scope of the variable is **local to the block** in which it is declared.
- Every time the block is entered, the **variable is initialized with the values declared.**
- The auto variables are stored in the **primary memory of the computer.**
- If auto variables are **not initialized** at the time of declaration, then they contain **some garbage value.**

3.19.2 register Storage Class

✓ When a variable is declared using register as its storage class, it is stored in **a CPU register** instead of RAM.

✓ Since the variable is stored in a register, the **maximum size of the variable is equal to the register size.**

✓ A register variable is declared in the following manner:

```
register int x;
```

✓ Register variables are used when **quick access to the variable** is needed.

✓ Each time the **block is entered**, the register variables defined in that **block are accessible** and the moment that **block is exited**, the variables become **no longer accessible for use**.

3.19.3 extern Storage Class

✓ The **extern** storage class is used to give a **reference of a global variable** that is visible to all the program files.

✓ Such global variables are declared like any other variables in one of the program files. .

✓ To declare a variable x as extern write,

```
extern int x;
```

- ✓ External variables may be declared outside any function source code file as any other variable is declared.
- ✓ Usually external variables are declared and defined in the **beginning of a source file.**
- ✓ **Memory is allocated for the external variables when the program begins execution and remains allocated until the program terminates.**
- ✓ In case if the external variable is not initialized, then it will be **initialized to zero by default.**
- ✓ External variables have **global scope,** i.e. these variables are visible and accessible from all the functions in the program.

3.19.4 static Storage Class

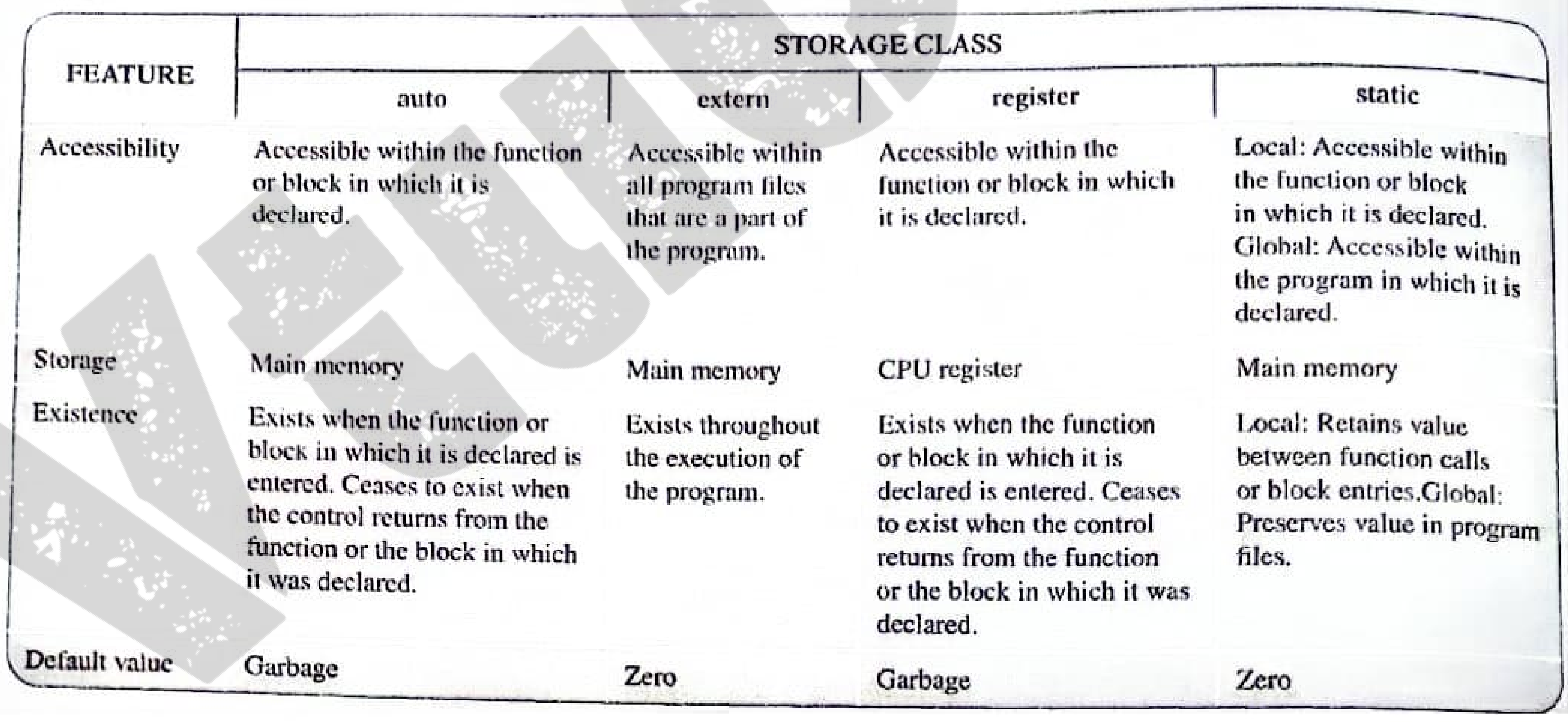
- ✓ static is the **default storage class for all global variables.**
- ✓ Static variables have a **lifetime over the entire program**. i.e., memory for the static variables is allocated when the program begins running and is freed when the program terminates.
- ✓ To declare an integer x as static, write

```
static int x = 10;
```

  Here x is a local static variable.
- ✓ Static local variables when defined within a function are **initialized at the runtime.**
- ✓ The static variables are initialized just once, when defined within a function it is not re-initialized when the function is called again and again.
- ✓ When a **static variable is not explicil1y initialized** by the programmer, then it is automa1ically **initialized to zero** when memory is allocated for it

Comparison of Storage Classes

Table 11.2 Comparison of storage classes



FEATURE	STORAGE CLASS			
	auto	extern	register	static
Accessibility	Accessible within the function or block in which it is declared.	Accessible within all program files that are a part of the program.	Accessible within the function or block in which it is declared.	Local: Accessible within the function or block in which it is declared. Global: Accessible within the program in which it is declared.
Storage	Main memory	Main memory	CPU register	Main memory
Existence	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Exists throughout the execution of the program.	Exists when the function or block in which it is declared is entered. Ceases to exist when the control returns from the function or the block in which it was declared.	Local: Retains value between function calls or block entries.Global: Preserves value in program files.
Default value	Garbage	Zero	Garbage	Zero

3.20 Recursion

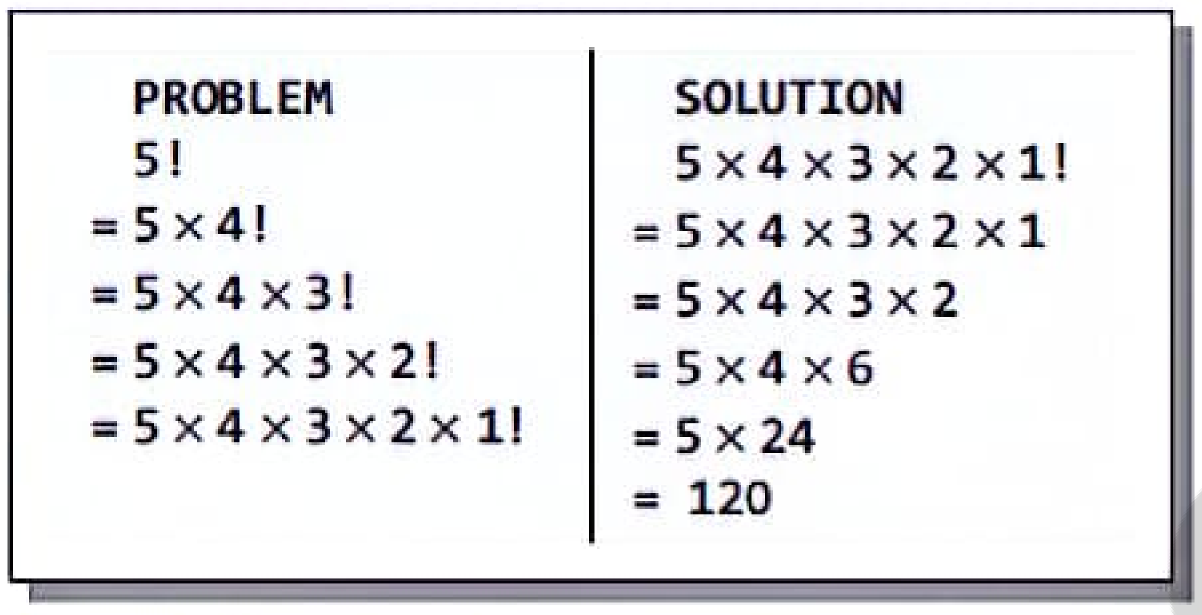
- ✓ **"The process in which a function calls itself again and again is called as Recursion".**
- ✓ **A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.**

✓ Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are:

- **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Ex: To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

In other words, $n! = n \times (n-1)!$



PROBLEM	SOLUTION
$5!$	$5\times4\times3\times2\times1!$
$=5\times4!$	$=5\times4\times3\times2\times1$
$=5\times4\times3!$	$=5\times4\times3\times2$
$=5\times4\times3\times2!$	$=5\times4\times6$
$=5\times4\times3\times2\times1!$	$=5\times24$
	$=120$

Figure 7.27 Recursive factorial function

✓ Every recursive function must have a base case and a recursive case. For the factorial function,

- **Base case** is when n = 1, because if n = 1, the result will be 1 as 1! = 1.
- **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as:

  factorial(n) = n × factorial (n–1)

Example:

1. Write a C program to calculate factorial of a given number.

```
#include<stdio.h>
int factorial(int n)
{
        if(n==1)
                return 1;
        else
                return (n*fact(n-1));
}

void main()
{
        int n,fact;
        printf("Enter a number:");
        scanf("%d",&n);
        fact=factorial(n);
        printf("\nFactorial of given number=%d",fact);
}
```

Output:

```
Enter a number: 5
Factorial of given number =120
```

Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using the **Euclid's algorithm** that states:

$$\text{GCD}(a, b) = \begin{cases} b, \text{ if } b \text{ divides } a \\ \text{GCD}(b, a \bmod b), \text{ otherwise} \end{cases}$$

Working

Assume `a = 62` and `b = 8`

```
GCD(62, 8)
   rem = 62 % 8 = 6
   GCD(8, 6)
          rem = 8 % 6 = 2
          GCD(6, 2)
                   rem = 6 % 2 = 0
          Return 2
   Return 2
Return 2
```

2. Write a C program to calculate the GCD of two numbers using recursive functions.

```
#include <stdio.h>
int GCD(int x, int y)
{
        int rem;
        rem = x%y;
        if(rem==0)
                return y;
        else
                return (GCD(y, rem));
}

void main()
{
        int x, y, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &x, &y);
        res = GCD(x, y);
        printf("\n GCD = %d", x, y, res);
}
```

Output:

```
Enter the two numbers: 8 12
GCD = 4
```

3. Write a C program to find the Fibonacci series using recursive function.

```
#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));
}
```

```
void main()
{
    int n, i, res;
    printf("Enter a value for n:");
    scanf("%d",&n);
    printf(" The Fibonacci series is: \n");
    for ( i =0; i < n ; i ++)
    {
        res = fibonacci(i);
        printf("%d\n", res);
    }
}
```

Output:

```
Enter a value for n:
5
The Fibonacci series is:
0
1
1
2
3
```

3.21 Function Parameters

- ✓ **"The list of variables defined in the function header within the parenthesis are called Function parameters".**
- ✓ There are 2 types of parameters in 'C' functions.
  - i. Actual parameters
  - ii. Formal parameters

i. Actual or Real Parameters

- ✓ **The variables that are used when a function is invoked are called actual parameters.**
- ✓ Actual parameters are used in the Calling function when a function is invoked.
- ✓ Actual parameters send values or addresses to the formal parameters. Formal parameters receive them and use the same values.
- ✓ Actual parameters can be constants, variables or expressions.

Ex: res=add(m,n);

ii. Formal or Dummy Parameters

- ✓ **The variables defined in the function header or function definition are called formal parameters.**
- ✓ All the variables should be separately declared and each declaration must be separated by commas.
- ✓ The formal parameters receive values form the actual parameters.
- ✓ If the formal parameters receive the address from the actual parameters, then they should be declared as pointers.
- ✓ The formal parameters should be only variables. Expressions and constants are not allowed.

Ex: int add(int a,int b);

Example Program: C Program to define actual and formal parameters.

```
#include<stdio.h>

int add(int a,int b)                        // Formal Parameters a, b
{
        int sum;
        sum=a+b;
        return sum;
}
```

```
void main()
{
        int m,n,res;
        printf("Enter the values for m,n\n");
        scanf("%d%d",&m,&n);
        res=add(m,n);                       // Actual parameters m,n
        printf("Sum=%d\n",res);
}
```

MODULE 3
Chapter2 - Arrays

3.1 Introduction

✓ **An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).**
✓ **The subscript is an ordinal number which is used to identify an element of the array.**

3.2 Declaration of Arrays

✓ An array must be declared before being used.
✓ Arrays are declared using the following **syntax:**
    type name[size];
✓ Declaring an array means specifying the following:
- **Data type**—the kind of values it can store, for example, int, char, float, double, or any other valid data type..
- **Name**—to identify the array.
- **Size**—the maximum number of values that the array can hold. i.e., the maximum number of elements that can be stored in the array.
  For example, if we write,
  int marks[10];

then the statement declares marks to be an array containing 10 elements. In C, the array index starts from zero. The first element will be stored in marks[0], second element in marks[1], and so on. Therefore, the last element, that is the 10th element, will be stored in marks[9]. Note that 0, 1, 2, 3 written within square brackets are the subscripts. In the memory, the array will be
stored as shown in Fig. 3.2.

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

Figure 3.2 Memory representation of an array of 10 elements

✓ Figure 3.3 shows how different types of arrays are declared.

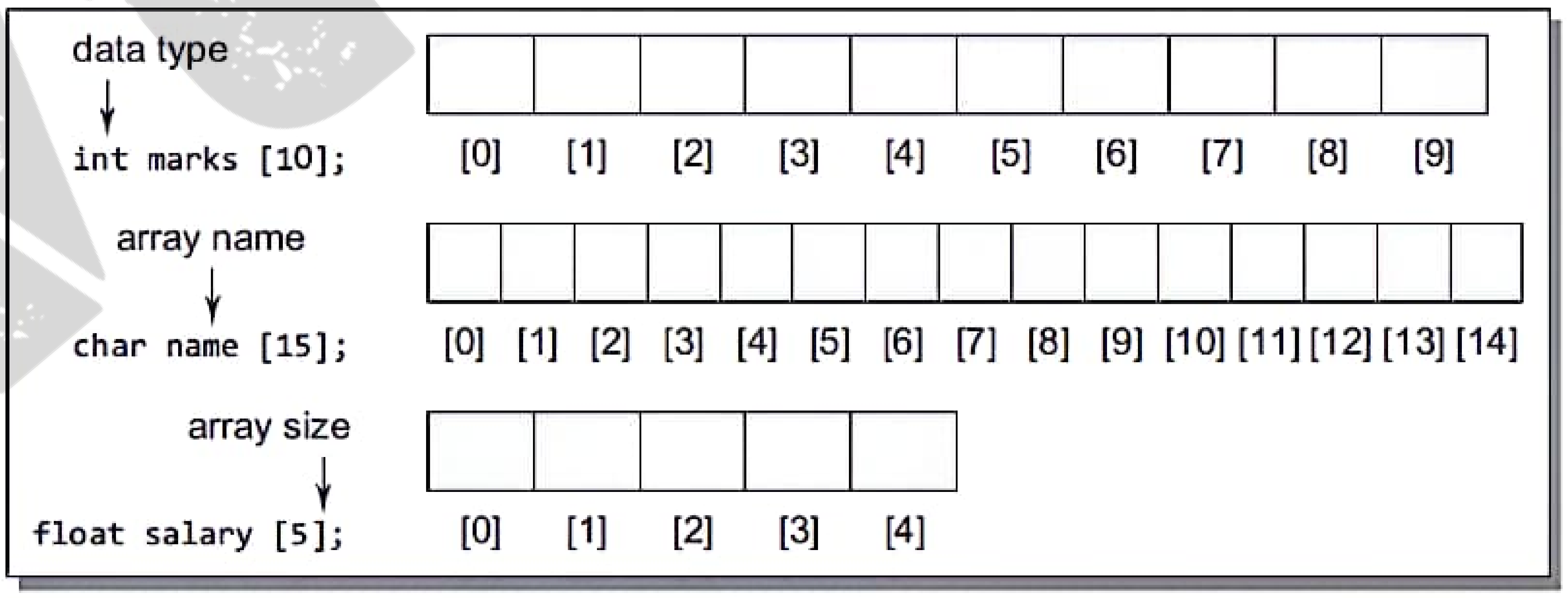


Figure 3.3 Declaring arrays of different data types and sizes

3.3 Accessing the Elements of an Array

- ✓ To access all the elements, **we must use a loop.** That is, **we can access all the elements of an array by varying the value of the subscript into the array.**
- ✓ But note that the subscript must be an integral value or an expression that evaluates to an integral value.
- ✓ As shown in Fig. 3.2, the first element of the array marks[10] can be accessed by writing marks[0]. Now to process all the elements of the array, we use a loop as shown in Fig. 3.4.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
      marks[i] = -1;
```

Figure 3.4 Code to initialize each element of the array to –1

- ✓ Figure 3.5 shows the result of the code shown in Fig. 3.4. The code accesses every individual element of the array and sets its value to –1. In the for loop, first the value of marks[0] is set to –1, then the value of the index (i) is incremented and the next value, that is, marks[1] is set to –1. The procedure continues until all the 10 elements of the array are set to –1.

– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1	– 1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

3.3.1 Calculating the Address of Array Elements

- ✓ The **array name** is a symbolic reference to the **address of the first byte of the array.**
- ✓ When we **use the array name**, we are actually referring to the **first byte of the array.**
- ✓ The **subscript or the index** represents the **offset from the beginning of the array to the element being referenced.**
- ✓ That is, with just the **array name and the index**, **C can calculate the address of any element in the array.**
- ✓ Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient.
- ✓ The address of other data elements can simply be calculated using the base address. The formula to perform this calculation is,

  Address of data element, A[k] = BA(A) + w(k – lower_bound)

  Here, A is the array,

  k is the index of the element of which we have to calculate the address,

  BA is the base address of the array A, and

  w is the size of one element in memory, for example, size of int is 2.

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
marks[4] = 1000 + 2(4 – 0)
         = 1000 + 2(4) = 1008
```

3.3.2 Calculating the Length of an Array

- ✓ The length of an array is given by the number of elements stored in it.
- ✓ The general formula to calculate the length of an array is

  Length = upper_bound – lower_bound + 1

  where, upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

Example 3.2 Let `Age[5]` be an array of integers such that

```
Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
```

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

```
Length = upper_bound – lower_bound + 1
```

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 – 0 + 1 = 5`

3.4 Storing Values in Arrays

- ✓ When we declare an array, we are just allocating space for its elements; no values are stored in the array.
- ✓ There are three ways to store values in an array. First, to **initialize the array elements during declaration**; second, to **input values for individual elements from the keyboard**; third, to **assign values to individual elements**.
- ✓ This is shown in Fig. 3.6.

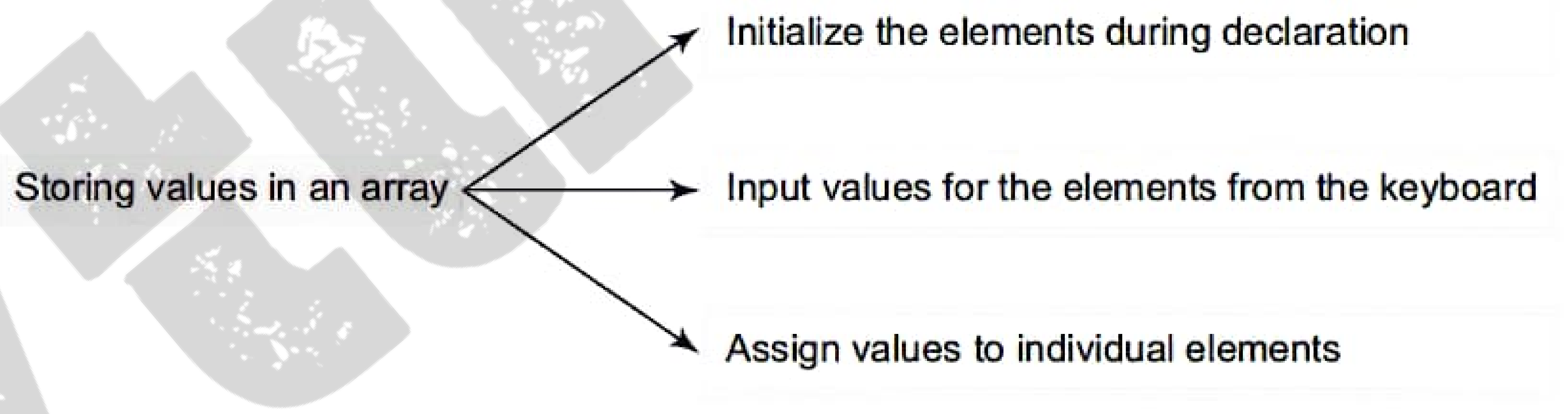

Storing values in an array
Initialize the elements during declaration
Input values for the elements from the keyboard
Assign values to individual elements

Figure 3.6 Storing values in an array

1. Initializing Arrays during Declaration

- ✓ The elements of an array can be initialized at the **time of declaration**, just as **any other variable.**
- ✓ When an array is initialized, we need **to provide a value for every element in the array.**
- ✓ Arrays are initialized by writing,

  type array_name[size]={list of values};

- ✓ Note that the values are written within curly brackets and every value is separated by a comma. It is a compiler error to specify more values than there are elements in the array. When we write,

  int marks[5]={90, 82, 78, 95, 88};

- ✓ An array with the name marks is declared that has enough space to store five elements. The first element, that is, marks[0] is assigned value 90. Similarly, the second element of the array, that is marks[1], is assigned 82, and so on. This is shown in Fig. 3.7.

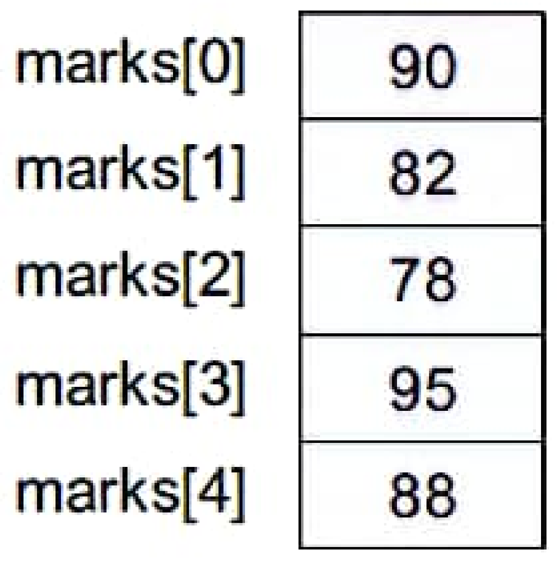


Figure 3.7 Initialization of array `marks[5]`

- ✓ While initializing the array at the time of declaration, the programmer may omit the size of the array. For example,

  int marks[ ]= {98, 97, 90};

  The above statement is absolutely legal. Here, the compiler will allocate enough space for all the initialized elements. Note that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros. Figure 3.8 shows the initialization of arrays.

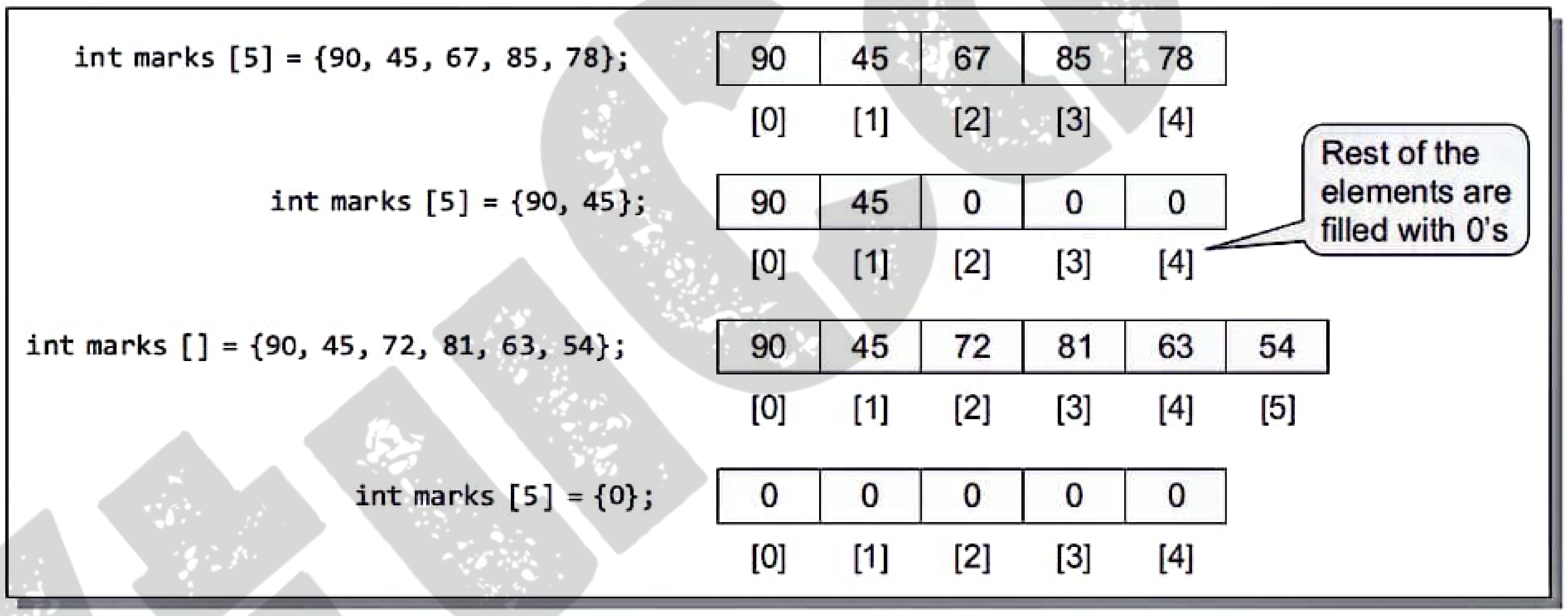


Figure 3.8 Initialization of array elements

2. Inputting Values from the Keyboard

- ✓ An array can be initialized by **inputting values from the keyboard.**
- ✓ In this method, a **while/do–while or a for loop** is executed to input the value for each element of the array. For example, look at the code shown in Fig. 3.9.

```
int i, marks[10];
for(i=0;i<10;i++)
        scanf("%d", &marks[i]);
```

Figure 3.9 Code for inputting each element of the array

- ✓ In the code, we start at the index i at 0 and input the value for the first element of the array. Since the array has 10 elements, we must input values for elements whose index varies from 0 to 9.

3. Assigning Values to Individual Elements

- ✓ The third way is to **assign values to individual elements of the array by using the assignment operator.**
- ✓ Any value that evaluates to the data type as that of the array can be assigned to the individual array element.
- ✓ A simple assignment statement can be written as

  marks[3] = 100;

  Here, 100 is assigned to the fourth element of the array which is specified as marks[3].
- ✓ To copy an array, you must copy the value of every element of the first array into the elements of the second array. Figure 3.10 illustrates the code to copy an array. In Fig. 3.10, the loop accesses each element of the first array and simultaneously assigns its value to the corresponding element of the second array. The index value i is incremented to access the next element in succession. Therefore, when this code is executed, arr2[0] = arr1[0], arr2[1] = arr1[1], arr2[2] = arr1[2], and so on.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
        arr2[i] = arr1[i];
```

Figure 3.10 Code to copy an array at the individual element level

- ✓ For example, if we want to fill an array with even integers (starting from 0), then we will write the code as shown in Fig. 3.11. In the code, we assign to each element a value equal to twice of its index, where the index starts from 0. So after executing this code, we will have arr[0] = 0, arr[1] = 2, arr[2] = 4, and so on.

```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
        arr[i] = i*2;
```

Figure 3.11 Code for filling an array with even numbers

3.5 Operations on Arrays

- ✓ There are a number of operations that can be performed on arrays.
- ✓ These operations include:
  - **Traversing an array**
  - **Inserting an element in an array**
  - **Searching an element in an array**
  - **Deleting an element from an array**
  - **Merging two arrays**
  - **Sorting an array in ascending or descending order**

3.5.1 Traversing an Array

- ✓ Traversing an array means **accessing each and every element of the array for a specific purpose.**
- ✓ Traversing the data elements of an array A can include **printing every element, counting the total number of elements, or performing any process on these elements**.

✓ The algorithm for array traversal is given in Fig. 3.12.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:       Apply Process to A[I]
Step 4:       SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

✓ In Step 1, we initialize the index to the lower bound of the array. In Step 2, a while loop is executed. Step 3 processes the individual array element as specified by the array name and index value. Step 4 increments the index value so that the next array element could be processed. The while loop in Step 2 is executed until all the elements in the array are processed, i.e., until I is less than or equal to the upper bound of the array.

Examples:

Write a program to read and display n numbers using an array.

```
#include <stdio.h>
void main()
{
        int i, n, a[10];
        printf("Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("Enter the array elements:\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("The array elements are:\n ");
        for(i=0;i<n;i++)
                printf("%d\n", a[i]);
}
```

Output

```
Enter the number of elements in the array: 5
Enter the array elements:
1
2
3
4
5
The array elements are:
1
2
3
4
5
```

3.5.2 Inserting an Element in an Array

✓ **If an element has to be inserted at the end of an existing array, then we just have to add 1 to the upper_bound and assign the value.** Here, we assume that the memory space allocated for the array is still available. For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements. But if it already has 10 elements, then we will not be able to add another element to it.

✓ Figure 3.13 shows an algorithm to insert a new element to the end of an array. In Step 1, we increment the value of the upper_bound. In Step 2, the new value is stored at the position pointed by the upper_bound.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Figure 3.13 Algorithm to append a new element to an existing array

✓ For example, let us assume an array has been declared as

int marks[60];

The array is declared to store the marks of all the students in a class. Now, suppose there are 54 students and a new student comes and is asked to take the same test. The marks of this new student would be stored in marks[55]. Assuming that the student secured 68 marks, we will assign the value as

marks[55] = 68;

✓ Consider an array whose elements are arranged in ascending order. Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array. To do this, we must first find the location where the new element will be inserted and then move all the elements (that have a value greater than that of the new element) one position to the right so that space can be created to store the new value.

Example 3.3 `Data[]` is an array that is declared as `int Data[20];` and contains the following values:

`Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};`

(a) Calculate the length of the array.
(b) Find the `upper_bound` and `lower_bound`.
(c) Show the memory representation of the array.
(d) If a new data element with the value 75 has to be inserted, find its position.
(e) Insert a new data element 75 and show the memory representation after the insertion.

Solution

(a) Length of the array = number of elements
Therefore, `length of the array = 10`
(b) By default, `lower_bound = 0` and `upper_bound = 9`
(c)

12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6th position will be moved one position towards the right to accommodate the new value

(e)

12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

Algorithm to Insert an Element in the Middle of an Array

✓ The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are

(a) A, the array in which the element has to be inserted
(b) N, the number of elements in the array
(c) POS, the position at which the element has to be inserted
(d) VAL, the value that has to be inserted

✓ In the algorithm given in Fig. 3.14, in Step 1, we first initialize I with the total number of elements in the array. In Step 2, a while loop is executed which will move all the elements having an index greater than POS one position towards right to create space for the new element. In Step 5, we increment the total number of elements in the array by 1 and finally in Step 6, the new value is inserted at the desired position.

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:         SET A[I + 1] = A[I]
Step 4:         SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

Initial `Data[]` is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling `INSERT(Data, 6, 3, 100)` will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

3.5.3 Deleting an Element from an Array

✓ **Deleting an element from an array means removing a data element from an already existing array.**

✓ **If the element has to be deleted from the end of the existing array, then we just have to subtract 1 from the upper_bound.**

✓ Figure 3.15 shows an algorithm to delete an element from the end of an array.

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Figure 3.15 Algorithm to delete the last element of an array

✓ For example, if we have an array that is declared as

int marks[60];

The array is declared to store the marks of all the students in the class. Now, suppose there are 54 students and the student with roll number 54 leaves the course. The score of this student was stored in marks[54]. We just have to decrement the upper_bound. Subtracting 1 from the upper_bound will indicate that there are 53 valid data in the array.

✓ Consider an array whose elements are arranged in ascending order. Now, suppose an element has to be deleted, probably from somewhere in the middle of the array. To do this, we must first find the location from where the element has to be deleted and then move all the elements (having a value greater than that of the element) one position towards left so that the space vacated by the deleted element can be occupied by rest of the elements.

Example 3.4 `Data[]` is an array that is declared as `int Data[10];` and contains the following values:

`Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};`

(a) If a data element with value 56 has to be deleted, find its position.

(b) Delete the data element 56 and show the memory representation after the deletion.

Solution

(a) Since the elements of the array are stored in ascending order, we will compare the value that has to be deleted with the value of every element in the array. As soon as `VAL = Data[I]`, where `I` is the index or subscript of the array, we will get the position from which the element has to be deleted. For example, if we see this array, here `VAL = 56`. `Data[0] = 12` which is not equal to 56. We will continue to compare and finally get the value of `POS = 4`.

(b)

12	23	34	45	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]

Algorithm to delete an element from the middle of an array

✓ The algorithm DELETE will be declared as DELETE(A, N, POS). The arguments are:

(a) A, the array from which the element has to be deleted

(b) N, the number of elements in the array

(c) POS, the position from which the element has to be deleted

✓ Figure 3.16 shows the algorithm in which we first initialize I with the position from which the element has to be deleted. In Step 2, a while loop is executed which will move all the elements having an index greater than POS one space towards left to occupy the space vacated by the deleted element. In Step 5, we decrement the total number of elements in the array by 1.

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:         SET A[I] = A[I + 1]
Step 4:         SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Figure 3.16 Algorithm to delete an element from the middle of an array

✓ Calling DELETE (Data, 6, 2) will lead to the following processing in the array.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

Figure 3.17 Deleting elements from an array

3.5.4 Merging Two Arrays

✓ **Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.** This operation is shown in Fig 3.18.

Array 1-	90	56	89	77	69							
Array 2-	45	88	76	99	12	58	81					
Array 3-	90	56	89	77	69	45	88	76	99	12	58	81

Figure 3.18 Merging of two unsorted arrays

✓ If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult. The task of merging can be explained using Fig. 3.19.

Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

Figure 3.19 Merging of two sorted arrays

3.5.5 Searching for a Value in an Array

✓ **Searching means to find whether a particular value is present in an array or not.**

✓ If the **value is present** in the array, then **searching is said to be successful** and the searching process gives the **location of that value in the array**. However, if the **value is not present** in the array, the searching process displays an **appropriate message** and in this case **searching is said to be unsuccessful**.

- ✓ There are two popular methods for searching the array elements: **linear search and binary search.**
- ✓ If the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

1. Linear Search

- ✓ Linear search, also called as **sequential search**, is a very simple method used for searching an array for a particular value.
- ✓ **It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.**
- ✓ Linear search is mostly used to search an **unordered list of elements** (array in which data elements are not sorted).
- ✓ For example, if an array A[] is declared and initialized as, int A[ ] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).
- ✓ Figure 14.1 shows the algorithm for linear search. In Steps 1 and 2 of the algorithm, we initialize the value of POS and I. In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array). In Step 4, a check is made to see if a match is found between the current array element and VAL. If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Figure 14.1 Algorithm for linear search

Example:

Write a program to search an element in an array using the linear search technique.

#include <stdio.h>

✓ Like C libraries, **programmers can also write their own functions and use them** from different points in the main program or any other program that needs its functionalities.
✓ When a big program is broken into comparatively smaller functions, then different programmers working on that project can **divide the workload by writing different functions.**

3.13 Using Functions

While using functions, we will be using the following terminologies:

✓ A function **f** that **uses another function g** is known as the **calling function**, and **g** is known as the **called function**.
✓ The **inputs that a function takes** are known as **arguments**.
✓ When a **called function returns some result back** to the calling function, it is said to **return** that **result.**
✓ The **calling function may or may not pass parameters** to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
✓ **Function declaration** is a declaration statement that identifies **a function's name, a list of arguments that it accepts, and the type of data it returns.**
✓ **Function definition** consists of a **function header** that identifies the function, followed by the **body of the function** containing the executable code for that function.

3.14 Types of Functions

i. Library Functions/Pre-Defined/ Built-in Functions

✓ C Library of C Compiler has a collection of various functions which perform some standard and pre-defined tasks.
✓ These functions written by designers of C Compilers are called as Library functions/Pre-Defined/Built-in functions.

Ex: sqrt(n)- computes square root of n.
pow(x,y)- computes x^y.
printf()- used to print the data on the screen.
scanf()- used to read the data from the keyboard.
abs(x)- computes absolute value of x.

ii. User-Defined/ Programmer Defined Functions

✓ The functions written by the programmer/user to do the specific tasks is called User-Defined/ Programmer Defined Functions.
✓ main( ) is the user defined function.

3.15 Elements of User-Defined Functions

✓ The three elements of user-defined functions are shown below:
i. Function Prototype/Declaration
ii. Function Definition
iii. Function Call

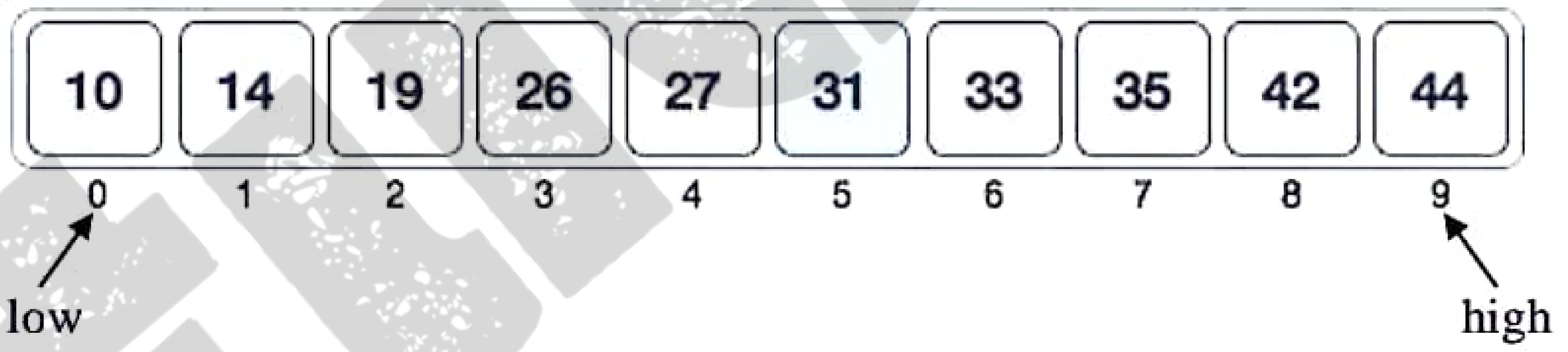
(c) Finally, if KEY is not present in the array, then eventually, HIGH will be less than LOW. When this happens, the algorithm will terminate and the search will be unsuccessful. Figure 14.2 shows the algorithm for binary search.

```
BINARY_SEARCH(A, lower_bound, upper_bound, KEY)
Step 1: [INITIALIZE] SET LOW = lower_bound
        HIGH = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while LOW <= HIGH
Step 3: SET MID = (LOW + HIGH)/2
Step 4: IF KEY = A[MID]
                SET POS = MID
                PRINT POS
                Go to Step 6
        ELSE IF KEY > A[MID]
                SET LOW = MID + 1
        ELSE
                SET HIGH = MID - 1
        [END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
                PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

How Binary Search Works?

✓ For a binary search to work, it is mandatory for the target array to be sorted. The following is our sorted array and let us assume that we need to search the location of key value **31** using binary search.

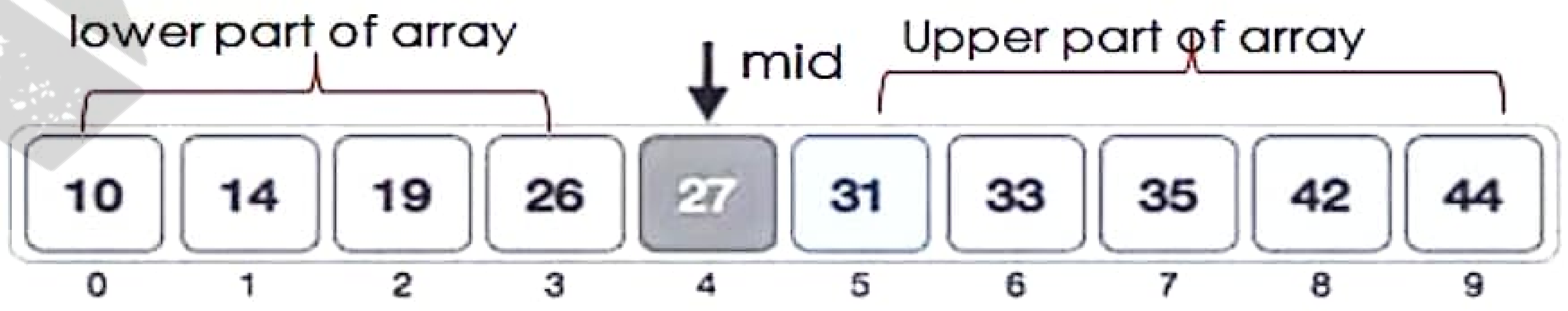
Here n=10, key=31



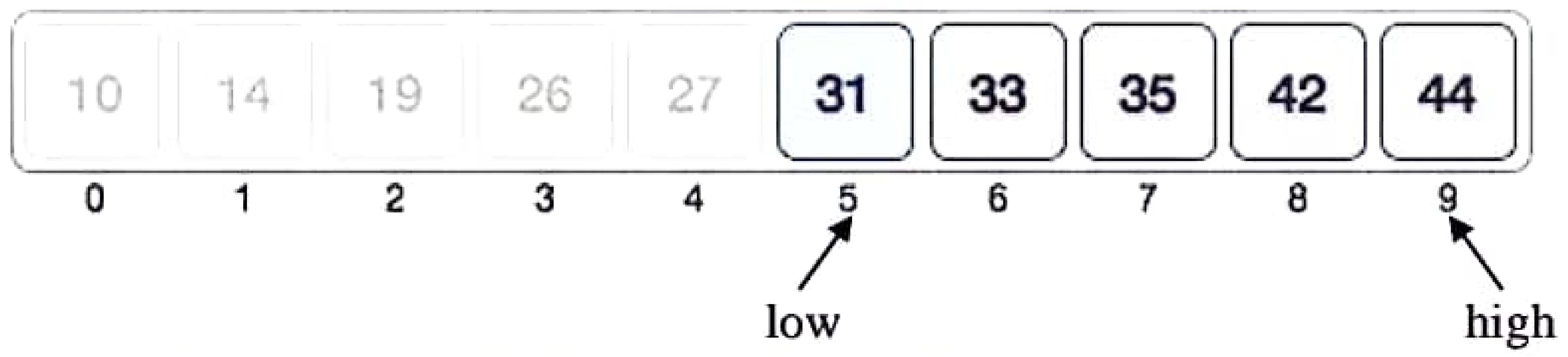
First, we shall determine middle position of the array by using this formula −

mid = (low + high) / 2

Here it is, mid=(0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



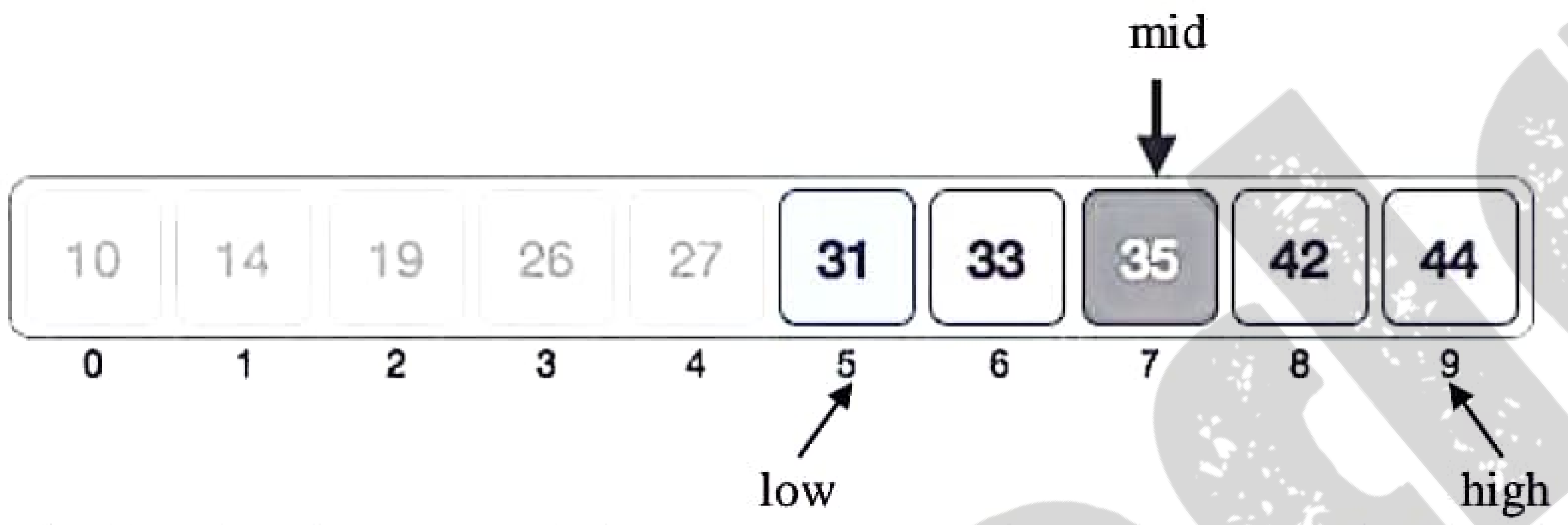
✓ Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. Since the key element is greater than the middle element, we should search the key element in the upper part of the array



We change our low to mid + 1 and find the new mid value again.

low = mid + 1=4+1=5
mid = (low + high) /2= (5+9)/2 =7

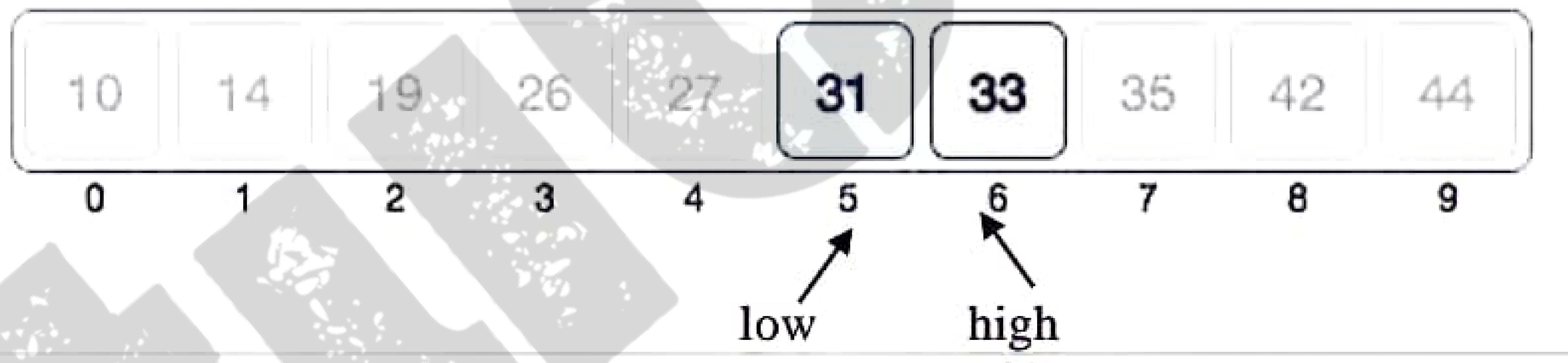
✓ Our new mid is 7 now. We compare the value stored at location 7 with our key value 31.



✓ The value stored at location 7 is not a match; rather it is more than what we are looking for. Since the key element is less than the middle element, we should search the key element in the lower part of the array.

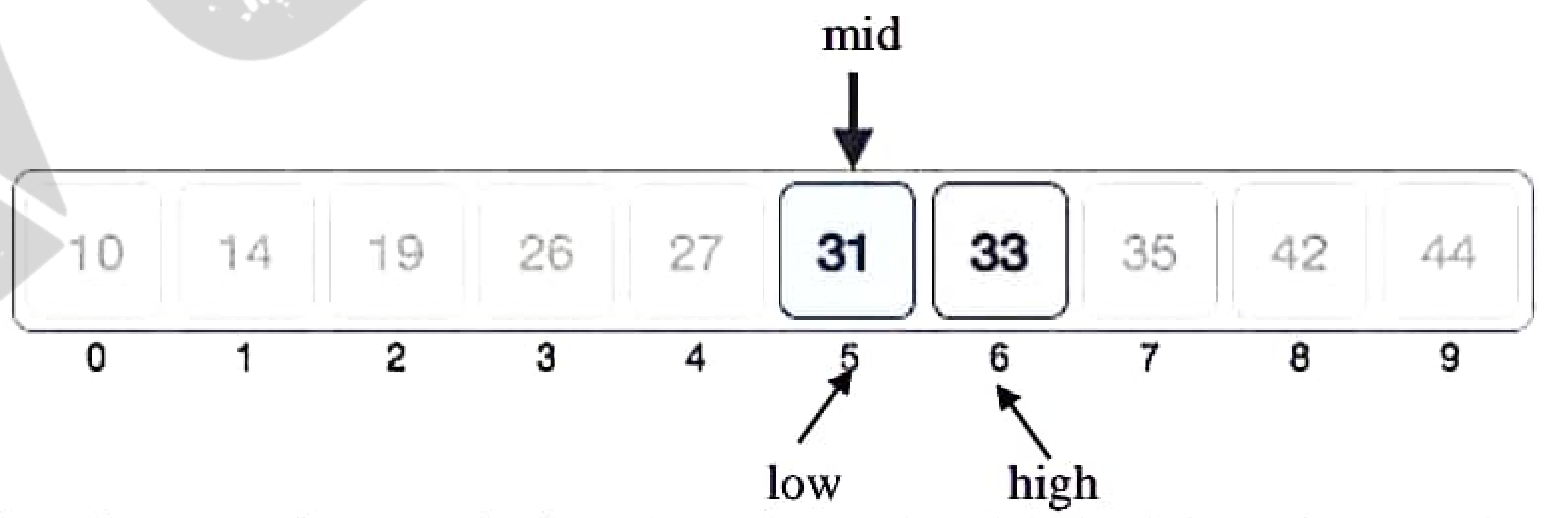
We change our high to mid - 1 and find the new mid value again.

high=mid-1 = mid -1=7-1=6

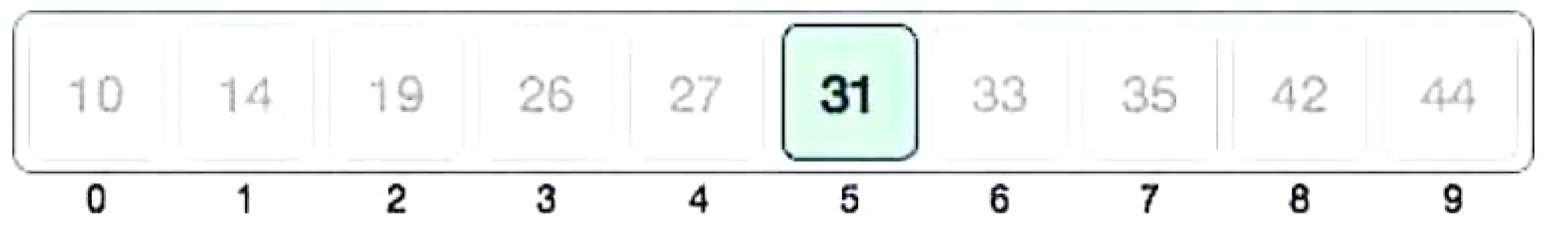


mid = (low + high) / 2 = (5+6)/2 =5

✓ Hence, we calculate the mid again. This time it is 5.



✓ We compare the value stored at location 5 with our key value. We find that it is a match.



✓ We conclude that the key value 31 is stored at position mid+1= 5+1=6.

- ✓ Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Example: C Program to search key elements in array using binary search algorithms.

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

void main()
{
        int  i, low, high, mid, n, key, a[100];
        printf("Enter number of elements in array:\n");
        scanf("%d",&n);
        printf("Enter integer numbers in ascending order:\n");
        for (i = 0; i< n; i++)
        scanf("%d",&a[i]);
        printf("Enter value to Search\n");
        scanf("%d", &key);
        low = 0;
        high = n - 1;
        while (low <= high)
        {
                mid = (low+high)/2;
                if (key == a[mid])
                {
                  printf("%d found at location %d.\n", key, mid+1);
                  exit(0);
                }
                if (key > a[mid] )
                        low = mid + 1;
                if (key < a[mid])
                        high = mid - 1;
        }
        printf(" %d is Not found! \n", key);

}
```

```
Output:
Enter number of elements
in array:
5
Enter integer numbers in
ascending order:
10
25
35
50
65
Enter a number to search:
65
65 is present at location 5
```

3.6 Passing Arrays to Functions

- ✓ Like variables of other data types, we can also pass an array to a function.
- ✓ In some situations, you may want to pass **individual elements of the array**; while in other situations, you may want to pass the **entire array** as shown in Fig. 3.20.

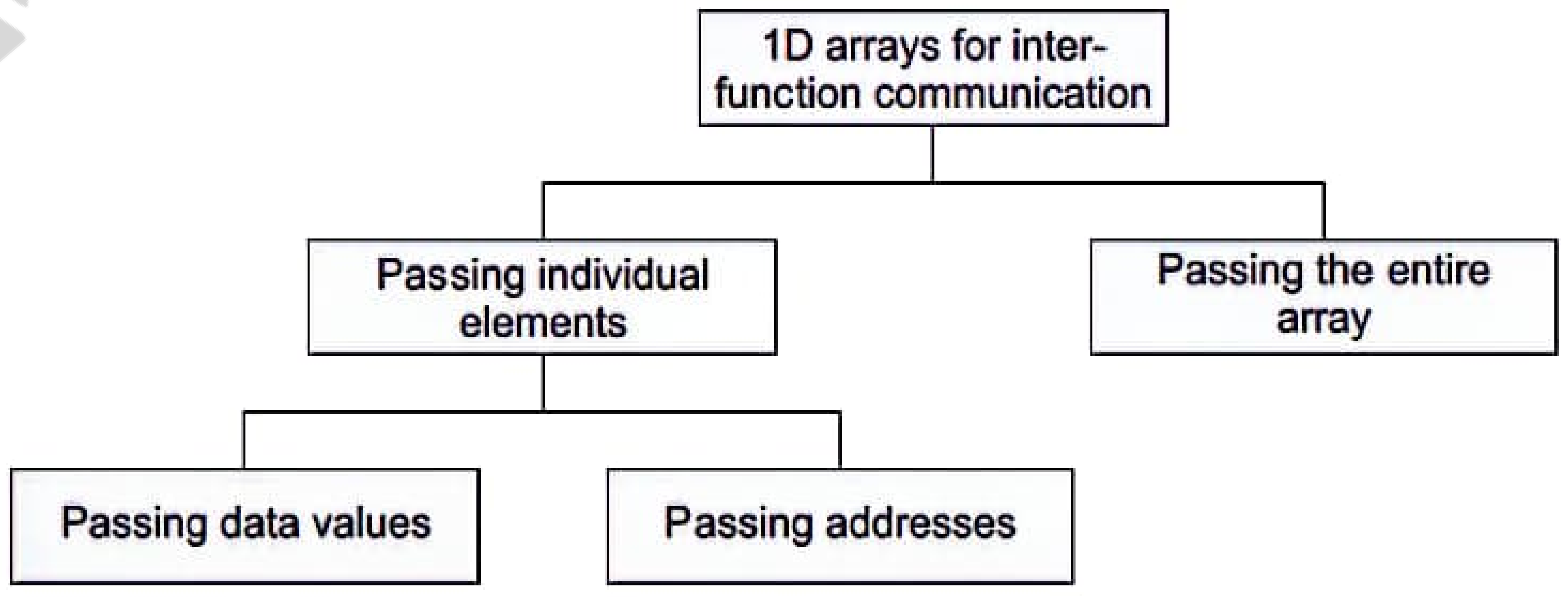

1D arrays for inter-function communication
Passing individual elements
Passing the entire array
Passing data values
Passing addresses

Figure 3.20 One dimensional arrays for inter-function communication

3.6.1 Passing Individual Elements

- ✓ The individual elements of an array can be passed to a function by passing either their **data values or addresses.**

1. Passing Data Values

- ✓ Individual elements can be passed in the same manner as we **pass variables of any other data type.**
- ✓ The condition is just that the **data type of the array element must match with the type of the function parameter.**
- ✓ Look at Fig. 3.21(a) which shows the code to pass an individual array element by passing the data value.

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(arr[3]);
}
```

Called function

```
void func(int num)
{
        printf("%d", num);
}
```

Figure 3.21(a) Passing values of individual array elements to a function

- ✓ In the above example, only one element of the array is passed to the called function. This is done by using the index expression. Here, arr[3] evaluates to a single integer value.

2. Passing Addresses

- ✓ Like ordinary variables, we can pass the **address of an individual array element by preceding the indexed array element with the address operator**.
- ✓ Therefore, to pass the address of the fourth element of the array to the called function, we will write &arr[3].
- ✓ However, in the called function, the value of the array element must be accessed using the indirection (*) operator. Look at the code shown in Fig. 3.21(b).

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(&arr[3]);
}
```

Called function

```
void func(int *num)
{
        printf("%d", *num);
}
```

Figure 3.21(b) Passing addresses of individual array elements to a function

3.6.2 Passing the Entire Array

- ✓ In C the array name refers to the first byte of the array in the memory.
- ✓ **The address of the remaining elements in the array can be calculated using the array name and the index value of the element.**
- ✓ Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array.
- ✓ Figure 3.22 illustrates the code which passes the entire array to the called function.

Calling function

```
main()
{
        int arr[5] ={1, 2, 3, 4, 5};
        func(arr);
}
```

Called function

```
void func(int arr[5])
{
        int i;
        for(i=0;i<5;i++)
                printf("%d", arr[i]);
}
```

Figure 3.22 Passing entire array to a function

3.7 Two-Dimensional Arrays

- ✓ **A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column.**
- ✓ The C compiler treats a two-dimensional array as an array of one-dimensional arrays.
- ✓ Figure 3.26 shows a two-dimensional array which can be viewed as an array of arrays.

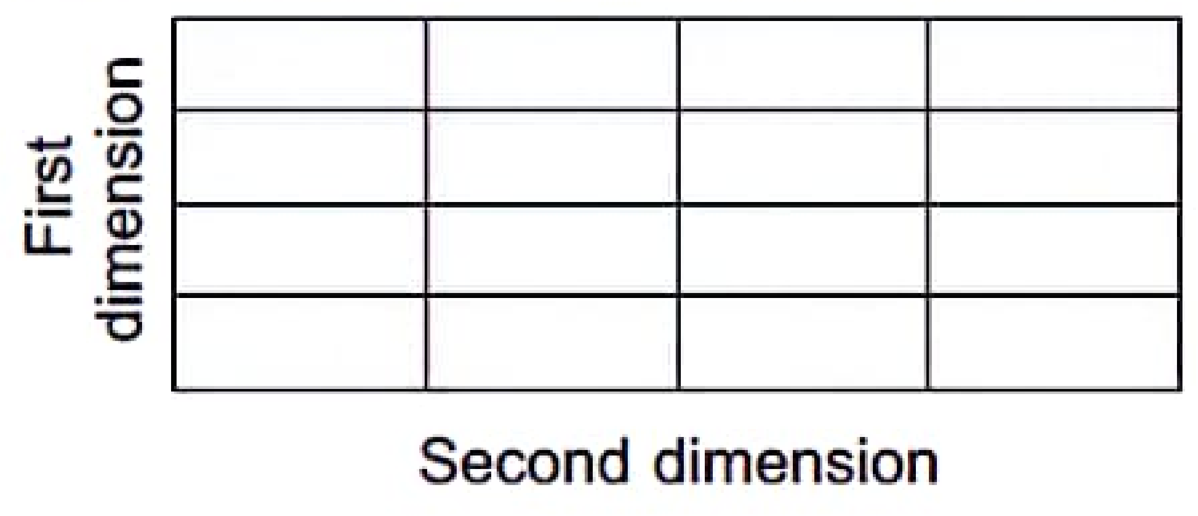


Figure 3.26 Two-dimensional array

3.7.1 Declaring Two-dimensional Arrays

- ✓ Any array must be declared before being used. The declaration statement tells the compiler the **name of the array, the data type of each element in the array, and the size of each dimension.**
- ✓ A two-dimensional array is declared as:

  data_type array_name[row_size][column_size];
- ✓ For example, if we want to store the marks obtained by three students in five different subjects, we can declare a two dimensional array as:

  int marks[3][5];
- ✓ In the above statement, a two-dimensional array called marks has been declared that has m(3) rows and n(5) columns. The first element of the array is denoted by marks[0][0], the second element as marks[0][1], and so on. Here, marks[0][0] stores the marks obtained by the first student in the first subject, marks[1][0] stores the marks obtained by the second student in the first subject.
- ✓ The pictorial form of a two-dimensional array is shown in Fig. 3.27.

Rows / Columns	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Figure 3.27 Two-dimensional array

- ✓ Hence, we see that a 2D array is treated as a collection of 1D arrays. Each row of a 2D array corresponds to a 1D array consisting of n elements, where n is the number of columns. To understand this, we can also see the representation of a two-dimensional array as shown in Fig. 3.28.

marks[0] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]
marks[1] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]
marks[2] –	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

Figure 3.28 Representation of two-dimensional array `marks[3][5]`

✓ There are two ways of storing a two-dimensional array in the memory. The first way is the **row major order** and the second is the **column major order**.

✓ In a **row major order, the elements of the first row are stored before the elements of the second and third rows.** That is, the elements of the array are stored row by row where n elements of the first row will occupy the first n locations. This is illustrated in Fig. 3.29.

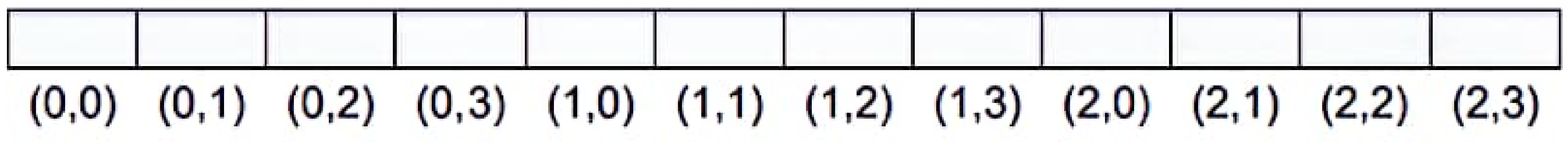


Figure 3.29 Elements of a 3 × 4 2D array in row major order

✓ However, when we store the elements in a **column major order, the elements of the first column are stored before the elements of the second and third column.** That is, the elements of the array are stored column by column where m elements of the first column will occupy the first m locations. This is illustrated in Fig. 3.30.

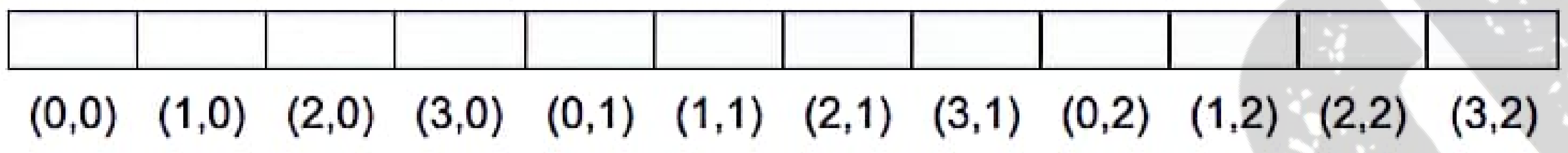


Figure 3.30 Elements of a 4 × 3 2D array in column major order

✓ If the array elements are stored in column major order,

Address(A[I][J]) = Base_Address + w{M (J – 1) + (I – 1)}

And if the array elements are stored in row major order,

Address(A[I][J]) = Base_Address + w{N (I – 1) + (J – 1)}

where w is the number of bytes required to store one element, N is the number of columns, M is the number of rows, and I and J are the subscripts of the array element.

Example 3.5 Consider a 20 × 5 two-dimensional array `marks` which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, `marks[18][4]` assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J])  = Base_Address + w{N (I – 1) + (J – 1)}
Address(marks[18][4]) = 1000 + 2 {5(18 – 1) + (4 – 1)}
                      = 1000 + 2 {5(17) + 3}
                      = 1000 + 2 (88)
                      = 1000 + 176 = 1176
```

3.7.2 Initializing Two-Dimensional Arrays

✓ A two-dimensional array is initialized in the **same way as a one-dimensional array is initialized**. For example,

int marks[2][3]={90, 87, 78, 68, 62, 71};

✓ Note that the initialization of a two-dimensional array is done row by row. The above statement can also be written as:

int marks[2][3]={{90,87,78},{68, 62, 71}};

✓ The above two-dimensional array has two rows and three columns. First, the elements in the first row are initialized and then the elements of the second row are initialized. Therefore,

marks[0][0] = 90 marks[0][1] = 87 marks[0][2] = 78

marks[1][0] = 68 marks[1][1] = 62 marks[1][2] = 71

- ✓ In case of one-dimensional arrays, we have discussed that if the array is completely initialized, we may omit the size of the array. The same concept can be applied to a two-dimensional array, except that only the **size of the first dimension can be omitted**. Therefore, the declaration statement given below is valid.

  int marks[][3]={{90,87,78},{68, 62, 71}};
- ✓ In order to initialize the **entire two-dimensional array to zeros**, simply specify the first value as zero. That is,

  int marks[2][3] = {0};
- ✓ The **individual elements** of a two-dimensional array can be initialized using the **assignment operator** as shown here.

  marks[1][2] = 79;
  or
  marks[1][2] = marks[1][1] + 10;

3.7.3 Accessing the Elements of Two-dimensional Arrays

- ✓ **The elements of a 2D array are stored in contiguous memory locations.**
- ✓ Since the two-dimensional array contains two subscripts, we will use **two for loops to scan the elements**.
- ✓ The first for loop will scan each row in the 2D array and the second for loop will scan individual columns for every row in the array.

Example:
Write a C program to read and print the elements of a 2D array.

```
#include <stdio.h>

void main()
{
        int arr[2][2], i, j, m, n;
        printf("Enter the size of the array:");
        scanf("%d%d",&m,&n);
        printf("Enter the elements of the array:\n);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d", &arr[i][j]);
                }
        }
        printf("The elements of the array are:\n);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t", arr[i][j]);
                }
                printf("\n");
        }
}
```

3.8 Operations on Two-Dimensional Arrays

- ✓ Two-dimensional arrays can be used to **implement the mathematical concept of matrices**.
- ✓ In mathematics, a **matrix is a grid of numbers, arranged in rows and columns.**
- ✓ Thus, using two dimensional arrays, we can perform the following operations on an m×n matrix:

1. Transpose

- ✓ Transpose of an m × n matrix A is given as a n × m matrix B, where

$$\mathbf{B_{i,j} = A_{j,i}.}$$

2. Sum

- ✓ Two matrices that are compatible with each other can be added together, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be added by writing:

$$\mathbf{C_{i,j} = A_{i,j} + B_{i,j}}$$

3. Difference

- ✓ Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. Two matrices are said to be compatible when they have the same number of rows and columns. The elements of two matrices can be subtracted by writing:

$$\mathbf{C_{i,j} = A_{i,j} - B_{i,j}}$$

4. Product

- ✓ Two matrices can be multiplied with each other if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, m × n matrix A can be multiplied with a p × q matrix B if n=p. The dimension of the product matrix is m × q. The elements of two matrices can be multiplied by writing:

$$\mathbf{C_{i,j} = \Sigma\ A_{i,k}\ B_{k,j}} \text{ for } k = 1 \text{ to } n$$

Example:

1. Write a C program to transpose a 3 × 3 matrix.

```
#include <stdio.h>

void main()
{
        int i, j, mat[3][3], transposed_mat[3][3];
        printf("\n Enter the elements of the matrix ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d", &mat[i][j]);
                }
        }
        printf("\n The elements of the matrix are ");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
```

```
                printf("%d\t", mat[i][j]);
            }
            printf("\n");
        }
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
                transposed_mat[i][j] = mat[j][i];
        }


        printf("\n The elements of the transposed matrix are ");
        for(i=0;i<3;i++)
        {
            for(j=0;j<3;j++)
            {
                printf("%d\t",transposed_ mat[i][j]);
            }
            printf("\n");
        }
}
```

Output

```
Enter the elements of the matrix
1 2 3 4 5 6 7 8 9
The elements of the matrix are
1 2 3
4 5 6
7 8 9
The elements of the transposed matrix are
1 4 7
2 5 8
3 6 9
```

2. Write a program to input two m × n matrices and then calculate the sum of their corresponding elements and store it in a third m × n matrix.

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

void main()
{
        int i, j, m, n,  p, q, a[5][5], b[5][5], c[5][5];
        printf("\n Enter the number of rows and columns in the first matrix : ");
        scanf("%d%d",&m,&n);

        printf("\n Enter the number of rows and columns in the second matrix : ");
        scanf("%d%d",&p,&q);

        if(m != p || n != q)
        {
            printf("\n Number of rows and columns of both matrices must be equal");
            exit(0);
        }

        printf("\n Enter the elements of the first matrix ");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }

        printf("\n Enter the elements of the second matrix ");
        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                {
                        scanf("%d",&b[i][j]);
                }
        }

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        c[i][j] = a[i][j] + b[i][j];
        }

        printf("\n The elements of the resultant matrix are ");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t ", c[i][j]);
                }
                printf("\n");
        }
}
```

3.9 Passing two-dimensional arrays to functions

- ✓ There are three ways of passing a two-dimensional array to a function.
- ✓ First, we can **pass individual elements of the array**. This is exactly the same as passing an element of a one-dimensional array.
- ✓ Second, we can pass a **single row of the two-dimensional array.** This is equivalent to passing the entire one-dimensional array to a function.
- ✓ Third, we can pass the **entire two-dimensional array to the function.**
- ✓ Figure 3.31 shows the three ways of using two-dimensional arrays for inter-function communication.

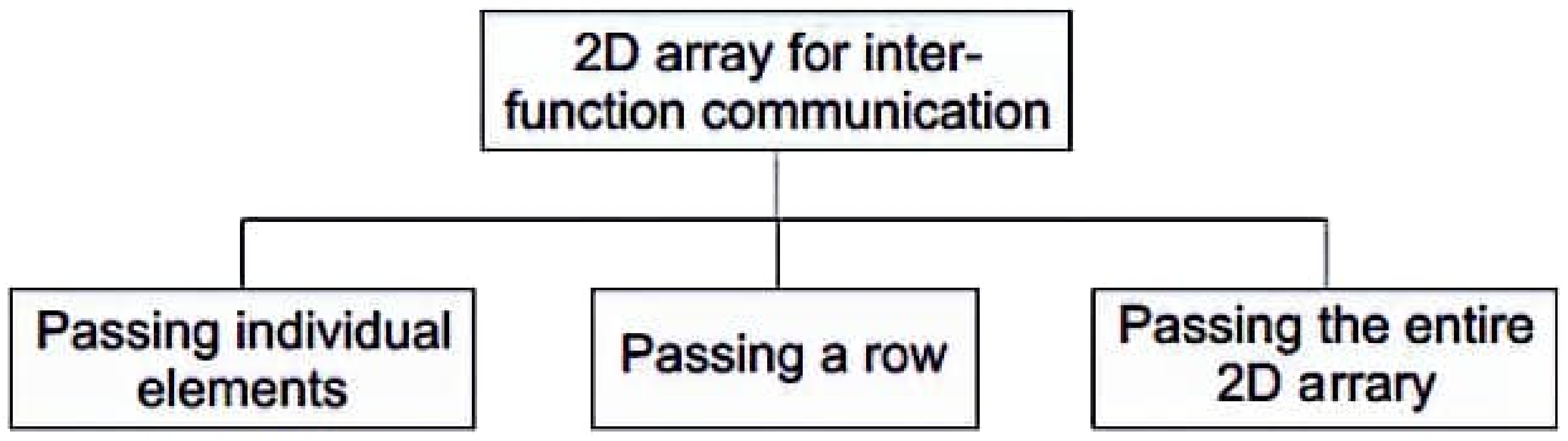

2D array for inter-function communication
Passing individual elements
Passing a row
Passing the entire 2D array

Figure 3.31 2D arrays for inter-function communication

1. Passing individual elements

- ✓ The individual elements of an array can be passed to a function by passing either their **data values or addresses.**

Passing Data Values

Calling function
```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(arr[1][1]);
}
```

Called function
```
void func(int num)
{
        printf("%d", num);
}
```

Passing Addresses

Calling function
```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(&arr[1][1]);
}
```

Called function
```
void func(int *num)
{
            printf("%d", *num);
}
```

2. Passing a Row

- ✓ A row of a two-dimensional array can be passed by **indexing the array name with the row number.**
- ✓ Look at Fig. 3.32 which illustrates how a single row of a two-dimensional array can be passed to the called function.

Calling function
```
main()
{
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    func(arr[1]);
}
```

Called function
```
void func(int arr[])
{
        int i;
        for(i=0;i<5;i++)
            printf("%d", arr[i]);
}
```

3. Passing the Entire 2D Array

- ✓ To pass a two-dimensional array to a function, we use the **array name as the actual parameter**. However, the parameter in the called function must indicate that the array has two dimensions.

Calling function

```
main()
{
   int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
   func(arr);
}
```

Called function

```
void func(int arr[5][5])
{
        for(i=0;i<5;i++)
                for(j=0;j<5;j++)
                    printf("%d", arr[i][j]);
}
```

3.10 Multi-Dimensional Arrays

- ✓ A multi-dimensional array in simple terms is **an array of arrays.**
- ✓ As we have **one index in a one-dimensional array, two indices in a two-dimensional array**, in the same way, we have **n indices in an n-dimensional array or multi-dimensional array.**
- ✓ Conversely, an **n–dimensional array is specified using n indices**.
- ✓ An n-dimensional $m_1 \times m_2 \times m_3 \times ... \times m_n$ array is a collection of $m_1 \times m_2 \times m_3 \times ... \times m_n$ elements. In a multi-dimensional array, a particular element is specified by using n subscripts as $A[I_1][I_2][I_3]...[I_n]$,
- ✓ Figure 3.33 shows a three-dimensional array. The array has three pages, four rows, and two columns.

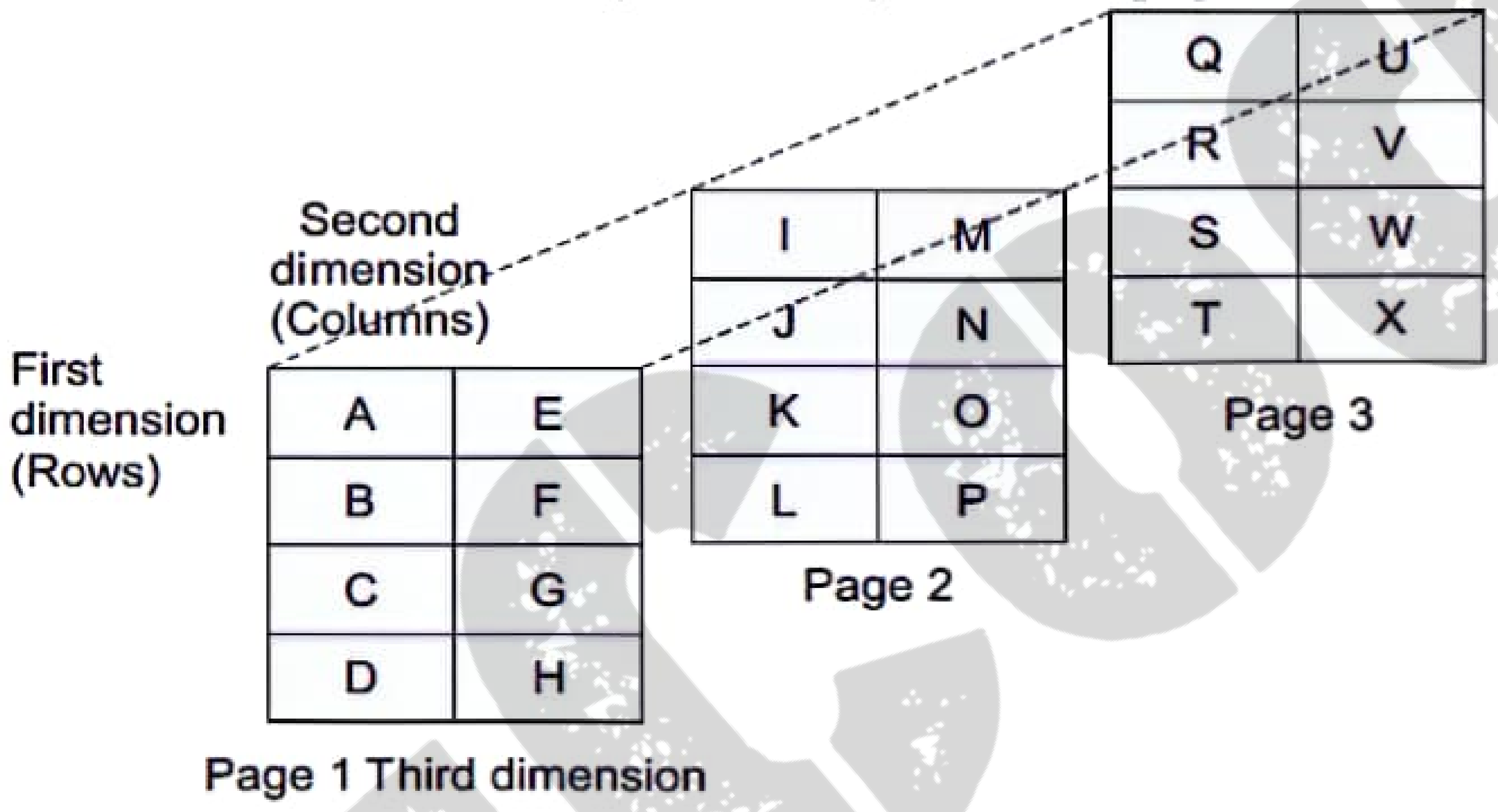


Figure 3.33 Three-dimensional array

Example:

1. Write a program to read and display a 2 × 2 × 2 array.

```
#include <stdio.h>

void main()
{
        int array[2][2][2], i, j, k;

        printf("\n Enter the elements of the matrix");
        for(i=0;i<2;i++)
        {
                for(j=0;j<2;j++)
                {
                        for(k=0;k<2;k++)
                        {
                                scanf("%d", &array[i][j][k]);
                        }
                }
        }
```

```
void main()
{
        int m,n,res;
        printf("Enter the values for m,n\n");
        scanf("%d%d",&m,&n);
        res=add(m,n);                    // Actual parameters m,n
        printf("Sum=%d\n",res);
}
```