MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

(A constituent unit of MAHE, Manipal)

CSS 2101-DATA STRUCTURES

4 – Credits [L-3 T-1 P-0 4]

School of Computer Engineering, MIT, Manipal

Course objectives

- Explain the fundamentals of data structures and their applications essential for **programming/problem solving**
- Analyze Linear Data Structures: **Stacks, Queues and Linked Lists**
- Analyze Non-Linear Data Structures: **Trees and Graphs**

Course outcomes

- Apply structures and recursion techniques to model and solve real-world problems.
- Apply linked list techniques to implement dynamic data structures and solve real-world problems.
- Implement stack and queue operations to address real-world computational scenarios.
- Analyze tree-based data structures to evaluate their efficiency in solving hierarchical data problems.
- Construct and evaluate graph representations and apply traversal algorithms.

Module 1: Introduction and Fundamentals

- Arrays – operations, applications,
- Pointers and array of pointers,
- Recursion – function recursion and applications,
- Memory allocation functions (malloc, calloc, free, realloc),
- Structures and array of structures,
- Searching Techniques – Linear Search, Binary Search, Sorting Techniques,
- Sparse Matrix – representation and operations

Module 2: LINKED LISTS

- Singly Linked List and Chains, Representing Chains in C,
- Doubly Linked Lists, Circular Linked Lists, Linked Lists with Header Node,
- Linked Lists Applications – polynomial operations, Additional List Operations – Operations for Chains, Operations for Circularly Linked Lists

Module 3: STACKS and QUEUES

- Stacks and operations,
- Queues – linear, circular, Evaluation of Expressions – infix, postfix, prefix and conversions,
- Multiple Stacks and Queues, Priority Queues and their representation, Double Ended Queue (Deque), Input/Output Restricted Queues

Module 4: TREES

- Terminology, Representation of Trees, Binary Trees – operations and expression trees
- Binary Tree Traversals: inorder, preorder, postorder, Additional Binary Tree Operations, Threaded Binary Trees,
- Binary Search Tree – definition, search, insertion, deletion,
- AVL Trees – rotations, balancing, Red-Black Trees

Module 5: GRAPHS

- Introduction, Definitions
- Graph Representations- adjacency matrix, adjacency list,
- Graph Traversals: Depth First Search (DFS), Breadth First Search (BFS)

Lesson Plan

Module	Lectures	Tutorials	Total Hours
Module 1: Introduction and Fundamentals	9	3	12
Module 2: Linked Lists	9	3	12
Module 3: Stacks and Queues	7	2	9
Module 4: Trees	8	3	11
Module 5: Graphs	3	1	4
Total	36	12	48

References

- Behrouz A. Forouzan, Richard F. Gilberg, A Structured Programming Approach Using C,(3e), Cengage Learning India Pvt. Ltd, India, 2007.
- Ellis Horowitz, Sartaj Sahani, Susan Anderson and Freed, Fundamentals of Data Structures in C, (2e), Silicon Press, 2007.
- Richard F. Gilberg, Behrouz A. Forouzan, Data structures, A Pseudocode Approach with C, (2e), Cengage Learning India Pvt. Ltd, India, 2009.
- Tenenbaum Aaron M., Langsam Yedidyah, Augenstein Moshe J., Data structures using C, Pearson Prentice Hall of India Ltd., 2007.
- Debasis Samanta, Classic Data Structures, (2e), PHI Learning Pvt. Ltd., India, 2010.
- https://onlinecourses.swayam2.ac.in/cec25_hs62/preview [Introduction to Data Structures, Punjabi University, Patiala].
- In addition to the above, **internet resources** are also used to aid student understanding and enhance the teaching-learning experience with practical examples and visualizations.

Module 1: Introduction and Fundamentals

C Program

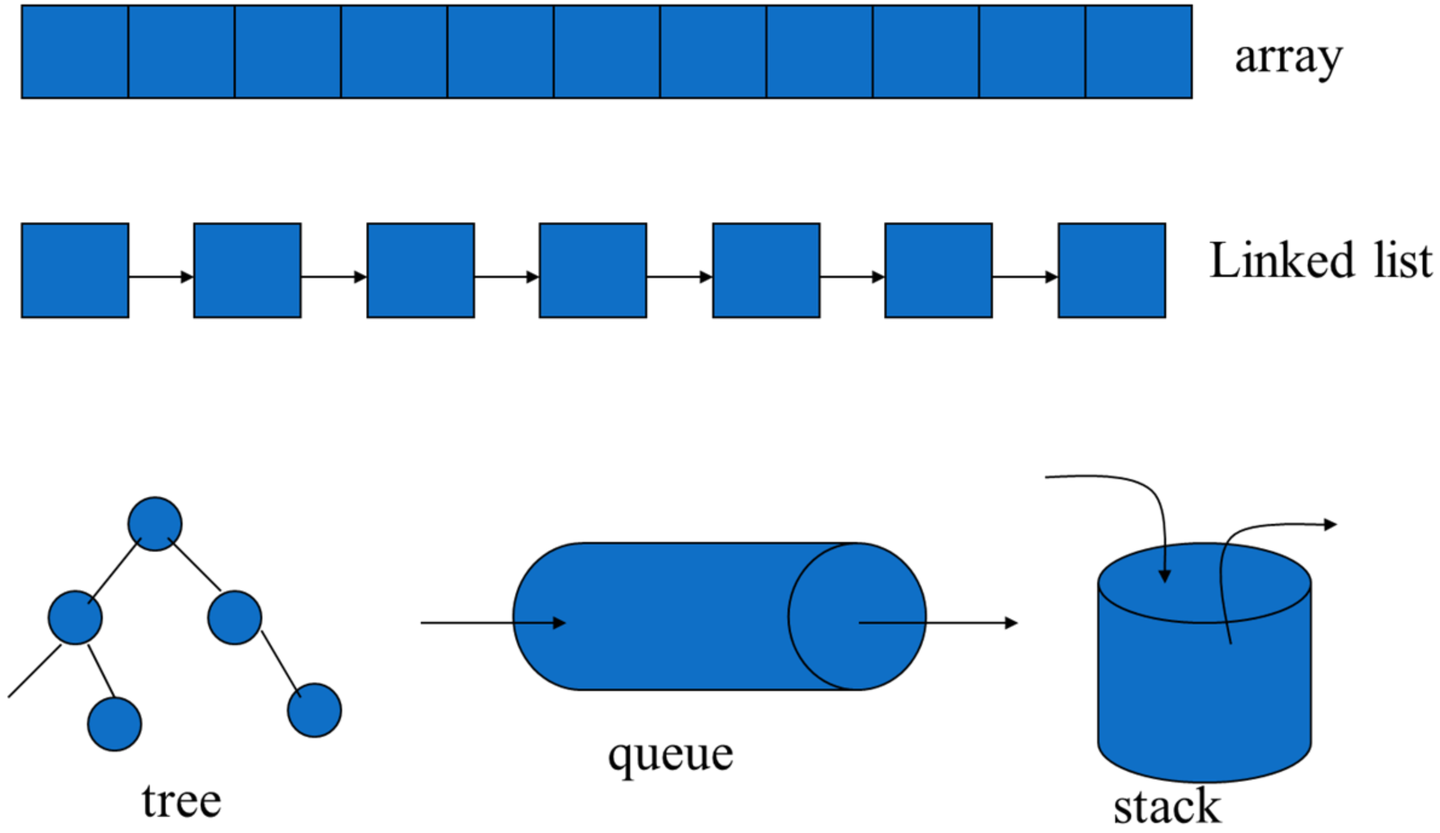
You will be using C programming language in this course.

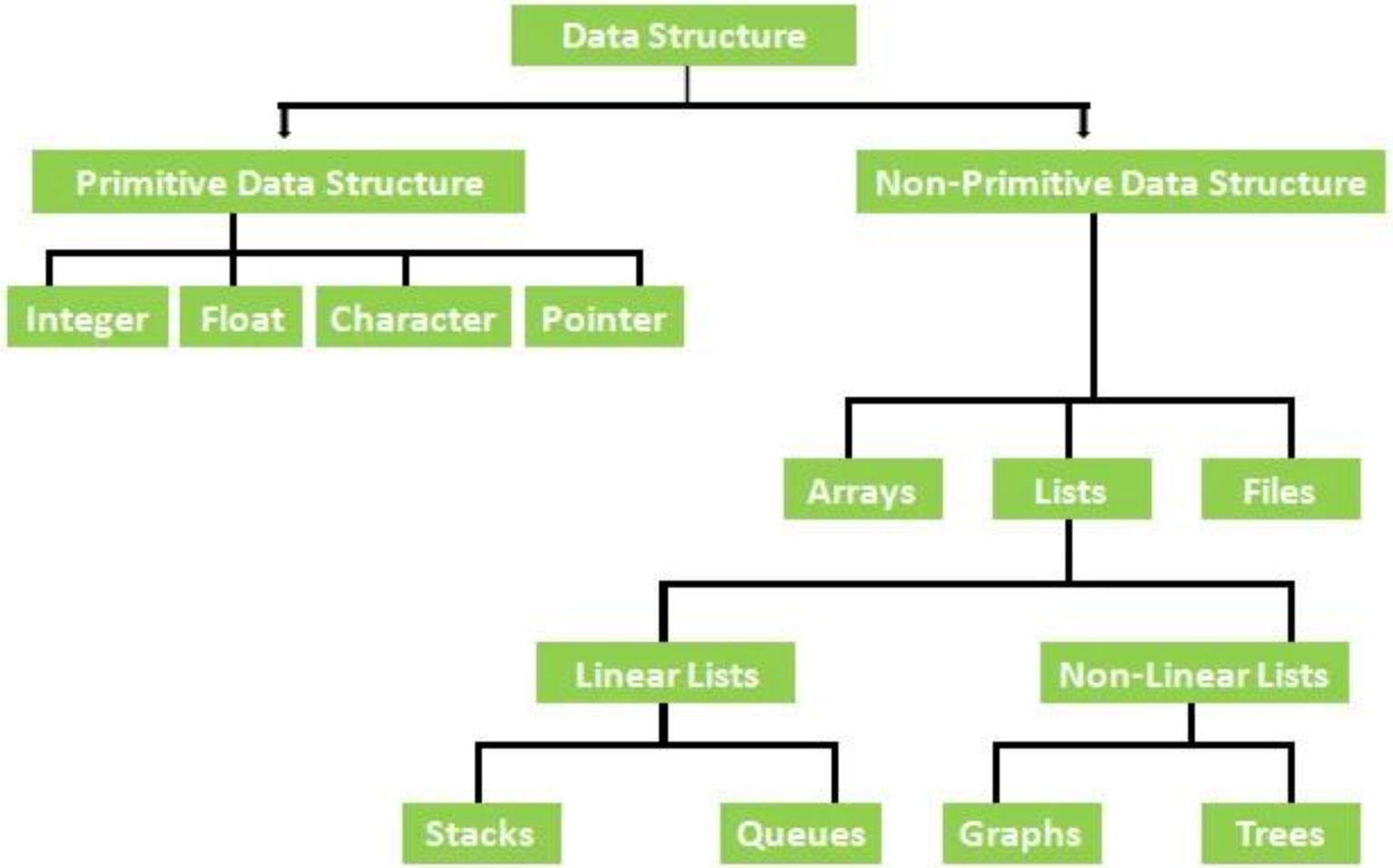
Data Structures

•A **data structure** is a way of storing and organizing data.

•It helps in making data **easy to use** and **efficient to process**.

•It defines how related pieces of information are managed in a program.




Data Structure
Primitive Data Structure
Non-Primitive Data Structure
Integer
Float
Character
Pointer
Arrays
Lists
Files
Linear Lists
Non-Linear Lists
Stacks
Queues
Graphs
Trees

Revisiting First-Year Concepts

Datatypes and Variables

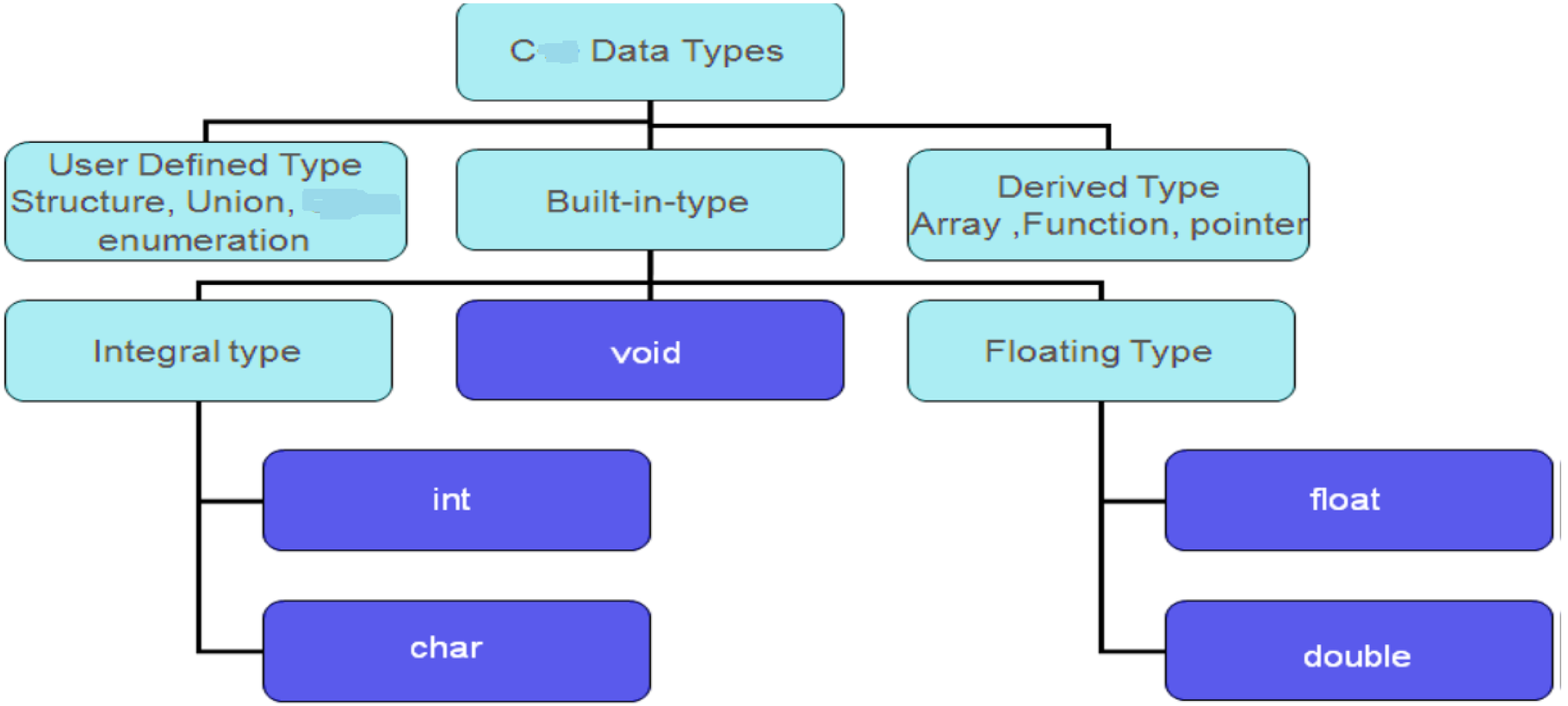
C Tokens

Preprocessor Directives

Control Statements

Loops

C - Data Types


C Data Types
User Defined Type
Structure, Union,
enumeration
Built-in-type
Derived Type
Array ,Function, pointer
Integral type
void
Floating Type
int
char
float
double

C Tokens

- Keywords → words that are basically sequence of characters defined by a computer language that have one or more fixed meanings.
  - They are also called *reserved words*.
  - Keywords cannot be changed. ex. int, float, do-while, if, else…
- Identifiers → words which have to be identified by keywords.
  - user defined names. ex. int amount, float avg,…
- Operators → +, -, *, %, /, …
- Strings → “Manipal”
- Constants → -15, 10
- Special Symbols → { } (,…

main()

- The main function is the point where all C programs start their execution, independently of its location within the source code.
- it is essential that all C programs have a main function.
- The word main is followed in the code by a pair of parentheses (). That is because it is a function declaration.
- Optionally, these parentheses may enclose a list of parameters within them.
- Right after these parentheses we can find the body of the main function enclosed in braces{ }.

Preprocessor Directives in C

- **Preprocessor directives** are **instructions** that are processed by the **preprocessor** before the actual **compilation** of code begins.

Directive	Purpose	Example
#include	Includes header files	#include <stdio.h>
#define	Defines constants or macros	#define PI 3.14
#undef	Undefines a macro	#undef PI
#ifdef / #ifndef	Conditional compilation	#ifdef DEBUG
#endif, #else, #elif	End or modify conditional blocks	

Simple C Program

```
// Display “This is my first C program”
// Single line comment
#include <stdio.h>
int main() // Entry point for program execution
{
   // Block of statements:
   printf("Welcome to Data Structures Course\n");
   // Block of statements:
   return 0;
}
```

Program to read and display a number

```
#include <stdio.h>

int main() {                    // Program body begins
   int number;                  // Variable declaration
   printf("Enter number: ");    // User-friendly message to prompt input
   scanf("%d", &number);        // Reading input value into the variable
   printf("The number is %d\n", number);  // Displaying the output
   return 0;                    // End of program
}
```

C **decision making** and **branching** statements

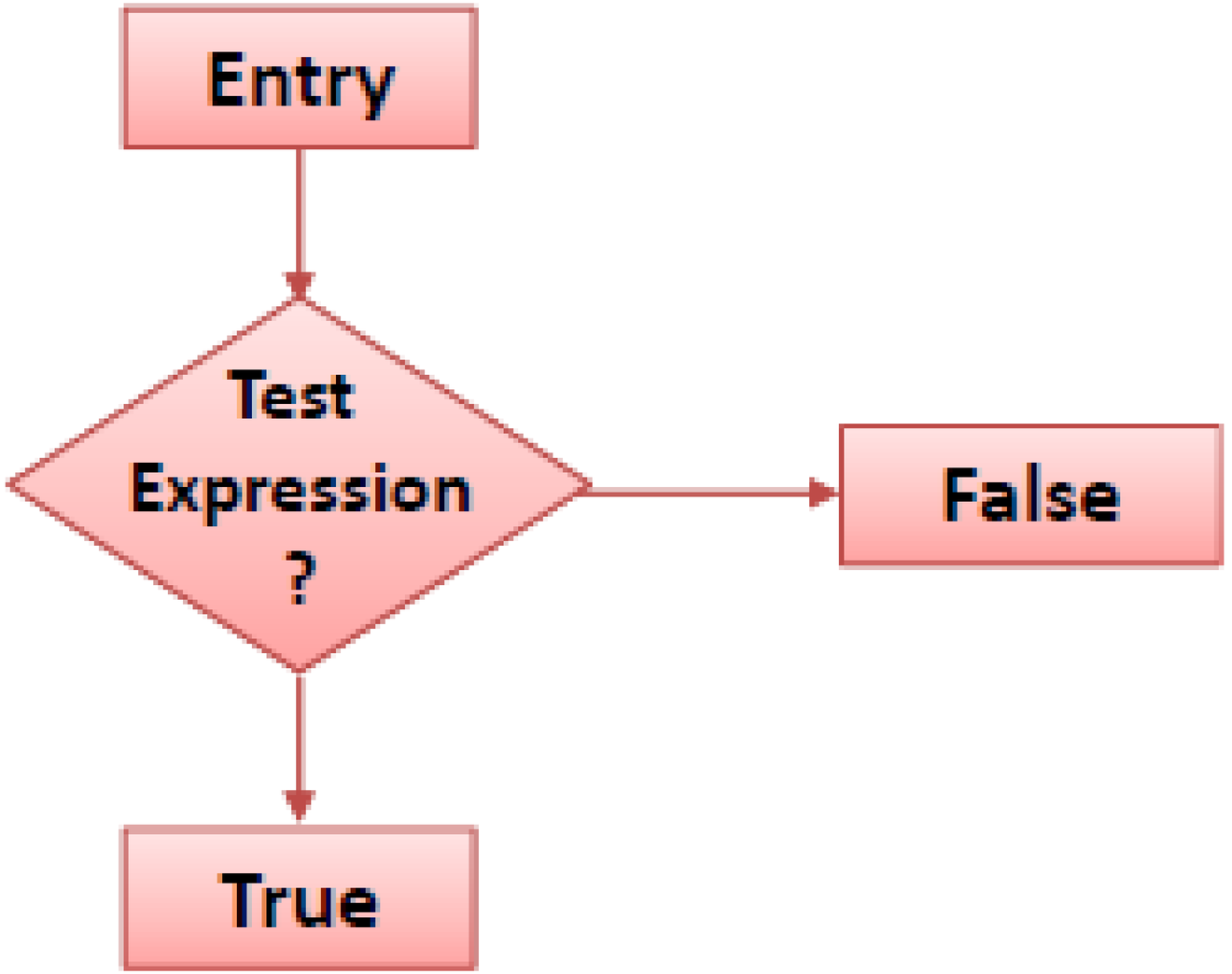
1. if Statement

2. switch statement

if statement

Purpose:

- Used to **control the flow** of execution of statements based on a **condition**.
- It is a **two-way decision** statement used with a **test expression**.



Different forms of **if** statement

1. Simple if statement.
2. if...else statement.
3. Nested if...else statement.
4. else if ladder.

Simple **if** Statement

General form of the simplest if statement:

if (*test Expression*)

{

statement-block;

}

statement_x;

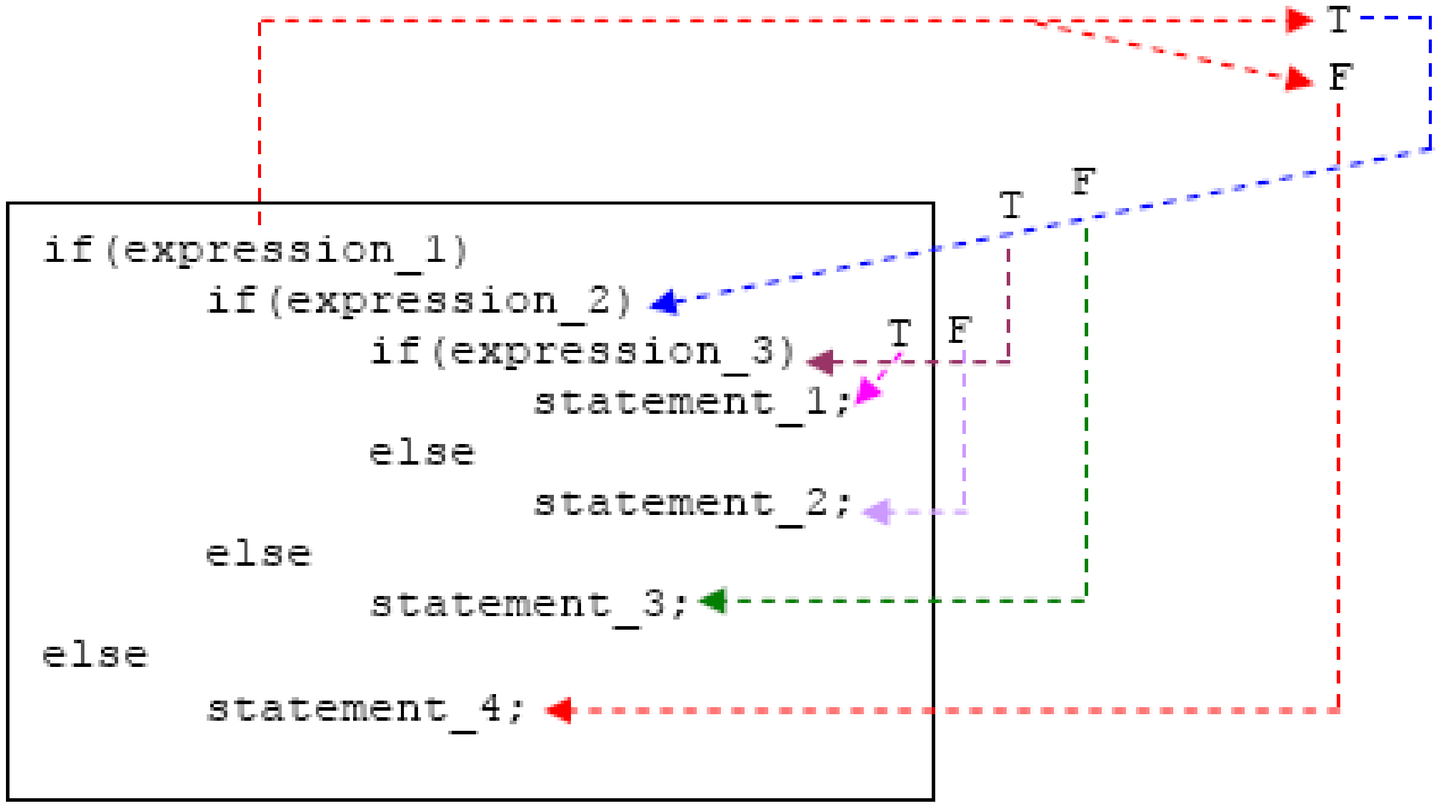
If else statement

- **Purpose:**
- Used to **execute one block of code if a condition is true**, and **another block if it is false**.
- Helps in **two-way branching** based on a condition.

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

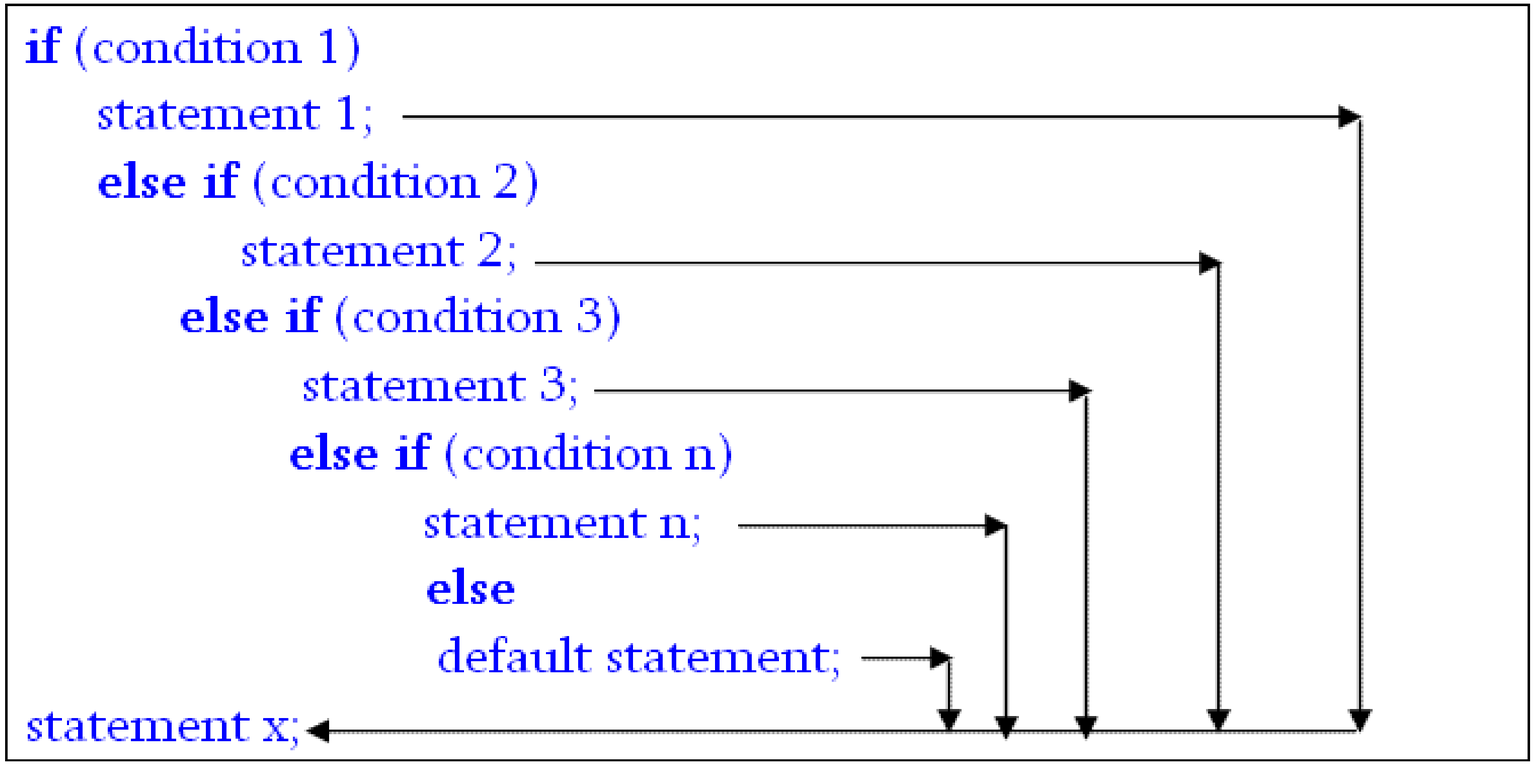
Nesting of if-else Statements

If else statement




```
next_statement;
```

else if Ladder



switch Statement

- Switch is multiple–branching statement - based on a condition, the control is transferred to one of the many possible points.
- The most flexible control statement in selection structure of program control.
- Enables the program to execute different statements based on an expression that can have more than two values. Also called multiple choice statements.

switch Statement

```
switch (expression) {
  case value_1:
    statement(s);
    break;
  case value_2:
    statement(s);
    break;
  ...
    case value_n:
    statement(s);
    break;
  default:
    statement(s);  // optional, executed if no match
}
next_statement;
```

switch- example

```
int main() {
   int mark = 85; // Example mark
   int index = mark / 10;
   char grade;
   switch (index) {
      case 10:
      case 9:
      case 8:
         grade = 'A';
         break;
      case 7:
      case 6:
         grade = 'B';
         break;
      case 5:
         grade = 'C';
         break;
      case 4:
         grade = 'D';
         break;
      default:
         grade = 'F';
         break;
   }
   printf("%c\n", grade);
   return 0;
}
```

Decision Making and Looping Control Structures

Looping (Iterative) Control Structures

- **Used to repeat a set of statements** multiple times based on a condition.
- Statements are executed **as long as the condition is true**.
- Also known as **loop control structures**.

Types of Looping Structures in C

while loop

- Entry-controlled loop
- Condition is checked **before** executing the body

do-while loop

- Exit-controlled loop
- Body is executed **at least once**, condition is checked **after**

for loop

- Entry-controlled loop
- Best when the **number of iterations is known**

While statement

Basic format:

while (test condition)

{

body of the loop

}

- **Entry controlled** loop statement
- Test condition is evaluated & if it is true, then body of the loop is executed.
- After execution, the test condition is again evaluated & if it is true, the body is executed again.
- This is repeated until the test condition becomes false, & control transferred out of the loop.
- **Body of loop may not be executed if the condition is false at the very first attempt**.

Do - While statement

General form:

```
do
        {
                body of the loop
        }
while (test condition);
```

for statement

The general form:

for (initialization; test condition; increment)

{

Body of the loop

}

Next statement;

Nesting of ***for*** loop

One *for* statement within another for statement.

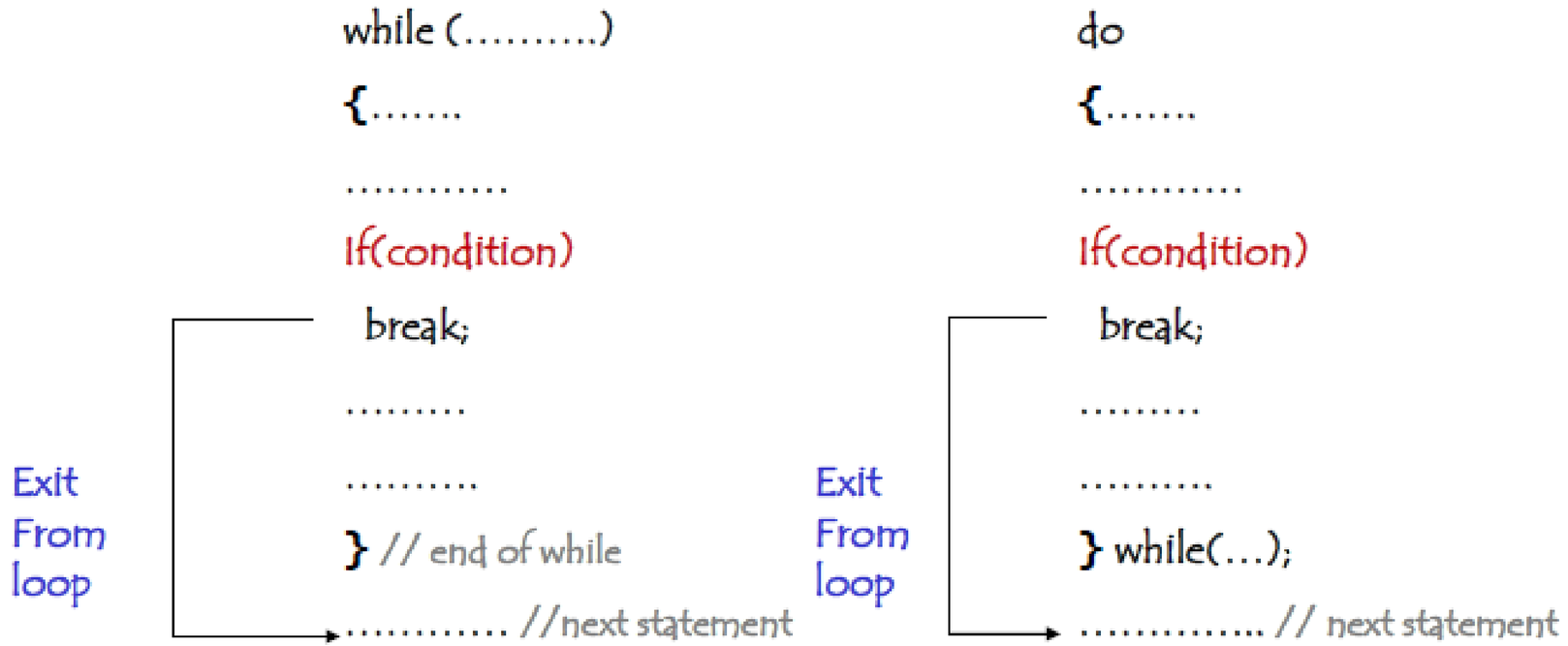
```
for (i=0; i< m; ++i)
        {.....
                ....
                for (j=0; j < n;++j)
                {......
                        .....
                } // end of inner 'for' statement
        }// end of outer 'for' statement
```

Jumping out of a loop

- An early exit from a loop can be accomplished by using the break statement.
- When the break statement is encountered inside a loop, the loop is immediately exited & the program continues with the statement immediately following the loop.
- When the loops are nested , the break would only exit from the loop containing it.

i.e., the break will exit only a single loop.

Exiting a loop with **break** statement

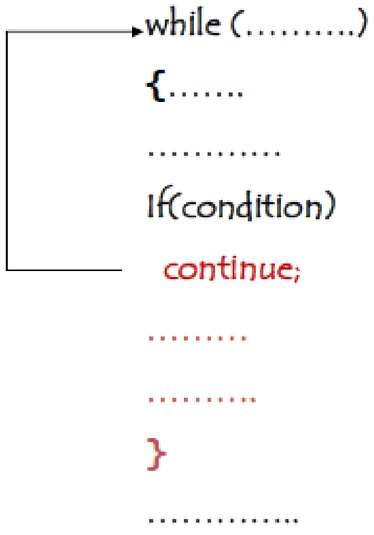


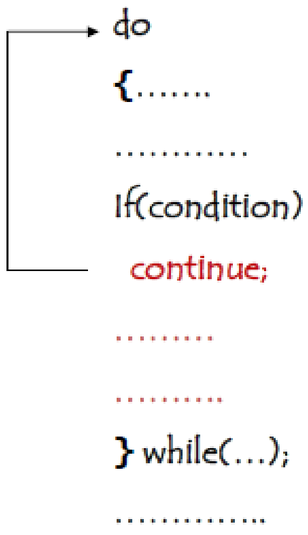
Skipping a part of loop

- Skip a part of the body of the loop under certain conditions

  Using continue statement.

- As the name implies, causes the loop to be continued with next iteration, after skipping rest of the body of the loop.





Abstract Data Type (ADT) Specification

What is an ADT?

An **Abstract Data Type (ADT)** is a **data type** where the:

- **Specification of objects** (what the data represents), and
- **Specification of operations** (what we can do with the data)

are **separated** from:

- **Implementation details** (how the data is stored and how operations are carried out).

Arrays

An array is a data structure that stores a fixed-size sequence of elements of the same data type. It is a container that can hold multiple values of the same type in a contiguous memory block.

The type of an array refers to the type of its elements, which can be any primitive data type (such as integers, floating-point numbers, characters)

Three different types of arrays

One-Dimension Array
Two-Dimension Array
Multi Dimension Array

Core Operations of the Array ADT

1. Create(j, list):
- Constructor function
- Returns an array with **j dimensions**
- list is a j-tuple defining the size of each dimension
- All values initially **undefined**

2. Retrieve(A, i):
- Observer function
- Returns value at index i if valid
- Otherwise, returns an **error**

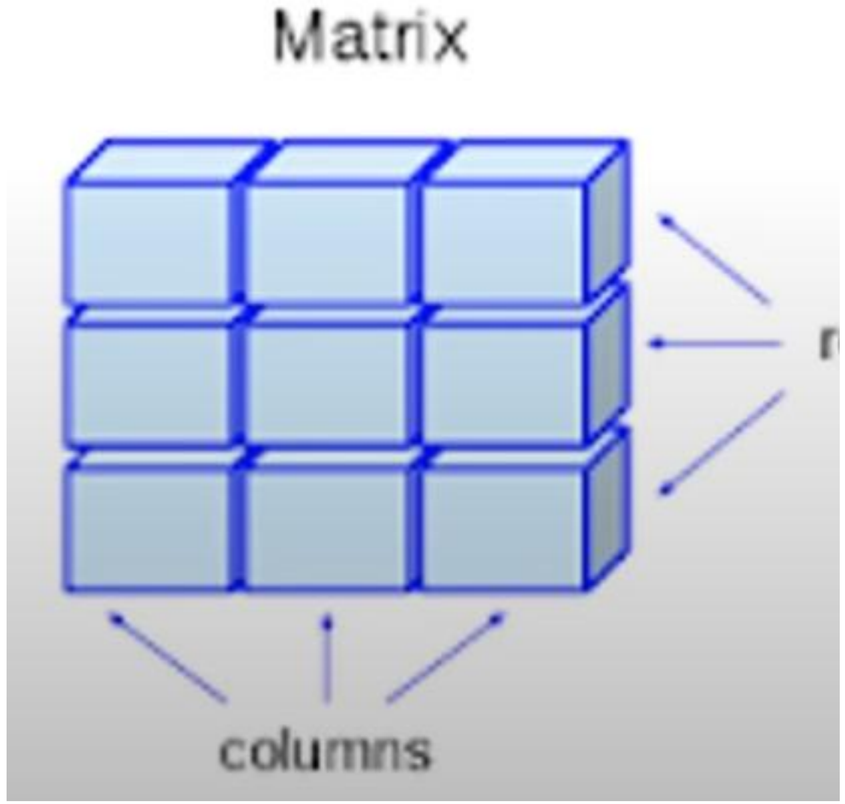
3. Store(A, z, x):
- Transformer function
- Updates array A by storing value x at index z
- Returns **error** if z is invalid

One-Dimension Array

```
#include <stdio.h>
int main() {
int num[5] = {10, 20, 30, 40, 50};
// Accessing elements of the array
printf("Element at index 0: %d\n", num[0]);
printf("Element at index 2: %d\n", num[2]);
// Modifying elements of the array
 num[3] = 12;
printf("Modified element at index 3: %d\n", num[3]);
 return 0;
 }
```

Two-Dimension Array

```
#include <stdio.h>
int main() {
 int matrix[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
// Accessing elements of the array
 printf("Element at row 0, column 0: %d\n", matrix[0][0]);
printf("Element at row 1, column 2: %d\n", matrix[1][2]);
// Modifying elements of the array
matrix[2][1] = 10;
printf("Modified element at row 2, column 1: %d\n",matrix[2][1]);
 return 0;
}
```



2D Arrays

- It is an ordered table of homogeneous elements.
- It is generally referred to as **matrix**, of some rows and some columns.
- It is also called a **two-subscripted variable**.

2D Arrays

- For example

```
int marks[5][3];
float matrix[3][3];
char page[25][80];
```

- The first example tells that marks is a 2-D array of 5 rows and 3 columns.
- The second example tells that matrix is a 2-D array of 3 rows and 3 columns.
- Similarly, the third example tells that page is a 2-D array of 25 rows and 80 columns.

2D Arrays

- Declaration

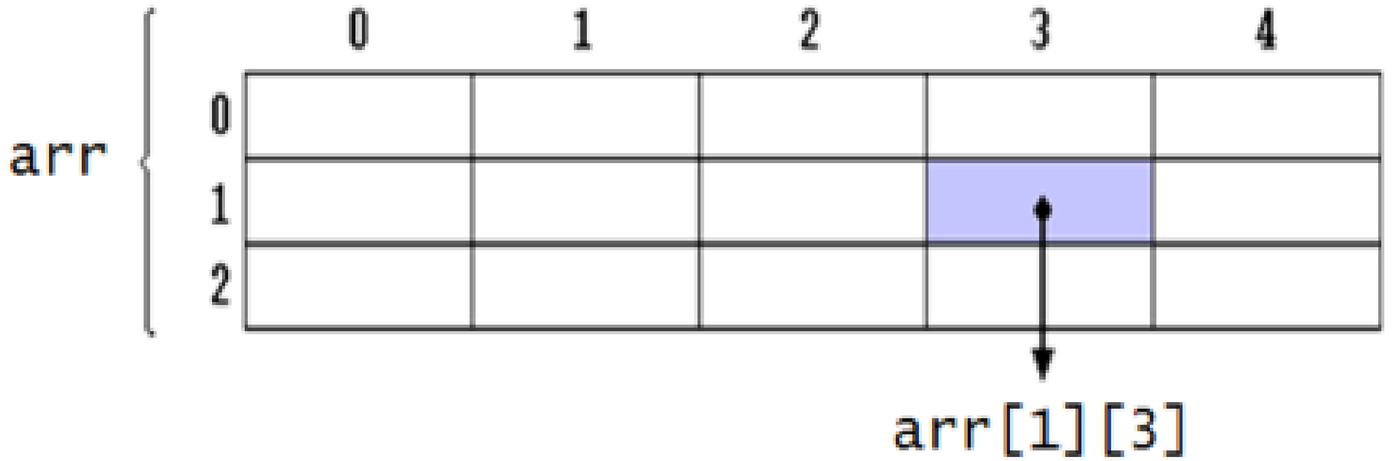
type array_name[row_size][column_size];

- For example,

  int arr [3][5];

  arr represents a two-dimensional array or table having 3 rows

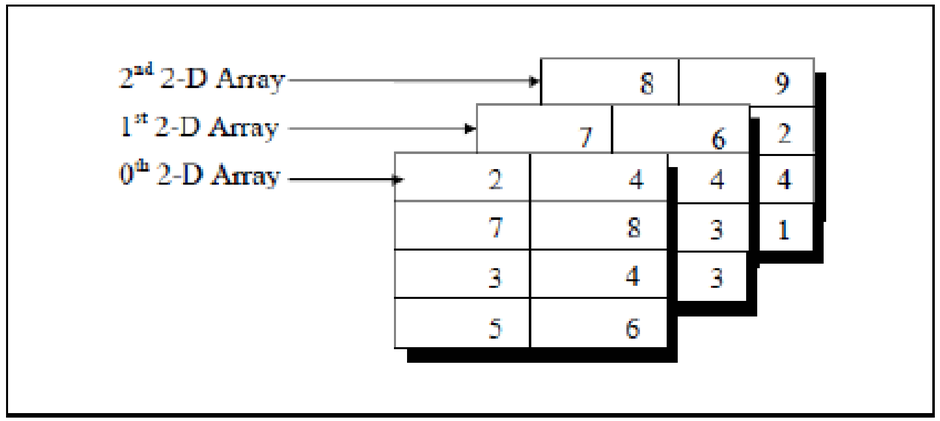
  and 5 columns and it can store 15 integer values.



Multi – dimensional arrays

```
int arr[3][4][2]= {
                    { { 2, 4 }, { 7, 8 }, { 3, 4 }, { 5, 6 } },
                    { { 7, 6 }, { 3, 4 }, { 5, 3 }, { 2, 3 } },
                    { { 8, 9 }, { 7, 2 }, { 3, 4 }, { 5, 1 }, }
                  } ;
```

A three-dimensional array can be thought of as an array of arrays of arrays.



Conceptual View

Try

- Write a C program to read two matrices A & B, create and display a third matrix C such that C (i, j) = max (A(i, j) , B(i, j))

Searching Techniques

Linear Search

```
linearSearch(arr[], n, target)
{
    for i = 0 to n-1
    {
        // If the current element is equal to the target, return the index
        if arr[i] == target
            return i
    }
    // If the target is not found in the array, return -1
    return -1
}
```

Linear Search Applications

- Linear search is simple and works well for small or unsorted datasets.
- **Finding a contact in a phone list** (unsorted)
- **Searching for a file in a folder** without any sorting
- **Checking attendance** by scanning a list of names
- **Inventory checks** in small retail systems
- **Debugging tools** that scan logs line by line

Binary Search

```
binarysearch(a[n], key, low, high)
while(low<high)
{
mid = (low+high)/2;
if(a[mid]=key)
   return mid;
elseif (a[mid] > key)
   high=mid-1;
 else
   low=mid+1;
 }
return -1;
```

Binary Search Applications

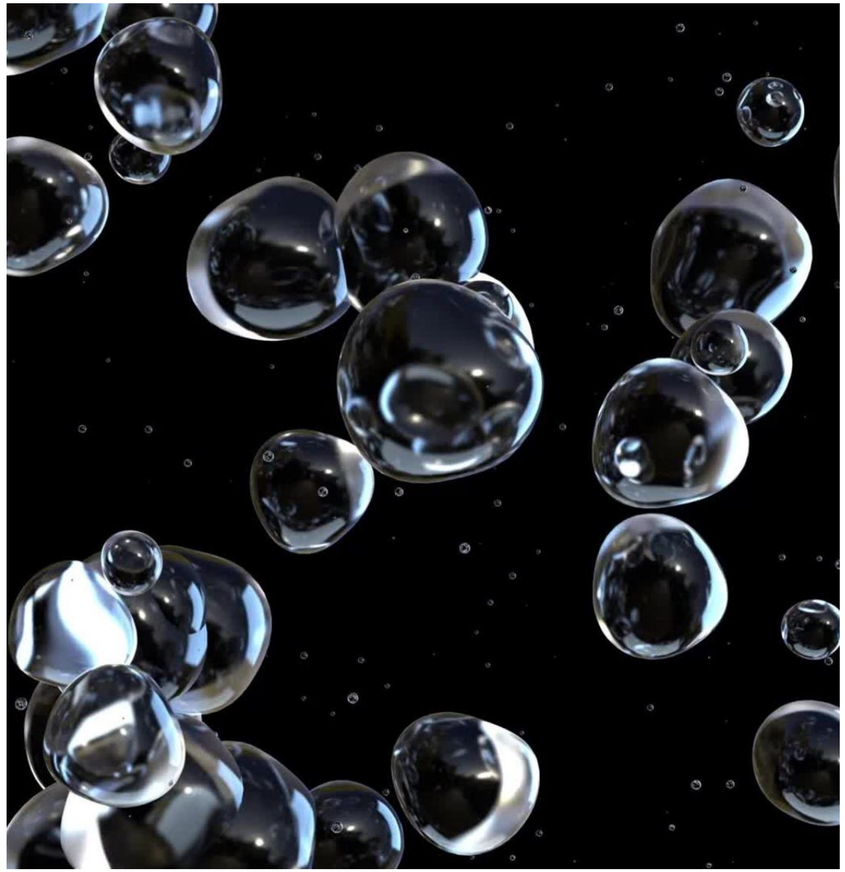
Binary search is efficient but requires sorted data.

1. **Dictionary or glossary lookup** (alphabetically sorted)
2. **Searching in a sorted database** (e.g., student records by roll number)
3. **Auto-complete suggestions** in search engines
4. **Gaming**: Finding optimal settings or values (e.g., binary search for difficulty tuning)
5. **Version control systems**: Git uses binary search to find the commit that introduced a bug (via git bisect)

Sorting Techniques

Bubble Sort

```
void bubbleSort(int arr[ ], int n) {
   for (int i = 0; i < n - 1; i++) {
      // Last i elements are already in place

      for (int j = 0; j < n - i - 1; j++) {
         if (arr[ j ] > arr[ j + 1 ]) {
            // Swap arr[j] and arr[j + 1]
             int temp = arr[ j ];
              arr[ j ] = arr[ j + 1 ];
             arr[ j + 1 ] = temp;
         }
      }
   }
}
```

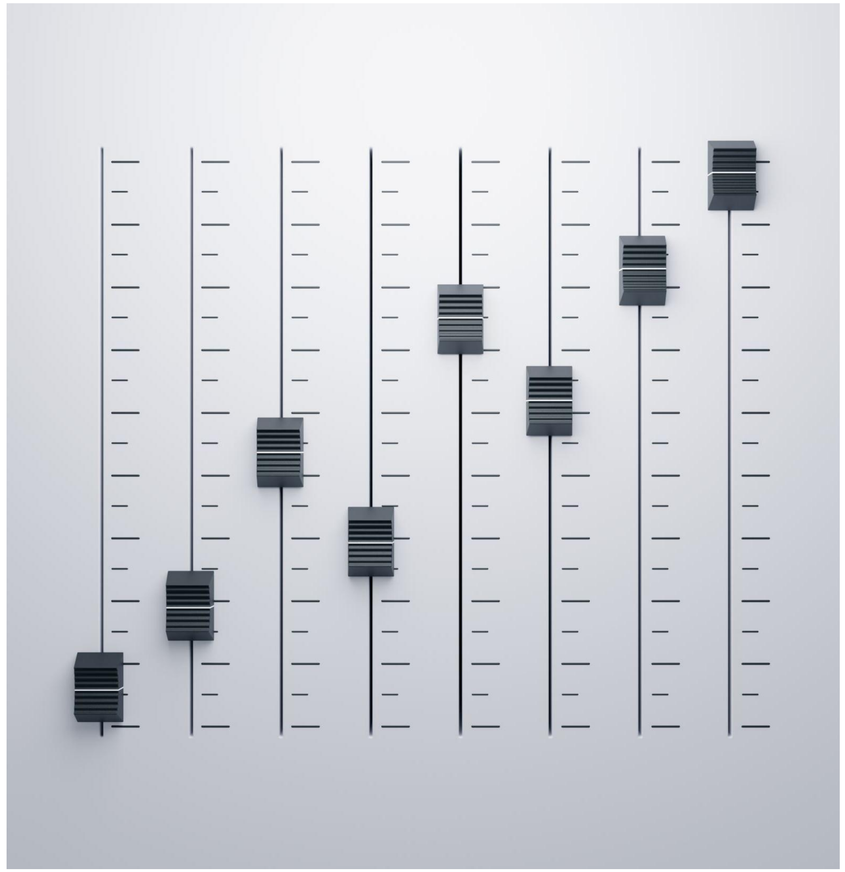


Bubble Sort Application

- Teaching basic sorting concepts in education
- Sorting **small datasets** in simple applications
- Reordering elements in animation or visual effects
- Detecting nearly sorted data for optimization

Selection Sort

```
void selectionSort(int arr[ ], int n) {
   for (int i = 0; i < n - 1; i++) {
      int minIndex = i;
      // Find the minimum element in the unsorted part
      for (int j = i + 1; j < n; j++) {
         if (arr[ j ] < arr[ minIndex ])
                                  minIndex = j;
      }
      // Swap the found minimum with the first element
      if ( minIndex != i ) {
            int temp = arr[ i ];
            arr[ i ] = arr[ minIndex ];
            arr[ minIndex ] = temp;
      }
   }
}
```

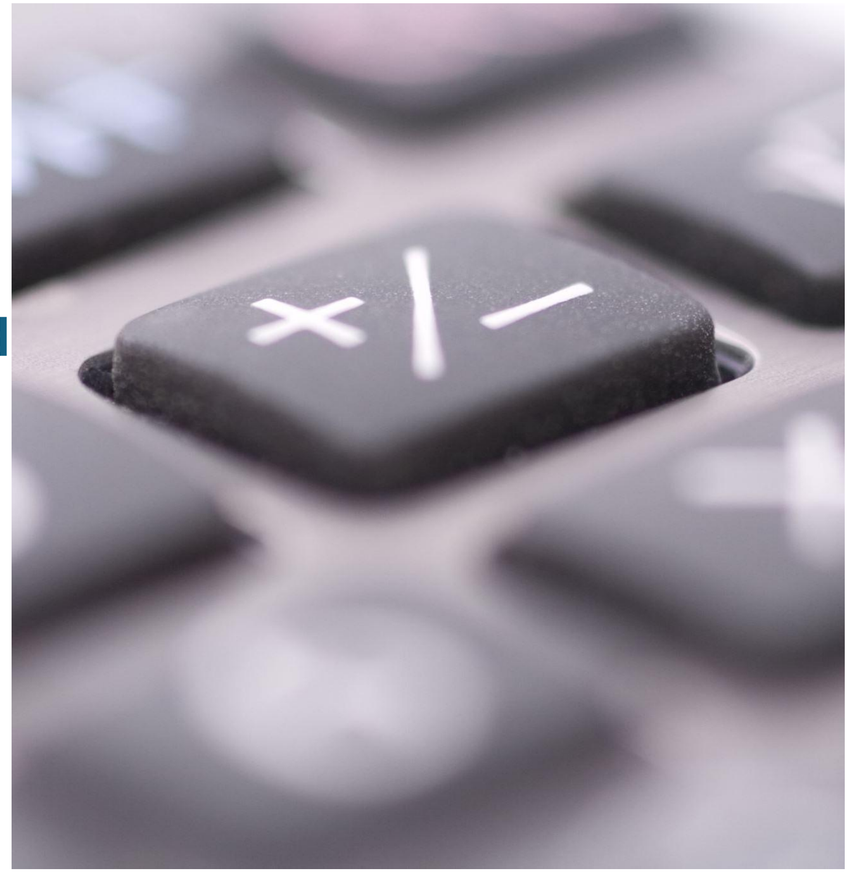


Selection Sort Application

- Choosing the smallest/largest item from a list (e.g., tournament rankings)
- Sorting a small list of names or numbers manually
- Used in embedded systems with limited memory
- Arranging students by marks when dataset is small

Insertion Sort

```
void insertionSort(int arr[ ], int n) {
   for ( int i = 1; i < n; i++ ) {
      int key = arr [ i ];
      int j = i - 1;

// Move elements greater than key to one position ahead
      while ( j >= 0 && arr[ j ] > key) {
         arr[ j + 1 ] = arr [ j ];
         j--;
      }
      arr[ j + 1 ] = key;
   }
}
```



Insertion Sort Application

- Sorting playing cards in hand (real-life analogy)
- Maintaining a sorted list in real-time (e.g., live leaderboard)
- Used in online algorithms where data arrives over time
- Efficient for small or nearly sorted datasets

Pointers

- A **pointer** is a variable whose value is the address of another variable,
  i.e., direct address of the memory location.
- The general form of a pointer variable declaration is –

```
type *var-name;
```

- Example:

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

Why Pointers in C Programming?

1. Efficient Memory Access

- Allows direct access to memory locations.
- Enables manipulation of data using memory addresses.

2. Function Argument Handling

- Enables **call by reference**.
- Allows functions to **modify original variables** passed as arguments.

3. **Dynamic Memory Allocation**

- Used with functions like malloc(), calloc(), free() for runtime memory management.
- Helps build flexible programs that adapt to varying memory needs.

Why Pointers in C Programming?

4. Efficient Array and String Handling

- Arrays and strings are passed as pointers, avoiding copying large data.
- Enables **pointer arithmetic** for fast traversal and processing.

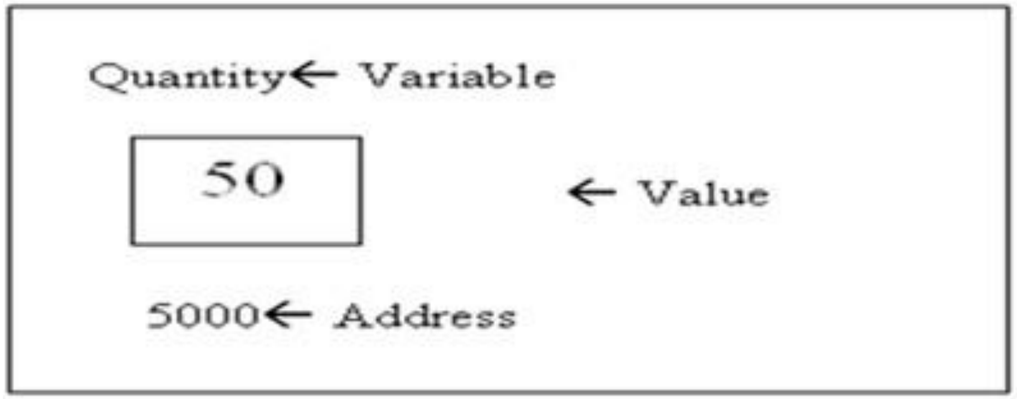
5. Building Complex Data Structures

- Essential for creating **linked lists**, **trees**, **graphs**, and other dynamic structures.

6. Low-Level System Programming

- Critical in systems programming, embedded systems, and interfacing with hardware.

Concept



int Quantity = 50;

- This statement instructs the system to find a location for the integer variable **Quantity** and puts the value **50** in that location.
- Assume that system has chosen address location 5000 for Quantity

Concept

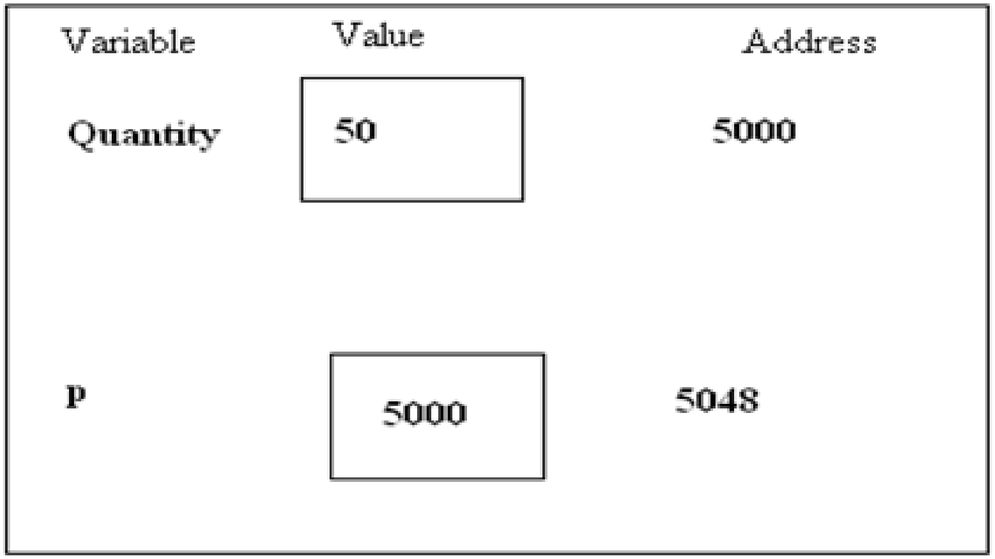
- During the execution of a program, the system always associates the name Quantity with a specific memory address — for example, **5000**.
  We can access the value **50** stored in Quantity either by using the **variable name Quantity** or by directly using its **memory address (5000)**.

- Since memory addresses are simply numbers, they can be assigned to variables just like any other data.
  To assign the address of Quantity (say, 5000) to a variable p, we write:

```
p = &Quantity;
```

- Variables that hold memory addresses are called **pointer variables**.

Concept



The operator **&** can be called **address of** and can be used only with a simple variable or an array element.

Pointer Operators and Their Usage in C

Operation	Syntax	Description
Creation	int *ptr;	Declares a pointer variable to hold an address of int
Address-of	&variable	Returns the memory address of variable
Dereferencing	*pointer	Returns the value stored at the memory address
Indirect Assignment	*pointer = val;	Stores val at the memory address pointed by pointer
Pointer Assignment	pointer2 = pointer1;	Copies address from one pointer to another

Pointer Basics in C: Address and Value Access

```
#include <stdio.h>
int main () {
   int var = 19;       // actual variable declaration
   int *ip;             // pointer variable declaration

   ip = &var;       // store address of var in pointer variable

   printf("Address of var variable: %x\n", &var);
   printf("Address stored in ip variable: %x\n", ip);
   printf("Value of *ip variable: %d\n", *ip);
   return 0;
}
```

Pointer Arithmetic

```
int x, *y, z, *q;
x = 3;
y = &x;                     // y points to x
printf("%d\n", x);          // outputs 3
printf("%d\n", y);          // outputs x's address, will seem like a random number to us
printf("%d\n", *y);         // outputs what y points to, or x (3)
printf("%d\n", *y+1);       // outputs 4 (print out what y points to + 1)
printf("%d\n", *(y+1));     // this outputs the item after x in memory – what is it?
z = *(&x);                  // z equals 3 (what &x points to, which is x)
q = &*y;                    // q points to 3 – note *& and &* cancel out
```

Pointer Manipulation with Variables and Arrays

```
int main() {
   int a = 5, b = 10, arr[5];
   int *ptr;
   ptr = &a;                          // ptr points to a
   b = *ptr;                          // b gets the value of a
   *ptr = 20;                          // a is now changed to 20
   ptr = &arr[0];                      // ptr points to the first element of arr
   *ptr = 1;                           // arr[0] is set to 1
   *(ptr + 1) = 2;                    // arr[1] is set to 2 using pointer arithmetic
   *(ptr + 2) = *(ptr) + *(ptr + 1);   // arr[2] = arr[0] + arr[1]
   printf("a = %d, b = %d\n", a, b);
   printf("arr[0] = %d, arr[1] = %d, arr[2] = %d\n", arr[0], arr[1], arr[2]);
   return 0;
}
```

Array Using a Pointer

```
int main() {
    int arr[4] = {12, 20, 39, 43};
    int *ptr;
    ptr = &arr[0];                  // ptr points to the beginning of the array
    printf("%d\n", arr[0]);         // prints 12
    printf("%d\n", *ptr);           // prints 12
    printf("%d\n", *ptr + 1);       // prints 13
    printf("%d\n", (*ptr) + 1);     // prints 13
    printf("%d\n", *(ptr + 1));     // prints 20
    ptr += 2;                       // ptr now points to arr[2]
    printf("%d\n", *ptr);           // prints 39
    *ptr = 38;                      // arr[2] is now 38
    printf("%d\n", *ptr - 1);       // prints 37
    ptr++;                          // ptr now points to arr[3]
    printf("%d\n", *ptr);           // prints 43
    (*ptr)++;                       // arr[3] becomes 44
    printf("%d\n", *ptr);           // prints 44
    return 0;
}
```

Try

- Write C program that utilizes pointers to increment the elements of an array

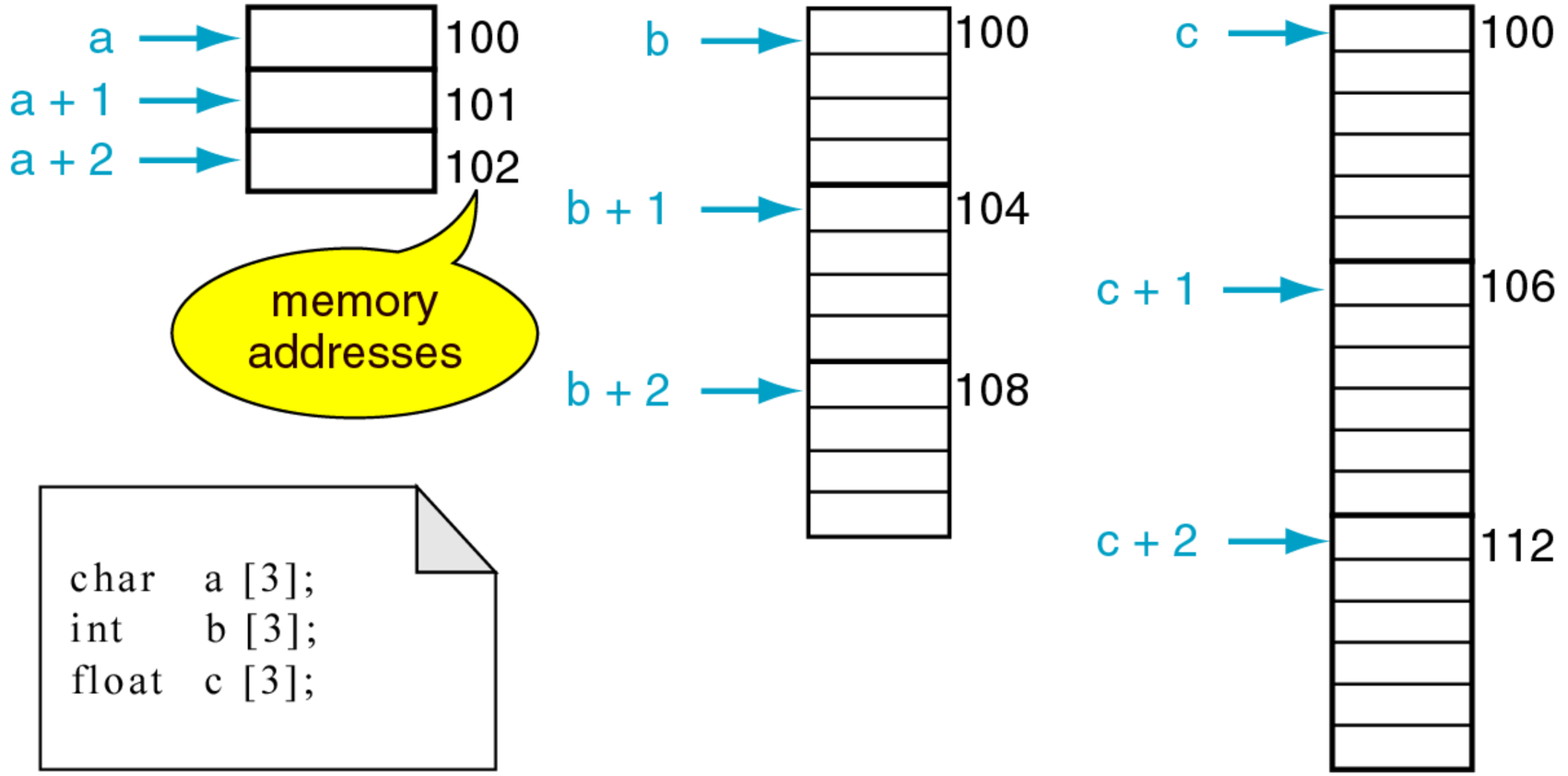
```c
#include <stdio.h>
void incrementArray(int *arr, int size) {
    for(int i = 0; i < size; i++) {
        *(arr + i) += 1;                    // Using pointer arithmetic to access array elements
    }
}
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    incrementArray(arr, size);
    for(int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Pointers and Strings

- The statement **char *cptr = name;** declares cptr as a **pointer to a character** and assigns it the **address of the first character** in the string name.
- The loop condition **while (*cptr != '\0')** continues until the **null terminator** ('\0') — marking the **end of the string** — is reached.
- When the while loop terminates, the pointer cptr points to the **null character** at the end of the string.
- The statement **length = cptr - name;** calculates the **length of the string** by subtracting the starting address (name) from the current pointer position (cptr).
- A **constant character string** (e.g., "Delhi") is treated as a **pointer to its first character**.
- The following statements are valid and demonstrate pointer assignment with string literals:

```
char *name;
name = "Delhi";
```

Pointer Arithmetic and different types



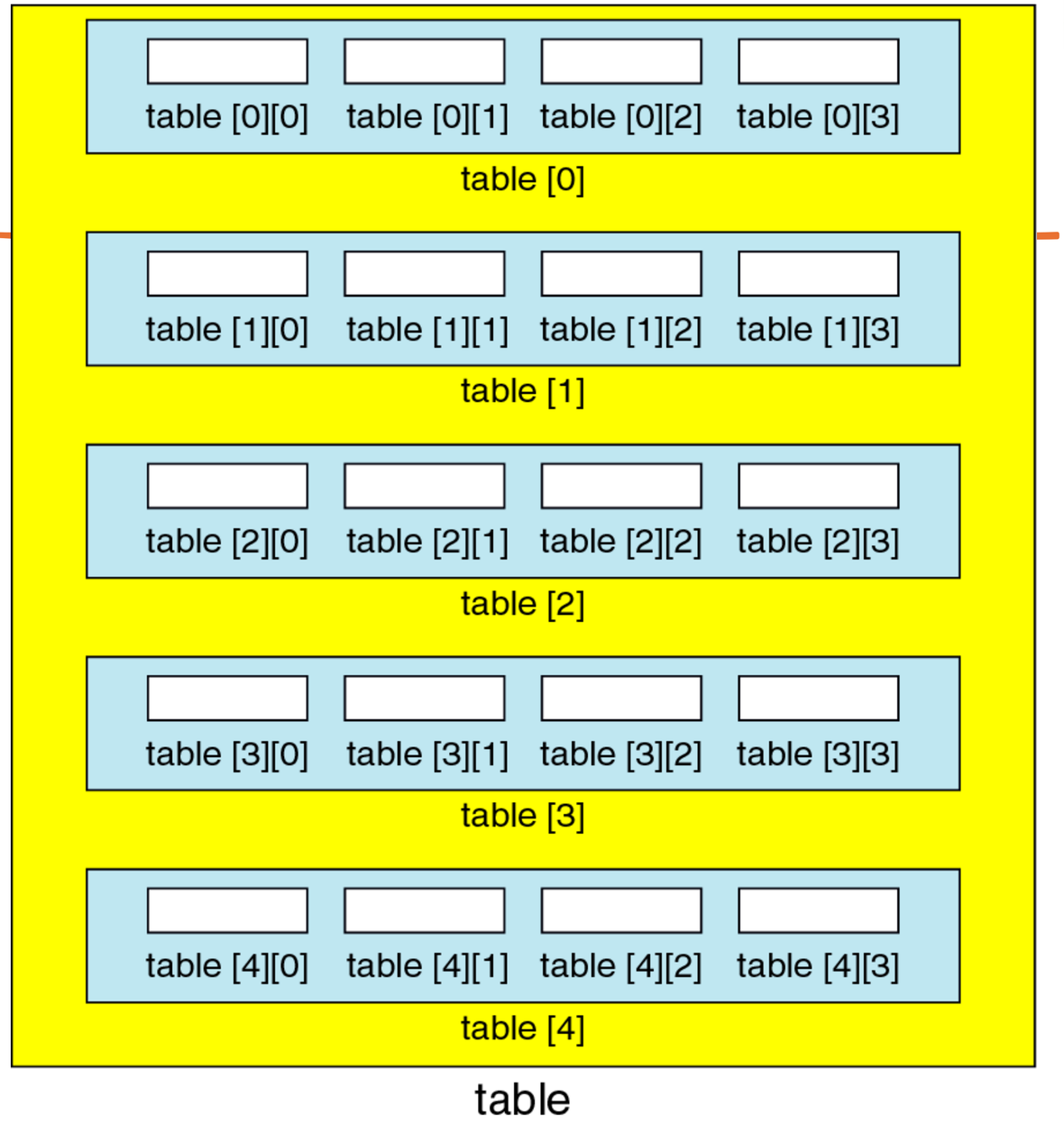
- Pointer arithmetic on different sized elements

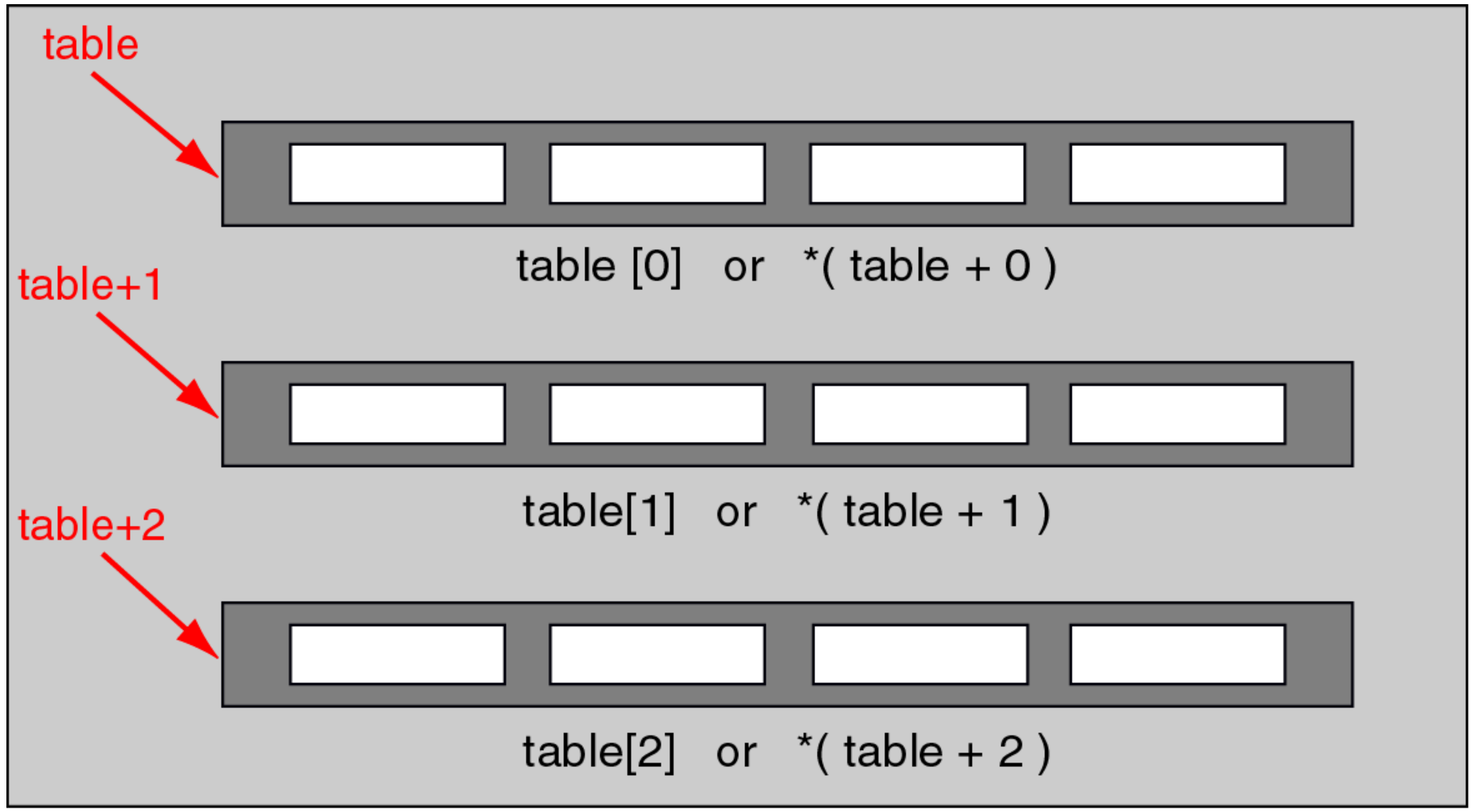
```c
#include <stdio.h>

int main() {
   // Using array notation
   char str1[] = “You are Welcome!";

   // Using pointer notation
   char *str2 = "Hello, Manipal!";

   // Printing both strings   ( the array name is a pointer itself!)
   printf("Array notation: %s \n", str1);
   printf("Pointer notation: %s \n", str2);

   // Accessing individual characters
   printf("str1[0] = %c\n", str1[0]);      // Array indexing
   printf("*str2 = %c\n", *str2);         // Pointer dereferencing

   // Modifying characters
   str1[0] = 'h';                    // Allowed: str1 is a modifiable array
   // str2[0] = 'h';                  // Not allowed: str2 points to a string literal

   printf("Modified str1: %s\n", str1);
// This would cause undefined behavior if uncommented
   return 0; // printf("Modified str2: %s\n", str2);
}
```

```c
// Pointer pointing to the array
   char *str3 = str1;

   // Print original string using both
   printf("Original str1: %s\n", str1);
   printf("Original str3: %s\n", str3);

   // Modify using array notation
   str1[0] = 'h';

   // Modify using pointer notation
   *(str3 + 8) = ‘w';

   // Print modified string using both
   printf("Modified str1: %s\n", str1);
   printf("Modified str3: %s\n", str3);
```

Pointers and 2D Array

- Just as in 1D array, the name of the array is a pointer constant to the first element of the array.
- Here, the first element is another array.
- Suppose, we have an array of integers.
- For a 2D array, when we dereference the array name, we get an array of integers (we do not get one integer).
- So, dereferencing of the array name of a 2D array is a pointer to a 1D array.





int table[3][4] ;

```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } /* for i */
```

Print table

Pointers and 2D Array

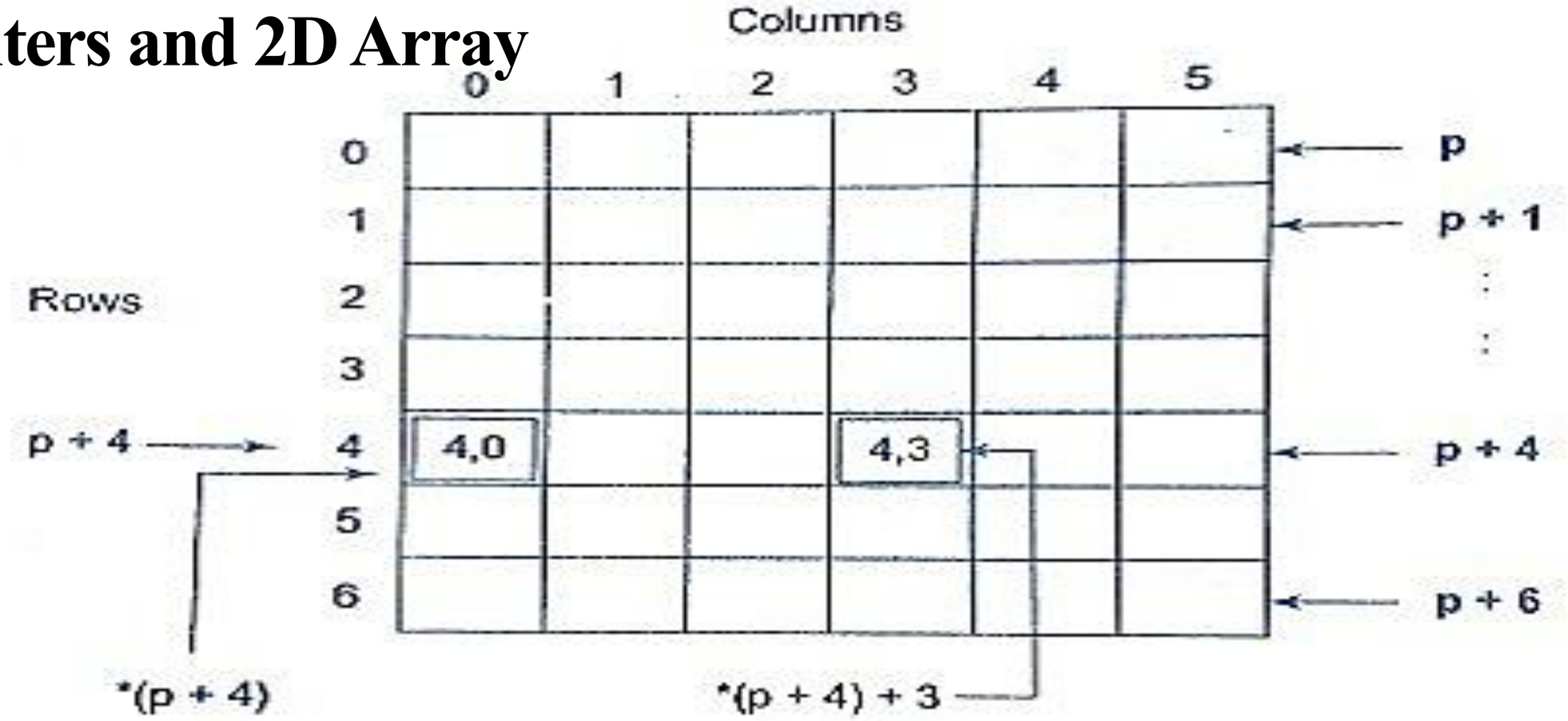
```
int a[][2]={{12, 22},
            {33, 44}};

int (*p)[2];
p=a;            // initialization
```

Element in 2d represented as

***(*(a+i)+j) or**

***(*(p+i)+j)**

Pointers and 2D Array



p	→	pointer to first row
p + i	→	pointer to ith row
*(p + i)	→	pointer to first element in the ith row
*(p + i) + j	→	pointer to jth element in the ith row
((p + i) + j)	→	value stored in the cell (i,j) (ith row and jth column)

Pointers and 2D Array

```
#include <stdio.h>
void main()
{
    int i, j, (*p)[2], a[][2]={   {12,  22},
                                  {33, 44}   };
    p=a;
    for( i=0; i<2;  i++)
    {
        for( j=0; j<2;  j++)
            printf("%d", *(*(p+i)+j));
    }
}
```

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions.

But logically the division is such that each function performs a specific task.

Defining a Function

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

Function Declarations

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

```
return_type function_name( parameter list );
```

- For the above defined function max(), the function declaration is as follows –

```
int max(int num1, int num2);
```

- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Example

```
/* function returning the max between two
numbers */

int max(int num1, int num2)
{

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Function

```
/* function declaration */
int max(int num1, int num2);
int main () {
   /* local variable definition */
   int a = 101;
   int b = 202;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
} // end of main
```

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
         result = num1;
   else
         result = num2;
   return result;
}
```

Function Arguments

1	Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

- While calling a function, there are two ways in which arguments can be passed to a function –

Call by value

In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.

```
void swap(int , int); //prototype of the function
int main() {
   int a = 101;
   int b = 202;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
    // printing the value of a and b in main
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);

// The value of actual parameters do not change by changing the formal parameters in call by value, a = 101, b = 202
}
```

Call by value cont.

```
void swap (int x, int y)
{
   int temp;
   temp = x;
   x=y;
   y=temp;
   printf("After swapping values in function x = %d, y
           = %d\n",x,y);
    // Formal parameters, a = 202, b = 101
}
```

Call by reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.

```
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 101;
   int b = 202;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
// printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
// The values of actual parameters do change in call by reference,
a = 101, b = 202
}
```

Call by reference

```
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d,
           b = %d\n",*a,*b);
// Formal parameters, a = 202, b = 101
}
```

Difference between call by value and call by reference

No	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Passing Array to Function

```
int minarray(int arr[],int size)
{
   int min=arr[0];
   int i=0;
    for(i=1;i<size;i++) {
              if(min>arr[i]) {
              min=arr[i];    }    }//end of for
   return min;
 }  //end of function
int main(){
  int i=0,min=0;
  int numbers[]={42,53,75,32,84,91};    //declaration of array
  min=minarray(numbers,6);              //passing array with size
  printf("minimum number is %d \n",min);
  return 0;
}
```

Functions- Categories

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and with return value.
4. Functions with no arguments but return a value.

Functions with no arguments and no return values.

```
// Function prototype
void sum();  // void return type, no parameters
int main() {
    sum();  // Just call the function
    return 0;
}
// Function definition
void sum() {
    int a, b, result;
    printf("Enter 2 numbers: ");
    scanf("%d %d", &a, &b);
    result = a + b;
    printf("Sum = %d\n", result);
}
```

Functions with arguments and no return values.

```
// Function prototype
void sum(int, int);  // void return type, with parameters
int main() {
    int a, b;
    printf("Enter 2 numbers: ");
    scanf("%d %d", &a, &b);
    sum(a, b);  // Call the function — no return value
    return 0;
}
// Function definition
void sum(int a, int b) {
    int result = a + b;
    printf("Sum = %d\n", result);
}
```

Functions with arguments and one return value.

```
// Function prototype
int sum(int, int);
int main() {
   int a, b, c;
   printf("Enter 2 numbers: ");
   scanf("%d %d", &a, &b);
   c = sum(a, b);  // Passing arguments to function
   printf("Sum = %d\n", c);
   return 0;
}
// Function definition
int sum(int a, int b) {
   int c;
   c = a + b;
   return c;  // One return value
}
```

Functions with no arguments with return value.

```
// Function prototype
int sum();
int main() {
   int c;
   c = sum();  // Function call (no arguments)
   printf("Sum = %d\n", c);
   return 0;
}
// Function definition
int sum() {
   int a, b, c;
   printf("Enter 2 numbers: ");
   scanf("%d %d", &a, &b);
   c = a + b;
   return c;  // Return the result to main
}
```

Passing 1D-Array to Function

```
// Function prototype
int sumArray(int arr[], int size);
int main() {
    int arr[100], n, i;
    printf("Enter the number of
                elements in the array: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);     }
    int total = sumArray(arr, n);  // Pass array to function
    printf("Sum of array elements = %d\n", total);
    return 0;
}
```

```
// Function definition
int sumArray(int arr[], int size) {
    int sum = 0, i;
    for (i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;   }
```

Passing 2D-Array to Function

```
// Function to compute sum of 2D array

int fn2d(int x[][10], int m, int n) {
   int i, j, sum = 0;
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         sum += x[i][j];
   return sum;
}
```

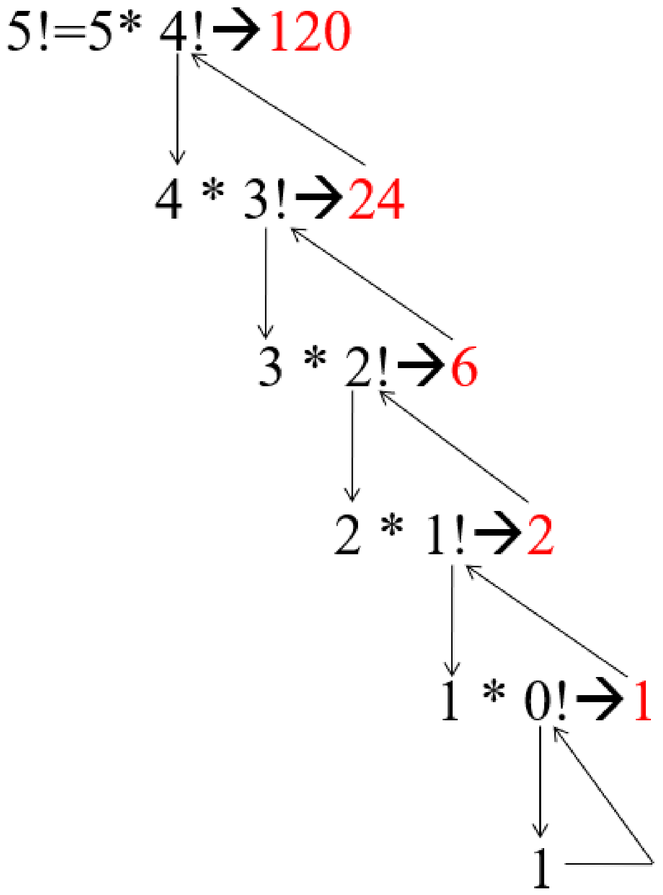
```
int main() {
   int i, j, m, n, a[10][10];
   printf("Enter number of rows and columns
            (max 10 each): ");
   scanf("%d %d", &m, &n);
   printf("Enter the elements of the
            matrix:\n");
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++) {
         printf("a[%d][%d]: ", i, j);
         scanf("%d", &a[i][j]);
      }
   }
   printf("Sum of elements of 2D array is
            %d\n", fn2d(a, m, n));
   return 0;
}
```

Recursion

Recursion is the process which comes into existence when a function **calls a copy of itself** to work on a smaller problem. Any function which calls itself is called **recursive function**, and such function calls are called **recursive calls.**

```
int fact (int);
int main() {
   int n,f;
   printf("Enter the number whose factorial you want to calculate?");
   scanf("%d",&n);
   f = fact(n);
   printf("factorial = %d",f); }
int fact(int n) {
   if (n==0)    {    return 0; }
   else if ( n == 1)  {     return 1;  }
   else    {   return n*fact(n-1);  }
}
```

Ex:



Binary Search Using Recursion

```
binarySearch(arr[], start, end, target)
{
    if (end >= start)  {
        mid = start + (end - start) / 2
        if (arr[mid] == target)
              return mid
        if (arr[mid] > target)
              return binarySearch(arr, start, mid-1, target)
        return binarySearch(arr, mid+1, end, target)
    }
  // If the target is not present in the array, return -1
    return -1
}
```

Try

- Write a C program to simulate the working of the Tower of Hanoi problem using recursion for n disks

Memory allocation functions

Static memory allocation

- memory is allocated at compile time.
- memory can't be increased while executing program.
- used in array.

Dynamic memory allocation

- memory is allocated at run time.
- memory can be increased while executing program.
- used in linked list.

Dynamic memory allocation

- The concept of **dynamic memory allocation in C language** *enables the programmer to allocate memory at runtime*.
- Dynamic memory allocation in C language is possible by 4 functions of <stdlib.h> header file.

- **malloc()**
- **calloc()**
- **realloc()**
- **free()**

Methods used for dynamic memory allocation

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is given below:

ptr = (cast-type*) malloc (byte-size) ;

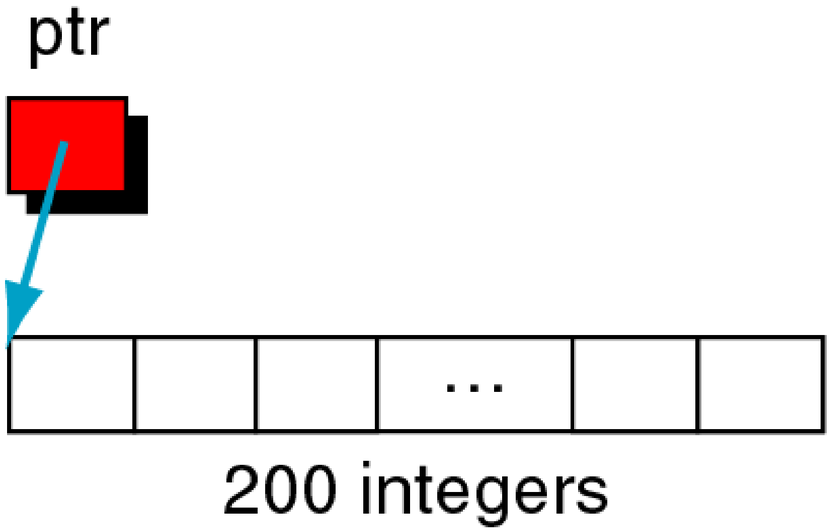
Example of malloc() function

```
int main( ) {
 int n,i,*ptr,sum=0;
  scanf("%d",&n);
  ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
  if(ptr==NULL)     {
     printf("Sorry! unable to allocate memory");
     exit(0);
  }
  printf("Enter elements of array: ");
  for(i=0;i<n;++i)    {
     scanf("%d",ptr+i);
     sum+=*(ptr+i);
   }
  printf("Sum=%d",sum);
  free(ptr);
  return 0;
}
```

calloc() function

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:

ptr = (cast-type*) calloc (number, byte-size) ;




```
if (!(ptr = (int *)calloc (200, sizeof(int))))
    /* No memory available */
    exit (100) ;

/* Memory available */
...
```

- Allocating memory for an array of 200 integers.

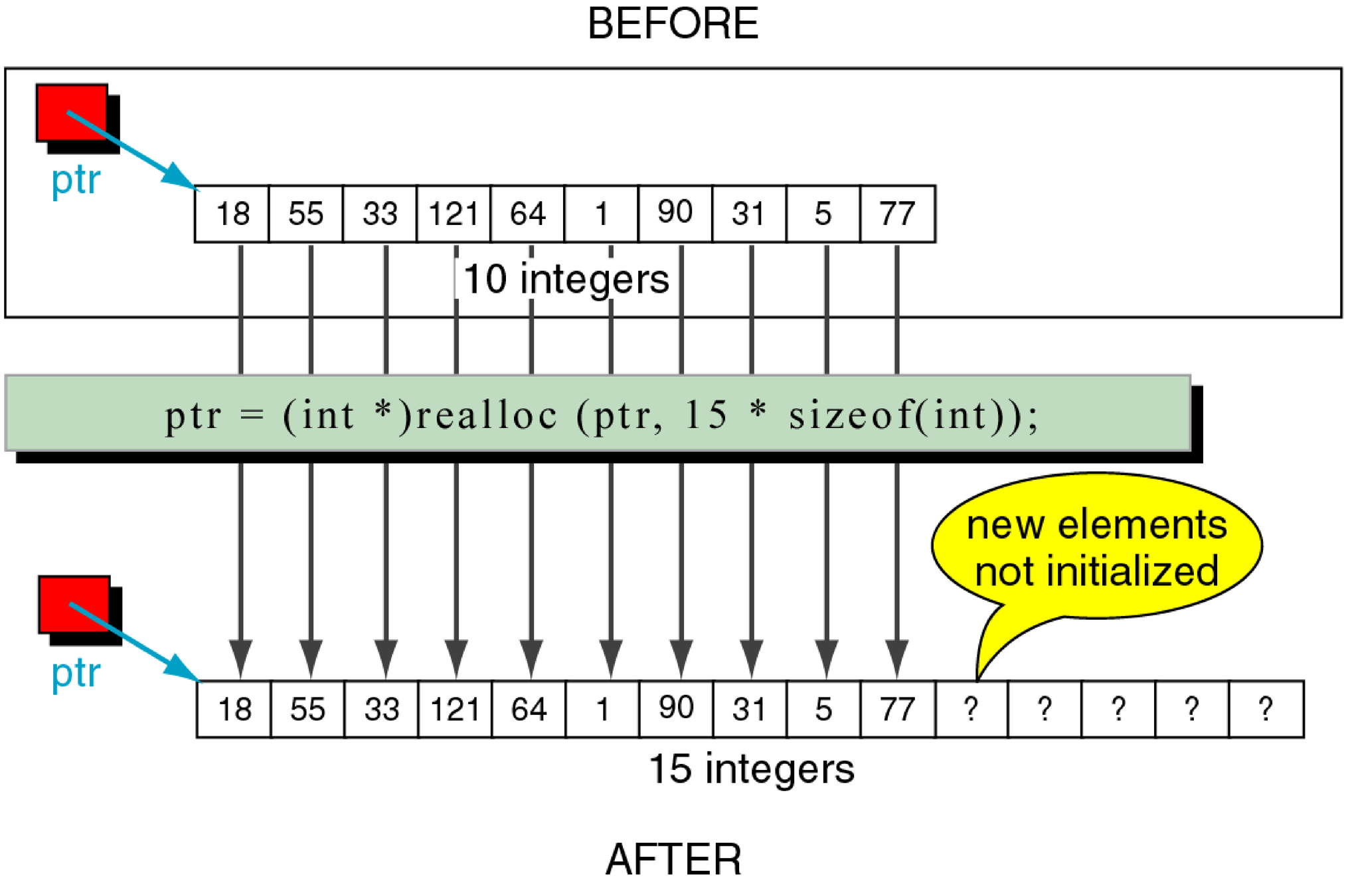
Example of calloc() function

```
int main( ) {
 int n,i,*ptr,sum=0;
  scanf("%d",&n);
 ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
  if(ptr==NULL)      {
     printf("Sorry! unable to allocate memory");
     exit(0);
  }
  printf("Enter elements of array: ");
  for(i=0;i<n;++i)    {
     scanf("%d",ptr+i);
     sum+=*(ptr+i);
  }
  printf("Sum=%d",sum);
 free(ptr);
 return 0;
}
```

realloc() function

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.
- Highly inefficient and needs to be used carefully.
- Given a pointer to a previously allocated block of memory, realloc changes the size of the block by deleting or extending the memory at the end of the block.
- If the memory cannot be extended (due to other allocations), realloc allocates a completely new block.
- Copies the existing memory allocation to the new allocation and deletes the old allocation.
- The programmer must ensure that other pointers to the data are correctly changed
- **syntax** of realloc() function

ptr = realloc(ptr , new-size) ;

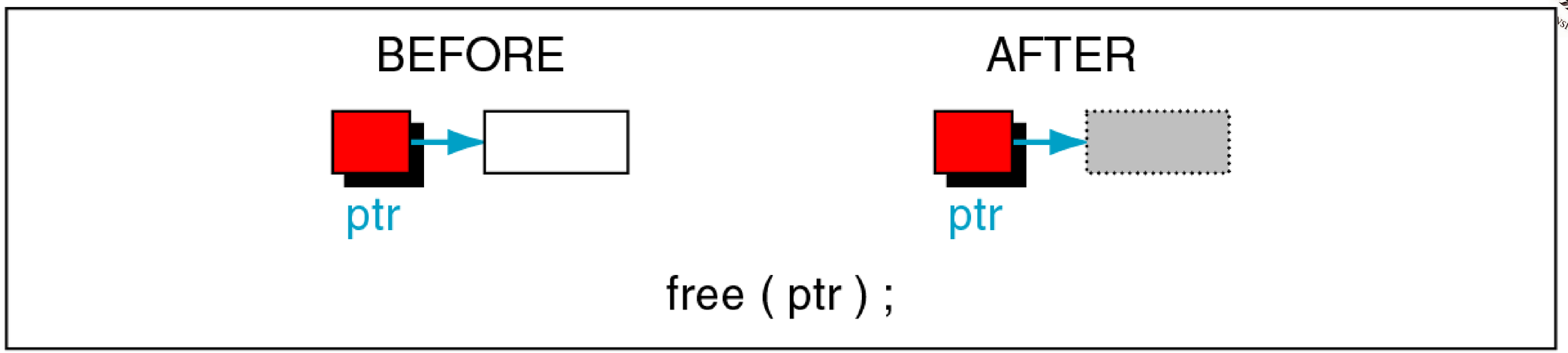

BEFORE
ptr
18
55
33
121
64
1
90
31
5
77
10 integers
ptr = (int *)realloc (ptr, 15 * sizeof(int));
new elements
not initialized
ptr
18
55
33
121
64
1
90
31
5
77
?
?
?
?
?
15 integers
AFTER

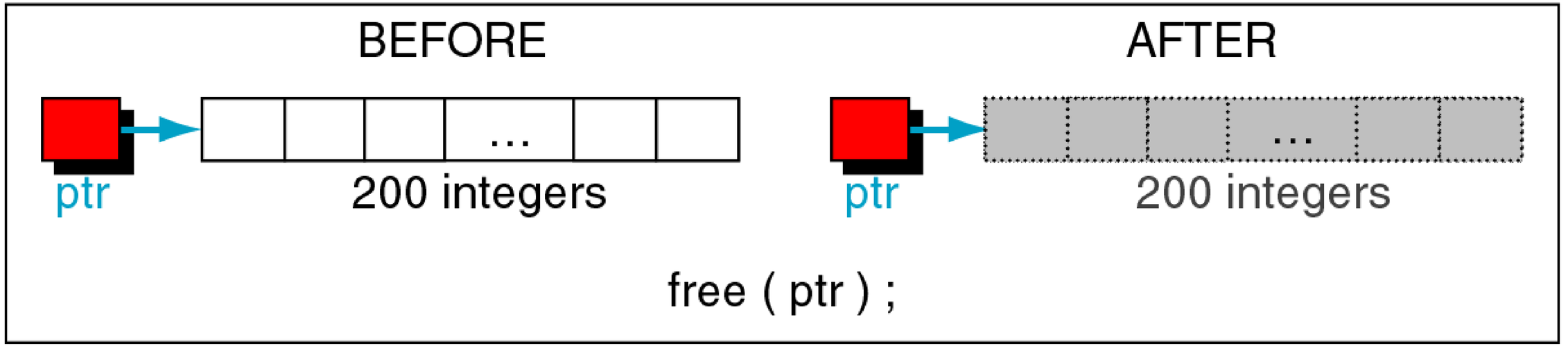
free() function

- The memory occupied by malloc() or calloc() functions must be released by calling free() function.
- Otherwise, it will consume memory until program exit.
- **Syntax** of free() function.

free(ptr) ;

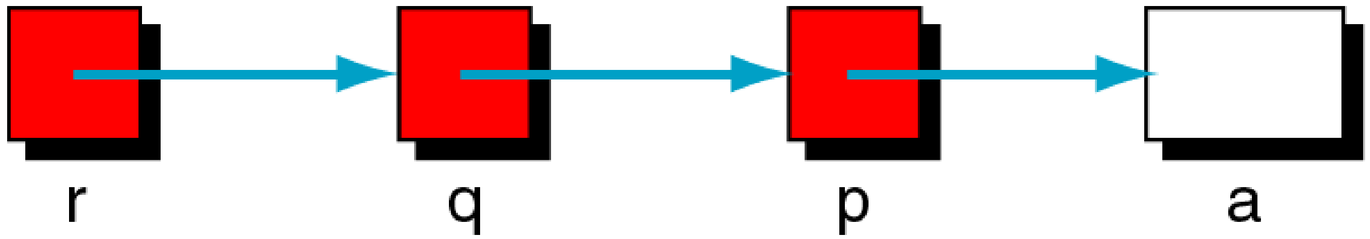
- It is an error to free memory-
  - –With a null pointer or,
  - –A pointer to other than the first element of an allocated block,
  - –A pointer that is a different type than the pointer that allocated the memory,
  - –To refer to memory after it has been released.





- Ex1: to release a single element allocated with a malloc back to the heap
- Ex2: : to release a 200 element array allocated with a calloc, all 200 elements are returned back to the heap

Pointer to Pointer



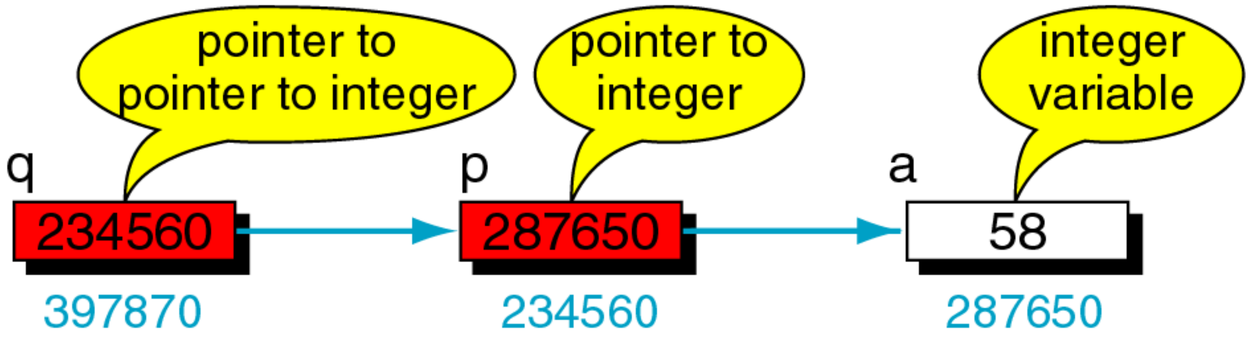
- All pointers have been pointing directly to data.
- Advanced data structures require pointers that point to other pointers.
- Ex : we can have a pointer pointing to another pointer pointing to an integer.
- You can have multiple levels on indirections.

Note: Although many levels of indirection can be used but practically not more than two levels are needed

Pointers to pointers

- Each level of pointer indirection requires a separate indirection operator when it is dereferenced.
- Ex: To refer to a using the pointer p, we have to dereference it once as:
  **p*
- To refer to a using the pointer *q*, we need to dereference it twice to get to the integer *a* as there are 2 levels of indirection (pointers) involved.
- By first dereference, we reference *p*, which is a pointer to an integer.
- *q* is a pointer to a pointer to an integer.
  ***q*

```
/* Local Declarations */
int       a;
int       *p;
int       **q;
```




```
/* Statements */
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

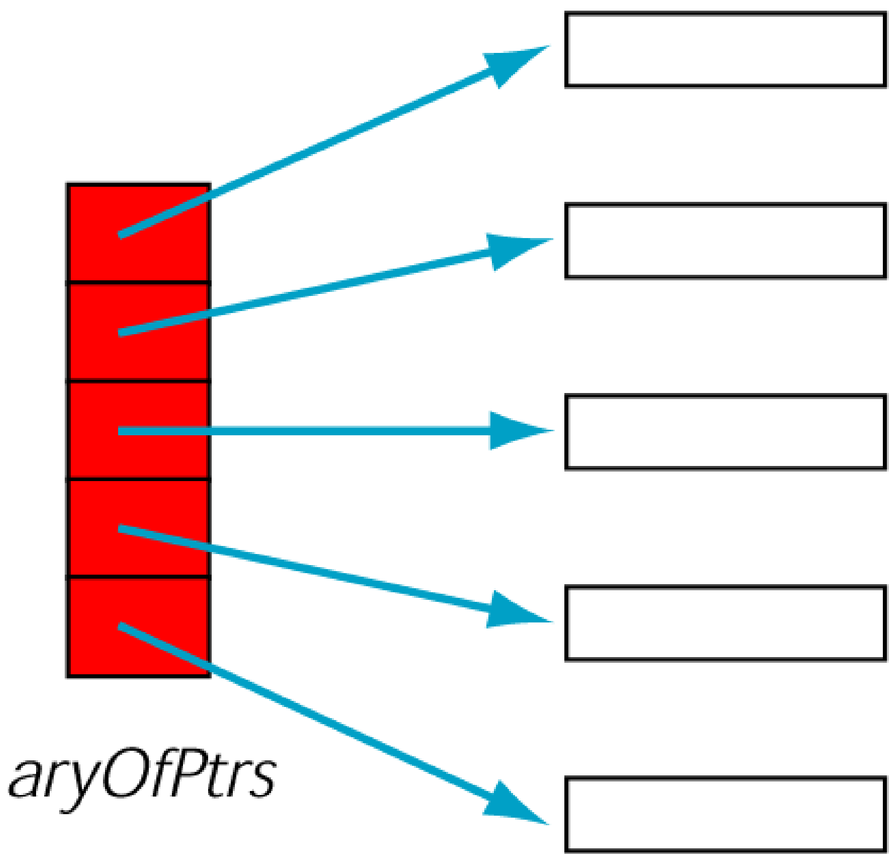
Array of Pointers

- Useful data structure.
- Needed when the size of the data in the array is variable.

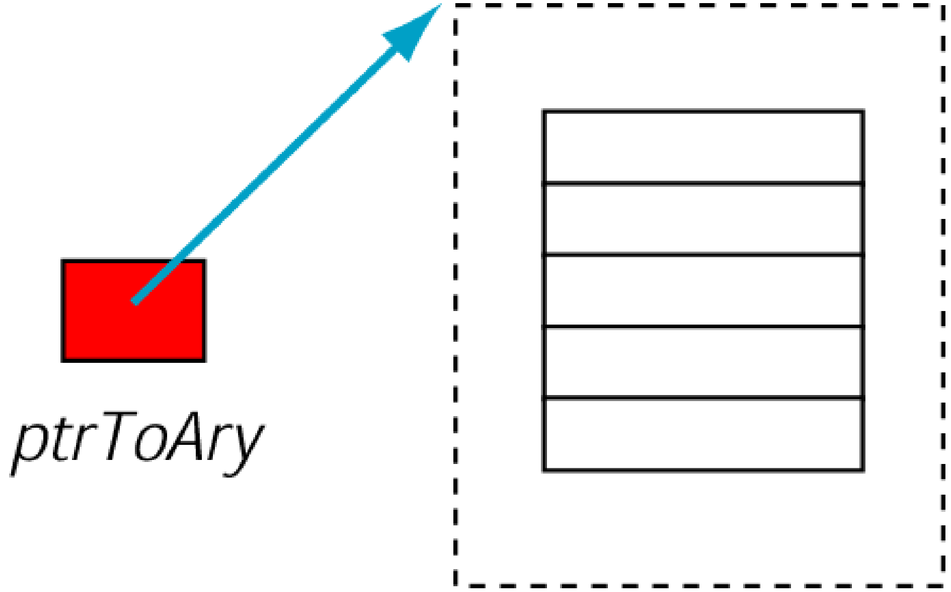
Ragged arrays (Note: For Your Reference Only)

- 2D arrays with an uneven right border.
- The rows contain different number of elements (size zero to max).
- 2d array wastes a lot of memory for this structure.
- So, create n number of 1D arrays that are joined through array of pointers.

Array of pointers vs. Pointer to Array



(a) An array of pointers



(b) A pointer to an array

Dynamic 2D Array Allocation in C using Pointer to Pointer

```
int **A;
A = (int **)malloc(2 * sizeof(int *));  // Step 1: allocate 2 row pointers
for (int i = 0; i < 2; i++)
{
    A[i] = (int *)malloc(2 * sizeof(int));  // Step 2: each row has 2 int
}
```

Creating a 2D array dynamically

```
int** createMatrix(int rows, int cols) {
   int **table = (int **) calloc (rows, sizeof(int *));
   if (table == NULL) {
      printf("Memory allocation failed. Exiting...\n");
      exit(1);                                  // 1 means abnormal termination
   }
   for (int i = 0; i < rows; i++) {
            table[i] = (int *)calloc(cols, sizeof(int));
            if (table[i] == NULL) {
                  printf("Memory allocation failed. Exiting...\n");
                   exit(1);
            }
   }

   return table;
}
```

Question & Discussion

- How are 2D arrays represented in memory?
- How do you dynamically allocate memory for a 2D array using malloc?
- How do you access elements of a 2D array using pointer dereferencing?

STRINGS

Strings

Definition

- A string is an array of characters.
- Any group of characters (except double quote sign) defined between double quotation marks is a constant string.
- Character strings are often used to build meaningful and readable programs.
- The common operations performed on strings are
  - Reading and writing strings
  - Combining strings together
  - Copying one string to another
  - Comparing a string with another
  - Extracting a portion of a string ..etc.

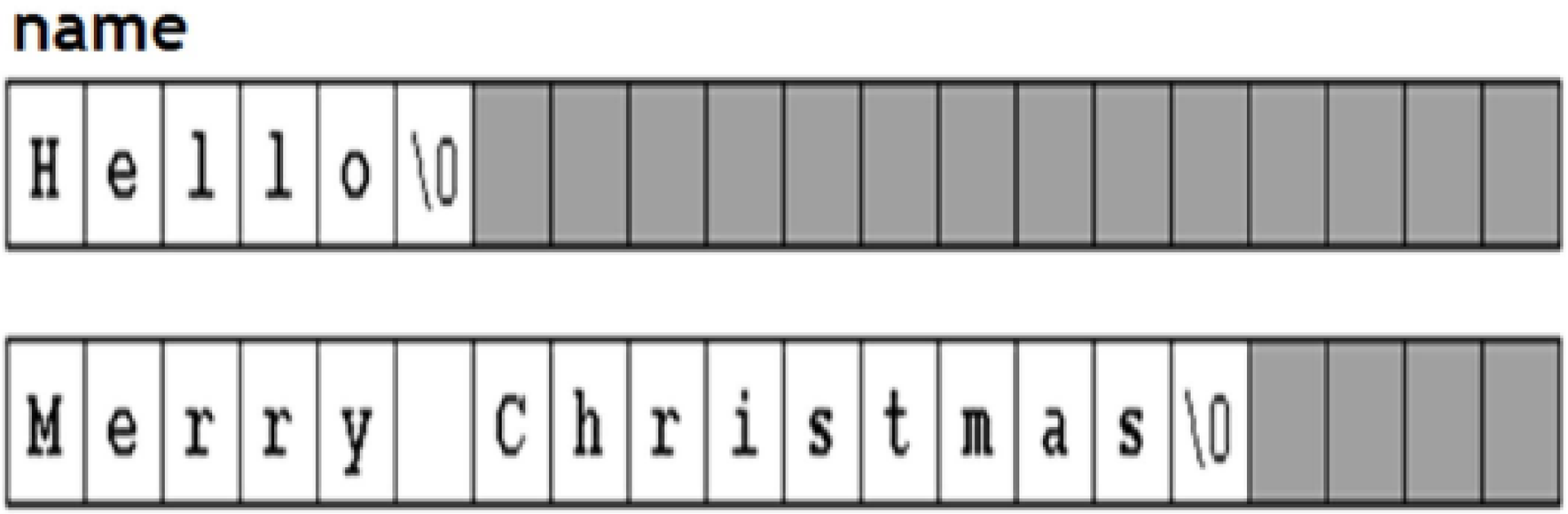
Strings

- Declaration and initialization

  char string_name[size];

The size determines the number of characters in the string_name.

- ✓ The character sequences "Hello" and "Merry Christmas" represented in an array name as follows :



Difference between scanf() and gets()

- scanf() stops reading when a space is found but gets() reads space.
- Thus scanf() can be used to read a word and gets() can be used to read a sentence.

```
int num;
char str[100];
scanf("%d %s", &num, str);
printf(“Entered Data is =%s”,str)
gets(str)
printf(“Entered Data is =%s”,str)
```

Library functions: String Handling functions(built-in)

- These in-built functions are used to manipulate a given string.
- These functions are part of **string.h** header file.
  - **strlen ( )**
    - ✓ gives the length of the string. E.g. strlen(string)
  - **strcpy ( )**
    - ✓ copies one string to other. E.g. strcpy(Dstr1,Sstr2)
  - **strcmp ( )**
    - ✓ compares the two strings. E.g. strcmp(str1,str2)
  - **strcat ( )**
    - ✓ Concatinate the two strings. E.g. strcat(str1,str2)

Length of string without using string function

```
int stringLength(char str[]) {
    int len = 0;
    while (str[len] != '\0') {
        len++;
    }
    return len;
}
```

String Concatenation without using string function

```
void stringConcat(char str1[], char str2[], char
result[]) {
    int i = 0, j = 0;
    while (str1[i] != '\0') {
        result[i] = str1[i];
        i++;
    }
    while (str2[j] != '\0') {
        result[i++] = str2[j++];
    }
    result[i] = '\0';
}
```

String Comparison without using string function

```
int stringCompare(char str1[], char str2[]) {
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0') {
        if (str1[i] != str2[i])
            return 0; // Not equal
        i++;
    }
    return (str1[i] == '\0' && str2[i] == '\0');
}
```

Insert Sub String without using string function

```
void insertSubstring(char mainStr[], char subStr[], int pos, char result[]) {
    int i = 0, j = 0;

    // Copy characters before position
    while (i < pos) {
        result[i] = mainStr[i];
        i++;
    }

    // Insert substring
    while (subStr[j] != '\0') {
        result[i++] = subStr[j++];
    }

    // Copy remaining characters
    j = pos;
    while (mainStr[j] != '\0') {
        result[i++] = mainStr[j++];
    }

    result[i] = '\0';
}
```

Try

- Delete Sub String without using string function

STRUCTURES

Structures

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling
- The variables in a structure can be int, float, and so on.
- This is unlike the array, in which all the variables must be the same type.
- The data items in a structure are called the *members* of the structure.

Structures

- Structures are user-defined aggregate types.
- They assist program organisation by
  - Grouping logically related data, and giving this set of variables a higher-level name and more abstract representation.
  - Enabling related variables to be manipulated as a single unit rather than as separate entities.
  - Reducing the number of parameters that need to be passed between functions.
  - Providing another means to return multiple values from a function.

Structures

The general format of a structure **definition** is

```
struct  tag_name
{
  Data_type member1;
  Data_type member2;
  …
};
```

e.g.

```
struct student
    {   int rollno;
        int age;
        char name[10];
        float height;
    };
```

Declaring Structure Variables

There are several equivalent ways to define/declare variables of a particular structure type.

Declare them at the structure definition

```
struct student
  { int rollno;
    int age;
    char name[10];
    float height;
}s1, s2, s3;
 /* Define 3 variables */
```

OR

```
struct student
{   int rollno;
    int age;
    char name[10];
    float height;
};
```

Define the variables at some point *after* the structure definition

```
struct student s1, s2, s3;
```

Declaring Structure Variables

Defining a Structure Variable in **main()** without using **struct** tag

- It can be in the main() as, definition of the structure should be done before,

```
student s1;
```

```
struct student
  { int rollno;
    int age;
    char name[20];
    };
```

Note:

Members of a structure themselves are not variables.

i.e. rollno alone does not have any value or meaning.

Member or dot operator

- The link between member and a structure variable is established using the member operator '.' which is also known as 'dot operator'

e.g.

```
s1. rollno;
s1. age;
s1. name;
```

Giving values to members

How to assign values to the members of student s1

```
strcpy(s1.name, "Rama");
s1.rollno = 1335;
s1.age = 18;
s1.height = 5.8;
```

```
struct student
  { int rollno;
    int age;
    char name[20];
    float height;
    }s1;
```

Structure: Example

```
struct book  {  // declaration
   char title[20];
   char author[15];
   int pages;
   float price;
   };
```

OR

```
struct {  // declaration
    char title[20];
    char author[15];
    int pages;
    float price;
    } b1,b2,b3;
```

```
void main( ){
struct book b1, b2, b3;
printf("Input values");
scanf("%s%s%d%f", b1.title, b1.author, &b1.pages, &b1.price);
//output
printf("%s%s%d%f", b1.title, b1.author, b1.pages, b1.price);
}
```

Structure initialization

```
main ( ){
  struct
    {  int rollno;
       int age;
       }stud ={20, 21};
       …
}
```

There is one-to-one correspondence between the members and their initializing values.

1. First one without tag name.

```
main ( ){
  struct stud
    {  int rollno;
       int age;
       };
       struct stud s1={20, 21};
       struct stud s2={21, 21};
}
```

2. Second one uses a tag name.

Structure initialization

```
struct stud
   {  int rollno;
      int age;
      }s1={20, 21};

   main ( )
   {
      struct stud s2={21, 21};
      …
      …
   }
```

3. Third one uses a tag name and defined outside the function.

Assign and Compare structure variables

If **student1** and **student2** belong to the same structure, then the following operations are valid:

1. `student1 = student2` // Assign **student2** to **student1**
2. `if(student1 .rollno == student2.rollno )` //comparison

   return 1 if they are equal,

   0 otherwise.

3. `if(student1 .rollno != student2 .rollno )` OR

   return 1 if they are not equal,

   0 otherwise.

The comparison should be done with respect to any member variables.

Assigning and Comparing structure variables : example

```
struct class1
{
int rollno;
char name[20];
float marks;
};
void main()
{
class1 s1={111, "Joe", 72.50};
class1 s2={222, "Rishi", 67.00};
class1 s3;
```

```
s3=s2; // assignment : s2 to
  s3


//comparison
if((s3. rollno ==s2.
   rollno)&&(s3.marks ==
   s2.marks))
   printf"Student3 and student2
      are same\n");
else
   printf"Student3 and student2
      are NOT same\n");
}
```

Operation on Individual members

Individual members are identified using **member operator**, the **dot operator.**

A member with the dot operator along with its structure variable can be treated like any other variable name.

```
if (s1.rollno  ==  111)
    s1.marks  +=  10.0;

float sum = s1.marks + s2.marks;

s1.rollno ++;
++ s1.rollno;  //applicable to numeric type members
```

The precedence of the **member operator** is higher than all arithmetic and relational operators and therefore no parentheses are required.

To be solved …

- Define a structure type, **`struct personal`** that would contain **`person name`**, **`date of joining`** **`(only day(int))`** and **`salary`**.

  Using this structure write a program to read the information for 3 persons from the keyboard and print all the details of person having **`highest salary`**.

Solution:

```
struct person  {
      char name[15];
      int doj;
      float sal;
      };
void main(){
struct person p1, p2, p3;
printf("Input values for person 1:- \n");
scanf("%s%d%f",p1.name,&p1.doj,&p1.sal;
printf("Input values for person 2:- \n");
scanf("%s%d%f",p2.name,&p2.doj,&p2.sal;
printf("Input values for person 2:- \n");
scanf("%s%d%f",p3.name,&p3.doj,&p3.sal;
```

Solution:

```
if (p1.sal>p2.sal && p1.sal >p3.sal){
   printf"Name: %s\n",p1.name);
   printf("Date of Joining: %d",p1.doj);
   printf("Salary:%f\n", p1.sal);}
else  if (p2.sal> p1.sal && p2.sal > p3.sal){
   printf"Name: %s\n",p2.name);
   printf("Date of Joining: %d",p2.doj);
   printf("Salary:%f\n", p2.sal);}
Else{
   printf"Name: %s\n",p3.name);
   printf("Date of Joining: %d",p3.doj);
   printf("Salary:%f\n", p3.sal);}
}
```

Problems...

Write programs to

1. Create a student record with name, rollno, marks of 3 subjects (m1, m2, m3). Display the details of the students in ascending order of their average marks.
2. Create an employee record with emp-no, name, age, date-of-joining (year), and salary. If there is 20% hike on salary per annum, compute the retirement year of each employee and the salary at that time. [standard age of retirement is 55]

Array of structures

We can define single or multidimensional arrays as structure variables.

```
struct marks
  {
        int  subject1;
        int  subject2;
        int  subject3;
  } ;
```

struct marks student[84];

- defines an array called student, that consists of 84 elements.
- Each element is defined to be the type struct marks.

Array of structures

```
struct marks
  {
      int subject1;
      int subject2;
      int subject3;
  } ;
main(){
  struct marks student[3]={ {45,47,49},
                            {43,44,45},
                            {46,42,43} };
```

This declares the student as an array of three elements student[0], student[1], student[2].

Array of structures

The members can be initialized as

```
student[0].subject1 = 45;
student[0].subject2= 47;
…
…
student[2].subject3= 43;
```

Stored in the memory as

```
student[1].subject2
```

refer to marks obtained in the second subject by the second student.

	Memory
student[0].subject1	**45**
student[0].subject2	**47**
student[0].subject3	**49**
student[1].subject1	**43**
student[1].subject2	**44**
student[1].subject3	**45**
student[2].subject1	**46**
student[2].subject2	**42**
student[2].subject3	**43**

Arrays within Structures

We can define single or multidimensional arrays inside a structure.

```
struct marks
   {   int  rollno;
       float  subject[3];
   } student[2] ;
```

The member subject contains 3 elements; subject[0], subject[1] & subject[2].

student[1].subject[2];

- Refer to the marks obtained in the third subject by the second student.

Arrays within structures : example

```
struct marks{
int total;
int sub[3];
};
void main(){
  marks student[3] ={{0,45,47,49}, {0,43,44,45}, {0,46,42,43}};
  int i, j ;

  for(i=0;i<=2;i++)  {
    for(j=0;j<=2;j++)
      student[i].total+=student[i].sub[j]; //students total
      }
```

Arrays within structures : example

```
Printf("Grand Total of each student.");

for(i=0;i<=2;i++)
   printf("Total of student["%d"]=%d",i,student[i].total;

}
```

Structures within Structures

Structure within structure means nesting of structures.

Consider the following structure defined to store information about students

```
struct student{
    int rollno;
    char name[15];
    struct { // marks for 3 subjects under structure marks
            int sub1;
            int sub2;
            int sub3;
     }marks;
}fs[3]; //3 students
```

Structures within Structures

```
struct m{
        int sub1;
        int sub2;
        int sub3;       };
```

Tag name is used to define inner structure **marks**

```
struct student{
        int rollno;
        char name[15];
        struct m marks;
      }fs[3];
```

The members contained in the inner structure namely sub1, sub2 and sub3 can be referred to as:

```
fs[i].marks.sub1;
fs[i].marks.sub2;
fs[i].marks.sub3;
```

Structures and functions

```
void read(book []); // prototype
void main() {
int i;
struct book b1[2];
printf("Enter ISBN, Author name & Price \n");
   read(b1); // function call

printf("The book details entered:\n");
for(i=0;i<2;i++){
   printf("%d Book:", i+1);
   printf("ISBN: %d",b1[i].isbn;)
   printf("Author: %s", b1[i].author);
   printf("Price: %f", b1[i].price);
  }
}
```

```
struct book
      {
      int isbn;
      char author[15];
      float price;
      };
```

```
function
void read(book a[])
{
 int i;
 for(i=0;i<2;i++){
   scanf("%d", &a[i].isbn));
   scanf("%s", a[i].author);
   scanf("%f",a[i].price);
  }
}
```

Structures and functions

```
void main() {
    int i;
    struct book b1[2];

    printf("Enter details for 2 books:\n");

    for(i=0; i<2; i++) {
        read(&b1[i]);
        printf("Book details entered:\n");
        printf("ISBN: %d\n", b1[i].isbn);
        printf("Author: %s\n", b1[i].author);
        printf("Price: %.2f\n", b1[i].price);
    }
}
```

```
struct book {
    int isbn;
    char author[15];
    float price;
};
```

function

```
void read(struct book *b) {
    printf("Enter ISBN, Author name & Price in:\n");
    scanf("%d %s %f", &b->isbn, b->author, &b->price);
}
```

Pointers and Structures

```
struct inventory
{
  char name[30];
  int number; float price;
}product[2], *ptr;
```

ptr=product;

Its members are accessed using the following notation

ptr→name ptr→number

ptr→price

Pointers and Structures

- The symbol → is called **arrow operator** (also known as member selection operator)
- The data members can also be accessed using
  - **(*ptr).number**
  - Parentheses is required because '.' has higher precedence than the operator *

Function Returning Pointers

```
int *larger (int *, int  *);
void main()
{
 int a=10, b=20, *p; p=larger (&a, &b);
 printf("%d",*p);
}
int *larger (int *x, int  *y)
{
 if (*x > *y)
 return(x); else
     return(y);
}
```

Try

Define a Student structure with:

- char name[50]
- int roll_no
- float marks

Write functions to:

- Read student data using pointer to structure
- Display student data using pointer to structure
- Find and display the student with highest marks using pointer-based access

Self-Referential Structures

Definition:

- A **structure that includes a pointer to itself**.
- Enables creation of **dynamic data structures** like:
  - Linked Lists, Stacks, Queues, Trees.

Key Features:

- Supports **dynamic memory management** using malloc() and free().
- Allows data objects to **grow or shrink at runtime**.
- Efficient for **insertion/deletion** operations.

Example

```
struct list_node {
  char data;
  struct list_node *link;
};
```

- ***Using typedef for clarity:***

```
typedef struct list_node *list_pointer;
struct list_node {
  char data;
  list_pointer link;
};
```

- ***Linking Nodes***

```
item1.link = &item2;
item2.link = &item3;
item3.link = NULL;
```

SPARSE MATRICES

Introduction

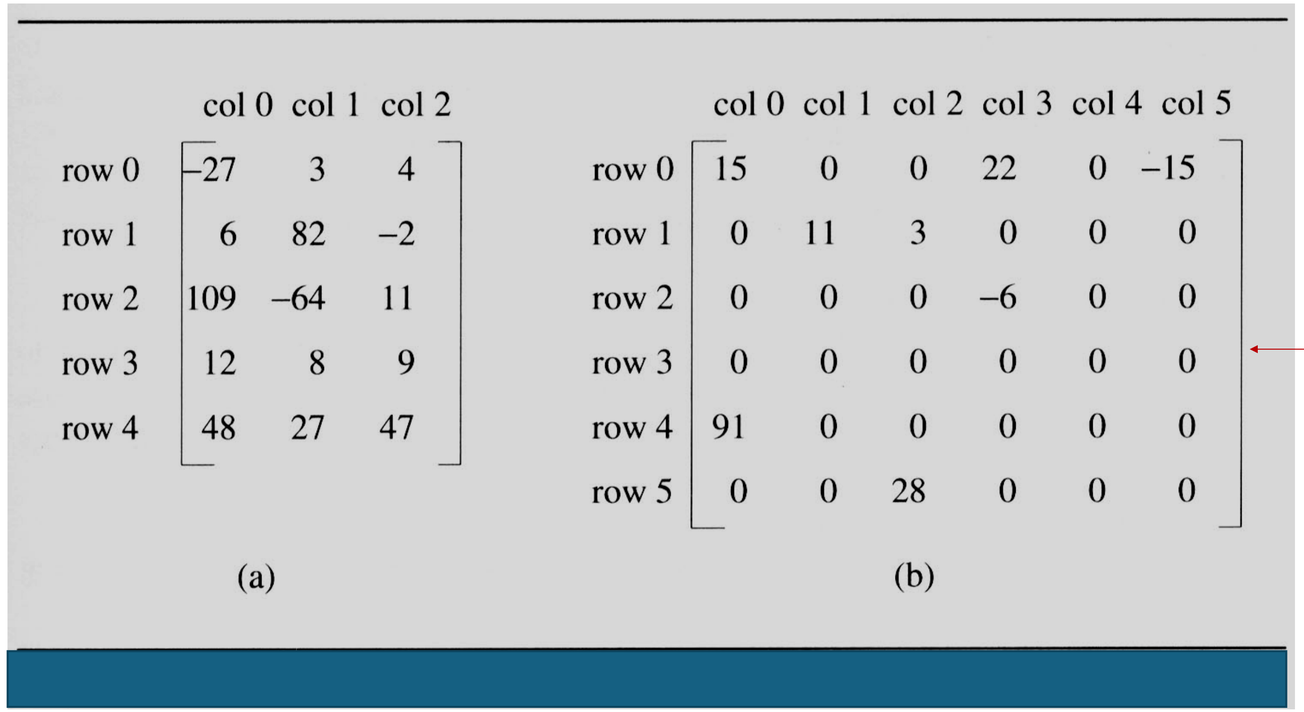
- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.
- **Why to use Sparse Matrix instead of simple matrix ?**
  - **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
  - **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

  Example:

  0 0 3 0 4
  0 0 5 7 0
  0 0 0 0 0
  0 2 6 0 0

Sparse Matrix

- In mathematics, a matrix contains *m* rows and *n* columns of elements, we write *m*x*n* to designate a matrix with *m* rows and *n* columns.
- A sparse matrix is **a matrix that is comprised of mostly zero values**



	col 0	col 1	col 2
row 0	−27	3	4
row 1	6	82	−2
row 2	109	−64	11
row 3	12	8	9
row 4	48	27	47

(a)

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	−15
row 1	0	11	3	0	0	0
row 2	0	0	0	−6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

(b)

sparse matrix data structure?

Sparse Matrix Representation

- Two common representations of Sparse Matrix :
- ✓ Array representation
- ✓ Linked list representation

- The standard representation of a matrix is a two-dimensional array defined as

A[max_rows][max_cols]

  - We can locate quickly any element by writing a[i][j].

- Sparse Matrix wastes space
  - Need for alternative form of representation
  - Representation of sparse matrix must store only the nonzero elements.
  - Each element is characterized by <row, col, val>

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row, column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Reading and representing Sparse matrix in array of objects format

Structure creation:

```
struct SM {
      int row; int col; int val;
   };

SM a[max_terms];
```

- a[0]. *row* contains number of rows.
- a[0]. *col* contains number of columns.
- a[0]. *value* contains total number of non zero entries.
- Positions 1 through 8 store the triples representing the non zero entries.
- Row index is in the field *row*; column index in *col*; value in field *value*.
- Triples are ordered by rows and within rows by columns.

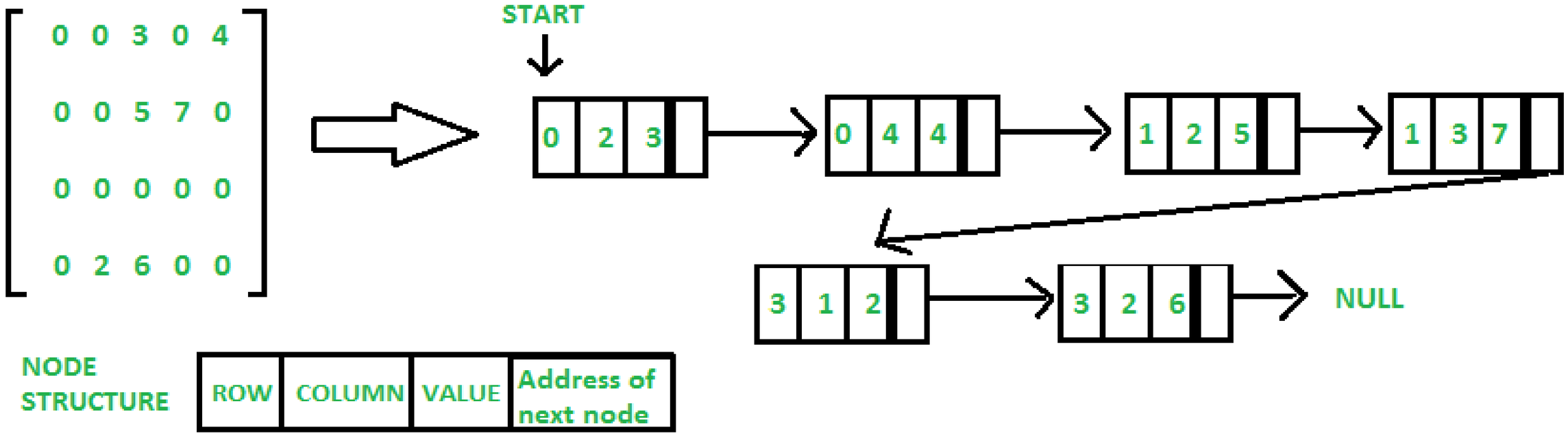
	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



Transpose of Sparse Matrix

Assign
A[i][j] to B[j][i]

place element <i, j, value> in element <j, i, value>

For all columns i

For all elements in column j

Scan the array "columns" times. The array has "elements" elements.

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
   int n,i,j, currentb;
   n = a[0].value;         /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

==> O(columns*elements)

EX: A[6][6] transpose to B[6][6]

i=1 j=8
a[i].col = 2 != i

Matrix A

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Row Col Value

0	6	6	8
1	0	0	15
2	0	4	91
3	1	1	11

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
   int n,i,j, currentb;
   n = a[0].value;        /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
        for (j = 1; j <= n; j++)
        /* find elements from the current column */
          if (a[j].col == i) {
          /* element is in current column, add it to b */
            b[currentb].row = a[j].col;
            b[currentb].col = a[j].row;
            b[currentb].value = a[j].value;
            currentb++;
          }
   }
}
```

Set Up row & column in B[6][6]

And So on...

Transpose of a Sparse matrix

- Discussion: compared with 2D array representation
  - O(columns * elements) vs O(columns * rows)
  - elements => columns * rows when non-sparse which is equivalent to O($columns^2$ * rows)
- Problem: Scan the array “column” times
  - In fact, we can transpose a matrix represented as a sequence of triplets in O (columns + elements) time.
- Solution:
  - First, determine the number of elements in each column of the original matrix.
  - Second, determine the starting positions of each row in the transpose matrix.

Fast Transpose of a Sparse matrix

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

transpose →

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Matrix A

Row Col Value

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

I = 7

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	3	4	6	8	8	9

Matrix A

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Row Col Value

0	6	6	8
1	0	0	15
2	0	4	91
3	1	1	11
4	2	1	3
5	2	5	28
6	3	0	22
7	3	2	-6
8	5	0	-15

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```