PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays - Initialization and Traversal Pointers

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays: Initialization and Traversal, Pointers:

- Array
  1. What is an Array?
  2. Properties of Arrays.
  3. Classification of Arrays
  4. Declaration and Initialization
  5. Representation of the Array
  6. Array Traversal

- Pointer
  1. What is a Pointer?
  2. Declaration and Initialization
  3. Arithmetic operations on Pointer
  4. Array Traversal using pointers
  5. Array and Pointer

What is an Array?

- A linear data structure, which is a Finite collection of similar data items stored in successive or consecutive memory locations
- **Only homogenous types of data** is allowed in any array. May contain all integer or all character elements, but not both together.
- char c_array [10]; short s_array [20];

Properties of Arrays

- Non-primary data type or secondary data type
- Memory allocation is contiguous in nature
- Elements need not be unique.
- Demands same /homogenous types of elements
- Random access of elements in array is possible
- Elements are accessed using index/subscript which starts from 0
- Memory is allocated at compile time.
- Size of the array is fixed at compile time and cannot be changed at runtime. Returns the number of bytes occupied by the array.
- Arrays are assignment incompatible.
- Accessing elements of the array outside the bound can have undefined behavior at runtime

Classification of Arrays

Category 1:

- Fixed Length Array
  - Size of the array is fixed at compile time
- Variable Length Array – Not discussed here

Category 2:

- One Dimensional (1-D) Array
  - Stores the data elements in a single row or column.
- Multi Dimensional Array
  - More than one row and column is used to store the data elements

Declaration and Initialization

Declaration: **Data_type Array_name[Size];**

Data_type: Specifies the type of the element that will be contained in the array
Array_name: Identifier to identify a variable as an array
Size: Indicates the max no. of elements that can be stored inside the array

Example: double x[15]; // Can contain 15 elements of type double, 0 to 14 are valid array indices or subscript

- Subscripts in array can be integer constant or integer variable or expression that yields integer
- C performs no bound checking. Care should be taken to ensure that the array indices are within the declared limits

Declaration and Initialization

- After an array is declared, it must be initialized.
- An array can be initialized at either **compile time** or at **runtime.**

Compile time Initialization

- data-type array-name[size] = { list of values };
- float area[5]={ 23.4, 6.8, 5.5 }; // Partial initialization
- int a[15] = {[2] = 29, [9] = 7, [14] = 48}; //C99's designated initializers
- int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48}; //C99's designated initializers
- int arr[] = {2, 3, 4}; // sizeof arr is decided
- int marks[4]={ 67, 87, 56, 77, 59 }; // undefined behavior
- Coding Examples

Runtime Initialization

- Using a loop and input function in C
- Coding examples

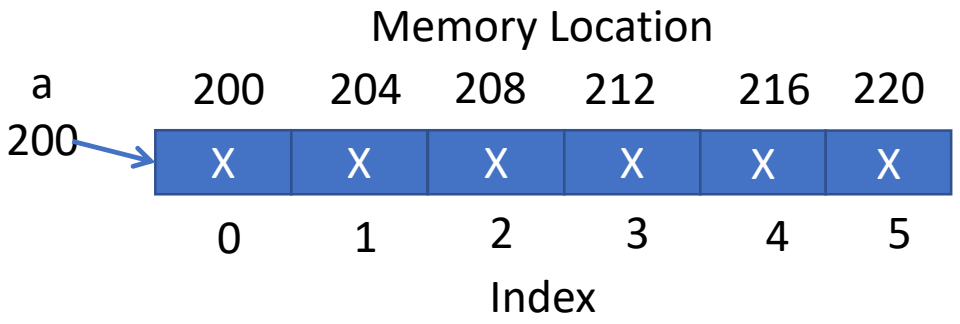
Representation of the Array

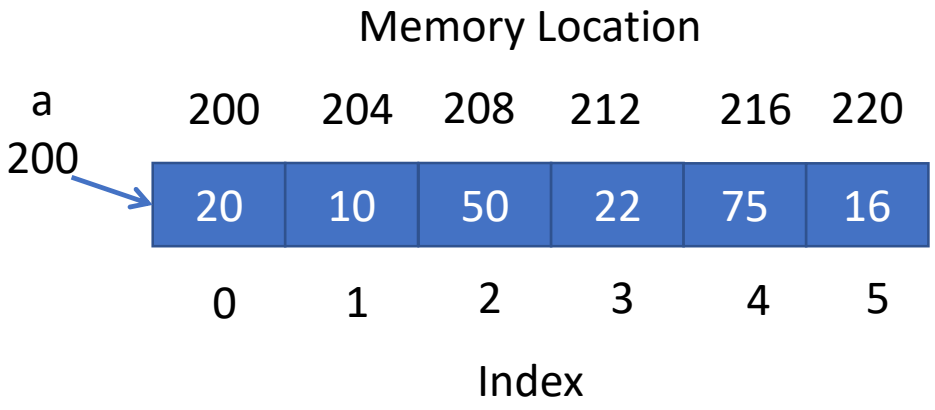
- int a[6];
- int a[6] = {20, 10, 50, 22, 75,16};

- Address of the first element is called the Base address of the array.

- Address of ith element of the array can be found using formula:

  Address of ith element = Base address + (size of each element * i)





Traversal

- Nothing but accessing each element of the array

- int a[10];
  - How do you access the 5th element of the array? // a[4]
  - How do you display each element of the array? // Using Loop
  - How much memory allocated for this?

    Number of bytes allocated = size specified*size of integer

- Anytime accessing elements outside the array bound is an undefined behavior

- Coding examples

Pointers

What is a Pointer?

- A variable which contains the address. This address is the location of another object in the memory

- Used to access and manipulate data stored in memory.

- Pointer of particular type can point to address of any value in that particular type.

- Size of pointer of any type is same/constant in that system

- Not all pointers actually contain an address

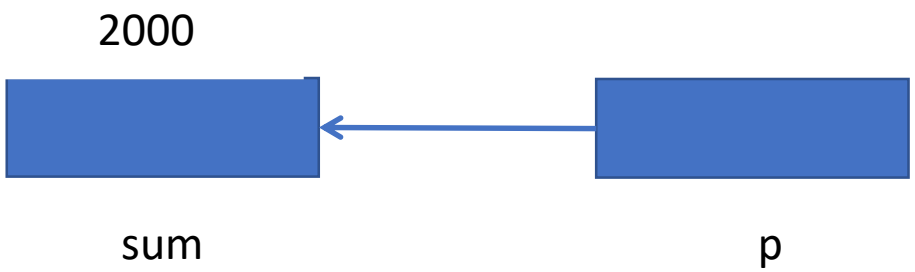
  Example: NULL pointer // Value of NULL pointer is 0.

- Pointer can have three kinds of contents in it
  - The address of an object, which can be dereferenced.
  - A NULL pointer
  - Undefined value // If p is a pointer to integer, then – int *p;

Declaration and Initialization

Declaration : Data-type *name;

- int *p;
  - Compiler assumes that any address that it holds points to an integer type.
- p= ∑
  - Memory address of sum variable is stored into p.



Example code:

```
int *p;      // p can point to anything where integer is
                stored. int* is the type. Not just int.
int a = 100;
p=&a;
printf("a is %d and *p is %d", a,*p);
```

Pointer Arithmetic Operations

1. Add an int to a pointer
2. Subtract an int from a pointer
3. Difference of two pointers when they point to the same array.

Note: Integer is not same as pointer.

- Coding examples

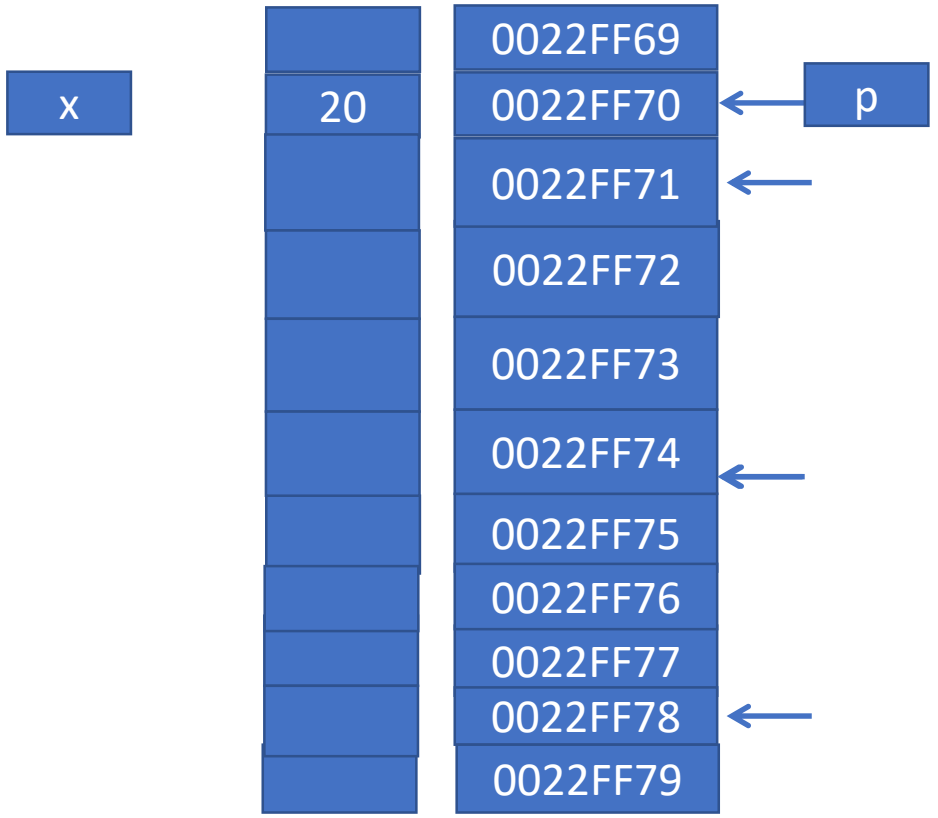
Pointer Arithmetic Operations continued..

Example Code:

```
int *p, x = 20;
p = &x;
printf("p     = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

p = 0022FF70
p+1 = 0022FF74
p+1 = 0022FF71
p+1 = 0022FF74
p+1 = 0022FF78



Array Traversal using Pointers

Consider int arr[] ={12,44,22,33,55}; int *p = arr; int i;

Coding examples to demo below points

- Array notation. Index operator can be applied on pointer.
- Pointer notation
- Using *p++
- Using *++p, Undefined behavior if you try to access outside bound
- Using (*p)++
- Using *p and then p++

Array and Pointer

- An array during compile time is an actual array but degenerates to a constant pointer during run time.

- Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

```
int *p1; float *f1 ; char *c1;
printf("%d %d %d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all
```

- An array is a constant pointer. It cannot point to anything in the world

Array and Pointer continued..

Example code:

```
int a[] = {22,11,44,5};
int *p = a;
a++;                    // Error : a is constant pointer
p++;                    // Fine
p[1] = 222;
a[1] = 222 ;            // Fine
```

- If variable i is used in loop for the traversal, a[i], *(a+i), p[i], *(p+i), i[a], i[p] are all same.

Array and Pointer continued..

Differences

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic operations on pointer variable is allowed. On array, not allowed.

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PES UNIVERSITY

PROBLEM SOLVING WITH C

Multi-Dimensional Arrays and Implementation of Matrix Operations

Prof. Sindhu R Pai
Department of Computer Science and Engineering

Agenda

1. Introduction
2. Two-Dimensional Array: Declaration
3. Two Dimensional Array: Initialization
4. Internal Representation of a 2D Array
5. Pointer and 2D Array
6. Three Dimensional (3D) Array
7. Practice Programs

Multi-Dimensional Arrays

PES UNIVERSITY

Introduction

- An array with more than one level or dimension.
- 2-Dimensional and 3-Dimensional and so on.

General form of declaring N-dimensional arrays:

Data_type Array_name[size1][size2][size3]..[sizeN];

PES UNIVERSITY

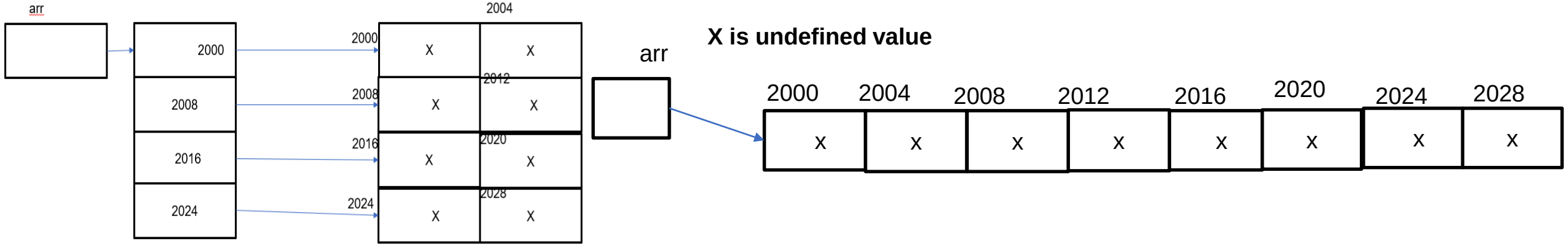
Two – Dimensional Array

- Treated as an array of arrays.
- Every element of the array must be of same type as arrays are homogeneous

Declaration

Syntax: data_type array_name[size_1][size_2]; // size_1 and size_2 compulsory

int arr[4][2];



PES UNIVERSITY

Two – Dimensional Array

Initialization

- **data_type array_name[size_1][size_2] = {elements separated by comma};**
- int arr[][] = {11,22,33,44,55,66}; // Error. Column size is compulsory
- int abc[3][2] ={{11,22},{33,44},{55,66}};//valid
- int arr[][2] = {11, 22, 33 ,44,55,66 } ;//valid. Allocates 6 contiguous memory locations and assign the values
- int arr[][3] = {{11,22,33},{44,55},{66,77}}; // partial initialization

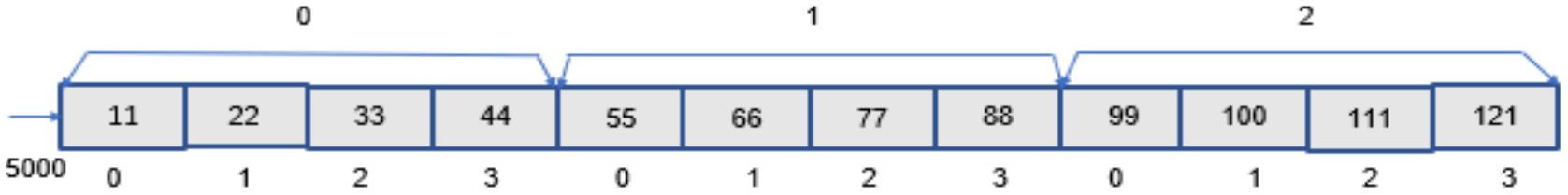
PES UNIVERSITY

Internal Representation of a 2D Array

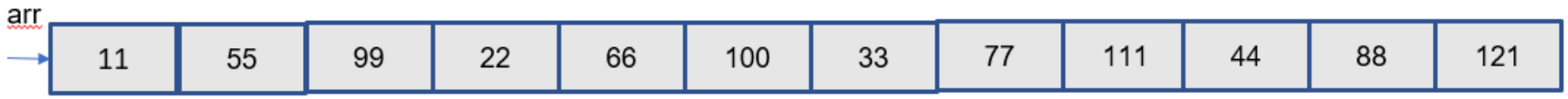
- 2D Array itself is an array, **elements are stored in contiguous memory locations**.

  Consider, int arr[3][4] = {{11, 22, 33, 44}, {55, 66, 77, 88}, {99, 100, 111, 121}};

- **Row major Ordering**: All elements of one row are followed by all elements of the next row and so on.



- **Column Major Ordering:** All elements of one column are followed by all elements of the next column and so on.



- **Generally, systems support Row Major Ordering.**

PES UNIVERSITY

Address of an Element in a 2D Array

- **Address of A[i][j] = Base_Address +((i * No. of columns in every row) + j)*size of every element;**
- Consider, int matrix[2][3]={1,2,3,4,5,6}; // base address is 100 and size of integer is 4 bytes

matrix[0][0] 100	1
matrix[0][1] 104	2
matrix[0][2] 108	3
matrix[1][0] 112	4
matrix[1][1] 116	5
matrix[1][2] 120	6

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

- Address of matrix[1][0]=100+((1*3)+0)*4=100+3*4=112

PES UNIVERSITY

Demo of C Code

- **To read and display a 2D Array**

PES UNIVERSITY

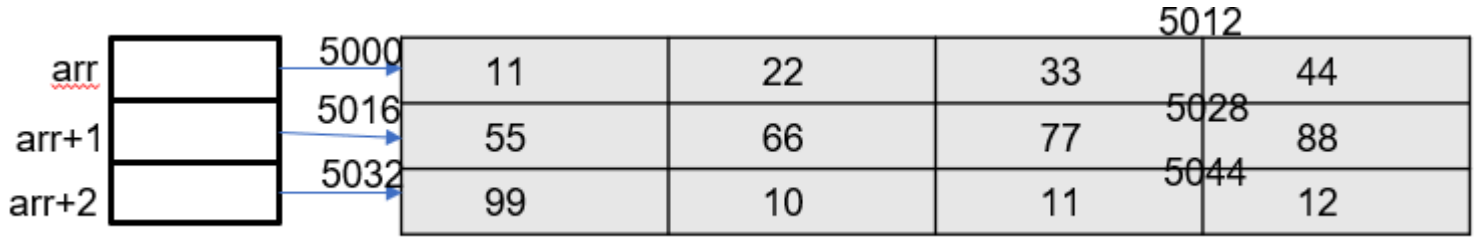
Pointer and 2D Array

- **Pointer expression for** a[i][j] is ***(*(a + i) + j)**
- Array name is a pointer to a row.
- (a + i) points to ith 1-D array.
- *(a + i) points to the first element of ith 1D array
- Coding examples

Pointer and 2D Array continued..

- Consider, int arr[3][4] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 100, 111, 121};

arr – points to 0th elements of arr- Points to 0th 1-D array-5000
arr+1-Points to 1st element of arr-Points to 1st 1-D array-5016
arr+2-Points to 2nd element of arr-Points to 2nd 1-D array
arr+ i Points to ith element of arr ->Points to ith 1-D array



PES UNIVERSITY

Pointer and 2D Array continued..

- int *p = arr; // assigning the 2D array to a pointer results in warning
- Using p[5] results in 66. But p[1][1] results in error. p doesn't know the size of the column.
- Solution is to create a **pointer to an array of integers.**

int (*p)[4] = arr; //subscript([ ]) have higher precedence than indirection(*)

- Think about the **size of p and size of *p!!**

PES UNIVERSITY

Three Dimensional (3D) Array

- Accessing each element by using three subscripts
- First dimension represents table ,2nd dimension represents number of rows and 3rd dimension represents the number of columns
- int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} }; //2 table 3 rows 2 coloumns.

PES UNIVERSITY

Practice Programs for Students

1. Given two matrices, write a function to find whether these two are identical.

2.Program to find the transpose of a given matrix.

3.Program to find the inverse of a given matrix.

4.Write a function to check whether the given matrix is identity matrix or not.

5. Write a program in C to find sum of right diagonals of a matrix.

6. Write a program in C to find sum of rows and columns of a matrix

THANK YOU

Prof. Sindhu R Pai
Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays - Initialization and Traversal Pointers

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays: Initialization and Traversal, Pointers:

- Array
  1. What is an Array?
  2. Properties of Arrays.
  3. Classification of Arrays
  4. Declaration and Initialization
  5. Representation of the Array
  6. Array Traversal

- Pointer
  1. What is a Pointer?
  2. Declaration and Initialization
  3. Arithmetic operations on Pointer
  4. Array Traversal using pointers
  5. Array and Pointer

What is an Array?

- A linear data structure, which is a Finite collection of similar data items stored in successive or consecutive memory locations
- **Only homogenous types of data** is allowed in any array. May contain all integer or all character elements, but not both together.
- char c_array [10]; short s_array [20];

Properties of Arrays

- Non-primary data type or secondary data type
- Memory allocation is contiguous in nature
- Elements need not be unique.
- Demands same /homogenous types of elements
- Random access of elements in array is possible
- Elements are accessed using index/subscript which starts from 0
- Memory is allocated at compile time.
- Size of the array is fixed at compile time and cannot be changed at runtime. Returns the number of bytes occupied by the array.
- Arrays are assignment incompatible.
- Accessing elements of the array outside the bound can have undefined behavior at runtime

Classification of Arrays

Category 1:

- Fixed Length Array
  - Size of the array is fixed at compile time
- Variable Length Array – Not discussed here

Category 2:

- One Dimensional (1-D) Array
  - Stores the data elements in a single row or column.
- Multi Dimensional Array
  - More than one row and column is used to store the data elements

Declaration and Initialization

Declaration: **Data_type Array_name[Size];**

Data_type: Specifies the type of the element that will be contained in the array
Array_name: Identifier to identify a variable as an array
Size: Indicates the max no. of elements that can be stored inside the array

Example: double x[15]; // Can contain 15 elements of type double, 0 to 14 are valid array indices or subscript

- Subscripts in array can be integer constant or integer variable or expression that yields integer
- C performs no bound checking. Care should be taken to ensure that the array indices are within the declared limits

Declaration and Initialization

- After an array is declared, it must be initialized.
- An array can be initialized at either **compile time** or at **runtime.**

Compile time Initialization

- data-type array-name[size] = { list of values };
- float area[5]={ 23.4, 6.8, 5.5 }; // Partial initialization
- int a[15] = {[2] = 29, [9] = 7, [14] = 48}; //C99's designated initializers
- int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48}; //C99's designated initializers
- int arr[] = {2, 3, 4}; // sizeof arr is decided
- int marks[4]={ 67, 87, 56, 77, 59 }; // undefined behavior
- Coding Examples

Runtime Initialization

- Using a loop and input function in C
- Coding examples

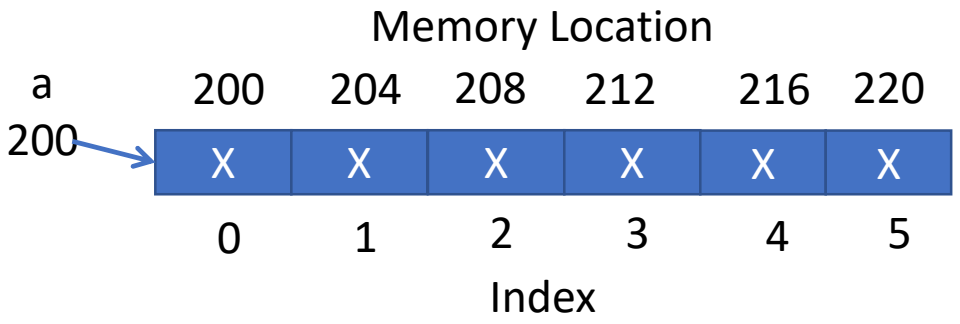
Representation of the Array

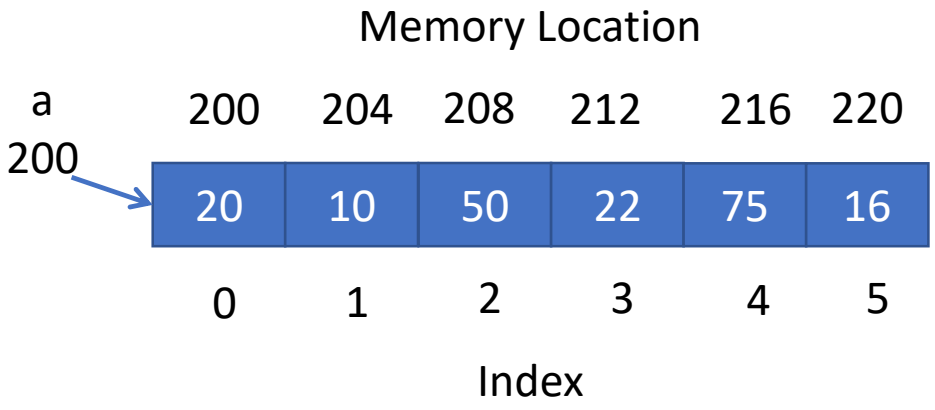
- int a[6];
- int a[6] = {20, 10, 50, 22, 75,16};

- Address of the first element is called the Base address of the array.

- Address of ith element of the array can be found using formula:

  Address of ith element = Base address + (size of each element * i)





Traversal

- Nothing but accessing each element of the array

- int a[10];
  - How do you access the 5th element of the array? // a[4]
  - How do you display each element of the array? // Using Loop
  - How much memory allocated for this?

    Number of bytes allocated = size specified*size of integer

- Anytime accessing elements outside the array bound is an undefined behavior

- Coding examples

Pointers

What is a Pointer?

- A variable which contains the address. This address is the location of another object in the memory

- Used to access and manipulate data stored in memory.

- Pointer of particular type can point to address of any value in that particular type.

- Size of pointer of any type is same/constant in that system

- Not all pointers actually contain an address

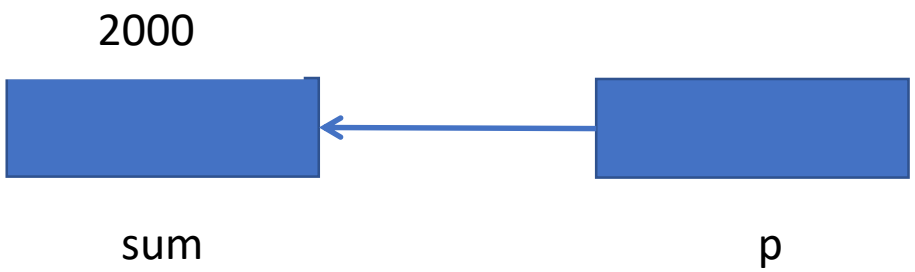
  Example: NULL pointer // Value of NULL pointer is 0.

- Pointer can have three kinds of contents in it
  - The address of an object, which can be dereferenced.
  - A NULL pointer
  - Undefined value // If p is a pointer to integer, then – int *p;

Declaration and Initialization

Declaration : Data-type *name;

- int *p;
  - Compiler assumes that any address that it holds points to an integer type.
- p= ∑
  - Memory address of sum variable is stored into p.



Example code:

```
int *p;      // p can point to anything where integer is
                stored. int* is the type. Not just int.
int a = 100;
p=&a;
printf("a is %d and *p is %d", a,*p);
```

Pointer Arithmetic Operations

1. Add an int to a pointer
2. Subtract an int from a pointer
3. Difference of two pointers when they point to the same array.

Note: Integer is not same as pointer.

- Coding examples

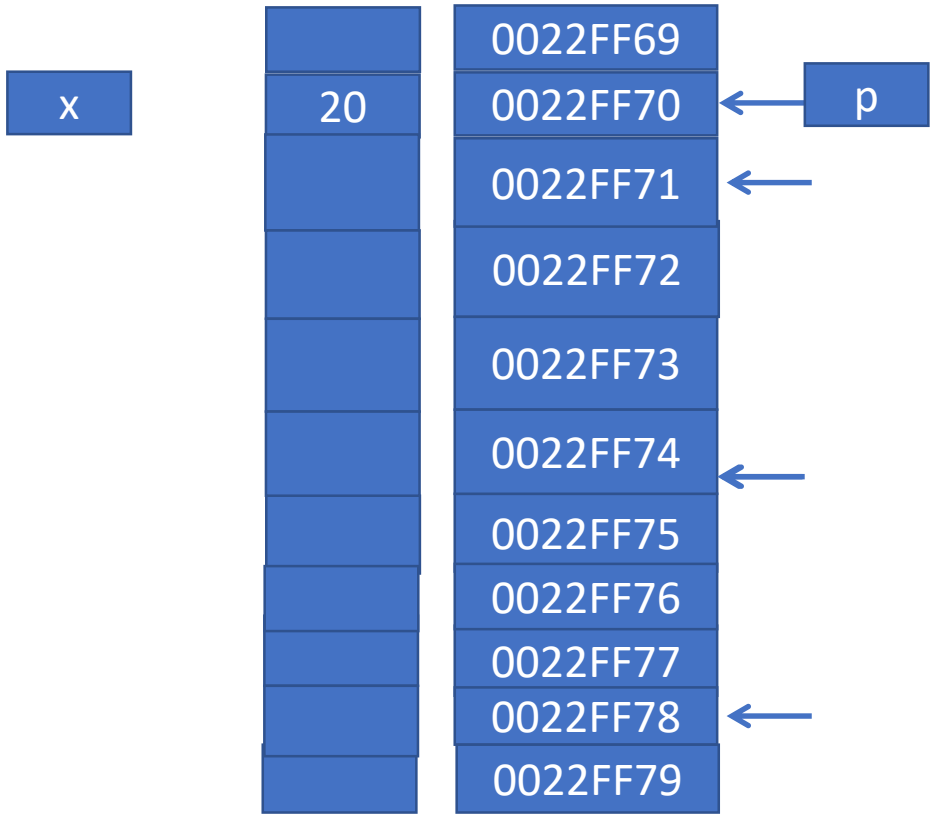
Pointer Arithmetic Operations continued..

Example Code:

```
int *p, x = 20;
p = &x;
printf("p     = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

p = 0022FF70
p+1 = 0022FF74
p+1 = 0022FF71
p+1 = 0022FF74
p+1 = 0022FF78



Array Traversal using Pointers

Consider int arr[] ={12,44,22,33,55}; int *p = arr; int i;

Coding examples to demo below points

- Array notation. Index operator can be applied on pointer.
- Pointer notation
- Using *p++
- Using *++p, Undefined behavior if you try to access outside bound
- Using (*p)++
- Using *p and then p++

Array and Pointer

- An array during compile time is an actual array but degenerates to a constant pointer during run time.

- Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

```
int *p1; float *f1 ; char *c1;
printf("%d %d %d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all
```

- An array is a constant pointer. It cannot point to anything in the world

Array and Pointer continued..

Example code:

```
int a[] = {22,11,44,5};
int *p = a;
a++;                    // Error : a is constant pointer
p++;                    // Fine
p[1] = 222;
a[1] = 222 ;            // Fine
```

- If variable i is used in loop for the traversal, a[i], *(a+i), p[i], *(p+i), i[a], i[p] are all same.

Array and Pointer continued..

Differences

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic operations on pointer variable is allowed. On array, not allowed.

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays and Functions

Prof. Sindhu R pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Arrays and Functions

1. Passing an Array to a function
2. Array as a Formal parameter and actual parameter
3. Pointer as a Formal parameter and Array as an Actual parameter
4. Passing individual array element to a function

Passing an Array to a function

- When array is passed as an argument to a function, arguments are copied to parameters of the function and parameters are always pointers.
- Array degenerates to a pointer at runtime.
- All the operations that are valid for pointer will be applicable for array too in the body of the function.
- Function call happens always at run time.

Array as a formal parameter and Actual parameter

- **Array being formal parameter** - Indicated using empty brackets in the parameter list.

  void myfun(int a[],int size);

- **Array being actual parameter** – Indicated using the name of the array

  Array a is declared as int a[5];

  Then myfun is called as myfun(a,n);

- Coding example to read and display the array elements

Pointer as a formal parameter and Array as an Actual parameter

- **Pointer being formal parameter** - Indicated using empty brackets in the parameter list.

  void myfun(int *a,int size);

- **Array being actual parameter** – Indicated using the name of the array

  Array a is declared as int a[5];

  Then myfun is called as myfun(a,n);

- Coding example to read and display the array elements

Passing individual Array elements to a function

- Indexed variables can be arguments to functions
- Program contains these declarations: int a[10]; int i; void myfunc(int n);
- Variables a[0] through a[9] are of type int, making below calls is legal

```
myfunc(a[0]);
myfunc(a[3]);
myfunc(a[i]);                    // i is between 0 and 9
```

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Functions in C

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Functions

1. Introduction to functions
2. Types of Functions
3. Function Definition, Call and Declaration

Introduction

- A sub-program to carry out a specific task
- Functions break large computing tasks into smaller ones.
- Enable people to build on what others have done instead of starting from scratch
- **Benefits:**
  - Reduced Coding Time – Code Reusability
  - Divide and Conquer - Manageable Program development
  - Reduced Debugging time
  - Treated as a black box - The inner details of operation are invisible to rest of the program

Types of Functions

- **Standard Library Functions**

  Must Include appropriate header files to use these functions.
  Already declared and defined in C libraries
  printf, scanf, etc..

- **User Defined functions**

  Defined by the developer at the time of writing program
  Developer can make changes in the implementation

Function Definition, Call and Declaration

Function Definition

- Provides actual body of the function

- Function definition format

```
return-type  function-name( parameter-list )
{
   declarations and statements
}
```

- Variables can be declared inside blocks

- Coding examples

Function Definition, Call and Declaration

Function Call

- Function can be called by using function name followed by list of arguments (if any) enclosed in parentheses

- Function call format
  function-name(list of arguments);

- The arguments must match the parameters in the function definition in it's type, order and number.

- Multiple arguments must be separated by comma. Arguments can be any expression in C

- Coding examples

Function Definition, Call and Declaration

Function Declaration/ Prototype

- All functions must be declared before they are invoked or called.
- Function can be declared by using function name followed by list of parameters (if any) enclosed in parentheses
- Function declaration format

  return_type function_name (parameters list);

- Use of identifiers in the declaration is optional.
- The parameter names do not need to be the same in declaration and the function definition.
- The types must match the type of parameters in the function definition in number and order.
- Coding examples

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Recursion

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Recursion

1. Introduction
2. Why Recursion?
3. Implementing Recursion – The Stack
4. Arguments and Return Values
5. Practice Programs

Introduction

Recursive Function

- A function that calls itself Directly or indirectly

- Each recursive call is made with a new, independent set of arguments
    - Previous calls are suspended

- Allows building simple solutions for complex problems

Why Recursion?

- Used to solve various problems by dividing it into smaller problems

- Some problems are *too hard* to solve without recursion
  - Most notably, the compiler!
  - Most problems involving linked lists and trees

Points to note while using Recursion

- The problem is broken down into smaller tasks
- Programmers need to be careful to define an exit condition from the function, otherwise results in an infinite recursion
- The exit condition is defined by the ***base case*** and the solution to the base case is provided
- The solution of the bigger problem is expressed in terms of smaller problems called as ***recursive relationship***

How is a particular problem solved using recursion?

- **Problem to be solved: To compute factorial of n.**
  - knowledge of factorial of (n-1) is required, which would form the recursive relationship
  - The base case for factorial would be n = 0
  - return 1 when n == 0 or n == 1 // base case
  - return n*(n-1)! when n!=0 // recursive relationship
  - Demo of C code

Implementing Recursion - The Stack

- **Definition –** ***The Stack***
  - A **last-in, first-out** data structure provided by the operating system for running each program
  - For temporary storage of automatic variables, arguments, function results, and other information
  - The storage for each function call.
  - Every single time a function is called, an area of the stack is reserved for that particular call.
- Known as ***activation record,*** similar to that in python

Implementing Recursion - The Stack continued..

- Parameters, results, and automatic variables allocated on the stack.
- Allocated when function or compound statement is entered
- Released when function or compound statement is exited
- Values are not retained from one call to next (or among recursions)

Arguments and Return values

1. Space for storing result is allocated by caller
   - On The Stack
   - Assigned by **return** statement of function
   - For use by caller
2. Arguments are values calculated by caller of function
   - Placed on The Stack by caller in locations set aside for the corresponding parameters
   - Function may assign new value to parameter
   - caller never looks at parameter/argument values again!
3. Arguments are removed when callee returns
   - Leaving only the result value for the caller

Practice Programs based on Recursion

1. Separate Recursive functions to reverse a given number and reverse a given string
2. Recursive function to print from 1 to n in reverse order
3. Find the addition, subtraction and multiplication of two numbers using recursion. Write separate recursive functions to perform these.
4. Find all combinations of words formed from Mobile Keypad.
5. Find the given string is palindrome or not using Recursion.
6. Find all permutations of a given string using Recursion

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Storage Classes in C

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Storage classes in C

- Introduction
- Automatic Variables
- External Variables
- Static Variables
- Register Variables
- Global Variables

Storage classes in C

Introduction

- To describe the features of a variable/function . Features include **scope(visibility) and life-time** to trace the existence of a particular variable/function during the runtime

- **List** of storage classes
  - **Automatic variables (auto)**
  - **External variables (extern)**
  - **Static variables(static)**
  - **Register variables (register)**
  - **Global Variables**

Automatic Variables

- A variable declared inside a function without any storage class specification is by default an automatic variable
- Created when a function is called and are destroyed automatically when the function execution is completed
- Also called as called local variables because they are local to a function. By default, assigned to undefined values
- Can be accessed outside their scope. But how ?
  - **By using Pointers**
- Coding Examples

External Variables

- To inform the compiler that the variable is declared somewhere else and make it available during linking
- Does not allocate storage for variables
- The default initial value of external integral type is 0 otherwise null.
- All functions are of type extern
- Coding Examples

Storage classes in C

Static Variables

- Tells the compiler to persist the variable until the end of program.
- Initialized only once and remains into existence till the end of program
- Can either be **local or global depending upon the place of declaration**
  - Scope of local static variable remains inside the function in which it is defined but the life time of is throughout that program file
  - Global static variables remain restricted to scope of file in each they are declared and life time is also restricted to that file only
- All static variables are assigned 0 (zero) as default value
- Coding Examples

Register Variables

- Registers are faster than memory to access. So, the variables which are most frequently used in a program can be put in registers using **register** keyword
- The keyword register hints to compiler that a given variable can be put in a register. It's compiler's choice to put it in a register or not.
- Compilers themselves do optimizations and put the variables in register. If a free register is not available, these are then stored in the memory only
- If & operator is used with a register variable, then compiler may give an error or warning
- Coding Examples

Global variables- Additional Storage class

- The variables declared outside all function are called global variables. They are not limited to any function.
- Any function can access and modify global variables
- Automatically initialized to 0 at the time of declaration
- Coding Examples

THANK YOU

Prof. Sindhu R Pai
Department of Computer Science and Engineering

sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Enums(Enumerations) in C

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Enumerations

- Introduction
- Enum creation
- Points wrt Enums
- Demo of Enums in C

Enumerations

Introduction

- A way of creating user defined data type to assign names to integral constants. Easy to remember names rather than numbers

- Provides a symbolic name to represent one state out of a list of states

- The names are symbols for integer constants, which won't be stored anywhere in program's memory

- Used to **replace #define chains**

Enumerations

Enum creation

- **Syntax:**

```
enum identifier { enumerator-list };        // semicolon compulsory
                                            // identifier optional
```

- Example:

```
enum Error_list { SUCCESS, ERROR, RUN_TIME_ERROR, BIG_ERROR };
```

- Coding Examples

Enumerations

Points wrt Enums

- Enum names are automatically assigned values if no value specified
- We can assign values to some of the symbol names in any order. All unassigned names get value as value of previous name plus one.
- Only integer constants are allowed. Arithmetic operations allowed-> + , -, *, / and %
- Enumerated Types are Not Strings. Two enum symbols/names can have same value
- All enum constants must be unique in their scope. It is not possible to change the constants
- Storing the symbol of one enum type in another enum variable is allowed
- One of the short comings of Enumerated Types is that they don't print nicely

Domo of Enum in C

- Demo of Enum points discussed in the previous slide

THANK YOU

Prof. Sindhu R Pai
Department of Computer Science and Engineering
sindhurpai@pes.edu

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Interface and Implementation

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Interface and Implementation

1. Introduction
2. Header file creation
3. Source file creation
4. Demo of C Code
5. Compilation steps
6. Intro to make utility
7. Creation of make file
8. Usage of make command

Interface and Implementation

Introduction

- **Interface is *declaration*** and **Implementation is *Definition***

- Good C code organizes
  - **Interfaces in header file**
  - **Implementations in source files**

- **Benefits**
  - Modularity
  - Recompilation time is reduced
  - Readability
  - Debugging is easier

Header file creation

- Use '**.h**' extension

- Typically contains
  - Function declaration (except statics)
  - Variable declaration (typically global)
  - User defined type declaration (read struct, union etc.)
  - Macro definition

Source file creation

- Use '**.c**' extension

- Typically contains
  - Function/variable definition
  - Static function declaration and definition (you don't want to expose these to your clients)

Demo of C Code

Problem to be solved: **Find whether the given number is a palindrome or not**

Compilation steps

- Source file compilation: **gcc –c 1_palin.c**
- Client file compilation: **gcc -c 1_palin_main.c**
- Linking object files: **gcc 1_palin.o 1_palin_main.o**
- Execution:
  - **a.exe // windows**
  - **./a.out // ubuntu**

Introduction to make utility

- What if you have many implementation files and modifications are done to few of these?
  - Developer must remember the files for which modifications done and recompile only them.
  - OR it is waste of time to recompile all the files and link objects of all again.
- Make is a Unix utility designed to start execution of a makefile.
- A makefile is a special file containing shell commands
- Use '**.mk**' extension preferably.
- A makefile that works well in one shell may not execute properly in another shell.

Creation of make file

- Contains a list of **rules to compile and link a series of files**. Rules are specified in two lines.
  - **Dependency line** – Made up of two parts. Target file(s) : source file(s)
  - **Action line** – Must be intended with a tab.
- Make command reads a make file and creates a **Dependency Tree**.
- **Target files are rebuilt using Action line if they are missing or older than the source files**.
- Create a sample make file to execute the palindrome problem.

Usage of make command

- Command to execute on **Ubuntu:**
  - **make -f filename.mk** // -f to read file as a make file
- Command to execute on **Windows using mingw**
  - **mingw32-make -f filename.mk** // -f to read file as a make file

Note: In windows, utility must be downloaded. If installed gcc using mingw, go to mingw/bin in cmd prompt and type **mingw-get install mingw32-make** and press enter

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

sindhurpai@pes.edu