PROBLEM SOLVING WITH C

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Strings, String manipulation & Errors

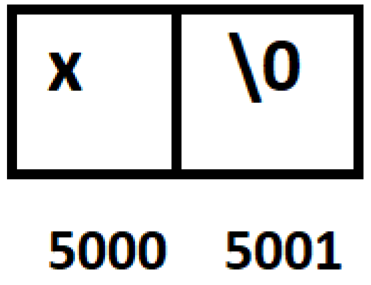
Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Strings, String manipulation & Errors

Introduction

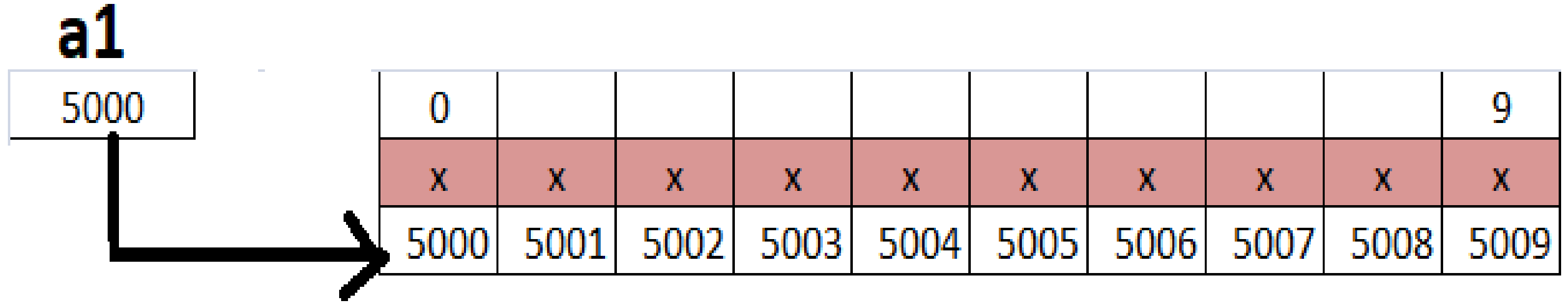
- An array of characters and terminated with a special character '\0' or NULL. ASCII value of NULL character is 0.
- String constants are always enclosed in double quotes. It occupies one byte more to store the null character.
- Example: **"X" is a String constant**



Declaration

Syntax: **char variable_name[size];** //Size is compulsory

```
char a1[10];            // valid declaration
char a1[];              // invalid declaration
```

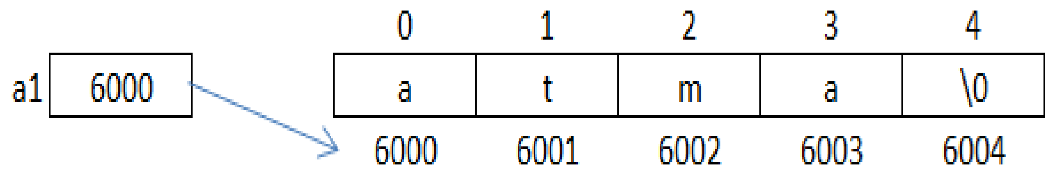


Initialization

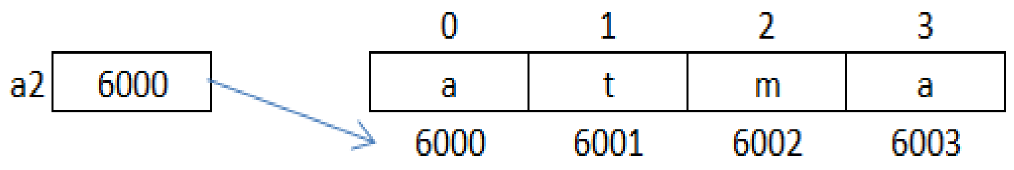
Syntax: **char variable_name[size] = {Elements separated by comma};**

Version 1:

- char a1[] = {'a', 't', 'm', 'a', '\0' };
- Shorthand notation: char a1[] = ”atma”;

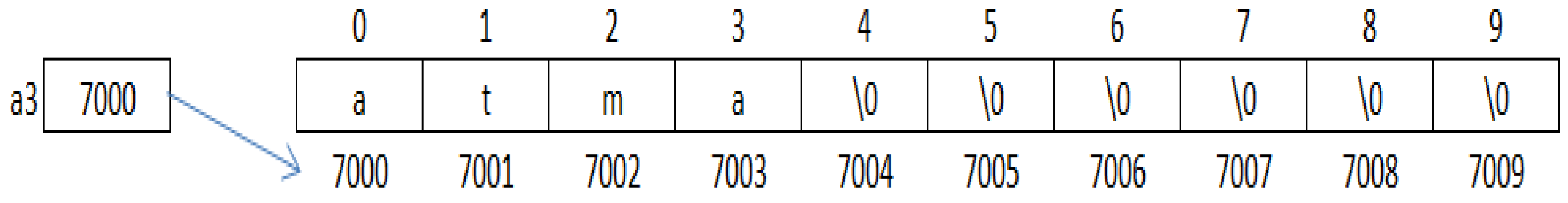


Version 2: char a2[] = {'a', 't', 'm', 'a' };



Version 3: Partial initialization

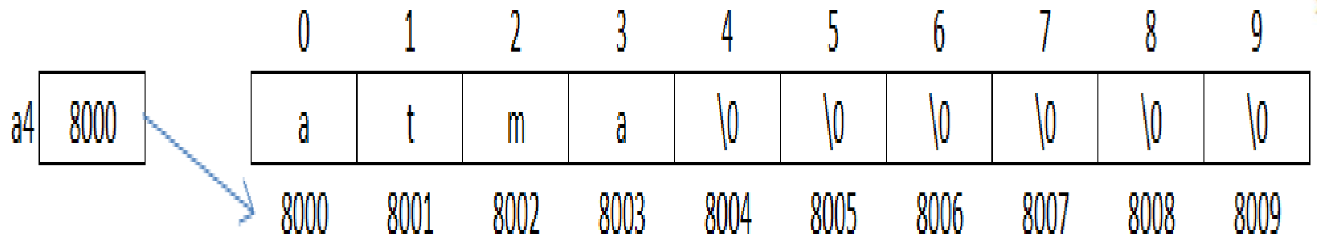
- char a3[10] = {'a','t','m','a'};



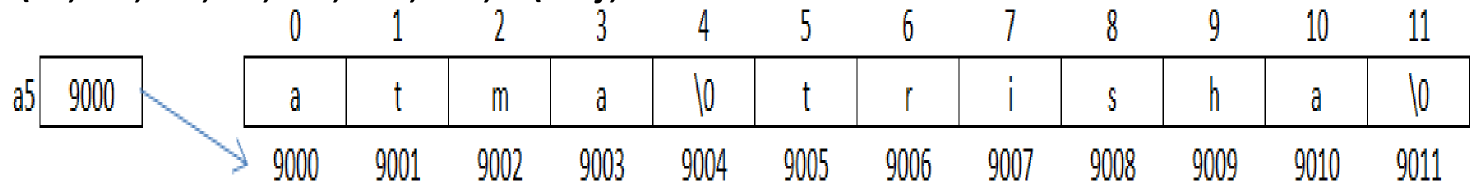
Initialization continued..

Version 4: Partial initialization

- char a4[10] = {'a','t','m','a', '\0'};
- char a4[10] = ”atma”;



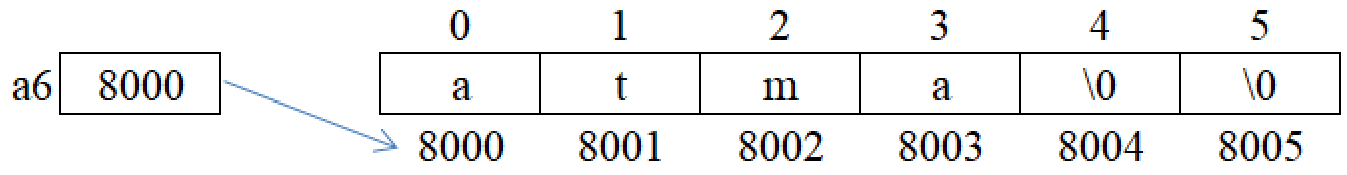
Version 5: char a5[ ] = {'a', 't', 'm', 'a', '\0', 't', 'r', 'i', 's', 'h', 'a', '\0' };

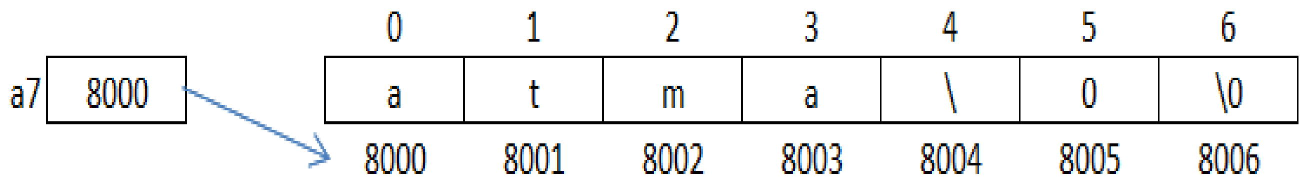


Version 6: char a6[ ] = "atma\0" ;

Version 7: char a7[ ] = "atma\\0" ;

Version 8: char a8[ ] = "at\0ma" ;





Demo of C Code

- To read and display a string in C
- Points to note:
  - If the string is hard coded, it is programmer's responsibility to end the string with '\0' character.
  - scanf terminates when white space is given in the user input.
  - scanf with %s will introduce '\0' character at the end of the string. printf with %s requires the address and will display the characters until '\0' character is encountered
  - If you want to store the entire input from the user until user presses new line in one character array variable, use [^\n] with %s in scanf

String v/s Pointer

- char x[] = "pes"; // x is an array of 4 characters with ‘p’, ‘e’, ‘s’, ‘\0

  Stored in the Stack segment of memory
- Can change the elements of x. x[0] = ‘P’;
- Can not increment x as it is an array name. x is a constant pointer.

- char *y = “pes”;

  y is stored at stack. “pes” is stored at code segment of memory. It is read only.
- y[0] = ‘P’ ; // undefined behaviour
- Can increment y . y is a pointer variable.

Built-in String manipulation Functions

- Expect '\0' and available in string.h
- In character array, if '\0' is not available at the end, result is undefined when passed to built-in string functions

- **strlen(a)** – Expects string as an argument and returns the length of the string, excluding the NULL character
- **strcpy(a,b)** – Expects two strings and copies the value of b to a.
- **strcat(a,b)** – Expects two strings and concatenated result is copied back to a.
- **strchr(a,ch)** – Expects string and a character to be found in string. Returns the address of the matched character in a given string if found. Else, NULL is returned.
- **strcmp(a,b)** – Compares whether content of array a is same as array b. If a==b, returns 0. Returns 1, if array a has lexicographically higher value than b. Else, -1.

User defined String functions

- Start with iterative solution
- Usage of pointer arithmetic operations
- Usage of recursion to get the solution for few of the below functions

1. my_strlen()
2. my_strcpy()
3. my_strcmp()
4. my_strchr()
5. my_strcat()

Error Demonstration

1. char * name[10] ; // declaring an array of 10 pointer pointing each one to a char

   char name[10];

2. name = “choco”; // Cannot assign a string by using the operator =

   char name[10] = “choco“ OR use strcpy()

3. printf("%s\n", *name);

   Provide only the address of the first element of the string

4. Coding examples to demo the error caused while applying built-in string functions on strings without specifying ‘\0’ at the end of the string

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

Dynamic Memory Management

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Dynamic Memory Management

1. Problem with the Arrays

2. Memory Allocation

3. Dynamic allocation

4. Use of malloc(), calloc(), realloc(), free()

Problem with the Arrays

- Few situations while coding:
  - Amount of data cannot be predicted beforehand
  - Number of data items keeps changing during program execution

- In such cases, use of fixed size array might create problems:
  - Wastage of memory space (under utilization)
  - Insufficient memory space (over utilization)

- **Example:** A[1000] can be used but what if the user wants to run the code for only 50 elements //memory wasted

Solution: Can be avoided by using the concept of Dynamic memory management

Memory Allocation

1. Static allocation

- decided by the compiler
- allocation at load time [before the execution or run time]
- example: variable declaration (int a, float b, a[20];)

2. Automatic allocation

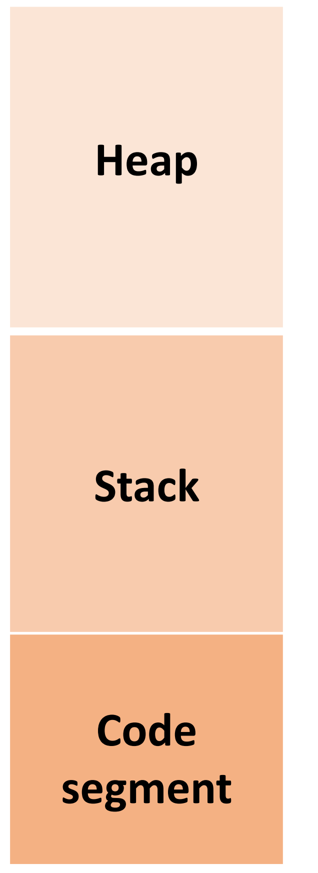
- decided by the compiler
- allocation at run time
- allocation on entry to the block and deallocation on exit
- example: function call (stack space is used and released as soon as callee function returns back to the calling function)

3. Dynamic allocation

- code generated by the compiler
- allocation and deallocation on call to memory allocation and deallocation functions

Dynamic Allocation

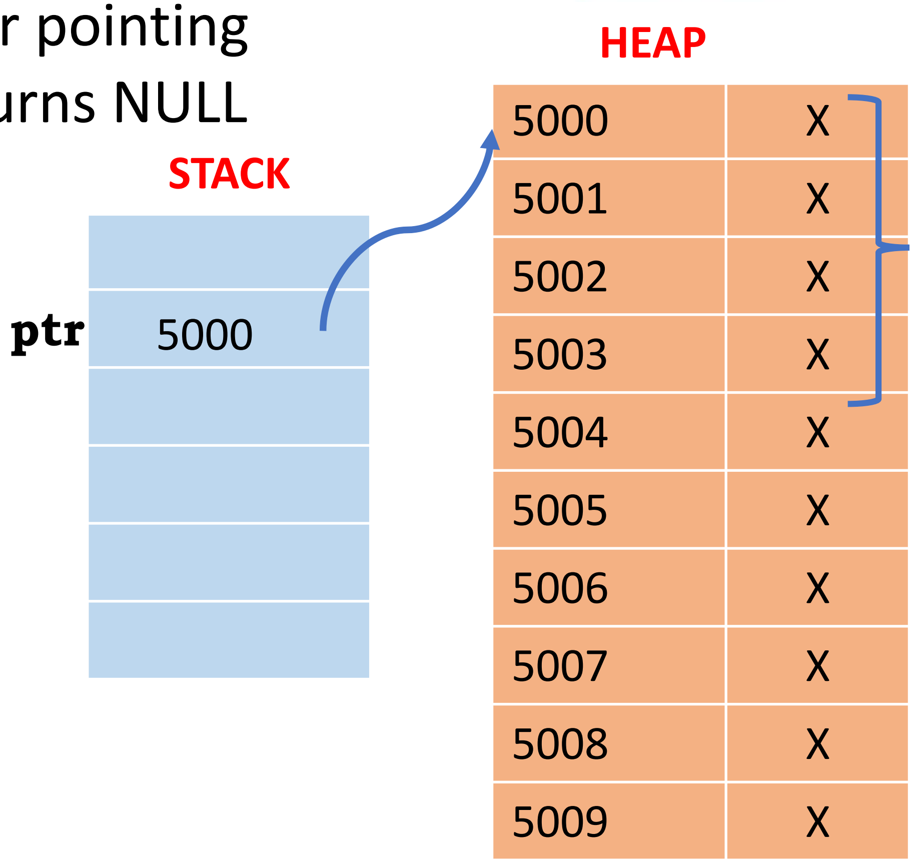
- Process of allocating memory at runtime/execution
- Uses the **Heap region of Memory segment**
- No operator in C to support dynamic memory management
- Library functions are used to dynamically allocate/release memory
  - malloc()
  - calloc()
  - realloc()
  - free()
- Available in stdlib.h



Memory space

malloc() - memory allocation

- Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space on success. Else returns NULL
- The return pointer can be type-casted to any pointer type
- Memory is not initialized
- **Syntax**:
  void *malloc(size_t N); // Allocates N bytes of memory
- **Example:**
  int* ptr = (int*) malloc(sizeof (int)); // Allocate memory for an int
- Coding example



calloc() - contiguous allocation

- Allocates space for elements, initialize them to zero and then returns a void pointer to the memory. Else returns NULL
- The return pointer can be type-casted to any pointer type
- **Syntax**:

  void *calloc(size_t nmemb, size_t size);

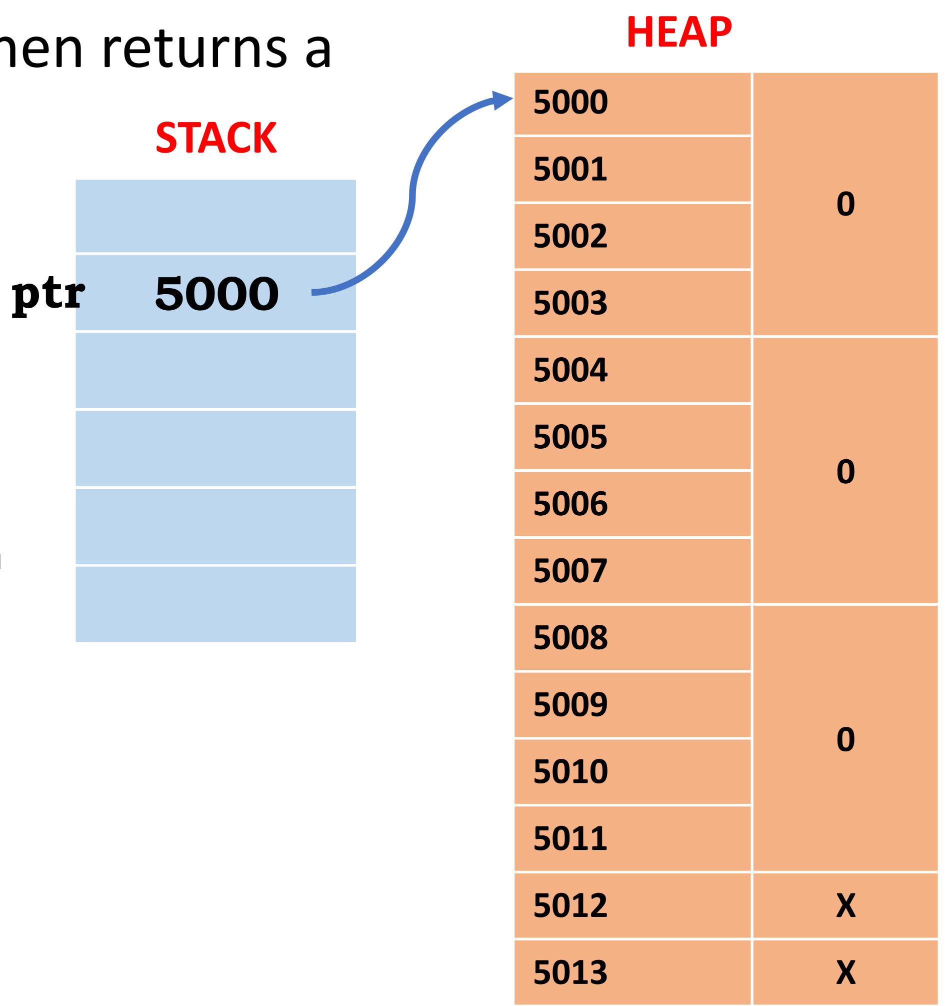
  //allocates memory for an array of nmemb elements of size bytes each

- **Example**:

  int* ptr = (int*) calloc (3,sizeof (int));

  //Allocating memory for an array of 3 elements of integer type

- Coding example



realloc() - reallocation of memory

- Modifies the size of previously allocated memory using malloc or calloc functions
- Returns a pointer to the newly allocated memory which has the new specified size. Returns NULL for an unsuccessful operation
- If realloc() fails, the original block is left untouched
- **Syntax:** void *realloc(void *ptr, size_t size);
- If ptr is NULL, then the call is equivalent to malloc(size), for all values of size
- If size is equal to zero, and ptr is not NULL, then the call is equivalent to free(ptr)
- This function can be used only for dynamically allocated memory, else behavior is undefined

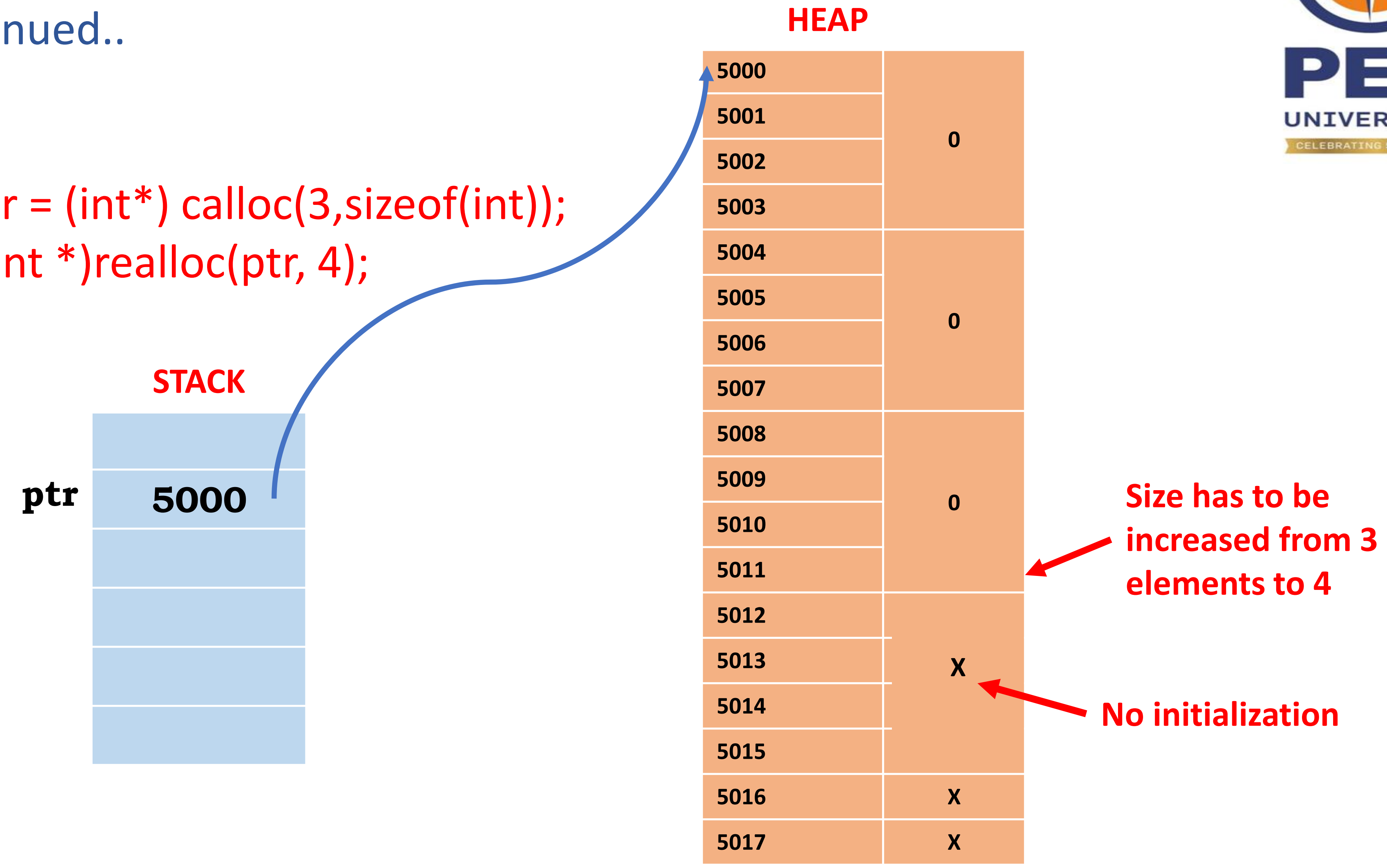
realloc() continued..

- The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location
- If the new size is larger than the old size, then it checks if there is an option to expand or not.
  - If the existing allocation of memory can be extended, it extends it but the added memory will not be initialized.
  - If memory cannot be extended, a new sized memory is allocated, initialized with the same older elements and pointer to this new address is returned. Here also added memory is uninitialized.
- If the new size is lesser than the old size, content of the memory block is preserved.
- Coding examples

realloc() continued..

Example:

```
int* ptr = (int*) calloc(3,sizeof(int));
ptr = (int *)realloc(ptr, 4);
```



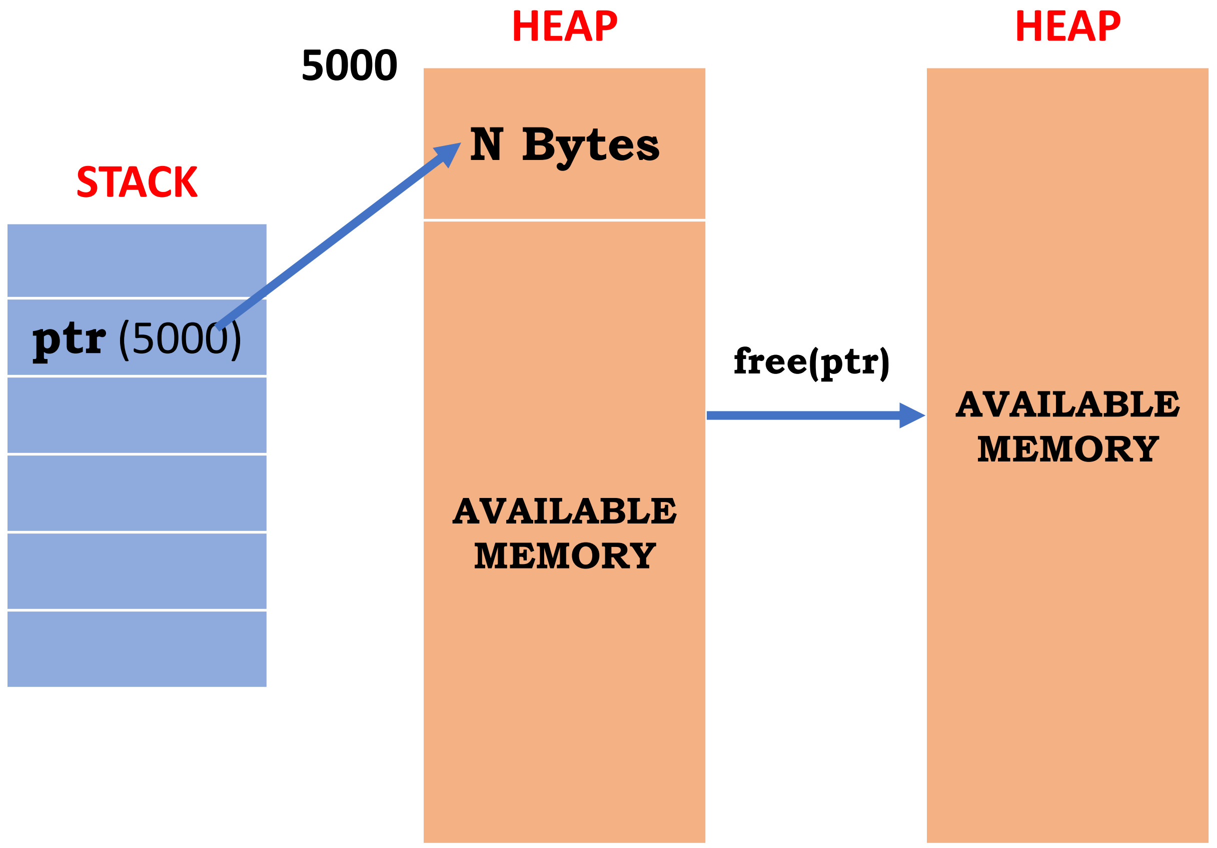
free()

- Releases the allocated memory and returns it back to heap

- **Syntax**:
  free (ptr); //ptr is a pointer to a memory block which has been previously created using malloc/calloc

- No size needs to be mentioned in the free().

- On allocation of memory, the number of bytes allocated is stored somewhere in the memory. This is known as **book keeping information**



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

Dynamic Memory Management continued..

Prof. Sindhu R Pai

Department of Computer Science and Engineering

1. Common Programming Errors

2. Demonstration of Errors

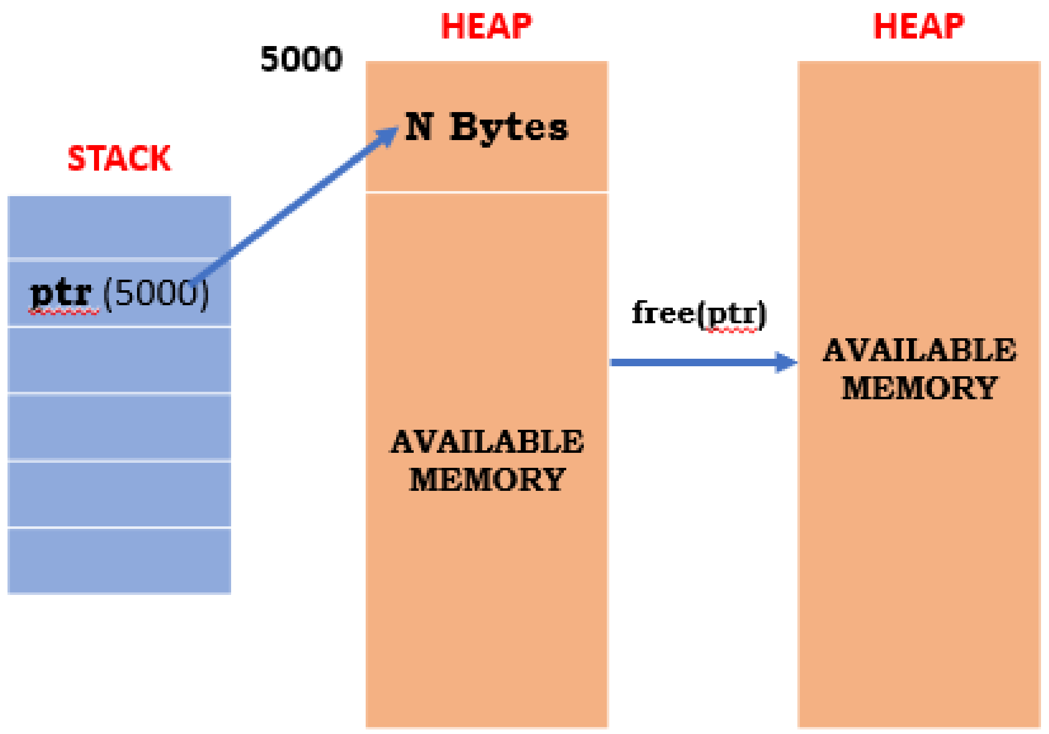
Common Programming Errors

- **Dangling Pointer:**
  - Points to a location which doesn't exist
  - Freeing the memory results in dangling pointer
  - Using new pointer variable to store return address in realloc results in dangling pointer
  - Dereferencing the dangling pointer results in undefined behavior
  - Can happen anywhere in the memory segment
  - Solution is assigning the pointer to NULL

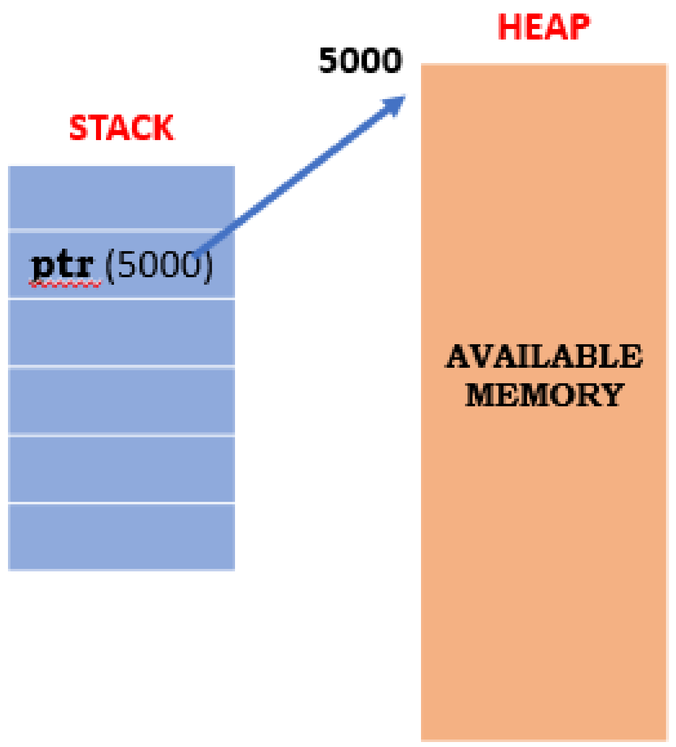
- **NULL Pointer:**
  - **Dereferencing the NULL pointer** results **in Guaranteed crash**

Pictorial Representation of Dangling pointer

- Pointer that points to the memory, which is de-allocated



Using free() to freed the memory space



Memory got freed but the pointer ptr is still pointing to the same address 5000.

Common Programming Errors continued..

- **Garbage:**
  - Garbage is a location which has no name and hence no access
  - If the same pointer is allocated memory more than once using the DMA functions, initially allocated spaces becomes a garbage.
  - **Garbage in turn results in memory leak.**
  - **Memory leak can happen only in Heap region**

- **Double free error:** **DO NOT TRY THIS**
  - If free() is used on a memory that is already freed before
  - Leads to **undefined behavior.**
  - Might corrupt the state of the memory manager that can cause existing blocks of memory to get corrupted or future allocations to fail.
  - Can cause the program to crash or alter the execution flow.

Demonstration of Errors

- C code demo resulting in Errors

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

Structures in C

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Structures in C

- Introduction
- Characteristics
- Declaration
- Accessing members
- Initialization
- Memory allocation
- Comparison

Introduction

- A user-defined data type that allows us to combine data of different types together.
- Helps to construct a complex data type which is more meaningful.
- Provides a single name to refer to a collection of related items of different types.
- Provides a way of storing many different values in variables of potentially different types under the same name.
- Generally useful whenever a lot of data needs to be grouped together.
- Creating a new type decides the binary layout of the type

Characteristics/Properties

- Contains one or more components(homogeneous or heterogeneous) – Generally known as data members. These are named ones.
- **Order of fields and the total size of a variable** of that type is decided when the new type is created
- Size of a structure depends on implementation. Memory allocation would be **at least equal to the sum of the sizes of all the data members** in a structure. Offset is decided at compile time.
- Compatible structures may be assigned to each other.

Syntax :

- Keyword **struct** is used for creating a structure.
- The format for declaring a structure is as below:

```
struct <structure_name>
{
        data_type member1;
        data_type member2;
        …..
        data_type memebern;
};            // semicolon compulsory
```

Example: User defined type Student entity is created.

```
struct Student
{
        int roll_no;
        char name[20];
        int marks;
};
```

Note: No memory allocation for declaration/description of the structure.

Declaration

- Members of a structure can be accessed only when instance variables are created

- If struct Student is the type, the instance variable can be created as:
  struct student s1; // s1 is the instance variable of type struct Student

  struct student* s2; // s2 is the instance variable of type struct student*.
  // s2 is pointer to structure

- Declaration (global) can also be done just after structure body but before semicolon.

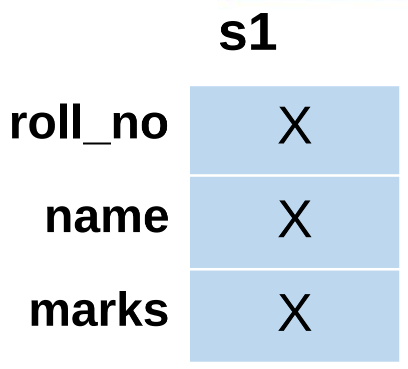


Fig. 1. After declaration, only undefined entries (X)

Initialization

- Structure members can be initialized using curly braces ‘{}’ and separated by comma.
- Data provided during initialization is mapped to its corresponding members by the compiler automatically.
- Further extension of initializations can be:

  1. **Partial initialization**: Few values are provided. Remaining are mapped to zero. For strings, ‘\0’.
  2. **Designated initialization**:
     - Allows structure members to be initialized in any order.
       - This feature has been added in C99 standard.
     - Specify the name of a field to initialize with **‘.member_name =’ OR ‘member_name:’** before the element value. Others are initialized to default value.

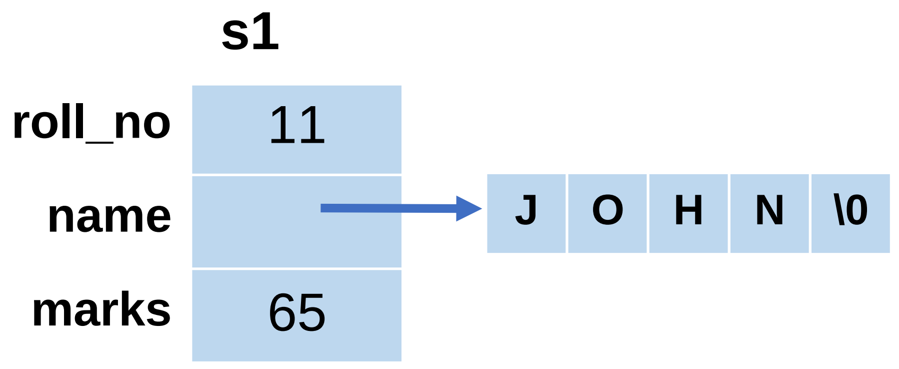


Fig. 2. After initialization, entries are mapped

Accessing data members

- Operators used for accessing the members of a structure.
  1. Dot operator (.)
  2. Arrow operator (->)
- Any member of a structure can be accessed using the structure variable as:

structure_variable_name.member_name

Example: s1.roll_no

//where s1 is the structure variable name and roll_no member is data member of s1.

- Any member of a structure can be accessed using the pointer to a structure as:

pointer_variable->member_name

Example: s2->roll_no

// where s2 is the pointer to structure variable and we want to access roll_no member of s2.

Memory allocation

- **At least equal to the sum of the sizes of all the data members.**
- Size of data members is implementation specific.
- Coding Examples

Comparison of structures

- Comparing structures in C is not permitted to check or compare directly with logical and relational operators.
- Only structure members can be compared with relational operator.
- Coding examples

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

PES UNIVERSITY
CELEBRATING 50 YEARS

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Array of Pointers to Structures

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Array of Pointers to Structures

- Array of Pointers
- Pointers to Structures
- Array of Pointers to Structures
- Demo of C Code

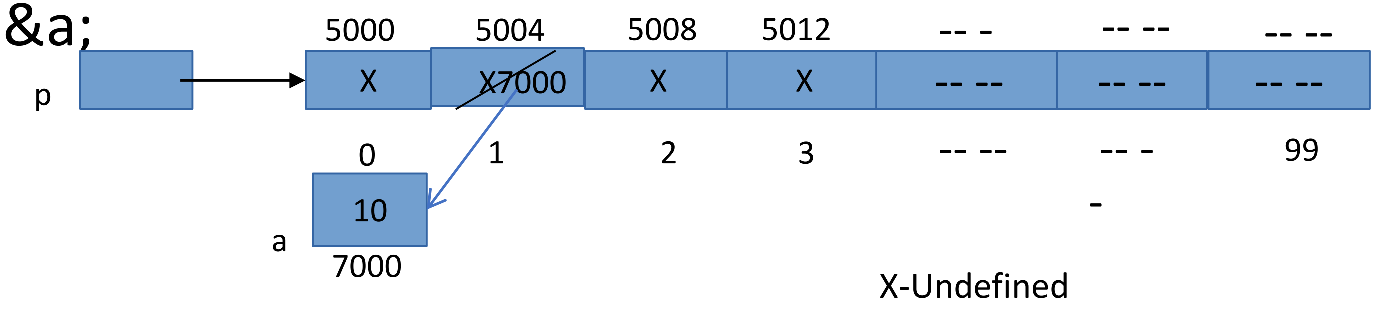
Array of Pointers

- Is an indexed set of variables in which the variables are pointers

  Syntax: int *var_name[array_size];

  Example: int *p[100];

- The element of an array of a pointer can be initialized by assigning the address of other element. Example : int a = 10; p[1] = &a;



- Used to create complex data structures such as linked lists, trees, graphs

Pointers to Structures

- Holds the address of structure variable

- Declaring a pointer to structure is same as pointer to variable

  Syntax: struct tagname *ptr;

- Members of the structure are accessed using arrow (->) operator.

Array of pointers to Structures

- Creating an array of structure variable

  Syntax: struct tagname array-variable[size];

- Creating a pointer variable to hold the address of structure variable

  Syntax: struct tagname *pointer-variable;

- Creating an array of pointers with size specified to hold the addresses of structures in the array of structure variable. **Syntax:** struct tagname *pointer_variable[size];

- Coding Examples

Array of Pointers to Structures

Demo of C Code

- Program to swap first and last elements of the array of structures using array of pointers and display the array of structures using array of pointers.

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

PROBLEM SOLVING WITH C
UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Sorting

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Sorting

1. Why Sorting?

2. Sorting Algorithms

3. Bubble Sort Algorithm

4. Demonstration of C Code

Why Sorting ?

- To access the data in a very quick time
- Think about searching for something in a sorted drawer and unsorted drawer
- If the large data set is sorted based on any of the fields, then it becomes easy to search for a particular data in that set.

Sorting Algorithms

- **Bubble Sort**
- Insertion Sort
- Quick Sort
- Merge Sort
- Radix Sort
- Selection Sort
- Heap Sort

Bubble Sort Algorithm

- An array is traversed from left and adjacent elements are compared and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is continued to find the second largest number and this number is placed in the second place from rightmost end and so on until the data is sorted.

Demonstration of C Code

- Demo of Bubble sort on structures

THANK YOU

Prof. Sindhu R Pai

Department of Computer Science and Engineering

PROBLEM SOLVING WITH C UE23CS151B

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Linked List

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Linked List

1. Introduction
2. Self Referential Structures
3. Characteristics
4. Operations on linked list
5. Pictorial Representation
6. Different Types
7. Applications

Introduction

- Collection of nodes connected via links.
- The node and link has a special meaning in C.
- Few points to think!!

  - Can we have pointer data member inside a structure? - Yes

  - Can we have structure variable inside another structure? - Yes

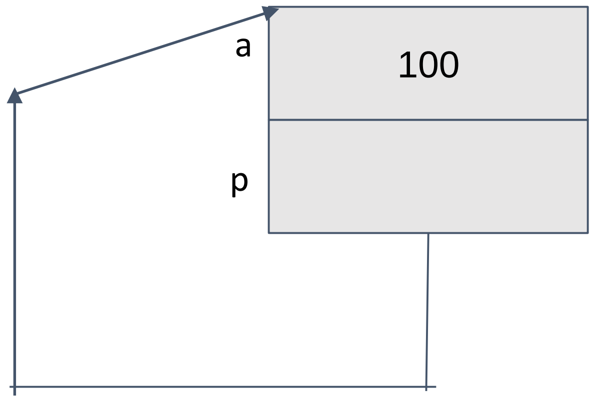
  - Can we have structure variable inside the same structure ? – No

Solution is: Have a pointer of same type inside the structure

Self Referential Structures

- A structure which has a pointer to itself as a data member

```
 struct node
{
              int a;
              struct node *p;
} ; // defines a type struct node
// memory not allocated for type declaration
```



- Variable declaration to allocate memory and assigning values to data members

```
struct node s;    s.a = 100;          s.p = &s;
```

Characteristics

- A data structure which consists of zero or more nodes.
- Every node is composed of two fields: data/component field and a pointer field
- The pointer field of every node points to the next node in the sequence
- Accessing the the nodes is always one after the other. There is no way to access the node directly as random access is not possible in linked list. Lists have sequential access
- Insertion and deletion in a list at a given position requires no shifting of element

Operations on Linked List

- Insertion
- Deletion
- Search
- Display
- Merge
- Concatenate

Pictorial Representation

- Structure definition of a node

```
struct node {
        int info; // component field
        struct node *link; // pointer field
};
typedef struct node NODE_T;
```

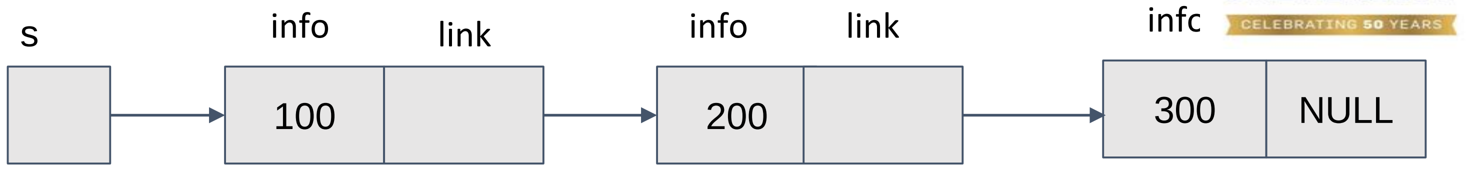


Fig1: Linked list Representation

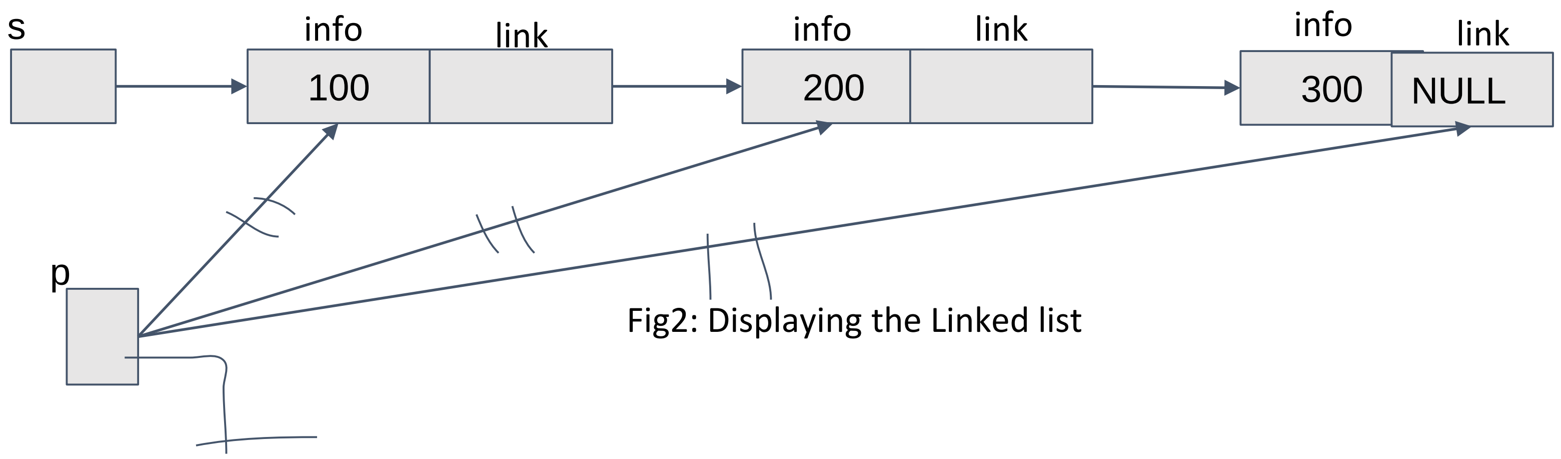
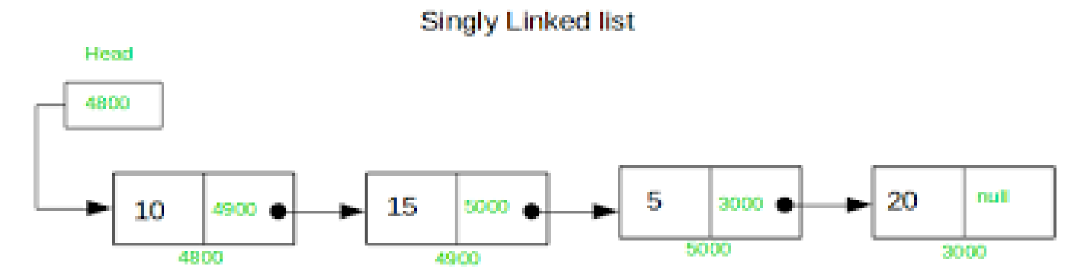
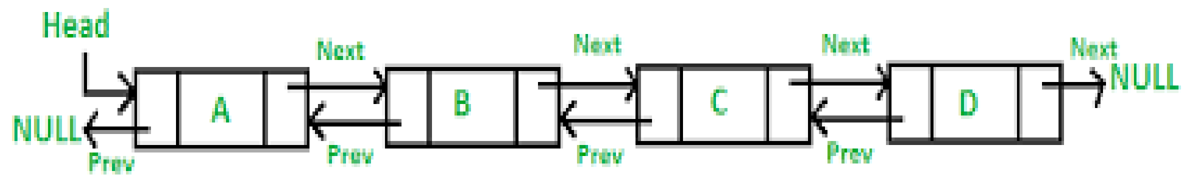


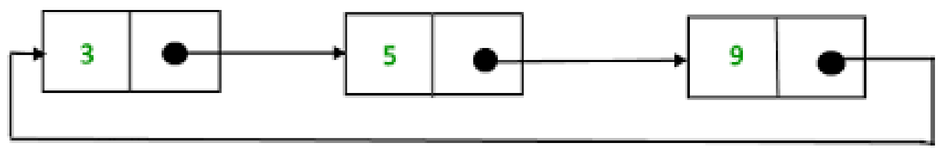
Fig2: Displaying the Linked list

Different Types

- Singly Linked List
- Doubly Linked List
- Circular Linked List







Applications

- Implementation of Stacks & Queues
- Implementation of Graphs
- Maintaining dictionary
- Gaming
- Evaluation of Arithmetic Expression

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

Unions in C

Prof. Sindhu R Pai

Department of Computer Science and Engineering

Unions in C

- What is Union?
- Accessing Union Members
- Union vs Structure

What is Union?

- A user defined data type which may hold members of different sizes and types
- Allow data members which are **mutually exclusive to share the same memory**
- Unions provide an efficient way of using the same memory location for multiple-purpose
    - At a given point in time, only one can exist
- The memory occupied will be large enough to hold the largest member of the union
    - **The size of a union is at least the size of the biggest component**
- All the **fields overlap** and they have the **same offset : 0**.
- Used while coding embedded devices

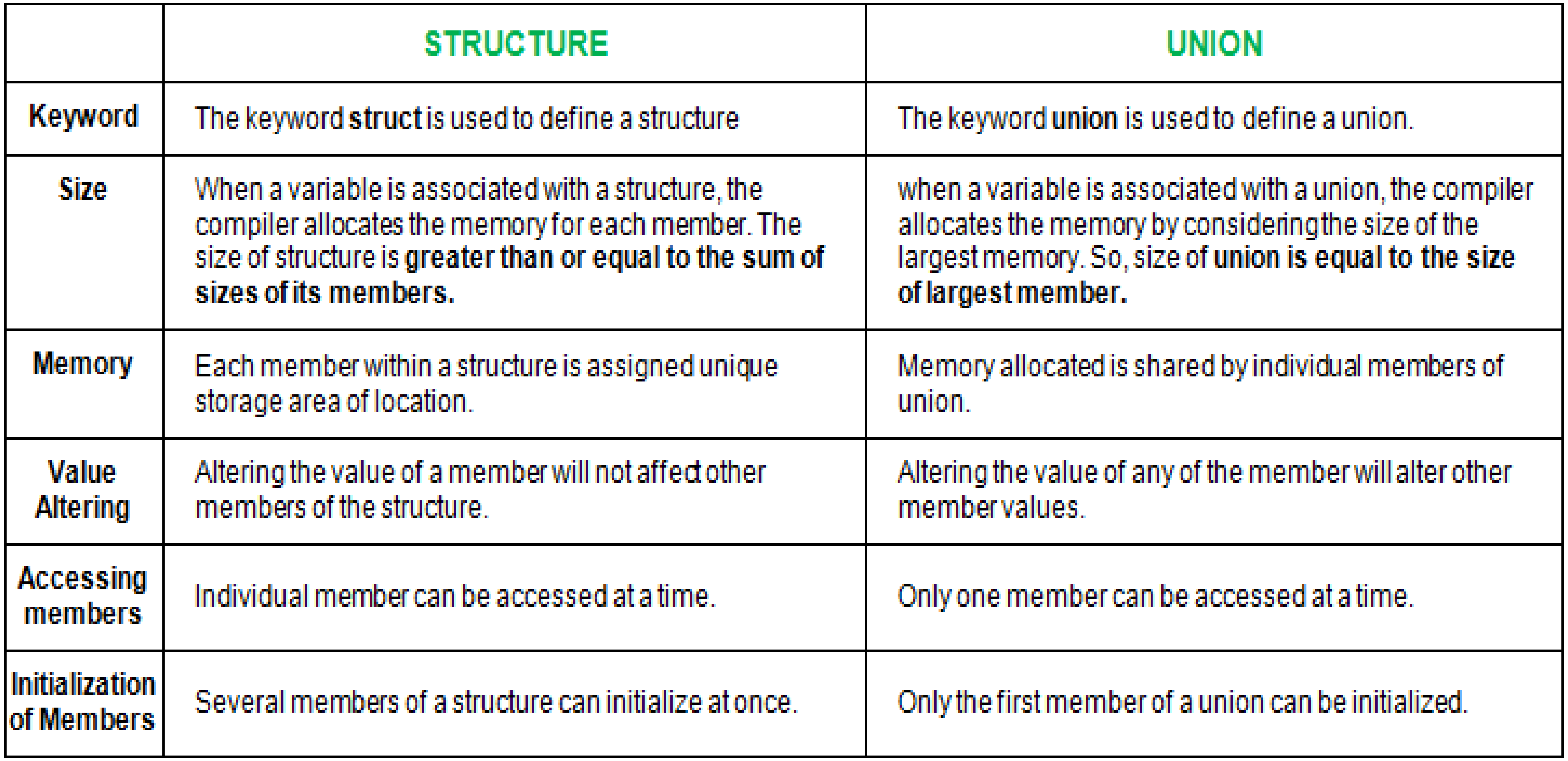
Accessing the Union members

- **Syntax:**

```
union Tag          // union keyword is used
{
data_type member1;     data_type member2;        … data_type member n;
};          // ; is compulsory
```

- To access any member using the variable of union type,
  - use the **Member Access operator (.)**
- To access any member using the variable of pointer to union type,
  - use the **Arrow operator (->)**
- Coding Examples

Union Vs Structure



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by consideringthe size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

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

Bit fields in C

Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C

Bit fields in C

- What is a Bit field?
- Bit field creation
- Few points on Bit fields

What is a Bit field ?

- Data structure that allows the programmer to allocate memory to structures and unions in bits in order to utilize computer memory in an efficient manner.
- The variables defined with a predefined width and can hold more than a single bit
- Consists of a number of adjacent computer memory locations which have been allocated to hold a sequence of bits.
- **Great significance in C programming** because of the following reasons
    - Used to reduce memory consumption
    - Easy to implement
    - Provides flexibility to the code

Bit field Creation

- **Syntax:** struct [tag] { type [member_name] : width ; };

  type - Determines how a bit-field's value is interpreted. May be int, signed int, or unsigned int

  member_name - The name of the bit-field

  width - Number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type. The largest value that can be stored for an unsigned int bit field is 2^n -1, where n is the bit-length

- **Example:** struct Status {

```
        unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be stored 0 and 1
        unsigned int bin2:1; // if it is signed int bin1:1 or int bin1:1, one bit is used to represent the sign
        };
```

- Coding examples

Few points on Bit fields

- The first field always starts with the first bit of the word.
- Cannot extract the address of bit field
- Should be assigned values that are within the range of their size. It is implementation defined to assign an out-of-range value to a bit field member
- Cannot have pointers to bit field members as they may not start at a byte boundary
- Array of bit fields not allowed
- Storage class cannot be used on bit fields
- Can use bit fields in union
- Unnamed bit fields results in forceful alignment of boundary

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

Priority Queue

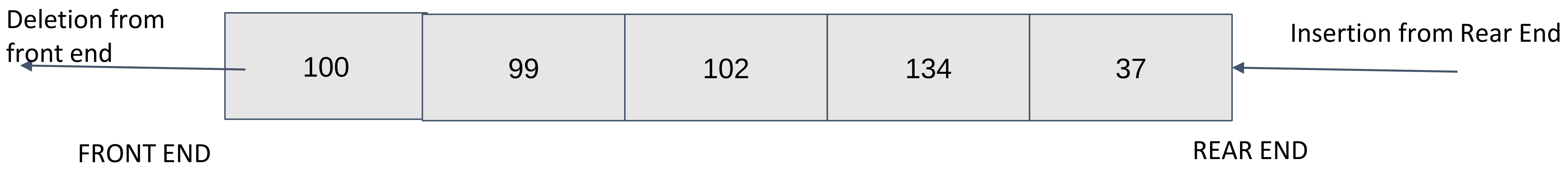
Prof. Sindhu R Pai
Department of Computer Science and Engineering

PROBLEM SOLVING WITH C
Priority Queue

1. Introduction to Queue
2. Operations on Queue
3. Types of Queues
4. Priority Queue
5. Applications of Priority Queue

Introduction to Queue

- A line or a sequence of people or vehicles awaiting for their turn to be attended or to proceed.
- In computer Science, a list of data items, commands, etc., stored so as to be retrievable in a definite order
- A Data structure which has 2 ends – **Rear end and a Front end**. Open ended at both ends
- Data elements are inserted into the queue from the Rear end and deleted from the front end.
- Follows the Principle of **First In First Out (FIFO)**



Operations on Queue

- Enqueue – Add (store) an item to the queue from the Rear end.
- Dequeue – Remove (access) an item from the queue from the Front end.

Types of Queues

- **Ordinary Queue** - Insertion takes place at the Rear end and deletion takes place at the Front end
- **Priority Queue** - Special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue
- **Circular Queue** - Last element points to the first element of queue making circular link.
- **Double ended Queue** - Insertion and Removal of elements can be performed from both front and rear ends

Priority Queue

- Type of Queue where each element has a "**Priority**" associated with it.
- Priority decides about the Deque operation.
- The Enque operation stores the item and the "Priority" information
- Types of Priority Queue:

  Ascending Priority Queue: Smallest Number - Highest Priority

  Descending Priority Queue: Highest Number - Highest Priority

Applications of Priority Queue

1. Implementation of Heap Data structure.
2. Dijkstra’s Shortest Path Algorithm
3. Prim’s Algorithm
4. Data Compression
5. OS - Load Balance Algorithm.

PES
UNIVERSITY
CELEBRATING 50 YEARS

THANK YOU

Prof Sindhu R Pai
Department of Computer Science and Engineering

sindhurpai@pes.edu