Unit #: 3

Unit Name: : Text Processing and User-Defined Types

Topic: Dynamic Memory Management

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Points to think:

- In case of an array, memory is allocated before the execution time. Is there over utilization or under utilization of memory?
- Can we take the size of the array at runtime and request to allocate memory for the array at runtime? – This is Variable length Array(VLA).

  It is not a good idea to use this as there is no functionality in VLA to check the non availability of the space. If the size of large, results in code crash without any intimation.
- Can we avoid this by allocating memory whenever and how much ever we require using some of the functions?

In C, Memory can be allocated for variables using different techniques.

1. Static allocation

decided by the compiler

allocation at load time before the execution or run time

2. Automatic allocation

decided by the compilerallocation at run time

allocation on entry to the block and deallocation on exit

3. Dynamic allocation

code generated by the compiler

allocation and deallocation on call to memory allocation functions:

malloc, calloc, realloc and deallocation function: free

Dynamic Allocation of Memory

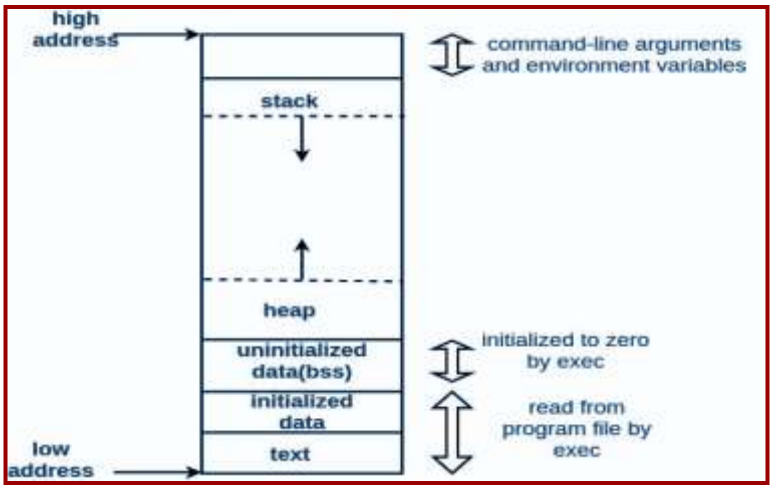
The process of **allocating memory at runtime/execution time** is known as dynamichere are different parts in Memory Layout.

Memory layout of C program

The Operating System does the memory management job for C Program. It helps in the **allocation** and **deallocation** of **memory blocks** either during **compile-time** or during the **run-time**. When the **memory** is **allocated** during **compile-time** it is **stored** in the **Static Memory** and it is

known as **Static Memory Allocation**, and when the **memory** is **allocated** during **run-time,** it is stored in the **Dynamic Memory** and it is known as **Dynamic Memory Allocation**.

There are different segments in the memory layout of a C Program as shown below. Let us know each of these segments to some extent.

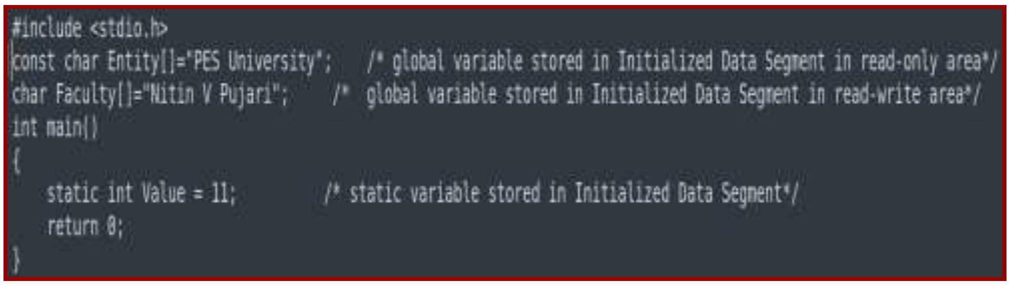

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized data
text
initialized to zero by exec
read from program file by exec
low address

Text segment

It contains **machine code** of the **compiled program**. **Usually**, it is **sharable** so that **only** a **single copy** needs to be in **memory** for **frequently** executed **programs**, such as **text editors**, the **C compiler**, the **shells**, and **so on**. The **text segment** of an **executable object file** is usually **read-only segment** that **prevents** a **program** from being **accidentally modified.**

Initialized Data Segment

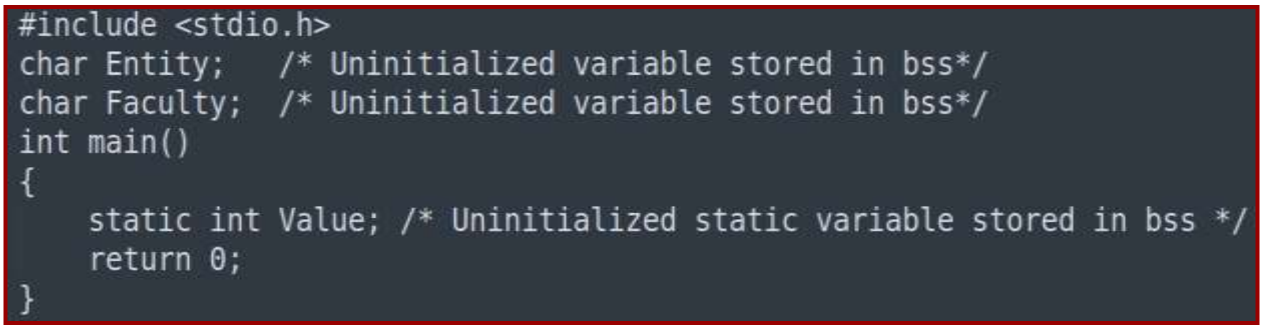
Stores all **global, static, constant, and external variables** - declared with extern keyword that are initialized beforehand. It is **not read-only,** since the values of the variables can be changed at run time. This segment can be further classified into initialized read-only area and initialized read-write area.



```
#include <stdio.h>
const char Entity[]="PES University";   /* global variable stored in Initialized Data Segment in read-only area*/
char Faculty[]="Nitin V Pujari";    /*  global variable stored in Initialized Data Segment in read-write area*/
int main()
{
    static int Value = 11;          /* static variable stored in Initialized Data Segment*/
    return 0;
}
```

Uninitialized Data Segment(bss)

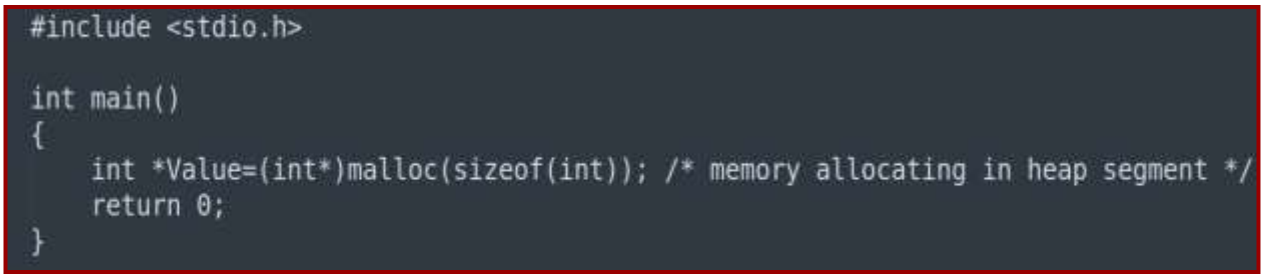
Data in this segment is initialized to arithmetic 0 before execution of the program. Uninitialized data starts at the end of the data segment and contains all **global variables and static variables that are initialized to 0** or **do not have explicit initialization in source code.**



```
#include <stdio.h>
char Entity;   /* Uninitialized variable stored in bss*/
char Faculty;  /* Uninitialized variable stored in bss*/
int main()
{
    static int Value; /* Uninitialized static variable stored in bss */
    return 0;
}
```

Heap Segment

It is the segment where dynamic memory allocation usually takes place. When some more memory need to be allocated using malloc and calloc function, heap grows upward. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.



```
#include <stdio.h>

int main()
{
    int *Value=(int*)malloc(sizeof(int)); /* memory allocating in heap segment */
    return 0;
}
```

Stack Segment

Is used to **store all local variables and is used for passing arguments to the functions** along with the return address of the instruction which is to be executed after the function call is completed.Local variables have a scope to the block where they are defined in, they are created when control enters into the block. All recursive function calls are added to stack.

Note: The stack and heap are traditionally located at opposite ends of the process's virtual address space

More details on Memory layout of C can be studied in the below link. https://www.geeksforgeeks.org/memory-layout-of-c-program/

There is **no operator in C to support dynamic memory management**. Library routines known as "Memory management functions" are used for allocating and freeing/releasing memory during execution of a program. These functions are defined in **stdlib.h**

1. **malloc():**

Allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space. Prototype**: void *malloc(size_t size);**

The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is **not initialized.** This returns a void pointer to the allocated memory that is suitably aligned for any kind of variable. On error, these functions return NULL. NULL may also be returned by a successful call to malloc() with a size of zero.

2. **calloc():**

Allocates space for elements, initialize them to zero and then return a void pointer to the memory. prototype: **void *calloc(size_t nmemb, size_t size);**

The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The calloc() function return a pointer to the allocated memory that is suitably aligned for any kind of variable. On error, this function returns NULL. NULL may also be returned by a successful call to calloc() with nmemb or size equal to zero.

3. **realloc():**

Modifies the size of previously allocated space using above functions. copies the old values into the new memory locations. Prototype: **void *realloc(void *ptr, size_t size);**

This function changes the size of the memory block pointed to by ptr to size bytes. 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, the added memory will not be initialized.

Note:

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).

Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done. The realloc() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from ptr, or NULL if the request fails. If size was equal to 0, either NULL or a pointer suitable to be passed to free() is returned. If realloc() fails the original block is left untouched, and it is not freed or moved

4. free():

Releases the allocated memory and returns it back to heap. Must be applied with malloc(), calloc() or realloc()

Let us consider the Memory Allocation using malloc

Coding Exampe_1:

```
int* p = (int*)malloc(sizeof(int)); //dynamic allocation for one integer
//address of that location is returned in p
*p = 10;
  printf("p = %d", *p);
```

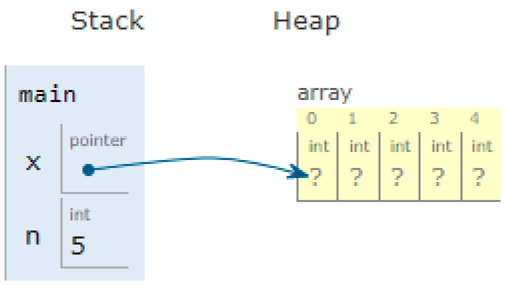
Coding Exampe_2:

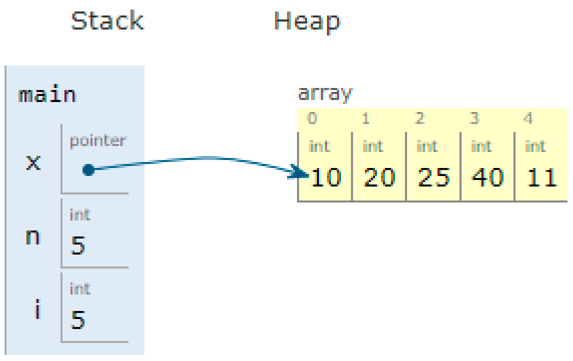
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *x; int n, i;
printf("Enter the number of elements\n");
scanf("%d",&n);
x = (int*)malloc(n * sizeof(int)); //memory will be allocated for 5 integer numbers
if (x == NULL)                     //to check if memory is allocated or not by malloc
   printf("Memory not allocated.\n");
else
{
      printf("Memory successfully allocated using malloc.\n");
      printf("Enter the integer values:\n ");
```

```
        for (i = 0; i < n; i++) // To read inputs from user
        {
            scanf("%d",&x[i]);
         }
        printf("Output: ");// Printing
        for (i = 0; i < n; i++)
              printf("%d\t", *(x+i));
    }
    return 0;
    }
```

Output:

```
    Enter the number of elements
    5
    Memory successfully allocated using malloc.
    Enter the integer values:
     10
     20
     25
     40
     11
     Output: 10   20      25      40      11
```





Coding Exampe_3: Using calloc

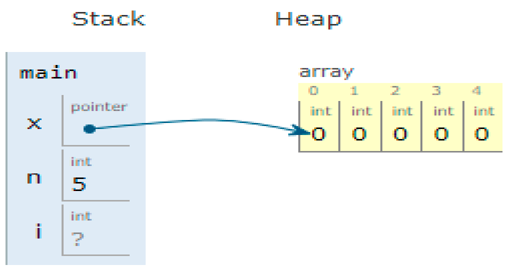
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int *x; int n, i  ;
        printf("Enter the number of elements\n");
        scanf("%d",&n);
        x = (int*)calloc(n,sizeof(int));
                // a block of n integer(array) is created dynamically
        if (x == NULL)          //to check if memory is allocated or not
                printf("Memory not allocated.\n");

        else
        {
                printf("Memory successfully allocated using calloc.\n");
                printf("Enter the integer values:\n ");
                for (i = 0; i < n; i++) // To read inputs from user
                {
                        scanf("%d",&x[i]);
                }
                printf("Output: ");// Printing
                for (i = 0; i < n; i++)
                        printf("%d\t", *(x+i));
        }
        return 0;
}
```

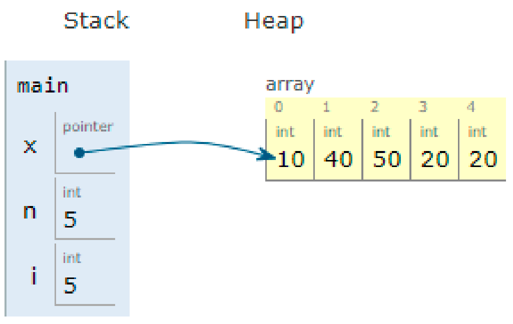
Output:

```
Enter the number of elements 5
Memory successfully allocated using calloc. Enter the integer values:
10
```

```
40
50
20
20
```

Output: 10 40 50 20 20





Coding Exampe_4: realloc

```
#include<stdio.h>
#include<stdlib.h>
int new_size;
int main()
{
    int* p = (int*)malloc(3*sizeof(int));
    if(p == NULL)
        printf("memory not allocated\n");
    else
    {
        int i;
        printf("initial address is %p\n",p);
        printf("enter three elements\n");
        for(i = 0;i<3;i++)
        {
            scanf("%d",&p[i]);
        }
```

```
                printf("entered elements are\n"); for(i = 0;i<3;i++)
                {
                        printf("%d\t",p[i]);
                }
        }
        printf("enter the new size\n");
         scanf("%d",&new_size);
         p = (int*)realloc(p,new_size*sizeof(int));
         if(p==NULL)
                        printf("not allocated\n");
        else
        {
                int i;
                printf("\nnew address is %p\n",p);
                for(i = 0;i<new_size;i++)
                {
                                printf("%d\t",p[i]);
                }
        }
        return 0;
}
```

Output 1:

```
initial address is 0x5570412b72a0
enter three elements
10
20
30
entered elements are 10    20    30
enter the new size
5
new address is 0x5570412b72a0
10   20   30      0        0
```

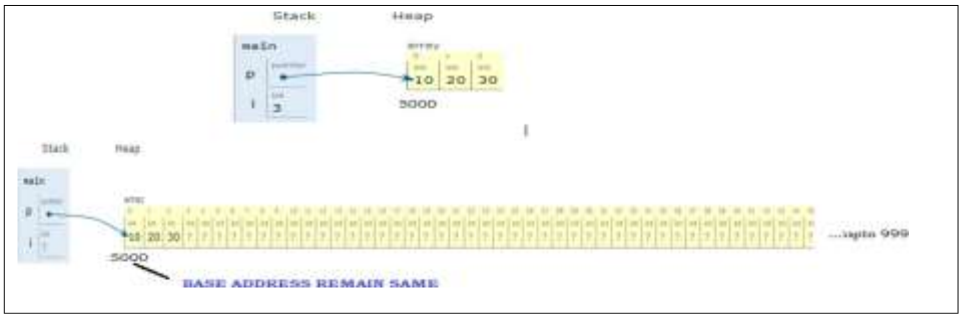
Output 2:

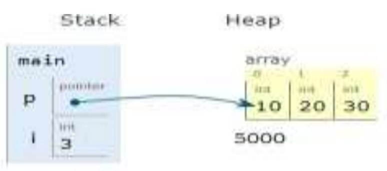
```
initial address is 0x555c214b42a0 enter three elements
10
20
30
entered elements are 10  20        30
enter the new size
2
new address is 0x555c214b42a0
10  20
```

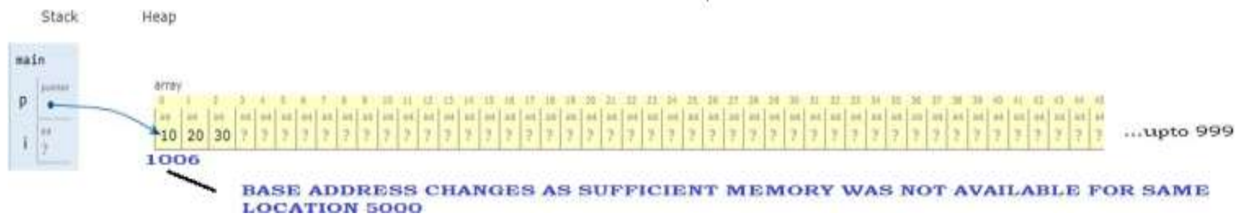
Output 3:

```
initial address is 0x5632e1fc52a0
enter three elements 10
20
30
entered elements are 10  20        30
enter the new size
15
new address is 0x5632e1fc5ae0
10  20      30  0     0   0  0   0        0         0         0         0         0  0   0
```

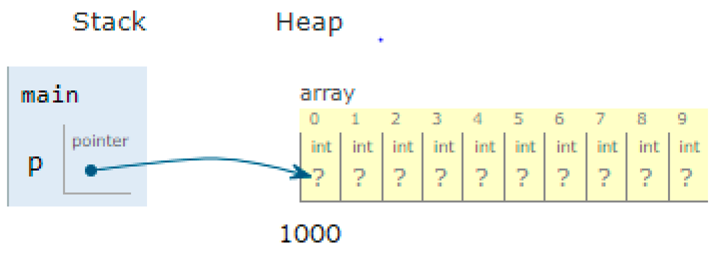
When new size is greater than 3, here are the pictorial representations

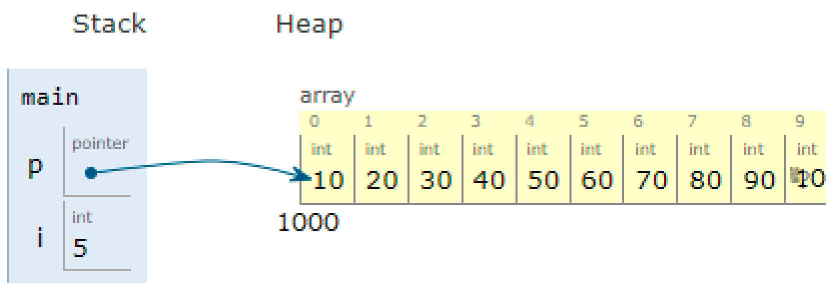


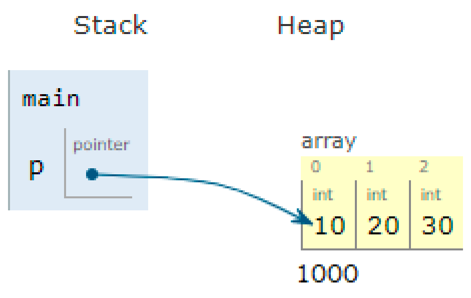



Stack
Heap
main
p
pointer
i
array
10 20 30 ? ? ?
...upto 999
1006
BASE ADDRESS CHANGES AS SUFFICIENT MEMORY WAS NOT AVAILABLE FOR SAME LOCATION 5000

When new size is less than 3, here is the pictorial representation





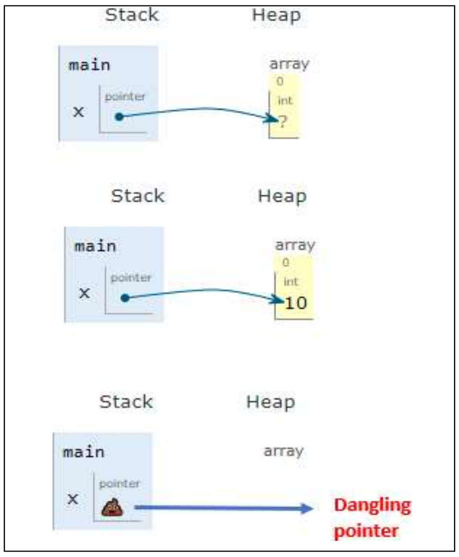


Coding Exampe_5: free

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *x;
    x = (int*)malloc(sizeof(int));
    *x = 10;
    printf ("\nx = %p", x);
    printf("\n x is pointing to = %d", *x);
    // Free the memory free(x);
    printf("\nMalloc Memory successfully freed.\n");
}
```

Output:

```
x = 0x561aedb012a0 x is pointing to = 10
Malloc Memory successfully freed.
```



How does free function knows how much memory must be returned/released?

On allocation of Memory using memory management functions, it stores somewhere in memory the # of bytes allocated. This is known as **book keeping information**. We cannot access this info. But implementation has access to it. The function free finds this size given the pointer returned by allocation functions and de-allocates the required amount of memory.

By observing both the diagrams above, we can get to know that block allocated using functions will be deallocated and hence the pointer becomes dangling.

Points to think:

- Can free function take one more argument along with the pointer? -- No.
- Can you use free on the same pointer twice?

Happy Coding!!

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Dynamic Memory Management – Common Errors

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC

Jan-May, 2022

Dept. of CSE

PES University

Common Programming Errors that might occur during Dynamic Memory Management are discussed one by one in this lecture notes

Dangling Pointer

A pointer which points to a location that doesn't exist is known as dangling pointer. It can happen anywhere in the memory segment. Solution is to assign the pointer to NULL. Situations that results in dangling pointer are as below.

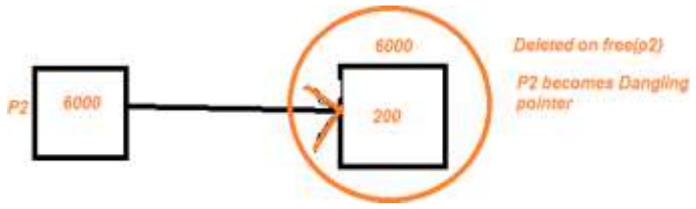
- Freeing the memory results in dangling pointer
- Using new pointer variable to store return address in realloc results in dangling pointer

Note: Dereferencing the dangling pointer results in undefined behavior

Case 1: Freeing the memory results in dangling pointer

Coding Example_1:

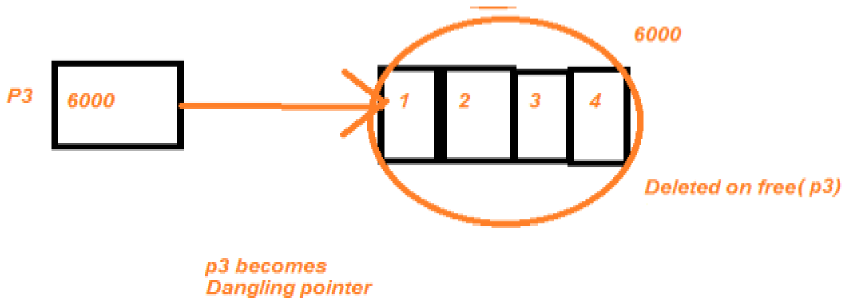
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
      int *p2 = (int*)malloc(sizeof(int));             // conversion from void* to int*
      printf("%p\n",p2);
     *p2 = 200;
      printf("%p\n", p2);
      printf(" %d\n", *p2);                          // 200
       free(p2);  return 0;
```



```
 }
```

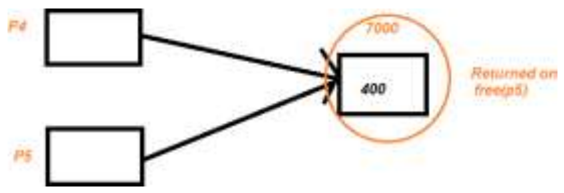
Coding Example_2:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    printf("Enter the number of integers you want to store\n");
    int n; int i;
    scanf("%d", &n);                                    // user entered 4
    int *p3 = (int*)malloc(n*sizeof(int));// initially all values undefinedvalues
    printf("Enter %d elements\n", n);
    for(i = 0;i<n; i++)
        scanf("%d",&p3[i]);   // (p3+i)// user entered 1 2 3  4
    printf("Entered elements are\n");
    for(i = 0;i<n; i++)
         printf("%d\n",p3[i]);                   // *(p3+i)
   free(p3);
   return 0;
}                                                 // returns the memory allocated using malloc
```



Coding Example_3:

```
int *p4 = (int*)malloc(sizeof(int));
*p4 = 400;
printf(" %d\n", *p4);
int *p5 = p4;
free(p5);                        // value of the pointer has the book keeping info available

//Both p4 and p5 becomes dangling pointer
```

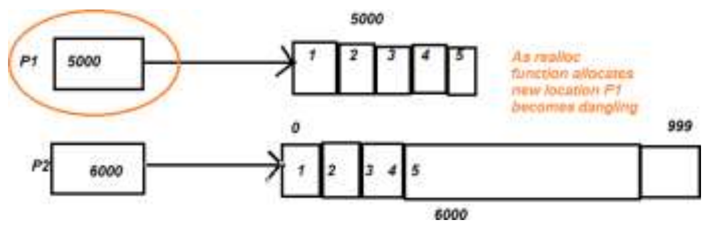


Case 2: Using new pointer variable to store return address in realloc results in dangling pointer

Coding Example_4:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *p1 = (int *) calloc(5, sizeof(int));
printf("before %p\n", p1);
printf("enter the elements\n");
for(int i = 0; i < 5; i++)
 scanf("%d", &p1[i]);//given input 1,2,3,4,5
 for(int i = 0; i < 5; i++)
 printf("%d ", p1[i]);
 int *p2 = (int*) realloc(p1, 1000*sizeof(int)); // p1 becomes dangling if new locations allotted
printf("after increasing size %p\n", p2); // same address as p1 if same location can be extended
//else Different address
printf("content at address pointed by p2 = %d\n", *p2);
printf("after realloc %p\n", p1);
printf("contents at address pointed by p1 = %d\n", *p1);
return 0;
}
```

//undefined behaviour- p1 becomes dangling pointer

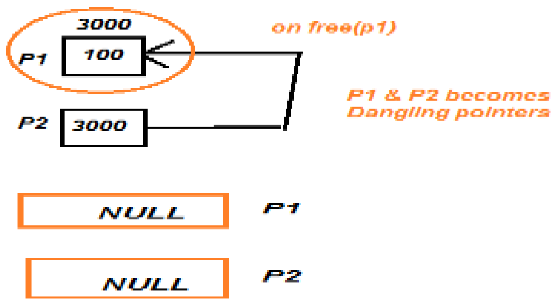

P1
5000
5000
1
2
3
4
5
As realloc function allocates new location P1 becomes dangling
P2
6000
0
1
2
3
4
5
999
6000

NULL Pointer

It is always a good practice to assign the pointer to NULL once after the usage of free on the pointer to avoid dangling pointers. This results in NULL Pointer. **Dereferencing the NULL pointer results in guaranteed crash**

Coding Example_5:

```
int *p1 = (int*)malloc(sizeof(int));
*p1 = 100;
printf(" %d\n", *p1); // 100int
*p2 = p1;
printf(" %d\n", *p2); // 100
free(p1); // p1 and p2 both becomes dangling pointer.p1 =
NULL; // safe programming
p2 = NULL;
printf(" %d\n", *p1);// Guaranteed crash
printf(" %d\n", *p2);
```



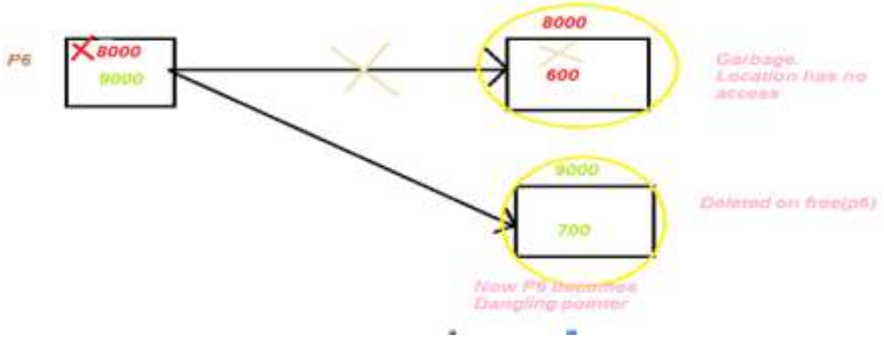
Assign Null Value after calling free(pointer).

Garbage and Memory Leak

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

Coding Example_6:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{

        int *p6 = (int*)malloc(sizeof(int));
        *p6 = 600;
        printf(" %d\n", *p6); // 600
        p6  =  (int*)malloc(sizeof(int));
// changing p6 loses the pointer to the
//location allocated by the previous malloc
// So memory allocated by previous malloc has no access. Hence becomes garbage
        *p6 = 700;
        printf(" %d\n", *p6);
        free(p6);
        return 0;
}
```


P6
8000
9000
8000
600
Garbage. Location has no access
9000
700
Deleted on free(p6)
Now P6 Becomes Dangling pointer

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 blocksof memory to get corrupted or future allocations to fail.
- Can cause the program to crash or alter the execution flow

Coding Example_7:

```
int* p = (int*)malloc(sizeof(int));
*p = 10;
printf("p = %d", *p);
free(p);
free(p);
```

The above code might lead to undefined behavior. So, it is a good practice to make the pointer NULL after the usage of free. By chance, if the pointer is freed again, it doesn't do anything with NULL.

```
free(p);
p = NULL; //Function free does nothing with a NULL pointer.
free(p);
```

Happy Freeing!!

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Programs on Strings

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

List of Problems

1. Program to find the count of given character in a given string.
2. WAP to input a string and print the Square of all the Numbers in a String. Take care of special cases. Input: wha4ts12ap1p and Output: 16 1 4 1
3. Write a function to find the position of First occurrence of the input character in a given string.
4. Write a function which takes two string arguments and find the Cartesian product of passed strings.

   Input: str1: abc str2: ab and Output: aa, ab, ba, bb, ca, cb
5. WAP to find the sum of all substrings of a string representing a number.

   Input : num = "1234”

   Output : 1670 Sum = 1+2 + 3 + 4 + 12 + 23 + 34 + 123 + 234 + 1234 = 1670

6. Write a function to reverse a string using recursion and test the function
7. Write a program to delete the characters from string1 which are present in string2.

   input: string1: whatsapp string2: wat output: hspp

8. Write a program to find if two strings are anagrams.

   Anagrams: Same set of letters with different arrangements. Number of times a letter present in one string must be equal to the number of times the letter is present on another string

A few Solutions

1. Program to find the count of given character in a given string.

Solution:

```
int main()
{
    char str1[100]; printf("enter the string\n");
    scanf("%[^\n]s",str1);      // To receive multiple words and store multiple words under one name
   printf("enter the character\n");
```

```
        fflush(stdin);
        char ch = getchar();
        int count = find_frequency(str1, ch);
        printf("%c is present in %s, %d times\n",ch, str1,count);
        return 0;
}
int find_frequency(char *s, char ch)
{
        int count = 0;
        for(; *s ; s++)                 // exit the loop when *s is '\0'
        {
                        if (*s == ch)   count++;
        }
        return count;
 }
```

2. WAP to input a string and print the Square of all the Numbers in a String. Take care of special cases. Input: wha4ts12ap1p and Output: 16 1 4 1

Solution:

```
#include<stdio.h>
int sqr(int r);
void extract_num_square(char *c);
int main()
{
        char ch[400];
        scanf("%s",ch); //wha4ts12ap1p
        extract_num_square(ch);
        return 0;
}
int sqr(int r)
{       return r*r;     }
```

```
void extract_num_square(char *c)
{
        while(*c != '\0')
        {
                if((*c >= '1' &&  *c <= '9'))
                {
                        printf("%c--",(*c));
                        printf("%d\n",sqr(*c-'0'));
                }
                c++;
        }
}
```

3. Write a function to find the position of First occurrence of the input character in a given string.

   Solution:

```
#include<stdio.h>
int find_first(char* ,char);
int main()
{
    char text[100];
    printf("enter the text\n");
    scanf("%[^\n]s",text);          // Till \n, receive input and store in one char array variable
    fflush(stdin);
    printf("enter the character which u want to find\n");
    char ch = getchar();
    int res = find_first(text,ch);
    if(res == -1)
```

```
                printf("not present in the text\n");
        else
                printf("available at %d position\n",res);
        return 0;
}
```

Different versions of find_first function are as below.

Version 1:

```
int find_first(char *a,char c)
{
     int p = -1;
     int i;
     int check = 0;
     for(i = 0;i<strlen(a);++i)
     {
          if(a[i]==c && check == 0)
         {
                      p = i; check = 1;
              }
         }
         return p;
}
```

Version 2:

```
int find_first(char *a, char c)
{
     int i = 0;
     while((a[i]) && a[i] != ch )
     {
                 i++;
     }
     return a[i]? i: -1;
}
```

Version 3:

```
int find_first(char *a,char ch)
{
      int i = 0;
      while(*a && *a != ch )
      {
                  a++;i++;
      }
      return (*a) ? i : -1;
 }
```

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Problem Solving – Text Processing(String Matching)

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C contructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Problem Solving: String Matching

The problem of finding occurrence(s) of a pattern string within another string or body of text is known as String Matching. Also known as exact string matching, string searching, text searching.

There are **different algorithms** for efficient string matching.
Brute Force Algorithm/ Naïve String-search Algorithm
Horspool Algorithm
Boyer Moore Algorithm
Rabin – Karp Algorithm
Knuth – Morris- Pratt Algorithm

Let us define the problem to be solved.

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function pattern_match(char txt[], int n, char pattern[], int m) that returns the position of the character in the text string when the first occurrence of pattern is matched with the text string.

Here, n is the length of the text string and m is the length of the pattern string. i is the loop variable to traverse through the text string. j is the loop variable to traverse through the pattern. Whenever characters do not match in text and pattern, increment only i. Then again ith character from text must be compared with jth character of Pattern. Whenever characters in both strings match, increment i and j both

Requirements to solve this problem are listed here.

- Read a Text string and Pattern string
- Usage of loop variables to traverse across two strings
- Action to be taken when characters do not match in text and pattern
- Action to be taken whenever characters in both strings match

Demonstration of the C Solution:

```
#include<stdio.h>
#include<string.h>
int pattern_match(char text[],int n,char pat[],int m);

int main()
{
        char text[100];
        char pat[100];
        printf("enter the string\n");
        scanf("%[^\n]s",text);
        fflush(stdin);
        printf("enter the pattern\n");
        scanf("%[^\n]s",pat);
        int n = strlen(text);
        int m = strlen(pat);
        int pos = pattern_match(text,n,pat,m);
        if(pos == -1)
                printf("pattern not found\n");
        else
                printf("pattern is found at %d\n",pos);
        return 0;
}


//version 1: Having multiple return statement in the same function is not a good
programmer's habit.
int pattern_match(char text[],int n,char pat[],int m)
{
        int i; int j;
        for(i = 0;i<=n-m;i++)
        {
                for(j = 0; j<m && text[i+j] ==pat[j];j++);
                if(j==m)
                        return i;
        }
        return -1;
}
```

Version 2: Modified the usage of return twice.

```
int pattern_match(char text[],int n,char pat[],int m)
{
        int i; int j;
        int res = -1; // added  inthis version
        for(i = 0;res == -1 && i<=n-m;i++)
        {
                for(j = 0; j<m && text[i+j] ==pat[j];j++);
                if(j==m)
                        res = i;  // modified in this version of code
        }
        return res;                //modified.
}
```

If res is initialized to -1 and res == -1 in the outer for loop, what happens? Are these two statements required? If yes, Why? Think!

Think about the solution for Pattern matching using Recursion.

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Structures in C

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Introduction

Structure is a user-defined data type in C language which allows us to combine data of different types together. It helps to construct a complex data type which is more meaningful. It is often convenient to have a single name with which to refer to a **collection of related items of different types**. Structures provide a way of storing many different values in variables of potentially different types under the same name. Structures are generally useful whenever a lot of data needs to be grouped together. For instance, they can be used to hold records from a database or to store information about contacts in an address book.

Why Structures?

Need of Structure is discussed through below mentioned programs. Let us try to interactively store the Roll_number, Total_marks and Name of 20 students.

```
char names[20][15];
int marks[20]; int roll_num[20];
for(int i=0;i<20;i++)
{
        printf("Enter Roll_number, marks and name\n"); scanf("%d",&roll_num[i]);
        scanf("%d",&marks[i]);
        scanf("%[^\n]s",names[i]);
}
```

Display the entries and check whether this code works. By using separate arrays for each of the details of students, nowhere we saying all these three are related. It means, no where we specifying that it is the same set of 20 students whose details are taken. To make sure that we have the relation between the data items, we need Structures.

Introduction to Structures

A structure creates a data type that can be used to group items of possibly different types into a single type. Grouping of attributes into a single entity is known as Structure. Creating a new type decides the binary layout of the type. Order of fields and the total size of a variable of that type is decided when the new type is created.

Characteristics/properties:

- A user defined data type/non-primary/secondary
- Contains one or more components – Generally known as data members. These are named ones.
- Collection of homogeneous or heterogeneous data members
- 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.
- The members of a structure do not occupy space in memory until they are associated with structure variable.
- Compatible structures may be assigned to each other.

To define a new type/entity, use the **keyword - struct**

The format for declaring a structure is as below:

```
struct Tag
{
        data_type member1;
        data_type member2;
        …..
        data_type membern;
};                                  // Don't forget semicolon here
```

where Tag is the name of the entire structure and Members are the elements within the struct.

Example: User defined type Student entity is created.

```
struct Student
{
        int roll_no;
        char name[20];
        int marks;
 };//No memory allocation for declaration/description of the structure.
```

Accessing members of a structure

All the data members inside a structure are accessible to the functions defined outside the structure. To access the data members in a function including main, we need to create a structure variable. We can access the members of a structure only when we create instance variables of that type. If struct Student is the type, we can create the instance variable as:

struct Student s1; // s1 is the instance variable of type struct Student. Now memory gets allocated for data members

struct Student* s2; // s2 is the instance variable of type struct Student*. Now memory gets allocated for pointer. s2 is pointer to structure

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

Suppose, s1 is the structure variable name and we want to access roll_no member of s1. Then, it can be accessed as: s1.roll_no

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

pointer_variable->member_name

Suppose, s2 is the pointer to structure variable and we want to access roll_no member of s2. Then, it can be accessed as: s2->roll_no

Declaration and Initialization

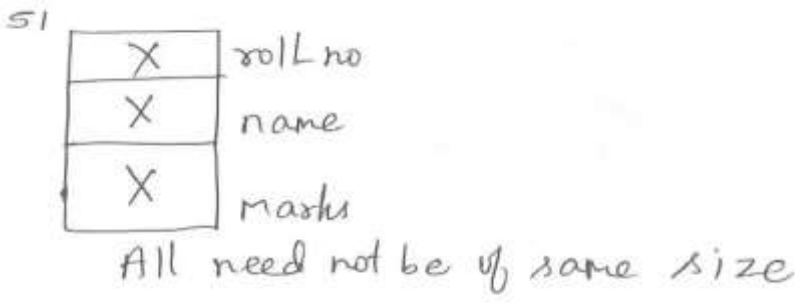
Coding Example_1:

```
struct student              // template declaration which describes how student entity looks like
{
        int roll_no;                    // these three are members of the structure
        char name[20];
        int marks;
};
```

Given the above user defined type, look at below client codes

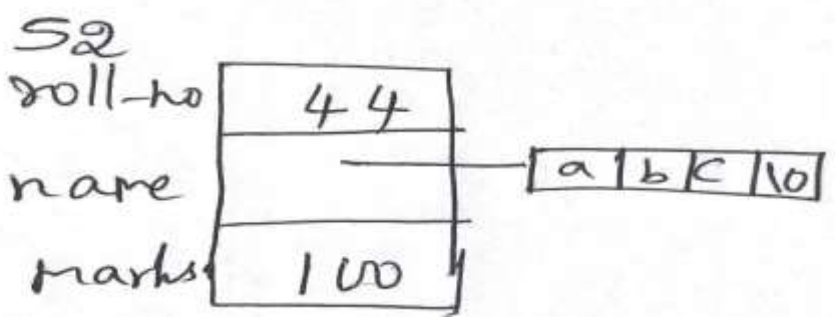
Version 1:

```
struct student s1;                    // Declaration
printf("%s\n",s1.name);
printf("%d\n",s1.roll_no);
printf("%d\n",s1.marks);
// some undefined value gets printed.
```



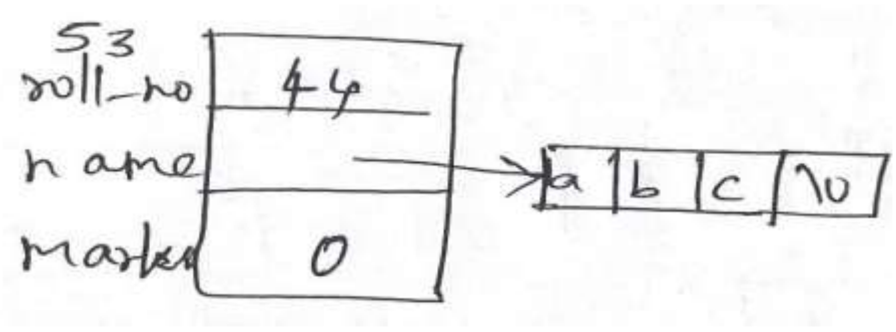
Version 2:

```
struct student s2 = {44, "abc", 100};                 // Initialization
printf("%d %d %s", s2.marks, s2.roll_no, s2.name);        // 100 44 abc
```



Version 3:

```
struct student s3 = {44, "abc"};        // Partial Initialization
// Explicitly few members are initialized. others are initialized to default value
printf("%d %d %s", s3.marks, s3.roll_no, s3.name);        // 0 44  abc
```



Version 4: Designated Initializers in C

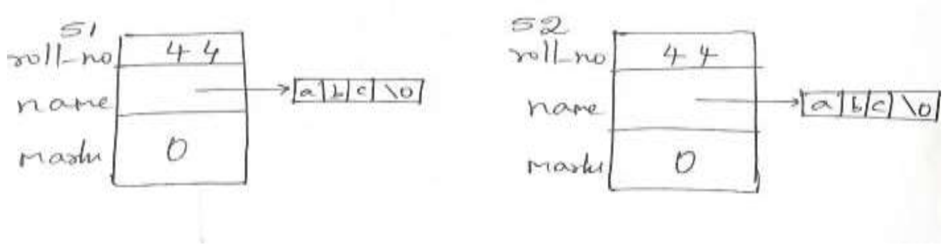
Specify the name of a field to initialize with '.member_name =' or 'member_name:' before the element value. others are initialized to default value.

struct student s1 = {.name = "abc", .roll_no = 44};

struct student s2 = {name : "abc", roll_no : 44};

printf("%d %d %s\n", s1.roll_no, s1.marks, s1.name); // 44 0 abc

printf("%d %d %s", s2.roll_no, s2.marks, s2.name); // 44 0 abc



Memory Allocation for Structure Variables

Every data type in C has alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.

More details, Refer to below links.

https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

https://www.geeksforgeeks.org/data-structure-alignment/

What is the minimum number of bytes allocated for the structure variable in each of these cases? – Sum of the sizes of all the data members. Size of data members is implementation specific.

What is the maximum number of bytes allocated for the structure variable in each of these cases? **Implementation Specific.**

Coding Example_2: Below programs are run on Windows7 machine and 32 bit GCC compiler

```
struct test
{       int i;      char j;        };
struct test1
{       char j;      int i;        };
struct test2
{      char k;     char j;     int i;      };
struct test3
{       int i;       char k;      int j;      };
int main()
{
        printf("size of the structure is %lu\n",sizeof(struct test));    //8bytes          4+4
        struct test t;
        printf("size of the structure is %lu\n",sizeof(t));                //     8bytes      4+4
        printf("size of the structure is %lu\n",sizeof(struct test1));  //8bytes          4+4
        printf("size of the structure is %lu\n",sizeof(struct test2));  //8bytes          4+4
        printf("size of the structure is %lu\n",sizeof(struct test3));  //12bytes         4+4+4
        return 0;
}
```

Comparison of Structures

Coding Example_3:

```
        struct testing
        {
                int a;   float b;          char c;
        };
```

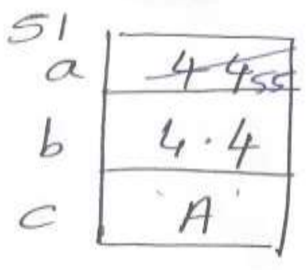
```
int main()
{
        struct testing s1 = {44, 4.4, 'A'};
        struct testing s2 = {44, 4.4, ‘A’};
        //printf(“%d”, s1== s2); // Compiletime Error.
        // == operator cannot be applied on structures for comparing
        // Think about s1- s2
        // Write a function to check for equality of every member of the structure
        printf(“%d”,is_equal(s1,s2));          // 1      // all fields are equal
        struct testing s3 = {33, 3.3, ‘A’};
        printf(“%d”,is_equal(s1,s3));          // 0      //first condition itself fails
        return 0;
}
int is_equal(struct testing s1, struct testing s2)
{           return (s1.a == s2.a && s1.b == s2.b && s1.c == s2.c);   }
```

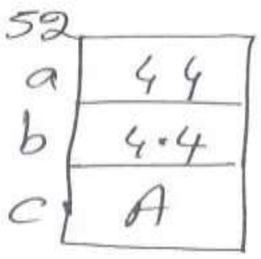
Member - wise copy

Structures of same type are assignment compatible. When you assign one structure variable to another structure variable of same type, member- wise copy happens. Both of them do not point to the same memory location. The compiler generates the code for member – wise copy.

Coding Example_4:

```
struct testing

{       int a;
        float b;
        char c;
};
int main()
{       struct testing s1 = {44,4.4, 'A'};
        struct testing s2 = s1;
        s1.a = 55;
        printf("%d\n",s1.a);
        printf("%d\n",s2.a);
        return 0;
}
```





Few points to think:

- Structures can be assigned even if the structure contains an array. Is it True?
- If the structure contains pointer, how member-wise copy happens?

Parameter passing and return

Consider the Player structure.

```
    struct player
    {
        int id;
        char name[20];
    };
```

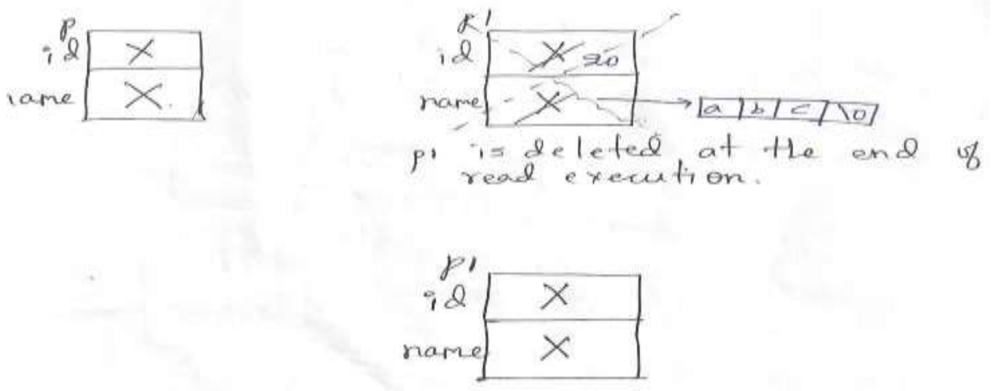
Coding Example_5: Functions to read and display the details of a player.

Version 1:

```
int main()
{       struct player p;
        printf("Enter id and name\n");
        read(p);        disp(p);
        return 0;
}
```

Parameter passing is always by value in C. So, copy of the argument is passed to the parameter p1. Whatever is modified, it is made to parameter p1.

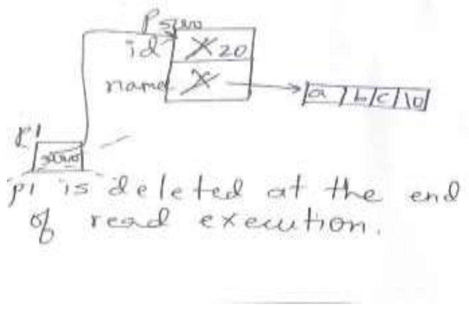
```
void read(struct player p1)
{
        scanf("%d ", &p1.id);           // user enters 20
        scanf("%[^\n]s", p1.name);
}
void disp(struct player p1)
{
        printf("%d\n",p1.id);                   // undefined values get printed
        printf("%s",p1.name);
}
```

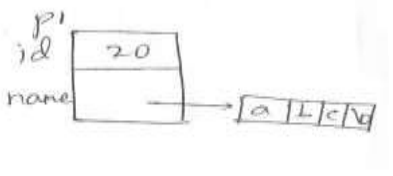


Version 2: If you want to change the structure, pass a pointer to a structure(l-value) as an argument.

```
int main()
{       struct player p;
        printf("Enter id and name\n");
        read(&p);       disp(p);                // passing &p to read()
        / / Think, can we also pass &p to disp(). Is there any harm?
        return 0;
}
```

```
void read(struct player *p1)
{
        scanf("%d ", &(p1->id));        // or &((*p1).id)       // user enters 20
        scanf("%[^\n]s", p1->name);                       // abc entered
}
void disp(struct player p1)
{
        printf("%d ",p1.id);            // actual values will be printed.
        printf("%s",p1.name);
}
```



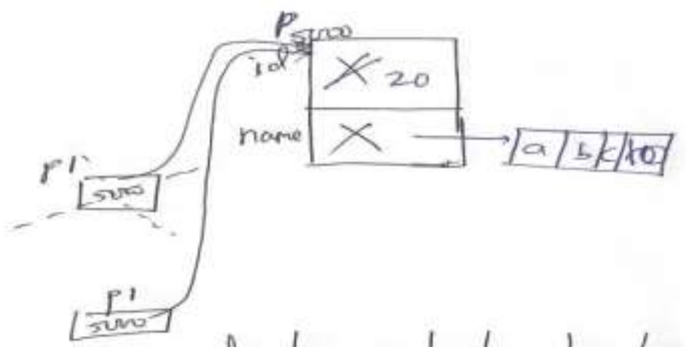


Version 3:

Assume the player structure contains many data members. In this case, can we just use structure variable to disp function? If we do so, this is considered as a bad practice. As every member of argument is copied to every member of parameter, it requires more space and more time to copy.

Also, when we are very sure that **we do not want to make any changes to the argument, parameter can be a pointer to constant structure**

```
void disp(const struct player* p1)

{       // p1.id = 34;        // throws error as p1 is a pointer to constant structure
        printf("%d ",p1->id);           // actual values will be printed.
        printf("%s",p1->name);
}
```



p1 is a pointer to constant structure.

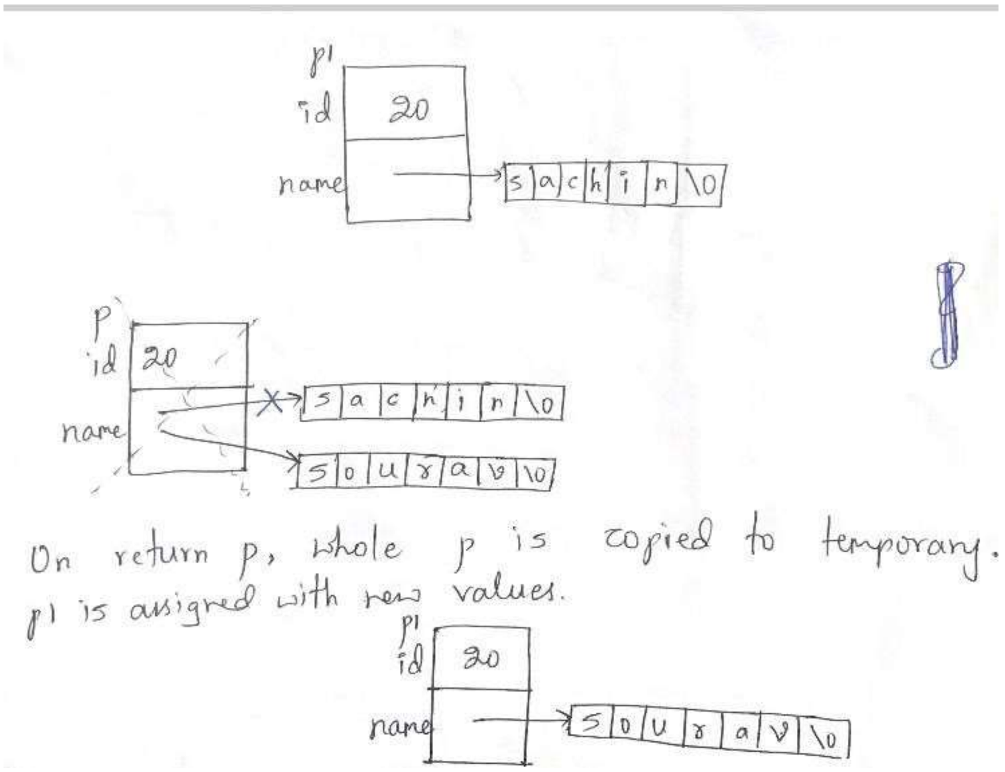
Can we return a structure variable? Yes.

Coding Example_6:

```
struct player
{
        int id;
        char name[20];
};
struct player modify(struct player);
int main()
{
        struct player p1={20,"sachin"};
        printf("before change %s",p1.name);
        p1=modify(p1);
        printf("after change %s",p1.name);
        return 0;
}
```

When the function modify() returns a changed parameter by value, it is copied to a temporary. Then in the client code, the temporary is assigned to p1.

```
struct player modify(struct player p)
{           strcpy(p.name,"Sourav");        return p;               }
```



Think about these points:

- Can we make const struct player p as the parameter for read?
- Can we say struct player *p as the parameter without changing the client code?
- If we change the client code, can we use struct player *p as the parameter?
- What changes must be done in modify() to copy Sourav to name
- Can we return pointer to structure?

Usage of typedef

- typedef is a keyword which is used to give a type, a new name. By convention, **uppercase** letters are **used** for these **definitions** to **remind** the **programmer** that the **type name** is really a **symbolic abbreviation**, but we can use **lowercase** as well
- Used to assign alternative names to existing data types. It is used to provide an interface for the client. Once a new name is created using typedef, the client may use this without knowing the underlying type.
- typedef is limited to giving symbolic names to types only where as #define can be used to

define values.

- typedef interpretation is performed by compiler where as #define statements are processed by the preprocessor.

I want to store the age of a person. Best way to define a variable is int age = 80; Alternatively, you can also say, typedef int age; //another name for int is age.

Then age a = 80;

Can we say age age = 80?   // Think about this.

If we know how to declare a variable, then prefixing the declaration with typedef makes that a new name.

```
int b[10];                  // b is an array variable to store 10 integers
typedef int c[10];          // c is a name for an array of 10 integers
```

typedef is mostly used with user defined data types when names of the data types become slightly longer and complicated to use in programs.

Coding Example_7:

Version 1:

```
struct player
{       int id;

        char name[20];
};
typedef struct player player_t;     // player_t is a new name to struct player.
int main()
{
         player_t  p1;                          // p1 is a variable

}
```

Version 2: You can include typedef when defining a new type itself.

```
typedef struct player
{
        int id;
        char name[20];
}player_t;              // player_t is a new name to struct player.
int main()
{
       player_t  p1;                              // p1 is a variable
}
```

Nested Structures

Structure written inside another structure is called as nesting of two structures. It can be done in two ways.

Coding Exampe_8:

1. **Separate structures:** Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct dob
{
      int date;
      int month;
      int year;
};
struct student              // template declaration which describes how student entity looks like
{
      int roll_no;            // these three are members of the structure
      char name[20];
      int marks;
      struct dob d1;                //using structure variable of dob inside student
 };
```

It can be **understood from example that dob (date of birth) is the structure with members as**

date, month and year. Dob is then used as a member in Student structure.

2. **Embedded structure:** It enables us to declare the structure inside the structure.

```
struct student          // template declaration which describes how student entity looks like
{
    int roll_no;         // these three are members of the structure
    char name[20];
    int marks;
    struct dob          //declaring structure dob inside student
    {
        int date;
        int month;
        int year;
    }d1;
};
```

Accessing Nested Structure Members: We can access the member of the nested structure by using **Outer_Structure.Nested_Structure.member** as given below:

```
int main()
{
    struct student s1 = {24, "John", 78, {20, 03, 2000}};   //Declaration and Initialization
    printf("\n Roll no. = %d, Name= %s, Marks = %d, dob = %d.%d.%d ", s1.roll_no,
    s1.name,s1.marks, s1.d1.date,s1.d1.month, s1.d1.year);
    return  0;

}
```

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Problem Solving - Structures

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Let us solve few problems related to structures in C. Here is the list.

- Create a Structure called hotel with following members c_name, No_days, phone number as nested structure with its members as mob and alternate number. Write a function to read details from the user. Also calculate the bill (per day charges is 2000 and 8.5%tax on total bill) and Display appropriate message.

- Create a structure called Book with members as Book_name, Author_name, Edition, Publication, year and price. Show the menu to the user to do the following: Read book data, Display the book data if the user enters the book name. Display appropriate message.

- Write a program to create a structure called weather_Report with data members rainfall, temperature and city name. Create a function to read and display the details of 3 cities. Also write a function to display the city name with maximum temperature.

- Create a user defined data type for date. Write a function which will increment the date by 1

Let us solve the first one.

Coding Example_1:

```
#include<stdio.h>
struct Hotel
{
        char c_name[10];
        int no_days;
        struct phone
        {
                int mob;
                int alt_num;
   } p;
};
struct Hotel read_details(struct Hotel h1);
```

```
void calculate_Bill(struct Hotel *h1);



int main()
{
        struct Hotel h,h1;
        h1=read_details(h);
        calculate_Bill(&h1);
        return 0;
}

struct Hotel read_details(struct Hotel h1)
// can also pass the pointer and in that case, no return is required
{
        printf("enter name");
        scanf("%[^\n]s",h1.c_name);
        printf("enter no of days");
        scanf("%d",&h1.no_days);
        printf("enter mobile number");
        scanf("%d",&h1.p.mob);
        printf("enter alternate number");
       scanf("%d",&h1.p.alt_num);
       return h1;
}



void calculate_Bill(struct Hotel *h1)
{
        int room_charges=2000;
        float tax_per=8.5;
        printf("\nNo of days %d\n",h1->no_days);
        float amount=h1->no_days*room_charges;
        float total=amount+ (amount*(8.5/100));
        printf("\n\nThe amount is %f",total);
        printf("\n\nVisit again,Thank you");
}
```

Few points to think!!

Can you pass h1 to Calculate_Bill? If yes, what changes, what changes to be done to the parameter?

Can you make the parameter of read_details as const?

Can you make the parameter of Calculate_Bill as const?

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Problem Solving – Array of structures

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Let us consider solving the below problem.

Coding Example_1:

Create a structure which holds various attributes (e.g. name, id, basic_salary, DA%, HRA%, total_salary etc.) of an employee. Write a program which allows you to scan these (except total_salary) attributes for n employees.

The program should support following operations:

i. calculate_total_salary (total salary of the selected employee)

ii. Max (find and display name of the employee with maximum basic salary)

Solution:

Client code provided here and I do not want to change the client at any cost. Adding the source files and header files as per this client only.

```
// employee_client.c
#include<stdio.h>
#include "employee.h"
int main()
{
    employee_t e[1000];
    printf("Enter the number of employees for which the details has to be entered\n");
    int n, id;
    scanf("%d", &n);
    printf("enter name, id and basic salary of %d employees\n",n);
    scan_details(e,n);
    printf("enter the id of the employee whose total salary you want to know\n");
    scanf("%d",&id);
    int tot_salary = calculate_total_salary(e,n,id);
    if (!tot_salary)
      printf("employee with entered id doesnt exist\n");
    else
      printf("total salary of employee with id %d is %d\n",id,tot_salary);
```

```
        printf("Employee details entered is as below\n");
        display_details(e,n);
        employee_t e_max = max(e,n);
        printf("employee with maximum basic salary is %s",e_max.name);
        return 0;
        }
```

Header file: employee.h

```
typedef struct employee
{
        char name[20];
        int id;
        int basic_salary;
        int total_salary;
}employee_t;
// You might want to have DA and HRA as data members. Try that. But make sure client is not
        touched
void scan_details(employee_t*, int);
void display_details(const employee_t*, int);
int calculate_total_salary(employee_t*, int,int);
employee_t max(employee_t*, int);
```

Implementations of these functions are available in **source file employee.c** as below.

```
#include"employee.h"
#include<stdio.h>
void scan_details(employee_t* e, int n)
{
        int i;
        for(i = 0;i<n;i++)
          scanf("%s %d %d",(e+i)->name,&(e+i)->id, &(e+i)->basic_salary);
}
```

```
void display_details(const employee_t* e, int n)
{
        int i;
        for(i = 0;i<n;i++)
          printf("%s\t%d\t%d\n",(e+i)->name,(e+i)->id, (e+i)->basic_salary);
}

int calculate_total_salary(employee_t* e, int n,int id)
{
        int i;
        int da;
        int hra;
        int t_salary = 0;
        for(i = 0;i<n;i++)
        {
          if((e+i)->id == id)
          {
                  da = 0.8*(e+i)->basic_salary;
                  hra = 0.2*(e+i)->basic_salary;
                  (e+i)->total_salary = da+hra+(e+i)->basic_salary;
                  t_salary = (e+i)->total_salary;
          }
        }
        return t_salary;
}
employee_t max(employee_t* e, int n)
{       int i;
        int max = 0;
        employee_t e_max;
        for(i = 0;i<n;i++)
        {
            if((e+i)->basic_salary > max)
            {
                max = (e+i)->basic_salary;
                e_max = *(e+i);
            }
        }
        return e_max;
    }
```

Coding Exampe_2: Create a structure to store details of Car like company name, model and year. Add functions to read and display the details of n cars. Also include functions to display only the model and company name of the car in the specific year or previous to it. Display appropriate message.

Solution:

```
#include<stdio.h>
struct car
{
  char company[20];
  char model[30];
  int year;
};

void display_year(struct car *arr_car,int n);
void read(struct car *arr_car,int n);
void display(struct car *arr_car,int n);
int main()
{
struct car arr_car[100] ;
int n;
printf("Enter how many car details you want to store\n");
scanf("%d",&n);
read(arr_car,n);
```

```
display(arr_car,n);
display_year(arr_car,n);
return 0;
}

void read(struct car *arr_car,int n)
{

   printf("Enter the car details:company name,model,year\n");
   for(int i=0;i<n;i++)
   {
     scanf("%s",arr_car[i].company);
     scanf("%s",arr_car[i].model);
     scanf("%d",&arr_car[i].year);
   }
}
void display(struct car *arr_car,int n)
{
   printf("The details of car  are:\n");

   for(int i=0;i<n;i++)
    {
      printf("make:%s\t",arr_car[i].company);
      printf("model:%s\t",arr_car[i].model);
      printf("year:%d\t",arr_car[i].year);
      printf("\n");
    }
}

void display_year(struct car *arr_car,int n)
{
  int yy;
  printf("Enter the year to display car details\n");
  scanf("%d",&yy);
  for(int i=0;i<n;i++)
  {
    if(arr_car[i].year==yy)
     {
        printf("Model:%s",arr_car[i].model);
        printf("Company:%s",arr_car[i].company);
     }
  }
}
```

Coding Example_3: Create a structure to store information about property like property ID, owner name, age, property tax and Annual income. Read the details of n properties. If the annual income is less than 5 lakhs, he/she is eligible to avail discount of 20% on property tax amount. Display the tax amount to be paid after discount using a separate function.

Solution:

```
#include <stdio.h>
struct info
{
    int ID;
    char name[30];
    int age;
    int tax;
    int income;
};
void read(struct info *s,int n);
void calculate_tax(struct info *s,int n);

int main()
```

```
{
   int n;
   struct info s[100];
   printf("Enter the number of entries ");
   scanf("%d",&n);
   read(s,n);
   calculate_tax(s,n);
return 0;
}

void read(struct info *s,int n)
{
for(int i=0;i<n;i++)
   {
      printf("Enter ID \n");
      scanf("%d",&s[i].ID);
      printf("Enter name\n");
      scanf("%s",&s[i].name);
      printf("Enter age\n");
      scanf("%d",&s[i].age);
      printf("Enter property tax amount \n");
      scanf("%d",&s[i].tax);
      printf("Enter annual income\n");
      scanf("%d",&s[i].income);
   }
}
void calculate_tax(struct info *s,int n)
{
  for(int j=0;j<n;j++)
   {
      if(s[j].income<500000)
```

```
        {
          s[j].tax=0.8*s[j].tax;
        }
        printf("%s should pay tax of %d\n",s[j].name,s[j].tax);
    }
}
```

Few more programs for you to solve are listed here.

- Program to read and display the details of n books using functions. Clearly separate interface and implementation.
- Program to display all information about a student who has scored highest marks in a class. Consider inputs: 5 students (info.- name, roll_no, dob, marks scored in 5 different subjects)
- Program to store the details of 5 flights. Also display all flights information sorted in order of their arrival times.
- Create a structure to maintain the details of bank account(Account number ,Name, type Balance) and display insufficient fund if the balance is less than or equal to zero.
- Create the structure to store the details of employees with details: (Emp_Id,Emp_name,Emp_phone,Emp_rewardpoint). Read the details of n employees and if the employee has got rewardpoints more than 1000, then display a message that "Gift voucher has been mailed " .

Happy Coding..!!

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Array of Pointers to structures

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviours.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Introduction

There are times when an array of pointers to structures is useful. For example, suppose the structure had 100 elements, that one of the elements was a person's surname and one wanted to sort the structure in alphabetical order of surnames. The sorting technique involves a lot of swapping and moving structure around. This is slower than moving pointers around because pointers occupy less space. So to speed up the sorting process and efficiency factor, array of pointers is used for the structure rather than the whole structure or record being accessed.

Let's understand the below sections in detail:

Array of Pointers

Pointer to Structure

Array of Pointers to Structure

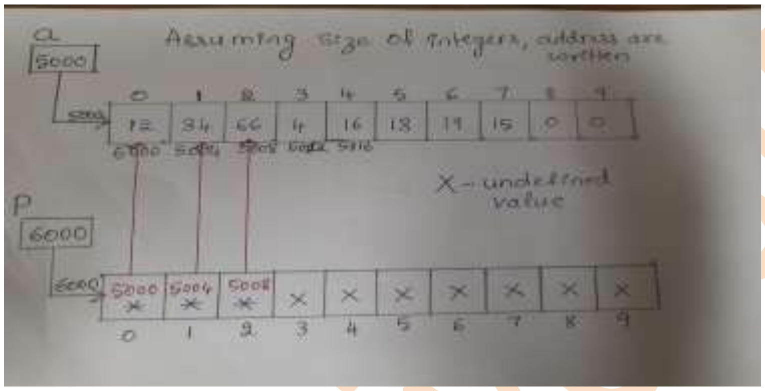
Array of Pointers

An array of pointers is an indexed set of variables in which the variables are pointers. Array and pointers are closely related to each other. The name of an array is considered as a pointer, i.e., the name of an array contains the address of an element.

Coding Example_1:

```
int i;
int a[10] = {12,34,66,4,16,18,19,15};
int *p[10]; // p is created with the intention of storing an array of addresses
p[0] = &a[0];
p[1] = &a[1];
p[2] = &a[2];
for(i = 0 ; i<4;i++)
{
        printf("%p  %p\n",p[i],&a[i]); // same address for all 3 pairs except the last
iteration as we have not    assigned.
}
```

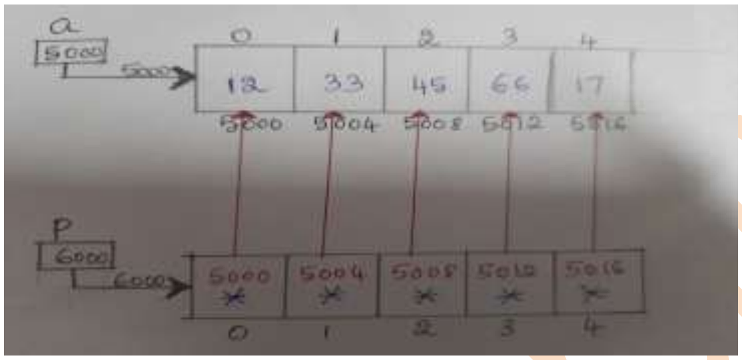
```
// printing array elements using p
for(i = 0 ; i<4;i++)
{
        printf("%d\t",*p[i]); // dereference every element of the array of pointer
}
```



Coding Example_2: Program to print the elements of the array using array of pointers

```
#include<stdio.h>
int main()
{               int a[5] = {12,33,45,66,17}; // a is an array
                printf("Using original array\n");
                for(int i = 0;i<5;i++)
                {       printf("%d ",a[i]);        }
                printf("\n");
                int *p[5]; // p is an array of pointers.
                for(int i = 0;i<5;i++)
                {     p[i] = &a[i]; // address of a[i] stored in p[i]      }
                printf("Using array of pointers\n");
                for(int i = 0;i<5;i++)
                {
                        printf("%d ",*p[i]);
                        // content at p[i] is displayed. All elements are displayed
```

```
        }
        printf("\n");
        return 0;
}
```



Pointers to Structures

A pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, a pointer to structure is a variable that can point to the address of a structure variable .Declaring a pointer to structure is same as pointer to variable:

Syntax: struct tagname *ptr;

There are two ways of accessing members of structure using pointer:

1. Using indirection (*) operator and dot (.) operator.
2. Using arrow (->) operator or membership operator.

Array of Pointers to Structures

An array of pointers to structure is also a variable that contains the address of structure variables. In order to create array of pointers for structures three declarations are defined. The first step in array of pointers for Structure is creation of an array of structure variable

Syntax: struct tag name array-variable[size];

The final step is to create an array of pointers with size specified to hold the addresses of structures in the array of structure variable.

Syntax: struct tag name *pointer_variable[size];

Coding Example_3: Printing the array of structures using array of pointers to structures

```
typedef struct Sample
{
        int a; float b;
}SAMPLE;
#include<stdio.h>
//void swap(SAMPLE a, SAMPLE b);
void swap(SAMPLE* a, SAMPLE *b);
int main()
{
        SAMPLE s[] = {{2,2.2},{1,1.1},{7,7.7},{4,4.4},{3,3.3}};
        SAMPLE *sp[1000];
        int n = sizeof(s)/sizeof(*s);
        printf("using array\n");
        for(int i = 0; i<n ; i++)
        {
                printf("%d %f\n",s[i].a, s[i].b);
        }
        printf("\nusing array of pointers\n");
        for(int i = 0; i < n;i++)
        {
                sp[i] = &s[i];
        }
        for(int i = 0; i<n ; i++)
        {
                printf("%d %f\n",sp[i]->a, sp[i]->b);
        }
```

Note: Write diagrams to understand better

Coding Example_4: Program to swap first and last elements of the array of structures and display the array of structures using array of structures and array of pointers

```
typedef struct Sample
{
        int a; float b;
}SAMPLE;
#include<stdio.h>
void disp1(SAMPLE *s, int n);  // s[ ]
void disp2(SAMPLE **sp, int n);  // *sp[ ]
void swap(SAMPLE **a, SAMPLE **b);
int main()
{
        SAMPLE s[] = {{2,2.2},{1,1.1},{7,7.7},{4,4.4},{3,3.3}};
        SAMPLE *sp[100];
        int n = sizeof(s)/sizeof(*s);
        printf("before swap using array of structures\n");
        disp1(s,n);
        for(int i = 0; i < n;i++)
        {       sp[i] = &s[i];          }
        printf("before swap using array of pointers to structures\n");
        disp2(sp,n);
        //swap(&s[0],&s[n-1]);  // sp[0], sp[n-1]
        swap(&sp[0],&sp[n-1]);  // 2000,   2004
        printf("after swap using array of structures\n");
        disp1(s,n); // no change in the original set
        printf("after swap using array of pointers to structures\n");
        disp2(sp,n);  // first and last structures are swapped.
        return 0;
}
```

```
void swap(SAMPLE **a, SAMPLE **b)
{
        SAMPLE *temp =  *a;
        //Sample *temp;
        //temp = *a;
        *a = *b;
        *b = temp;
}
void disp1(SAMPLE *s, int n)  // SAMPLE s[ ]
{
        for(int i = 0; i < n;i++)
                {
                        printf("%d %f\n",s[i].a,s[i].b);
                }
}
void disp2(SAMPLE **sp, int n)
{
        for(int i = 0; i < n;i++)
                {
                        printf("%d %f\n",sp[i]->a,sp[i]->b);
                }
}
```

Few points to think!!

- Swapping using array of pointers is only swapping the pointers and not the array of structures. Write the diagram for the swap code to understand better.
- What is the significance of using Array of pointers to structures?- To avoid memberwise copy while swapping. If the structure has too many data members, this array of pointer makes sense.

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Sorting

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine.

CObj2: Map algorithmic solutions to relevant features of C programming language constructs.

CObj3: Gain knowledge about C constructs and its associated eco-system.

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors.

CObj5: Get insights about testing and debugging C Programs.

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs.

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions.

CO3: Understand prioritized scheduling and implement the same using C structures.

CO4: Understand and apply sorting techniques using advanced C constructs.

CO5: Understand and evaluate portable programming techniques using pre-processor directives and conditional compilation of C Programs.

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

Introduction

There are so many things in our real life that we need to search for, like a particular record in the database, roll numbers in the merit list, a particular telephone number in a telephone directory, a particular page in a book etc. The concept of sorting came into existence, making it easier for everyone to arrange data in order to make the searching process easy and efficient. **Arranging the data in ascending or descending order is known as sorting.**

Why Sorting?

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. Sometimes in real world applications, it is necessary to arrange the data in a sorted order to make the searching process easy and efficient.

Example: Candidates selection process for an Interview- Arranging candidates for the interview involves sorting process (sorting based on qualification, Experience, skills or age etc.

Sorting Algorithms

Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. Depending on the data set, Sorting algorithms exhibit different time and space complexity.

Following are samples of sorting algorithms.

1. **Bubble Sort -- Dealt in detail**
2. Insertion Sort
3. Quick Sort
4. Merge Sort
5. Radix Sort
6. Selection Sort
7. Heap Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is suitable for small and medium data sets.

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.

Coding Example_1: Sorting array of integers using Bubble sort (dealt in unit 2)

```
#include <stdio.h>

// A function to implement bubble sort
void bubbleSort(int *p, int n)
{
  int i, j,temp;
  for (i = 0; i < n-1; i++)

     // Last i elements are already in place
     for (j = 0; j < n-i-1; j++)
        if (*(p+j) > *(p+j+1))
           {
                 temp=*(p+j);
                 *(p+j)=*(p+j+1);
                 *(p+j+1)=temp;
                            }
}
```

```
/* Function to print an array */
void printArray(int *p, int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", *(p+i));

}
// Driver program to test above functions
int main(void)
{
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr)/sizeof(arr[0]);
   bubbleSort(arr, n);
   printf("Sorted array:  ");
   printArray(arr, n);
   return 0;
}
```

Coding Example _2 :Using array of structures, sort the structures in the array based on the data member – roll_no

```
#include<stdio.h>
typedef struct Student
{
         int marks;
         char name[100];
         int roll_no;
         float cgpa;
} STUDENT;
```

```
void input_data(STUDENT *s, int n);
void display_data(STUDENT *s, int n);
void sort(STUDENT *s, int n);

int main()
{
        STUDENT s[1000];
        int n;
        printf("Enter the number of students :");
        scanf("%d", &n);
        input_data(s,n);
        printf("Entered details before sorting:  \n");
        display_data(s,n);
        sort(s,n);
        printf("After sorting, the sorted list printed using the array of structures:\n\n");
        display_data(s,n);
}

void input_data(STUDENT *s, int n)
{
        int i;
        for(i = 0 ; i < n; i++,s++) //to be used when the last scanf is used
        //for(  i = 0 ; i < n; i++) //To be used when we use any of first four scanf
        {
                printf("Enter the details as marks,name,roll_no,cgpa: \n");
                // The following are the different variations which can be demonstrated.
                //scanf("%d%s%d%f", &(s[i].marks), s[i].name, &(s[i].roll_no), &(s[i].cgpa));
                //scanf("%d%s%d%f", &((*(s+i)).marks), (*(s+i)).name, &(s[i].roll_no),
&(s[i].cgpa));
```

```
                    //scanf("%d%s%d%f", &((s+i)->marks), (s+i)->name, &(s[i].roll_no), &(s[i].cgpa));
                     scanf("%d%s%d%f", &(s->marks), s->name, &(s->roll_no), &(s->cgpa));

          }
}

void sort(STUDENT *s, int n)
{
          int i,j;
          for( i = 0 ; i < n ; i++)
          {
                    for( j = 1; j < n-i; j++)
                    {
                              if(s[j-1].roll_no > s[j].roll_no)
                              {
                                        STUDENT t;
                                        t = s[j-1];
                                        s[j-1] = s[j];
                                        s[j] = t;
                              }
                    }
          }
}


void display_data(STUDENT *s, int n)
{
          int i;
          for( i = 0 ; i < n; i++)
          {
```

```
		printf("%d\t%s\t%d\t%0.2f\n", (s[i].marks), s[i].name, (s[i].roll_no), (s[i].cgpa));
	}
}
```

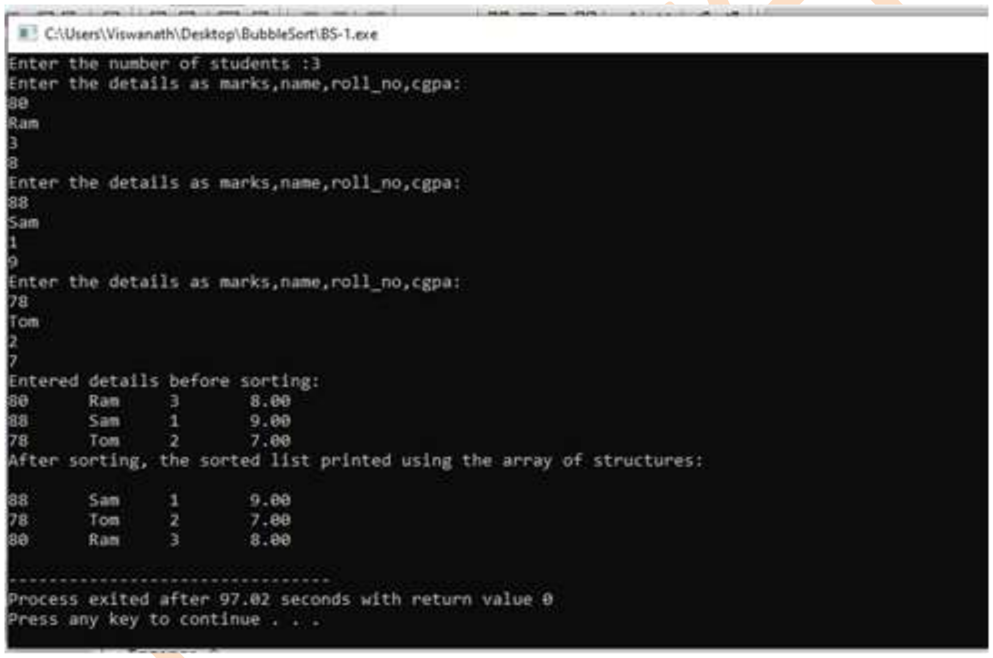
Coding Example_3: Using array of pointers to structures, sort the structures in the array based on the data member – roll_no , without affecting the array of structures

```
#include<stdio.h>
typedef struct Student
{
	int marks;
	char name[100];
	int roll_no;
	float cgpa;
} STUDENT;

void input_data(STUDENT *s, int n);
void sort(STUDENT **, int n);
void init_structures_pointers(struct Student **sp,struct Student *s,int n);
void display_data_pointers(STUDENT **sp,int n);
void display_data(STUDENT *s, int n);
int main()
{
	STUDENT s[1000];
	int n;
	printf("Enter the number of students: ");
	scanf("%d", &n);
	input_data(s,n);
```

```
        struct Student* sp[n];
        init_structures_pointers(sp,s,n);
        sort(sp,n);
        printf("After sorting,printing the list using array of structures\n");
        display_data(s,n); // Observe that there is no change in the array of structures.
        printf("After sorting,printing the list using array of pointers\n");
        display_data_pointers(sp,n);  // observe the sorted list printed using pointers


        return 0;
}
void init_structures_pointers(struct Student **sp,struct Student *s,int n)
{
        int i;
        for( i = 0 ; i < n; i++)
        {
                sp[i] = &s[i]; //(s+i)
        }
}

void sort(struct Student *sp[],int n)
{
        int i,j;
        for( i = 0 ; i < n ; i++)
        {

                for( j = 1; j < n-i; j++)
                {
                        if(sp[j-1]->roll_no > sp[j]->roll_no)
                        {
                                STUDENT *t;
                                t = sp[j-1];
                                sp[j-1] = sp[j];
                                sp[j] = t;
                        }
                }
        }
}
void input_data(STUDENT *s, int n)
{
        int i;

        for(  i = 0 ; i < n; i++)
        {
                printf("Enter the details as marks,name,roll_no,cgpa \n");
                scanf("%d%s%d%f", &((s+i)->marks), (s+i)->name, &(s[i].roll_no), &(s[i].cgpa));

        }
}

void display_data_pointers(STUDENT **sp,int n)
{
        int i;
        for( i = 0; i < n ; i++)
        {
                printf("%d\t%s\t%d\t%0.2f\n", sp[i]->marks, sp[i]->name,(*(sp[i])).roll_no, sp[i]-
>cgpa);
        }
}
```

```
void display_data(STUDENT *s, int n)
{
        int i;
        for( i = 0 ; i < n; i++)
        {

                printf("%d\t%s\t%d\t%0.2f\n", (s[i].marks), s[i].name, (s[i].roll_no), (s[i].cgpa));
        }
}
```

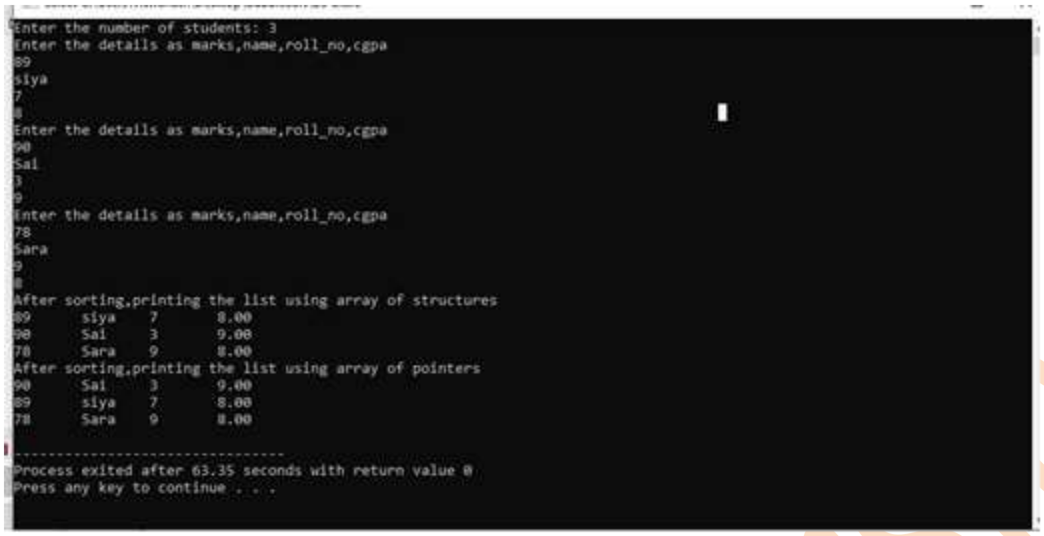
Coding Example_2 Execution Output



C:\Users\Viswanath\Desktop\BubbleSort\BS-1.exe

```
Enter the number of students :3
Enter the details as marks,name,roll_no,cgpa:
80
Ram
3
8
Enter the details as marks,name,roll_no,cgpa:
88
Sam
1
9
Enter the details as marks,name,roll_no,cgpa:
78
Tom
2
7
Entered details before sorting:
80      Ram     3       8.00
88      Sam     1       9.00
78      Tom     2       7.00
After sorting, the sorted list printed using the array of structures:

88      Sam     1       9.00
78      Tom     2       7.00
80      Ram     3       8.00

--------------------------------
Process exited after 97.02 seconds with return value 0
Press any key to continue . . .
```

Coding Example_3 Execution Output



```
Enter the number of students: 3
Enter the details as marks,name,roll_no,cgpa
89
siya
7
8
Enter the details as marks,name,roll_no,cgpa
90
Sai
3
9
Enter the details as marks,name,roll_no,cgpa
78
Sara
9
8
After sorting,printing the list using array of structures
89      siya    7       8.00
90      Sai     3       9.00
78      Sara    9       8.00
After sorting,printing the list using array of pointers
90      Sai     3       9.00
89      siya    7       8.00
78      Sara    9       8.00

--------------------------------
Process exited after 63.35 seconds with return value 0
Press any key to continue . . .
```

More points to think!

- What changes to added to sort function to sort in descending order?
- If there is one more field in the structure, if we want to sort the array of structure base on that field, should we write one more bubble sort function inside which the comparisons of the new field is done before swapping the structures?
- Is there any simpler solution to avoid the repetitions of these functions with only small change in the swap condition? – Callback. We will discuss in further lecture note**s**

Happy Sorting using array of Pointers!!

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Linked List

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC

Jan-May, 2022

Dept. of CSE

PES University

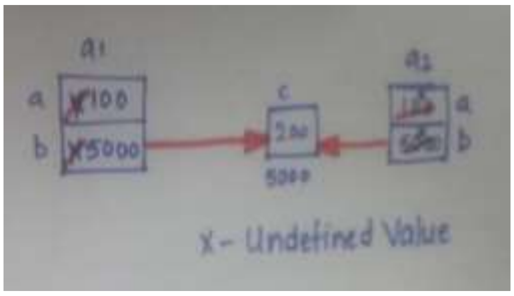
Introduction

Linked List can be defined as collection **of objects called nodes that are randomly stored and logically connected in the memory via links.** The node and link has a special meaning which we will be discussing in this chapter. Before we deal with this in detail, we need to answer few questions.

Q1. Can we have a pointer data member inside a structure? Yes. Refer to below codes.

Coding Example_1:

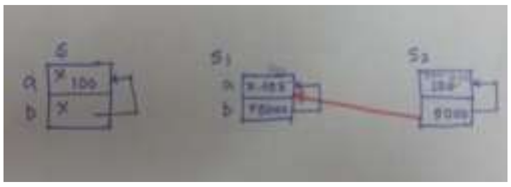
Version 1:



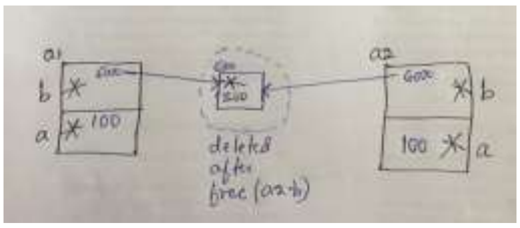
```
#include<stdio.h>
struct A
{
        int a;
        int *b;
};
int main()
{
        struct A a1;
        struct A a2;
        a1.a = 100;
        int c = 200;
        a1.b = &c;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b));        // 100 200
        a2 = a1;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b));        // 100 200
        a1.a = 300;
        *(a1.b) = 400;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b));        // 100 400
        printf("a1 values:%d and %d\n", a1.a, *(a1.b));        // 300 400

        struct A *p=&a1;
        printf("a1 values:%d and %d\n", p->a,*(p->b));         // 300 400
        return 0;
}
```

Version 2:

```
struct Sample {
        int a;     int *b;
};
#include<stdio.h>
int main() {
      struct Sample s;
      s.a = 100;
      s.b = &(s.a);
      printf("%d %d",s.a,*(s.b));
      struct Sample s1;
      s1.a = 100;
      s1.b = &(s1.a);
      printf("%d %d\n",s1.a,*(s1.b));
      struct Sample s2 = s1;
      printf("%p %p\n",s1.b,s2.b);
      printf("%d %d\n",s2.a,*(s2.b));
      s2.a = 200;
      printf("%p %p\n",s1.b,s2.b);
      printf("%d %d\n",s1.a,*(s1.b));
      printf("%d %d\n",s2.a,*(s2.b)); // very imp
      *(s2.b) = 300;
      printf("%p %p\n",s1.b,s2.b);
      printf("%d %d\n",s1.a,*(s1.b));
      printf("%d %d\n",s2.a,*(s2.b));
      s2.b = &(s2.a);
      *(s2.b) = 400;
      printf("%p %p\n",s1.b,s2.b);
      printf("%d %d\n",s1.a,*(s1.b));
      printf("%d %d\n",s2.a,*(s2.b));
      return 0;
}
```

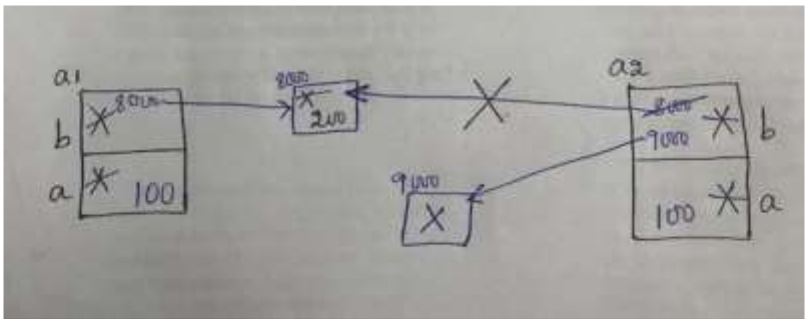


```
#include<stdio.h>
#include<stdlib.h>
struct A {
        int   a;
        int *b;
};
```



Version 3:

```
int main()
{
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        a2 = a1;
        printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 200
        free(a2.b); // a1.b too becomes dangling pointer
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 undefined behaviour
        return 0;
}
```



Version 4:

```
int main() {
        struct A a1; struct A a2; a1.a = 100;
        a1.b = (int*) malloc(sizeof(int));
        *(a1.b) = 200;
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        a2 = a1;
        a2.b = (int*) malloc(sizeof(int));
        // changing this to a1.b creates garbage and output differs
        printf("a2 values:%d and %d\n", a2.a, *(a2.b)); // 100 junk value
        printf("a1 values:%d and %d\n", a1.a, *(a1.b)); // 100 200
        free(a1.b);     free(a2.b);     return 0;
}
```

Q2. Can we have a structure variable inside another structure? Yes. Refer to below code.

Coding Example_2:

```
#include<stdio.h>
struct A
{
        int a;
};

struct B
{
        int a;
        struct A a1;
}; // structure variable a1 as a data member in B

int main()
{
        printf("sizeof A %d\n",sizeof(struct A));
        printf("sizeof B %d\n",sizeof(struct B)); // more than A
        return 0;
}
```

Q3. Can we have a structure variable inside the same structure? – No.

Coding Example_3:

```
struct A {
        int a;
        struct A a1;
};
```

//Compile time Error: field a1 has incomplete type

// size of struct A we cannot find as the field a1 which itself is a structure of the same kind.

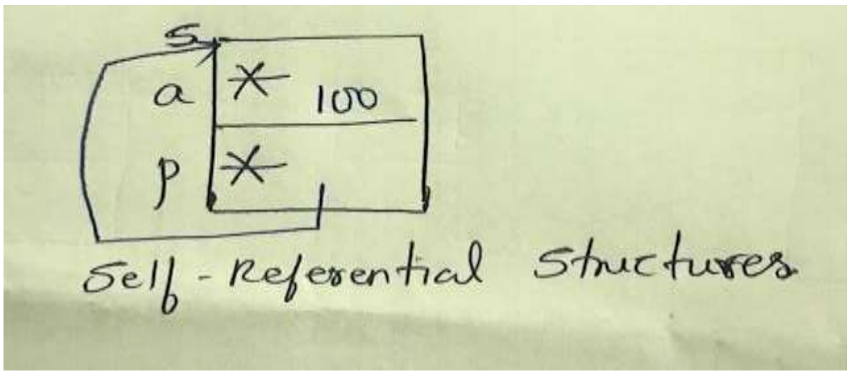
The solution to the previous version is to **have a pointer of the same type inside the structure**. The size of a pointer to any type is fixed. Hence, the size of the structure can be computed by the compiler at compile time.

Self Referential Structures

A structure which has a **pointer to itself as a data member** is called a self - referential structure. In this type of structure, the object of the same structure points to the same data structure and refers to the data types of the same structure. It can have one or more pointers pointing to the same type of structure as their member. It is **widely used in dynamic data structures such as trees, linked lists**, etc.

Coding Example_ 4:

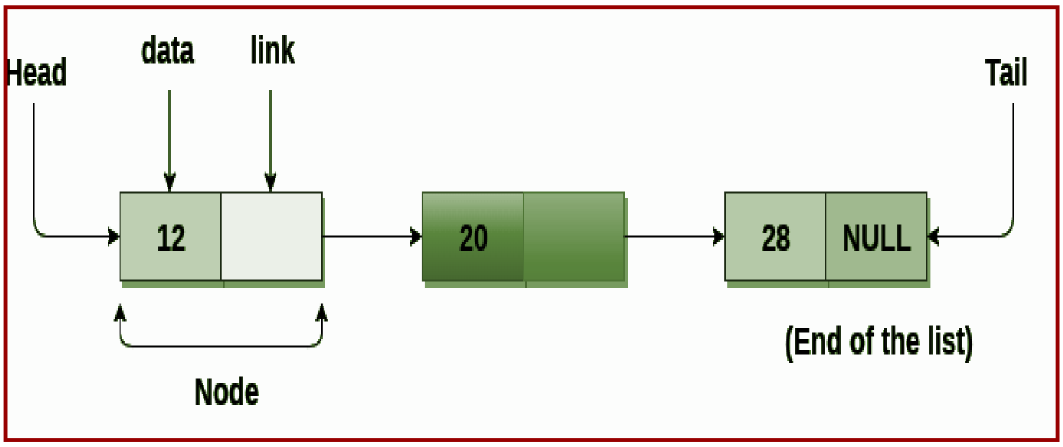
```
#include<stdio.h>
struct Sample
{
	int a;
	struct Sample *p;
};
int main()
{
	printf("%d\n",sizeof(struct Sample) ); // implementation specific
	struct Sample s;
	s.a = 100;
	s.p = &s;
	printf("%d %d %d\n", s.a, s.p->a, s.p->p->a); // all 100
	return 0;
}
```



Now let us get back to the Linked list and its implementation in detail.

Linked List

Linked List can be defined as collection of objects called **nodes that are randomly stored and logically connected in the memory via links**. A node **contains minimum two fields** - data stored at a particular address and the pointer which contains the address of the next node in the memory. The **last node of the list contains pointer to the null.** The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list, which is used for optimized utilization of space. **Size of the list is limited to the memory size and doesn't need to be declared in advance.**



Characteristics

1. A data structure that consists of zero or more nodes. Every node is composed of minimum two fields: a data/component field and a pointer field. The pointer field of every node point to the next node in the sequence.
2. We can access the nodes one after the other. There is no way to access the node directly as random access is not possible in a linked list. Lists have sequential access.
3. Insertion and deletion in a list at a given position requires no shifting of elements.

Differences between Arrays and Linked List

1. Array - a collection of same type data elements

   Linked list -a collection of unordered linked elements known as nodes.
2. Array- traversal through indexes.

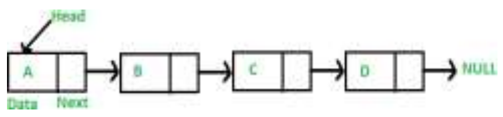
   Linked list - traversal through the head until we reach the last node.
3. Array - Elements are stored in contiguous address space

   Linked List - Elements are at random address spaces.

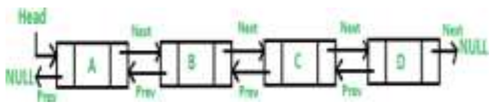
4. Array – Random access is faster.
   Linked List – Random access is slower.
5. Array- Insertion, and Deletion of an element is not that efficient.
   Linked List - Insertion and Deletion of an element is efficient.
6. Array - Fixed Size.
   Linked List – Dynamic Size
7. Array - Memory allocation decided during compile time/ static allocation.
   Linked List - Memory allocation decided during runtime / dynamic allocation.

Types of Linked List

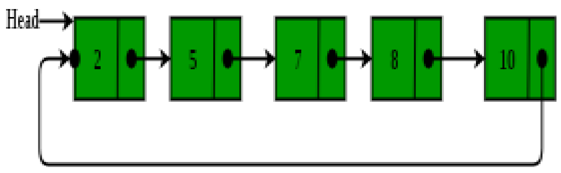
i. Singly Linked List



ii. Doubly Linked List



iii. Circular Linked List



Operations on Linked List

- **Insertion Operation on the list** includes Insertion at the beginning of the list, Insertion at the end of the list and Insertion at a Specific Node in the List
- **Deletion Operation on the list** includes Deletion at the beginning of the list, Deletion at the end of the list and Deletion of a Specific Node in the List
- **Other Operations on the list** such as Traversing the list, Searching the list and Sorting the List, etc,

Refer the below links to get a detailed information on Linked List types and its representation.

1. https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm#
2. https://www.geeksforgeeks.org/linked-list-set-1-introduction/
3. https://www.geeksforgeeks.org/doubly-linked-list/
4. https://www.geeksforgeeks.org/circular-linked-list/

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Linked List Implementation

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC

Jan-May, 2022

Dept. of CSE

PES University

//Implementation of Singly linked list
// InsertFront, DeleteFront, Display, DestroyList operations

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

typedef struct node
{
        int info;
        struct node *next;
}NODE;

NODE* insertFront(NODE* head,int ele);
void display(NODE *head);
NODE* deleteFront(NODE* head,int *pele);
NODE* freeList(NODE* head);

int main()
{
        NODE *head=NULL;
        int choice;
        int ele;

        do
        {
                printf("1.InsertFront 2.Display 3.DeleteFront\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:printf("Enter an integer\n");
                                        scanf("%d",&ele);
```

}

```
                    head=insertFront(head,ele);
                    break;
            case 2:display(head);
                    break;
            case 3: if(head!=NULL)
                    {
                        head=deleteFront(head,&ele);
                        printf("Deleted element is %d\n",ele);
                    }
                    else
                        printf("List is already empty\n");
                    break;
        }
    }while(choice<4);
    head=freeList(head);
}


NODE *createNode(int ele)
{
    NODE* newNode=malloc(sizeof(struct node));
//We assume memory is always allocated to the newnode and
//hence not checking for newNode==NULL
    newNode->info=ele;
    newNode->next=NULL;

    return newNode;
}
```

```
NODE* insertFront(NODE* head,int ele)
{
        NODE* newNode=createNode(ele);

        newNode->next=head;
        head=newNode;

        return head;
}
void display(NODE *head)
{
        if(head==NULL)
                printf("Empty List\n");
        else
        {
                NODE *p=head;
                while(p!=NULL)
                {
                        printf("%d ",p->info);
                        p=p->next;
                }
        }
}
```

```
NODE* deleteFront(NODE* head,int *pele)
{
        NODE *p=head;
        *pele=head->info;
        head=head->next;
        free(p);

        return head;
}

NODE* freeList(NODE* head)
{
        NODE *p=head;
        while(head!=NULL)
        {
                head=head->next;
//              printf("Freeing %d\n",p->info);
                free(p);
                p=head;
        }
        return head;
}
```

Unit : 3

Unit Name: Text Processing and User-Defined Types

Topic: Unions in C

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

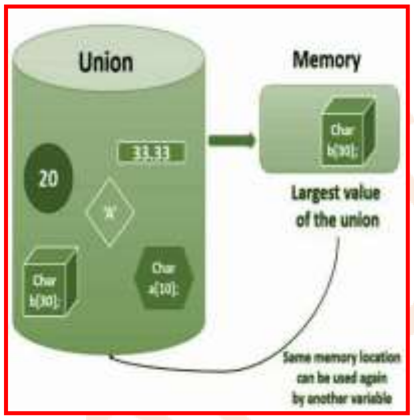
CO4: Understand and apply sorting techniques using advanced C contructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,
Jan - May, 2022
Dept. of CSE,
PES University

Introduction

A Union is a user-defined datatype in the C programming language. It is a collection of variables of different datatypes in the same memory location. We can define a union with many members, but at a given point of time, only one member can contain a value. Union unlike structures, share the same memory location. Using Union in C will save Memory Space in a given context.


Union
Memory
33.33
20
'A'
Char b[30];
Char a[10];
Char b[30];
Largest value of the union
Same memory location can be used again by another variable

Both structure and union are the custom data types that store different types of data together as a single entity. The members of structure and union can be entities of any type, such as other structures, unions, or arrays. Both structures or unions can be passed by value to a function and also return to the value by functions, the argument will need to have the same type as the function parameter.

Note: To access members, we use the '.' operator or '->' in a given context.

C unions are used to save memory. To better understand a union, think of it as a chunk of memory that is used to store variables of different types. When we want to assign a new value to a field, the existing data is replaced with new data. C **unions allow data members which are mutually exclusive to share the same memory**. This is quite important when memory is valuable, such as in embedded systems. Unions are mostly used in embedded programming where direct access to memory is needed.

Defining a union:

You can define a union with many members, but **only one member can contain a value at any given time**. **The memory occupied by a union will be large enough to hold the largest member of the union. So, the size of a union is at least the size of the biggest component. At a given point in time, only one can exist. All the fields overlap and they have the same offset: 0.**

The format for defining new type using union is as below:

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

Example:

```
union car
{
        char name[10];
        float price;
};
```

Now, a variable of car type can store a floating-point number or a string, i.e., the same memory location is used to store multiple types of data**. In the above example, the data type will occupy at least 10 bytes of memory space. Because this is the maximum space that can be occupied by a character array (Plus padding bytes, if necessary).**

Coding Example_1: Displays the total memory size occupied by the union

```
#include <stdio.h>
union car
{
        char name[10];          //assuming, 1 byte for char
        float price;            // assuming, 4 bytes for float
};
int main( )
{
        union car c;
        printf( "Memory size occupied by data in bytes : %d\n", sizeof(union car));
        return 0;
}
```

Output: Memory size occupied by data in bytes : 12

Note: Output may vary depending on the number of bytes of padding

Accessing Union Members using union variable:

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Coding Example_2:

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

union car
{
        char name[10];
        float price;
};
```

```
 int main()
 {
        union car c;
        strcpy(c.name, "Benz");
        c.price=4000000.00;
        printf( "car name: %s\n", c.name);
        printf( "car price: %f\n", c.price);
        return 0;
 }
```

Output: When the above code is compiled and executed, the value of name gets corrupted because the final value assigned to the variable price has occupied the memory location, and this is the reason that the value of price member is printed well.

Coding Example_3: Prints all the members of union

```
#include <stdio.h>
#include<string.h>
union car
{
        char name[10];
        float price;
};
int main( )
{       union car c;
        strcpy( c.name, "Benz");
        printf( "car name: %s\n", c.name);
        c.price=4000000.00;
        printf( "car price: %f\n",c.price);
        return 0;
 }
```

Output:

```
        Car name: Benz
        Car price:4000000.000000
```

Coding Example_4: Size of union and accessing the union members

```
#include<stdio.h>

union X
{       int i;
        int j;
        double k;
};              // ; compulsory

struct Y
{       int i;
        int j;
        double k;
};

int main()
{
        printf("sizeof X is %lu\n",sizeof(union X));
        union X x1;
        x1.i = 23;
        x1.j = 23;
        x1.i = 49;
        printf("%d %d\n",x1.i, x1.j);
        printf("sizeof Y is %lu\n",sizeof(struct Y));
        return 0;
}
```

Output:

```
sizeof X is 8
49 49
sizeof Y is 16
```

Coding Example_5: Size of union and accessing the union members

```
#include<stdio.h>
union Z
{
        int  a;
        int b[3];
};
int main()
{
        union Z z;
        z.a = 23;
        printf("size of Z is %d bytes\n",sizeof(z));
        printf("%d %d %d\n",z.b[0], z.b[1],z.b[2]);
        return 0;
}
```

Output:

```
size of Z is 12 bytes
23 0 2179072            // 23   junk value  junk value
```

Coding Example_6: All the fields overlap and they have the same offset: 0 in union

```
#include<stdio.h>
#include<stddef.h> // offsetof function is declared in stddef.h
union A
{
        int x;
        int y;
        int z;
};
struct B
{
        int x;
        int y;
        int z;
 };
 int main()
 {
        // assumption int occupies four bytes
        printf("%lu\n",offsetof(union A,y)); // 0
        printf("%lu\n",offsetof(struct B,y));  // 4
        printf("%lu\n",offsetof(struct B,z));  // 8
}
```

Structure v/s Union:

Structure:

1. The keyword struct is used to define a structure
2. When a variable is associated with a structure, memory is allocated to each member of the structure. The size of the structure is greater than or equal to the sum of its members.
3. Each member within a structure is assigned unique storage area location.
4. In Structure, the address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.
5. In Structure, altering the value of a member will not affect other members of the structure.
6. All members can be accessed at a time.

Union:

1. The keyword union is used to define a union.
2. When a variable is associated with a union, memory is allocated by considering the size of the largest data member. So, the size of a union is at least equal to the size of its largest member.
3. Memory allocated for union is shared by individual members of union
4. For unions, the address is same for all the members of a union. This indicates that every member begins at the same offset.
5. Altering the value of any of the member will alter other member values.
6. In Unions, only one member can be accessed at a time.

Similarites between structures and unions:

1. Both are user-defined data types used to store data of same or different types as a single unit.
2. Their members can be objects of any type, including other structures, unions, or arrays. A member can also consist of a bit of a field.
3. Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must be of the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. **". "**operator is used for accessing members.

We know that the fields of a union will share memory, so in the main program we ask the user which data he/she would like to store, and depending on the user's choice, the appropriate field will be used. By this way, we can use the memory efficiently.

Coding Example_7: Structure containing the union

```
#include<stdio.h>
struct test
{       int i;
        union test2
        {
                char a[20];
                float k;
                int j;
        }u;
};
int main()
{
        printf("%lu\n",sizeof(struct test));      // 24
        union test2 t2;             // allowed
        t2.k=45.6;
        printf("k is %f\n",t2.k);
        struct test t;
        t.u.j=78;
        printf("j is %d\n",t.u.j);
        t.u.k=45.6;
        printf("k is %f\n",t.u.k);
        return 0;
}
```

Output:

```
24
k is 45.599998
j is 78
k is 45.599998
```

Coding Example_8: Usage of anonymous union inside a structure

```
#include<stdio.h>
struct student
{
        union
        {                                       //anonymous union (unnamed union)
                char name[10];
                int roll;
        };
         int mark;
 };
 int main()
  {
        struct student stud;
        char choice;
        printf("\n You can enter your name or roll number ");
        printf("\n Do you want to enter the name (y or n): ");
        scanf("%c",&choice);
        if(choice=='y'||choice=='Y')
        {       printf("\n Enter name: ");
                scanf("%s",stud.name);
                printf("\n Name:%s",stud.name);
        }
        else
        {       printf("\n Enter roll number");
                scanf("%d",&stud.roll);
                printf("\n Roll:%d",stud.roll);
        }
        printf("\n Enter marks");
        scanf("%d",&stud.mark);
        printf("\n Marks:%d",stud.mark);
        return 0;
}
```

Output:

```
You can enter your name or roll number
Do you want to enter the name (y or n) : y
Enter name: john
Name:john
Enter marks: 45
Marks:45
```

Coding Example_9: Usage of anonymous union inside a union

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
union test
```

```
{
        int i;
        union
        {
                char a[20];
                float k;
        };
};
int main()
{
        printf("%lu",sizeof(union test));       // 20
        union test t;
        t.i=78;                         // One member at a time from union
        printf("\ni is %d\n",t.i);
        strcpy(t.a,"sindhu");
        printf("a is %s\n",t.a);
        t.k=45.5;
        printf("k is %f",t.k);
        return 0;
}
```

Output:

```
20
i is 78
a is sindhu
k is 45.500000
```

Coding Example_10: Structure inside Union

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

```
{
        char name[30];
        int rollno;
        float percentage;
};
union details
{
        struct student s1;
};
int main()
{
        union details  set;
        printf("Enter details:");
        printf("\nEnter name : ");
        scanf("%s", set.s1.name);
        printf("\nEnter roll no : ");
        scanf("%d", &set.s1.rollno);
        printf("\nEnter percentage :");
        scanf("%f", &set.s1.percentage);
        printf("\nThe student details are : \n");
        printf("\nName : %s", set.s1.name);
        printf("\nRollno : %d", set.s1.rollno);
        printf("\nPercentage : %f", set.s1.percentage);
        return 0;
}
```

Output:

```
Enter details:
Enter name : sindhu
Enter roll no : 123
Enter percentage :67
```

```
The student details are :
Name : sindhu
Rollno : 123
Percentage : 67.000000
```

Happy Coding using Unions and Structures!!

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Bitfields

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C contructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,
Jan - May, 2022
Dept. of CSE,
PES University

Introduction

In programming terminology, a bit field is a **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. Bit fields are of great significance in C programming, because of the following reasons:

- Used to reduce memory consumption.
- Easy to implement.
- Provides flexibility to the code.

In C, we can specify the size (in bits) of the **structure and union members**. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range. The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit.

Declaration:

```
struct
{
        type [member_name] : width ;
};
```

type - An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.

member_name -The name of the bit-field.

width -The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Coding Example_1:

```
#include<stdio.h>
struct Status
{
      unsigned int bin1:1; // 1 bit is allocated for bin1. only two digits can be stored 0 &1.
```

```
          unsigned int bin2:1;
             // if it is signed int bin1:1   or int bin1:1,        one bit is used to represent the sign
};
int main()
{
           printf("Size of structure is %lu\n",sizeof(struct Status));     // 4 bytes
           struct Status s;
           printf("enter the number");
          //scanf("%d",&s.bin1);    // Error
           // We cannot access the address of bit - field. system is byte addressable.
           return 0;
  }
```

Coding Example_2: Demonstration of the maximum value that can be stored in bitfields. Results in warning. If you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as below. The age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so.

```
struct {
           unsigned int age : 3;
} Age;
#include <stdio.h>
#include <string.h>
struct
{
           unsigned int age : 3;
} Age;
int main( )
{
              Age.age = 4;
              printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
```

```
        printf( "Age.age : %d\n", Age.age );
        Age.age = 7;
        printf( "Age.age : %d\n", Age.age );
        Age.age = 8;
        printf( "Age.age : %d\n", Age.age );
        return 0;
}
```

When the above code is compiled it will compile with a warning as shown

```
Coding Exampe_2.c: In function 'main':
17:14: warning: unsigned conversion from 'int' to 'unsigned char:3' changes value from '8' to '0' [-Woverflow]
   Age.age = 8;
             ^
```

Output:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Coding Example_3: Demonstration of assigning signed value to an Unsigned variable for which bit fields are specified.

```
#include<stdio.h>
struct Status
{
        unsigned int bin1:4;   // 4 bits is allocated for bin1. 0 to 15, any number can be used.
        unsigned int bin2:2;   // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));          // again 4 bytes
        struct Status s1;
        s1.bin1=-7;     // it is unsigned. but sign is given while assigning
        s1.bin2=2;
        printf("%d %d",s1.bin1,s1.bin2);    // 9 2
        return 0;
}
```

Coding Example_4: Structure having two members with signed and unsigned variable for which bit fields are specified.

```
#include<stdio.h>
 struct Status
{
          int bin1:4; // one bit is used for representing the sign. Another 3 bits for data
          unsigned int bin2:2; // 2 bits allocated for bin2. 0 to 3, any number can be used
};
int main()
{
         printf("Size of structure is %lu\n",sizeof(struct Status));  // again 4 bytes
         struct Status s1;
         s1.bin1=-7; s1.bin2=2;
         printf("%d %d",s1.bin1,s1.bin2);                              // -7 2
        return 0;
}
```

Coding Example_5: Array of bit fields not allowed. Below code results in Error.

```
#include<stdio.h>

struct Status
{
      unsigned char bin1[10]:40;     // invalid type
};
```

```
int main()
{
            printf("Size of structure is %lu\n",sizeof(struct Status));
            return 0;
}
```

Coding Example_6: We cannot have pointers to bit field members as they may not start at a byte boundary. This code results in error.

```
#include<stdio.h> struct
Status
{
          int bin1:4; // 1 bit is used for representing the sign. Another 3 bits for data
          int *p:2;
};
int main()
{
          printf("Size of structure is %lu\n",sizeof(struct Status));
          return 0;
}
```

Coding Example_7: Storage class can't be used for bit field. This code results in error.

```
 #include<stdio.h>
  struct Status
  {
                 static int bin1:32;// Any storage class not allowed for bit field
                 int p:2;
  };
  int main()
  {
                 printf("Size of structure is %lu\n",sizeof(struct Status));
                  return 0;
  }
```

Coding Example_8: It is implementation defined to assign an out-of-range value to a bit field member. This code results in error.

```
#include<stdio.h>
struct Status
{
        int bin1:33;    // Error: width exceeds its type
        int p:2;
};
int main()
{
        printf("Size of structure is %lu\n",sizeof(struct Status));
        return 0;
}
```

Coding Example_ 9: Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized. A zero-width bit field can cause the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field. Below code demonstrates the force alignment on next boundary.

```
#include<stdio.h>
struct Status
 {
            int bin1:4;      // one bit is used for representing the sign. Another 3 bits for data
            //int i:0;// you cannot allocate 0 bits for any member inside the structure. Error
             int :0;// A special unnamed bit field of size 0 is used to force alignment on
            // next boundary. observe no member variable. only size is 0 bits
              unsigned int bin2:2;   // 2 bits allocated for bin2. 0 to 3, any number can be used
 };
 int main()
 {
            // observe the size of the structure. Now 8 bytes
```

```
        printf("Size of structure is %lu\n",sizeof(struct Status));          // 8 bytes
        struct Status s1;
        s1.bin1=-7;
        s1.bin2=2;
        printf("%d %d",s1.bin1,s1.bin2);          // -7 2
        return 0;
}
```

Happy coding using Bit fields!!

Unit #: 3

Unit Name: Text Processing and User defined Types

Topic: Priority Queue

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and its respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

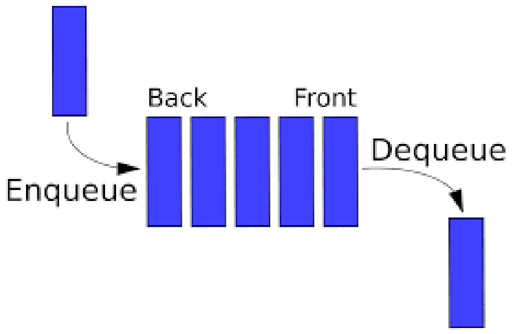
Jan - May, 2022

Dept. of CSE,

PES University

Introduction

Queue: A line or a sequence of people or vehicles waiting 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 Linear data structure which has 2 ends - Rear end and a Front end**. Data elements are inserted into the queue from the rear (back) end and deleted from the front end. Hence a queue is also known as **First In First Out (FIFO)** data structure.



Two major Operations on a queue:

- enqueue() − add (store) an item to the queue from rear end.
- dequeue() − remove (access) an item from the queue from front end.

Others operations may include,

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.
- isempty() − Checks if the queue is empty.

Types of Queue:

- **Ordinary queue** - insertion takes place at rear and deletion takes place at front.
- **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 first element of queue making circular link.
- **Double ended queue** - insertion and removal of elements can be performed from either from the front or rear.

Priority Queue

A data structure which is like a regular **queue** but where additionally each element has a "**priority**" associated with it. This priority decides about the deque operation. The enqueue operation stores the item and the priority information.

Types of Priority Queue:

- **Ascending priority queue:**
  - The smallest number, the highest priority.
- **Descending priority queue:**
  - The highest number, the highest priority.

Applications of Priority Queue:

1. Dijkstra's Algorithm
2. Prims Algorithm
3. Data Compression
4. Artificial Intelligence
5. Heap Sort

Priority Queue can be implemented using different ways.

- Using an Unordered Array
- Using an Ordered Array
- Using an Unordered Linked list
- Using an Ordered Linked list
- Using Heap

Happy Queueing!!

Unit #: 3

Unit Name: : Text Processing and User-Defined Types

Topic: String in C

Course objectives:

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

Course outcomes:

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C contructs

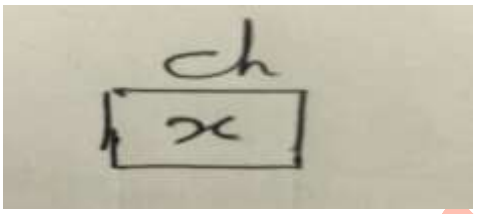
CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,
Jan - May, 2022
Dept. of CSE,
PES University

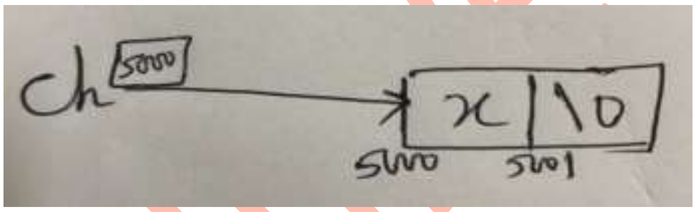
String in C

A **string in C is an array of characters and terminated with a special character '\0'** or NULL. ASCII value of NULL character is 0. Each character occupies 1 byte of memory. There is **NO datatype available to represent a string in C**.

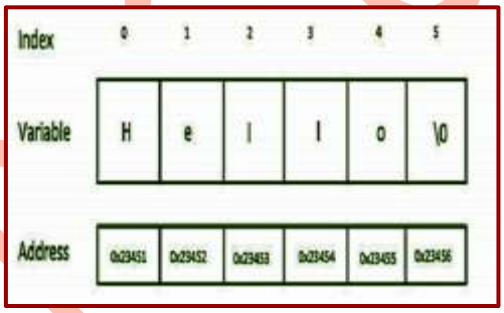
char ch = 'x' ; // character constant in single quote. always of 1 byte



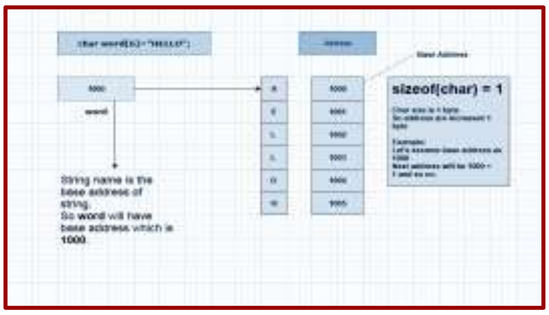
char ch[] = "x" ; // String constant. always in double quotes. Terminated with '\0'. Always 1 byte more when specified between " and " in initialization.



Note: When the compiler encounters a sequence of characters enclosed in the double quotes, it appends a null character '\0' at the end by default



Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456



String name is the base address of string.
So word will have base address which is 1000.

Declaration and Initialization of String

Declaration:

char variable_name[size]; // Size of the array must be specified compulsorily.

Initialization:

If the size is not specified, the compiler counts the number of elements in the array and allocates those many bytes to an array.

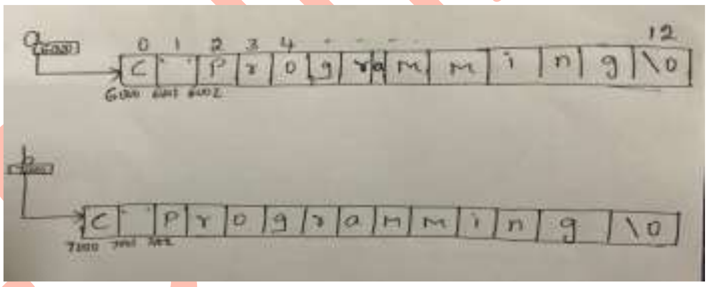
If the size is specified, it allocates those many bytes and unused memory locations are initialized with default value '\0'. This is partial initialization.

If the string is hard coded, it is programmer's responsibility to end the string with '\0'character.

char a[] = { 'C', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm, 'i', 'n', 'g', '\0' };

char b[] = "C Programming" ;

C standard gives shorthand notation to write initializer's list. b is a shorthand available only for storing strings in C.

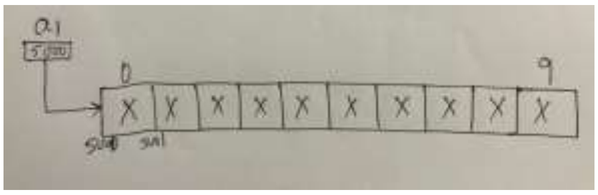


Let us consider below code segments.

char a1[10];

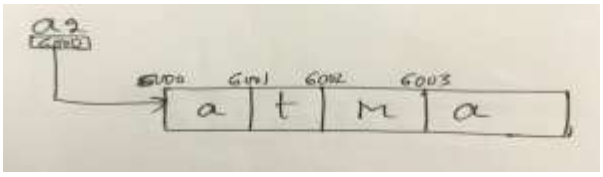
// **Declaration:** Memory locations are filled with undefined values/garbage value

printf("sizeof a1 is %d",sizeof(a1)); // 10



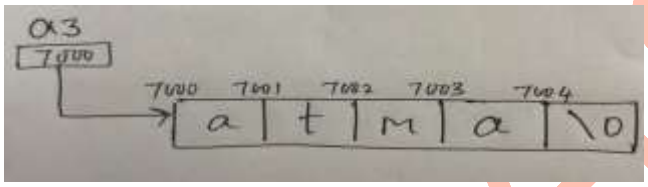
char a2[ ] = {'a','t','m','a'};//**Initialization**

printf("sizeof a2 is %d",sizeof(a2)); // 4 but cannot assure about a2 while printing



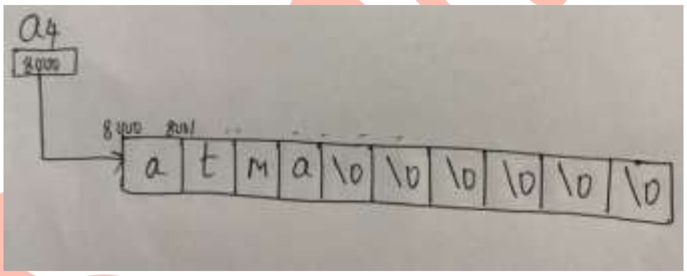
char a3[ ] = "atma" ;

printf("sizeof a3 is %d",sizeof(a3)); // 5sure about a3 while printing



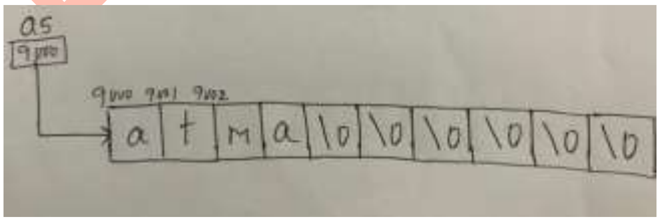
char a4[10] = {'a','t','m','a'}// **Partial Initialization**

printf("sizeof a4 is %d",sizeof(a4));// 10sure about a4 while printing

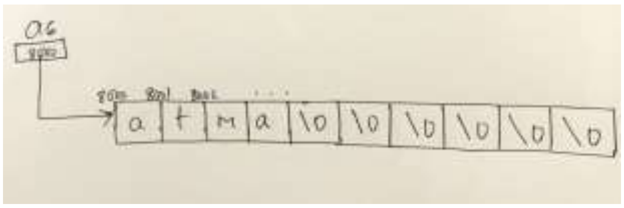


char a5[10] = "atma" ;

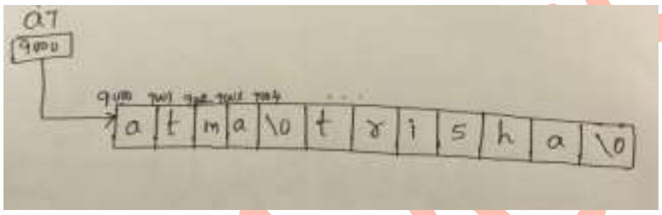
printf("sizeof a5 is %d",sizeof(a5));// 10 sure about a5 while printing



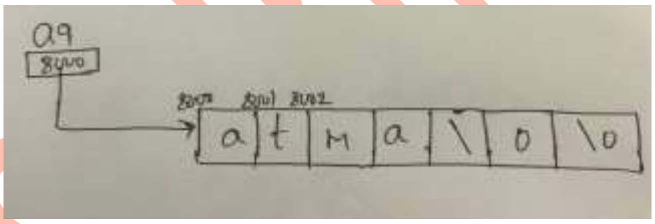
```
char a6[10 ] = {'a','t','m','a', '\0'};
printf("sizeof a6 is %d",sizeof(a6));// 10 sure about a6 while printing
```



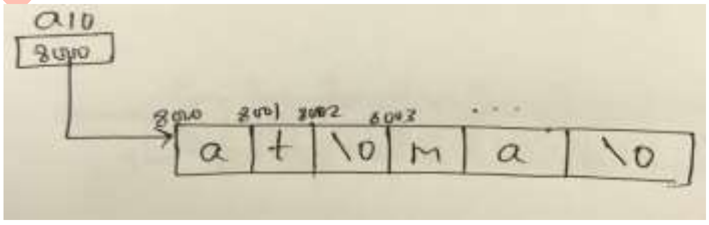
```
char a7[ ] = {'a', 't', 'm', 'a', '\0', 't', 'r', 'i', 's', 'h', 'a', '\0' };
printf("sizeof a7 is %d",sizeof(a7));  // 12 a7 will be printed only till first '\0'
```



```
char a9[ ] = "atma\\0" ;
printf("sizeof a9 is %d",sizeof(a9));// 7 sure about a9 while printing
```



```
char a10[ ] = "at\0ma" ;
printf("sizeof a10 is %d",sizeof(a10));//6 a10 will be printed only till first '\0'
```



Read and Display Strings in C

As usual, the formatted functions, **scanf and printf are used to read and display** string. **%s is the format specifier for string.**

Consider char str1[] = {'a', 't', 'm', 'a', 't', 'r', 'i', 's', 'h', 'a', '\0' };
One way of printing all the characters is using putchar in a loop.

```
int i;
for (i = 0; i<sizeof(str1); i++)
        printf("%c",str1[i]);        //each element of string is a character. So %c.
```

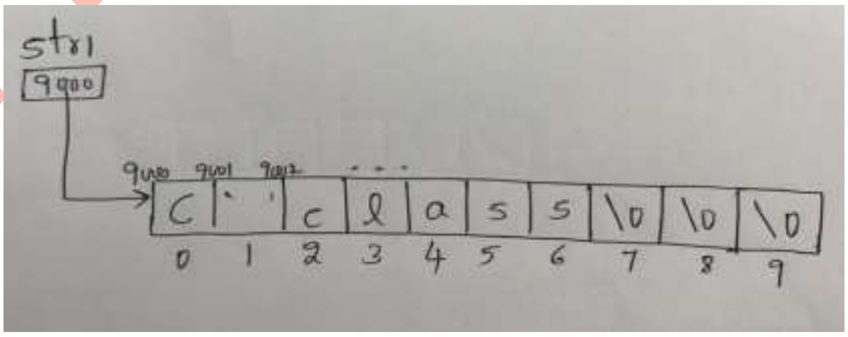
Think about using while(str1[i] != '\0') rather than sizeof

```
while(str1[i] != '\0')
{
    printf("%c",str1[i]);
    i++;
}
```

Also taking each character from the user using getchar() as we did in Unit_1 Last problem solution, looks like a tedious task. So, use %s format specifier and No loops required. **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.**

Consider,

```
char str1[10] = "C class" ;
printf("%s", str1);  // C class
```

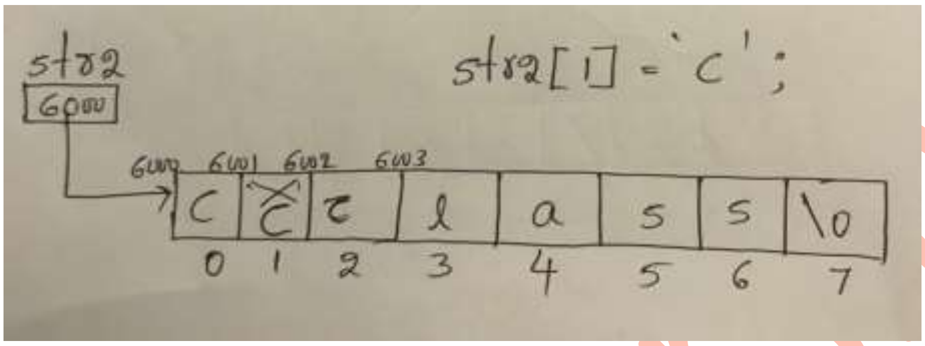


char str2[ ] = "C class" ;

printf("%s\n", str2);// C class

str2[1] = 'C' ;

printf("%s\n", str2); // CCclass



Let us deal with how to take the input from the user for a string variable.

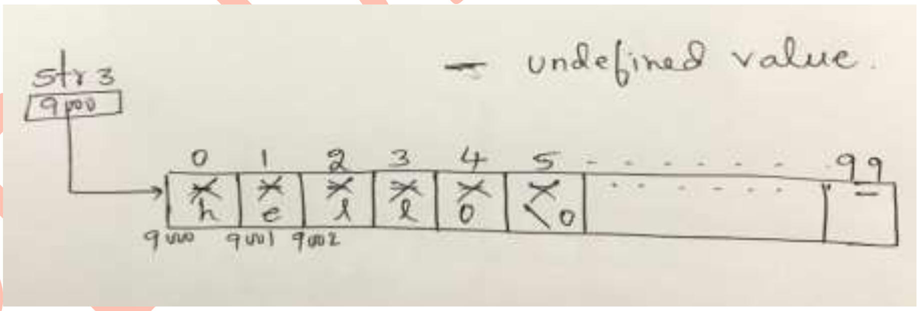
Coding Example_1:

char str3[100] ;

printf("Enter the string");

scanf(("%s", str3);   // User entered Hello Strings and pressed enter key

printf("%s\n",str3);   // Hello   . Reason: scanf terminates when white space in the user input



Coding Example_2:

char str4[100] ;

char str5[100] ;

printf("Enter the string");

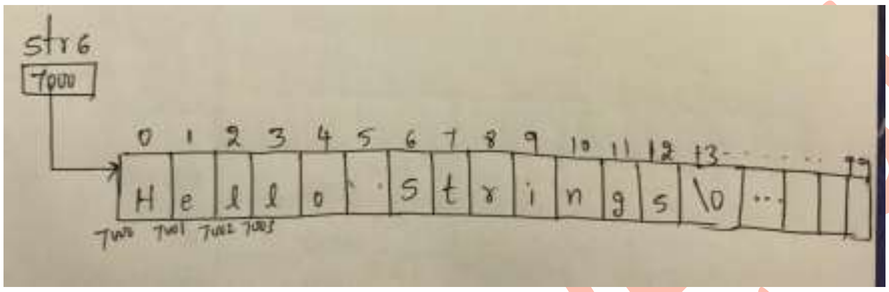
scanf(("%s %s", str4, str5);   // User entered Hello Strings and pressed enter key

printf("%s   %s",str4, str5);   // Hello   Strings

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. Till \n encountered, everything has to be stored in one variable.

Coding Example_3:

```
char str6[100] ;
printf("Enter the string");
scanf(("%[^\n]s", str6);                    // User entered Hello Strings and pressed enter key
printf("%s", str6);                  // Hello Strings
```

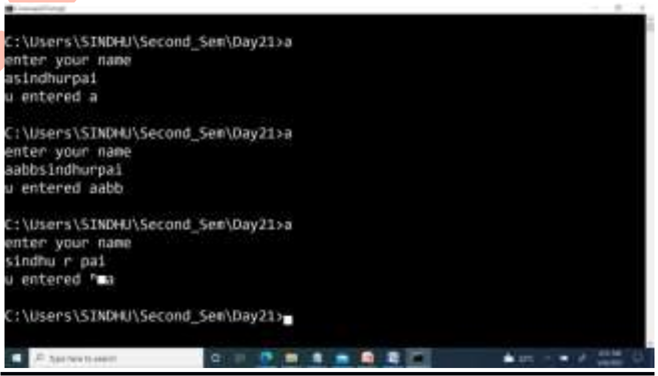


If you want few restricted characters only to be allowed in user input, how do you handle it?

Coding Example_4:

```
char str6[100] ;
printf("enter your name\n");
scanf("%[abcd]s",str6); // [ ] chracter class, starting with either a or b or c or d.
//When it encounters other characters, scanf terminates
printf("u entered %s\n",str6);
```

Output:



```
C:\Users\SINDHU\Second_Sem\Day21>a
enter your name
asindhurpai
u entered a

C:\Users\SINDHU\Second_Sem\Day21>a
enter your name
aabbsindhurpai
u entered aabb

C:\Users\SINDHU\Second_Sem\Day21>a
enter your name
sindhu r pai
u entered ⌐■a

C:\Users\SINDHU\Second_Sem\Day21>
```

PES University

Coding Example_5: How to negate the above character class

char str7[100] ;

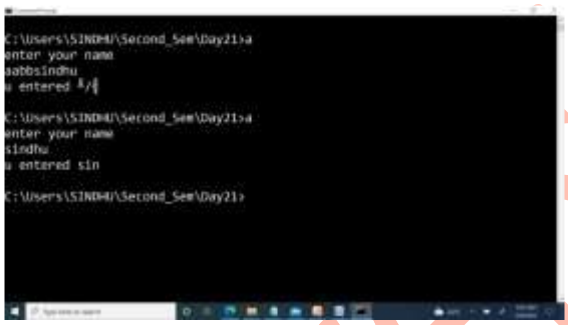
printf("enter your name\n");

scanf("%[^abcd]s",str6); // neither a nor b nor c nor d

//When it encounters any of these characters, scanf terminates

printf("u entered %s\n",str6);

Output:



```
C:\Users\SINDHU\Second_Sem\Day21>a
enter your name
aabbsindhu
u entered [illegible]

C:\Users\SINDHU\Second_Sem\Day21>a
enter your name
sindhu
u entered sin

C:\Users\SINDHU\Second_Sem\Day21>
```

Pointers v/s String

As array and pointers are related, strings and pointers go hand in hand.Let us consider few examples**.**

Coding Example_6:

char *p1 = "pesu";

//p1 is stored at stack. "pes" is stored at code segment of memory. It is read only.

printf("size is %d\n", sizeof(p1));   // size of pointer

// This statement assigns to p variable a pointer to the character array

printf("p1 is %s", p1) ;   //pesu   p1 is an address. %s prints until \0 is encountered

p++;   // Pointer may be modified to point elsewhere.

printf("p1 is %s", p1) ;   //esu

p1[1] = 'S' ;   // No compile time Error

printf("p1 is %s", p1) ; //**Behaviour is undefined if you try to modify the string contents**

Coding Example_7:

```
char p2[] = "pesu";   // Stored in the Stack segment of memory
printf("size is %d\n", sizeof(p2));      // 5 = number of elements in array+1 for '\0'
printf("p2 is %s", p2) ;          //pesu
p2[1] = 'E';    //Individual characters within the array may be modified.
printf("p2 is %s", p2) ;          //pEsu
p2++;  // Compile time Error
// Array always refers to the same storage. So array is a Constant pointer
```

Built-in String Functions

Even if the string is not a primitive type in C, there are few string related functions available in **string.h** header file. Include this header file if you are using any functions from this. Result is undefined if '\0' is not available at the end of character array which is passed to builtin string functions. These **string functions expect '\0'.**

Here are few which are available.

- **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 +ve, if array a has lexicographically higher value than b. Else, -ve.

Few more to think about: strncmp, strcmpi, strncat, strrev, strlwr, strupr, strrchr, strstr, strrstr,strset, strnset

Refer to below code segments to understand few builtin string functions.

Coding Example_8: String Length and String copy

```
char str1 = "pes";
printf("string length is %d\n",strlen(str1));            // 3
char a[10]="abcd";
char b[20],c[10];
//b=a;      //we are trying to equate two addresses. Array Assignment incompatible
strcpy(b,a);              // copy a to b.
printf("a is %s and b is %s\n",a,b);
printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));
```

Coding Example_9: Comparing two strings using Lexicographical ordering

```
char a[10]="abcd"; char b[10]="abcd"; char c[10]="AbCD";char d[10]="abD";
int i=strcmp(a,b);
int j=strcmpi(a,c); // ignore case
int k=strncmp(a,d,2); // compare only n characters , here 2
printf("first %d--cmp\t%d--cmpi\t%d--ncmp\n",i,j,k);
i=strcmp(a,d);
j=strcmpi(a,d);
k=strncmp(a,d,3);
printf("later %d--cmp\t%d--cmpi\t%d--ncmp\n",i,j,k);
```

Coding Example_10: String concatenation

```
char a[100]="abcd";
char b[100]="pqr";
char c[100]="whatsapp";
strcat(a,b);              // Make sure a is big enough to hold a and b both.
printf("a is %s and b is %s\n",a,b);
printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));
// make sure size of a is big enough to hold b as well
strncat(a,c,2);
printf("a is %s and size is %d\n",a,strlen(a));
```

Coding Example_11: Finding character in a given string

```
char ch[] = "This is a test";
//printf("present at %p\n",strchr (ch, 't')); // returns address
char *pos = strchr(ch,’t’);
int p = pos - str1; // pointer arithmetic
if(pos)
    printf( "char at pos %d",p);
else
    printf("not available");
```

String operations using User defined functions

Let us write user defined functions to understand few string operations in detail.

Coding Example_12:

```
char mystr[ ] = "pes";
printf("Length is %d\n", my_strlen(mystr));
```

Different versions of my_strlen() implementation are as below.

Version 1:Iterate through every character of the string using a variable count. Stop iteration when NULL character is encountered. Return the value of i.

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

Version 2: Run the loop till *s becomes '\0' character. Increment the pointer and the counter when *str is not NULL.

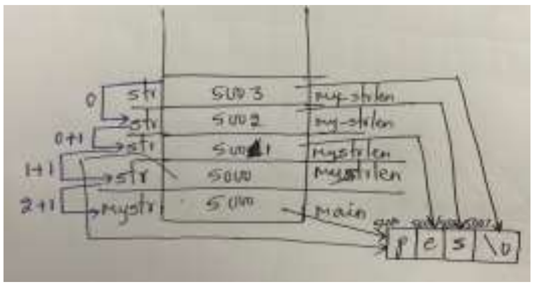
```
int my_strlen(char *str)
{
        int  i =0;
        while(*str){            // str[i] != '\0'   // *(str+i) != '\0'    ---> all these are same
                i++;     str++;
        }
        return i;
}
```

Version 3: Using recursion and pre-increment

```
int my_strlen(char *str)
{
      if (!(*str))
              return 0;
       else
              return 1+my_strlen(++str);
}
```

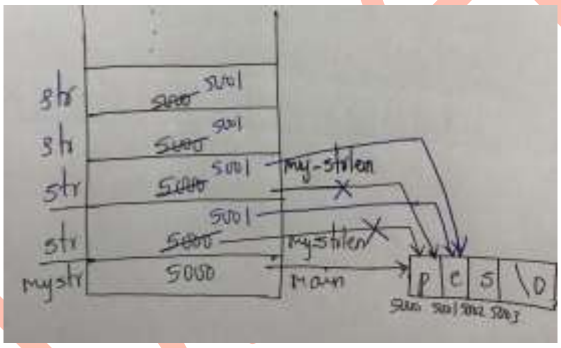
Version 4: Using recursion and no change in the pointer in the function call

```
int my_strlen(char *str)
{        if (!(*str))
              return 0;
        else   return 1+my_strlen(str+1);
}}
```



Version 5: Using recursion and post-increment

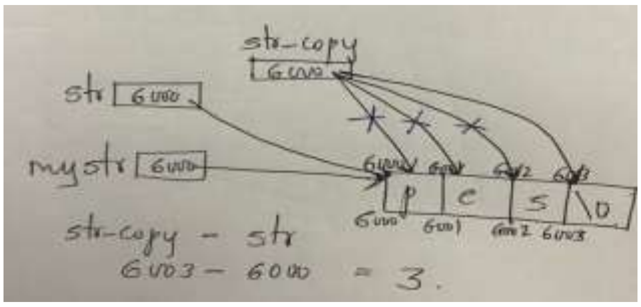
```
int my_strlen(char *str)
{
        if (!(*str)) return 0;
        else   return 1+my_strlen(str++);
}
```



Version 6: Using Pointer Arithmetic

Use a local pointer which points to the first character of the string. Keep incrementing this till '\0' is found. Then subtract this from the pointer specified in the parameter which points to the beginning of the string. This finds the length of the string.

```
int my_strlen(char *str)
{
        char *str_copy = str;
        while(str_copy){     str_copy++;           }
        return str_copy - str;
 }
```



Coding Example_13:

```
        char mystr1[] = "pes university";
        char mystr2[100];
        printf("%s\n",mystr1);
        my_strcpy(mystr2, mystr2);
        printf("%s\n",mystr2);
```

Different versions of my_strcpy() implementation are as below.

Version 1: Using arrays

```
void my_strcpy(char *b, char *a)
{
        // copy a to b
        int i =0;
        while(a[i] != '\0')
        {
                b[i]=a[i];
                i++;
        }
        b[i] = '\0';        // append '\0' at the end
}
```

Version 2: Using pointers

```
void my_strcpy(char *b, char *a)
  {     // copy a to b
        while(*a != '\0')
        {       *b = *a;
                b++;a++;
        }
		*b = '\0';		// append '\0' at the end
}
```

Version 3:

```
void my_strcpy(char *b, char *a)
{		while((*b = *a) != '\0')
		{			b++;	a++;	}
}
```

Version 4:

```
void my_strcpy(char *b, char *a)
{
		while(*b++ = *a++);		// same as for(;*b++ = *a++;);
}
```

Coding Example_14:

```
		char  str1[100];  char  str2[100];
		printf("enter the first string\n");
		scanf("%s",str1);
		printf("enter the second string\n");
		scanf("%s",str2);
		int res = my_strcmp(str1,str2);
		printf("result is %d\n",res);
		if(!res)
			printf("%s and %s are equal\n",str1,str2);
		else if(res > 0)
			printf("%s is higher than %s\n",str1,str2);
		else
			printf("%s is lower than %s\n",str1,str2);
```

Different versions of my_strcmp() implementation are as below.

Version 1:

Returns <0 if a<b, 0 if a==b , >0 if a>bint

```
my_strcmp(char *a, char *b)
{           int i;
            for(i = 0; b[i]!= '\0' && a[i] != '\0' && b[i]==a[i]; i++);
            return a[i]-b[i];
}
```

Version 2: pointer version

```
int my_strcmp(char *a, char *b)
{           for(;*b && *a && *b == *a; a++,b++);
            return *a - *b;
}
```

Coding Example_15:

```
            char str1[100];
            printf("enter the string\n");
            scanf("%s",str1);
            char ch = getchar();
            char *p = my_strchr(str, ch);
            printf("present in this address %d",p);
            if(p)
              printf("present in %d position\n", p - a);
            else
              printf("character not present\n");
```

Different versions of my_strchr() implementation are as below.

Version 1:

```
char* mystrchr(char *a, char c)
   {        char *p = NULL;
            char *s = a;
            while(*s != '\0' && p==NULL)
            {        if(*s == c)
                             p = s;
                     s++;
            }
            return p;
}
```

Version 2:

```
char *mystrchr(char *a,char c)
{           while(*a && *a != c)
            {        a++;     }
            if (!(*a)){  return NULL; }      // Can replace if else with return !(*a)? NULL: a;
            else     { return a; }
}
```

Coding Example_16:

```
            char str1[100]; char str2[100];
            printf("enter first string\n");
            scanf("%s",str1);
            printf("enter second string\n");
            scanf("%s",str2);
            strcat(str1,str2);
            printf("str1 is %s and str2 is %s\n",str1,str2);
```

Different versions of my_strcat() implementation are as below for the logic: my_strcat appends the entire second string to the first

Version 1:

```
void my_strcat(char *a,char *b)
   {        int i = 0;
            while(a[i]!= '\0')
            {              i++;           }              // while can be replaced with strlen()
            int j;
            for(j = 0;b[j] != '\0';j++,i++)
                    a[i] = b[j];
            a[i] = '\0';
     }
```

Version 2:

```
void my_strcat(char *a,char *b)
   {        while(*a){ a++; };            // increment a till NULL.
            while(*b)
            {       *a = *b;   a++;    b++;        }
         // Once NULL, copy each character from b to a and increment a and b both till b
becomes '\0'
            *a = '\0';       // assign '\0' character to a's end
}
```

Try to mimic the implementations of other functions like strncmp, strcmpi and strncat!!

Happy mimicking!!