**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **19/June/2020** | **Name:** | **Prashantha naik** |
| **Course:** | **C Programming** | **USN:** | **4al17ec074** |
| **Topic:** | **Module 5: Structures & Unions** | **Semester & Section:** | **6th b** |
| **GitHub Repository:** | **prashanth\_course** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
| **Report – Report can be typed or hand written for up to two pages.**  **Structures**  A structure is a user-defined data type that groups related variables of different data types.  A structure declaration includes the keyword struct, a structure tag for referencing the structure, and curly braces { } with a list of variable declarations called members.  For example:  struct course {  int id;  char title[40];  float hours;  };  **Structures with Structures**  The members of a structure may also be structures.  For example, consider the following statements:  typedef struct {  int x;  int y;  } point;  typedef struct {  float radius;  point center;  } circle;  **Unions**  A union allows to store different data types in the same memory location.  It is like a structure because it has members. However, a union variable uses the same memory location for all its member's and only one member at a time can occupy the memory location.  A union declaration uses the keyword union, a union tag, and curly braces { } with a list of members.  Union members can be of any data type, including basic types, strings, arrays, pointers, and structures.  For example:  union val {  int int\_num;  float fl\_num;  char str[20];  };  **Pointers to Unions**  A pointer to a union points to the memory location allocated to the union.  A union pointer is declared by using the keyword union and the union tag along with \* and the pointer name.  For example, consider the following statements:  union val {  int int\_num;  float fl\_num;  char str[20];  };  union val info;  union val \*ptr = NULL;  ptr = &info;  ptr->int\_num = 10;  printf("info.int\_num is %d", info.int\_num);  **Unions as Function Parameters**  A function can have union parameters that accept arguments by value when a copy of the union variable is all that is needed.  For a function to change the actual value in a union memory location, pointer parameters are required.  For example:  union id {  int id\_num;  char name[20];  };  void set\_id(union id \*item) {  item->id\_num = 42;  }  void show\_id(union id item) {  printf("ID is %d", item.id\_num);  } |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **19/June/2020** | **Name:** | **Prashantha naik** | |
| **Course:** | **C Programming** | **USN:** | **4al17ec074** | |
| **Topic:** | **Module 6: Memory Management** | **Semester&Section:** | **6th b** | |
| **Git hub repository** | **prashanth\_couse** |  |  | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| **Report – Report can be typed or hand written for up to two pages.**  **Memory Management**  **Dynamic memory allocation** is the process of allocating and freeing memory as needed. Now you can prompt at runtime for the number of array elements and then create an array with that many elements. Dynamic memory is managed with pointers that point to newly allocated blocks of memory in an area called the heap.  **The malloc Function**  The malloc() function allocates a specified number of contiguous bytes in memory.  For example:  #include <stdlib.h>  int \*ptr;  /\* a block of 10 ints \*/  ptr = malloc(10 \* sizeof(\*ptr));  if (ptr != NULL) {  \*(ptr + 2) = 50; /\* assign 50 to third int \*/  }  **The free Function**  The free () function is a memory management function that is called to release memory. By freeing memory, you make more available for use later in your program.  For example:  int\* ptr = malloc(10 \* sizeof(\*ptr));  if (ptr != NULL)  \*(ptr + 2) = 50; /\* assign 50 to third int \*/  printf("%d\n", \*(ptr + 2));  free(ptr);  **The calloc Function**  The calloc() function allocates memory based on the size of a specific item, such as a structure.  The program below uses calloc to allocate memory for a structure and malloc to allocate memory for the string within the structure:  typedef struct {  int num;  char \*info;  } record;  record \*recs;  int num\_recs = 2;  int k;  char str[ ] = "This is information";  recs = calloc(num\_recs, sizeof(record));  if (recs != NULL) {  for (k = 0; k < num\_recs; k++) {  (recs+k)->num = k;  (recs+k)->info = malloc(sizeof(str));  strcpy((recs+k)->info, str);  }  }  **The realloc Function**  The realloc() function expands a current block to include additional memory.  For example:  int \*ptr;  ptr = malloc(10 \* sizeof(\*ptr));  if (ptr != NULL) {  \*(ptr + 2) = 50; /\* assign 50 to third int \*/  }  ptr = realloc(ptr, 100 \* sizeof(\*ptr));  \*(ptr + 30) = 75;  **Allocating Memory for Strings**  When allocating memory for a string pointer, you may want to use string length rather than the sizeof operator for calculating bytes.  Consider the following program:  char str20[20];  char \*str = NULL;  strcpy(str20, "12345");  str = malloc(strlen(str20) + 1);  strcpy(str, str20);  printf("%s", str); | | | |