**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **19-06-2020** | **Name:** | **Kavya M M** |
| **Course:** | **C programming** | **USN:** | **4AL17EC040** |
| **Topic:** | 1. **Structures and union** 2. **Memory allocation** | **Semester & Section:** | **6th A** |
| **Github Repository:** | **Kavya\_ECE040** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
|  |
| 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.  Declarations Using Structures:   * To declare variables of a structure data type, you use the keyword struct followed by the struct tag, and then the variable name. For example, the statements below declares a structure data type and then uses the student struct to declare variables s1 and s2   Accessing Structure Members:  You access the members of a struct variable by using the **.** (dot operator) between the variable name and the member name. For example, to assign a value to the age member of the s1 struct variable, use a statement like  Using typedef   * The typedef keyword creates a type definition that simplifies code and makes a program easier to read. * typedef is commonly used with structures because it eliminates the need to use the keyword struct when declaring variables.   Pointers to Structures  Just like pointers to variables, pointers to structures can also be defined.  struct myStruct \*struct\_ptr; defines a pointer to the *myStruct* structure.  struct\_ptr = &struct\_var; stores the address of the structure variable *struct\_var* in the pointer *struct\_ptr*.  struct\_ptr -> struct\_mem; accesses the value of the structure member *struct\_mem*.  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.   Unions are used for memory management. The largest member data type is used to determine the size of the memory to share and then all members use this one location. This process also helps limit memory fragmentation. Memory management is discussed in a later lesson.  Accessing Union Members  You access the members of a union variable by using the . dot operator between the variable name and the member name. When assignment is performed, the union memory location will be used for that member until another member assignment is performed.  Trying to access a member that isn't occupying the memory location gives unexpected results.  Memory Management   * Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the stack.  An int variable, for example, is typically allocated 4 bytes when declared. We know this by using the sizeof operator:   int x;  printf ("%d", sizeof(x));   * As another example, an array with a specified size is allocated contiguous blocks of memory with each block the size for one element   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.  Memory Management Functions   * The stdlib.h library includes memory management functions. The statement #include <stdlib.h> at the top of your program gives you access to the following: * malloc(*bytes*) Returns a pointer to a contiguous block of memory that is of size *bytes*. * calloc (*num\_items, item\_size*) Returns a pointer to a contiguous block of memory that has *num\_items* items, each of size *item\_size* bytes. Typically used for arrays, structures, and other derived data types. The allocated memory is initialized to 0. * Realloc (*ptr, bytes*) Resizes the memory pointed to by ptr to size bytes. The newly allocated memory is not initialized.  free(*ptr*) Releases the block of memory pointed to by ptr.   The malloc Function   * The malloc () function allocates a specified number of contiguous bytes in memory.   malloc returns a pointer to the allocated memory.   * Notice that sizeof was applied to \*ptr instead of int, making the code more robust should the \*ptr declaration be changed to a different data type later.   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   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  The realloc Function   * The realloc () function expands a current block to include additional memory. * realloc leaves the original content in memory and expands the block to allow for more storage. |