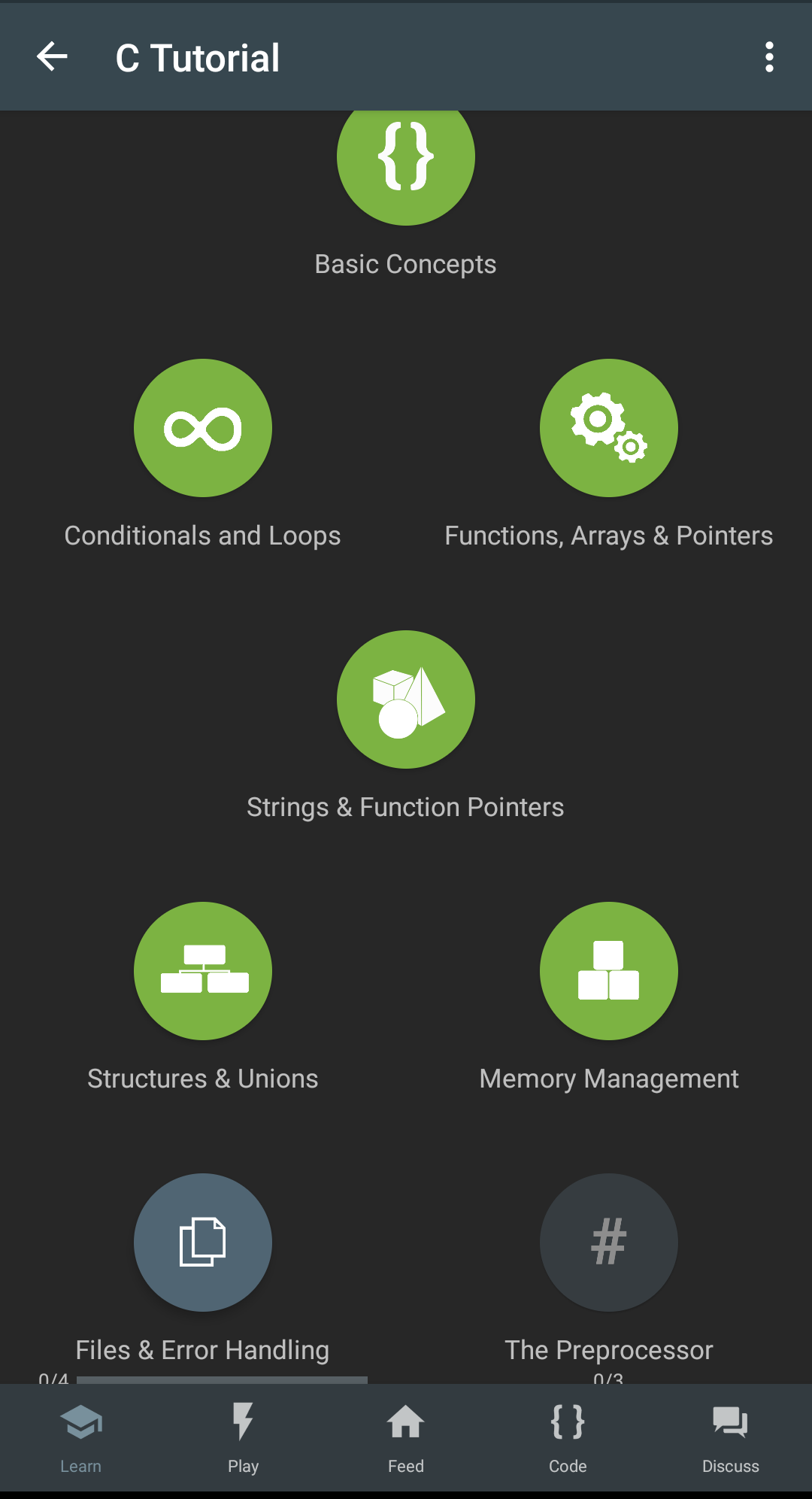
**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **19/06/2020** | **Name:** | **Lavanya B** |
| **Course:** | **C programming** | **USN:** | **4al17ec043** |
| **Topic:** | **Structure and union**  **Memory management** | **Semester & Section:** | **6th A** |
| **Github Repository:** | **Lavanya-B** |  |  |

**Imageof the session**

****

**Report**

**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.**

**To declare variables of a structure data type, you use the keyword struct followed by the struct tag, and then the variable name.**

**Eg:**

**#include <stdio.h>**

**struct student {**

**int age;**

**int grade;**

**char name[40];**

**};**

**int main()**

**struct student s1;**

**struct student s2;**

**s1.age = 19;**

**s1.grade = 9;**

**sprintf(s1.name, "John Bighimer");**

**s2.age = 22;**

**s2.grade = 10;**

**sprintf(s2.name, "Batman Jokerson");**

**printf("Student: %s, %d\n", s1.name, s1.age);**

**printf("Student: %s, %d\n", s2.name, s2.age);**

**return 0;**

**}**

**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.**

**Accessing the members of a union variable by using the . dot operator between the variable name and the member name.access the members of a union variable by using the . dot operator between the variable name and the member name.**

**Eg:**

**#include <stdio.h>**

**union val {**

**int int\_num;**

**float fl\_num;**

**char str[20];**

**};**

**int main() {**

**union val test;**

**test.int\_num = 123;**

**test.fl\_num = 98.76;**

**strcpy(test.str, "hello");**

**printf("%d\n", test.int\_num);**

**printf("%f\n", test.fl\_num);**

**printf("%s\n", test.str);**

**return 0;**

**}**

**Pointers to union**

**Eg:**

**#include <stdio.h>**

**#include <string.h>**

**union val {**

**int int\_num;**

**float fl\_num;**

**char str[20];**

**};**

**int main() {**

**union val info;**

**union val \*ptr = NULL;**

**ptr = &info;**

**ptr->int\_num = 10;**

**printf("info.int\_num is %d", info.int\_num);**

**return 0;**

**}**

**Memory management**

**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.**

**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.**

**Eg:**

**#include <stdio.h>**

**int main() {**

**int arr[10];**

**printf("%d", sizeof(arr));**

**return 0;**

**}**

* **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.**

**Eg: for malloc()**

**#include <stdio.h>**

**#include <stdlib.h>**

**int main() {**

**int \*ptr;**

**ptr = malloc(10\*sizeof(\*ptr));**

**if (ptr != NULL) {**

**\*(ptr+2) = 50;**

**}**

**printf("3rd elem equals to %d", \*(ptr + 2));**

**return 0;**

**}**

**Eg: for free()**

**#include <stdio.h>**

**#include <stdlib.h>**

**int main() {**

**int \*ptr;**

**ptr = malloc(10\*sizeof(\*ptr));**

**if (ptr != NULL)**

**\*(ptr+2) = 50;**

**printf("%d\n", \*(ptr+2));**

**free(ptr);**

**return 0;**

**}**

**Eg: for calloc()**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**typedef struct {**

**int num;**

**char \*info;**

**} record;**

**int main() {**

**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);**

**}**

**}**

**for (k = 0; k < num\_recs; k++) {**

**printf("%d\t%s\n", (recs+k)->num, (recs+k)->info);**

**}**

**return 0;**

**}**

**Eg: for realloc()**

**#include <stdio.h>**

**#include <stdlib.h>**

**int main() {**

**int \*ptr;**

**ptr = malloc(10\*sizeof(\*ptr));**

**if (ptr != NULL) {**

**\*(ptr+2) = 50;**

**}**

**ptr = realloc(ptr, 100\*sizeof(\*ptr);**

**\*(ptr+30) = 75;**

**printf("%d %d", \*(ptr+2), \*(ptr+30));**

**return 0;**

**}**