Advanced C programming

Ramtin Behesht Aeen

June 2024

Contents

Part I

Basic C Data Structures, Pointers, and File Systems

Chapter 1

Basic C Data Structures

1.1 Arrays

1.1.1 Concepts

In C: Array is a collection of consecutive objects with same data type

- Array is a variable

- Array has a data type and name with square bracket

- Within the brackets are the number of elements in the array

```
1  float best_score[3] = {
2  1.1, 2.1, 3.1, 4.1
3  };
```

- In C Arrays are non dynamic it means that their size be altered as the program runs.

- Arrays in C have no bounds checking, so it is possible to reference an element, which is not exist. calling element outside the Array declaration

- Arrays have a lot in commons with pointers

- It is possible to declare an arrays length as the program runs but it is mostly avoided, so jest set the value in the code and know that it can not be increased when program runs

Dynamic Arrays

Example of declare an array with use of pointers and still set its length at runtime:

- Code:

```
1
2
3   printf("Enter the size of the Array: ");
4   scanf("%d",&arraySize);
5
6   // Dynamically allocate memory for the array
7   myArray = (int *)malloc(arraySize * sizeof(int));
8
9   //Checking wheter rhe memory allocation was successful or not:
10  if(myArray == NULL) {
11          fprintf(stderr, "Memory allocation failed\n");
12          return 1;
13  }
14
15  //Initialize array with values:
16  for(int i = 0; i < arraySize; i++) {
17          myArray[i] = i * i;
18  }
19
20  //Print Arrays Values
21  for (int i = 0; i < arraySize; i++){
22          printf("myArray[%d] = %d\n",i, myArray[i]);
23  }
24
25  //Free the allocated memory
26  free(myArray);
27  return 0;
28
29  }
```

- Compile result:

```
1           Enter the size of the Array: 10
2           myArray[0] = 0
3           myArray[1] = 1
4           myArray[2] = 4
5           myArray[3] = 9
6           myArray[4] = 16
7           myArray[5] = 25
8           myArray[6] = 36
9           myArray[7] = 49
10          myArray[8] = 64
11          myArray[9] = 81
```

In this code, arraySize is determined at runtime based on the user's input. The malloc function is used to allocate the required amount of memory. It's important to free the allocated memory with free when you're done using the dynamically allocated array to prevent memory leaks. Keep in mind that dynamic memory allocation allows for flexible array sizes, but it also requires careful management of the allocated memory.

Example of declare an array with out useing pointers and still set its length at runtime(Using Variable Length Arrays (VLA)):

```
#include <stdio.h>

int main(){
        int arraySize;

        //Ask the user for the array size:
        printf("Enter the size of the array: ");
        scanf("%d", &arraySize);

        //declare VLA based on the user input:
        int myArray[arraySize];

        //Intitialize array with values
        for (int i = 0; i < arraySize; i++){
                myArray[i] = i * i;
        }

        //Print array values:
        for (int i = 0; i < arraySize; i++){
        printf("myArray[%d] = %d\n", i, myArray[i]);
        }

        return 0;
}
```

- Compile result:

```
        Enter the size of the Array: 5
        myArray[0] = 0
        myArray[1] = 1
        myArray[2] = 4
        myArray[3] = 9
        myArray[4] = 16
```

In this code, myArray is a VLA whose size arraySize is determined by the user input at runtime. No pointers are used, and the array is directly accessed by its indices. Keep in mind that not all compilers support VLAs, and their use is controversial due to potential risks such as stack overflow. Also, VLAs are not part of the ISO C++ standard, so they are

not portable across all platforms or languages. For these reasons, dynamic memory allocation with pointers is generally preferred for arrays with sizes determined at runtime.

1.1.2   Working With Arrays

dublicating an array

- In this code the process of duplicated an array is demonstrated:

⚠ Note: Duplicate must have the same or greater number of elements.(We as an Programmer must enforce this rule, because Compile wont check it)

```c
#include <stdio.h>

int main(){
        int original_array[5] = {10, 20, 30, 40, 50};
        int duplicate[5];

        for ( int i = 0; i < 5; i++){
                duplicate[i] = original_array[i];
        }

        puts("Arrays Values \n");

        for ( int j = 0; j < 5; j++ ){
                printf("Element#%d %3d == %3d \n", j,
                        original_array[j], duplicate[j]);
        }


}
```

- Compile Result:

```
Arrays Values

Element#0 10 ==  10
Element#1 20 ==  20
Element#2 30 ==  30
Element#3 40 ==  40
Element#4 50 ==  50
```

1.1.3   Passing an array to an function

passing the whole array

an entire array has been passed to the function. In the function, the argument is a character variable array with empty bracket and indicates that the entire

array has passed. within the function the array has modified and output.

- Code:

```
#include <stdio.h>

void print_array_func(char input_array[]){

        for ( int x = 0; x < 6; x++){
                input_array[x]++;
                putchar(input_array[x]);
        }
}

int main(){

        char text[] = "Gdkkn ";

        print_array_func(text);
        putchar('\n');

        return(0);

}
```

- Compile Result:

```
    Hello!
```

passing the arrays elements individually

- Code:

```c
#include <stdio.h>

void print_arrays_char( char a ){
        a++;
        putchar(a);
}

int main(){
        char text[] = "Gdkkn";

        for( int x = 0; x < 6; x++ ){
                print_arrays_char(text[x]);
        }
        putchar('\n');
}
```

- Compile Result:

```
Hello
```

retunring an array

⚠ Note: While individual array elements can be returned sequentially, to return an entire array created within a function, you must utilize pointers.

⚠ Wrong Way of returning an Array:

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

int make_array_func(void){
        int array[5];
        for ( int x = 0; x < 5; x++){
                array[x] = rand() % 10 + 1;
        }
        return(array);
}

int main(){
        int r[5];
        r = make_array_func();
        puts("Here are your 5 random numbers:");
        for (int x = 0; x < 6; x++)
                printf("%d\n",r[x]);
}
```

- Compile Result:

```
1    01_04_returning_an_array.c:12:15: warning: returning int *
         from a function with return type int makes integer from
         pointer without a cast [-Wint-conversion]
2    12 |        return(array);
3       |                   ^
4    01_04_returning_an_array.c: In function main:
5    01_04_returning_an_array.c:19:11: error: assignment to
         expression with array type
```

return an array from a function by returning a pointer to the array

in C, you can return an array from a function by returning a pointer to the array. However, you need to ensure that the array you're returning is not a local array inside the function, as it will be destroyed once the function scope ends. One way to do this is by dynamically allocating the array on the heap using malloc. Here's how you can modify your function to return an array using a pointer:

- The asterisk * before make_array_func in the function declaration indicates that the function returns a pointer.

- Code:

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    int* make_array_func(void){
5            //Dynamically allocate an array of 5 integers
6            int *array = malloc( 5 * sizeof(int));
7
8            if (array == NULL){
9                    //Handle memory allocation failure
10                   fprintf(stderr, "Memory allocation failed\n");
11                   exit(EXIT_FAILURE);
12           }
13           //Asigning values to the Array:
14           for ( int x = 0; x < 5; x++){
15                   array[x] = rand() % 10 + 1;
16           }
17
18           return array;
19   }
20
21   int main(){
22           int *r;
23
24           r = make_array_func();
```

```
25
26                  puts("Here are 5 random numbers:");
27                  for ( int x = 0; x < 5; x++) {
28                          printf("%d\n", r[x]);
29                  }
30                  //Frreing up the memory!! Dont Forgot that
31                  free(r);
32                  return 0;
33
34          }
```

- Compile Result:

```
1          Here are 5 random numbers:
2          4
3          7
4          8
5          6
6          4
```

1.1.4   Multi-Dimensional Array

- Example code of multi-dimensional array for creating a three by three matrix, which is fulled randomly with X or O:

```
1                  #include <stdio.h>
2                  #include <stdlib.h>
3                  #include <time.h>
4
5
6                  int main(){
7                          char tic_tac_toe [3][3];
8                          char array[] = {'x', 'o'};
9
10                         //Initiallize the board
11                         for ( int i = 0; i < 3; i++ ){
12                                 for ( int j = 0; j < 3; j++){
13                                         // Assign 'x' or 'o' randomly
                                              to the board
14                                         // rand() % 2 will Generate
                                              numbers between 0 and 1
15                                         tic_tac_toe[i][j] =
                                              array[rand() % 2];
16
17                                 }
18                         }
19
20                         //Printing the board
21                         for ( int i = 0; i < 3; i++ ){
```

```
22                        for ( int j = 0; j < 3; j++){
23                                printf(" %c ",
                                        tic_tac_toe[i][j]);
24                        }
25                        putchar('\n');
26                }
27
28                return(0);
29        }
```

- Compile Result:

```
1        o  o  x
2        x  o  x
3        x  x  x
```

1.1.5   Exercises

1.2   Structure

1.2.1   Concepts

- A structure is a container of multiple variable types.

- The variables can be different data types, the same data type, or mixed and matched in various quantities.

- All the variables relate to each other or describe a complex data structure like record of a Database.

- The key word 'Struct' will declare a structure, which is followed by name of the Structure and set of the .

- For using the structure we should define a variable with type of Struct

- Structure members can be accessed by using the structure varibale name, a dot and the name of the member(e.g, line 16 in Example)

- Example Code:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct animal {
5        int id;
6        int life_span;
7        char sound[20];
8        char class[10];
9  } ;
```

```
10
11  int main(){
12
13      struct animal german_shepherd;
14      struct animal chinchilla_persians;
15
16      german_shepherd.id = 31415;
17      german_shepherd.life_span = 20;
18      //german_shepherd.sound = "Bark";
19      strcpy(german_shepherd.sound,"Bark");
20      //german_shepherd.class = "Dog"
21      strcpy(german_shepherd.class,"Dog")
22
23      chinchilla_persians.id = 31416 ;
24      chinchilla_persians.life_span = 10;
25      //chinchilla_persians.sound = "Meow";
26      strcpy(chinchilla_persians.sound,"Meow");
27      //chinchilla_persians.class = "Cat";
28      strcpy(chinchilla_persians.class,"Cat");
29
30      printf("%d: German Shepherd is a %s breed, thus it will %s and
              live for %d ",
31              german_shepherd.id,
32              german_shepherd.class,
33              german_shepherd.sound,
34              german_shepherd.life_span
35          );
36      puts("\n");
37      printf("%d: Chinchilla Persians. is a %s breed, thus it will %s
              and live for %d ",
38              chinchilla_persians.id,
39              chinchilla_persians.class,
40              chinchilla_persians.sound,
41              chinchilla_persians.life_span
42          );
43      puts("\n");
44      return(0);
45  }
```

- Compile Result:

```
1  31415: German Shepherd is a Dog breed, thus it will Bark and live
       for 20
2  31416: Chinchilla Persians. is a Cat breed, thus it will Meow and
       live for 10
```

⚠️ Note: At lines 18, 20, 25, 27, there is an incorrect usage of memory allocation for the struct 'animal'. The reason is that the struct 'animal' is designed to use a block of memory that accommodates a 20-character

array and a 10-character array, along with 4 for an integer and another 4 for an another int, for example Declaring a variable of type struct animal (e.g., german_shepherd) will allocate memory on the stack. Assigning a string literal to german_shepherd.sound (e.g., german_shepherd.sound = "Bark") is incorrect because it attempts to copy the address of the string literal, which is a pointer, into an array This will fail because the compiler expects german_shepherd.sound to be an array within the struct, which is 38 bytes long in this context, and "Bark" is a string literal represented as a char array that is 5 bytes long (including the null terminator '\0' that signifies the end of C strings) Thus, the correct way to assign a string to an array is by using a function like strcpy, which copies the characters of the string into the array, including the null terminator, without needing the length hardcoded In C, when you want to copy a string into a character array, you should use functions like strcpy from the ¡string.h¿ library. This function handles the null-termination and prevents buffer overflow by not exceeding the array's allocated size. Always ensure that the destination array is large enough to hold the source string and the null terminator. Alternatively it is possible to define a struct which points to char arrays of any length:

```
1  struct animal {
2      int id;
3      int life_span;
4      char *sound;
5      char *class;
6  } ;
```

- Example Code of creating a struct variable using the structure definition statement (line 9)

```
1   #include <stdio.h>
2   #include <string.h>
3
4   struct animal {
5           int id;
6           int life_span;
7           char sound[20];
8           char class[10];
9   } german_shepherd ;
10
11  int main(){
12
13      german_shepherd.id = 31415;
14      german_shepherd.life_span = 20;
15      //german_shepherd.sound = "Bark";
16      strcpy(german_shepherd.sound,"Bark");
17      //german_shepherd.class = "Dog"
18      strcpy(german_shepherd.class,"Dog");
```

```
19
20      printf("%d: German Shepherd is a %s breed, thus it will %s and
             live for %d ",
21          german_shepherd.id,
22          german_shepherd.class,
23          german_shepherd.sound,
24          german_shepherd.life_span
25      );
26
27      puts("\n");
28      return(0);
29  }
```

- Compile Result:

```
1   31415: German Shepherd is a Dog breed, thus it will Bark and live
         for 20
```

- Example of Declaring and initialling values while defining an Structure(line 9):

```
1   #include <stdio.h>
2   #include <string.h>
3
4   struct animal {
5           int id;
6           int life_span;
7           char sound[20];
8           char class[10];
9    } german_shepherd = {31415, 20, "Bark", "Dog"} ;
10
11   int main(){
12
13          printf("%d: German Shepherd is a %s breed, thus it will %s
                 and live for %d ",
14               german_shepherd.id,
15               german_shepherd.class,
16               german_shepherd.sound,
17               german_shepherd.life_span
18             );
19
20          puts("\n");
21
22          return(0);
23   }
```

- Compile Result:

```
1   31415: German Shepherd is a Dog breed, thus it will Bark and live
        for 20
```

1.2.2 Nesting Structures

A structure can contain any type of variable as a member, and this includes other structures, which are referred to as nested structures. The code below demonstrates two structures: the first one is Date, which contains three members, and the second is Author, which includes a Date structure named birthday and a character array named name. Lines 17 to 20 populate the members of the Author structure variable. To access members of a nested structure, a dot operator is used for each level of nesting.

```c
#include <stdio.h>
#include <string.h>
int main(){
        struct Date {
                int month;
                int day;
                int year;
        };

        struct Author {
                struct Date birthday;
                char name[30];
        };

        struct Author author;

        author.birthday.day = 15;
        author.birthday.month = 10;
        author.birthday.year = 1844;
        strcpy(author.name, "Nietzsche");

        printf("%s was born on /%d/%d/%d",
                author.name,
                author.birthday.day,
                author.birthday.month,
                author.birthday.year
                );
        puts("\n");

}
```

```
1   Nietzsche was born on /15/10/1844
```

In this code, another Date structure was added to the Author structure. Multiple copies of a single structure can be used as members within another structure, each provided with its own unique variable name.

```c
#include <stdio.h>
#include <string.h>
int main(){
        struct Date {
                int month;
                int day;
                int year;
        };

        struct Author {
                struct Date birthday;
                struct Date died;
                char name[30];
        };

        struct Author author;

        author.birthday.day = 15;
        author.birthday.month = 10;
        author.birthday.year = 1844;

        author.died.day = 25;
        author.died.month = 8;
        author.died.year = 1900;

        strcpy(author.name, "Nietzsche");

        printf("%s was born on /%d/%d/%d and died on /%d/%d/%d",
                author.name,
                author.birthday.day,
                author.birthday.month,
                author.birthday.year,
                author.died.day,
                author.died.month,
                author.died.year
                );
        puts("\n");

}
```

```
Nietzsche was born on /15/10/1844
```

Example code of pre setting the structure with nested Structures

```c
#include <stdio.h>
#include <string.h>
int main(){
        struct Date {
                int month;
                int day;
                int year;
        };

        struct Author {
                struct Date birthday;
                struct Date died;
                char name[30];
        };

        struct Author author = {
                {15, 10, 1844},
                {25, 8, 1900},
                "Nietzsche"
        };


        printf("%s was born on /%d/%d/%d and died on /%d/%d/%d",
                author.name,
                author.birthday.day,
                author.birthday.month,
                author.birthday.year,

                author.died.day,
                author.died.month,
                author.died.year
        );

        puts("\n");

}
```

Compile Result:

```
Nietzsche was born on /10/15/1844 and died on /8/25/1900
```

1.2.3   Array of Structures

In C, just like any other variable, it is possible to create an array of structures, with each element of the array being occupied by an individual structure. at this example code we have an array with 3 elements, which is called author and from line ... to ... values are assigned to it:

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

 int main(){

        struct Date {
                int month;
                int day;
                int year;
        };

        struct Author {
                struct Date birthday;
                char name[30];
        };

        struct Author authors[3];

        authors[0].birthday.day = 15;
        authors[0].birthday.month = 10;
        authors[0].birthday.year = 1844;
        strcpy(authors[0].name, "Nietzsche");


        authors[1].birthday.day = 21;
        authors[1].birthday.month = 2;
        authors[1].birthday.year = 1789;
        strcpy(authors[1].name, "Schopenhauer");


        authors[2].birthday.day = 5;
        authors[2].birthday.month = 5;
        authors[2].birthday.year = 1813;
        strcpy(authors[2].name, "kierkegaard");

        for ( int x = 0; x < 3; x++ ){
                printf("Author#%d: %s was born on /%d/%d/%d \n",
                        x + 1,
                        authors[x].name,
                        authors[x].birthday.day,
                        authors[x].birthday.month,
                        authors[x].birthday.year
                        );
        };

        puts("\n");

 }
```

Compile Result:

```
1    Author#1: Nietzsche was born on /15/10/1844
2    Author#2: Schopenhauer was born on /21/2/1789
3    Author#3: kierkegaard was born on /5/5/1813
```

Example Code of Pre-Setting the Structure array value while defining it:

```
1    #include <stdio.h>
2    #include <string.h>
3
4    int main(){
5        struct Date {
6                int month;
7                int day;
8                int year;
9        };
10
11        struct Author {
12                struct Date birthday;
13                char name[30];
14
15        } authors[3] = {
16                { { 10, 15, 1844 }, "Nietzsche" },
17                { { 2, 21, 1789}, "Schopenhauer" },
18                { { 5, 5, 1813}, "kierkegaard"}
19        };
20
21
22        for ( int x = 0; x < 3; x++ ){
23                printf("Author#%d: %s was born on /%d/%d/%d \n",
24                        x + 1,
25                        authors[x].name,
26                        authors[x].birthday.day,
27                        authors[x].birthday.month,
28                        authors[x].birthday.year
29                        );
30        };
31
32        puts("\n");
33
34    }
```

Compile Result:

```
1    Author#1: Nietzsche was born on /15/10/1844
2    Author#2: Schopenhauer was born on /21/2/1789
3    Author#3: kierkegaard was born on /5/5/1813
```

1.2.4   Sending a Structure to a function

Sending one member of the Structure to the function

Example Code of passing one member of the Structure to the function:

Compile Result:

```
1       Nietzsche
2       was born on /15/10/1844
3       Schopenhauer
4       was born on /21/2/1789
5       kierkegaard
6       was born on /5/5/1813
```

Sending one member of the Structure to the function

Passing an Array of the structures to the function:

```
1   #include <stdio.h>
2   #include <string.h>
3
4
5   struct Date {
6           int month;
7           int day;
8           int year;
9   };
10
11  struct Author {
12          struct Date birthday;
13          char name[30];
14  };
15
16
17
18  /* Defining the Function:*/
19                      // struct structName arrayName
20  void show_author_infos(struct Author writers[]);
21
22  /* Defining the Function:*/
23  void show_author_name(char* inputString);
24
25  int main(){
26
27
28          struct Author authors[3];
29
30          authors[0].birthday.day = 15;
```

```
31          authors[0].birthday.month = 10;
32          authors[0].birthday.year = 1844;
33          strcpy(authors[0].name, "Nietzsche");
34
35
36          authors[1].birthday.day = 21;
37          authors[1].birthday.month = 2;
38          authors[1].birthday.year = 1789;
39          strcpy(authors[1].name, "Schopenhauer");
40
41
42          authors[2].birthday.day = 5;
43          authors[2].birthday.month = 5;
44          authors[2].birthday.year = 1813;
45          strcpy(authors[2].name, "kierkegaard");
46
47          show_author_infos(authors) ;
48    }
49
50
51    void show_author_infos(struct Author writers[]){
52
53          for ( int x = 0; x < 3; x++ ){
54
55                  /*Pasing the Author name to the function:*/
56                  show_author_name(writers[x].name);
57
58                  printf("was born on /%d/%d/%d \n",
59                          writers[x].birthday.day,
60                          writers[x].birthday.month,
61                          writers[x].birthday.year
62                      );
63          };
64
65    };
66
67
68    void show_author_name(char *inputString){
69          printf("%s \n", inputString);
70    };
```

Code Explanations:

1. **Structure Definition:** Structures are defined to represent complex data types that group together different related variables. In this code, 'Date' and 'Author' structures are defined to hold information about authors and their birth dates.

   ```c
   struct Date {
       int month;
       int day;
       int year;
   };

   struct Author {
       struct Date birthday;
       char name[30];
   };
   ```

2. **Function Prototypes:** Before the main function, prototypes for 'show_author_infos' and 'show_author_name' are declared. These prototypes inform the compiler about the functions' existence before their actual definitions later in the code.

   ```c
   void show_author_infos(struct Author writers[]);
   void show_author_name(char* inputString);
   ```

3. **Initialization of Structure Array:** An array of 'Author' structures is created and initialized in the 'main' function. Each element of the array represents an author and is populated with their name and birth date.

   ```c
   struct Author authors[3];
   // Initialization of authors array with names and birth dates
   ```

4. **Passing Structure Array to Function:** The array of 'Author' structures is passed to the 'show_author_infos' function. This is done by reference, which means the function has access to the original array and can modify it if needed.

   ```c
   show_author_infos(authors);
   ```

5. **Iterating Over Structure Array:** Inside 'show_author_infos', a loop iterates over the array of 'Author' structures. For each author, the function 'show_author_name' is called to print the author's name.

```
for (int x = 0; x < 3; x++) {
    show_author_name(writers[x].name);
    // ...
}
```

6. **Accessing Structure Members:** Still within the loop, the members of each 'Author' structure are accessed to print the author's birth date. The 'printf' function is used with format specifiers to display the day, month, and year.

```
printf("was born on %d/%d/%d\n",
        writers[x].birthday.day,
        writers[x].birthday.month,
        writers[x].birthday.year);
```

7. **Printing Author's Name:** The 'show_author_name' function receives a string (the name of the author) and prints it. This function demonstrates how to pass a single member of a structure to a function.

```
void show_author_name(char *inputString) {
    printf("%s \n", inputString);
}
```

⚠ Note: In C programming, the order in which you define structures and functions is crucial for successful compilation. Structures must be defined before functions that use them. This is because functions need to be aware of the structure definitions to handle them properly.

If you attempt to define a structure within a function before declaring it globally, the compiler will not recognize the structure in other parts of your program, leading to errors. Structures should be defined at the global scope if they are to be used by multiple functions.

Here's the correct order to avoid compilation errors:

Define all structures at the global scope. Declare function prototypes that will use these structures. Define functions that implement the declared prototypes. By following this order, you ensure that when the functions are compiled, the compiler has already seen the complete structure definitions and knows how to handle them.

Remember, a well-organized code structure is key to a smooth compilation process and the creation of maintainable and error-free programs.

1.2.5   Exercises

1.3   Union

1.4   Chapter Exercises

Chapter 2

Pointers

2.1   Concepts

A variable has 5 main properties:

- variable type (in this example variable type is: int)

- variable name (in this example variable name is: variable)

- variable value (in this example variable size is: 20)

- variable location in the memory (in this example variable location is: 0x7ffdd65ea944)

- number of bytes, that it has occupied int the memory (in this example variable occupies: 4 bytes)

```c
#include <stdio.h>

int main(){

        int variable = 20;

        printf("This variable value is: %d \n ", variable);
        printf("This variable occupies %lu bytes \n", sizeof( int ) );
        printf("This variable Address in memory is: %p \n", &variable);

        return(0);
}
```

Compile Result:

```
This variable value is: 20
This variable occupies 4 bytes
This variable Address in memory is: 0x7ffdd65ea944
```

Keynotes about the Pointers:

- A pointer is a variable that stores the memory address of another variable as its value.

- A pointer is a variable, so its value can be changed too

- A pointer can manipulate the address, which is holding.

- Pointer like other variables needing a datatype, and variable name, which is prefixed with an asterisk '*'

- Pointer like other variables should initialized and then used.

- Pointers are assigned the address of another variable, which has an <u>same data type</u>.

- ampersand operator (&) fetches variable's address.

- Pointer <u>with</u> * represent the <u>Data</u> stored at that memory location (**Line 12**)

- Pointer <u>without</u> * represent the <u>memory address</u>, in which the data is stored on.(**Line 9**)

2.2　Pointers in action:

Code:

```
1   #include <stdio.h>
2   int main() {
3          int numerical_var = 42;
4          int *numerical_var_ptr;
5
6          numerical_var_ptr = & numerical_var;
7
8          printf("Address of the variable numerical_var is: %p \n",
                   &numerical_var);
9          printf("the value stored in numerical_var_ptr is: %p \n",
                   numerical_var_ptr);
10
11         printf("Value of the variable numerical_var is: %d \n",
                   numerical_var);
12         printf("value of the memory address in numerical_var is: %d
                   \n", *numerical_var_ptr);
13
14         return(0);
15  }
```

After compiling and running the above code, the output will be something similar to:

```
1  Address of the variable numerical_var is: 0x7ffdea32e3fc
2  the value stored in numerical_var_ptr is: 0x7ffdea32e3fc
3  Value of the variable numerical_var is: 42
4  value of the memory address in numerical_var is: 42
```

1. **Variable Declaration and Initialization (Line 1):** An integer variable is declared and initialized to the value 42. This variable will store an integer value.

2. **Pointer Declaration (Line 2):** A pointer variable is declared. This pointer is intended to reference an integer value by storing the address of an integer variable.

3. **Address Assignment (Line 4):** The address of the integer variable is assigned to the pointer using the address-of operator (&).

4. **Address Printing (Line 6):** The address of the integer variable is printed using the `printf` function with the %p format specifier, which is used for pointers.

5. **Pointer Value Printing (Line 7):** The value stored in the pointer, which is the address of the integer variable, is printed.

6. **Variable Value Printing (Line 9):** The value of the integer variable is printed directly.

7. **Dereferencing and Value Printing (Line 10):** The pointer is dereferenced using the asterisk (*) operator to access the value stored at the referenced address. The value is then printed.

2.3   Incrementing Pointers Variable

Workers Are Working Here

2.4   Pointers and Arrays

Workers Are Working Here

2.5   malloc

2.5.1   concepts

- Argument of malloc function is the number of bytes desired.

- Return value is a memory location or the NULL constant

- Dont forgot to import ¡stdlib¿ header File

- Pointer will be used to access the memory chunck allocated by malloc()

- We use the free() function to release the memory

- at line .... in '(char *)malloc...' section we are type casting the result as an character pointer which should match the variable declaration at line ....

Here's an example of using malloc with pointers:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(){
5
6           char *buffer;
7
8           buffer = (char *)malloc( sizeof(char) * 128 );
9           if (buffer == NULL){
10                  puts("Unable to allocate memory");
11                  exit(1);
12          }
13          puts("Buffer allocated");
14          free(buffer);
15          puts("Buffer freed");
16   }
```

Compile Results:

```
1   Buffer allocated
2   Buffer freed
```

2.5.2   Why do we use 'malloc' when working with pointers?

In C programming, malloc is used for dynamic memory allocation, which means allocating memory at runtime rather than at compile time. Here are some reasons why malloc is used with pointers:

1. **Lifetime of Data:** When you allocate memory using malloc, the data persists beyond the scope of the function in which it was created. This is useful when you need to return a pointer to a data structure from a function or when the data needs to be accessed by multiple functions throughout the program1.

2. **Variable Size:** malloc allows you to allocate memory of a size that is determined at runtime. This is particularly useful when the size of the data structure cannot be known until the program is running2.

3. **Flexibility:** With malloc, you can allocate and deallocate memory as needed during the execution of your program. This provides flexibility in managing memory usage, especially for large or complex data structures2.

4. **No Stack Overflow:** Allocating large amounts of memory on the stack can lead to stack overflow.malloc allocates memory on the heap, which typically has a much larger size limit than the stack2.

5. **Dynamic Data Structures:** Many data structures like linked lists, trees, and graphs are dynamic and require memory to be allocated and deallocated as elements are added or removedmalloc is essential for implementing these structures2.

2.6   Using Pointers and Structures

2.6.1   Pointer as an Structure Member

Code:

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <string.h>
4
5    struct Person {
6            char *name;
7            int Id;
8    };
9
10   int main() {
11
12           struct Person person;
13           char buffer[32];
14
15           printf("Enter your name: ");
16           fgets(buffer, 32, stdin);
17
18           /* Allocate Storage */
19           person.name = (char *)malloc( strlen(buffer) + 1 );
20
21           if (person.name == NULL)
22           {
23                   puts("Unable to allocate Storage");
24                   exit(1);
25           }
26
27           /* Copying the buffer to the allocated storage */
28           strcpy( person.name , buffer);
29
30
```

```
31          printf("Enter the Id: ");
32          scanf("%d", &person.Id);
33
34          printf("User Name is: %s", person.name);
35          printf("User Id is: %d \n", person.Id);
36
37          return(0);
38  }
```

Compile Results:

```
1  Enter your name: Ramtin
2  Enter the Id: 31415
3  User Name is: Ramtin
4  User Id is: 31415
```

2.6.2   Structure Pointer

Code: In this example, the Structure is a pointer, which is called 'employee'.
So, in order to access its members, the structure pointer operator (->) instead
of the dot notation should be used.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5          struct person {
6                  char name[32];
7                  int age;
8
9          };
10
11         struct person *employee;
12
13         /* Allocating storage for the POinter Structure: */
14         employee = (struct person *)malloc( sizeof(struct person) * 1 );
15         if (employee == NULL){
16                 puts("Unable to allocate Storage");
17                 exit(1);
18         }
19
20         /*Getting the name of the Employee strucutre:*/
21         printf("Enter the Employees name: \n");
22         fgets(employee->name, 32, stdin);
23
24         /*Getting the age of the Employee strucutre:*/
25         printf("Enter the Employees age: \n");
26         scanf("%d", &employee->age);
27
```

```
28          printf("Employee's name is: %s", employee -> name);
29          printf("Employee's name is: %d \n", employee -> age );
30
31          return(0);
32  }
```

Compile Results:

```
1  Enter your name: Ramtin
2  Enter the Id: 31415
3  User Name is: Ramtin
4  User Id is: 31415
```

2.6.3   Pointer Member of a Structure Pointer

Example Code:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   struct Person {
6           char *name;
7           int age;
8   };
9
10  int main() {
11
12          char buffer[32];
13          struct Person *employee;
14
15          /* Allocating Storage for the Structure: */
16          employee = (struct Person *)malloc( sizeof( struct Person) * 1);
17          if(employee == NULL){
18                  printf("Unable to allocate Storage");
19                  exit(1);
20          }
21
22          printf("Enter the Employess's name:");
23          fgets(buffer, 32, stdin);
24
25          /* Allocating storage for the pointer member of the pointer
                 Structure, which is named 'name'*/
26          employee -> name = (char *)malloc( sizeof(buffer) );
27          if (employee -> name == NULL){
28                  printf("Unable to allocate Storage");
29                  exit(1);
30          }
31
```

```
32          /* Copying the buffer to the name: */
33          strcpy(employee->name, buffer);
34          /* Getting he age: */
35          printf("Enter your age: ");
36          scanf("%d", &employee->age);
37
38          printf("Employee's name: %s", employee->name);
39          printf("Employee's age: %d \n", employee->age);
40
41  };
```

Compile Results:

```
1  Enter the Employess's name:Reza
2  Enter your age: 22
3  Employee's name: Reza
4  Employee's age: 22
```

2.7 initialization of the struct in depth

2.7.1 Manual Assignment

2.7.2 Using memset

2.7.3 Designated Initializers

2.8 Function Pointer

function pointer will save the memory address of an function Code:

```
1  #include <stdio.h>
2
3  void function(int x)
4  {
5          printf("X: %d \n", x);
6  }
7
8  int main(){
9
10         void (*function_pointer)(int);
11
12         function_pointer = &function;
13
14         (*function_pointer)(4);
15
16         return(0);
17  }
```

Compile Results:

```
1 #include <stdio.h>
2
3 double add(double x, double y){
4        return x + y;
5 }
6
7 int main(){
8
9        double (*add_function_ptr) (double, double);
10
11        add_function_ptr = add;
12
13        double result = add_function_ptr(20, 10);
14
15        printf("result: %f \n", result);
16
17        return 0;
18 }
```

2.8.1   Array of pointers to functions

```
#include <stdio.h>

double add(int x, int y){
       return x + y;
}

double subtract(int x, int y){
       return x - y;
}

double multiply(int x, int y){
       return x * y;
}

double divide(int x, int y){
       return x / y;
}


int main(){
       double (*opration_array[]) (int, int) = {add, subtract,
             multiply, divide};

       double result = (*opration_array[2])(3, 20);

```

```
25          printf("result: %f \n", result);
26
27  }
```

```
1  result: 60.000000
```

2.8.2   A Function That Return a Pointer To a Function:

```
1  #include <stdio.h>
2
3  double add(int x, int y){
4          return x + y;
5  }
6
7  double subtract(int x, int y){
8          return x - y;
9  }
10
11  double multiply(int x, int y){
12          return x * y;
13  }
14
15  double divide(int x, int y){
16          return x / y;
17  }
18
19
20  void ( *select_operation() ) {
21          int option = 0;
22          printf("Select an operation: \n");
23          printf("Enter 1 for Subtracting\n");
24          printf("Enter 2 for adding\n");
25          printf("Enter 3 for multipling\n");
26          printf("Enter 4 for dividing\n");
27
28          scanf("%d", &option);
29
30          switch (option){
31                  case 1: return subtract;
32                  case 2: return add;
33                  case 3: return multiply;
34                  case 4: return divide;
35                  default: return NULL;
36
37          }
38
39  }
40
```

```
41  int main(){
42
43          double (*operation) (int, int) = select_operation();
44          printf("result: %f \n", operation(20, 5));
45  }
```

Compile Result:

```
1  Select an operation:
2  Enter 1 for Subtracting
3  Enter 2 for adding
4  Enter 3 for multipling
5  Enter 4 for dividing
6  3
7  result: 100.000000
```

2.8.3   A Function That Get a Pointer To a Function As an Input:

```
1   #include <stdio.h>
2   #include <stdbool.h>
3
4   bool temperature_check_in_celsius(int temperature){
5           if (temperature <= 0) return true;
6           else return false;
7   }
8
9
10  bool temperature_check_in_fahrenheit(int temperature){
11          if (temperature <= 32) return true;
12          else return false;
13  }
14
15
16  void is_forzen ( bool (*temperature_check) (int) ){
17          int temperature;
18          printf("Enter the temperature: ");
19          scanf("%d", &temperature);
20
21          if(temperature_check(temperature)){
22                  printf("It's frozen");
23          }
24          else{
25                  printf("It's NOT frozen! \n");
26          }
27  }
28
29
```

```
30   int main(){
31          printf("Celsius \n");
32          is_forzen(temperature_check_in_celsius);
33
34          printf("Fahrenheit \n");
35          is_forzen(temperature_check_in_fahrenheit);
36
37   }
```

Compile Result:

```
1   Celsius
2   Enter the temperature: 10
3   It's NOT frozen!
4   Fahrenheit
5   Enter the temperature: 10
6   It's frozen
```

2.9   Exercises

Chapter 3

Simulating Objects and Classes in C

3.1 Concepts

- **Setter (Set the Value)** and **Getters (Get the Values) Methods**: These are essential OOP concepts for setting and retrieving an object's attribute values.

- The program illustrates OOP principles in C, a language without native OOP support, by utilizing structures and function pointers.

- A structure named 'Person' is defined to represent an object with an 'age' attribute, akin to a class in OOP languages.

- Function pointers 'set' and 'get' are incorporated into the 'Person' structure to simulate methods for setting and getting the 'age'.

- Two functions, 'setAge' and 'getAge', are defined to manipulate the 'age' attribute of a 'Person' object.

- In the main function, an instance of the 'Person' structure is instantiated, and the function pointers are assigned to point to 'setAge' and 'getAge'.

- The program employs these function pointers to set and retrieve the age of a 'Person' object, exemplifying encapsulation by concealing the internal state from external access.

```c
#include <stdio.h>

 struct Person {
        int age;
        // In structures we can not define functions
        // But we can Define variables
        // If we can define variable, so we can define pointer
            variables too
        // and if we can define pointer variables, we can define
            pointer functions too
        void (*set)(struct Person *, int);
        int (*get)(struct Person *);
};


void setAge(struct Person * instance, int age){
        instance -> age = age;
};

int getAge(struct Person * instance){
        return(instance-> age);
};

/* Creating Objects: */
int main(){
        struct Person person1;
        // Binding:
        person1.set = setAge;
        person1.get = getAge;

        //Setting the age for the Object person1 to 18:
        person1.set(\&person1, 18);
        printf("The age is: %d \n", person1.get(\&person1));

        return(0);
}
```

3.2   Objects

3.3   Inheritance

3.4   Polymorphism

3.5   Encapsulation

Chapter 4

Implementing simple Data structures in C

4.1 Empty

Chapter 5

Implementing simple Algorithms in C

5.1   implementing simple qsort algorithm

```
1   #include <stdio.h>
2   #include <malloc.h>
3
4
5   struct sorter {
6           int n_item, start, end;
7           int *data;
8   };
9
10  //The function is intended to create and return a pointer to a struct
        sorter
11  struct sorter * sorter_create(int n){
12          int *data;
13          struct sorter *s = malloc(sizeof(struct sorter));
14
15          if (s == NULL){
16                  return NULL;
17          }
18
19          data = malloc(sizeof(int) * n);
20
21          if (s->data == NULL){
22                  free(s);
23                  return NULL;
24          }
25
26          *s = (struct sorter){
27                  .n_item = n,
```

```
28                    .start = 0,
29                    .end   = 0,
30                    .data  = data
31            };
32
33            return s;
34    }
35
36    void sorter_destroy(struct sorter *s){
37
38    }
39
40
41    /*
42     * Adding values to data structure and simultaneously sorting them.
43     */
44    void sorter_add(struct sorter *s, int value){
45
46    }
47
48    int sorter_get(struct sorter *s){
49            return 0;
50    }
51
52
53
54
55    void main(void){
56
57            struct sorter *s;
58            //it is going to sort 32 elements
59            s = sorter_create(32);
60
61            sorter_add(s, 3);
62            sorter_add(s, 1);
63            sorter_add(s, 2);
64
65            for (int i = 0 ; i < 3; i++){
66                    printf("%d \n", sorter_get(s));
67            }
68
69
70
71    }
```

Code Explanation:

1. This function is intended to create and return a pointer to a struct sorter. The parameter n likely represents the number of items or the size of the data array.

2. The sorter_create function is intended to create and initialize a new struct sorter instance. Here is why such a function is useful:

```
struct sorter * sorter_create(int n){
        ...
}
```

   (a) **Encapsulation:** By using a function to create the structure, you encapsulate the creation logic. This makes the code cleaner and easier to maintain.

   (b) **Initialization:** The function can initialize the structure's members (like n_item, start, end, and the data array) to appropriate values, ensuring the structure is in a valid state when it's created.

   (c) **Memory Allocation:** If the data array needs to be dynamically allocated based on the value of n, the function can handle this allocation. This is especially important for the flexible array member data[].

when ever you are implemenitg anything in systeam first define the datastrature, why anything else will come after that..