MODULE IV NOTES

Prepared by

Partha Pratim Paul

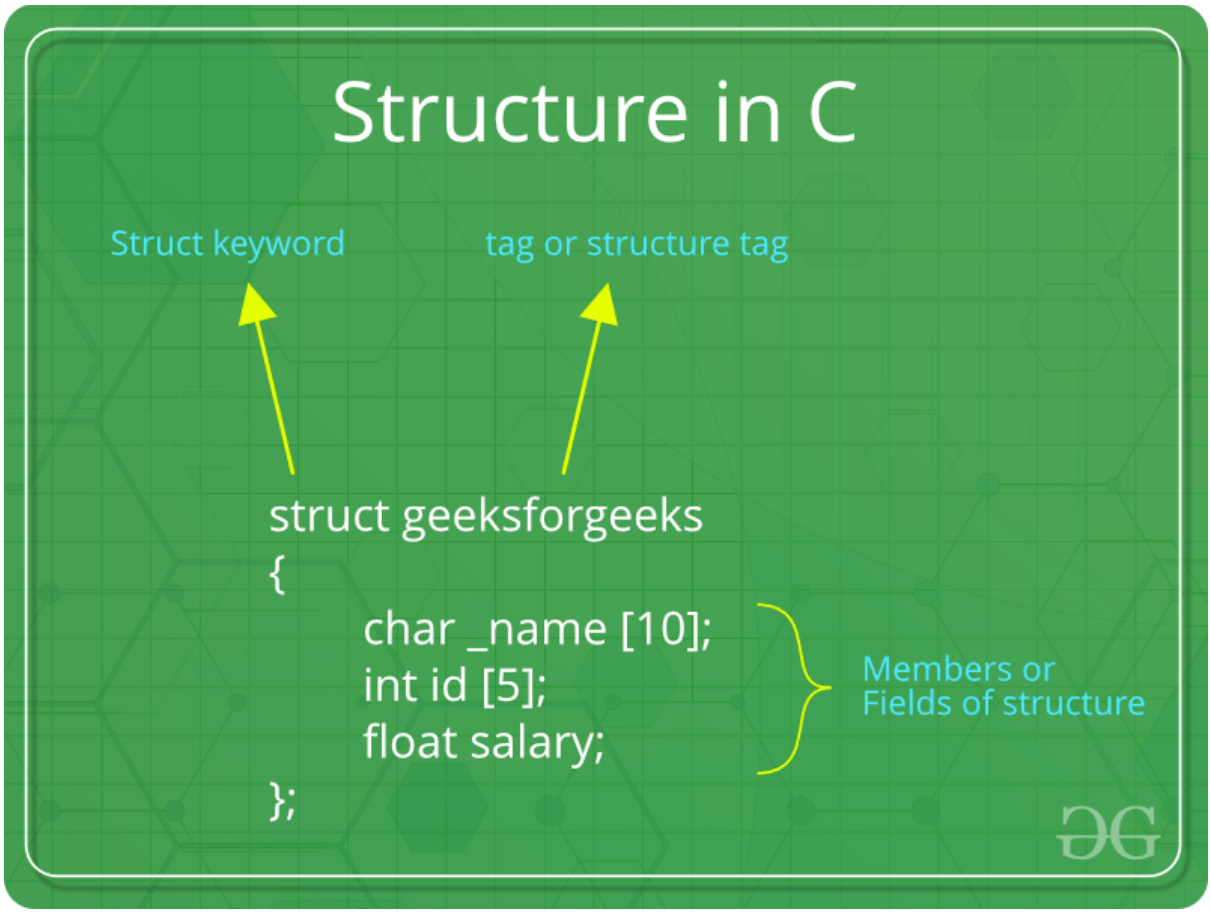
Asst. Prof

CSE

Brainware University

C Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct keyword** is used to define the structure in the C programming language. The items in the structure are called its **member** and they can be of any valid data type.



C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the following syntax:

Syntax

struct structure_name {

data_type member_name1;

data_type member_name1;

....

....

};

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

struct structure_name {

data_type member_name1;

data_type member_name1;

....

....

}***variable1, varaible2, ...***;

2. Structure Variable Declaration after Structure Template

// structure declared beforehand

struct *structure_name* ***variable1, variable2***, ...;

Access Structure Members

We can access structure members by using the **(.) dot operator.**

Syntax

structure_name.member1;

strcuture_name.member2;

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

Initialize Structure Members

Structure members **cannot be** initialized with the declaration. For example, the following C program fails in the compilation.

```
struct Point
{
  int x = 0;  // COMPILER ERROR:  cannot initialize members here
  int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

We can initialize structure members in 3 ways which are as follows:

1. Using Assignment Operator.
2. Using Initializer List.
3. Using Designated Initializer List.

1. Initialization using Assignment Operator

struct *structure_name* str;

str.member1 = value1;

str.member2 = value2;

str.member3 = value3;

.

.

.

2. Initialization using Initializer List

struct *structure_name* str = { value1, value2, value3 };

3. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.

struct *structure_name* str = { .member1 = value1, .member2 = value2, .member3 = value3 };

The Designated Initialization is only supported in C but not in C++.

Example of Structure in C

The following C program shows how to use structures

```
// C program to illustrate the use of structures
#include <stdio.h>

// declaring structure with name str1
struct str1 {
	int i;
	char c;
	float f;
	char s[30];
};

// declaring structure with name str2
struct str2 {
	int ii;
	char cc;
	float ff;
} var; // variable declaration with structure template

// Driver code
int main()
{
	// variable declaration after structure template
	// initialization with initializer list and designated
	// initializer list
	struct str1 var1 = { 1, 'A', 1.00, "GeeksforGeeks" },
					var2;
	struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };

	// copying structure using assignment operator
```

```
    var2 = var1;

    printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var1.i, var1.c, var1.f, var1.s);
    printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var2.i, var2.c, var2.f, var2.s);
    printf("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.ii,
        var3.cc, var3.ff);

    return 0;
}
```

Output

```
Struct 1:
   i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
   i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
   i = 5, c = a, f = 5.000000
```

C Pointers

What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.

datatype ***** ***ptr***;

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

int ***ptr**;

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the **(&) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example

int var = 10;
int * **ptr**;
ptr = **&**var;

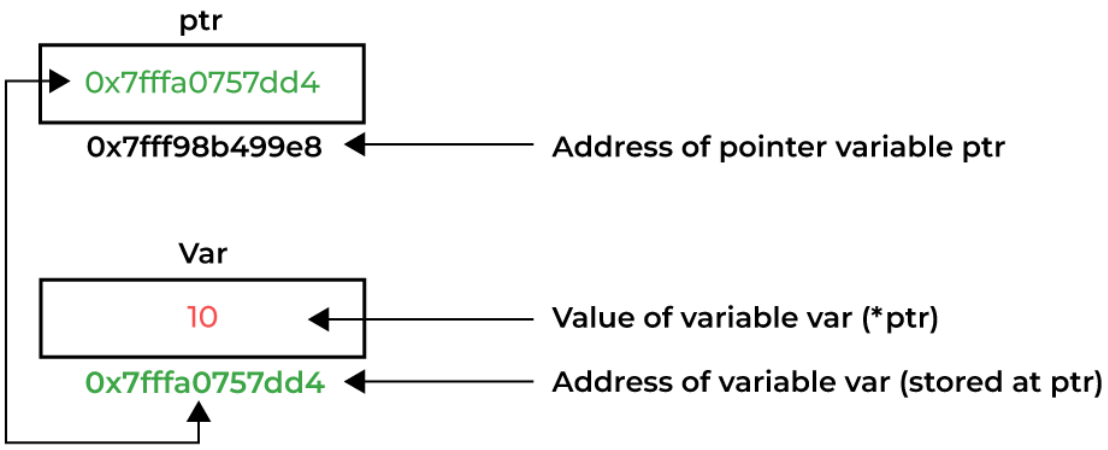
We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

Example

int ***ptr = &**var;

3. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.



// C program to illustrate Pointers

#include <stdio.h>

```
void geeks()
{
	int var = 10;

	// declare pointer variable
	int* ptr;

	// note that data type of ptr and var must be same
	ptr = &var;

	// assign the address of a variable to a pointer
	printf("Value at ptr = %p \n", ptr);
	printf("Value at var = %d \n", var);
	printf("Value at *ptr = %d \n", *ptr);
}

// Driver program
int main()
{
	geeks();
	return 0;
}
```

Output

```
Value at ptr = 0x7fff1038675c
Value at var = 10
Value at *ptr = 10
```

Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax

int *ptr;

These pointers are pronounced as **Pointer to Integer.**

Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as Pointer to Arrays. We can create a pointer to an array using the given syntax.

Syntax

char *ptr = &*array_name*;

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called Structure Pointer or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax

struct struct_name *ptr;

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the function pointer for this function will be

Syntax

int (*ptr)(int, char);

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or pointers-to-pointer. Instead of pointing to a data value, they point to another pointer.

Syntax

datatype ** pointer_name;

Dereferencing Double Pointer

*pointer_name; // get the address stored in the inner level pointer
**pointer_name; // get the value pointed by inner level pointer

6. NULL Pointer

The Null Pointers are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

data_type **pointer_name* = NULL;
or
pointer_name = NULL

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The Void pointers in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax

void * *pointer_name*;

8. Wild Pointers

The Wild Pointers are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

Example

int *ptr;
char *str;

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax

data_type * const *pointer_name*;

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

const *data_type* * *pointer_name*;

Other Types of Pointers in C:

There are also the following types of pointers available to use in C apart from those specified above:

- **Far pointer:** A far pointer is typically 32-bit that can access memory outside the current segment.
- **Dangling pointer:** A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- **Huge pointer:** A huge pointer is 32-bit long containing segment address and offset address.
- **Complex pointer:** Pointers with multiple levels of indirection.
- **Near pointer:** Near pointer is used to store 16-bit addresses means within the current segment on a 16-bit machine.
- **Normalized pointer:** It is a 32-bit pointer, which has as much of its value in the segment register as possible.
- **File Pointer:** The pointer to a FILE data type is called a stream pointer or a file pointer.

Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Disadvantages of Pointers

Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.

- Pointers are majorly responsible for memory leaks in C.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.