9/10/2018

Data Structure

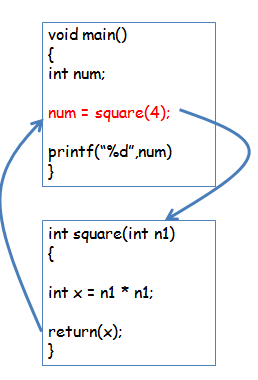
Made By : Mehedi Hasan

**Definition :** In computer science, a data structure is a data organization, management and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.[Wikipedia](https://en.wikipedia.org/wiki/Data_structure)

Before Midterm Exam, we know the Topics . Which is

* **Function in C**
* **Stack**
* **Queue**
* **Linked List**
* **Array**
* **Structure**

**Function in C** : A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. The C standard library provides numerous built-in functions that your program can call.



A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

**Defining a Function :**

The general form of a function definition in C programming language is as follows –

return\_type function\_name( parameter list ) {

body of the function

}

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

* Return Type − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.
* Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* Function Body − The function body contains a collection of statements that define what the function does.

**Example :**

Given below is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum value between the two –

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

**Function Declarations :**

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

**Calling a Function :**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –

Max value is : 200

**Function Arguments :**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function –

|  |  |
| --- | --- |
| **Sr.No.** | **Call Type & Description** |
| 1 | [**Call by value**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [**Call by reference**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

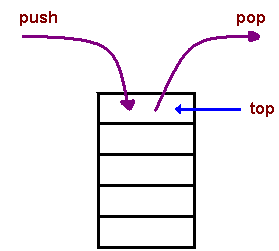
By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

[reference](https://www.tutorialspoint.com/)

**Stack :** A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a recursive data structure. Here is a structural definition of a Stack:

a stack is either empty or it consistes of a top and the rest which is a stack;



* Applicationsb : The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
* Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

**Backtracking :** This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

* Language processing:
* space for parameters and local variables is created internally using a stack.
* compiler's syntax check for matching braces is implemented by using stack.
* support for recursion.

**Implementation :** In the standard library of classes, the data type stack is an adapter class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:

public interface StackInterface<AnyType>

{

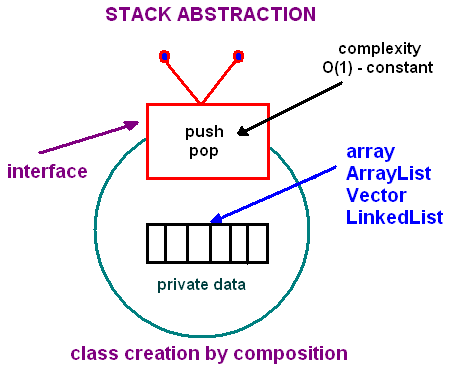
public void push(AnyType e);

public AnyType pop();

public AnyType peek();

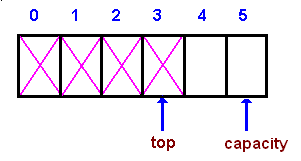
public boolean isEmpty();

}

The following picture demonstrates the idea of implementation by composition.

Another implementation requirement (in addition to the above interface) is that all stack operations must run in constant time O(1). Constant time means that there is some constant k such that an operation takes k nanoseconds of computational time regardless of the stack size.

**Array-based implementation :**

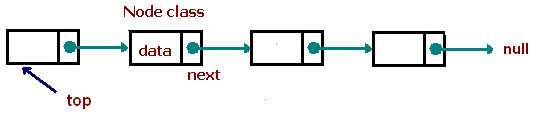


In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable top that refers to the top element in the stack and the capacity that refers to the array size. The variable top changes from -1 to capacity - 1. We say that a stack is empty when top = -1, and the stack is full when top = capacity-1.

In a fixed-size stack abstraction, the capacity stays unchanged, therefore when top reaches capacity, the stack object throws an exception.

In a dynamic stack abstraction when top reaches capacity, we double up the stack size.

**Linked List-based implementation :**

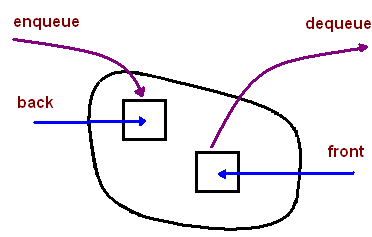


Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.

**Queues :**

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



**Implementation :**

In the standard library of classes, the data type queue is an adapter class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type of the underlying data structure, a queue must implement the same functionality. This is achieved by providing a unique interface.

interface QueueInterface‹AnyType>

{

public boolean isEmpty();

public AnyType getFront();

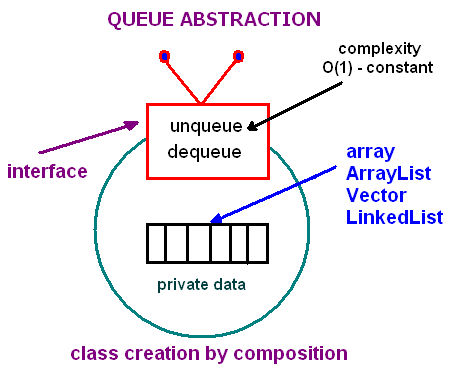
public AnyType dequeue();

public void enqueue(AnyType e);

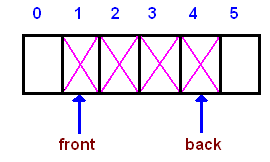
public void clear();

}

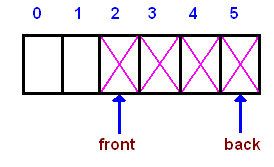
Each of the above basic operations must run at constant time O(1). The following picture demonstrates the idea of implementation by composition.



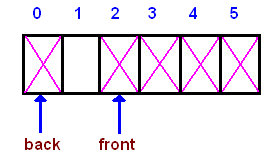
**Circular Queue :** Given an array A of a default size (≥ 1) with two references back and front, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item - we increase the front index. Here is a picture that illustrates the model after a few steps:



As you see from the picture, the queue logically moves in the array from left to right. After several moves back reaches the end, leaving no space for adding new elements.



However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until front. Such a model is called a wrap around queue or a circular queue



Finally, when back reaches front, the queue is full. There are two choices to handle a full queue:a) throw an exception; b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example, 8%5 is 3). By using the modulo operator, we can view the queue as a circular array, where the "wrapped around" can be simulated as "back % array\_size". In addition to the back and front indexes, we maintain another index: cur - for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

[reference](https://www.cs.cmu.edu/)

**Linked lists :**

**Introduction :**

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. However, understanding pointers is crucial to understanding how linked lists work, so if you've skipped the pointers tutorial, you should go back and redo it. You must also be familiar with dynamic memory allocation and structures.

Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array.

**Linked lists have a few advantages over arrays:**

Items can be added or removed from the middle of the list

There is no need to define an initial size

**However, linked lists also have a few disadvantages:**

There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.

Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.

Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

**What is a linked list?**

A linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.

A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.



**Let's define a linked list node:**

typedef struct node {

int val;

struct node \* next;

} node\_t;

Notice that we are defining the struct in a recursive manner, which is possible in C. Let's name our node type node\_t.

Now we can use the nodes. Let's create a local variable which points to the first item of the list (called head).

node\_t \* head = NULL;

head = malloc(sizeof(node\_t));

if (head == NULL) {

return 1;

}

head->val = 1;

head->next = NULL;

This can go on and on, but what we should actually do is advance to the last item of the list, until the next variable will be NULL.

**Iterating over a list**

Let's build a function that prints out all the items of a list. To do this, we need to use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

void print\_list(node\_t \* head) {

node\_t \* current = head;

while (current != NULL) {

printf("%d\n", current->val);

current = current->next;

}

}

**Adding an item to the end of the list :**

To iterate over all the members of the linked list, we use a pointer called current. We set it to start from the head and then in each step, we advance the pointer to the next item in the list, until we reach the last item.

void push(node\_t \* head, int val) {

node\_t \* current = head;

while (current->next != NULL) {

current = current->next;

}

/\* now we can add a new variable \*/

current->next = malloc(sizeof(node\_t));

current->next->val = val;

current->next->next = NULL;

}

The best use cases for linked lists are stacks and queues, which we will now implement:

**Adding an item to the beginning of the list (pushing to the list)**

To add to the beginning of the list, we will need to do the following:

* Create a new item and set its value
* Link the new item to point to the head of the list
* Set the head of the list to be our new item

This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it.

Since we use a function to do this operation, we want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself.

void push(node\_t \*\* head, int val) {

node\_t \* new\_node;

new\_node = malloc(sizeof(node\_t));

new\_node->val = val;

new\_node->next = \*head;

\*head = new\_node;

}

**Removing the first item (popping from the list)**

To pop a variable, we will need to reverse this action:

* Take the next item that the head points to and save it
* Free the head item
* Set the head to be the next item that we've stored on the side

Here is the code:

int pop(node\_t \*\* head) {

int retval = -1;

node\_t \* next\_node = NULL;

if (\*head == NULL) {

return -1;

}

next\_node = (\*head)->next;

retval = (\*head)->val;

free(\*head);

\*head = next\_node;

return retval;

}

**Removing the last item of the list :**

Removing the last item from a list is very similar to adding it to the end of the list, but with one big exception - since we have to change one item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

int remove\_last(node\_t \* head) {

int retval = 0;

/\* if there is only one item in the list, remove it \*/

if (head->next == NULL) {

retval = head->val;

free(head);

return retval;

}

/\* get to the second to last node in the list \*/

node\_t \* current = head;

while (current->next->next != NULL) {

current = current->next;

}

/\* now current points to the second to last item of the list, so let's remove current->next \*/

retval = current->next->val;

free(current->next);

current->next = NULL;

return retval;

}

**Removing a specific item**

To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.

**Here is the algorithm:**

* Iterate to the node before the node we wish to delete
* Save the node we wish to delete in a temporary pointer
* Set the previous node's next pointer to point to the node after the node we wish to delete
* Delete the node using the temporary pointer

There are a few edge cases we need to take care of, so make sure you understand the code.

int remove\_by\_index(node\_t \*\* head, int n) {

int i = 0;

int retval = -1;

node\_t \* current = \*head;

node\_t \* temp\_node = NULL;

if (n == 0) {

return pop(head);

}

for (i = 0; i < n-1; i++) {

if (current->next == NULL) {

return -1;

}

current = current->next;

}

temp\_node = current->next;

retval = temp\_node->val;

current->next = temp\_node->next;

free(temp\_node);

return retval;

}

[reference](https://www.learn-c.org/en/)

**Array :**

Array is the most inportant section in C language. In C language, Array is the collection of Data which is the same type data and also accessing using a common name.Array works on likely dimensional. Array are three type. Which is :

* One dimensional Array.
* Two dimensional Array.
* Dynamic Array or Three dimensional Array.

**One dimensional Array Or 1D Array :**

One Dimensional Array or 1D Array or Single Array is likely as a linear array or a list. Accessing its elements involves a single subscript which can either represent a row or column index. Most important think of array is array index start at 0 not 1.

**For Example :**

#include<stdio.h>

int main ()

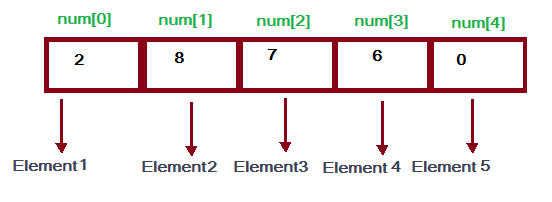
{

int num[5];

num[4] = {2,8,7,6,0};

return 0;

}



Here, num is an array and num is the 5 int type value. num[5] value firstly is 2 than 8 than 7 than 6 and lastly 0. As array count firstly 0 so the num index 0 or num[0] =2.Simillary num[1]=8, num[2]=7, num[3]=6, num[4]=0 it is input until of number of array declaration. num[5] means the here the num array has 5 block and this block also a value and num[5] of 5 is declaration of array size.

**Two dimensional Array Or 2D Array:**

Two Dimensional Array or 2D Array is likely as a Matrix or a table. A matrix can be represented as a table of rows and columns. We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that’s not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration.

**For Example :**

#include<stdio.h>

int main()

{

int A[5][4];

A[5][4]={ {2,8,1,6},

{1,6,5,3},

{3,2,6,4},

{2,9,7,2},

{9,3,1,5}

};

for(i=0; i<5; i++)

{

for(j=0; j<4; j++)

{

printf(“%d”, A[i][j]);

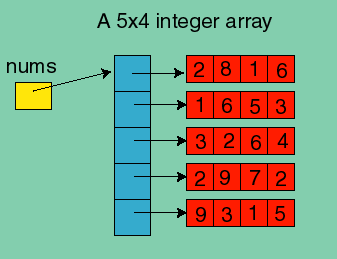
}

}

return 0;

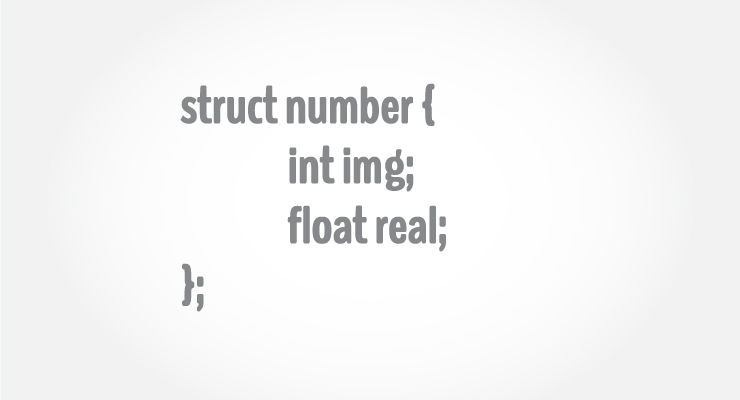
}

**Output is Likely This :**



Here, A is an array and this array also Two Dimensitional Array because of a declaration. Here A [5][4] declaration means that in A array has 5 row and 4 coloum. This array size is 5\*4 = 20 its means than only an array has 20 intager type value.

**Structure :**

****

Structure is a collection of variables of different types under a single name.

**For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

**Structure Definition in C**

Keyword struct is used for creating a structure.

**Syntax of structure**

struct structure\_name

{

data\_type member1;

data\_type member2;

.

.

data\_type memeber;

};

We can create the structure for a person as mentioned above as:

struct person

{

char name[50];

int citNo;

float salary;

};

This declaration above creates the derived data type.

**Structure variable declaration**

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

struct person

{

char name[50];

int citNo;

float salary;

};

int main()

{

struct person person1, person2, person3[20];

return 0;

}

Another way of creating a structure variable is:

struct person

{

char name[50];

int citNo;

float salary;

} person1, person2, person3[20];

In both cases, two variables person1, person2 and an array person3 having 20 elements of type struct person are created.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

* Member operator(.)
* Structure pointer operator(->) (is discussed in structure and pointers tutorial)

Any member of a structure can be accessed as:

structure\_variable\_name.member\_name

Suppose, we want to access salary for variable person2. Then, it can be accessed as:

person2.salary

**Example of structure :**

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)

#include <stdio.h>

struct Distance

{

int feet;

float inch;

} dist1, dist2, sum;

int main()

{

printf("1st distance\n");

// Input of feet for structure variable dist1

printf("Enter feet: ");

scanf("%d", &dist1.feet);

// Input of inch for structure variable dist1

printf("Enter inch: ");

scanf("%f", &dist1.inch);

printf("2nd distance\n");

// Input of feet for structure variable dist2

printf("Enter feet: ");

scanf("%d", &dist2.feet);

// Input of feet for structure variable dist2

printf("Enter inch: ");

scanf("%f", &dist2.inch);

sum.feet = dist1.feet + dist2.feet;

sum.inch = dist1.inch + dist2.inch;

if (sum.inch > 12)

{

//If inch is greater than 12, changing it to feet.

++sum.feet;

sum.inch = sum.inch - 12;

}

// printing sum of distance dist1 and dist2

printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);

return 0;

}

**Output**

1st distance

Enter feet: 12

Enter inch: 7.9

2nd distance

Enter feet: 2

Enter inch: 9.8

Sum of distances = 15'-5.7"

### Keyword typedef while using structure

Writing struct structure\_name variable\_name; to declare a structure variable isn't intuitive as to what it signifies, and takes some considerable amount of development time.

So, developers generally use typedef to name the structure as a whole. For example:

typedef struct complex

{

int imag;

float real;

} comp;

int main()

{

comp comp1, comp2;

}

Here, typedef keyword is used in creating a type comp (which is of type as struct complex).

Then, two structure variables comp1 and comp2 are created by this comp type.

**Structures within structures :**

Structures can be nested within other structures in C programming.

struct complex

{

int imag\_value;

float real\_value;

};

struct number

{

struct complex comp;

int real;

} num1, num2;

Suppose, you want to access imag\_value for num2 structure variable then, following structure member is used.

num2.comp.imag\_value

[reference](https://www.programiz.com/)