Pointer to a Structure in C

We have already learned that a pointer is a variable which points to the address of another variable of any data type like `int`, `char`, `float` etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
struct dog
{
    char name[10];
    char breed[10];
    int age;
    char color[10];
};

struct dog spike;

// declaring a pointer to a structure of type struct dog
struct dog *ptr_dog

```

This declares a pointer `ptr_dog` that can store the address of the variable of type `struct dog`. We can now assign the address of variable `spike` to `ptr_dog` using `&` operator.

```
ptr_dog = &spike;
```

Now `ptr_dog` points to the structure variable `spike`.

Accessing members using Pointer

There are two ways of accessing members of structure using pointer:

1. Using indirection (`*`) operator and dot (`.`) operator.
2. Using arrow (`->`) operator or membership operator.

Let's start with the first one.

Using Indirection (*) Operator and Dot (.) Operator

At this point `ptr_dog` points to the structure variable `spike`, so by dereferencing it we will get the contents of the `spike`. This means `spike` and `*ptr_dog` are functionally

equivalent. To access a member of structure write `*ptr_dog` followed by a dot(`.`) operator, followed by the name of the member. For example:

`(*ptr_dog).name` - refers to the `name` of dog
`(*ptr_dog).breed` - refers to the `breed` of dog

and so on.

Parentheses around `*ptr_dog` are necessary because the precedence of dot(`.`) operator is greater than that of indirection (`*`) operator.

Using arrow operator (->)

The above method of accessing members of the structure using pointers is slightly confusing and less readable, that's why C provides another way to access members using the arrow (`->`) operator. To access members using arrow (`->`) operator write pointer variable followed by `->` operator, followed by name of the member.

```
ptr_dog->name   // refers to the name of dog
ptr_dog->breed  // refers to the breed of dog

```

and so on.

Here we don't need parentheses, asterisk (`*`) and dot (`.`) operator. This method is much more readable and intuitive.

We can also modify the value of members using pointer notation.

```
strcpy(ptr_dog->name, "new_name");
```

Here we know that the name of the array (`ptr_dog->name`) is a constant pointer and points to the 0th element of the array. So we can't assign a new string to it using assignment operator (`=`), that's why `strcpy()` function is used.

```
--ptr_dog->age;
```

In the above expression precedence of arrow operator (`->`) is greater than that of prefix decrement operator (`--`), so first `->` operator is applied in the expression then its value is decremented by 1.

The following program demonstrates how we can use a pointer to structure.

```
#include<stdio.h>

struct dog
{
```

```
    char name[10];
    char breed[10];
    int age;
    char color[10];
};

int main()
{
    struct dog my_dog = {"tyke", "Bulldog", 5, "white"};
    struct dog *ptr_dog;
    ptr_dog = &my_dog;

    printf("Dog's name: %s\n", ptr_dog->name);
    printf("Dog's breed: %s\n", ptr_dog->breed);
    printf("Dog's age: %d\n", ptr_dog->age);
    printf("Dog's color: %s\n", ptr_dog->color);

    // changing the name of dog from tyke to jack
    strcpy(ptr_dog->name, "jack");

    // increasing age of dog by 1 year
    ptr_dog->age++;

    printf("Dog's new name is: %s\n", ptr_dog->name);
    printf("Dog's age is: %d\n", ptr_dog->age);

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Dog's name: tyke
Dog's breed: Bulldog
Dog's age: 5
Dog's color: white

After changes

Dog's new name is: jack
Dog's age is: 6

```

How it works:

In lines 3-9, we have declared a structure of type dog which has four members namely `name`, `breed`, `age` and `color`.

In line 13, a variable called `my_dog` of type `struct dog` is declared and initialized.

In line 14, a pointer variable `ptr_dog` of type `struct dog` is declared.

In line 15, the address of `my_dog` is assigned to `ptr_dog` using `&` operator.

In lines 17-20, the `printf()` statements prints the details of the dog.

In line 23, a new name is assigned to `ptr_dog` using the `strcpy()` function, because we can't assign a string value directly to `ptr_dog->name` using assignment operator.

In line 26, the value of `ptr_dog->age` is incremented by `1` using postfix increment operator. Recall that postfix `++` operator and `->` have the same precedence and associates from left to right. But since postfix `++` is used in the expression first the value of `ptr_dog->age` is used in the expression then it's value is incremented by `1`.

Pointers as Structure Member in C

We can also have a pointer as a member of the structure. For example:

```
struct test
{
    char name[20];
    int *ptr_mem;
};

struct test t1, *str_ptr = &t1;

```

Here `ptr_mem` is a pointer to `int` and a member of structure `test`.

There are two ways in which we can access the value (i.e address) of `ptr_mem`:

1. Using structure variable - `t1.ptr_mem`
2. Using pointer variable - `str_ptr->ptr_mem`

Similarly, there are two ways in which we can access the value pointed to by `ptr_mem`.

1. Using structure variable - `*t1.ptr_mem`
2. Using pointer variable - `*str_ptr->ptr_mem`

Since the precedence of dot(`.`) operator is greater than that of indirection(`*`) operator, so in the expression `*t1.ptr_mem` the dot(`.`) is applied before the indirection(`*`) operator. Similarly in the expression `*str_ptr->ptr_mem`, the arrow (`->`) operator is applied followed by indirection(`*`) operator.

The following program demonstrates everything we have learned so far in this lesson.

```
#include<stdio.h>

struct student
{
    char *name;
    int age;
    char *program;
    char *subjects[5];
};

int main()
{
    struct student stu = {
                                "Lucy",
                                25,
                                "CS",
```

```
                                {"CS-01", "CS-02", "CS-03", "CS-04", "CS-05" }
                            };

    struct student *ptr_stu = &stu;
    int i;

    printf("Accessing members using structure variable: \n\n");

    printf("Name: %s\n", stu.name);
    printf("Age: %d\n", stu.age);
    printf("Program enrolled: %s\n", stu.program);

    for(i = 0; i < 5; i++)
    {
        printf("Subject : %s \n", stu.subjects[i]);
    }

    printf("\n\nAccessing members using pointer variable: \n\n");

    printf("Name: %s\n", ptr_stu->name);
    printf("Age: %d\n", ptr_stu->age);
    printf("Program enrolled: %s\n", ptr_stu->program);

    for(i = 0; i < 5; i++)
    {
        printf("Subject : %s \n", ptr_stu->subjects[i]);
    }

    // signal to operating system program ran fine
    return 0;
}
```

Expected Output:

```
Accessing members using structure variable:

Name: Lucy
Age: 25
Program enrolled: CS
Subject : CS-01
Subject : CS-02
Subject : CS-03
Subject : CS-04
Subject : CS-05
Accessing members using pointer variable:

Name: Lucy
Age: 25
Program enrolled: CS
Subject : CS-01
Subject : CS-02
Subject : CS-03
Subject : CS-04
Subject : CS-05
```

How it works:

In lines 3-9, a structure `student` is declared which have four members namely: `name`, `age`, `program` and `subjects`. The type of members is as follows:

Name	Type
`name`	a pointer to `char`
`age`	`int`
`program`	a pointer to `char`
`subjects`	an array of `5` pointers to `char`

In lines 13-18, a variable `stu` of type `struct student` is declared and initialized. Since `name` and `program` are pointers to char so we can directly assign string literals to them. Similarly, `subjects` is an array of 5 pointers to char, so it can hold 5 string literals.

In line 20, a pointer variable `ptr_stu` of type `struct student` is declared and assigned the address of `stu` using `&` operator.

From lines 25-27, three `printf()` statement is used to print `name`, `age` and `program` using structure variable `stu`.

In lines 29-32, a for loop is used to loop through all the elements of an array of pointers `*subjects[5]`. And print the names of the subjects using structure variable.

From lines 36-38, three `printf()` statement is used to print `name`, `age` and `program` using pointer variable `ptr_stu`.

In lines 40-43, a for loop is used to loop through all the elements of an array of pointers `*subjects[5]`. And print the names of the subjects using pointer variable.

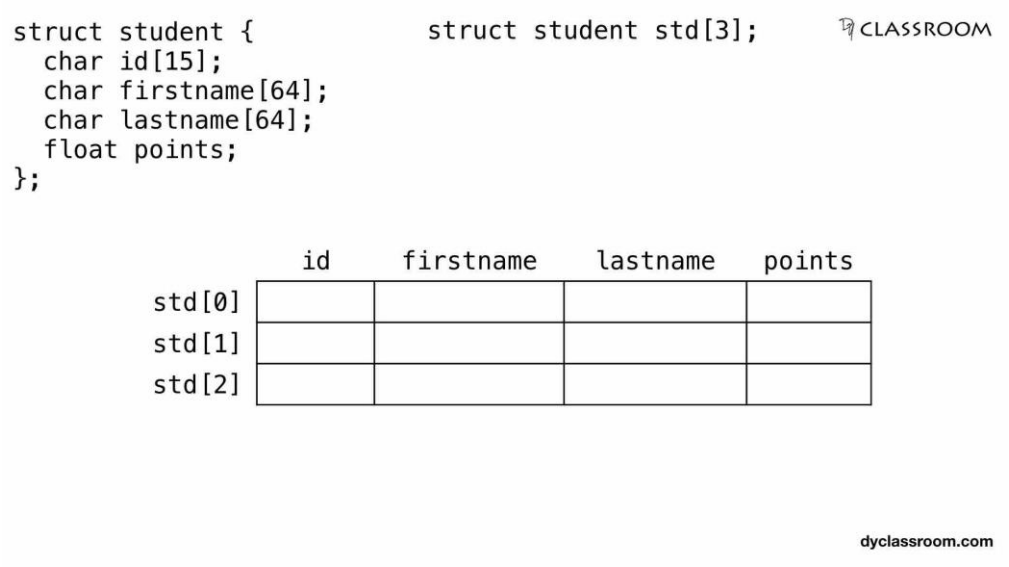
C - Pointers and Array of Structures

Create an array of structure variable

In the following example we are considering the `student` structure that we created in the previous tutorial and we are creating an array of student structure variable `std` of size 3 to hold details of three students.

```
// student structure

struct student {

  char id[15];

  char firstname[64];

  char lastname[64];

  float points;

};



// student structure variable

struct student std[3];
```

We can represent the `std` array variable as following.



Accessing each element of the structure array variable via pointer

For this we will first set the pointer variable `ptr` to point at the starting memory location of `std` variable. For this we write `ptr = std;`.

Then, we can increment the pointer variable using increment operator `ptr++` to make the pointer point at the next element of the structure array variable i.e., from str[0] to str[1].

We will loop three times as there are three students. So, we will increment pointer variable twice. First increment will move pointer `ptr` from std[0] to std[1] and the second increment will move pointer `ptr` from std[1] to std[2].

To reset the pointer variable `ptr` to point at the starting memory location of structure variable `std` we write `ptr = std;`.

Complete code

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

```
int main(void) {


  // student structure
  struct student {
    char id[15];
    char firstname[64];
    char lastname[64];
    float points;
  };


  // student structure variable
  struct student std[3];


  // student structure pointer variable
  struct student *ptr = NULL;


  // other variables
  int i;


  // assign std to ptr
  ptr = std;


  // get detail for user
  for (i = 0; i < 3; i++) {
    printf("Enter detail of student #%d\n", (i + 1));
    printf("Enter ID: ");
```

```
    scanf("%s", ptr->id);
    printf("Enter first name: ");
    scanf("%s", ptr->firstname);
    printf("Enter last name: ");
    scanf("%s", ptr->lastname);
    printf("Enter Points: ");
    scanf("%f", &ptr->points);

    // update pointer to point at next element
    // of the array std
    ptr++;
  }

  // reset pointer back to the starting
  // address of std array
  ptr = std;

  for (i = 0; i < 3; i++) {
    printf("\nDetail of student #%d\n", (i + 1));

    // display result via std variable
    printf("\nResult via std\n");
    printf("ID: %s\n", std[i].id);
    printf("First Name: %s\n", std[i].firstname);
    printf("Last Name: %s\n", std[i].lastname);
    printf("Points: %f\n", std[i].points);
```

```
    // display result via ptr variable
    printf("\nResult via ptr\n");
    printf("ID: %s\n", ptr->id);
    printf("First Name: %s\n", ptr->firstname);
    printf("Last Name: %s\n", ptr->lastname);
    printf("Points: %f\n", ptr->points);

    // update pointer to point at next element
    // of the array std
    ptr++;
  }

  return 0;
}
```

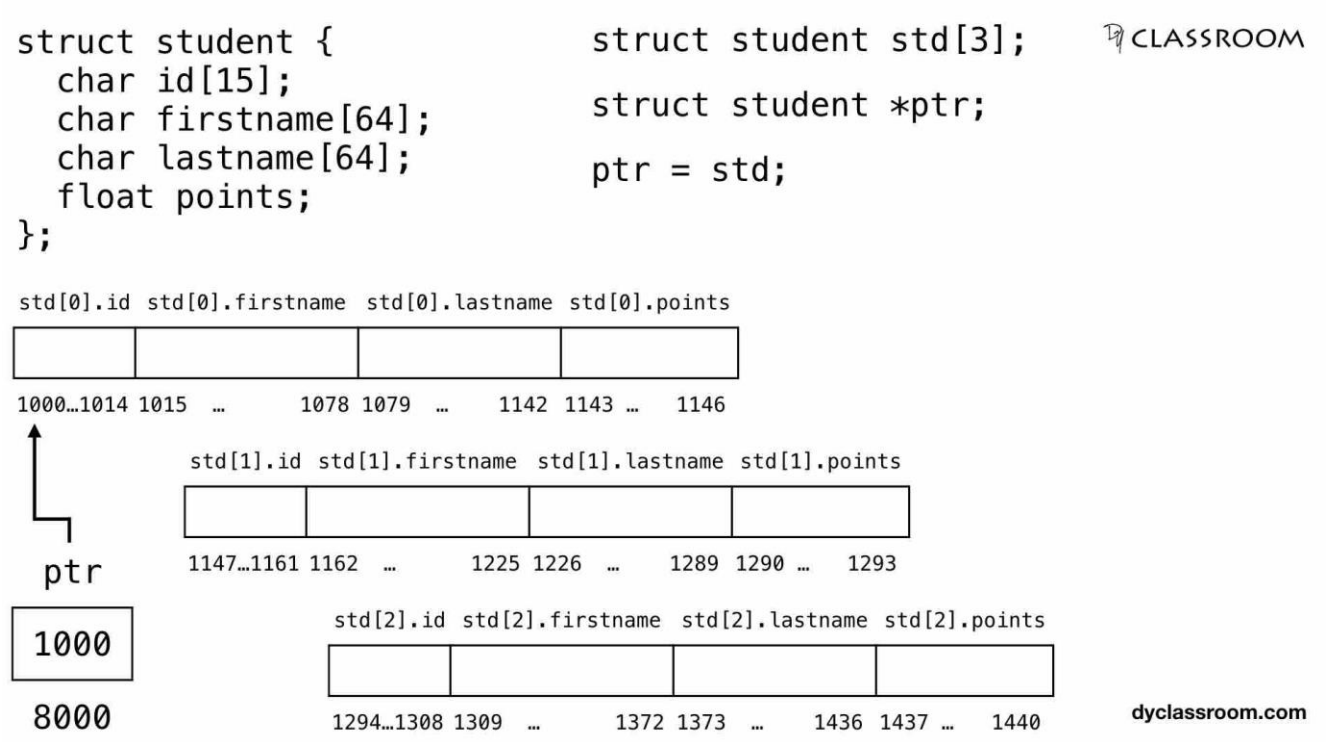
Output:

```
Enter detail of student #1
Enter ID: s01
Enter first name: Yusuf
Enter last name: Shakeel
Enter Points: 8

Enter detail of student #2
Enter ID: s02
Enter first name: Jane
```

```
Enter last name: Doe

Enter Points: 9



Enter detail of student #3

Enter ID: s03

Enter first name: John

Enter last name: Doe

Enter Points: 6



Detail of student #1



Result via std

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000



Result via ptr

ID: s01

First Name: Yusuf

Last Name: Shakeel

Points: 8.000000



Detail of student #2



Result via std
```

```
ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000


Result via ptr

ID: s02

First Name: Jane

Last Name: Doe

Points: 9.000000


Detail of student #3


Result via std

ID: s03

First Name: John

Last Name: Doe

Points: 6.000000


Result via ptr

ID: s03

First Name: John

Last Name: Doe

Points: 6.000000
```

We can represent the `std` array variable in memory as follows.



Points to note!

Each student data takes 147 bytes of memory.

Member	**Data Type**	**Size**
id	char	15 bytes
firstname	char	64 bytes
lastname	char	64 bytes
points	float	4 bytes

And the array size is 3 so, total 147x3 i.e., 441 bytes is allocated to the `std` array variable.

The first element `std[0]` gets the memory location from 1000 to 1146.

The second element `std[1]` gets the memory location from 1147 to 1293.

And the third element `std[2]` gets the memory location from 1294 to 1440.

We start by first making the `ptr` pointer variable point at address 1000 which is the starting address of the first element `std[0]`.

Then moving forward we increment the pointer `ptr++` so, it points at the memory location 1147 i.e., the starting memory location of second element `std[1]`.

Similarly, in the next run we point `ptr` at memory location 1294 i.e., starting location of third element `std[2]`.

To access the members of the structure via pointer we use the `->` arrow operator.