Introduction to Programming

Week – *8*, Lecture – *2*

Structures in C – Part 1

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

Collection of variables… of the same type

We already know how to create a collection of variables of the *same* type

- We can use arrays for the same

Collection of variables… of the same type

We already know how to create a collection of variables of the *same* type

- We can use arrays for the same

We also know how to access each variable in the collection

- We use an index – that starts from `0` and goes up to `size-1`

Collection of variables… of the same type

We already know how to create a collection of variables of the *same* type

- We can use arrays for the same

We also know how to access each variable in the collection

- We use an index – that starts from `0` and goes up to `size-1`

We also saw how an array can be allocated dynamically

- Using an appropriate pointer variable along with the `malloc()` function

Collection of variables... of the same type

We already know how to create a collection of variables of the *same* type
- We can use arrays for the same

We also know how to access each variable in the collection
- We use an index – that starts from `0` and goes up to `size-1`

We also saw how an array can be allocated dynamically
- Using an appropriate pointer variable along with the `malloc()` function

However, what if we need a collection of variables of *different* type?

Collection of variables… of the same type

We already know how to create a collection of variables of the *same* type

- We can use arrays for the same

We also know how to access each variable in the collection

- We use an index – that starts from `0` and goes up to `size-1`

We also saw how an array can be allocated dynamically

- Using an appropriate pointer variable along with the `malloc()` function

However, what if we need a collection of variables of *different* type?

We will now discuss a solution to that problem – structures !!

What are structures?

Structures are used to create collections of variables of different types

- For instance, a structure can contain `x` integers, `y` floats, `z` characters etc.

What are structures?

Structures are used to create collections of variables of different types

- For instance, a structure can contain `x` integers, `y` floats, `z` characters etc.

However, structures are fairly different from arrays

- Before using a structure, we must define a *template*

What are structures?

Structures are used to create collections of variables of different types

- For instance, a structure can contain `x` integers, `y` floats, `z` characters etc.

However, structures are fairly different from arrays

- Before using a structure, we must define a *template*
- This template provides the exact details of the variables that the structure contains, e.g.

```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

What are structures?

Structures are used to create collections of variables of different types

- For instance, a structure can contain `x` integers, `y` floats, `z` characters etc.

However, structures are fairly different from arrays

- Before using a structure, we must define a *template*
- This template provides the exact details of the variables that the structure contains, e.g.

```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

- The above structure is called `sample_structure`, and it contains *three member variables*

What are structures?

Structures are used to create collections of variables of different types

- For instance, a structure can contain `x` integers, `y` floats, `z` characters etc.

However, structures are fairly different from arrays

- Before using a structure, we must define a *template*
- This template provides the exact details of the variables that the structure contains, e.g.

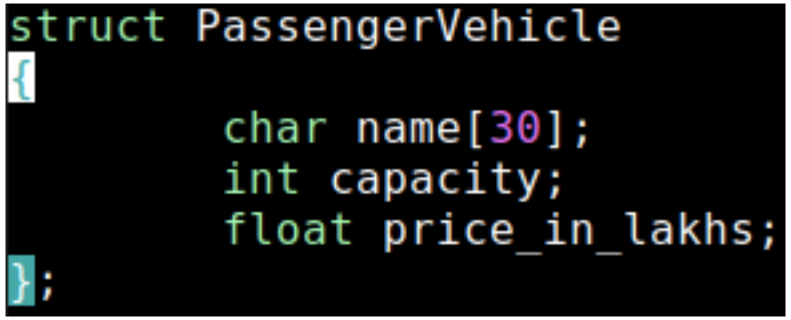
```
struct sample_structure
{
        int i;
        float f;
        char c;
};
```

- The above structure is called `sample_structure`, and it contains *three member variables*

We can then create instances of `sample_structure`, just like we do it for any other types, e.g.

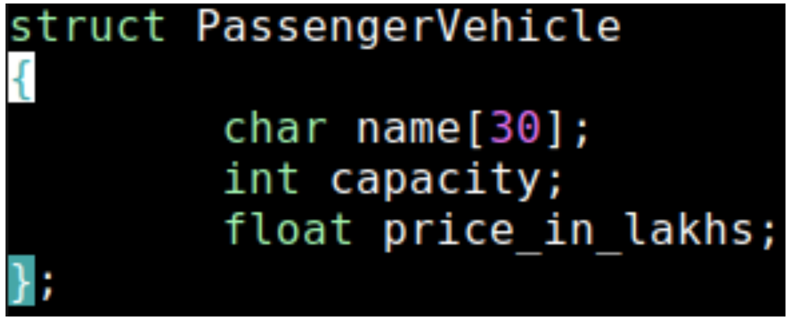
```
struct sample_structure s1, s2;
```

Example – The PassengerVehicle



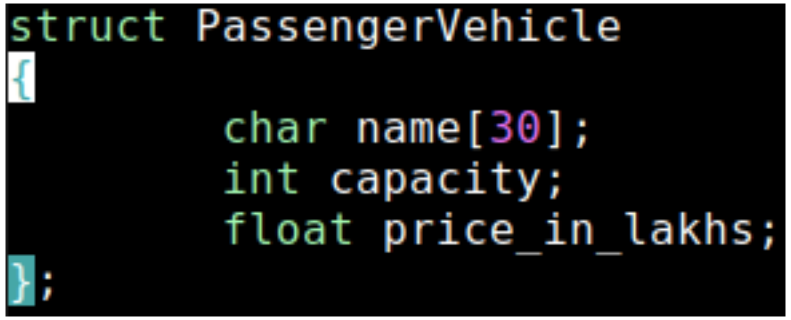
```
struct PassengerVehicle
{
        char name[30];
        int capacity;
        float price_in_lakhs;
};
```

Example – The PassengerVehicle


```
struct PassengerVehicle
{
        char name[30];
        int capacity;
        float price_in_lakhs;
};
```

Here, we declare a structure template called `PassengerVehicle`

Example – The `PassengerVehicle`

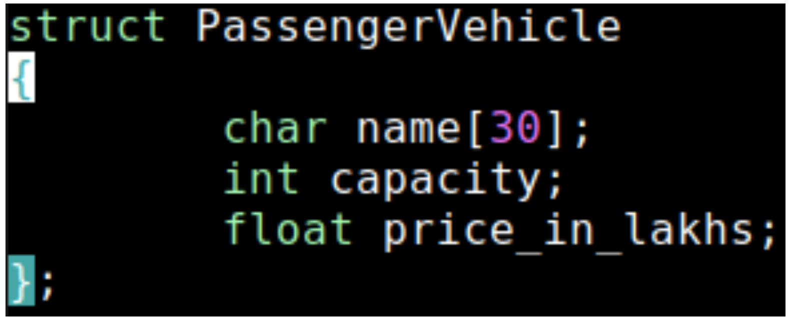


```
struct PassengerVehicle
{
        char name[30];
        int capacity;
        float price_in_lakhs;
};
```

Here, we declare a structure template called `PassengerVehicle`

It has three member variables, with different data types

Example – The `PassengerVehicle`



```
struct PassengerVehicle
{
        char name[30];
        int capacity;
        float price_in_lakhs;
};
```

Here, we declare a structure template called `PassengerVehicle`

It has three member variables, with different data types

We keep this declaration in a header file called `Car.h`, so that we can use this structure in multiple programs

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

To access a particular member variable, we use the dot (**.**) operator…

- … placed between the structure variable, and the member variable

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

To access a particular member variable, we use the dot (.) operator…

- … placed between the structure variable, and the member variable, e.g.
  ```
  s1.i = 5;
  s1.f = 4.5;
  s1.c = 'c';
  ```

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

To access a particular member variable, we use the dot (.) operator…

- … placed between the structure variable, and the member variable, e.g.

```
s1.i = 5;
s1.f = 4.5;
s1.c = 'c';
```

The operations that you can and cannot do with member variables, depend on their types…

- … which in turn, is the same as what you can do with that data type in general

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

To access a particular member variable, we use the dot (.) operator…

- … placed between the structure variable, and the member variable, e.g.
```
s1.i = 5;
s1.f = 4.5;
s1.c = 'c';
```

The operations that you can and cannot do with member variables, depend on their types…

- … which in turn, is the same as what you can do with that data type in general
- Essentially, member variables have the same properties as any other variable of the same type

Accessing member variables

When we create an instance of a structure, contiguous memory is allocated…

- … to store all the member variables of the structure
- The members appear in the allotted space, in the order they are declared in the structure declaration

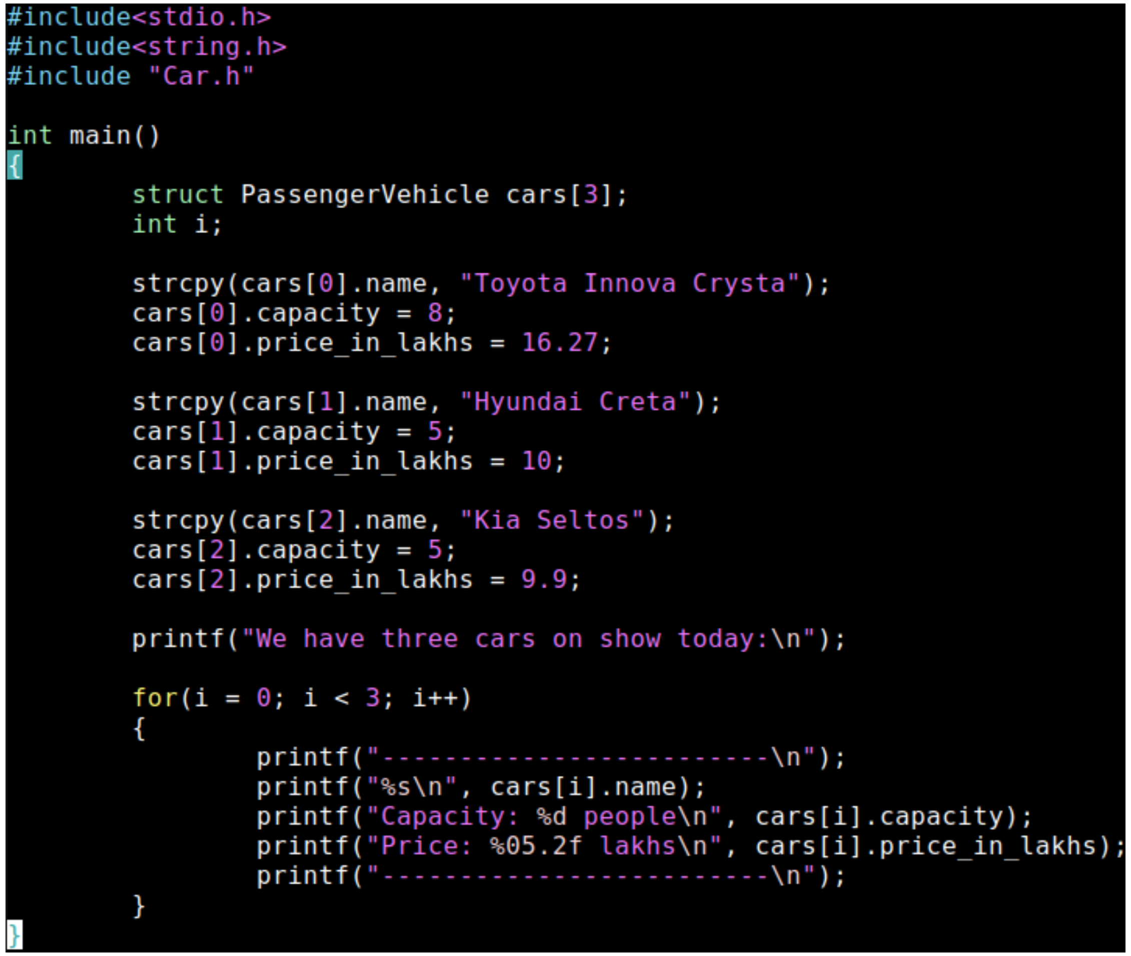
To access a particular member variable, we use the dot (`.`) operator…

- … placed between the structure variable, and the member variable, e.g.
  ```
  s1.i = 5;
  s1.f = 4.5;
  s1.c = 'c';
  ```

The operations that you can and cannot do with member variables, depend on their types…

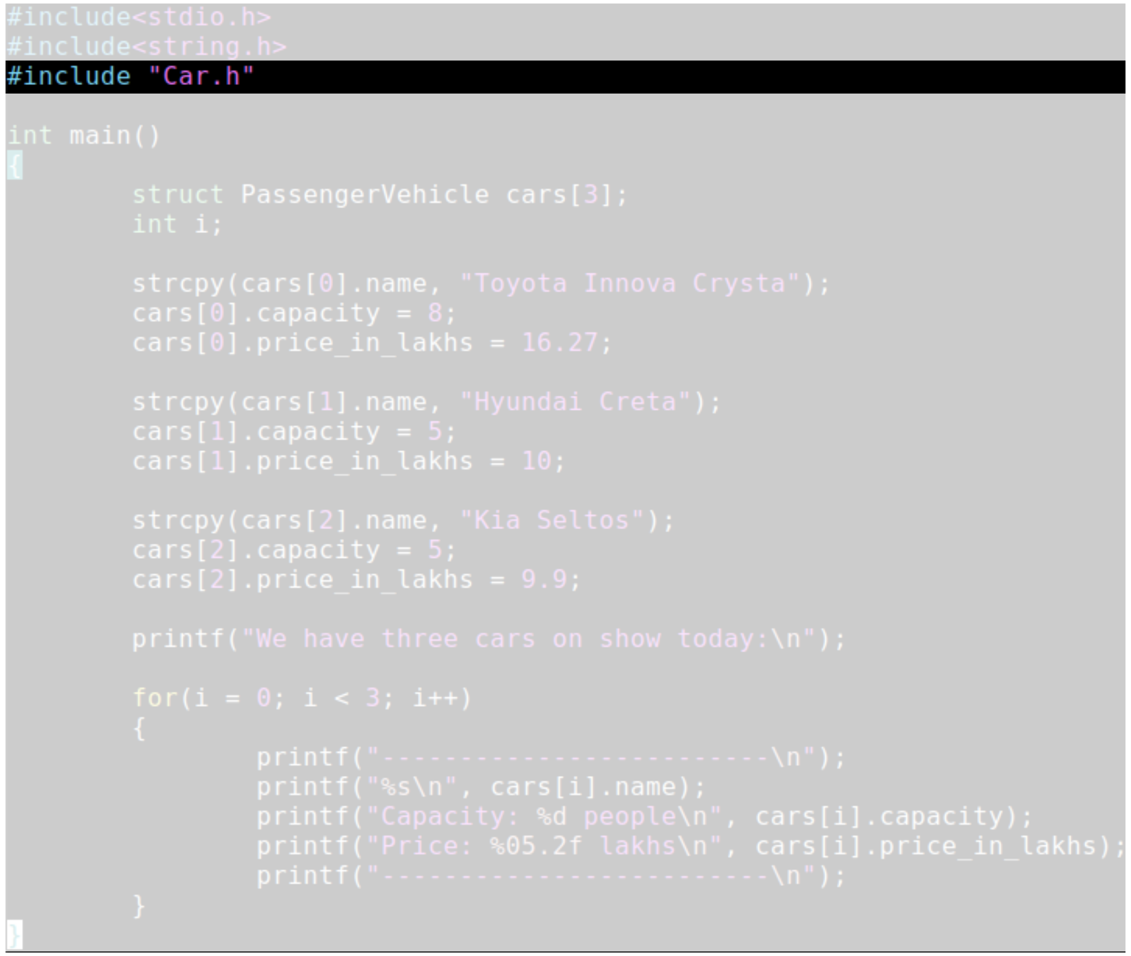
- … which in turn, is the same as what you can do with that data type in general
- Essentially, member variables have the same properties as any other variable of the same type
- For example, `s1.i` can be used in every context where an `int` variable may appear

Example – Creating some simple instances



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

int main()
{
        struct PassengerVehicle cars[3];
        int i;

        strcpy(cars[0].name, "Toyota Innova Crysta");
        cars[0].capacity = 8;
        cars[0].price_in_lakhs = 16.27;

        strcpy(cars[1].name, "Hyundai Creta");
        cars[1].capacity = 5;
        cars[1].price_in_lakhs = 10;

        strcpy(cars[2].name, "Kia Seltos");
        cars[2].capacity = 5;
        cars[2].price_in_lakhs = 9.9;

        printf("We have three cars on show today:\n");

        for(i = 0; i < 3; i++)
        {
                printf("-------------------------\n");
                printf("%s\n", cars[i].name);
                printf("Capacity: %d people\n", cars[i].capacity);
                printf("Price: %05.2f lakhs\n", cars[i].price_in_lakhs);
                printf("-------------------------\n");
        }
}
```

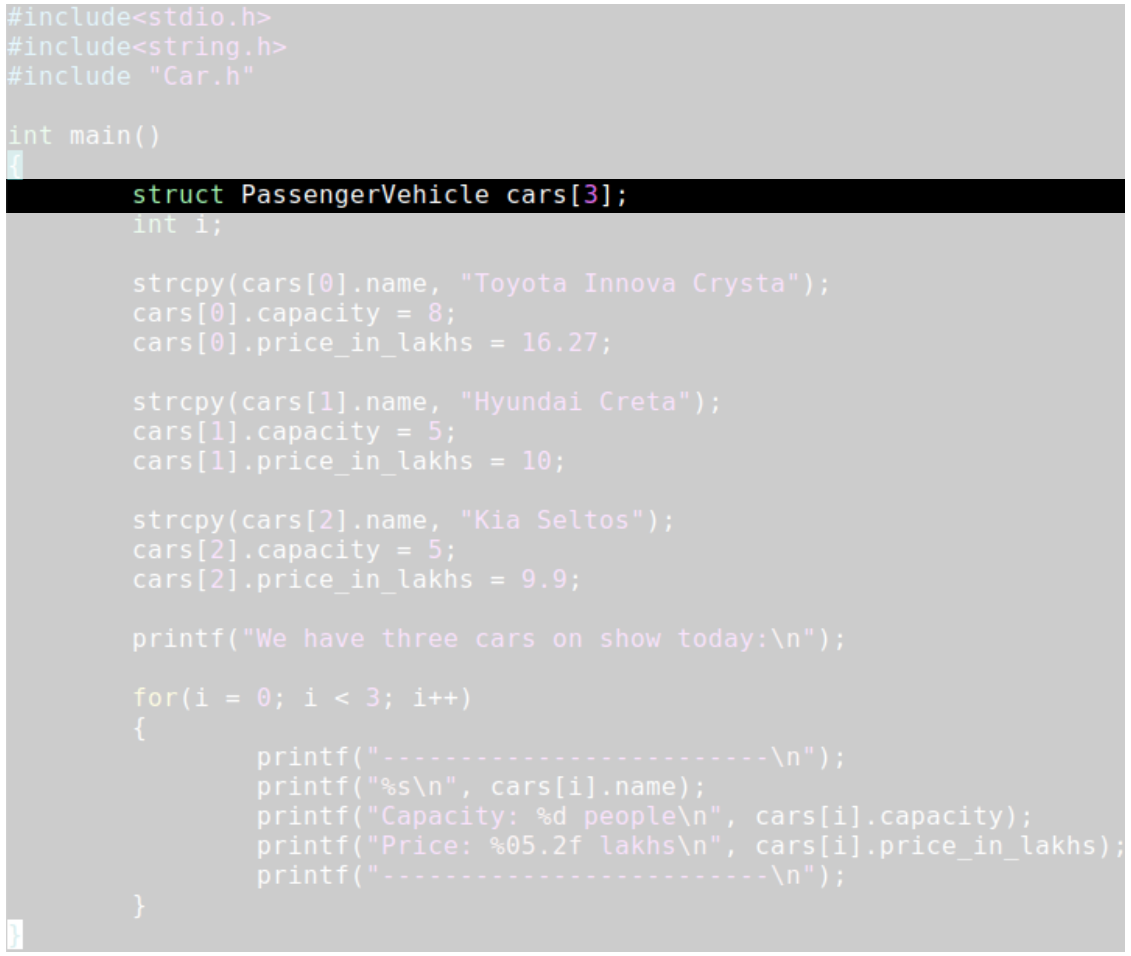
Example – Creating some simple instances



```
#include<stdio.h>
#include<string.h>
#include "Car.h"

int main()
{
        struct PassengerVehicle cars[3];
        int i;

        strcpy(cars[0].name, "Toyota Innova Crysta");
        cars[0].capacity = 8;
        cars[0].price_in_lakhs = 16.27;

        strcpy(cars[1].name, "Hyundai Creta");
        cars[1].capacity = 5;
        cars[1].price_in_lakhs = 10;

        strcpy(cars[2].name, "Kia Seltos");
        cars[2].capacity = 5;
        cars[2].price_in_lakhs = 9.9;

        printf("We have three cars on show today:\n");

        for(i = 0; i < 3; i++)
        {
                printf("-------------------------\n");
                printf("%s\n", cars[i].name);
                printf("Capacity: %d people\n", cars[i].capacity);
                printf("Price: %05.2f lakhs\n", cars[i].price_in_lakhs);
                printf("-------------------------\n");
        }
}
```

Including the header file, allows us to create instances of the `PassengerVehicle`

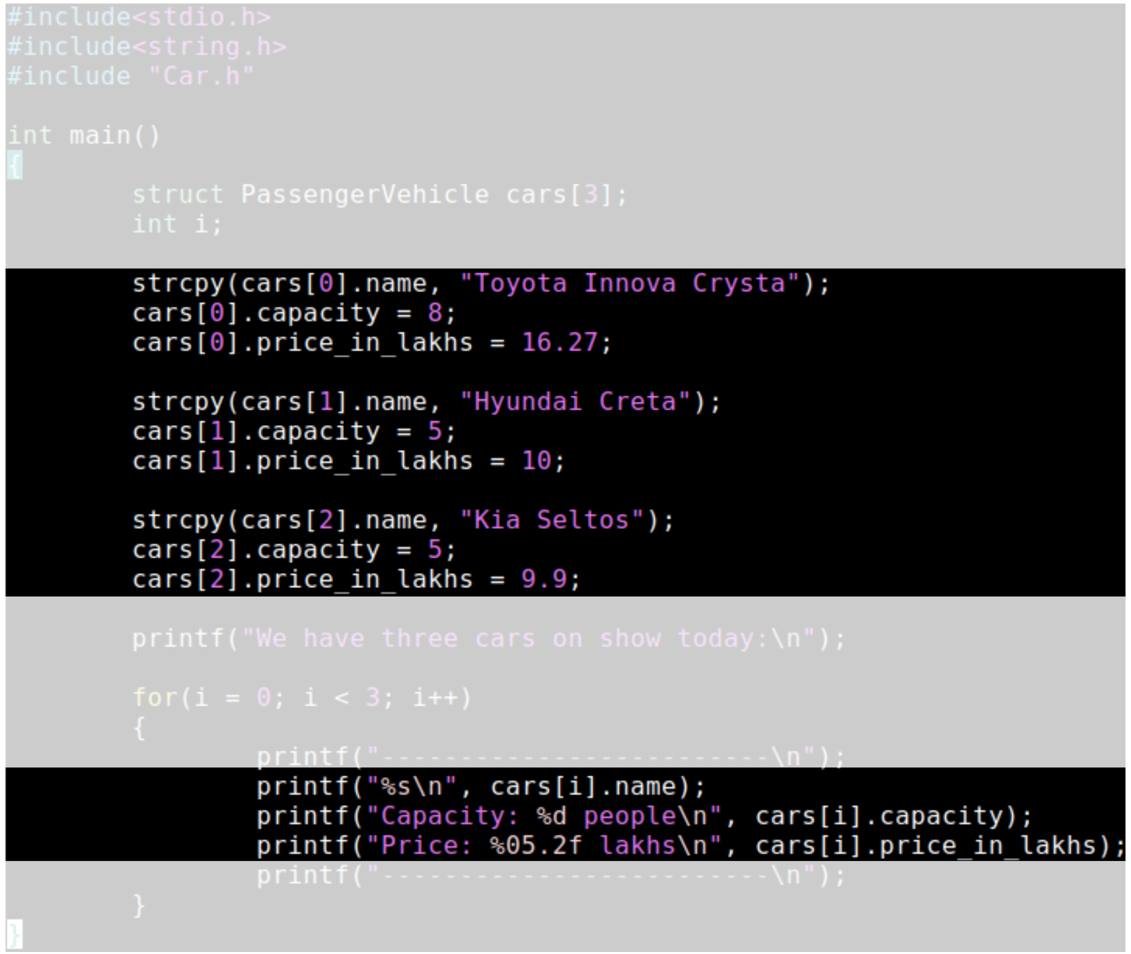
Example – Creating some simple instances



```
#include<stdio.h>
#include<string.h>
#include "Car.h"

int main()
{
        struct PassengerVehicle cars[3];
        int i;

        strcpy(cars[0].name, "Toyota Innova Crysta");
        cars[0].capacity = 8;
        cars[0].price_in_lakhs = 16.27;

        strcpy(cars[1].name, "Hyundai Creta");
        cars[1].capacity = 5;
        cars[1].price_in_lakhs = 10;

        strcpy(cars[2].name, "Kia Seltos");
        cars[2].capacity = 5;
        cars[2].price_in_lakhs = 9.9;

        printf("We have three cars on show today:\n");

        for(i = 0; i < 3; i++)
        {
                printf("-------------------------\n");
                printf("%s\n", cars[i].name);
                printf("Capacity: %d people\n", cars[i].capacity);
                printf("Price: %05.2f lakhs\n", cars[i].price_in_lakhs);
                printf("-------------------------\n");
        }
}
```

Here, we are creating an array of structure variables

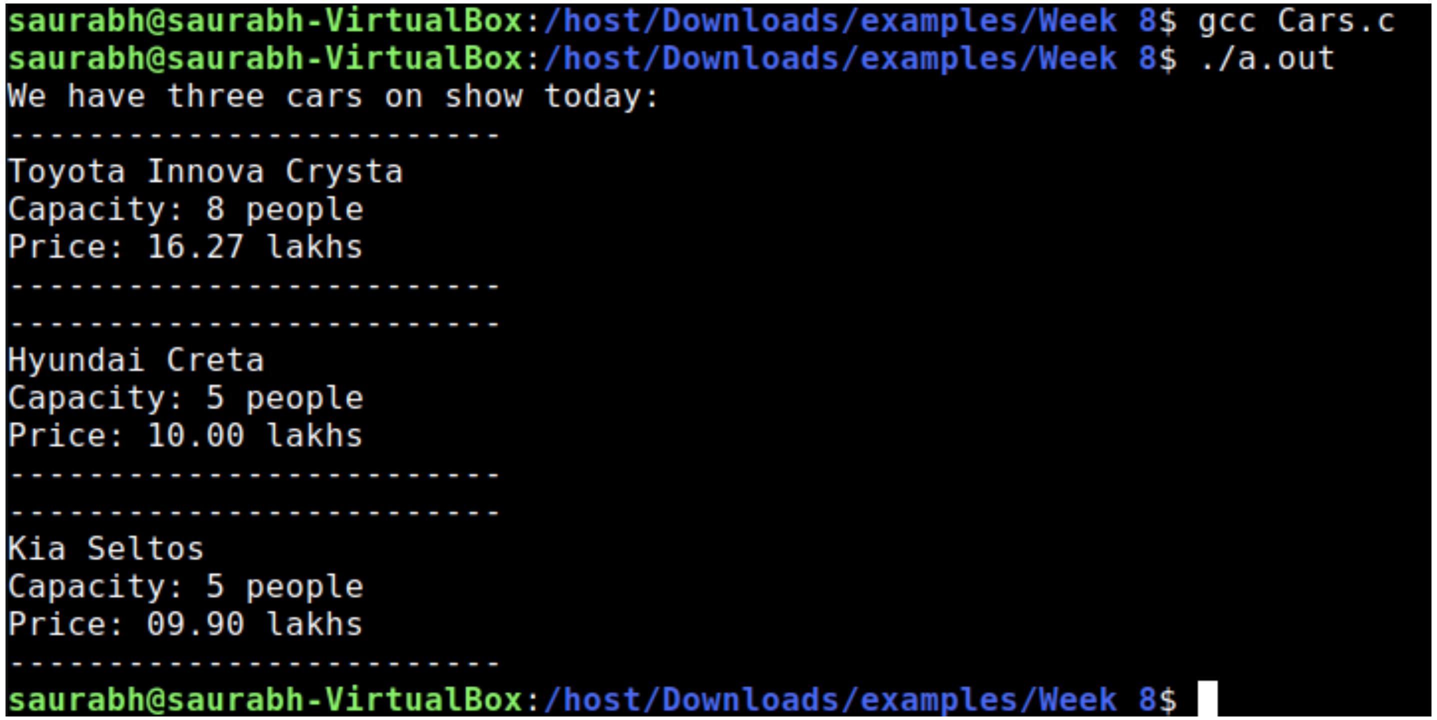
Example – Creating some simple instances



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

int main()
{
        struct PassengerVehicle cars[3];
        int i;

        strcpy(cars[0].name, "Toyota Innova Crysta");
        cars[0].capacity = 8;
        cars[0].price_in_lakhs = 16.27;

        strcpy(cars[1].name, "Hyundai Creta");
        cars[1].capacity = 5;
        cars[1].price_in_lakhs = 10;

        strcpy(cars[2].name, "Kia Seltos");
        cars[2].capacity = 5;
        cars[2].price_in_lakhs = 9.9;

        printf("We have three cars on show today:\n");

        for(i = 0; i < 3; i++)
        {
                printf("------------------------\n");
                printf("%s\n", cars[i].name);
                printf("Capacity: %d people\n", cars[i].capacity);
                printf("Price: %05.2f lakhs\n", cars[i].price_in_lakhs);
                printf("------------------------\n");
        }
}
```

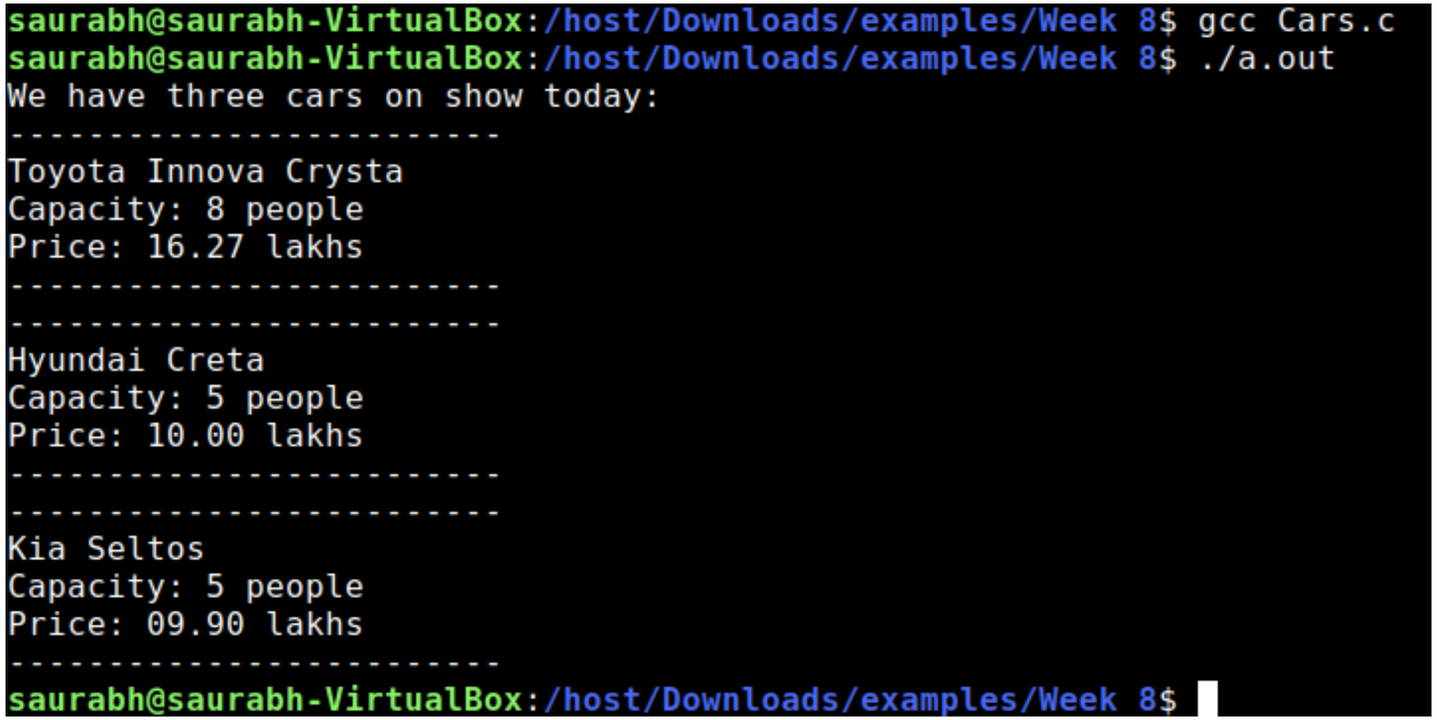
This is how we can access the individual member variables of the structure variables

Example – Creating some simple instances



```
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ gcc Cars.c
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ ./a.out
We have three cars on show today:
-------------------------
Toyota Innova Crysta
Capacity: 8 people
Price: 16.27 lakhs
-------------------------
-------------------------
Hyundai Creta
Capacity: 5 people
Price: 10.00 lakhs
-------------------------
-------------------------
Kia Seltos
Capacity: 5 people
Price: 09.90 lakhs
-------------------------
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ 
```

Example – Creating some simple instances



```
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ gcc Cars.c
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ ./a.out
We have three cars on show today:
-------------------------
Toyota Innova Crysta
Capacity: 8 people
Price: 16.27 lakhs
-------------------------
-------------------------
Hyundai Creta
Capacity: 5 people
Price: 10.00 lakhs
-------------------------
-------------------------
Kia Seltos
Capacity: 5 people
Price: 09.90 lakhs
-------------------------
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$
```

The value of member variables can be printed in the same fashion as any other variable of the same type

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword can create an *alias* for the declared structure

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword can create an *alias* for the declared structure

We usually use it to avoid writing the `struct` keyword, every time we create a new instance

- However, `typedef` is a general keyword, and can be used even without structures

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword can create an *alias* for the declared structure

We usually use it to avoid writing the `struct` keyword, every time we create a new instance

- However, `typedef` is a general keyword, and can be used even without structures

Any non-existing type name (it should not have been declared before) can be used with `typedef`

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword can create an *alias* for the declared structure

We usually use it to avoid writing the `struct` keyword, every time we create a new instance

- However, `typedef` is a general keyword, and can be used even without structures

Any non-existing type name (it should not have been declared before) can be used with `typedef`

- For example,
  ```
  typedef struct sample_structure new_type;
  ```

The `typedef` keyword

A keyword in C, which is often used when using structures, is the `typedef` keyword

The `typedef` keyword can create an *alias* for the declared structure

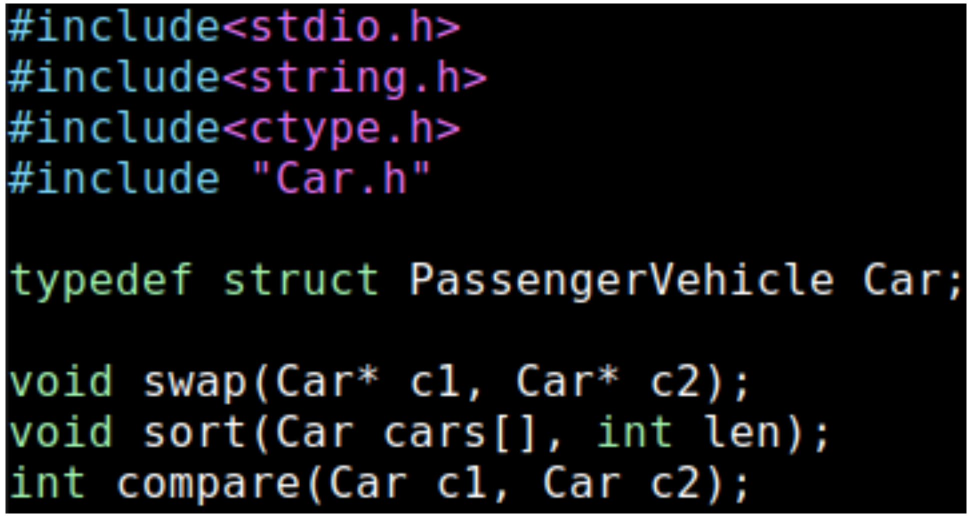
We usually use it to avoid writing the `struct` keyword, every time we create a new instance

- However, `typedef` is a general keyword, and can be used even without structures

Any non-existing type name (it should not have been declared before) can be used with `typedef`

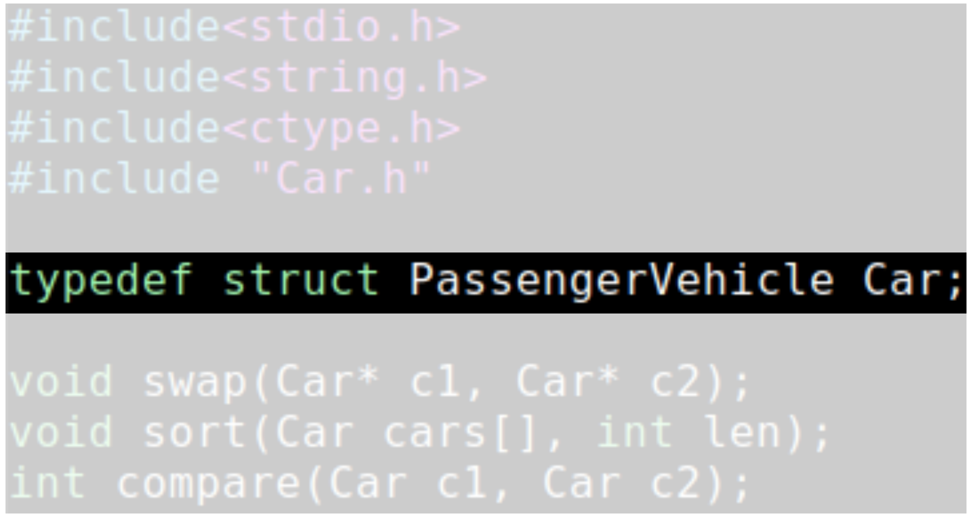
- For example,
  ```
  typedef struct sample_structure new_type;
  ```
- Then, we can simply use this short-cut to create instances of `sample_structure`
  ```
  new_type s3; // equivalent to writing struct sample_structure s3;
  ```

Example – Using `typedef` for short-cuts



```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include "Car.h"

typedef struct PassengerVehicle Car;

void swap(Car* c1, Car* c2);
void sort(Car cars[], int len);
int compare(Car c1, Car c2);
```

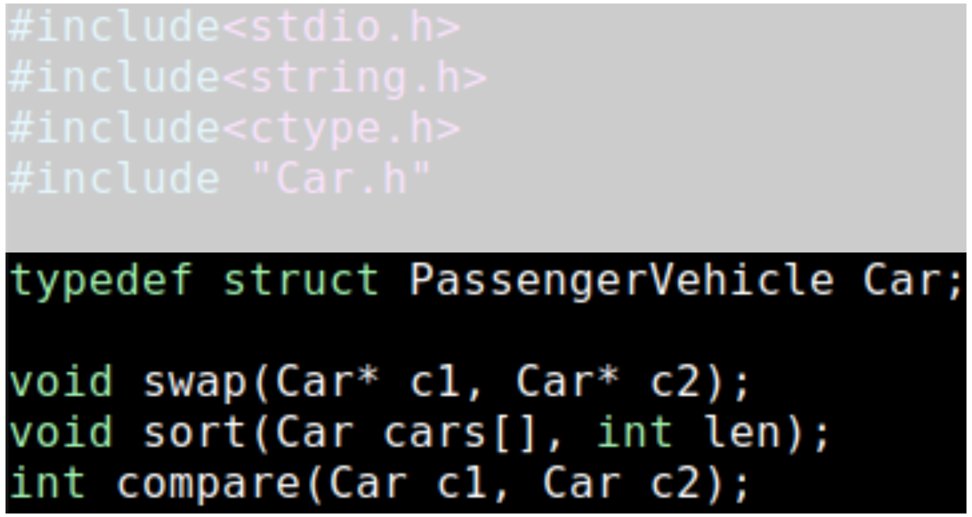
Example – Using `typedef` for short-cuts



```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include "Car.h"

typedef struct PassengerVehicle Car;

void swap(Car* c1, Car* c2);
void sort(Car cars[], int len);
int compare(Car c1, Car c2);
```

Here, we are creating an alias for the `PassengerVehicle` structure, called `Car`

Example – Using `typedef` for short-cuts



```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include "Car.h"

typedef struct PassengerVehicle Car;

void swap(Car* c1, Car* c2);
void sort(Car cars[], int len);
int compare(Car c1, Car c2);
```

Here, we are creating an alias for the `PassengerVehicle` structure, called `Car`

After that, we can use `Car` as a short-cut to writing: `struct PassengerVehicle`

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

- It can be done in exactly the same way as any other variable in C, for example:

```
struct sample_structure s2 = s1; // s1 is also of type sample_structure
```

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

- It can be done in exactly the same way as any other variable in C, for example:
  ```
  struct sample_structure s2 = s1; // s1 is also of type sample_structure
  ```

A pointer variable to a structure can be created in a similar fashion to any other type

- For instance:
  ```
  struct sample_structure* ptr = &s2;
  ```

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

- It can be done in exactly the same way as any other variable in C, for example:
  ```
  struct sample_structure s2 = s1; // s1 is also of type sample_structure
  ```

A pointer variable to a structure can be created in a similar fashion to any other type

- For instance:
  ```
  struct sample_structure* ptr = &s2;
  ```
- The pointer variable, points to the starting address of the structure variable…
- … and adding one, would take the pointer forward by the total size of the structure in bytes

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

- It can be done in exactly the same way as any other variable in C, for example:
  ```
  struct sample_structure s2 = s1; // s1 is also of type sample_structure
  ```

A pointer variable to a structure can be created in a similar fashion to any other type

- For instance:
  ```
  struct sample_structure* ptr = &s2;
  ```
- The pointer variable, points to the starting address of the structure variable…
- … and adding one, would take the pointer forward by the total size of the structure in bytes
- To access a particular member, the `*` and `.` operators can be used together, like
  ```
  (*ptr).i = 6;
  ```
- Note that the parentheses are necessary here since `.` has a higher precedence than `*`

Copying and Pointing to Structures

A structure variable can be copied into another structure variable of the *same type*

- It can be done in exactly the same way as any other variable in C, for example:
  `struct sample_structure s2 = s1; // s1 is also of type sample_structure`

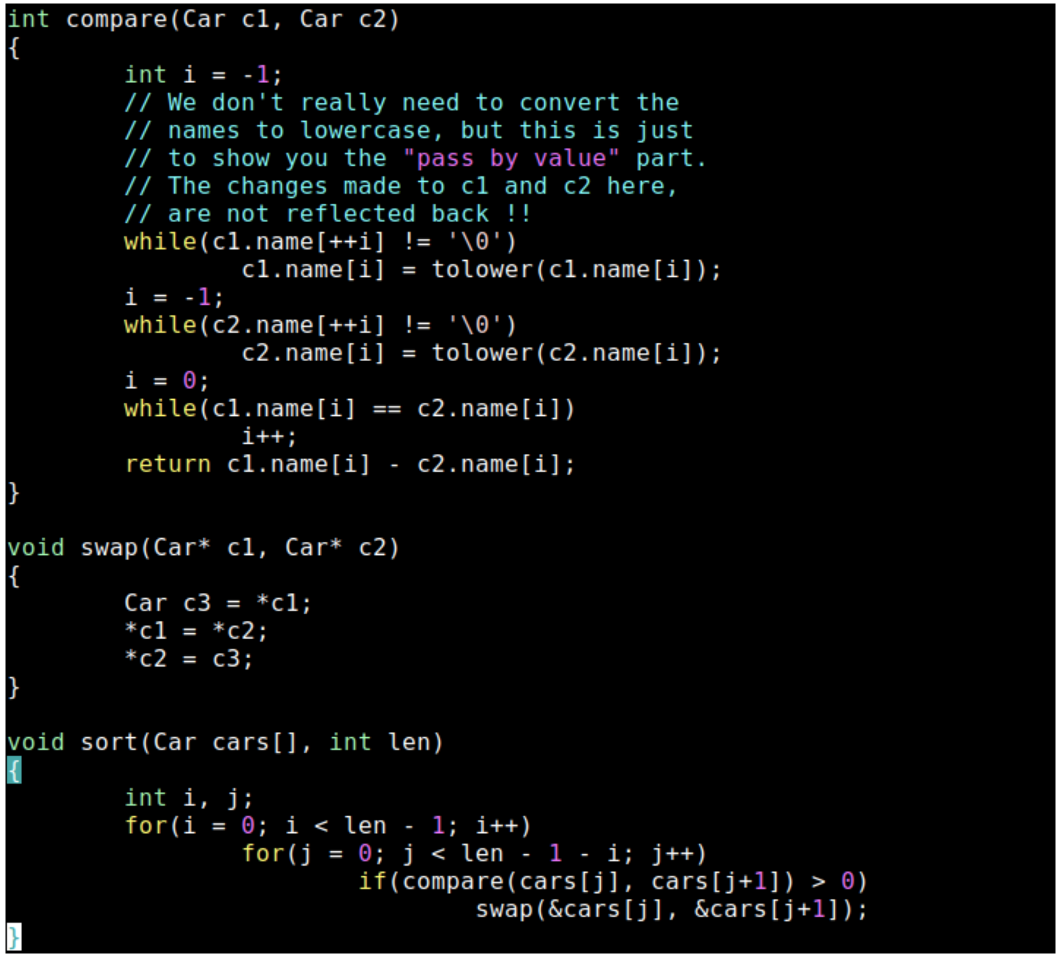
A pointer variable to a structure can be created in a similar fashion to any other type

- For instance:
  `struct sample_structure* ptr = &s2;`
- The pointer variable, points to the starting address of the structure variable…
- … and adding one, would take the pointer forward by the total size of the structure in bytes
- To access a particular member, the `*` and `.` operators can be used together, like
  `(*ptr).i = 6;`
- Note that the parentheses are necessary here since `.` has a higher precedence than `*`

The rules related to passing and returning arguments with function calls, also apply to structures

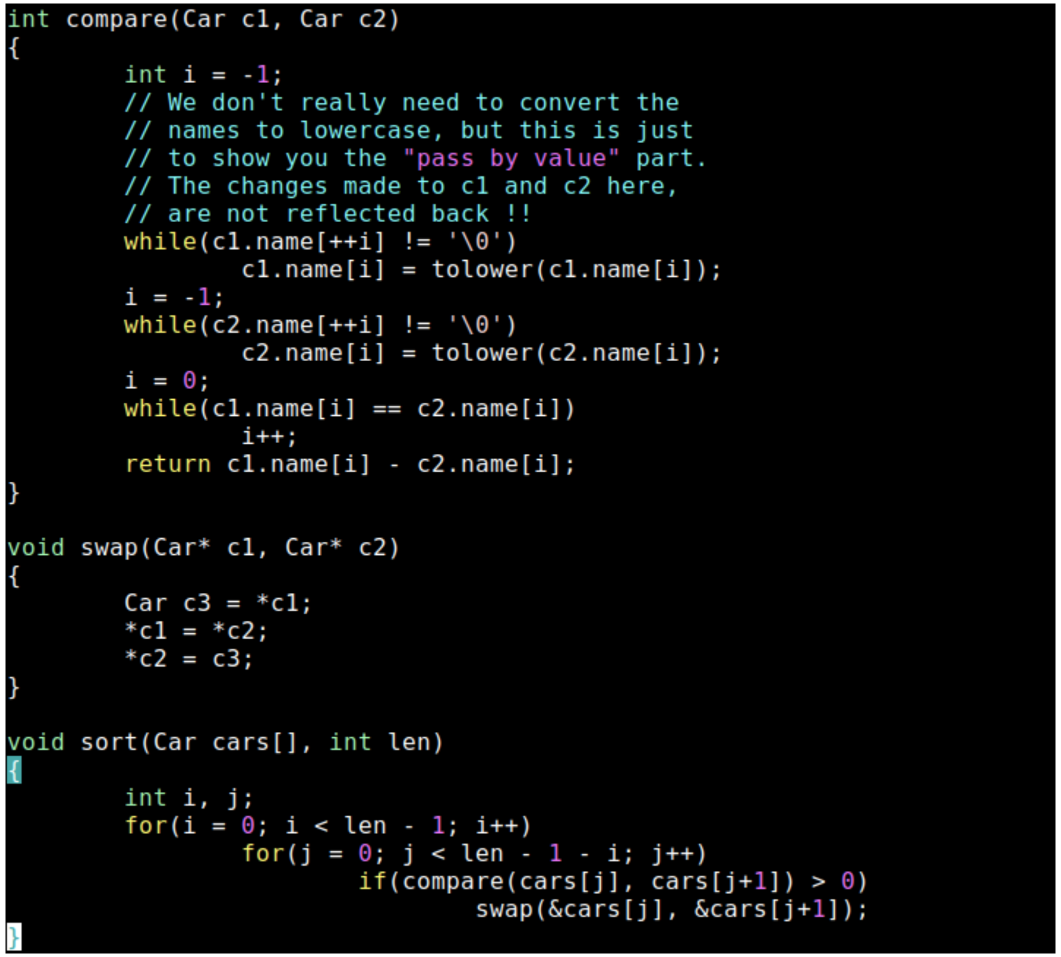
- So, structure variables can be passed by value as well as reference, in a similar fashion

Example – Sorting Structure Variables



```
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

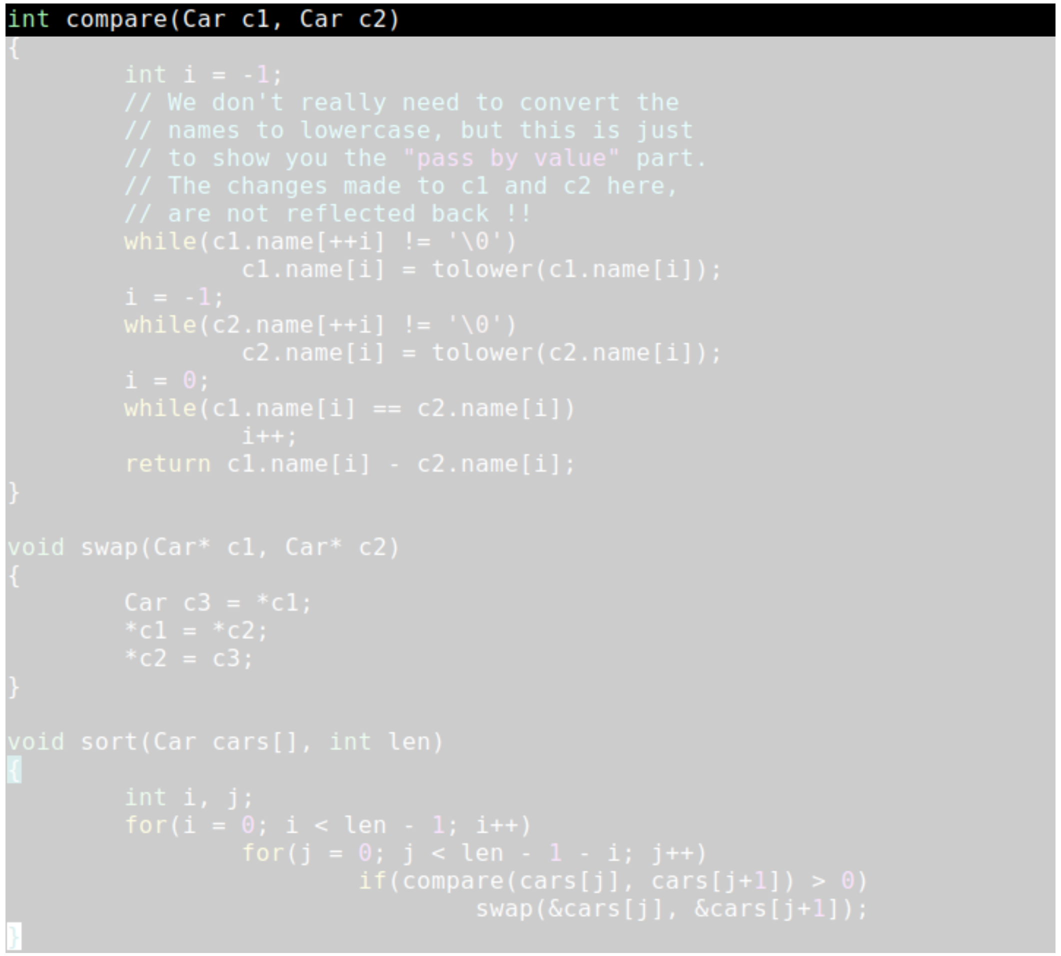
Example – Sorting Structure Variables



```
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

These are some examples of passing structure variables to functions

Example – Sorting Structure Variables

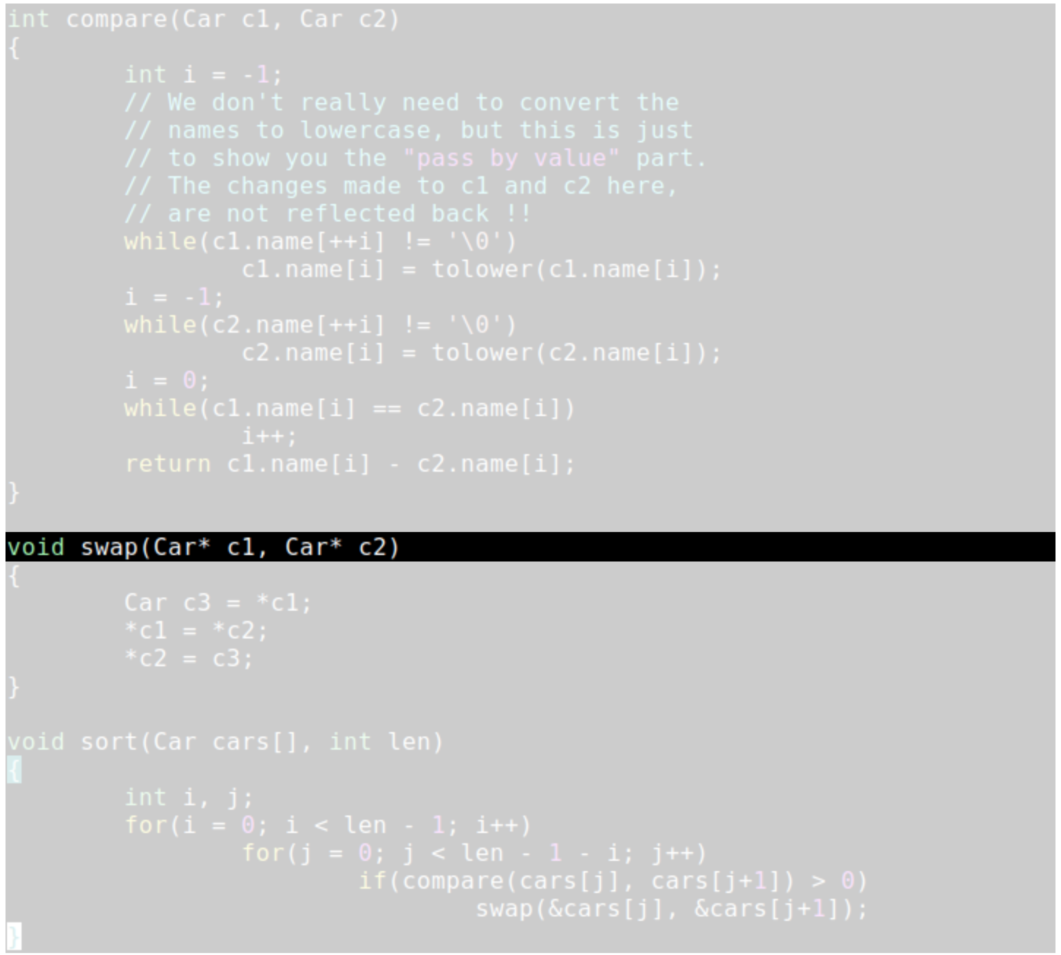


```c
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

These are some examples of passing structure variables to functions

This is an example of Passing structures by value…

Example – Sorting Structure Variables

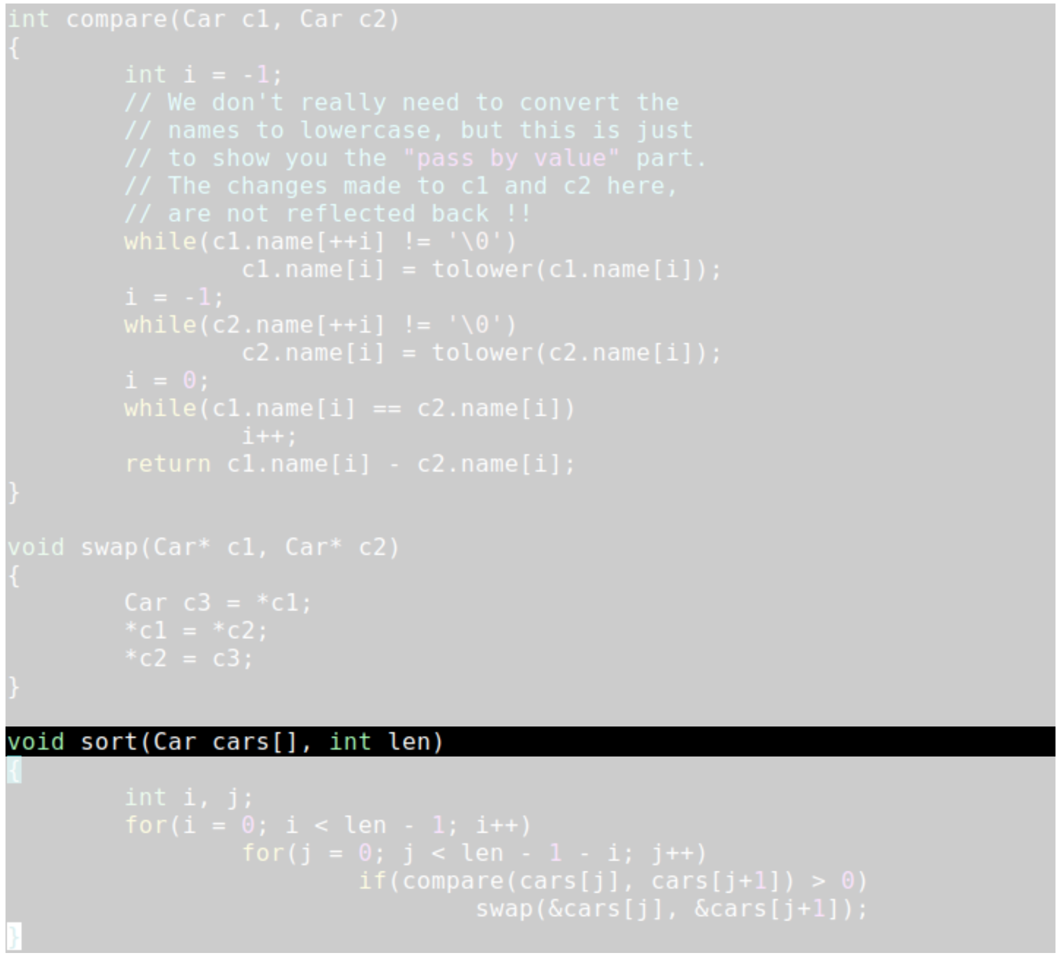


```
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

These are some examples of passing structure variables to functions

… and this is an example of passing structures by reference

Example – Sorting Structure Variables

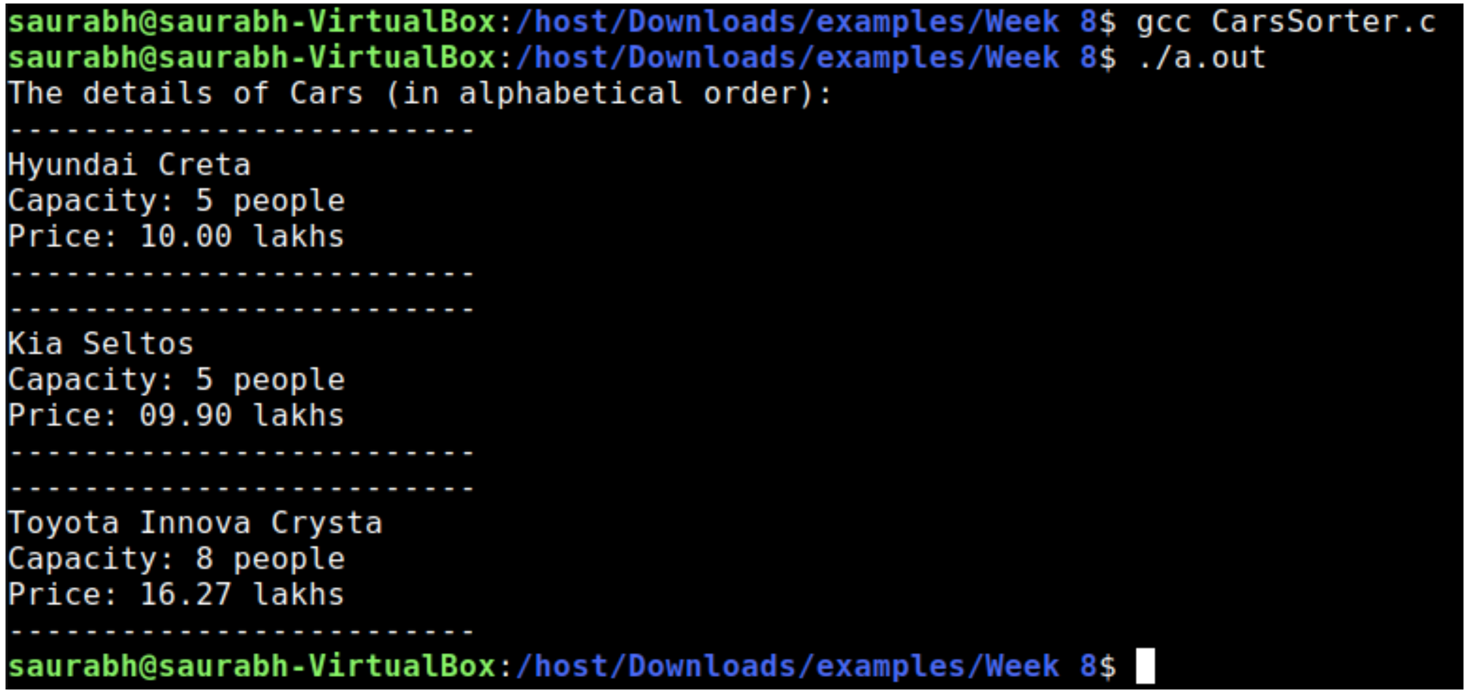


```
int compare(Car c1, Car c2)
{
        int i = -1;
        // We don't really need to convert the
        // names to lowercase, but this is just
        // to show you the "pass by value" part.
        // The changes made to c1 and c2 here,
        // are not reflected back !!
        while(c1.name[++i] != '\0')
                c1.name[i] = tolower(c1.name[i]);
        i = -1;
        while(c2.name[++i] != '\0')
                c2.name[i] = tolower(c2.name[i]);
        i = 0;
        while(c1.name[i] == c2.name[i])
                i++;
        return c1.name[i] - c2.name[i];
}

void swap(Car* c1, Car* c2)
{
        Car c3 = *c1;
        *c1 = *c2;
        *c2 = c3;
}

void sort(Car cars[], int len)
{
        int i, j;
        for(i = 0; i < len - 1; i++)
                for(j = 0; j < len - 1 - i; j++)
                        if(compare(cars[j], cars[j+1]) > 0)
                                swap(&cars[j], &cars[j+1]);
}
```

These are some examples of passing structure variables to functions

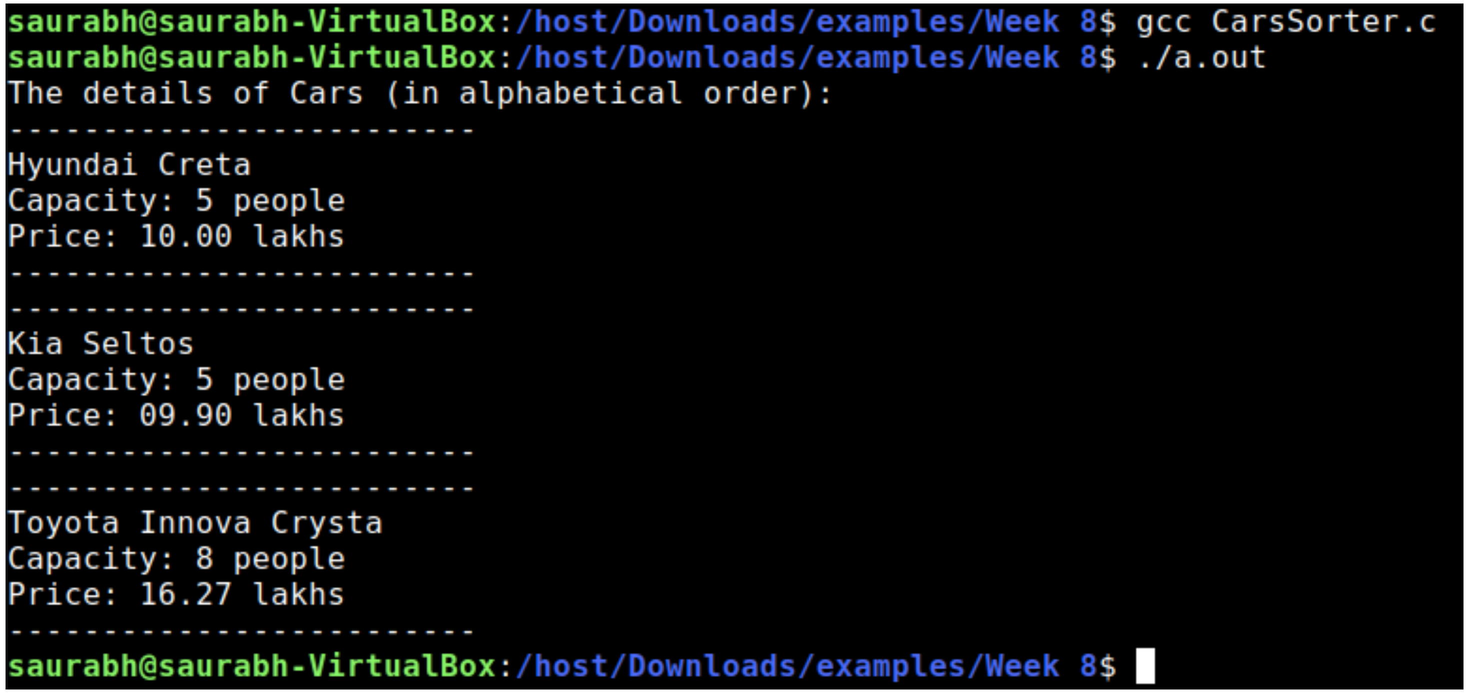
This is an example of passing arrays of structures to functions, which, as usual, are passed by reference

Example – Sorting Structure Variables



```
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ gcc CarsSorter.c
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ ./a.out
The details of Cars (in alphabetical order):
-------------------------
Hyundai Creta
Capacity: 5 people
Price: 10.00 lakhs
-------------------------
-------------------------
Kia Seltos
Capacity: 5 people
Price: 09.90 lakhs
-------------------------
-------------------------
Toyota Innova Crysta
Capacity: 8 people
Price: 16.27 lakhs
-------------------------
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$
```

Example – Sorting Structure Variables



```
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ gcc CarsSorter.c
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ ./a.out
The details of Cars (in alphabetical order):
-------------------------
Hyundai Creta
Capacity: 5 people
Price: 10.00 lakhs
-------------------------
-------------------------
Kia Seltos
Capacity: 5 people
Price: 09.90 lakhs
-------------------------
-------------------------
Toyota Innova Crysta
Capacity: 8 people
Price: 16.27 lakhs
-------------------------
saurabh@saurabh-VirtualBox:/host/Downloads/examples/Week 8$ 
```

You can see the affects of sorting in the `main()` function

Homework !!

We have created a header file in the Vehicle example

- Read more about header files in C, and when is it a good idea to use them
- You may start from here: https://www.geeksforgeeks.org/header-files-in-c-cpp-and-its-uses/

Also figure out why `Car.h` was put inside quotes, and not `<>`

Additional Reading

Though the amount of space taken by a structure should be the sum of spaces of all its variables…

- … there is a chance that it takes a bit more space in the memory than that

You can use the sizeof() operator to figure it out

The additional space is used as a *padding* for proper alignment of data in the memory

These two links may be worth checking out, if you wish to know more:
https://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/
https://stackoverflow.com/questions/11906486/size-of-struct-containing-double-field