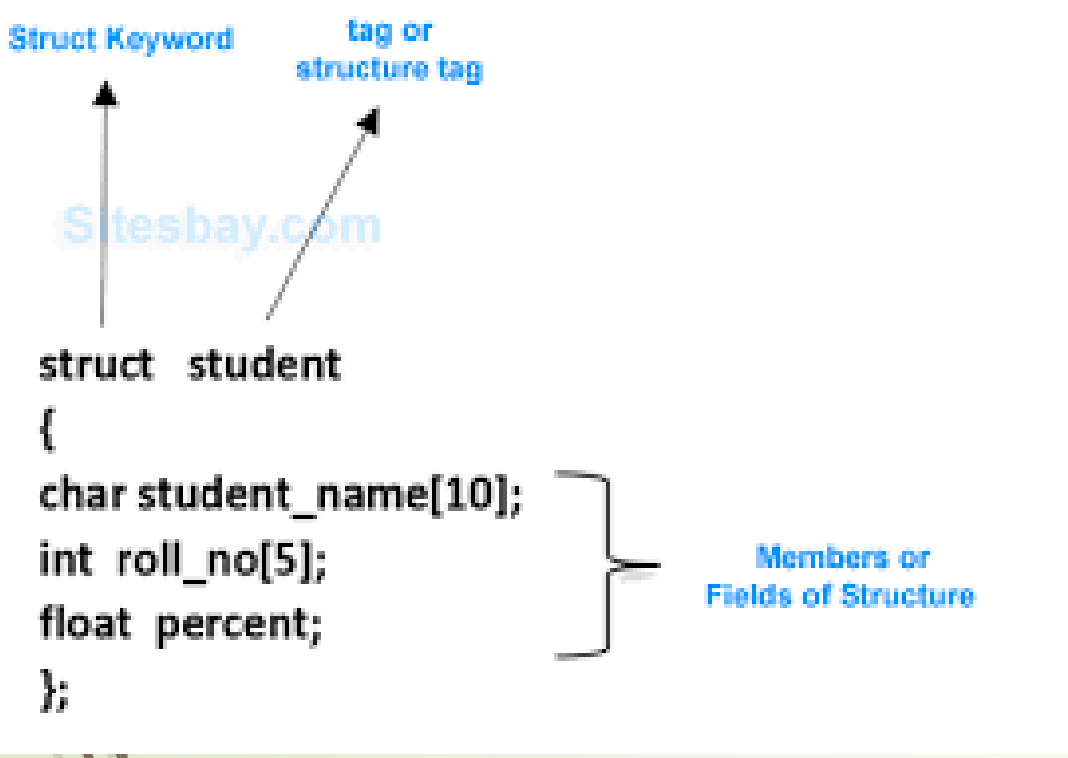
Why we use Structure

- Structures allow to store different data types information inside it at contiguous memory location. It is a collection of different data types which are at contiguous memory locations.
- If we want to represent too many information(too many variables) of same data type as an single entity then we go **for Array** but if we want to represent too many information(too many variables) **of different data types** as an single entity then we go for Structure.

Example

- Suppose you are having record of 100 students. Each student is having name(which is of char[] type), id(int type) and marks(float type). All these three properties is uniquely owned by each student. Each student will carry this three different data type information.
- Now, suppose you have to pass complete information of all students to one function for some reason. At that point rather than passing each and every information(name, id, marks, etc...) one by one for every student will not be a good way of programming. In such case we will prefer approach in which can pass complete student information as an single entity of one by one student.
- So, where ever we need to refer different data types information as an single entity then we go for structure and above is the real time example where we would like to go for structure.

struct: A collection of a fixed number of components in which the components are accessed by name. The components may be of different types. The components of a struct are called the members of the struct.



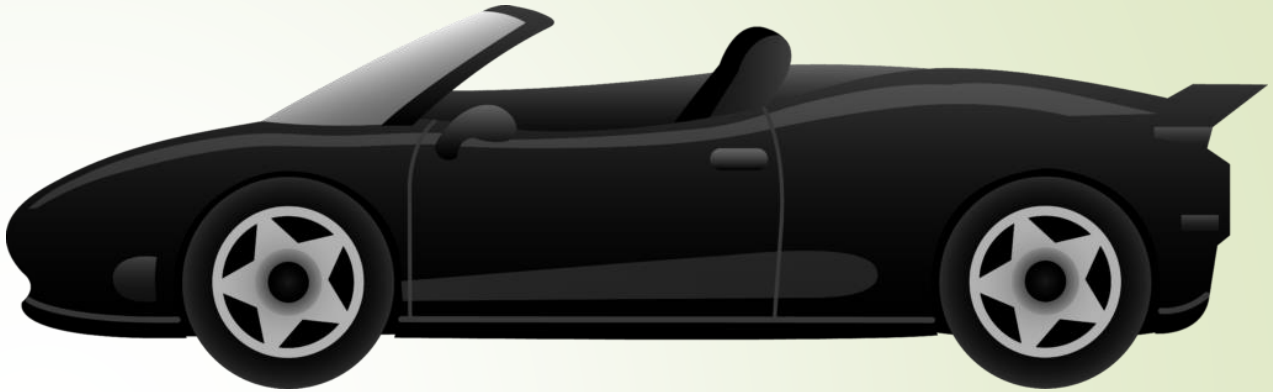
As you can see in the syntax above, we start with the struct keyword, then your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

Some advantages of structural programming includes; - It is friendly and easy to understand, it is similar to English vocabulary of words and symbols, it is easy to learn and maintain, it is problem oriented rather than machine oriented, it is independent of machine on which it is used and last.

Problem statement

I have a Garage and I want to store all of the information/record about cars which are in my garage.

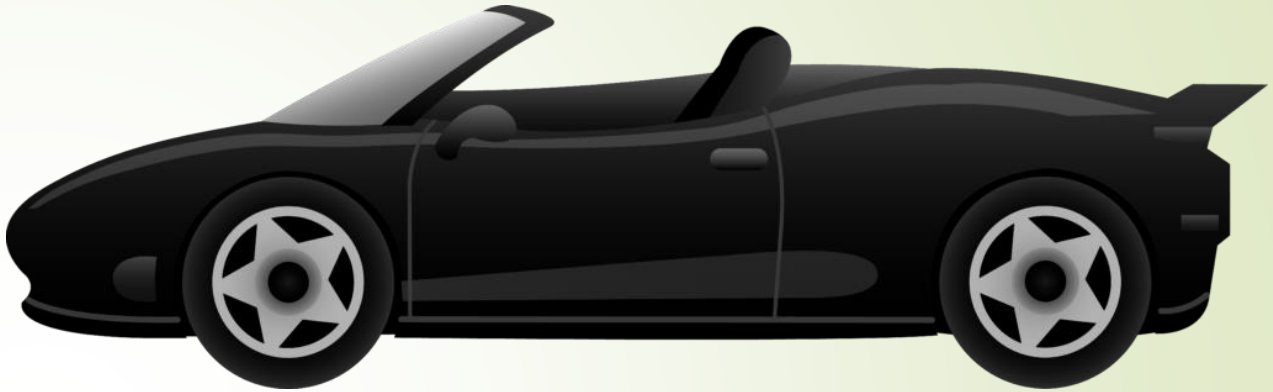
Car 1 specification



Engine	DDIS 190 Engine
Fuel Type	Petrol
Tank capacity	37
Seating Capacity	5
Milage	40.74 km/h

We want to store all of this information, so you can create separate variables.

Car 2 specification



Engine	Kappa Dual
Fuel Type	Diesel
Tank capacity	37
Seating Capacity	5
Milage	55.74 km/h

Car 2 specification

- What if we define our own type which can accommodate all records.
- **Structure is only the solution**
- ***Structure is user defined data type used to group elements of different type into a single type.***

Struct

- Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C++ that allows to combine data items of different kinds.
- Structures are used to represent a record.

Structure in Global Scope

- The code we have seen has the structure in global scope that means every function can access it.

Structure in Local Scope

- It means that only one function can access it data members in which it is defined.

methods to declare variables of structure

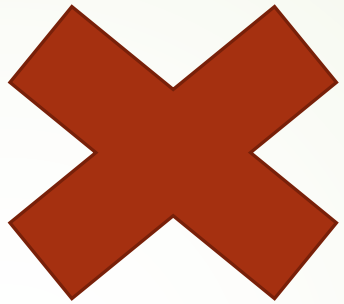
```
struct cars
{
        char *engine;
        char * fuelType;
        int tnakCap;
        int SC;
        float milage;
};

Int main()
{
         cars car1,car2;
}
```

Initialize data members

```
Struct cars
{
        int SC=5;
        float milage=3.2;
};

Int main()
{
         cars car1,car2;
}
```



Initialize data members

```
Struct cars
{
        int SC;
        float milage;
};

Int main()
{
         cars car1={5,37.8};
         cars car2={2,44.8};


}
```

Another way

You can also declare struct variables when you define the struct. For example, consider the following statements:

```
struct studentType
{
        string firstName;
        string lastName;
        char courseGrade;
        int testScore;
        int programmingScore;
        double GPA;

} tempStudent;
```

Access of data members

You can also declare struct variables when you define the struct. For example, consider the following statements:

```
struct Student{
        string firstName;
        string lastName;
        char courseGrade;
        int testScore;
        int programmingScore;
        double GPA;

} newStudent;
```

newStudent.GPA = 0.0;

Similarly, the statements:

newStudent.firstName = "John";

newStudent.lastName = "Brown";

Assignment

We can assign the value of one struct variable to another struct variable of the same type by using an assignment statement.

```
struct Student{
        string firstName;
        string lastName;
        char courseGrade;
        int testScore;
        int programmingScore;
        double GPA;

} newStudent;

Student student;
student = newStudent;
```

Comparison

To compare struct variables, you compare them member-wise. For example, suppose that newStudent and student are declared as shown earlier.

```
if (student.firstName == newStudent.firstName &&
student.lastName == newStudent.lastName)
```

```
if (student == newStudent) //illegal
.
.
.
```

struct Variables and Functions

- A struct variable can be passed as a parameter either by value or by reference, and
- A function can return a value of type struct.

```
void readIn(Student& student)
{
        int score;
        cin >> student.firstName >> student.lastName;
        cin >> student.testScore >> student.programmingScore;
        cin >> student.GPA;
        score = (student.testScore +student.programmingScore) / 2;
        if (score >= 90)
        student.courseGrade = 'A';
        else if (score >= 70)
        student.courseGrade = 'B';
        Else:
        student.courseGrade = 'F';
}
readIn(newStudent);
```

Question

Write a program to help a local restaurant automate its breakfast billing system. The program should do the following:

a. Show the customer the different breakfast items offered by the restaurant.

b. Allow the customer to select more than one item from the menu.

c. Calculate and print the bill.

Assume that the restaurant offers the following breakfast items (the price of each item is shown to the right of the item):

Plain Egg	$15.00
Omelet Egg	$12.00
Paratha	$12.00
French Toast	$1.99
Fruit Basket	$2.49
Coffee	$5.0
Tea	$0.75

Question cont

Use an array, **menuList,** of the struct **menuItemType**. Your program must contain at least the following functions:

- Function **getData**: This function loads the data into the array menuList**.**
- Function **showMenu:** This function shows the different items offered by the restaurant and tells the user how to select the items.
- Function **printCheck**: This function calculates and prints the check. (Note that the billing amount should include a 5% tax.)

A sample output is:

Welcome to Johnny's Restaurant

Bacon and Egg	$2.45
Muffin	$0.99
Coffee	$0.50
Tax	$0.20
Amount Due	$4.14

Format your output with two decimal places. The name of each item in the output must be left justified. You may assume that the user selects only one item of a particular type.

Structural programming vs OOP

Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process.	Object Oriented Programming is designed which focuses on **data**.
Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**.	Object Oriented Programming supports **inheritance, encapsulation, abstraction**, **polymorphism**, etc.
In Structured Programming, Programs are divided into small self contained **functions**.	In Object Oriented Programming, Programs are divided into small entities called **objects**.
Structured Programming is **less** secure as there is no way of **data hiding**.	Object Oriented Programming is more secure as having data hiding feature.
Structured Programming can solve **moderately** complex programs.	Object Oriented Programming can solve any **complex** programs.
Structured Programming provides **less reusability**, more function dependency.	Object Oriented Programming provides more reusability, less function **dependency**.

Principles of Object-Oriented Systems

Major Elements − By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are −

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction

Abstraction means to focus on the essential features of an element or object in OOP Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information. The main purpose of abstraction is hiding the unnecessary details from the users.

We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Encapsulation is a process of wrapping the data and the code, that operate on the data into a single entity.
The user need not worry about internal details and complexities of the system.

variable can be accessed and manipulated only by using the methods get() and set() that are present within the class. Therefore we can say that, the variable a and the methods set() as well as get() have bound together that is encapsulation.

Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem.