Structures

Structures

- Structures provide a way to unify several variables of different types into a single, new variable type which can be assigned its own type name.

- We use structures (`structs`) to group together elements of a variety of data types that have a logical connection.

- Think of a structure like a “super-variable”.

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

- Once we have defined a structure, which we typically do in separate .h files or atop our programs outside of any functions, we have effectively created a new type.

- That means we can create variables of that type using the familiar syntax.

- We can also access the various **fields** (also known as **members**) of the structure using the dot operator (`.`)

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
mycar.plate = “CS50”;
mycar.odometer = 50505;
```

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
mycar.plate = “CS50”;
mycar.odometer = 50505;
```

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
mycar.plate = “CS50”;
mycar.odometer = 50505;
```

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
mycar.plate = “CS50”;
mycar.odometer = 50505;
```

Structures

- Structures, like variables of all other data types, do not need to be created on the stack. We can dynamically allocate structures at run time if our program requires it.

- In order to access the fields of our structures in that situation, we first need to dereference the pointer to the structure, and then we can access its fields.

Structures

```c
// variable declaration
struct car *mycar = malloc(sizeof(struct car));
```

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
(*mycar).year = 2011;
(*mycar).plate = “CS50”;
(*mycar).odometer = 50505;
```

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
(*mycar).year = 2011;
(*mycar).plate = “CS50”;
(*mycar).odometer = 50505;
```

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
(*mycar).year = 2011;
(*mycar).plate = “CS50”;
(*mycar).odometer = 50505;
```

Structures

- This is a little annoying. And so as you might expect, there's a shorter way!

- The arrow operator (`->`) makes this process easier. It's an operator that does two things back-to-back:
  - First, it **dereferences** the pointer on the left side of the operator.
  - Second, it **accesses** the field on the right side of the operator.

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
(*mycar).year = 2011;
(*mycar).plate = “CS50”;
(*mycar).odometer = 50505;
```

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
mycar->year = 2011;
mycar->plate = “CS50”;
mycar->odometer = 50505;
```