Hello C...TYI!

Domhnall O'Hanlon

March 21, 2017

Why Structures?

We've seen lots of examples so far of where we would want to store multiple pieces of information. Up until now we've always used an array, however the main drawback of using an array is that it can only store one **type** of information, like an array of ints or an array of chars. The main advantage of using a struct is that it allows you to use different types of data.

Convention

The convention in C is to create your structs in a seperate header `.h` file, but for the time being we'll create them in our main.c files. In this lesson we'll create a basic car structure, make a few different cars and then modify them using a few functions we'll create ourselves.

Creating a struct

In something as complex as car, there are lots of different properties that we might want to assign. We can use strings to represent things like the make, model, features etc. numbers to represent properties such as horse power or top speed and arrays to capture the variety of engine sizes available, or perhaps types of fuel. The properties of a struct are contained between curly braces, and just like when you initialise any other data type, you must end your declaration with a semicolon.

```c
struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}; /* don't forget the semicolon */
```

Using a Struct

Now that we've created a prototype for all cars in general, we can now go ahead and make specific cars by using our car struct.

```
    /* Add a new car by using our struct */
    struct car bv = {"Bugatti", "Veyron", 1000, 407};
```

The Dot Operator

The dot operator, `.`, is used to assess specific members (elements) of a struct. In the example of our car structure, it has properties, or elements corresponding to the make, model, horse power and, top speed. Each of these can be accessed using the syntax `name.element` like in the example below:

```
    printf("Make: \t %s\n", bv.make);
    printf("Model: \t %s\n", bv.model);
    printf("Horse Power: %d\n", bv.bhp);
    printf("Top Speed \t %d \n", bv.topSpeed);
```

Typedef

```
/* TYPEDEF: we don't have to be limited to just ints, chars, arrays etc - now that we can cr
can create, or define, our own types using typedef */

#include <stdio.h>
#include <stdlib.h>

typedef struct car{
    char* make;
    char* model;
    int bhp, topSpeed;
}car; /* don't forget the semicolon */

/*create a function which prints the top speed of any car */
void getTopSpeed(struct car anyCar){
    printf("The top speed of the %s %s is: %d km\\hr. \n",
        anyCar.make, anyCar.model, anyCar.topSpeed);
}

int main(int argc, char* argv[])
{
    /* Creating cars without having to use struct all the time */
    car bv = {"Bugatti", "Veyron", 1000, 407};
    car vg = {"Volkswagen", "Golf", 227,170};

    /* Use the function to get the top speed of the cars */
    getTopSpeed(bv);
    getTopSpeed(vg);

    return 0;
}
```

Example Code

A quick example of sending data to functions and making it globally accessible throughout your application.

Here is the complete code for this section:

```
/*
Now that we're using pointers the variables will update correctly.
 */

#include <stdio.h>
#include <stdlib.h>

typedef struct person {
    char* name;
    int bankBalance;
} person;

void developApp();

int main(int argc, char* argv[])
{
    /* create a few people */
    person mz = {"Mark", 500};
    person es = {"Eduardo", 10000};

    developApp(&mz, 1000);

    /*print out mz.bankBalance here too */
    printf("In main() the balance is: %d ", mz.bankBalance);
    return 0;
}
```

```c
void developApp(person *somebody, int sales)
{
    printf("%s just launched a new app\n", somebody->name);

    somebody->bankBalance += sales;

    printf("%s's new net worth is: %d \n", somebody->name, somebody->bankBalance);
```

Summary