Tutorial 10 - 14.01.2020

Group 06 - Moritz Makowski

Repetition - Structs and Pointers

Today's Agenda

- Repetition: Structs
    - Why use structs?
    - Defining/Initiliazing a struct
    - Using/Manipulating a struct
- Repetition: Pointers
    - Variables (values) vs. pointers
    - Referencing and dereferencing
    - Memory allocation inside functions
    - Pointers and structs
- **Exercise 10.1: Practice Project - Car Dealership**

Using Functions?

Problem: One long block of code which is hard to understand/debug

Idea: Separating logical sections of functionality into functions

Benefits:

- Enclosed functionality $\rightarrow$ Easier to write/understand/debug
- Reusing functionality $\rightarrow$ No copy-pasting code
- Descriptive function names $\rightarrow$ Readable/self-documenting code

Using Structs

Problem: A lot of similar variable names in the same scope which can easily be grouped together

Scenario: When implementing real-life-processes you often have objects with different attributes. Often you have multiple instances of one type of object (also called class) which all have the same set of attributes.

Example: You want to write a simulation which contains a bunch of geomentric 2D objects, e.g. a few `circles`, `rectangles` and `triangles`. Each object has dimensions and coordinates.

Basic approach (without using structs)

1) Initializing:

```
float circle_1_radius = 0.5;
float circle_1_x = 2.5;
float circle_1_y = 1.2;

float circle_2_radius = 0.5;
float circle_2_x = 2.5;
float circle_2_y = 1.2;

float circle_3_radius = 0.5;
float circle_3_x = 2.5;
float circle_3_y = 1.2;
```

2) A list of circles:

```
float circles[3][3] = {
  {circle_1_radius, circle_1_x, circle_1_y},
  {circle_2_radius, circle_2_x, circle_2_y},
  {circle_3_radius, circle_3_x, circle_3_y}
};
```

3) Iterating over all circles:

```
for (int i=0; i<3; i++) {
  // You can access the i-th circle (=row) with circles[i]
  // and all its attributes with circles[i][0]/...[1]/...[2]

  printf("Circle:\n");
  printf("radius = %d\n", circles[i][0]);
  printf("x = %d\n", circles[i][1]);
  printf("y = %d\n", circles[i][2]);
};
```

Advanced approach (now using structs)

1) Initializing:

```
struct Circle {
  float radius, x, y;
}

struct Circle circle_1 = {0.5, 2.5, 1.2};
struct Circle circle_2 = {0.5, 2.5, 1.2};
struct Circle circle_3 = {0.5, 2.5, 1.2};
```

2) A list of circles:

```
struct Circle circles[3] = {circle_1, circle_2, circle_3};
```

3) Iterating over all circles:

```
for (int i=0; i<3; i++) {
  // You can access the i-th circle-"element" with circles[i]

  printf("Circle:\n");
  printf("radius = %d\n", circles[i].radius);
  printf("x = %d\n", circles[i].x);
  printf("y = %d\n", circles[i].y);
}
```

Big disadvantages when not using structs:

- No single circle-"element", just a bunch of unrelated variables. Only you know what belongs together.
- Way more effort to pass "elements" to functions
- Less readable code because real-world hierachies (objects, attributes of objects) are only related by descriptive variable naming (which is not required...)

Defining a Struct

```c
struct Circle {
  float radius;
  float x;
  float y;
  char name[32];
};
```

… or …

```c
struct Circle {
  float radius, x, y;  // Possible because all are floats
  char name[32];
};
```

Initializing a Struct

```
struct Circle circle_1;
circle_1.radius = 5.0;
circle_1.x = 3.2;
circle_1.y = 1.0;
circle_1.name = "Point A71";
```

... or ...

```
struct Circle circle_1 = {
    .radius = 5.0,
    .x = 3.2,
    .y = 1.0,
    .name = "Point A71"
};
```

... or ...

```
struct Circle circle_1 = {5.0, 3.2, 1.0, "Point A71"};
```

Accessing a Struct

```
printf("circle_1: \"%s\"\n", circle_1.name);
printf("radius = %d\n", circle_1_.radius);
printf("x = %d\n", circle_1_.x);
printf("y = %d\n", circle_1_.y);
```

Manipulating a Struct

```
circle_1.x = circle_1.x + 0.1;
```

… or …

```
circle_1.x += 0.1;
```

*Some comments, **outlook**:*

*This way of programming which is beginning with using structs is called **object oriented programming**, a technique where you separate state-related memory (variables) as well as constants and also functions into different classes of objects. So most functions and variables belong to a class or an object.*

Don't worry! This is not part of Engineering Informatics 1!

*Classes are blueprints for objects. The concept of a rectangle is called a **class** but there is not single real-world object associated with it. All objects that are constructed according to a specific blueprint (class) are called **instances of that class**.*

So in our example `Circle` *is a class and* `circle_1`, `circle_2` *and* `circle_3` *are instances of that class with the same set of attributes but different values.*

If you're confused now, thats ok, you will learn more about this in your 4th semester in Engineering Informatics 2.

Pointers

As you recall everything stored in memory has an address - defining its location in memory. When we **define a variable** in our code, this variable automatically **gets assigned a memory slot**.

```
int a = 14;  // An integer with the value 14 is now stored
             // at a specific location in memory
```

You can see and work with this adress.

```
int *pointer_to_a = &a;  // This variable stores that address
printf("pointer_to_a = %p", pointer_to_a);
```

A pointer is a data type which is associated with one of the primitive data types (integers, floating point numbers).

An integer stores a whole number. An integer-pointer stores the adress to a memory-slot storing an integer.

Therefore for each primitive data type `int`, `long`, `float`, `double`, etc. and also `struct ...` there also is a pointer-data-type.

You intialize a pointer by adding a `*` in front of the pointer-name.

You can get the address of a variable of a primitive data type with `&`.

```
int *pointer_to_a = &a;
```

When you want to access/manipulate the variable a pointer points to you also use `*`.

```
*(pointer_to_a) = *(pointer_to_a) + 3;
```

… or …

```
*(pointer_to_a) += 3;
```

Getting the address (`&`) = **"Referencing"**

Getting the value (`*`) = **"Dereferencing"**

When a pointer should "not point anywhere", its best to assign it the value `0`, which is not a valid memory adress. This is also called a **"NULL-Pointer"**.

Memory Allocation Inside Functions

What is wrong with the following code?

```
int *pointer_to_square(int number) {
  int square = number * number;
  return &square;
}
```

The scope of the variable square is only local (inside the function `pointer_to_square`).

Once the execution of that function is over, **all local variables get "destroyed"** (= "memory being freed"). So the **pointer** which is returned **points to memory that has been freed** already once the pointer can be used from the outside.

The best strategy to solve this issue is to dynamically allocate the memory:

```c
int *pointer_to_square(int number) {
  int *square = malloc(sizeof(int));
  *square = number * number;
  return square;
}
```

Remember to `free` the memory afterwards!

Pointers and Structs

Example:

```
struct Circle {
  float radius, x, y;
}
```

Scenario: There is a struct but you can only access a pointer to this struct.

```
struct Circle circle_1 = {...}
struct Circle *pointer_to_circle_1 = &circle_1;
```

1) Dereferencing and Manipulation

```
(*pointer_to_circle_1).x += 0.1;
```

2) Using the `->` notation

```
pointer_to_circle_1->x += 0.1;
```

Exercise 10.1: Car Dealership

Given two structs:

```
struct Car {
  char brand[20];
  char model[20];

  // The price at which a dealers
  // sells this car to a customer
  int price;
};

struct Dealer {
  int account_balance;
  struct Car *cars[20];
};
```

Task: Implement the Following Functions:

1) Dealer Initialization/Manipulation

2) Dealer Analysis

3) Customer Initialization

4) Customer-Dealer Analysis

5) Printing Functions

6) Memory Deallocation Functions

1) Dealer Initialization/Manipulation

`init_dealer` initializes a car dealer (malloc/calloc) with an empty list `cars` (only NULL pointers) and the given `account_balance`.

```
struct Dealer *init_dealer(int account_balance);
```

`add_car` initializes an instance of `struct Car` (malloc/calloc) and adds the pointer to that instance to the dealer's list `cars`. The dealers `account_balance` is ignored.

```
void add_car(struct Dealer *dealer,
  char brand[32], char model[32],
  int price);
```

`remove_car` removes a given car instance from a dealers inventory. The methods frees up the cars memory and shifts all cars inside the dealers car list so that there is no null-pointer inbetween two cars inside the array.

```
void remove_car(struct Dealer *dealer, struct Car *car);
```

2) Dealer Analysis

`car_count` returns the number of cars a dealer owns.

```
int car_count(struct Dealer *dealer);
```

`total_dealer_value` returns a dealers account_balance if he were to sell all of his cars.

```
int total_dealer_value(struct Dealer *dealer);
```

`mean_car_price` returns the average selling price for the cars of a given dealer.

```
int mean_car_price(struct Dealer *dealer);
```

Now there is a third struct:

```c
struct Customer {
  char desired_brand[20];
  char desired_model[20];
  int account_balance;
  struct Car *car;
};
```

3) Customer Initialization

`init_customer` initializes a customer (malloc/calloc) with a null-pointer as its car.

```
struct Customer *init_customer(struct Customer *dealer,
int account_balance, char desired_brand[32], char desired_model[32]);
```

4) Customer-Dealer Analysis

`customer_car_rating` returns:

- `3` if the given car matches the desired brand and model and the customer can afford it
- `2` (only if not `3`) if the given car matches the desired brand at a price which the customer can afford
- `1` (only if not `2` / `3`) if the customer **can** afford the given car
- `0` (only if not `1` / `2` / `3`) if the customer **can't** afford the given car

```
int customer_car_rating(struct Customer *customer, struct Car *car);
```

`car_available` returns the best possible match for any car in a given dealers inventory for a given customer.

```
int car_available(struct Dealer *dealer, struct Customer *customer);
```

`buy_car` transfers the most favorable car from a dealer to a customer (most favorable = The cheapest car among the car with the (same) highest match rating). Update The customers `car` -pointer as well as the dealers `cars` -list as well as both account balances. Returns the match rating of the car being bought.

```
int buy_car(struct Dealer *dealer, struct Customer *customer);
```

5) Printing Functions

Print out a `car` in a representational format.

```c
void print_car(struct Car *car);
```

Print out a `dealer` in a representational format. You may reuse `print_car`.

```c
void print_dealer(struct Dealer *dealer);
```

Print out a `customer` in a representational format. You may reuse `print_car`.

```c
void print_customer(struct Customer *customer);
```

6) Memory Deallocation Functions

Free up the memory of a `dealer` and all its contents.

```c
void free_dealer(struct Dealer *dealer);
```

Free up the memory of a `customer` and all its contents.

```c
void free_customer(struct Customer *customer);
```

See You Next Week!

All **code examples** and **exercise solutions** (available right after my tutorial) on **GitHub**.
https://github.com/dostuffthatmatters/Engineering-Informatics-1-MSE-WS1920.


THE AMOUNT OF WTFS PER MINUTE
IS TOO DAMN HIGH!
memegenerator.net