Prog C — td5

version **#0.0.1-dirty**

Assistants C/Unix 2024 <assistants@tickets.forge.epita.fr>

Contents

*https://intra.forge.epita.fr

1 Structures

1.1 Introduction

Structures are **user-defined data types** that allow us to combine data items of any type together. It is somewhat similar to an array, but structures can store data of different types.

Let's start with an example: when using an array, you may want to keep associated information, like the number of elements inside this array. To make this possible, you would declare an array and on the side, an integer which forces you to maintain the two variables separately. However, these variables are co-dependent. Ideally, we want to combine the array and the integer to make it easier to update accordingly. This is what structures are for.

Suppose we want to implement coordinates in a three dimensional euclidean plan. We need a structure with three fields of type int.

```c
struct coord
{
     int x;
     int y;
     int z;
};
```

`struct` is the *C* keyword associated with structures and used with both definitions and declarations. `coord` is the name of the defined structure meanwhile `x` is one of its **fields**.

1.2 Initialization

Structures can be used like any other types. Therefore, there are no differences with other types when dealing with declarations.

```c
struct coord c;
```

`struct coord` defines the actual type of the variable and `c` is the variable's name. Initializing it this way will leave all its fields blank, thus we need to fill them. One way to define and put data in the fields would be:

```c
struct coord c =
{
     .x = 1,
     .y = -2,
     .z = 10,
};
```

1.3 Accessing structure members

Since a structure is a group of data, we need to be able to access each field individually. We use '**.**' or '**->**', which are called **accessor**, to both read and write in our structures. We use **.** when the structure is not a pointer and **->** when it is.

Here is an example of writing data inside a structure, using both `accessors`:

```c
struct coord c;

c.x = 10;
c.y = 5;
c.z = -2;

struct coord *c_ptr = &c; /* use the address of c */
c_ptr->x = 15;
c_ptr->y = -54;
c_ptr->z = 42;

// Set all other fields in the same way
```

Here is an example of reading data from a structure:

```c
int x = c.x;
int y = c.y;
int z = c.z;
```

Tips

The syntax `ptr->field` for pointer is a syntactic sugar[1] over `(*ptr).field`

let's see an example:

```c
struct coord c = {.x=1, .y=3, .z=4};
struct coord *c_ptr = &c;

int a = c_ptr->x;
int b = (*c_ptr).x;

int d = a == b; // true
```

[1] Syntatic sugar is a syntax within a programming language that makes a contruct easier to read or to express.

1.4 Function arguments

Since our structure is a type, it can of course be passed as a function argument. Let's say that we want to get the field `x` of our coord.

```c
int get_coord_x(const struct coord *coord)
{
    return coord->x;
}
```

Tips

As seen in the constness section, in a function where we do not modify the variable, it is a good practice to use the `const` keyword so that the compiler knows it should never be modified in the function. Correctly used, it can ease the debugging of a program.

Be careful!

When passing a structure as an argument, it is most of the time better to pass it as a pointer. This is because the structure can be very large and copying it can be very expensive. Passing it as a pointer will only copy the address of the structure and not each of its fields, which is much faster and lighter.

1.5 Practice

- Create a structure `complex` representing a complex number which has two `int` fields: `re` and `im` representing the real and imaginary part of the complex number.
- Write a function `print_complex` printing the value hold by pointer to a complex number given as argument.
- Write a function `add_complex` taking two pointers of complex number as arguments and returning the sum of their value.
- Write a function `mul_complex` taking two pointers of complex number as arguments and returning the product of their value.

Here is an example of an output for a given `main`:

```c
#include <stdio.h>

int main(void)
{
    struct complex a = {.re = 3, .im = 5};
    struct complex b = {.re = -4, .im = 2};

    print_complex(&a);
    print_complex(&b);
    print_complex(add_complex(&a, &b));
    print_complex(mul_complex(&a, &b));

    return 0;
}
```

...will print:

```
3 + i * 5
-4 + i * 2
-1 + i * 7
-22 + i * -14
```

1.6 Headers

As you saw previously, to have a clear and great files organisation. the *C* language has headers files (`.h`). You have seen you can declare you functions prototypes inside the headers files.

Another usage of these files is to declare the structures inside too. This allows you to share you structures over you whole project.

By example:

```c
/**
** \file coord.h
*/

#ifndef COORD_H
#define COORD_H

struct coord
{
     int x;
     int y;
     int z;
};

#endif /* ! COORD_H */
```

1.7 Vectors

1.7.1 Explanation

You have just seen structures so let's use them in combination with arrays. A **vector** is basically an array associated with its size.

Suppose we define a structure that contains an array of integers and the number of elements stored as an int.

This data structure used for integers will look like the one below:

```c
struct int_vector
{
    int array[64];
    int size;
};
```

1.7.2 Practice

Vector min

Goal

Write a function that takes a vector of integers and returns the minimum value it contains. The size of the table will always be correct and superior to zero.

Prototype:

```
int int_vector_min(struct int_vector vec);
```

Vector max

Goal

Write a function that takes a vector of integers and returns the maximum value it contains. The size of the table will always be correct and superior to zero.

```
int int_vector_max(const struct int_vector vec);
```

Structure of the vector:

```
struct int_vector
{
    size_t size;
    int data[INT_VECTOR_LENGTH];
};
```

I must not fear. Fear is the mind-killer.