Piscine — Tutorial D3

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

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

Copyright

Contents

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

1 Structures

1.1 Problem

Let's say we want to develop a function that takes two points in a plane and computes the distance between them. The prototype of the function could look like this:

```c
float distance(float p1_x, float p1_y, float p2_x, float p2_y);
```

A point has only two coordinates and the prototype of this function is already very long. Imagine how long it would be if we were in three dimensions! Another issue is the lack of cohesion between values. For example, `p1_x` is technically not related in any way to `p1_y`. We need a way, to regroup related variables under one name we can use in our code.

At this point in time, the only types you have seen are **primitive** data types: they are types built in the language and can be used as basic data (integer, float...). You will see by practicing, that programming problems and concepts will become complex. So, the question is: *How to represent these concepts in our type system?*

1.2 Syntax

Let's go back to our cartesian point. One point is composed of:

- A float `x`, the field which represents the abscissa.
- A float `y`, the field which represents the ordinate.

In *C* , we can define a new type for this concept as follows:

```c
/* A point is both: */
struct point
{
    float x; /* an "x" field...       */
    float y; /* ...and a "y" field  */
}; /* Don't forget the trailing semicolon! */

struct point p; /* p is a variable of type struct point */
p.x = 42;       /* The x field of p is 42 */
p.y = 51;       /* The y field of p is 51 */
```

Therefore, we have defined a new type, `struct point`, which can be used like any other type:

```c
void point_print(struct point p)
{
    printf("This point is (%f, %f)\n", p.x, p.y);
}
```

Our method `distance` could then be written like this:

```c
float distance(struct point p1, struct point p2);
```

The great force of the structure definition, is that we can use it to create even more derived types! For example, let's create a segment, composed of two `struct point`.

```
struct segment
{
    struct point p1;
    struct point p2;
};
```

You can also declare a structure along with a variable of this type in one go using the following syntax:

```
struct foo
{
    int bar;
    // other fields
} foo_var;
foo_var.bar = 42;
```

1.3 Size of a structure

It can be interesting to compute the size of a structure. For example, what is the size of this structure?

```
struct trap
{
    int i;
    char c;
};

printf("%zu\n", sizeof(struct trap)); // What will this print?
```

At first we could say that the size of a structure is equal to the sum of its fields. Here, if we suppose that `sizeof(int) == 4`, the size of the structure will be 5 bytes (the size of a char is **always** one byte). Try this, and you will see that this is wrong!

Be careful!

You may notice that the size of this structure is actually 8. It is due to compiler's optimization that will favor memory access on *aligned addresses*. Thus, the compiler will add padding bytes to make fields' addresses aligned.

If it is not clear for you, just remember that **the size of a structure is at least the sum of its fields,** but can be greater for performance reasons.

1.4 Array of structures

Like we said in the first part, a structure is used to store information about one particular object. But if we need to have more structures of this particular object, then an array of structures is needed.

For example:

```
struct student
{
    char name[24];
```

(continues on next page)

(continued from previous page)

```
    int age;
    int average_grade;
} students[100];
```

Here, students[0] stores the information of the first student, students[1] stores the informations of the second student and so on.

In order to access the age field of the third student, you just have to do:

```
int age = students[2].age
```

If you need to initialize an array of struct, it is done the same way as primitive types. Also, in a structure initializer, you can specify the name of a field by adding “.fieldname =” before the element value:

```
struct student
{
    char name[24];
    int age;
    int average_grade;
};

struct student students[] = {
    { .name = "Antoine", .age = 21, .average_grade = 19 },
    { .name = "Paul", .age = 21, .average_grade = 19 },
};
```

1.5 Exercise

The use of structures is simple. We advise you to take the next exercise seriously, as it will prove useful for the rest of your *C* studies.

1.5.1 Pairs

Goal

In this exercise, we will use a simple pair structure containing integers:

```
struct pair
{
    int x;
    int y;
};
```

Write the following functions:

```
struct pair three_pairs_sum(const struct pair pair_1, const struct pair pair_2,
                            const struct pair pair_3);
```

`three_pairs_sum` takes three `pairs` and sums each x field together, and each y field together. The summed x and y fields are stored into a new structure which is returned by the function.

```
struct pair pairs_sum(const struct pair pairs[], size_t size);
```

`pairs_sum` takes an array of `size` `pairs` and sums each x field together, and each y field together. The summed x and y fields are stored into a new structure which is returned by the function.

2 Enumerations

2.1 Problem

Let us imagine you are coding a video game, and you wish to represent a direction in which a character is moving: it can go to the north, south, east, or west. How can we represent this?

Possible solutions here could be using either strings or numerical values to represent each direction, but these have inherent issues:

- If we choose to go with strings, we end up having to use much more memory than really needed to store every character in every possible string, and comparing strings to check if they are identical is non-trivial and takes time. Think about `strcmp`: in the worst-case scenario you have to go through the whole string to realize they differ at the last character. This might not be much, but if you do a lot of comparisons with multi-character strings, the number of operations will sum up over time.
- If we choose to go with numerical values, all these optimization concerns go away but at one cost: readability. Now you have to remember that 1 means “north”, 2 means “west” and so on. Introducing such *magic values* makes the code harder to read, and if it is harder to read it is harder to debug and more prone to bugs. Magic values should be avoided at all cost.

Fortunately for us, the *C* language has what we need: **enumerations**.

2.2 Definition

Enumerations are a user defined data type which can be used to name arbitrary values in order to make code easy to read and maintain.

If we go back to our video game example, we could use an enumeration like this one:

```
/* A direction is either: */
enum direction
{
    NORTH, /* this value    */
    SOUTH, /* OR this value */
    EAST,  /* OR this one   */
    WEST,  /* OR that one   */
}; /* Again, trailing semicolon! */
```

Tips

As with macros, we usually name enumerations in capitals. It helps differentiate them from variables and function names.

Similarly to an integer having multiple possible values {0, 1, 2, 3, ...}, `enum direction` is a type that has four possible values: {`NORTH`, `SOUTH`, `EAST`, `WEST`}. Their usage is very simple:

```c
void print_direction(enum direction dir)
{
    /* A switch is also commonly used to work with enums */
    if (dir == NORTH)
        puts("I'm facing north!");
    else if (dir == SOUTH)
        puts("I'm facing south!");
    /* ... */
}

int main(void)
{
    enum direction my_dir = SOUTH;
    print_direction(my_dir);
    return 0;
}
```

Enumerations are actually named integers so handling them is mostly like handling `int` values: comparing them takes no time, and they do not take much space memory-wise, with the added benefit that the code is clear and easily understandable.

The consequence is that when defining an `enum`, it is possible to assign specific numerical values to each enum value, which can be useful for instance when working with error codes.

For example, we could use the following enum to represent error codes for `ls`:

```c
enum error_code
{
    OK = 0,
    MINOR_ERR = -1,
    MAJOR_ERR = -2,
};

int main(void)
{
    enum error_code err = OK;

    // Main code goes here, change err value if needed...

    switch (err)
    {
    case OK:
       puts("No problem detected.");
       break;
    case MINOR_ERR:
       puts("Minor problem detected.");
       break;
    case MAJOR_ERR:
       puts("Major problem detected.");
       break;
    }
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Going further...

Enumerations actually serve quite a similar purpose to macros: avoiding magic values. That being said, when dealing with multiple possible values or states for a similar concept, one should prefer using enumerations over macros due to the added benefit of typing. Enumerations define a custom type which can be used to express more meaning in program, whereas macros are only a shortcut to a value and are not typed per se.

For example: when doing a switch on an enum, the compiler is smart enough to be able to determine whether all possible values are handled, even without having a default case. This proves quite useful as if we were to add a possible value to a given enum, all switches on this enum would trigger a warning until that new case is taken into account. This provides better code safety, and is really useful for maintaining code.

When no integer values are associated with the enumerations, the default value of the first one is guaranteed to be 0, and the following values are increments of their predecessors.

```
enum only_default
{
    A,
    B,
    C,
};

enum non_default
{
    D,
    E = 41,
    F,
};

int main(void)
{
    printf("%d\n", A); // prints 0, A has default value of 0
    printf("%d\n", B); // prints 1, B follows A so its value is 0 + 1 = 1
    printf("%d\n", C); // prints 2, C follows B so its value is 1 + 1 = 2
    printf("%d\n", D); // prints 0, D has default value of 0
    printf("%d\n", E); // prints 41, E has a specific value
    printf("%d\n", F); // prints 42, F follows E so its value is 41 + 1 = 42

    return 0;
}
```

Going further...

It is possible for two values of an enumeration to have the same numerical value. The following code is valid:

```c
enum my_enum
{
    ZERO,
    ONE,
    ALSO_ONE = 1,
};

int main(void)
{
    printf("%d, %d, %d\n", ZERO, ONE, ALSO_ONE); // prints 0, 1, 1

    return 0;
}
```

3 Macro Genericity

3.1 What is genericity?

Generic code is code defining behaviour without taking specific types into account. A generic function behaves the same no matter the type of its arguments. This allows to write an algorithm once that will work for every possible type as input, which greatly reduces the need for code duplication.

Although generic code is arguably not the main focus of the *C* language, there are a few ways to write generic code in *C*, which we will see progressively.

The first and most straight-forward way is using macros.

3.2 Macro expansion

The main mechanism behind macros is called *macro expansion*. The idea is that macros will be replaced by their value at compile-time. Although this is already useful to eliminate *magic values* as you have seen earlier this week, this also allows to write generic code thanks to parameter macros.

When a macro with parameters is expanded, its whole body is inserted where it is called, no matter the type of its parameters as we can see in the example below. Hence, the same macro can be used for multiple parameter types: it is **generic**.

```c
#include <stdio.h>

#define TOTO int tata

int main(void)
{
    TOTO = 42;
    printf("The answer is %d\n", tata);

    return 0;
}
```

We can see what the code would look like with macro expansion using the **-E** flag of gcc, which allows us to not run the `linker` and stop after the `preprocessing` stage.

```
42sh$ gcc -Wall -Werror -Wextra -Wvla -std=c99 -pedantic -E main.c
...
# 5 "main.c"
int main(void)
{
    int tata = 42;
    printf("The answer is %d\n", tata);

    return 0;
}
```

3.3 Macro with parameters

Macros with parameters, often called function-like macros, can be defined using the following syntax:

```
#define MACRO_NAME([[Parameter, ]* Parameter]) replacement-text
```

When the preprocessor expands such a macro, it incorporates the arguments you specify. Sequences of whitespaces before and after each argument are not part of the argument.

The parameter list is a comma-separated list of identifiers for the macro's parameters. When you use a function-like macro, you must use as many arguments as there are parameters in the macro definition.

Moreover, make sure that there is no whitespace between the macro's name and the `(`. If there is a whitespace, it will be expanded as a macro without parameters.

As the following examples illustrate, you should generally enclose your parameters and your replacement text in parentheses.

```
#define SUM ((2) + (2))
#define UNSAFE_SUM (2) + (2)

int foo = 10 * SUM;        // foo = 10 * ((2) + (2)) = 40
int bar = 10 * UNSAFE_SUM; // bar = 10 * (2) + (2) = 22
```

```
#define MULT(A, B) ((A) * (B))
#define UNSAFE_MULT(A, B) A * B

int foo = MULT(10 + 1, 10);        // foo = ((10 + 1) * (10)) = 110
int bar = UNSAFE_MULT(10 + 1, 10); // bar = 10 + 1 * 10 = 20

float baz = MULT(-3.0, 2.0); // this generic macro can be used with multiple types
```

3.4 Macro with parameters (cont.)

Moreover, a *function-like* macro is not a function. It is still replacement text. Therefore, be careful of calls with side-effects. For example, take the absolute function-like macro.

```c
#define ABS(A) (((A) < 0) ? -(A) : (A))

int main(void)
{
    int i = -42;
    int j = ABS(++i);  // j = 40
}
```

As the macro is just text replacement, `++i` is called twice. This is not the same behavior as a function.

Be careful!

The point of macros is **not** to make your code faster. Although depending on the context they *might* make it faster, this will not always be the case, depending on how the compiler optimizes your code. Macros do come with disadvantages, mostly that they are not strongly typed, which is useful when you need genericity but can make them hard to debug.

Whenever possible, always prefer writing functions over writing function-like macros.

Going further...

C programmers often write `static inline` functions in header files as a substitute for function-like macros. You will see this in more detail later on.

3.5 Exercises

3.5.1 Goal

You must write to a file named `macro.h` the following macros:

```c
#define MIN(A, B) ...
#define MAX(A, B) ...
```

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