PROG C — td2

version **#0.0.1-dirty**

ASSISTANTS C/UNIX 2024 <assistants@tickets.forge.epita.fr>

Copyright

Contents

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

1 Arrays

An array is a collection of elements **of the same type**. Each element is identified by an index specifying its position within the array.

1.1 Declaration

```
type var_name[N];
```

With `N` being the total number of items that can be stored in the array. `N` is called the dimension of the array and is always a positive integer. Moreover `N` **must** be known at compile time.

Be careful!

Once the size of an array is fixed, it can **not** be changed anymore. There is actualy a solution to this problem but the concepts needed to understand it will be introduced later on.

Going further...

Compile time refers to the period when the code is converted to machine code, whereas the run time is when the program is executed. Literals such as `10` or `'a'` are values known at compile time. These notions will be further explained later, for now you can look at the Compile Time page on Wikipedia

Example:

```
int tab[8];
```

Here, `tab` is an array that contains eight integers.

Tips

Elements in an array are designated by their indexes (which correspond to their position) in that array. In *C*, arrays are indexed from 0. This means that the indexes of their N elements go from 0 to N-1.

1.2 Initialization

An array can be initialized as any other variable.

```
int arr[5] =
{
    3, 42, 51, 90, 34
};
```

Or, by specifying only the first few elements of the array:

```
int arr[5] =
{
```

(continues on next page)

variable name:	arr				
elements:	**3**	**42**	**51**	**90**	**34**
index:	0	1	2	3	4

Table 1: int array of dimension 5

(continued from previous page)

```
    1, 2, 3
};
```

variable name:	arr				
elements:	**1**	**2**	**3**	**0**	**0**
index:	0	1	2	3	4

Table 2: int array of dimension 5 with only the 3 first elements initialized

The two non-specified elements are then initialized to 0. Thus, it is possible to initialize an array entirely to 0 this way:

```
int arr[24] =
{
    0
};
```

or this way:

```
int arr[24] = {};
```

However, the following syntax cause an *undefined behaviour* as we do not initialiaze the variable. We only declare it.

```
int arr[24];
```

Be careful!

The expression *undefined behaviour* means that the behavior for this action (in this case, initialization an array of length 24 without value) is not specified by the language, and therefore is dependant of the specific compiler implementation. This means that you do not know the values contained in the array. The execution may continue, potentially leaving your program in an erroneous state.

It is also possible to omit the dimension of an array if (and **only** if) you completely initialize it during its definition. The dimension will then be determined based on the number of values:

```
int arr[] =
{
    3, 42, 51, 90, 34
};
```

Here, `arr` is an int array of dimension 5.

1.3 Accessing elements

To access an element of an array, we use the bracket operator `[]`:

```
arr[index]
```

- where `index` can go from `0` to `N - 1` (`N` being the dimension of the array).
- `index` can, for example, be an arithmetic expression.

```
int arr[5] =
{
    1, 2, 3, 4, 5
};
int a = 0;

a = arr[2];          /* OK */
a = arr[3 + 1];      /* OK */
a = arr[5];      /* Undefined behaviour */
```

variable name:	arr					
elements:	**1**	**2**	**3**	**4**	**5**	**?**
index:	0	1	2	3	4	5

Table 3: int array of dimension 5 with access to index 5

The bracket operator `[]` is also used to assign a value in an array:

```
int arr[5] =
{
    1, 2, 3, 4, 5
};
```

```
arr[2] = 42;         /* {1, 2, 42, 4, 5} */
```

Going further...

When manipulating arrays, it is a good practice to store values like indexes or array length into `size_t` variables. The `size_t` type stores only positive integers and is defined in the `stddef.h` header.

1.3.1 Practice

Now that you know how to handle arrays in **C**, let's practice.

- What is the index of the first value of an array in C ?
- Is this initialisation correct ? If it isn't, why ?

```
int arr[5] =
{
    0, 1, 2, 3, 4, 5
};
```

- What is the value of `a` in the following example ?

```
int arr[6] =
{
    1, 2, 3, 4, 5, 6
};

int a = arr[2];
```

- And in this one ?

```
int arr[6] =
{
    1, 2, 3, 4, 5, 6
};

int a = arr[6];
```

1.4 The main function

If arguments are to be passed to a program, the prototype of the `main` function will be as follows:

```
int main(int argc, char *argv[])
```

`argc` contains the number of arguments given to the program, plus one (because the first argument is the program's name), and `argv` is an array of strings containing the arguments passed to the program.

Going further...

You can read `char *argv[]` as "`argv` is an array of `char *`". This is actually true and you will see later that `char *` corresponds to the string type. This means that `argv` is an array of strings.

Here is an example of passing arguments to a program:

```
42sh$ ./path/to/my_prog toto titi tata
```

In the above example, `argc` will have the value `4` and `argv` will contain the following:

- `argv[0] = "./path/to/my_prog"`
- `argv[1] = "toto"`
- `argv[2] = "titi"`
- `argv[3] = "tata"`
- `argv[4] = NULL`

Tips

`argv` is always `NULL` terminated, meaning that `argv[argc]` is always `NULL`. `NULL` is a special value we will see later on.

1.4.1 Practice: Max array

Goal

Write a function that returns the maximum value of an array of integers given as argument. If the array is empty you should return INT_MIN. The size of the array will always be correct.

```
int max_array(const int array[], size_t size);
```

1.4.2 Practice: Array vice max

Goal

Write a function that returns the vice-maximum (the *second* largest value) of an array of integers given as argument. Assume that the vector always contains at least two elements, that its size will always be correct and that all elements will have a different value.

Prototype:

```
int array_vice_max(const int vec[], size_t size);
```

2 Recursion

2.1 Introduction on Algorithms

An algorithm is, by essence, a sequence of instructions, describing how to perform a task. In the field of computer science, we could be more specific when defining this notion. One must understand that an algorithm always has a purpose, it is written to solve a precise and well defined issue. Moreover, one must keep in mind that efficiency is a key goal to factor in when writing algorithms.

> [...] we want good algorithms in some loosely defined aesthetic sense. One criterion [...] is the length of time taken to perform the algorithm [...]. Other criteria are adaptability of the algorithm to computers, its simplicity and elegance, etc
>
> ---*The Art of Computer Programming: Introduction to Algorithms* - **Donald Knuth**

2.2 Definition

> Many useful algorithms are **recursive** in structure: to solve a given problem, they call themselves one or more times to deal with closely related subproblems. [...] They break the problem into several subproblems that are similar to the original problem but smaller in size, [...] and then combine these solutions to create a solution to the original problem.
>
> ---*Introduction to Algorithms (3rd edition)* - **Cormen, Leiserson, Rivest, Stein**

In other words, recursion is a programming technique where a function or an algorithm **calls itself** until a stopping condition is met.

At each call, the input changes from the last input call, until it matches the stopping condition. At this point, the function returns a value and the recursion is completed, from the last call to the first one.

Thus, a generic pattern of a recursive function can be defined with four different parts:

1. A **stopping condition** (to prevent infinite recursion)
2. Pre-order operation (before the `recursive call`)
3. The **recursive call**
4. Post-order operation (after the `recursive call`)

2.3 Example: print_sequence

Here is an example:

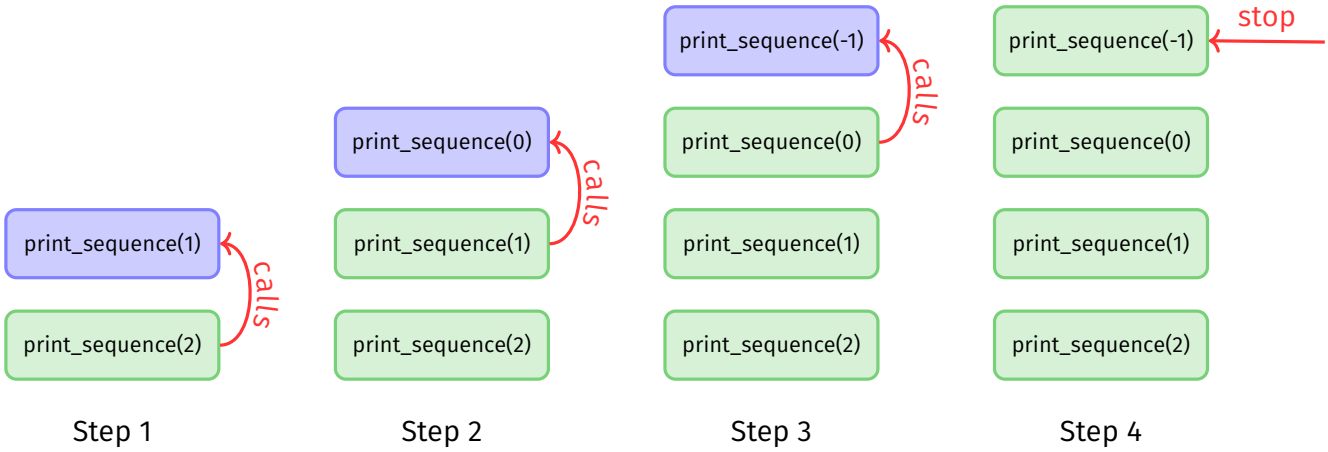
```c
#include <stdio.h>

void print_sequence(int x)
{
    if (-1 == x) /* Stopping condition */
        return;

    // A pre-order operation could be here
    print_sequence(x - 1); /* Recursive call */
    printf("%d\n", x); /* Post-order operation */
}

int main(void)
{
    print_sequence(100);
    return 0;
}
```

As you see above, the function `print_sequence(int x)` has a condition that will stop the recursion when `-1 == x` and a call to itself `print_sequence(x - 1);` inside its body. Let us see what it does:

```
42$ ./print_sequence
0
1
2
3
4
[...]
97
98
99
100
```

The function will print all integers from `0` to `x`.

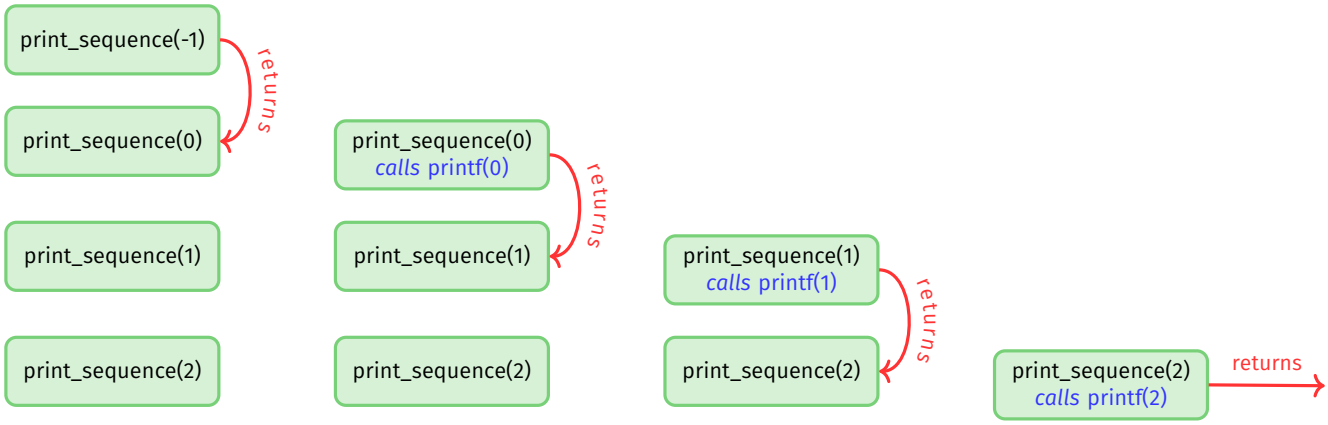
Let us see what happens in detail what happens if we call `print_sequence(2)`.



Step 1 Step 2 Step 3 Step 4

- In Step 1, we have the first call to `print_sequence(int x)` with `x = 2`. It calls `print_sequence(1)`.
- In Step 2, the same happens, except it is `print_sequence(0)` that is called.
- In Step 3, `print_sequence(0)` calls `print_sequence(-1)`.
- In Step 4, `print_sequence(-1)` reaches the stopping condition (`-1 == x`).

Thus, it returns.



Step 5 Step 6 Step 7 Step 8

- In Step 5, as `print_sequence(-1)` is fully executed, `print_sequence(0)` can be processed.
- In Step 6, `print_sequence(0)` executes the next instruction, which is `printf(0)`, and then it returns.
- In Step 7, same as Step 6, it executes `printf(1)`, and returns.
- In Step 8, finally, `print_sequence(2)` will print 2 and return.

The recursion ends.

2.4 Example: print_sequencev2

Let us continue the previous example:

```c
void print_sequencev2(int x)
{
    if (-1 == x) /* Stopping condition */
        return;

    printf("%d\n", x); /* Pre-order operation */
    print_sequencev2(x - 1); // Recursive call
}
```

The function `print_sequencev2(int x)` is a little different than `print_sequence(int x)`. First we print the integer, then we make the recursive call. Try to understand why the two functions have different execution flows.

The examples above demonstrate that the instructions before the recursion are executed from the first call to the last call. Whatever comes after the recursion is executed from the last call to the first call.

It is crucial to have a stopping condition in recursive functions. Try the following function `infinite_recursion(void)` below and see what happens.

```c
void infinite_recursion(void)
{
    infinite_recursion();
}

int main(void)
{
    infinite_recursion();
    return 0;
}
```

When you execute this program, it will respond with `Segmentation fault (core dumped)`. Each call to the recursive function will occupy space in memory until the maximum capacity is exceeded, causing the program to be aborted.

Tips

In case the execution of your program is too long, you can press `Ctrl + C` in your terminal in order to stop the execution.

Going further...

Later you will see what a `Segmentation fault` is and how it is triggered. For now, you just need to know that a `Segmentation fault` is triggered because your program has had a problem with its memory.

Here is another example with a function returning a numerical value, computing the n-th power of two:

```c
int pow_of_two(unsigned n)
{
```

(continues on next page)

(continued from previous page)

```c
    if (0 == n) /* Stopping condition */
        return 1;

    return 2 * pow_of_two(n - 1);
}

int main(void)
{
    int res = pow_of_two(3);
    printf("Result : %d\n", res);
    return 0;
}
```

Tips

The `unsigned` qualifier means that the variable can only take positive values.

Going further...

If you try to run the above program with the value 31, you will cause an **overflow**. It happens when you try to store a value larger than the maximum value the variable can hold.

```c
// int takes value from [-2^31, 2^31 - 1] or [-2 147 483 648, 2 147 483 647]
int i = 2 147 483 647; /* i = 2^31 - 1 */
i = i + 1; /* i = -2 147 483 648 */
// Here i = 2^31 - 1 + 1 = 2^31. However int takes value until 2^31 - 1.
// Hence, here i overflowed and started again the range of possible value.
// It took the first value available, that is -2^31.

// char takes value from [-2^7, 2^7 - 1] or [-128, 127]
char c = 120;
c = c + 30; /* 120 + 30 = 128 + 22 = -128 + 22 = - 106 */
```

Be careful!

Take special care about the stopping condition of your recursion. You need to handle all the cases properly, otherwise you will have an infinite recursion with the same error as above (`Segmentation fault`). For example, what happens if, in the last example, you call `pow_of_two` with the value -2 ?

Going further...

Let us talk about tail recursion.

A tail recursive function is a function in which recursive calls are the last evaluated expressions. As such, a recursive call cannot be part of another expression.

Let us takes the factorial function as example:

```c
unsigned long facto(unsigned long n)
{
    if (n <= 1)
        return n;
    return n * facto(n - 1);
}
```

In this example, `facto(n - 1)` is not the last evaluated expression, it is `n * facto(n - 1)`. Therefore this function cannot be considered as a tail recursive function.

But, why tail recursive function are different from normal recursive function ?

Some compilers and interpreters can use these functions to perform an optimization called `tail call optimization`. They will transform the tail recursive function into an iterative process, a loop.

This optimization solves the two mains issues of recursion: the performance and the call stack limitation.

The performance issue is about the cost of a function call, you may not see it now, but function calls are expensive operations compared to a loop. Therefore, for the same computation, recursion will be slower than iteration.

About the call stack limitation. Your computer has a range of reserved memory called the **stack**. This is where some variables are stored and where function calls are stored too.

When multiple function calls are stacked in the stack we call this a **call stack**. You have an example of **call stack** in the diagrams above. Recursion can create huge call stack and this is a problem because the stack has a limited size. Therefore, if you stack more function calls than the stack can handles, it will lead to a stack-overflow error. You don't have this problem with loops because it does not need the stack.

That is why **tail call optimization** and **tail recursive** functions are great. With the translation into a loop, we erase the function calls, which removes all risks of stack-overflow errors and increases the performance of the function.

Here is by example the tail recursive version of `factorial`:

```c
unsigned long facto(unsigned long n, unsigned long result)
{
    if (n <= 1)
        return result;
    return facto(n - 1, n * result);
}
```

2.5 Exercises

2.5.1 Fact and fibo

For this part, you should do the exercises `Factorial` and `Fibonacci`.

Fact

Goal

Implement the function `fact` with the following prototype:

```
unsigned long fact(unsigned n);
```

It computes the factorial of `n` ($n!$) and returns the result. As a recall:

$$\forall n \in \mathbb{N}, n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n \neq 0 \end{cases}$$

You **must** use recursion (i.e.: loops are forbidden).

Fibo

Goal

Write the function that computes the *Fibonacci sequence* recursively. This sequence is defined by U_n as follow:

- $U_0 = 0$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

```
unsigned long fibonacci(unsigned long n);
```

Practice

Now that you have written fact and fibo in recursive, try to rewrite these programs in iterative.

Tips

The difference between a recursive and iterative program is that the recursive one calls itself, to execute instructions, while the iterative one uses loops.

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