Piscine — Tutorial D11

version #

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

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

1 Function pointers

You have learned that you can have pointers to integers, strings, arrays, or even structures. As it turns out, you can also have pointers to functions, store them, call them, etc. Well used, they can bring elegant solutions to factorize your code, or to write more generic code.

1.1 Creating a function pointer

The syntax to describe a function pointer type is the following:

```
return_type (*ptr_name)(argument types, ...)
```

As it can be bothersome to write such a complex type every time, an alias is usually used. For example, to manipulate binary operations like sum or difference, we could do:

```c
typedef int (*bin_op)(int, int);

int sum(int a, int b)
{
    return a + b;
}

int main(void)
{
    int (*op)(int, int);    // Verbose declaration
    op = sum;               // op is a variable

    bin_op op2 = sum;       // Using the alias
                            // op2 is a variable

    int res = op(2, 3);     // Call the function pointed by op
    res = (*op)(2, 3);      // The same call, without syntactic sugar

    int res2 = op2(2, 3);   // Call the function pointed by op2

    return res == res2;     // res and res2 are equal
}
```

Tips

Read this article for a technique to parse complex function pointer types, and use cdecl to play with the syntax.

2 Higher-order functions

2.1 Function as argument

It is possible for a function to take a function pointer as parameter. The pointer is called *callback*, because you are calling a function and asking it to call the function provided back.

It can be used for event-driven programming. For example, the `atexit(3)` function of the standard library takes a function pointer as parameter, and calls it when the program is about to end.

```
int atexit(void (*callback)(void))
{
    // Store the callback
    // ...
}
```

Function pointers can also be used for *functional programming*, to introduce genericity to the program. For example, the standard library function `qsort(3)` takes as parameter a comparator (a function that will be able to compare two elements of the array). This way, it is possible to sort forward, backward, in alphabetical order, or in any order with the same algorithm, just by changing the comparator. Here is the prototype of `qsort(3)`:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This function sorts the array in-place, with `nmemb` elements of size `size`. The order is defined by the comparison function `compar`. This genericity allows `qsort` to sort any type of data or even structures according to specific fields:

```
#include <stdlib.h>

struct student
{
    int uid;
    char login[8];
    int promo;
};

int compare_student(const void *a, const void *b)
{
    const struct student *e1 = a;
    const struct student *e2 = b;
    return (e1->uid - e2->uid);
}

int main(void)
{
    struct student *students;
    int nb_students = 0;

    // Add students
    // ...
```

(continues on next page)

(continued from previous page)

```
    // Sort by student's uid
    qsort(students, nb_students, sizeof (struct student), compare_student);
}
```

You also might want to take a look at the `bsearch(3)` function which takes a comparator as parameter and does a binary search. You can take a look at its man page for more information.

2.2 Returning a function

A function can also return a function pointer. This can be useful for example if you have a function pointer array, and you are looking for a specific one. In the next example, we declare a function `get_function`, which takes a name in parameter, and returns a binary operator (a function taking two `int`s as parameters and returning an `int`). The syntax to declare it is a bit convoluted, and is as follows:

```
int (*get_function(char *name))(int, int);
```

2.3 Combining both

Now that you have an idea of the syntax for function pointers, imagine a function that would take a function as parameter, and return another function. For example, have a look at the prototype of the `signal(2)` function:

```
void (*signal(int sig, void (*func)(int)))(int);
```

The syntax becomes rather obscure, so to clarify it, we will use aliases, as shown previously:

```
typedef void (*sighandler_type)(int);
sighandler_type signal(int sig, sighandler_type func);
```

2.3.1 Exercises

2.3.2 Goal

In this exercise you will write your first basic functional functions.

map

Write the `map` function, to apply a function (`func`) to every element of an `int` array.

```
void map(int *array, size_t len, void (*func)(int *));
```

For example:

```c
void times_two(int *a)
{
    *a *= 2;
}

int main(void)
{
    int arr[] = {1, 4, 7};
    map(arr, 3, times_two);
    // arr == {2, 8, 14}
}
```

foldr

As for the previous function, `foldr` takes a function to apply to each element of the array. However, the function also takes an accumulator as parameter, initially 0. For example, on the array `{1, 2, 3}`:

`foldr(sum, {1, 2, 3}) <==> sum(1, sum(2, sum(3, 0)))`

The call to `sum(3, 0)` is the new accumulator for `sum(2, 3)`. The function finally returns the last value of the accumulator (6 in our case).

```c
int foldr(int *array, size_t len, int (*func)(int, int));
```

foldl

`foldl` is similar to `foldr`, but traverses the array backwards:

`foldl(sum, {1, 2, 3}) <==> sum(sum(sum(0, 1), 2), 3)`

Here is the prototype:

```c
int foldl(int *array, size_t len, int (*func)(int, int));
```

3 Function Pointers (Advanced)

3.1 Reminders

The syntax to describe a function pointer type is the following:

```
return_type (*ptr_name)(argument types, ...)
```

3.2 Function Pointer Table

Let us take a look at a new usage of function pointer: function pointer tables.

This pattern is useful to factorize code and remove big switch control-flow structures.

A function pointer table is often created using an array of function pointers.

In this example, you will dig into the way to use this pattern and when it can help a lot to simplify code.

```
int calculate(const char operator, const int lhs, const int rhs)
```

Where the operator can be -, +, / or *.

The classic way to implement it would be:

```
int calculate(const char operator, const int lhs, const int rhs)
{
    switch(operator)
    {
    case '+':
        return lhs + rhs;
    case '-':
        return lhs - rhs;
    case '*':
        return lhs * rhs;
    case '/':
        if (rhs == 0)
            errx(1, "You cannot divide by 0");
        return lhs / rhs;
    default:
        errx(1, "The operation: %c is not implemented", operator);
    };
    return 0;
}
```

It could be implemented using a function pointer table:

```
typedef int (* const operation)(const int, const int);

int mul(const int lhs, const int rhs)
{
    return lhs * rhs;
}

int add(const int lhs, const int rhs)
{
    return lhs + rhs;
}

int div(const int lhs, const int rhs)
{
    if (rhs == 0)
        errx(1, "You cannot divide by 0");
    return lhs / rhs;
}
```

(continues on next page)

(continued from previous page)

```c
int sub(const int lhs, const int rhs)
{
    return lhs - rhs;
}

static const operation operations[] =
{
    ['*' - '*'] = mul,
    ['+' - '*'] = add,
    ['-' - '*'] = sub,
    ['/' - '*'] = div,
};

int calculate(const char operator, const int lhs, const int rhs)
{
    // Check for out of bounds, or missing operation handler
    if (operator < '*' || operator > '/' || !operations[operator - '*'])
        errx(1, "The operation: %c is not implemented", operator);
    return operations[operator - '*'](lhs, rhs);
}
```

The order of the functions in the operations array is really important, take a look at `man ascii` to understand the order.

In this example we have only 4 cases, but in a more complex situation the advantages of the function pointer table would be flagrant.

Going further...

Instead of having to deal with the operations between characters, we could instead use an enum to represent the type of the operation. This is especially useful when dealing with an AST for instance.

```c
enum oper
{
    ADD,
    SUB,
    MUL,
    DIV,
};

static const operation operations[] =
{
    [ADD] = add,
    [SUB] = sub,
    [MUL] = mul,
    [DIV] = div,
};
```

3.3 Exercise

3.3.1 Goal

The goal of this exercise is to implement a really simple command interpreter.

The different commands you need to implement are:

- `help`
- `hello`
- `print`
- `exit`
- `cat`

3.3.2 Help

The `help` command needs to output different information about the available commands:

```
42sh$ ./fctptr_cmd | cat -e
cmd$ help$
The available commands are:$
help$
hello$
print string$
exit$
cat file$
```

3.3.3 Hello

The `hello` command is simply writing hello:

```
42sh$ ./fctptr_cmd | cat -e
cmd$ hello$
hello$
```

3.3.4 Print

The `print` command is writing the given string on `stdout`:

```
42sh$ ./fctptr_cmd | cat -e
cmd$ print toto$
toto$
```

3.3.5 Exit

The `exit` command is just exiting the program.

3.3.6 Cat

The `cat` command is outputing the content of a file on `stdout`:

```
42sh$ echo tata > tata
42sh$ ./fctptr_cmd | cat -e
cmd$ cat tata$
tata$
```

3.3.7 Structure

To understand the power of function pointers you will use this structure to implement this intrepreter:

```
typedef int (*handler)(const char *arg1);
struct cmd
{
    handler handle;
    const char *command_name;
};
```

With this structure you can associate the name of the command and how to execute it.

3.4 Function Pointers for Genericity

Another good use of function pointers is to be generic. Let us imagine you are working on a generic linked list using `void*` elements. It would look somewhat like that:

```
struct list
{
    void *value;
    struct list *next;
};
```

Now for instance you might want your linked list to have a `deep_free` function which would free the list and all of its elements. A naive implementation would be:

```
void deep_free(struct list *head)
{
    if (!head)
        return;

    struct list *next = head->next;

    if (head->value)
        free(head->value);
```

(continues on next page)

(continued from previous page)

```
    free(head);
    deep_free(next);
}
```

While it might seem to work fine at first glance, this implementation has issues regarding the way it frees its elements. Using `free(3)` to free the values of the list is not necessarily appropriate since we could have values which are complex and need their own free function, or values which are not even `malloc(3)`ed and should not be `free(3)`ed. To solve this problem we need to specify the free function that should be used in `deep_free` like such:

```
typedef void (*const free_function)(void*);

void deep_free(struct list *head, free_function free_value)
{
    if (!head)
        return;

    struct list *next = head->next;

    if (head->value && free_value)
        free_value(head->value);

    free(head);
    deep_free(next);
}
```

Another equivalent version could be to directly specify the `free_value` function pointer in the definition of the structure like such:

```
typedef void (*const free_function)(void*);

struct list
{
    void *value;
    free_function free_value;
    struct list *next;
};

void deep_free(struct list *head)
{
    if (!head)
        return;

    struct list *next = head->next;

    if (head->value && head->free_value)
        head->free_value(head->value);

    free(head);
    deep_free(next);
}
```

This works fine in the case where every element of the list needs to be freed using the same function,

but if we want our list to have different types of elements it will obviously not work. To have such a list, we need to specify a free function for each element of the list, and the simplest way to do so is to have a specific `element` type which will bundle both the data value we want to store and its associated free function (as well as potentially other information we might need).

```
typedef void (*const free_function)(void*);

struct element
{
    void *data;
    free_function free;
};

void free_element(struct element *elt)
{
    if (!elt)
        return;

    if (elt->data && elt->free)
        elt->free(elt->data);

    free(elt);
}

struct list
{
    struct element *value;
    struct list *next;
};

void deep_free(struct list *head)
{
    if (!head)
        return;

    struct list *next = head->next;

    free_element(head->value);
    free(head);
    deep_free(next);
}
```

The interesting thing here is that if we can “package” specific behavior (here a free function) in a given structure, we can also package other more interesting ones, which can help factorize codes in some cases.

3.5 Exercise

3.5.1 Goal

You have to implement a *generic* insertion sort. It will take two arguments: an array of pointers and a **comparison function**. The array of pointers is **NULL** terminated to indicate the end. The function prototype is:

```
typedef int (*f_cmp)(const void*, const void*);
void insertion_sort(void **array, f_cmp comp);
```

The argument `comp` is a pointer to a comparison function which returns an integer lower, equal or greater than zero if the first argument is respectively lower, equal or greater than the second argument. For example, the function `strcmp(3)` matches this description.

Your implementation performance will be tested: don't try to trick us with a simple bubble sort.

4 Shell Functions

4.1 Syntax

A function is a kind of named code block, like you are used to in other programming languages.

```
hello()
{
  echo "Hello $1!"
}

# Also works fine
hello() {
  echo "Hello $1!"
}

# Same here, but the space after '{' and the ';' before '}' are mandatory when
# declaring a function on a single line (same goes for any code block)
hello(){ echo "Hello $1!";}
```

Going further...

You might sometimes encounter this alternative syntax:

```
function hello
{
  echo "Hello $1!"
}
```

This syntax works fine on *bash* and *ksh* and is quite common but it is **not** POSIX compliant. Always prefer the first syntax for scripting.

A function body **cannot** be empty, same goes for conditions and loop bodies. To call a function you just need to write its name, as for any command.

Arguments passed to the function will be accessible using positional parameters; the `$#`, `$*`, `$@` and `${n}` variables are overridden inside a function.

```
42sh$ hello() {
> echo "Hello $1!"
> }
42sh$ hello Brian
Hello Brian!
```

- function definition must always be placed **before** invocation.
- *recursion* is working (though it is pretty slow).
- function calls do not naturally create subshells, everything stays *global* inside functions, so any variable declaration or environment modification inside a function will remain after leaving it (apart from positional parameters which are restored).

4.2 Return value

To leave a function, you can use the builtin `return`, however it is only used to exit the function with a particular return code, meaning changing the value of `$?`.

You cannot return a variable, but two alternatives are available:

- Everything is global so you can simply set a variable inside your function and use it after calling your function.
- As said before, it has the same behavior as a command, so you can output the data you want to return and then save it into a variable with command substitution.

Here is an example for each case:

```sh
#!/bin/sh

# Here we will use command substitution to fetch our result, as it spawns a
# subshell 'res' will only be alive inside the command substitution

concat() {
  res=""
  for arg; do
    res="${res}${arg}"
  done
  echo "$res"
}

echo "concat res: $(concat 1 2 3 4 5)"
```

```sh
#!/bin/sh

# Here we simply call our function without a subshell, thus total res its
# value after the function call.

concat() {
  res=""
```

(continues on next page)

(continued from previous page)

```
  for arg; do
    res="${res}${arg}"
  done
}

concat 1 2 3 4 5
echo "concat res: $res"
```

In both cases the output is the following:

```
42sh$ ./script.sh
concat res: 12345
```

4.3 Exercise

4.4 Goal

Write a script taking a number as argument and printing the factorial of this number on the standard output.

```
42sh$ ./fact.sh
42sh$ echo "$?"
1
42sh$ for i in $(seq 1 5); do ./fact.sh "$i"; done
1
2
6
24
120
42sh$ echo "$?"
0
```

5 Functional Testing

At this point, you should be familiar with the concept of testing and why testing code is an essential part of coding. In particular, you have been introduced to *unit testing* which consists of testing each and every part of your software in isolation. Although this is often an efficient way of testing the code you write and make sure it behaves properly, sometimes testing parts independently can be insufficient, if not inappropriate. In some cases, we prefer to test the software we wrote as a whole: this is *functional testing*.

5.1 When to use functional testing

Functional testing is appropriate when you want to test your program *end to end*, like if it were used by a real user. You want to make sure that you get the correct output for given set of inputs. Keep in mind that functional tests do not replace unit tests, they actually complement each other.

Unit tests are often easy to write since they tend to be small and focus on specific behaviors that you want to test. Since they are small, they tend to be quite fast to execute and accurate when they crash: if you only test one behavior in a test and the test crashes, you know what makes it crash.

On the other hand, functional tests can be harder to write and will usually focus on the general behavior of the program. They tend to be more extensive and take more time to run than unit tests. They allow for large tests covering easily more cases than unit tests, which are more specific.

5.2 Writing functional tests

The easiest way to write a functional testsuite is using scripting languages, most frequently *shell* or *python*. For now we will focus on using *shell* scripts: this will allow us to easily handle the inputs and outputs of our programs on the command line, which will be convenient.

5.3 Example

Let us imagine we want to implement a clone of `echo(1)`. We will use the following `my_echo.c`:

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc != 1)
        printf("%s", argv[1]);

    for (int i = 2; i < argc; i++)
        printf(" %s", argv[i]);

    printf("\n");

    return 0;
}
```

We will therefore write a small shell testsuite to compare the outputs of `my_echo` and `echo(1)`. A quick way to do so is using `diff(1)`, which shows the differences between files given as argument. The first step of our test suite will therefore be to redirect the outputs of the `echo(1)` and `my_echo` to files. Now we can process the diff of both files using `diff(1)`.

Going further...

As always, do not hesitate to check the manpage of `diff(1)`, it has many options which can be of great use.

```
#!/bin/sh

REF_OUT=".echo.out"
TEST_OUT=".my_echo.out"

echo a b c > "$REF_OUT"
./my_echo a b c > "$TEST_OUT"

diff "$REF_OUT" "$TEST_OUT"
```

Going further...

Specifying the path of output files like such works but is not the cleanest way to proceed. It would be far better to have them be in your `/tmp` directory for instance in order to avoid clogging your working directory. `mktemp(1)` could be useful too.

OK, we got a basic testcase somewhat covered. Let us add more tests then!

```
#!/bin/sh

REF_OUT=".echo.out"
TEST_OUT=".my_echo.out"

echo a b c > "$REF_OUT"
./my_echo a b c > "$TEST_OUT"

diff "$REF_OUT" "$TEST_OUT"

echo 42 sh > "$REF_OUT"
./my_echo 42 sh > "$TEST_OUT"

diff "$REF_OUT" "$TEST_OUT"

echo -n a b c > "$REF_OUT"
./my_echo -n a b c > "$TEST_OUT"

diff "$REF_OUT" "$TEST_OUT"

echo test -n a b c > "$REF_OUT"
./my_echo test -n a b c > "$TEST_OUT"

diff "$REF_OUT" "$TEST_OUT"
```

We could add more tests here of course, but as you know adding tests through copy-pasting is far from optimal, we want to avoid code duplication. Thankfully, we know how to write functions in shell, so we can refactor this piece of code.

```
#!/bin/sh

REF_OUT=".echo.out"
TEST_OUT=".my_echo.out"

testcase() {
  echo $@ > "$REF_OUT"
```

(continues on next page)

(continued from previous page)

```
  ./my_echo $@ > "$TEST_OUT"

  diff "$REF_OUT" "$TEST_OUT"
}

testcase a b c
testcase 42 sh
testcase -n a b c
testcase test -n a b c
```

This example is really basic and can (should) be improved in many ways, but that is the main gist of what functional testing is like. Interesting things to add here would be better error message formatting, displaying more precisely which testcases succeed and which ones fail, storing tests in files, and so on.

6 Git internals

You have previously used high-level Git commands without particularly knowing how these internally work. In this section, we will take a closer look at Git's inner workings. This will be *extremely* useful to better understand how to use the tool, and to debug various situations you will certainly encounter during your ING1.

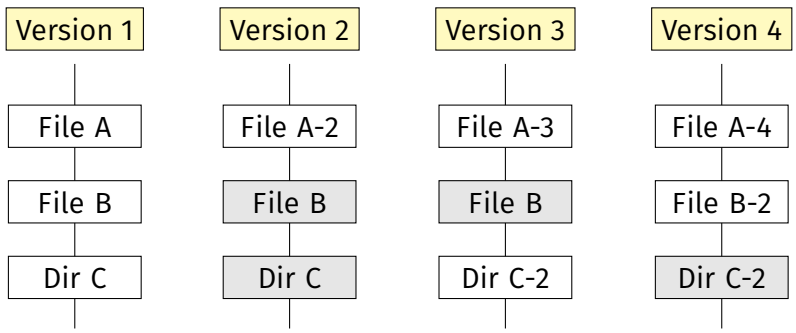
Git's philosophy is based on three core ideas:

- The notion of a snapshot
- The internal objects
- The index

6.1 Snapshot

Git is different from other VCS tools in its way to capture changes and data. To store a version of your project, Git acts as if it were taking a picture of every file and folder in your project at a specific time.

Obviously, files that have not been modified will not be stored multiple times. In the following figure, grey boxes represent files that were not altered. Git will thus use efficient compression techniques to reduce disk usage.



The most important notion to remember is that each saved version (more commonly called *commit*) is **self-sufficient** and contains all the information needed to describe the project in the state of the given commit.

6.2 Objects

All your information stored by Git is kept in internal objects. There are various types of objects:

- *blob*: pure data, blobs are literally the content of your tracked files.
- *tree*: an abstract layer that allows us to represent a hierarchy (folder and sub-folder structure). It contains a list of references to either blobs or other trees.
- *commit*: metadata related to some project changes (author, date, commit message, ...) and a reference to a tree object (corresponding to the new project state).
- *tag*: metadata related to some other objects (usually a commit).

Each object is stored in the `.git/objects` directory and referenced using a **unique** hash.

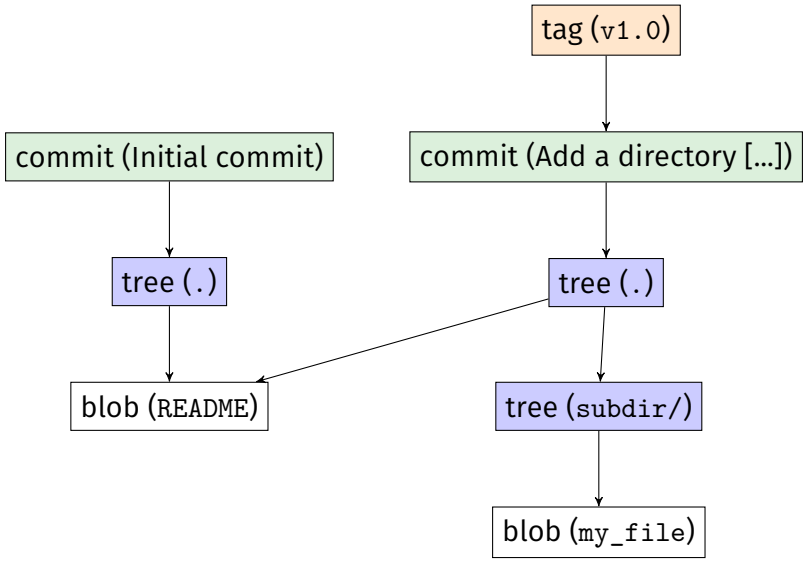
Here is a simple example of setting up a new repository, with a few objects:

```
42sh$ git init git-internals
42sh$ cd git-internals
42sh$ echo "This is a README" > README
42sh$ git add README
42sh$ git commit -m "Initial commit"

42sh$ mkdir subdir
42sh$ echo "This file is inside a directory!" > subdir/my_file
42sh$ git add subdir
42sh$ git commit -m "Add a directory with a file inside it"

42sh$ git tag -a v1.0 -m "My first tag!"
```

In the last example, 8 internal objects were created: 2 blobs, 3 trees, 2 commits and 1 tag. The tree objects were created automatically when we committed and added a directory, the rest was explicitly created by us. To better understand the current internal state of our repository, here is a visual representation of the objects relationships:



You can inspect each object content using the low-level Git command `git cat-file [OBJECT]`, either to show the object's type (using the `-t` option) or by printing its content (using the `-p` option).

Tips

To reference a Git object, you can use its unique hash. Most of the time, the first few characters of the hash is enough to identify it.

```
42sh$ # The first blob (or file) we have created:
42sh$ git cat-file b07f0ed953b8a24983dd5048cd2019b595692d74 -t
blob
42sh$ git cat-file b07f0ed953b8a24983dd5048cd2019b595692d74 -p
This is a README

42sh$ # The tree that was created during the second commit:
42sh$ git cat-file 95aee3cdb6514db4297f700aa0a6ead78fef877d -t
tree
42sh$ git cat-file 95aee3cdb6514db4297f700aa0a6ead78fef877d -p
100644 blob b07f0ed953b8a24983dd5048cd2019b595692d74        README
040000 tree d76b255ef131dd4a7e53f479948f0e63f108dc09        subdir

42sh$ # The first commit (the hash will differ because the metadatas are not the same)
42sh$ git cat-file cb46ab4e0e6a39c9b6c81eaade6eb0223deb18e6 -t
commit
42sh$ git cat-file cb46ab4e0e6a39c9b6c81eaade6eb0223deb18e6 -p
tree a7ed1c46b6237a06696a8b77a54de88f4f3094fb
author Xavier Login <xavier.login@epita.fr> 1632577080 +0200
committer Xavier Login <xavier.login@epita.fr> 1632577080 +0200

Initial commit

42sh$ # Finally, the tag we have created and pointing on the second commit:
42sh$ git cat-file bed521f6bb15bdde8303b8ef66db88fff8e871d3 -t
tag
42sh$ git cat-file bed521f6bb15bdde8303b8ef66db88fff8e871d3 -p
object e3370b06cf2a1f22559eb97dd5572300735321cc
type commit
```

(continues on next page)

(continued from previous page)

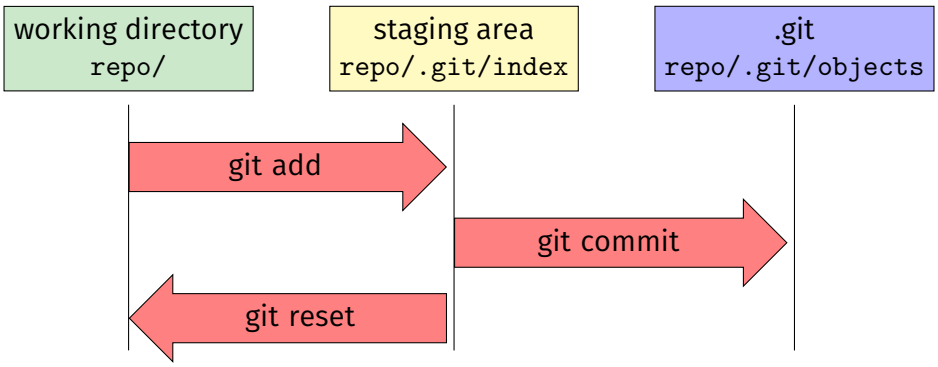
```
tag v1.0
tagger Xavier Login <xavier.login@epita.fr> 1632577100 +0200

My first tag!
```

Going further...

Take some time to explore an existing `.git/` directory. Read the Git documentation to understand the other folders and files inside a `.git/` folder.

6.3 Index



When you work on a repository, the folder you have on your machine is known as your *current working directory* and is a local version of your files and project.

When editing files, you might like to keep and save some changes: you will thus `git add` these particular changes. This action will move a list of files and store it in the Git *index*. This zone is also named the *staging area* and is used as a temporary buffer before creating a *commit* object.

Tips

It is possible to cancel some changes that were added to the index by using the `git reset` command.

Finally, when you are satisfied with the current changes in the index, the next step is to *commit*: the modifications on the files that were staged are now committed in your *local repository* and are part of your Git history.

Some Git commands have a `--staged` option available to better distinguish your intent. For example, running `git diff --staged` will show the difference between your latest commit and what is currently in the index, instead of simply showing the difference with your current working directory.

6.4 Going further

As always, we **highly** encourage you to dive deeper into the tools you will use on a regular basis to understand how to use them efficiently and how they work. The Pro Git book is easily the best and most complete resource you can find online. There are also several series of articles that are worth a look such as: The curious coder's guide to Git and Git from the Bottom Up.

It is my job to make sure you do yours.