# Astuces 42 par quelqu'un qui l'a pas fait

Julia PHAM BA NIEN

23 février 2024

# Table des matières

# 1 Les fichiers

Comme à 42 on n'a pas le droit à la stdlib C, il faut la recréer, ce qui inclue stdio.
Pour avoir plus d'info sur les fonctions que je vais lister, faites

```
man 2 <nom de la fonction> # si syscall comme "open"
man 3 <nom de la fonction> # si partie de la stdlib comme "memset"
```

Tout fichier est représenter par un nombre (int), son "file descriptor" qui va être abbrévié "fd" un peu partout.
Il y a l'entrée standard (stdin) ayant pour fd 0 (sous unix), la sortie standard (stdout) ayant pour fd 1 et la sortie d'erreur (stderr) ayant pour fd 2.
Vous pouvez mettre ça pour aider la lisibilité (mais on peut directement remplacer par ses valeurs, c'est pas grand chose à apprendre) :

```c
#define STDIN 0
#define STDOUT 1
#define STDERR 2
```

Pour obtenir le fd d'un fichier dont on a le chemin (relatif ou absolue), on utilise la fonction/syscall open.

```c
#include <fcntl.h>
const char *file_path = "le fichier";
// pour lire le fichier
int fd = open(file_path, O_RDONLY);

// pour écrire sur un fichier
int fd = open(file_path, O_WRONLY);

// il faut fermer le fd après l'avoir open
close(fd);
```

Pour écrire ou lire sur un fichier, il faut avoir une zone qui contient la mémoire ou qui peut la continir, on utilise "write" et "read"

```c
#include <unistd.h>

char buf[1000];

write(STDOUT, "hello, world\n", 13); // écrit 13 caractères de "
    hello world\n" dans le stdout

read(STDIN, buf, 100); // lis 100 caractères du stdin dans buf
```

Ces fonctions retournes le nombre d'octets effectivement lu/écrits.
Pour lire sur un fichier (pas le stdin) on peut retourner en arrière ou en avant avec lseek.

```c
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

où whence est soit SEEK_SET (le premier caractère), SEEK_CUR (là où on est dans la lecture), SEEK_END (la fin du fichier) et retourne l'offset depuis le début du fichier.

# 2 La norminette

La norminette, le truc pour te faire écrire du bon code qui t'oblige à faire des affreusités, un bon exemple de la loi de Goodhard : `https://fr.wikipedia.org/wiki/Loi_de_Goodhart`
Les exemples que je vais présenter sont assez courts pour que la norminette ne pose pas de problèmes, mais ce qui est important est que les idées peuvent se retranscrire sur des fonctions plus longues.
Ces techniques ont tendance à rendre le code "moins bon", donc à utiliser que lorsque cela est nécessaire.

## 2.1 Réduire le nombre de lignes

De ce que j'ai entendu, un des plus gros problèmes est d'écrire des fonctions assez petites.

### 2.1.1 Introduction de variables

On va essayer de réécrire le code suivant :

```
int iterfib(int n) {
  int a;
  int b;
  int c;

  a=1;
  b=0;

  while (n--) {
    c = a;
    b = a;
    a += c;
  }
  return b;
}
```

On remarque qu'on as besoin de 3 int, on peut donc le réécrire :

```
int iterfib(int n) {
  int t[3];

  t[0]=1;
  t[1]=0;

  while (n--) {
    t[2] = t[0];
    t[1] = t[0];
    t[0] += t[2];
  }
  return t[1];
}
```

Ce qui sauve 2 lignes !

Mais ce tricks semble ne marcher seulement pour les éléments du même type, mais avec un peu de connaissance sur la mémoire on peut le faire pour tous.
On peut transformer

```
int a;
float b;
double c;
char d[8];
long long *e;
```

En

```
char t[sizeof(int) + sizeof(float) + sizeof(char d[8]) + sizeof(
    long long *)];
```

(on peut remplacer par les valeurs si les contraintes des 80 caractères embête)
Et alors

```
a = *((int *) t);
b = *((float *) (t + 4)); // la taille de a
c = *((double *) (t + 8)); // la taille de a et b
d = (char *) (t + 16); // la taille de a et b et c, on ne deference
     pas
e = *((long long *) (t + 24)); // la taille de a et b et c et d
```

Ce qui avec une macro se fait

```
#define GET(type, mem, offset) (*((type *) (mem + offset)))
```

(mais les macros c'est interdit donc l'utilisez pas)

### 2.1.2 Variables constantes

Lorsque qu'on marque une variable en "const", on ne peut pas modifier sa valeur.
Donc on ne peut pas l'introduire, puis l'initialiser plus tard.
Ainsi, on a le droit de faire une introduction et initialisation en même temps, sur la même ligne !
Un bon example est la fonction memset 15

### 2.1.3 Opération prefixe et postfixe

Utiliser (et abuser) ++ et le double - postfixe, c'est bien.
Néanmoins la norminette empêche d'avoir des gros abus.

### 2.1.4 Tail Call Optimisation

Tout lisper est sensé le savoir, mais peu sont des lispers...
Si on a le code :

```
return_t function(args...) {
  if (cond(args...))
    return g(args...);
  return function(f(args...));
}
```

Ce code est équivalent (à usage de stack près) à :

```
return_t function(args...) {
  while (!cond(args...))
    args... = f(args...);
  return g(args...);
}
```

Normalement, on part de la première version vers la seconde, car elle n'utilise pas autant le stack et est plus rapide (et ne va pas StackOverflow).
MAIS, la première version prends moins de lignes, SURTOUT si il y a plusieurs variables et la transformation "args... = f(args...)" modifie beaucoup de variables interliées.
Ne pas utiliser si on s'attends à ce qu'il y ait plus d'un million d'appels récursifs pour éviter les StackOverflow.
Exemple avec fibonnacci :

```
int fib(int n) {
  int a;
  int b;
  int c;

  a = 1;
  b = 0;

  while (n--) {
    c = b;
    b = a;
    a += c;
  }
  return b;
}
```

Qui devient :

```
int _fib(int a, int b, int n) {
  if (!n)
    return b;
  return _fib(a + b, a, n-1);
}

int fib(int n) {
  return _fib(1, 0, n);
}
```

BEAUCOUP PLUS COURT !!!

### 2.1.5 Fonctions prenant des pointeurs

Si notre fonction est trop longue, on peut se dire "il faut que je la découpe en plus petit morceau", mais que faire quand le code dans une boucle est trop long ? Il suffit de faire une autre fonction qui fera la même chose mais qui prendras nos variables en temps que pointeur pour les modifiers.

### 2.1.6 Multiple expressions en une seule ligne

(cette technique ne viens pas de moi)

Chose assez peu utiliser normalement en C, mais on peut mettre plusieurs expressions (opération renvoyant une valeur) en une seule ligne.
Pour ce faire, la syntaxe est

```
(expr1, expr2, ..., exprn)
```

Et c'est une expression, ayant pour valeur celle de "exprn".
Cela peut paraître inutile jusqu'à qu'on se rappelle que les fonctions/expressions peuvents avoir des effets de bords.
Exemple :

```
int main(void) {
  // trucs
  ...

  close(fd1);
  close(fd2);
  close(fd3);
  return 0;
}
```

peut s'écrire

```
int main(void) {
  // trucs
  ...

  return (close(fd1), close(fd2), close(fd3), 0);
}
```

Néanmoins on n'a pas le droit d'avoir pour expression une assignation pour la norminette.
Enfin, normalement ...
Il suffit de prendre la variable en pointeur et la valeur dans une fonction.
ainsi

```
int main(void) {
  // trucs
  ...

  a = 3;
  b = 54;
  c = 13;
  return f(a, b, c);
}
```

peut devenir

```
int a_int(int *x, int val) {
  *x = val;
  return val;
}

int main(void) {
  // trucs
  ...

  return (a_int(&a, 3), a_int(&b, 54), a_int(&c, 13), f(a, b, c));
}
```

## 2.2 Problèmes de mémoires

Une autre contrainte qui est "difficile" lorsqu'on commence à avoir à gerer la mémoire manuellement est de ne pas oublier de free après l'avoir utiliser.
Pour débugger je conseil d'avoir pour flags : "-O2 -pipe -g -fsanitize=address,leak,undefined -Wall -Wextra -Werror -pedantic -std=c99"
Explication des flags :

1. -Wall -Wextra -Werror : pour avoir des erreurs de compilations en cas de choses "douteuses".
2. -pedantic -std=c99 : la même, mais pour avoir un code C "standard"
3. -O2 : pour que le code créé soit plus rapide
4. -pipe : pour que la compilation soit plus rapide
5. -g : ajoute des informations de débuggage pour par exemple gdb
6. -fsanitize=address,leak,undefined : transformes les leaks, accès d'addresses non alloués, et les actions non définis en erreur en runtime et indique où cela s'est passé

Et sinon utiliser valgrind.

### 2.2.1 Ne pas allouer

Ceci est la meilleur manière de pas avoir de leaks, ça paraît évident, mais il y a beaucoup de situations où une allocation paraît être la seule solution mais auquel ce n'est pas le cas.
Si notre besoin est de taille fixe (et que la mémoire ne sort pas du scope) on peut plutôt faire un tableau sur le stack.
On peut aussi creer des strctures de données spécialiser pour faire différentes sans allocation comme concatener deux chaînes de caractères (une structure prenant deux chaînes), prendre une sous chaine (structure ayant une chaine et une fin), une liste de tout les mots d'une chaine pour faire une opération (creer un itérateur à état à la place) etc.

### 2.2.2 Lifetime et ownership

Ceci n'est pas une méthode, mais une manière de formaliser le programme permettant de plus facilement penser à où free.
Ceci est surtout utiliser en "rust", mais les programmeurs en C sont aussi obligés d'y penser, même si c'est de manière moins explicite.
On va dire que la durée de vie (lifetime) d'une valeur est de quand à quand elle est accéssible avec une valeur "correcte".
Le but est de free à la toute fin des lifetimes des valeurs (si elles ont été alloués)
Pour ce, on dit qu'une valeur à pour proriétaire (son ownership) une variable.
Pour une fonction qui va prendre cette variable en paramètre, il y a deux cas à distinguer

1. L'ownership est transmis : c'est maintenant à cette fonction que reviens le devoir de free la valeur, et l'endroit l'ayant appeler ne doit pas utiliser cette valeur après.
2. L'ownership est emprunter : on peut utiliser la valeur comme on veut mais il ne faut pas le free dans cett fonction

Lorsque vous creez une fonction, pensez bien à dans quel catégories vos arguments (si alloués) sont, si possible le mettre en commentaire.
Une autre opération pouvant influencer sur l'ownership est l'assignation.
Or, cette fois, il est pratique de dire que l'ownership est toujours transmis.
Et la dernière est le renvoie, qui est évidemment transmis vers l'endroit l'ayant appelé.
Si vous avez ça en tête, une valeur devrait avoir qu'un seul owner, et savoir quand la free devient évident (juste à la fin de la lifetime de la dernière variable l'ayant).

### 2.2.3 Arena Allocators

Cette technique vous prendra certainement trop de temps à coder pour la piscine, mais l'idée peut être utile pour d'autre choses ou pour après la piscine.
Un allocateur est quelquechose qui permet d'allouer de la mémoire.
Creer ses propres allocateur est surtout utiliser en "zig" mais est tout à fait faisable en C aussi.
L'idée est la même qu'avoir une gros buffer dans le stack et tout mettre dessus, néanmoins, cette fois ce n'est pas dans le stack.
Donc on alloue une grosse zone mémoire au départ, on l'utilise dans le programme, et on free cette zone à la fin, de tel manière qu'on n'a pas besoin de free au milieu du programme.
Un problème est qu'on ne peut pas free avant la toute fin.
Un exemple d'implémentation : (la piscine est là pour vous apprendre à coder donc ne copier pas tout sans comprendre, le débuggage sera bien pire)

```c
struct arena {
  char *data; // où les allocations iront
  int size; // la taille des données
  int pos; // où on en est dans les allocations
  struct arena *next; //  si il n'y a plus de place dans data on écrit dans un prochain arena
};

struct arena *init_arena(int size) {
  struct arena *a;

  a = malloc(sizeof(struct arena)); // checkez vous si le malloc marche
  a->data = malloc(size);
  a->size = 0;
  a->pos = 0;
  a->next = 0;
  return a;
}
```

```c

int free_arena(struct arena *a) {
  int r;

  r = 0;
  if (a->next)
    r |= free_arena(a->next);

  r |= free(a->data);
  r |= free(a);

  return r;
}

void *alloc_arena(struct arena *a, int size) {
  void *rv;

  while (1) {
    if (a->pos+size <= a->size) {
      rv = a->data + pos;
      a->pos += size;
      return rv;
    }

    if (!a->next)
      a->next = init_arena(size > a->size ? size : a->size);

    a = a->next;
  }
}

void *calloc_arena(struct arena *a, int size) {
  void *rv;

  rv = alloc_arena(a, size);
  ft_memset(rv, 0, size); // on peut aussi itérer sur rv mais il y a plus efficasse donc je vais supposer que votre memset l'est plus
  return rv;
}

// avoir un reset est utile si on sait qu'on n'as plus besoin des données d'avant mais qu'on n'as pas envie de réalloué
void reset_arena(struct arena *a) {
  while (a) {
    a->pos = 0;
    a = a->next;
  }
}
```

Et ainsi, maintenat pour éviter tout leaks de mémoire vous pouvez faire :

```c
int main(void) {
  struct arena *a = init_arena(10000);

  fonction_allouante_difficile_a_tracer(a);
  fonction_allouante_difficile_a_tracer2(a);
  fonction_allouante_difficile_a_tracer3(a);
```

```
7
8   return free_arena(a);
9 }
```

Attention par contre, on ne peut plus check pour l'addressage out of bound en général (à moins de creer des fat pointer où choses de ce genre).
Vous pouvez réutiliser cette technique pour gerer les ouvertures (et fermeture) de fichier de la même manière.

# 3 Structures de données

## 3.1 Fat pointer

Imaginons que vous avez un pointeur, comment savoir où sa zone termine ?
Il y a deux moyens pour ça, avoir une valeur impossible qu'on met à la fin (l'approche des chaînes de caractères) (appeler valeur sentinel) ou sinon avoir une valeur de taille.
Lorsqu'on met un pointeur et sa taille ensemble, on dit que c'est un "fat pointer".
Il y a plusieurs avantages

1. permet de connaître la taille en O(1)
2. permet le check de l'out of bound en O(1)
3. peut être utiliser même sans connaître de valeurs de fin
4. permet de prendre des sous chaînes arbitraire (où ça commence et où ça termine) sans avoir à alloué (alors que l'approche sentinel ne peut seulement controler la position de départ sans allouer)

Exemple pour faire ces choses :

```
1  struct fat_char {
2    char *data;
3    int size;
4  }
5
6  // taille
7  int len_fc(struct fat_char fc) {
8    return fc.size;
9  }
10
11 // checking bound et indexage
12 char index_fc(struct fat_char fc, int index) {
13   if (index > len_fc(fc)) {
14     // il y a eu une erreur et vous pouvez la gere comme vous
       voulez
15   }
16   return fc.data[index];
17 }
18
19 // sans avoir de valeur de fin
20 struct fat_int {
21   int *data;
22   int size;
```

```
}

// sous chaîne arbitraire
struct fat_char sous(struct fat_char fc, int deb, int fin) {
  fat_char fc_new;

  fc_new.data = fc.data + deb;
  fc_size = fin - deb;
  return fc_new;
}
```

## 3.2 Linked list

Cette structure est faite pour pouvoir rajouter des éléments à une liste, ce que l'on pouvais pas jusqu'à présent.
L'idée est que chaque élément sait quel va être le suivant (va avoir un pointeur vers le suivant).
Mon implémentation de l'allocateur arena est un exemple de tel.
C'est plus facile pour faire des opérations récursives, mais en général moins efficasse (le temps d'acceder aux éléments, deferencer les pointeurs etc.)
En soit surtout utile lorsqu'on a pas besoin de stocker beaucoups d'élélements.
Cela peut aussi être utile pour partager une chaîne de fin entre plusieurs listes, mais cela complique beaucoup la gestion de mémoire à tel point qu'il est souvent préférable de juste réalloué cette fin à chaque fois.
niveau template ça ressemble à ça :

```
struct link_T {
  T val;
  struct link_T *next;
};

void add(struct link_T *l, T el) {
  struct link_T *node;

  while (l->next)
    l = l->next;

  node = malloc(sizeof(struct link_T));
  node->val = el;
  node->next = 0;
  l->next = node;
}

T get(struct link_T *l, int n) {
  while (n--)
    l = l->next;
  return l->val;
}

int free_l(struct link_T *l) {
  if (!l)
    return;
  free_l(l->next);
  free(l);
```

```
}
```

où “T” est un type, souvent “int” ou “char”

## 3.3 Vector

Voici la structure de donné la plus utile (et la plus utilisé) pour avoir un ensemble de données qui va potentiellement grandir.
L'idée est d'utiliser un fat pointer avec un peu trop de mémoire en temps que buffer, et agrandir d'un multiple de la taille lorsqu'on a plus de place pour rendre l'ajout d'éléments O(1) en amortie.
Template d'implémentation :

```
struct vector_T {
  T *data;
  unsigned int len;
  unsigned int size;
}

void grow(struct vector_T *t) {
  unsigned int new_size;

  new_size = t->size == 0 ? 1 : 2*t->size;
  T *new_data = malloc(new_size * sizeof(T));
  if (t->data) {
    ft_memcpy(new_data, t->data, t->len * sizeof(T));
    free(t->data);
  }
  t->data = new_data;
  t->size = new_size;
}

void add(struct vector_T *t, T el) {
  if (t->len == t->size)
    grow(t);
  t->data[(t->len)++] = el;
}

T get(struct vector_T *t, int n) {
  return t->data[n];
}

T pop(struct vector_T *t) {
  return t->data[--(t->len)];
}

T top(struct vector_T *t) {
  return t->data[(t->len) - 1];
}

int free_v(struct vector_T *l) {
  if (t->data)
    free(t->data);
  t->size = 0;
  t->len = 0;
}
```

où “T” est un type, souvent “int” ou “char”
Le ”free” dans grow peut faire penser que cette structure sera inutilisable en conjonction avec un arena, mais c’est pas vraiment vrai, on peut ne pas free, juste cette structure prendra 4 fois plus de place que nécéssaire dans le pire des cas (et 2 fois en cas général).

# 4 Algorithmes types

Chacun de ces algorithmes ont pour but de donner une idée, ils vont seront peut-être pas utile mais ces techniques peuvent permetre de rendre son code plus rapide (ce qui est pas forcément utile mais c’est cool)

## 4.1 Puissance

Pour calculer $x^n$ où $n \in \mathbb{Z}$, l’approche la plus ”directe” est de faire

```
T pow(T x, int n) {
  if (n == 0)
    return one_of_type_T;

  if (n < 0)
    return pow(inverse(x), -n);

  return x * pow(x, n-1);
}
```

où “T” est un type représentant des nombres et “inverse(x)” retourne l’inverse de “x” dans le type “T”.
Si “T” est un type d’entier et que $x \geq 2$, on peut directement retourner 0.
Cette approche est en $O(n)$ temp et pareil en espace (les appelles fonctions utilisent la stacks)

En utilisant plutôt la relation (si $n \geq 0$) $x^n = \begin{cases} (x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si } n \equiv 0 \mod 2 \\ x(x^2)^{\lfloor \frac{n}{2} \rfloor} & \text{si } n \equiv 1 \mod 2 \end{cases}$

On peut faire mieux, c’est à dire :

```
T pow(T x, int n) {
  if (n == 0)
    return one_of_type_T;

  if (n < 0)
    return pow(inverse(x), -n);

  if (n & 1)
    return x * pow(x*x, n >> 1);
  return pow(x*x, n >> 1);
}
```

Ce qui est $O(\log(n))$ temps et mémoire.
Et après quelques opération pour le transformer en forme tail recursive et appliquant la tranformation en while devient :

```
T pow(T x, int n) {
  T rv;

  rv = one_of_type_T;

  if (n < 0) {
    x = inverse(x);
    n = -n;
  }

  while (n) {
    if (n & 1)
      rv *= x;

    x *= x;
    n >>= 1;
   }

  return rv;
}
```

Ce qui est en $O(1)$ mémoire et $O(\log n)$ temps.
Cette technique peut aussi être utile pour trouver des nombres de fibonnacci car $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$, donnant un algorithme en $O(\log n)$ pour trouver le $n$-ième nombre de fibonnacci.

## 4.2 memset

Oui, *il y a des optimisations qu'on peut faire avec memset*
Familiarisons nous avec l'implémentation classique :

```
void memset(char *dest, unsigned char byte, unsigned int n) {
  int i;

  i = -1;
  while (++i < n)
    dest[i] = byte;
}
```

Le truc, c'est qu'un "char" c'est petit, et si on pouvait faire toutes ses opérations sur un "unsigned long long", ça serait en théorie 8 fois plus rapide.
Un problème avec ça est que il se peut que $n$ ne soit pas divisible par 8, ainsi, au départ, on va "aligner" les opérations à faire.

```
void ft_memset(char *dest, int b, unsigned long n) {
  unsigned short *sd;
  unsigned int *id;
  unsigned long long *lld;
  const unsigned short s = (((unsigned short) (b & 0xff)) << 8) | (
    b & 0xff);
  const unsigned int i = (((unsigned int)s) << 16) | s;
  const unsigned long long ll = (((unsigned long long)i) << 32) | i
    ;

  if (n & (1 << 0))
```

```
    *(dest++) = (char) (b & 0xff);
  sd = (unsigned short *)dest;
  if (n & (1 << 1))
    *(sd++) = s;
  id = (unsigned int *)sd;
  if (n & (1 << 2))
    *(id++) = i;
  lld = (unsigned long long *)id;
  while (n >= (1 << 3)) {
    n -= 1 << 3;
    *(lld++) = ll;
  }
}
```

Ceci n'est pas "à la norme" car il y a trop de variables créé, mais vous pouvez réutiliser des technique de création de variable plus haut pour en introduire moins.

## 4.3 Résolution d'un sudoku

Je ne vais pas parler des vrais méthodes de comment le faire de manière "logique". La technique s'appelle "backtracking", en gros, on avance autant qu'on peut, et si on a une contradiction on change notre dernière étape.
On va supposer qu'on a déjà une fonction qui dit si on peut poser un nombre sans avoir de contradction immédiate (elle est un peu chiante à écrire) et pour display les plateaux de sudoku.

```
void _solve(int board[81], int i) {
  int w;

  if (i >= 81)
    return;
  if (!empty(board, i))
    return _solve(board, i+1);
  w = -1;
  while (++w <= 9) {
    board[i] = w;
    _solve(board, i+1);
  }
  board[i] = empty_state;
}

void solve(int board[81]) {
  _solve(board, 0);
}
```

## 4.4 Evaluation d'une position au morpion

Ici je vais montrer un exemple du minimax, enfin, une version plus spécifique (mais plus souvent utile), le négamax.
L'idée est que ton meilleur coup est celui qui rends le meilleur de votre adversaire le moins bon.
Et on fait ça récursivement jusqu'à la fin du jeu.

Je vais supposer que la position est dans un "int board[9]", ayant -1 pour le X, 1 pour le O et 0 pour les cases vides, et qu'on ne demande pas si la position a été gagné il y a plusieurs tours.

```
int has_draw(int board[9]) {
  int i;

  i = -1;
  while (++i < 9)
    if (board[i])
      return 0;

  return 1;
}

int abs(int a) {
  return (-1 + 2*(a >= 0)) * a; // "a >= 0 ? a : -a" mais les opérations ternaires sont interdites
                                // peut se faire en 4 lignes avec un if mais c'est bien de savoir faire court
}

int has_won(int board[9]) {
  int i;

  i = -1;
  while (++i < 3) {
    if (abs(board[3*i] + board[3*i+1] + board[3*i+2]) == 3 || abs(board[i] + board[i+3] + board[i+6]) == 3)
      return 1;
  }
  if (abs(board[0] + board[4] + board[8]) == 3 || abs(board[2] + board[4] + board[6]) == 3)
    return 1;
  return 0;
}

int max(int a, int b) {
  return (a > b) * a + (a <= b) * b; // "a > b ? a : b" mais les opérations ternaires sont interdites
                                     // peut se faire en 4 lignes avec un if mais c'est bien de savoir faire court
}

int eval_pos(int board[9], int player) { // player est 1 ou -1
  int best;
  int i;

  best = -1;
  if (has_draw(board))
    return 0;
  if (has_won(board))
    return -1; // ça veut dire que au coup de l'autre il y a eu un triplet
  i = -1;
  while (++i < 9) {
    if (board[i]) {
```

```
47          board[i] = player;
48          best = max(best, -eval_pos(board, -player));
49          board[i] = 0; // on nettois derrière soit :)
50        }
51      }
52      return best;
53  }
```

# 5 Ressources externes

— bitwise hacks : `https://graphics.stanford.edu/~seander/bithacks.html`
— man : `https://www.man-linux-magique.net/index.html`
— la norminette : `https://github.com/42School/norminette/`
— what is an allocator anyway (youtube) : `https://www.youtube.com/watch?v=vHWiDx_l4V0`
— musl libc (comme glibc mais écrit de manière plus propre, il y a beaucoup de bonnes idées) : `https://github.com/seL4/musllibc`