Conventions de codage

30 juin 2025

1 Indentation

L'indentation sera de deux caractères. Elle sera composée uniquement d'espaces (les tabulations n'étant pas représentées de la même façon selon les éditeurs de texte, leur utilisation n'est pas portable).

2 Casse et identificateurs

Afin d'avoir une casse uniforme, nous utiliserons la même casse que celle de la bibliothèque standard :

— les identificateurs seront écrits en minuscules (`void exit(int status);`) ;

— les noms des macros seront en majuscules [1] :

```
#define STRING_MAX_LENGTH 512
```

Nous utiliserons le caractère `_` pour séparer les mots constituant un identificateur ou le nom d'une macro : `int ma_fonction(int un_parametre)`.

Remarque : notons que dans la norme ISO, les identificateurs de fonctions n'utilisent pas le caractère `_` et les mots sont souvent abrégés, ce qui peut compliquer la lecture (`wcsrtombs`). Par contre le caractère `_` est utilisé dans les noms de macros et les identificateurs de type (`CLOCKS_PER_SEC`, `uint_fast64_t`).

Les noms de variables devront être explicites. Toutefois, nous pourrons nous autoriser des noms courts lorsque les variables auront une portée restreinte (`k`, `i`, `j` pour des entiers ; `x`, `y`, `z` pour des réels).

De même, les noms des fonctions seront explicites. Nous éviterons ainsi d'appeler nos fonctions `f`, `g`, ...

3 Espaces

— Les opérateurs et les opérandes seront séparés par des espaces :

1. Il est à noter que la norme ISO ne se conforme pas complètement à cette règle puisqu'elle spécifie également des macros écrites en minuscules (`va_start`, `or_eq`).

```
a = b + c;
z == creal(z) + cimag(z) * I;
```

Il y aura tout de même des exceptions pour les opérateurs unaires ainsi que `[]`, `()`, `.` et `->` :

```
*ptr = NULL;
--i;
tab[i] = 0;
(2 * i) + 3
char c = (char) i;
p.x = 0;
s->n = 0;
```

— On mettra une espace entre les mots clefs et les parenthèses ainsi qu'entre la parenthèse fermante et l'accolade ouvrante :

```
if (i < 0) {
  return EXIT_FAILURE;
}
while (1) {
  printf("Je␣suis␣un␣Highlander\n");
}
```

— On ne mettra pas d'espace entre les identificateurs de fonction et les parenthèses :

```
int ma_fonction(int i) {
  return i + 1;
}

a = ma_fonction(1);
```

— L'étoile indiquant un type pointeur sera collée à l'identificateur et non pas au type (comme dans la norme ISO) :

```
char *strcat(char *dest, const char *src);
```

mais, lorsque l'étoile est encadrée de noms de types, nous écrirons :

```
void foo(const char * const *string_array);
```

4 Instructions

Nous utiliserons le style de Kernighan et Ritchie pour mettre en forme nos instructions.

— Nous ne mettrons qu'une seule déclaration/instruction par ligne :

```
int a;
int b;
```

```
if (scanf("%d%d", &a, &b) != 2) {
  fprintf(stderr, "Erreur␣de␣lecture\n");
  exit(EXIT_FAILURE);
}
```

— L'instruction `if` s'écrira avec l'accolade ouvrante en fin de ligne et le `else` éventuel après l'accolade fermante :

```
if (a >= 0) {
  printf("a␣est␣positif␣ou␣nul\n");
} else {
  printf("a␣est␣strictement␣négatif\n");
}
```

Nous mettrons toujours les accolades, même s'il n'y a qu'une seule instruction dans le bloc, cela nous évitera d'oublier de les mettre lorsqu'on rajoutera une instruction au bloc.

— Dans le cas de `if` imbriqués, on se permettra l'écriture suivante :

```
if (a == 0) {
  printf("a␣est␣nul\n");
} else if (a > 0) {
  printf("a␣est␣strictement␣positif\n");
} else {
  printf("a␣est␣strictement␣négatif\n");
}
```

— Le `while` et le `for` s'écriront comme suit :

```
int k = 0;
while (k < 10) {
  printf("%d\n", k);
  ++k;
}
for (int k = 0; k < 10; ++k) {
  printf("%d\n", k);
}
```

— Le switch sera présenté comme suit :

```
switch (fork()) {
case -1:
  perror("fork");
  exit(EXIT_FAILURE);
  break;
case 0:
  printf("Je␣suis␣le␣fils\n");
  exit(EXIT_SUCCESS);
  break;
default:
```

```
  printf("Je␣suis␣le␣père\n");
  break;
}
```

Ce peut être une bonne habitude de toujours mettre le `break` dans les `case` même s'ils sont inutiles car ça permet de ne pas les oublier quand ils sont nécessaires (chose que font souvent les étudiants)...

— Dans les définitions de fonctions, l'accolade ouvrante sera placée en fin de ligne et l'accolade fermante en début de ligne :

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

Il est difficile de donner des règles précises pour la césure des lignes trop longues. Dans tous les cas, l'indention devra être incrémentée de quatre espaces.

5 Déclarations

— Les variables sont déclarées au plus près de leur première utilisation :

```
printf("Veuillez␣entrer␣un␣entier\n");
int a;
if (scanf("%d", &a) != 1) {
  fprintf(stderr, "Erreur␣de␣lecture\n");
  exit(EXIT_FAILURE);
}
```

— Nous initialiserons les variables dès leur définition quand c'est possible :

```
int i = 0;
struct point p = { .x = 0, .y = 0 };
```

— Les listes d'initialisation contiendront une espace après `{` et avant `}`, de même nous mettrons une espace après `,` :

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

— Nous éviterons au maximum d'utiliser des variables globales.

6 Et aussi

— En l'absence de contraintes particulières, nous préfèrerons utiliser les `double` en lieu et place des `float`.

— À la forme `int main()` nous préfèrerons `int main(void)` et plus généralement nous mettrons `void` pour toutes les fonctions n'acceptant pas de paramètre.

— Les programmes devront se terminer en retournant `EXIT_SUCCESS` ou `EXIT_FAILURE`.

— On prendra bien soin de toujours vérifier la valeur de retour des fonctions, que ce soit pour `scanf` ou pour les appels système.
— Nous placerons la fonction `main` le plus « haut » possible dans le code :

```
#include <stdlib.h>
#include <stdio.h>

void foo(void);
void bar(void);

int main(void) {
  foo();
  bar();

  return EXIT_SUCCESS;
}

void foo(void) {
  printf("foo\n");
}

void bar(void) {
  printf("bar\n");
}
```