Programmation en C
Le Nommage

Régis Barbanchon

L1 Info-Math, Semestre 2

Conventions typographiques pour les identificateurs

Deux grandes conventions : le snake-case et le camel-case.

Le snake-case (litt. “la casse en serpent”) :

- la casse est constante (tout minuscule ou tout majuscule).
- les mots sont séparés par un underscore (tiret bas).

Deux variantes de snake-case :

```
lower_snake_case // all characters are lowercase
UPPER_SNAKE_CASE // all characters are uppercase
```

Le camel-case (litt. la “casse en bosse de chameau”)

- les mots qui en suivent un autre sont capitalisés.
- les mots sont accolés sans caractère séparateur.

Deux variantes de camel-case :

```
lowerCamelCase // first word is not capitalized
UpperCamelCase // first word is capitalized
```

Usage des conventions typographiques (upper-snake-case)

On utilise le **upper-snake-case** pour :

- les noms des macros :

```
#define ARRAY_CAPACITY 10

int main (void) {
  char array [ARRAY_CAPACITY];
  ...
}
```

- les noms des constantes énumératives :

```
enum {
  DIR_NORTH, DIR_EAST, DIR_SOUTH, DIR_WEST
};

int main (void) {
  int direction= DIR_NORTH;
  ...
}
```

Usage des conventions typographiques (lower-snake-case)

On utilise le **lower-snake-case** pour :

- les noms des variables, tableaux inclus :

```
#include <string.h>
int main (void) {
  char my_name [256];
  strcpy (my_name, "Regis");
  int my_name_length= strlen (first_name);
  ...
}
```

- les noms des champs dans les structures :

```
#include <string.h>
int main (void) {
  struct {
    char first_name [256];
    char last_name [256];
  } teacher;
  strcpy (teacher.first_name, "Regis");
  strcpy (teacher.last_name, "Barbanchon");
  ...
}
```

Usage des conventions typographiques (upper-camel-case)

On utilise le **upper-camel-case** pour :

- les noms des types utilisateur :

```
typedef enum {
  DIR_NORTH, DIR_EAST, DIR_SOUTH, DIR_WEST
} CompassDirection;

typedef struct {
  char first_name [256], last_name [256];
} PersonIdentity;

int main (void) {
  CompassDirection dir= DIR_NORTH;
  PersonIdentity teacher;
  strcpy (teacher.first_name, "Regis");
  ...
}
```

- les noms des fonctions utilisateur :

```
int CountIntegersBetween  (int bound1, int bound2) {
  return 1 + abs (bound2 - bound1);
}
```

Grandes catégories de fonctions

Il y a 3 grandes catégories de fonctions :

1. **la fonction de commande (ou procédure) :**
   Son but est de modifier l'environnement (effet de bord).
   ex : *remplir un tableau avec une valeur donnée.*
   ex : *afficher un tableau sur un flux de sortie.*

2. **la fonction de requête Booléenne (ou prédicat)** :
   Son but est de répondre à une question fermée (oui/non).
   ex : *cet entier est-il pair ?*
   ex : *ce tableau contient-il cette valeur ?*

3. **la fonction de requête (non-Booléenne) :**
   Son but est de répondre à une question ouverte.
   ex : *combien d'entiers pairs y-a-t-il dans ce tableau ?*
   ex : *quelle est la somme des entiers de ce tableau ?*

Nommage des procédures (version impérative)

La procédure a pour vocation de former une instruction complète.
Son nom doit donc former une phrase de commande :

- Il contient toujours un **verbe d'action**.
- Ce verbe est à l'**infinitif de valeur impérative**.
- Ce verbe est muni de **son complément** (d'objet ou de lieu).

```
void FillIntArrayWith (int array[], int length, int value) {
  for (int k= 0; k < length; k++)
    array[k]= value;
}
```

```
void PrintIntArray (int const array[], int length, FILE * out) {
  fprintf (out, "[ ");
  for (int k= 0; k < length; k++)
    fprintf (out, "%d ", array[k]);
  fprintf (out, "]\n");
}
```

```
int main (void) {
# define CAPACITY 10
  int my_array [CAPACITY];
  FillIntArrayWith (my_array, CAPACITY, 666);
  PrintIntArray (my_array, CAPACITY, stdout);
  ...
}
```

Nommage des prédicats (version impérative)

Le prédicat a pour vocation principale de former l'expression des conditionnelles et boucles (*si/tant que... ce nombre est pair...*).
Son nom doit donc s'intégrer à une proposition conditionnelle :

- Il contient toujours un **verbe d'état** ou assimilé.
- Ce verbe est au **présent de l'indicatif** à la 3eme personne.
- Ce verbe est au moins **muni de son sujet**.
- La forme est **affirmative** (utiliser ! pour former la négative).

```
bool IntIsMuliple0f (int number, int factor) {
  return number % factor == 0;
}
```

```
bool IntArrayContains (int const array[], int length, int value) {
  for (int k= 0; k < length; k++)
    if (array[k] == value) return true;
  return false;
}
```

Remarque : bons et mauvais usages des prédicats

Pas de mise en équation de la logique avec `true/false` :

```
if (IntIsMultipleOf (number, factor) ==  true) { ... } // Bad!
if (IntIsMultipleOf (number, factor) != false) { ... } // Bad!
```

```
if (IntIsMultipleOf (number, factor)) { ... } // Good!
```

De même avec les négatives :

```
while (IntArrayContains (array, value) !=  true) { ... }  // Bad!
while (IntArrayContains (array, value) == false) { ... }  // Bad!
```

```
while ( ! IntArrayContains (array, value)) { ... } // Good!
```

Pas d'étude de cas fictive pour les équivalences de prédicats :

```
bool IntIsEven (int number) {
  if (IntIsMultipleOf (number, 2)) return true;  // Bad! fictive case.
  else                             return false;
}

bool IntIsOdd (int number) {
  if (IntIsMultipleOf (number, 2)) return false;  // Bad! fictive case.
  else                             return true;
}
```

```
bool IntIsEven (int number) { return   IntIsMultipleOf (number, 2); } // Good!
bool IntIsOdd  (int number) { return ! IntIsMultipleOf (number, 2); } // Good!
```

Nommage des requêtes (version impérative)

La requête a pour vocation de former une expression à l'intérieur d'une instruction. Son nom est donc **souvent un groupe nominal** correspondant à la réponse à sa question.

```
int RandomIntBetween (int low_bound, int high_bound) {
  assert (low_bound < high_bound);
  int value_count= high_bound - low_bound + 1;
  return low_bound + rand() % value_count;
}
```

```
int LeftmostIndexOfCharInString (char const string[], char character) {
  for (int k= 0; string[k] != '\0'; k++)
    if (string[k] == character) return k;
  return -1;
}
```

Mais **on peut aussi** nommer une requête comme une procédure, et **expliciter le verbe** qui ordonne de répondre à la question :

```
int DrawRandomIntBetween (int low_bound, int high_bound) {...}
int FindLeftmostIndexOfCharInString (char const string[], char character) {...}
```

Nommage des méthodes (fonctions "Orientées Objet")

Lorsqu'on pense "orienté objet", toute fonction est sous la responsabilité d'un type d'objet. Celui-ci (= **la classe**) est transféré **devant le nom de** la fonction (= **la méthode**), **séparé d'elle par un underscore**. La classe est souvent :

- le complément d'objet pour les procédures...

```
void IntArray_FillWith (int self[], int length, int value);
void IntArray_Print (int const self[], int length, FILE * out);
```

- le sujet pour le prédicats...

```
bool IntArray_Contains (int const self[], int value);
bool Int_IsMultipleOf (int self, int factor);
bool Int_IsEven (int self);
bool Int_IsOdd (int self);
```

- le complément d'objet ou de lieu pour les requêtes...

```
int String_LeftmostIndexOfChar (char const self[], char character);
int Int_RandomBetween (int low_bound, int high); // no self
```

Si la classe est en paramètre, elle est toujours en 1er (`self`).