Bonnes pratiques de Programmation

Introduction

Les bonnes pratiques (ou règles de codage) sont un ensemble de règles visant à uniformiser les pratiques de développement afin que "tout le monde s'entende bien". Le but est que le code soit **facile à lire** pour soi-même et pour les autres, que sa **logique soi évidente**, qu'il soit **explicite** et **soigné**. Il doit également être maintenable dans le temps bien que cela ne nous concerne pas encore, de même pour la portabilité.

> Remarque : Les "bonnes pratiques" restent tout de même subjectives puisque la lisibilité est subjective. Presque tout ce qui se trouve dans ce document provient des sources citées, mais faire autrement n'est pas toujours un problème !

Il existe de nombreuses règles réparties en plusieurs thèmes :

1. L'organisation des fichiers
2. L'indentation
3. Les conventions de nommage
4. Les commentaires et la documentation
5. Les recommandations sur les déclérations
6. Les recommandations sur l'écriture des instructions, des structures et l'usage des parenthèses (dont accolades etc...).

Par exemple, du code redondant, 200 caractères sur une même ligne ou bien des noms de variables à une lettre constituent des crimes (parmi d'autres), que vous ne commettez évidemment plus.

> Remarque : Certaines de ces règles dépenderont également du langage, documenter son code est assez universel mais les convention de nommage sont souvent différentes, de même pour l'organisation des fichiers. Nous ferons donc (si nécessaire) la distinction entre OCaml et C.

Liens utiles

- Pour le C : GNU Coding Standards
- Pour le OCaml : OCaml Programming Guidelines
- Pour le Python : PEP 8

Vous pourriez tenter d'ingérer ces textes sacrés de la programmation mais je vous propose un résumé sur ce qui est le plus important pour nous.

I - Le nommage

Que ce soient les noms des variables ou ceux des fichiers il est important d'avoir une organisation claire pour que même sans éditeur de texte vous puissiez identifier chaque élément du programme. Ces noms servent en quelques sortes de premiers "commentaires" pour votre code, ils doivent donner des **informations utiles**. Cependant il est recommandé de ne pas le faire pour les caractères qui prêtent à confusion comme `l et O` qui se confondent facilement avec `1 et 0`.

> Remarque : Pour ce qui est temporaire ou “local” on peut se permettre des noms assez courts puisque c’est utilisé dans un seul contexte où les commentaires suffisent, il est important que le nom donne des informations si c’est pertinent.

Dans nos deux langages et dans bien d’autres **snake_case** pour nommer les variables. Il est explicitement interdit d’utiliser des majuscules ! En C elles sont réservées aux macros, aux énums et aux constantes. En OCaml elles sont réservées aux constructeurs et aux modules.

Pour les fonctions et les fichiers on utilisera ici cette même convention.

> Remarque : Historiquement on demandait un maximum de 14 caractères en C afin d’éviter des conflits sur les vieux systèmes, mais aujourd’hui ce n’est plus le cas (dans le cas général).

C - Particularités

Les constantes sont nommées en **screaming snake case**, de plus elles sont généralement précédées du mot clé `const` .

```c
const int MA_CONSTANTE = 42;
```

Je vous renvoie sur cette page où vous pourrez prendre connaissance de toutes les subtilitées liées à la délcaration de variables et de constantes.

OCaml - Particularités

En OCaml il est recommandé de nommer explicitement vos **exceptions**, jusque là rien de nouveau.

Mais il est également recommandé d’utiliser le plus possible les exceptions données par défaut.

Si vous avez besoin d’exprimer le fait de ne pas avoir trouver un élément, plutôt que de redéfinir une nouvelle exception il est préférable d’utiliser `Not_found`.

> Remarque : On utilise du **snake case** en commançant par une majuscule pour les exceptions.

Enfin, lorsqu’une fonction anonyme (`fun x -> foo...`) est trop grande nommez-là.

II - Formattage

Le formattage du code est la structure visuelle que vous lui donnez, c’est l’équivalent de la présentation d’une copie, si elle est sale personne ne voudra (ou ne pourra) la lire.

Indentation (très important) L’indentation permet d’identifier très rapidement les blocs de code, il est indispensable de la respecter.

```c
// Mal indenté
for (i = 0; i < 42; i++) {
    foo();
if (i == 7) {
bar();
}
}

// Au lieu
for (i = 0; i < 42; i++) {
    foo();
    if (i == 7) {
        bar();
```

```
    }
}
```

```
(* Mal indenté *)
for i = 0 to 42 do
foo ();
if i = 7 then
bar ();
done;

(* Au lieu de *)
for i = 0 to 42 do
    foo ();
    if i = 7
        then bar ();
done;
```

En OCaml on placera les `|` du pattern-matching sous le `let`, on ne fait pas d'alignement excessif. De même, on met les flèches directement après.

```
(* On n'écrit pas *)
let f = function
          | C1          -> 1
          | Long_name _ -> 2
          | _           -> 3

(* Mais *)
let f = function
  | C1 -> 1
  | Long_name _ -> 2
  | _ -> 3..
```

Utilisation des opérateurs Il est plus lisible de séparer ses opérateurs par des espaces. Vous pouvez également vous servir de parenthèses.

```
// Peu lisible ou difficilement
x*y
x * y+z
x*y/z+a-b


// Au lieu de
x * y
x * y + z
(x * y / z) + (a - b)
```

> On remarque bien ici la subjectivité, les parenthèses sont inutiles et nous aurions raison de ne pas les mettre, mais ce n'est pas interdit. Quand on peut contribuer à une meilleure lisibilité on a tout à fait le droit.

Placement des accolades & déclaration d'une fonction En C, lors de la déclaration d'une fonction, on place les accolades sur la première colonne, c'est-à-dire qu'on revient à la ligne. C'est surtout utile pour certains outils de recherche. Dans une fonction il ne faudra pas le faire pour autre chose (`while`, `for`, `if`…) car cela perturberait ces outils. Cependant cela reste autorisé pour les `struct` et les `enums` (qui peuvent rester en une ligne s'ils tiennent).

De plus, le nom de la fonction suit la même règle, il est placé à partir de la première colonne et est donc séparé du type de retour.

```
// Un exemple
typedef struct position {int x; int y;};
typedef struct joueur
{
    int id;
    char* nom;
    int* items;
};

void
foo (joueur next_player, position pos)
{
    bar();
};
```

On placera également des accolades lorsque l'on imbriquera des `if` & `else`.

```
if (i == 42) {
    foo();
} else {
    bar();
}
```

Lorsque qu'il y en a plusieurs d'imbriqués ont hésitera pas à rassembler `else` et le `if` qui suit ensemble.

```
if (i == 42) {
    foo();
} else if (i % 2 == 0) {
    bar();
}
```

Taille du code Il existe une limite de caractères recommandée en largeur, c'est de là que vient cette ligne verticale dans vos environnements. En C un maximum de 79 caractères est recommandé pour 80 en OCaml. L'objectif est que le code soit visible sur n'importe quel environnement. Il est également recommandé de faire tenir vos fonctions sur un écran si possible, plus si vous n'avez vraiment pas le choix.

Dans certains cas, respecter la limite horizontale peut sembler impossible mais voici quelques outils qui vous seront utiles.

- Sauter des lignes entre vos arguments et vos opérateurs (avant) s'ils ne tiennent pas sur une ligne.

  ```
  void
  foo (int longueur_matrice, int largeur_matrice, int** matrice,
      int element, int* ligne_retour, int* colonne_retour)
  {
      if (longueur_matrice > largeur_matrice
          && element > 0
          && element < largeur_matrice) foo();
  }
  ```

- En OCaml, mettre un `\` dans une chaîne de caractère permet d'omettre les espaces en début de ligne après un retour.

```
let universal_declaration =
  "-1- Programs are born and remain free and equal under the law;\n\
   distinctions can only be based on the common good." in
  ....
```

III - Mauvaises Pratiques

Tout comme on peut trouver des atrocités en langue, on peut trouver des crimes de guerre en programmation. On cite dans cette partie les plus courants.

Les if & else inutiles Lorsque vous manipulez des valeurs booléennes avec des opérateurs, il ne faut pas oublier que vous manipulez des valeurs booléennes avec des opérateurs… A ce titre, vous savez que vous pouvez utiliser cette valeur sans passer par des conditions.

En effet, tout comme `1 + 4` renvoie `5`, `3 == 5` renverra généralement `false` tandis que `21 > 12 && 42 > 7` renverra `true`.

Exemple plus concret :

```
// Mauvaise pratique
bool
f (int x, int y)
{
    // "si j'aurais su j'aurais pas venu"
    if (x > y || x % 2 == 0) {
        return false;
    } else {
        return true;
    }
}

// Bonne pratique
bool
g (int x, int y)
{
    return !(x > y || x % 2 == 0);
}
```

> Remarque : Cette mauvaise pratique est présente partout puisqu'elle ne dépend pas du langage. Elle ne vous fera pas voir.

Programmer dans les fichiers d'en-tête Puisque les extensions ne sont que des conventions pour nous, l'ordinateur accepte que l'on exécute n'importe quel fichier. Vous pourriez prendre votre cours dans un fichier `.png` mais vous ne le faites pas bien sûr. De la même manière vous ne devez pas coder d'instruction dans les fichiers d'en-tête, ils sont réservés aux déclarations (variables, prototypes de fonctions etc…). Cela est même considéré (par les humains) comme une erreur de le faire !
Je vous renvoie au cours sur la compilation pour en savoir plus sur les fichiers d'en-tête.

Affectation dans les conditions Que ce soit par erreur ou par curiosité (en espérant pas par besoin), vous savez peut-être déjà que dans certains langages on peut affecter une variable dans une condition puisque l'affectation a une valeur de retour.

Exemple en C :

```
void
f (int x)
```

```
{
    if (x = 42) {
        printf("%d", x);
    } else {
        printf("%d", 0);
    }
}
```

On décide d'appeler `f` avec l'argument `x = 2024`, à votre avis qu'est-ce qui sera affiché par cette fonction ?

Réponse : `42`

En effet, l'affectation renvoie la valeur affectée, puisqu'elle est différente de `0` (ici `2024`), la condition est remplie puis la valeur de `x` nouvellement affectée est affichée. Ainsi l'affectation dans une condition renverra toujours `true` à moins d'affecter une valeur nulle.

Il s'agit là plus d'une erreur que d'une mauvaise pratique mais dans la situation où cela vous paraîtrait malin de la faire sachez que ce sera incompréhensible.

Ignorer les warnings Les warnings ne sont pas là pour rien, c'est écrit dessus. Préférez les résoudre plutôt que de les bypass. De plus, un programme rempli de warnings dégage un parfum "d'amateur".

Lors de votre pattern-matching en OCaml, faites en sorte que celui-ci soit exhaustif, les développeurs le disent, abusez-en mais faites-le bien. Evitez-donc les "catch-all" (ou "fourre-tout" en français) comme `_ -->` ou `x -->`.

Lorsque vous souhaitez ignorer le résultat d'une expression, précisez-le explicitement, ne laissez pas le warning.

```
(* Ne pas faire *)
List.map f l;
print_newline ()


(* Mais *)
let _ = List.map f l in
print_newline ()

(* Ou bien avec la fonction ignore *)
ignore (List.map f l);
print_newline ()
```

Si le compilateur vous averti, c'est qu'il y a une erreur potentielle, dans cet exemple l'intention du développeur aurait dû être celle-ci :

```
List.iter f l;
print_newline ()
```

Gestion des importations Utiliser les fonctions importés depuis `module` en écrivant `module.fonction` est certes précis mais long. Il est donc possible d'ouvrir les modules mais il faut faire attention aux collisions.

```
let f () =
  Format.print_string "Hello World!"; print_newline ()
```

Ce code est pose problème car il n'appelle pas `Format.print_newline` pour vider la file d'attente et afficher `"Hello World!"`. Au lieu de cela, `"Hello World!"` est coincé dans la file d'attente, tandis que `Stdlib.print_newline` produit un retour chariot sur la sortie standard.

Si `Format` se trouve dans un fichier et que la sortie standard est le terminal, l'utilisateur aura du mal à constater qu'il manque un retour chariot dans le fichier, tandis qu'un faux retour chariot apparaissait à l'écran !

Cependant pour une question de lisibilité il peut être parfois vivement recommandé d'ouvrir le module. De plus, ouvrir de grandes bibliothèques comme celles contenant des entiers de précision arbitraires afin de ne pas surcharger le prgramme qui lest utilise est bien.

```
open Num

let rec fib n =
  if n <= 2 then Int 1 else fib (n - 1) +/ fib (n - 2)
```

Dans un programme où les définitions de types sont partagées, il est avantageux de rassembler ces définitions dans un ou plusieurs modules sans implémentations (contenant uniquement des types). Il est alors acceptable d'ouvrir systématiquement le module qui exporte les définitions de types partagées.

IV - Commentaires & Documentation

Lorsque vous programmez, vous avez une logique et une réflexion qui ne seront pas les mêmes que celles de quelqu'un d'autre ou même de vous dans quelques semaines. Commenter votre code est donc une nécessité pour un programme maintenable dans le temps et partageable. On recommande même souvent de toujours coder en anglais pour pouvoir demander de l'aide plus facilement.

Chaque programme fois commencer par un commentaire expliquant brièvement à quoi il sert. Au début de chaque fichier, indiquez également son utilité.

Pour chaque fonction, indiquez la spécification de la fonction en détail (ce qu'elle fait, types d'arguments, leur utilité…). Il n'est cependant pas nécessaire de répéter la déclaration de l'argument. Il est attendu de donner des informations pertinentes comme un élément inhabituel ou propre à la fonction.

En C, il est recommandé de laisser deux espaces après une phrase dans un commentaire et de les commencer par une majuscule, cela facilitera la génération automatique de la documentation. Il y a beaucoup de détails à ce propos à aller voir, il n'y a pas besoin de le détailler ici.

Les commentaires dits "inoffensifs", ceux qui apportent des informations triviales sont inutiles, et sont même une nuisance. Voici un exemple de ce que cela pourrait être :

```
(*
  Function print_lambda:
  print a lambda-expression given as argument.

  Arguments: lam, any lambda-expression.
  Returns: nothing.

  Remark: print_lambda can only be used for its side effect.
*)
let rec print_lambda lam =
  match lam with
  | Var s -> printf "%s" s
  | Abs l -> printf "\\ %a" print_lambda l
  | App (l1, l2) ->
     printf "(%a %a)" print_lambda l1 print_lambda l2
```

N'hésitez pas à utiliser des `assert` qui donnent une information claire tout en apportant une réelle vérification.

Sources

- Page wikipédia sur les règles de codage
- Cours d'algo de Pierre-Antoine Champin
- GNU Coding Standards
- OCaml Programming Guidelines
- Cours de bonnes pratiques en C sur developpez.net
- Github de ocamlformat autof-formatter pour ocaml
- wikiversité - constantes et variables