Informatique *Cours*
Bonnes pratiques de la programmation Ipesup

Objectifs

Quelques conseils en matière de programmation, que ce soit dans l'écriture du code source, sa compilation, son exécution ou sa mise au point. Ces conseils sont valables autant pour OCaml que pour C, ou plus largement pour tout langages.

1 Code source

Il existe de nombreuses manières d'écrire un même programme. Si l'on n'évalue qu'un programme comme une boîte noire, on peut négliger la qualité de son code source. Pourtant, un code mal structuré rend sa relecture ultérieure difficile, complique son partage, et même en l'absence de relecture future, rend sa mise au point inutilement pénible. Il est donc essentiel d'écrire un code propre et organisé. Quelques règles simples suffisent à y parvenir avec un effort minimal.

Lisibilité

C et OCaml n'imposent pas une indentation reflétant la structure du code ; ainsi

```
if (x == 0)
printf("x = %d\n", x);
x++;
```

peut laisser croire à tort que `x++` dépend du test.
Il appartient donc au programmeur de *faire en sorte que la structure soit visible dans le code.* Cela passe par une indentation soignée et par l'alignement volontaire d'éléments similaires .

Il faut également *éviter les lignes excessivement longues* : même si les outils les acceptent, un code est plus lisible lorsqu'il est découpé en opérations élémentaires occupant chacune une ligne d'environ 80 caractères, avec leurs résultats stockés dans des variables intermédiaires. Contrairement à une idée répandue chez les débutants, multiplier les variables n'a pas d'impact négatif sur les performances.

De la même manière, on améliore la clarté en *décomposant un programme en fonctions élémentaires*, souvent réutilisables.

Le choix des noms joue aussi un rôle majeur. Pour des variables locales, des noms très courts, voire une seule lettre, sont suffisants, surtout dans des fonctions brèves. Pour un tableau, appeler les arguments `a` et `n` est parfaitement acceptable :

```
void dutch_flag(int a[], int n) {
```

En revanche, les fonctions méritent des noms explicites : `dutch_flag` indique clairement son rôle, particulièrement important si la fonction est destinée à être réutilisée dans d'autres fichiers.

Définir un intervalle

Pour manipuler un intervalle dans un tableau ou une chaîne, il est recommandé d'utiliser un indice gauche *inclus* et un indice droit *exclu*.
Ainsi `void f(int a[], int lo, int hi)` agit sur les cases `a[lo..hi[`.
En général $0 \leq lo \leq hi \leq n$ où `n` est la taille du tableau. Cette convention permet :

- de connaître immédiatement la taille de l'intervalle : `hi - lo` ;
- de le couper naturellement en $lo \dots m$ et $m \dots hi$ avec $lo \leq m \leq hi$;
- de représenter tout le tableau par $lo = 0$ et $hi = n$.

Commentaires

Les commentaires facilitent la compréhension du code, à condition de ne pas paraphraser ce que le programme exprime déjà. Un bon commentaire doit apporter une *information supplémentaire* que le code ne donne pas immédiatement.

Les commentaires les plus importants sont les spécifications des fonctions, en particulier celles qui sont exportées. Elles décrivent les entrées, le résultat et les effets produits. Par exemple :

```
// entrée : tableau a de taille n, valeurs 0,1,2
// sortie : a trié en place dans l'ordre croissant
void dutch_flag(int a[], int n);
```

Le premier commentaire explicite les hypothèses sur les arguments. Le second décrit précisément l'effet attendu.

Au-delà des spécifications, les commentaires peuvent clarifier des points subtils, notamment en explicant les *invariants*.

Invariants

Un **invariant de boucle** est une propriété préservée à chaque itération.
Par exemple `// a[0..i[ est trié`. On peut rendre l'invariant encore plus explicite :

```
int b = 0, i = 0, r = n;
while (i < r) {
//      0          b        i        r        n
//      +---------+--------+--------+--------+
// a    |   0     |    1   |  ???   |   2    |
//      +---------+--------+--------+--------+
```

pour rendre claire que `a[0..b[` contient uniquement 0, `a[b..i[` uniquement 1, etc.
Un second type d'invariant utile est l'**invariant de structure**, qui exprime des relations toujours vraies au sein d'une structure de données. Par exemple :

```
struct ArrStack {
int capacity;
int size; // 0 <= size <= capacity
int *data; // tableau de taille capacity
};
```

Les deux commentaires explicitent les contraintes clés : la taille est bornée par la capacité, et `data` pointe vers un tableau correspondant.

2 Compilation

En C comme en OCaml, l'exécution nécessite une phase de *compilation* : analyse syntaxique et typage statique avant la création de l'exécutable.

Ces étapes peuvent produire des **erreurs**, bloquant la génération du programme, comme des erreurs de syntaxe ou de typage :

```
file.c:48: error: expected '{' at end of input
file.c:28:11: error: too many arguments to function 'f'
```

Le travail alterne donc : édition, compilation, exécution. `éditer` ↔ `compiler` ↔ `exécuter`

Le compilateur peut aussi émettre des **avertissements**. Même si l'exécutable est produit, ils indiquent souvent une erreur probable :

```
warning: comparison between pointer and integer
```

Il faut donc les lire attentivement : chaque erreur détectée à la compilation évite un échec à l'exécution. Les compilateurs offrent de nombreuses options pour gérer les avertissements ; en C, l'usage systématique de `-Wall` est recommandé.

Compiler souvent

Il ne faut jamais attendre la fin de l'écriture du programme pour compiler. Compiler régulièrement simplifie la mise au point et évite de longues séances de débogage. On peut parfaitement compiler du code incomplet : uniquement certaines fonctions, ou même des blocs provisoires . En C, on peut aussi *déclarer* une fonction sans la définir, afin de vérifier le reste du code avant d'y revenir.

Dans la pratique, compiler est souvent aussi simple qu'un raccourci clavier ; certains éditeurs compilent automatiquement à la sauvegarde.

3 Exécution

Théorème de Rice

> *Toute propriété non triviale portant sur la fonction calculée par un programme est indécidable.*

Plus précisément : soit P une propriété qui dépend uniquement du comportement sémantique d'un programme, et qui n'est ni toujours vraie ni toujours fausse. Il n'existe aucun algorithme capable de décider, pour un programme donné, si celui-ci satisfait P.

Ainsi, le compilateur ne peut détecter qu'une faible part des erreurs possibles : il ne peut assurer ni l'absence d'accès hors bornes, ni l'absence de division par zéro ou de déréférencement nul, ni même la correction fonctionnelle du programme. L'exécution peut donc « planter », et il faut alors identifier la cause pour corriger le code.

C

En C, une erreur peut se manifester par `Illegal instruction` (ex. division par zéro) ou `Segmentation fault` (accès mémoire invalide) :

```
$ ./program
Segmentation fault
```

Pour diagnostiquer l'origine, on compile avec `-g` et on utilise `gdb` :

```
gcc -g -o program file.c
----
$ gdb ./program
(gdb) run
Program received signal SIGSEGV
print_list (...) at file.c:20
20 printf("%d\n", l->head);
```

On voit que `l` est très probablement nul. La commande `bt` permet d'afficher la trace d'appels :

```
(gdb) bt
#0 print_list at file.c:20
#1 test at file.c:61
#2 main at file.c:71
```

OCaml

En OCaml, les erreurs d'exécution prennent la forme d'exceptions non rattrapées, par exemple `Invalid_argument` ou `Not_found` :

```
$ ./program
Fatal error: exception Not_found
```

Avec `-g` et `OCAMLRUNPARAM=b`, on obtient la trace :

```
ocamlc -g file.ml -o program
-----
OCAMLRUNPARAM=b ./program
Raised at File.f in file "file.ml", line 7
Called from File.g in file "file.ml", line 9
Called from File in file "file.ml", line 11
```

On identifie alors précisément la fonction fautive, la ligne et la chaîne d'appels.

Débugger avec `printf`

Quand les outils précédents ne suffisent pas (par exemple lorsqu'un programme ne termine pas) il est utile d'instrumenter directement le code. Quelques affichages bien placés permettent souvent d'identifier l'origine du problème.

La fonction `printf`, en C comme en OCaml, est idéale pour afficher l'état de variables :

```
printf("appel avec i=%d et j=%d\n", i, j);
```

Parfois, il suffit simplement de vérifier qu'un point précis du code est atteint.

OCaml

En OCaml, on peut écrire :

```
Format.printf "ICI@.";
```

On utilise `Format` plutôt que `Printf` car la directive `"@."` force immédiatement l'affichage et le retour à la ligne.

Programmation défensive

Pour détecter les erreurs d'exécution avant qu'elles ne se produisent, on utilise l'instruction `assert` en C comme en OCaml. Elle évalue une expression booléenne : si elle vaut `true`, le programme continue ; sinon, l'exécution s'interrompt et l'emplacement exact de l'assertion est signalé.

Exemple

En C : `assert(p != NULL)` empêche qu'un pointeur nul provoque plus loin une erreur de segmentation. Le message indique alors la ligne fautive :

```
program: file.c:33: main: Assertion ‘p != NULL’ failed.
```

Les expressions passées à `assert` peuvent être coûteuses (vérifier qu'un tableau est trié, qu'une matrice est symétrique, qu'un nombre est premier, etc.). Pendant le développement, cela ne pose pas problème, mais une fois le programme stabilisé, on peut désactiver les assertions : option `-DNDEBUG` en C, `-noassert` en OCaml.

Cas particulier d'assert false en OCaml

`assert false` a le type `unit`, mais elle peut prendre n'importe quel type, donc s'utiliser là où une valeur est attendue. Bien qu'elle échoue systématiquement, elle est utile dans deux cas :

- pour marquer un point du code théoriquement inatteignable ;
- pour remplacer temporairement du code non encore écrit.

Par exemple, pour qu'une fonction ne doit jamais être appelée sur une liste vide :

```
let rec random_element = function
| [] -> assert false
| x::l -> ...
```

Cette particularité permet de garder un typage correct.
`assert false` est également compilée de manière spéciale : elle n'est pas supprimée par `-noassert`. Si l'exécution l'atteint, elle échoue immédiatement, ce qui permet d'obtenir une localisation précise sans activer manuellement la trace d'exécution. L'ouvrage l'emploiera dans plusieurs endroits.

4 Validation, test

Une fois le programme exécuté sans erreur, il faut encore vérifier qu'il produit le bon résultat. Parfois, une simple vérification manuelle suffit ; mais le plus souvent, il faut tester la correction du programme sur de nombreuses entrées. Pour un programme interactif, on peut utiliser un script ; pour des fonctions, il est plus simple d'écrire du code de test qui les appelle et compare leurs résultats.

Exemple

Considérons par exemple une fonction OCaml de tri :

```
let awesome_sort (l : int list) : int list = ...
```

On crée un fichier `test_sort.ml` pour la comparer à `List.sort` :

```
let test n m =
let l = List.init n (fun _ -> Random.int m) in
assert (awesome_sort l = List.sort Stdlib.compare l)
```

puis on l'appelle avec diverses tailles et plages de valeurs :

```
let () =
for n = 0 to 10 do test n 1; test n 2; test n 100 done
```

Ce simple test couvre déjà plusieurs cas importants : liste vide, liste d'un élément etc. On peut ensuite tester les performances sur de grands `n`.

Comme on ne regarde pas le code de `awesome_sort`, il s'agit d'un test *en boîte noire*.

Pour aller plus loin, on peut analyser directement le code et concevoir des tests adaptés : c'est le test *en boîte blanche*.

Exemple

Un tri par insertion en C `void insertion_sort(int a[], int n)` contient une boucle interne avec un test critique :

```
while (j > 0 && a[j-1] > v) { ... }
```

Le test `j > 0` empêche l'accès à `a[-1]` : on le voit en triant un tableau en ordre inverse. Le test `a[j-1] > v` garantit que l'insertion se prolonge tant que nécessaire : on le valide en inspectant le résultat final.
Il faut vérifier que le tableau est trié et aussi qu'il contient les mêmes éléments qu'au départ. Contrairement aux listes immuables d'OCaml, le tableau est modifié en place. Il faut donc comparer les histogrammes avant/après :

```
void test_sort(int n, int m) {
int *a = calloc(n, sizeof(int));
...
for (int v = 0; v < m; v++) assert(hold[v] == hnew[v]);
free(a); free(hold); free(hnew);
}
```

Mesurer les performances

Tester un programme implique aussi d'évaluer ses performances, en particulier le temps d'exécution. Depuis le `shell`, on peut utiliser la commande `time` :

```
$ time ./program
real 0m3.907s
```

Ici, l'exécution a consommé 3,9 secondes de CPU. Une bonne pratique consiste à répéter la mesure cinq fois, éliminer les deux extrêmes, puis moyenner les trois restantes.

Si l'on souhaite mesurer uniquement une portion du programme (par exemple après une phase de préparation coûteuse comme l'allocation d'un grand tableau) on peut utiliser le temps CPU cumulé depuis le début.

— En C, on utilise `clock()` :

```
... préparation ...
clock_t start = clock();
... code à mesurer ...
clock_t stop = clock();
printf("%f\n", (double)(stop - start) / CLOCKS_PER_SEC);
```

— En OCaml, on utilise `Sys.time()` :

```
let start = Sys.time ()
... code à mesurer ...
let stop = Sys.time ()
let () = Format.printf "%f." (stop -. start)
```

On peut ensuite répéter la mesure pour différentes tailles d'entrée et tracer la courbe de performance.

Pour mesurer l'usage mémoire :

— En C on dispose de `mallloc_stats`

```
#include <malloc.h>

int main(void) {
    printf("Avant les allocations :\n");
    malloc_stats();
    ... allocations...
    printf("\nAprès les allocations :\n");
    malloc_stats();
}
```

— En Ocaml de `Gc.stat`

```
let () =
  let s1 = Gc.stat () in
  Printf.printf "Avant : heap_words = %.0f\n%!" s1.Gc.heap_words;

  let _a = Array.make 5_000_000 0 in  (* allocation *)

  let s2 = Gc.stat () in
  Printf.printf "Après : heap_words = %.0f\n%!" s2.Gc.heap_words;
```

5 Quelques conseils

Voici quelques recommandations finales, à relire régulièrement au fur et à mesure que l'on progresse en programmation.

Prendre du plaisir

Apprendre à programmer demande du temps, comme tout artisanat : accumuler de l'expérience, maîtriser les outils, s'approprier les concepts. Toute occasion de coder, même brève ou simple, est utile, et il est important d'y prendre plaisir.

Papier et crayon à côté du clavier

Un schéma ou un calcul rapide clarifie souvent une idée plus efficacement qu'une longue réflexion mentale. D'où l'importance d'avoir systématiquement de quoi écrire à portée de main. Une partie de ces notes pourra ensuite être intégrée dans le code, notamment sous forme de commentaires.

L'optimisation prématurée est la source de tous les maux

Il ne faut pas rechercher des micro-optimisations qui rendent le code illisible. Le compilateur sait déjà transformer des conditions ou déplier des fonctions. Mieux vaut coder clairement, en utilisant des opérations cohérentes avec le sens du programme. L'optimisation importante concerne surtout le choix d'algorithmes et de structures adaptés : optimiser marginalement un mauvais algorithme ne compensera jamais son inefficacité globale.

Factoriser, mais raisonnablement

Éviter le copier-coller est essentiel : il multiplie les erreurs. On doit donc exploiter les abstractions du langage (fonctions, modules, objets, etc.) pour ne définir chaque élément qu'une seule fois. Mais il ne faut pas pousser la généralisation à l'excès : un code trop abstrait devient opaque. La programmation exige de trouver un équilibre entre factorisation élégante et lisibilité pratique.