Bonnes pratiques

Lycée Thiers

Crédits

- Un mémo du CNRS pour `Python`
- Cette page D'Emmanuel Delahaye.
- « Informatique MP2I/MPI » Ellipse.

Code bien écrit

Un algorithme, ou code "bien écrit" doit avoir les propriétés suivantes :

- Être facile à lire, pas soi-même mais aussi par les autres.

Code bien écrit

Un algorithme, ou code "bien écrit" doit avoir les propriétés suivantes :

- Être facile à lire, pas soi-même mais aussi par les autres.
- Avoir une organisation logique et évidente.

Code bien écrit

Un algorithme, ou code "bien écrit" doit avoir les propriétés suivantes :

- Être facile à lire, pas soi-même mais aussi par les autres.
- Avoir une organisation logique et évidente.
- Être explicite, montrer clairement les intentions du développeur.

Code bien écrit

Un algorithme, ou code "bien écrit" doit avoir les propriétés suivantes :

- Être facile à lire, pas soi-même mais aussi par les autres.
- Avoir une organisation logique et évidente.
- Être explicite, montrer clairement les intentions du développeur.
- Être soigné et robuste au temps qui passe.

Indentation

- En C et OCaml, l'indentation ne fait pas sens

```
1 if (x==0)
2         printf("x=%d\n",x);
3         x++;
4
```

On peut croire que `x` n'est incrémenté que si il est nul : erreur.

Indentation

- En C et OCaml, l'indentation ne fait pas sens

```
1 if (x==0)
2         printf("x=%d\n",x);
3         x++;
4
```

  On peut croire que `x` n'est incrémenté que si il est nul : erreur.

- C'est au programmeur de faire un effort pour que le code montre la structure.
  On peut choisir d'aligner des éléments comparables pour insister sur leurs similarités

```
1 if (c>0) {...}
2 else if (c<0) {...}
3
```

Factorisation du code

- Pour des raisons historiques, ne pas dépasser 80 caractères par ligne.

Factorisation du code

- Pour des raisons historiques, ne pas dépasser 80 caractères par ligne.
- Pour des expressions longues, *factoriser* le travail en calculs plus petits stockés dans des variables élémentaires.

Factorisation du code

- Pour des raisons historiques, ne pas dépasser 80 caractères par ligne.
- Pour des expressions longues, *factoriser* le travail en calculs plus petits stockés dans des variables élémentaires.
- Décomposer un programme en sous-fonctions élémentaires de quelques lignes.
  Non seulement on y gagne en *lisibilité* mais aussi en *réutilisabilité*.

Choix des noms

Fichiers, types, fonctions, variables.

- Variables locales à une fonction : utiliser des noms courts à une lettre. Cette lettre n'est pas choisie au hasard (par exemple `t` pour un tableau, `i` pour un entier).

Choix des noms

Fichiers, types, fonctions, variables.

- Variables locales à une fonction : utiliser des noms courts à une lettre. Cette lettre n'est pas choisie au hasard (par exemple `t` pour un tableau, `i` pour un entier).
- Les noms de fonctions ont intérêt à être explicite : par exemple `int dichot(int a[], int n, int x)` pour une fonction qui fait une recherche dichotomique dans un tableau trié.

Choix des noms

Fichiers, types, fonctions, variables.

- Variables locales à une fonction : utiliser des noms courts à une lettre. Cette lettre n'est pas choisie au hasard (par exemple `t` pour un tableau, `i` pour un entier).
- Les noms de fonctions ont intérêt à être explicite : par exemple `int dichot(int a[], int n, int x)` pour une fonction qui fait une recherche dichotomique dans un tableau trié.
- On peut utiliser des underscore si le nom de fonction contient plusieurs mots `bool has_cycle(graphe g)` qui indique par un bouléen si le graphe possède un cycle.

Choix des noms

Fichiers, types, fonctions, variables.

- Variables locales à une fonction : utiliser des noms courts à une lettre. Cette lettre n'est pas choisie au hasard (par exemple `t` pour un tableau, `i` pour un entier).
- Les noms de fonctions ont intérêt à être explicite : par exemple `int dichot(int a[], int n, int x)` pour une fonction qui fait une recherche dichotomique dans un tableau trié.
- On peut utiliser des underscore si le nom de fonction contient plusieurs mots `bool has_cycle(graphe g)` qui indique par un bouléen si le graphe possède un cycle.
- On peut préférer séparer les mots par des majuscules : `bool hasCycle(graphe g)` (notation à-la-Java).
  OCaml limite par ailleurs l'usage des majuscules en première lettre.

Commentaires

- Un commentaire doit être une valeur ajoutée. Ne pas paraphraser le code. Exemple de commentaire inutile

```
1 // si x> 0, incrémenter, sinon décrémenter
2 if (x>0) {x++} else {x--}
3
```

Commentaires

- Un commentaire doit être une valeur ajoutée. Ne pas paraphraser le code. Exemple de commentaire inutile

```
1 // si x> 0, incrémenter, sinon décrémenter
2 if (x>0) {x++} else {x--}
3
```

- Indiquer les entrées-sorties

```
1 /*Entrées : a tableau trié d'entiers
2              n taille de a, x valeur cherchée*/
3 //Sortie : position de x
4 int dichot (int a[], int n, int x)
5
```

Le premier commentaire est une *précondition* : il sous-entend que le code ne vérifie pas ces hypothèses et peut planter en cas de non respect.
Le second commentaire est une *spécification* : il précise le comportement de la fonction

Commentaires

Invariant de boucle

- Si le programme contient une boucle, une propriété maintenue à chaque itération est appelée un *invariant* de boucle. Il est utile de préciser cet invariant pour expliquer le code et pour une future preuve de correction.

Commentaires

Invariant de boucle

- Si le programme contient une boucle, une propriété maintenue à chaque itération est appelée un *invariant* de boucle. Il est utile de préciser cet invariant pour expliquer le code et pour une future preuve de correction.
- Exemple du tri insertion

```
1  for(int i = 0 ; i <= n - 1 ; i++)// tri insertion
2       {          // Inv :a[0..i[ est trié
3                  x = T[i];
4                  j = i;
5                  while( j > 0 & T[j-1] > x )
6                      {
7                          T[j] = T[j-1];
8                          j = j - 1;
9                  }
10                 T[j] = x;}
11
```

Commentaires

Invariant de boucle

On peut aussi représenter l'invariant par un dessin comme dans l'algorithme du drapeau hollandais (classement des éléments d'un tableau selon trois catégories ordonnées)

```
1 int b=0, i=0, r=n ;
2 while (i<r){
3   //    0      b      i      r      n
4   //   +------+------+------+------+
5   //a  |  0   |   1  |  ??  |  2   |
6   //   +------+------+------+------+
7   }
```

On comprends que à chaque tour `a[0..b[` ne contient que la valeur 0, `a[b..i[` ne contient que la valeur 1 etc..

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)
- Par exemple, on peut introduire une fonction

```
1 void f(int a[], int g, int d)
```

pour travailler sur les indices dans $[\![g, d-1]\!]$ avec l'hypothèse $0 \leq g \leq d \leq |a|$.
Dans ce cas :

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)
- Par exemple, on peut introduire une fonction

```
1 void f(int a[], int g, int d)
```

pour travailler sur les indices dans $[\![g, d-1]\!]$ avec l'hypothèse $0 \leq g \leq d \leq |a|$.
Dans ce cas :

- le nombre d'éléments concernés est $d - g$

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)
- Par exemple, on peut introduire une fonction

```
1 void f(int a[], int g, int d)
```

pour travailler sur les indices dans $[\![g, d-1]\!]$ avec l'hypothèse $0 \leq g \leq d \leq |a|$.
Dans ce cas :

- le nombre d'éléments concernés est $d - g$
- Si on doit couper cet intervalle en deux ce sera avec $[\![g, m]\!]$ et $[\![d, m]\!]$ pour un $g \leq m \leq d$

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)
- Par exemple, on peut introduire une fonction

```
1 void f(int a[], int g, int d)
```

pour travailler sur les indices dans $[\![g, d-1]\!]$ avec l'hypothèse $0 \leq g \leq d \leq |a|$.
Dans ce cas :

- le nombre d'éléments concernés est $d - g$
- Si on doit couper cet intervalle en deux ce sera avec $[\![g, m]\!]$ et $[\![d, m]\!]$ pour un $g \leq m \leq d$
- le tableau tout entier correspond $g = 0$, $d = |a|$.

A propos des intervalles

- Pour représenter un intervalle (dans un tableau, une chaîne etc.) on peut utiliser systématiquement un indice gauche *inclus* et un indice droit *exclu* (comme le `range` de `Python`)
- Par exemple, on peut introduire une fonction

```
1 void f(int a[], int g, int d)
```

pour travailler sur les indices dans $[\![g, d-1]\!]$ avec l'hypothèse $0 \leq g \leq d \leq |a|$.
Dans ce cas :

  - le nombre d'éléments concernés est $d - g$
  - Si on doit couper cet intervalle en deux ce sera avec $[\![g, m]\!]$ et $[\![d, m]\!]$ pour un $g \leq m \leq d$
  - le tableau tout entier correspond $g = 0$, $d = |a|$.

- Nous respectons cette convention le plus possible.

Invariant de structure

- Un *invariant de structure* décrit une propriété toujours vraies pour les valeurs d'une structure de données.

Invariant de structure

- Un *invariant de structure* décrit une propriété toujours vraies pour les valeurs d'une structure de données.
- Il est utile de préciser cet invariant (même incomplètement) au niveau de la définition du type

```
1 struct ArrStack{
2   int capacity;
3   int size; // 0<= size <= capacity
4   int *data; // tableau de taille capacity
5       }
6
```

Compiler aide à trouver les erreurs

- On détecte les erreurs de syntaxe :

```
asup.c:9:1: error: expected declaration or statement
 }
```

Compiler aide à trouver les erreurs

- On détecte les erreurs de syntaxe :

```
asup.c:9:1: error: expected declaration or statement
 }
```

- ou encore celles de typage

```
asup.c:9:5: error: too many arguments to function 'f'
```

Compiler aide à trouver les erreurs

- On détecte les erreurs de syntaxe :

```
asup.c:9:1: error: expected declaration or statement
 }
```

- ou encore celles de typage

```
asup.c:9:5: error: too many arguments to function 'f'
```

- En cas d'erreur, aucun exécutable n'est produit.

Compiler aide à trouver les erreurs

- On détecte les erreurs de syntaxe :

```
asup.c:9:1: error: expected declaration or statement
 }
```

- ou encore celles de typage

```
asup.c:9:5: error: too many arguments to function 'f'
```

- En cas d'erreur, aucun exécutable n'est produit.
- Le cycle de travail consiste en de fréquents aller-retours entre l'édition du fichier source et la compilation.

Compiler aide à trouver les erreurs

- On détecte les erreurs de syntaxe :

```
asup.c:9:1: error: expected declaration or statement
 }
```

- ou encore celles de typage

```
asup.c:9:5: error: too many arguments to function 'f'
```

- En cas d'erreur, aucun exécutable n'est produit.
- Le cycle de travail consiste en de fréquents aller-retours entre l'édition du fichier source et la compilation.
- Compiler souvent ! La compilation n'est pas nécessairement chronophage avec un bon `Makefile` (cf plus tard)

Avertissements

- Si le comilateur émet un avertissement plutôt qu'une erreur, il va poursuivre jusqu'à la production de l'exécutable.

Avertissements

- Si le comilateur émet un avertissement plutôt qu'une erreur, il va poursuivre jusqu'à la production de l'exécutable.
- Un avertssement peut être négligé en première approche mais il devra être résolu avant le rendu du projet final.

Avertissements

- Si le comilateur émet un avertissement plutôt qu'une erreur, il va poursuivre jusqu'à la production de l'exécutable.
- Un avertssement peut être négligé en première approche mais il devra être résolu avant le rendu du projet final.
- Voici un exemple où le compilateur repère une variable non utilisée. Ce n'est pas propre !

```
asup.c:8:7: warning: unused variable 'x' [-Wunused-va
```

Avertissements

- Si le comilateur émet un avertissement plutôt qu'une erreur, il va poursuivre jusqu'à la production de l'exécutable.
- Un avertssement peut être négligé en première approche mais il devra être résolu avant le rendu du projet final.
- Voici un exemple où le compilateur repère une variable non utilisée. Ce n'est pas propre !

```
asup.c:8:7: warning: unused variable 'x' [-Wunused-va
```

- Utiliser l'option `-Wall` !

Compiler des programmes incomplets

- On peut n'avoir écrit que certaines fonctions du programme, et on peut très bien les compiler avec l'option `-c` qui produit un fichier objet sans édition de lien.

Compiler des programmes incomplets

- On peut n'avoir écrit que certaines fonctions du programme, et on peut très bien les compiler avec l'option `-c` qui produit un fichier objet sans édition de lien.
- Avec le mécanisme de l'arrêt prématuré (`abort()` en C), des assertions (`assert` en C et OCAML) ou des exceptions (`failwith` en OCAML), on peut ne compiler que des morceaux de codes qui ne sont que partiellement écrits :

```
1 if (n>100)
2     n=n-10;
3 else{
4   // TODO : je verrai plus tard
5   abort(); //quitter le programme prématurément
6    }
7
```

Utiliser des prototypes

- En C, l'écriture du prototype d'une fonction `f` permet d'appeler `f` avant d'avoir écrit son code.

Utiliser des prototypes

- En C, l'écriture du prototype d'une fonction `f` permet d'appeler `f` avant d'avoir écrit son code.
- Dans cet exemple, `f` n'a pas encore de corps. Mais la compilation avec `gcc -c` permet de se rendre compte que dans le corps de `main`, on appelle `f` avec trop d'arguments :

```
1  #include <stdio.h>
2
3  int f(int x); // f sera explicitée plus tard
4
5  int main(void)
6  {
7    f(2,3);
8    return 0;
9  }
10
```

Le compilateur ne détecte pas tout

- Un théorème célèbre (*th. de Rice*) a pour conséquence qu'aucun compilateur ne peut prévoir toutes les erreurs possibles.

Le compilateur ne détecte pas tout

- Un théorème célèbre (*th. de Rice*) a pour conséquence qu'aucun compilateur ne peut prévoir toutes les erreurs possibles.
- Par exemple un compilateur ne peut pas garantir dans tous les cas qu'on ne divisera jamais par 0, qu'on n'accèdera jamais à un tableau en dehors de ces bornes ou qu'on ne tombera jamais dans une boucle infinie.

Le compilateur ne détecte pas tout

- Un théorème célèbre (*th. de Rice*) a pour conséquence qu'aucun compilateur ne peut prévoir toutes les erreurs possibles.
- Par exemple un compilateur ne peut pas garantir dans tous les cas qu'on ne divisera jamais par 0, qu'on n'accèdera jamais à un tableau en dehors de ces bornes ou qu'on ne tombera jamais dans une boucle infinie.
- Les raisons pour lesquelles un programme plante sont incomplètement indiquée à l'exécution. Mais cette information lacunaire est quand même utile.
  Avec un *debugger* on peut aller plus loin dans la recherche du bug.

Erreur de segmentation

- L'*erreur de segmentation* est un plantage d'une application qui a tenté d'écrire dans une zone mémoire qui ne lui était pas allouée.

Erreur de segmentation

- L'*erreur de segmentation* est un plantage d'une application qui a tenté d'écrire dans une zone mémoire qui ne lui était pas allouée.
- Dans cet exemple le pointeur `variable_entiere` n'est pas initialisé et contient donc une valeur quelconque qui a de forte chance d'être une zone mémoire interdite en écriture.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      int *variable_entiere;
7      scanf("%d", variable_entiere);
8      return EXIT_SUCCESS;
9  }
10
```

Erreur de segmentation

Après compilation et exécution

```
 $ ./a.out
2
Erreur de segmentation (core dumped)
```

A noter que l'option `-Wall` détecte que le pointeur n'est pas initialisé.

En travaux.