Table des matières

Introduction

Ce document a pour but de vous montrer les diverses notions de style de programmation ("coding style").
La programmation ne consiste en effet pas seulement en la réalisation de programmes qui fonctionnent.
Il est indispensable que chaque programme soit codé de manière propre et lisible.

Au début, vous n'allez programmer que de petits projets, seuls. Plus tard vous programmerez en groupe sur des projets de taille bien plus importante. Dès lors, il est important d'adopter de bonnes habitudes dès le début de l'apprentissage.

Lorsque vous soumettez un programme, gardez à l'esprit que vous vous adressez à un programmeur ne connaissant pas l'énoncé. Il n'est donc pas nécessaire de détailler chaque commande, mais bien d'expliquer les divers algorithmes utilisés et de spécifier le rôle des variables et instructions non triviales.

Souvenez-vous que dans le temps consacré à un code source durant sa vie, seul **20%** du temps est consacré à la programmation, les **80%** restants sont consacré à la maintenance de ce code (mise à jour, ...). Il est donc indispensable que ce code soit **clair**.

Nous vous **obligeons** à suivre le style qui est défini dans ce document.

Nous avons bien conscience que le style que nous définissons ici n'est pas le seul style de programmation valide.
Il en existe bien d'autres, qui sont tout aussi valides et compréhensibles, vous constaterez que d'autres professeurs utilisent un style différent de celui proposé ici, sans que cela nuise à la compréhension ou à la lisibilité de leurs programmes.

Si nous vous imposons ce style, c'est parce que nous croyons qu'il est bien de vous habituer à utiliser un style clair et également de vous préparer aux réalités de l'entreprise. Souvent, vous serez tenus de suivre un style de programmation qui n'est pas le vôtre, tout simplement par souci de standardisation.

Pour rappel, le cours utilise exclusivement le langage C, dans sa version C99.

Le présent document a été réalisé par **Pierre RONDOU** dans la cadre du cours de *Projets de programmation* et est inspiré de *A guide to coding style* du Professeur **Justus PIATER**.

Si vous constatez une erreur, approximation ou pensez à une amélioration, n'hésitez pas à me contacter : **prondou@gmail.com**

Chapitre 1

Les commentaires

Ce chapitre va traiter de l'utilisation des commentaires dans un programme.
Pour rappel, nous vous demandons de commenter votre programme comme si vous parliez à un autre programmeur. Cela signifie qu'il est indispensable de décrire le programme et son utilisation, mais pas chaque commande usuelle, l'abus de commentaire étant nuisible.

L'utilisation de noms de variables clairs aide grandement à la compréhension, réduisant de fait l'utilisation de commentaires.

1.1 Langue utilisée

Vous pouvez indifféremment utiliser l'anglais ou le français pour vos commentaires.
Néanmoins, évitez l'anglais si vous le ne maîtrisez pas.
On ne regardera pas à l'orthographe des commentaires, mais ceux-ci doivent rester compréhensibles !

Il est par contre **interdit** de mélanger plusieurs langues dans les commentaires.
Les noms de variables doivent être obligatoirement en langue **anglaise**.

1.2 Caractères spéciaux

L'utilisation de caractères spéciaux (i.e, non ASCII) n'est pas autorisée.

1.3 L'en-tête.

Un commentaire d'en-tête doit reprendre les informations suivantes :
- Le nom, prénom de l'auteur.
- Une description générale de l'énoncé
- Éventuellement, une petite explication du fonctionnement du programme (algorithmes, ...).

Un exemple de bon en-tete :

```
1  /* Project 1 by Francois Deschamps 08-02-2010
2   *
3   * This program computes the middle-point between
4   * two entered coordinates and the slope of the line
5   * going through these points before displaying them
6   * in the console.
7   *
8   */
```

1.4 Les commentaires de fonction

1.4.1 Positionnement des commentaires de fonction

Les commentaires de fonctions doivent être positionnés au niveau du **prototype**, de la fonction.
Pour rappel, le prototype d'une fonction se situe dans un fichier header (.h) ou bien au au début d'un fichier source (.c).

1.4.2 Format des commentaires de fonction

Pour vos commentaires de fonctions, vous **devez** utiliser un style de commentaire "formaté" : le **DOxygen**.

```
1  /**
2   * Returns an Image object that can then be painted on the screen.
3   * The url argument must specify an absolute {@link URL}. The name
4   * argument is a specifier that is relative to the url argument.
5   *
6   * This method always returns immediately, whether or not the
7   * image exists. When this applet attempts to draw the image on
8   * the screen, the data will be loaded. The graphics primitives
9   * that draw the image will incrementally paint on the screen.
10  *
11  * @param  url  an absolute URL giving the base location of the image
12  * @param  name the location of the image, relative to the url argument
13  * @return      the image at the specified URL
14  * @see         Image
15  */
```

Listing 1.1 – Tiré de *A guide to Coding Style*, J. Piater.

L'utilisation de commentaires formatés permet la génération rapide d'une documentation au format HTML à partir du code source.

1.5 Le positionnement des commentaires

Les commentaires doivent être positionnés **avant** l'instruction qu'ils décrivent. Au pire, ils peuvent être sur la **même ligne** que l'instruction.

1.6 Le format des commentaires

Le langage C permet deux types de commentaires :

– Le commentaire par ligne (//).
  Dans ce cas, tout ce qui suit sur la même ligne sera considéré comme commentaire par le compilateur.
– Le commentaire par bloc (/* .... */).
  Dans ce cas, tout ce qui se trouve entre le symbole de début et celui de fin est considéré comme commentaire.

Les règles pour le cours sont les suivantes :
– Vous pouvez, pour les commentaires d'une seule ligne, utiliser l'un ou l'autre style.
– Pour les commentaires à plusieurs lignes, vous **devez** utiliser les commentaires par bloc.

Ainsi, le type de commentaire suivant n'est pas correct.

```
1  while (stop != EOF) {       //Invariant permet de ne pas
2    x1 = x2;                  //lancer la boucle si ctrl-d
3    y1 = y2;                  //a la 1er entree
```

Listing 1.2 – Un mauvais exemple.

Chapitre 2

Noms de variables, constantes et fonctions

Le choix du nom des variables, constantes et fonctions est primordial.
Si ce choix est bien réalisé, il permet en effet de comprendre l'utilité de la variable au premier coup d'œil.

2.1 Conventions générales

Voici quelques conventions utilisées dans le choix des noms :
- Évitez d'utiliser des noms raccourcis qui rendent la description des variables moins précise (ex, utilisez *firstName* plutôt que *fName*).
- Évitez les noms trop longs. À l'opposé du point précédent, il est inutile de préciser trop en avant les noms de variables ou de fonctions (ex, utilisez *setLength* plutôt que *setTheLengthField*).
- Évitez d'utiliser des noms trop proches, que ce soit au point de vue du sens ou de l'écriture, les uns des autres.

2.2 Conventions spécifiques aux noms de variables

Les noms de variables doivent absolument reprendre la signification de ce qu'ils contiennent.
La seule exception à cette règle est les compteurs des boucles *for* qui sont eux généralement nommés i, j, k, l, ...

Les noms de variables doivent commencer par une **minuscule**, si le nom de la variable est composé, les mots suivant seront accolés au précédent et commenceront par une **majuscule**. Ex, on utilisera **salesOrder** plutôt que **sales_order**.

2.3 Conventions spécifiques aux noms de constantes

Les noms de constantes doivent être écrit en **lettres capitales**, si la constante est composée de plusieurs mots, ceux-ci seront séparés par un caractère de soulignement

(*underscore*). Ex : **DAY_OF_WEEK**

2.4 Conventions spécifiques aux noms de fonctions

Tout comme les noms de variables, les noms de fonctions doivent commencer avec une **minuscule** et, si le nom de la fonction est composé, les mots suivant seront accolés aux précédents et commenceront par une **majuscule**

Les noms de fonctions devraient toujours commencer par un **verbe** dénotant le type d'action que la fonction va effectuer. Ex : addTwo, setName, ...

2.5 Conventions spécifiques aux structures

Les noms de structures doivent commencer par une **Majuscule**, et suivent ensuite le modèle des noms de variables.
Notez que généralement, les noms de structures propres se terminent par **_t**.

Vous devrez néanmoins utiliser **typedef** qui permet de définir un type à partir d'une structure, enlevant la nécessité du **_t**.
Un nom de type suit le même principe de noms qu'une structure.

```
1 typedef struct Node_t {
2   void *content;
3   struct Node_t *next;
4 } Node;
```

Listing 2.1 – tiré de *A guide to Coding Style*, J. Piater.

Chapitre 3

Organisation du code

L'organisation du code est primordiale à sa compréhension. Un code mal aéré ou mal indenté sera difficilement compréhensible, même s'il respecte les points précédemment évoqués.

3.1 Longueur des lignes

Une ligne ne pourra jamais dépasser **80 caractères**.
Il n'y a pas de règle claire lorsque vous tombez sur un cas avec une instruction qui fait dépasser la longueur d'une ligne, néanmoins, la manière dont vous "couperez" vos lignes doit être **claire** et **cohérente**.
Il est évidemment **interdit** de placer plus d'une instruction par ligne, à l'exception de certains cas dans l'instruction **switch**.

3.2 L'indentation

3.2.1 Exemple

L'indentation consiste à insérer des espaces avant une instruction afin de séparer les blocs de code en différents niveaux.
C'est un aspect essentiel dans la lisibilité du code :

```
1  if (unlikely(prev->policy == SCHED_RR)) {
2      if (!prev->counter) {
3          prev->counter = NICE_TO_TICKS(prev->nice);
4          move_last_runqueue(prev);
5      }
6  }
7  switch (prev->state) {
8      case TASK_INTERRUPTIBLE:
9          if (signal_pending(prev)) {
10             prev->state = TASK_RUNNING;
11             break;
12             }
13         default:
14             del_from_runqueue(prev);
15     case TASK_RUNNING:;
16 }
17 prev->need_resched = 0;
```

Listing 3.1 – Source : Linux Source Code.

```
1  if (unlikely(prev->policy == SCHED_RR)) {
2  if (!prev->counter) {
3  prev->counter = NICE_TO_TICKS(prev->nice);
4  move_last_runqueue(prev);
5  }
6  }
7  switch (prev->state) {
8  case TASK_INTERRUPTIBLE:
9  if (signal_pending(prev)) {
10 prev->state = TASK_RUNNING;
11 break;
12 }
13 default:
14 del_from_runqueue(prev);
15 case TASK_RUNNING:;
16 }
17 prev->need_resched = 0;
```

Listing 3.2 – Le meme code source, sans indentation.

La différence entre les deux est flagrante.

3.2.2 Nombres d'espaces d'indentation

Dans le cadre de ce cours, vous **devez** utiliser **3** espaces pour votre indentation.

3.2.3 Utilisation de la tabulation

L'utilisation de la tabulation pour l'indentation est **interdite** pour ce cours. En effet, celle-ci représente 8 espaces blancs par défaut. Les éditeurs modernes permettent de fixer le nombre d'espaces blanc **affichés**, mais une fois ouvert dans un autre éditeur, le réglage est perdu.
La tabulation est interdite essentiellement car elle brise souvent la règle des 80 caractères.

Notez que la plupart des éditeurs modernes permettent de remplacer automatiquement le caractère de tabulation par le nombre d'espaces blancs de votre choix.

3.3 Les accolades

Les accolades sont indispensables au bon fonctionnement du programme, nous vous laissons quelques degrés de liberté quant à leur organisation :

3.3.1 Le positionnement de l'accolade d'ouverture

Dans le cadre de ce cours, vous **devez** mettre l'accolade d'ouverture sur une **nouvelle ligne**.

Exemple :

```
1  for (int i = 0; i < length, i++)
2  {
3      tab[i] = 0;
4  }
```

Listing 3.3 – Sur une nouvelle ligne.

3.3.2 Cas d'une instruction unique

Dans le cas où le bloc est composé d'une unique instruction, il est permis de supprimer les accolades.

```
1  for (int i = 0; i < length, i++)
2      tab[i] = 0;
```

Listing 3.4 – Sans accolade.

Notez que si vous adoptez ce système, vous vous retrouverez dans des cas où la non-utilisation d'accolades rend le code confus. C'est le cas par exemple des boucles imbriquées :

```
1  for (int w = 0; w < with; w++)
2      for (int h = 0; h < height; h++)
3          for (int d = 0; d < depth; d++)
4          {
5              a[w][h][d] = 0;
6              b[w][h][d] = 1;
7              c[w][h][d] = 2;
8          }
9  if (length < -1)
10 {
11    ...
12 }
13 ...
```

Listing 3.5 – boucles imbriquées sans accolades.

```
1  for (int w = 0; w < with; w++)
2  {
3      for (int h = 0; h < height; h++)
4      {
5          for (int d = 0; d < depth; d++)
6          {
7              a[w][h][d] = 0;
8              b[w][h][d] = 1;
9              c[w][h][d] = 2;
10     }
11   }                    // unnecessary, but more readable
12 }                      // unnecessary, but more readable
13 if (length < -1) {
14    ...
15 }
16 ...
```

Listing 3.6 – boucles imbriquées avec accolades.

Le premier cas peut prêter à confusion et, en tous cas, rend la compréhension du code plus compliquée. Vous devez utiliser la deuxième version si vous tombez face à cette situation.

Il reste néanmoins un cas de boucles imbriquées où l'utilisation d'accolades n'est pas indispensable.

```
1  for (int w = 0; w < with; w++)
2      for (int h = 0; h < height; h++)
3          for (int d = 0; d < depth; d++)
4              a[w][h][d] = 0;
5
6  if (length < -1)
7  {
8    ...
9  }
10 ...
```

Listing 3.7 – boucles imbriquées sans accolades, mais claires.

Ce dernier cas est suffisamment explicite que pour vous puissiez vous passer des accolades.

3.4 L'aération du code

L'aération du code est, tout comme l'indentation, une étape indispensable pour favoriser la compréhension de votre code.

3.4.1 Exemple

```
1  while (stillValidInputs) {     // as long as new keys are entered
2    element = malloc(sizeof(DataPair)); // allocate memory to *element
3    if (element != NULL) {       // check memory allocation ok
4      printf("New key: ");
5      fgetstr(element -> key, 20); // call own function
6      /* Continue if string length > 0 */
7      if (strlen(element -> key) != 0) {
8        printf("New value: ");
9        fgetstr(element -> value, 20);
10       int index = findDataPairIndex(database, element -> key);
11       if (index != -1) {
12         DataPair *existingDataPair = vecGet(database, index);
13         strcpy(existingDataPair -> value, element -> value);
14       }
15       else {
16       /* Store element in database */
17       vecSet(database, vecSize(database), element);
18     }
19   }
20     else {
21       stillValidInputs = 0;       // no more valid inputs
22       free(element);
23     }
24   }
25 }
```

Listing 3.8 – Code non aéré.

```
1  while (stillValidInputs) { // As long as new keys are entered
2
3    element = malloc(sizeof(DataPair)); // allocate memory to *element
4    if (element != NULL) {       // check memory allocation ok
5      printf("New key: ");
6      fgetstr(element -> key, 20); // call own function
7
8      /* Continue if string length > 0 */
9      if (strlen(element -> key) != 0) {
10       printf("New value: ");
11       fgetstr(element -> value, 20);
```

```
12          int index = findDataPairIndex(database, element -> key);
13
14          if (index != -1) {
15            DataPair *existingDataPair = vecGet(database, index);
16            strcpy(existingDataPair -> value, element -> value);
17          }
18
19          else {
20          /* Store element in database */
21          vecSet(database, vecSize(database), element);
22        }
23      }
24        else {
25          stillValidInputs = 0;       // no more valid inputs
26          free(element);
27        }
28      }
29  }
```

Listing 3.9 – Le meme code, aéré.

Notez qu'il reste d'autres erreurs de style, mais ce le but de cet exemple n'est pas de les corriger.

3.4.2 Comment aérer son code

Il n'existe pas réellement de standard pour l'aération d'un code. Néanmoins, vous remarquerez vite qu'il est important de grouper les instructions liées entres elles.

3.5 Les espaces blancs

L'utilisation d'espaces blancs est **obligatoire** dans les cas suivants :
- Autour (à droite **et** à gauche des opérateurs (+, -, =, ...).
- Après les virgules.
- Après les point-virgules dans une instruction *for*.
- Après les mots clés (if, else, for, ...).

L'utilisation d'espace blancs est **interdite** dans les cas suivants :
- Autour des opérateur de sélection (., ->, []) (Ex : node->next).
- Après une parenthèse ouverte.
- Avant une parenthèse fermée.
- À coté d'un opérateur unaire (*, ++, –, !).

3.6 Espaces en fin de ligne

Veuillez noter qu'il est interdit de laisser des espaces blancs en fin de ligne.

3.7 Ligne vide en fin de fichier

Il faut absolument laisser une (et une seule) ligne vide à la fin de chaque fichier source (.h et .c).