Conventions de programmation en C et Feedback projet

Dans ce document nous avons rassemblé les points principaux d'une *discipline de programmation* que nous vous demandons d'adopter.

Les codes, par exemple **E11** ou **O5**, sont employés pour vous donner un feedback rapide sur les rendus (intermédiaires) des projets.

Code E *Conventions d'écriture*

E1 ***Noms*** des **fonctions**, des **variables**, **modèles de structure** et des **symboles** (#define) -> chacun dispose d'un espace de nom indépendant :

- utilisation des majuscules/minuscules:
  - **E11** ***Constante symbolique*** (avec #define): 100% en MAJUSCULES, ex : **`VITESSE_MAX`**
  - **E12** ***variable (y compris tableau et pointeur), fonction*** : 100% en minuscules, ex : **`dessiner( )`**
  - **E13** ***Modèle de structure*** (C), Classe (C++): première lettre en Majuscule, les suivantes en minuscules, ex : **`Forme`**

    Type *construit avec* **`typedef`** *:* si le type est basé sur un ***modèle de structure***, au choix :
    Soit ajouter **`_t`** à la suite du nom de modèle de struct, ex : **`Forme_t`**
    Soit utiliser le nom du modèle de struct 100% en MAJUSCULE, ex : **`FORME`**

- **E14** ***Nom composé*** de plusieurs mots: avec un caractère souligné entre chaque mot,
  ex: **`date_debut, nb_eleves, moyenne_finale`**, etc...
  Variante acceptée : les mots supplémentaires commencent avec une majuscule,
  ex : **`dateDebut, nbEleves, moyenneFinale`**, etc…

- **E15** ***nom à une lettre*** pour les variables de boucle **`: i,j,k`**... ou pour les variables utilisées très fréquemment sans risque d'ambiguïté. D'une manière générale, il faut privilégier les noms courts mais parlant (voir ci-dessous)

- **E16** choisir des ***noms parlant*** pour variable, fonction, module. Méthode de travail : le nom doit être facilement prononçable et ***permettre de deviner le but*** de l'entité nommée. L'article suggéré en biblio p 5 va plus loin dans cette voie.

- **E161** ***noms numérotés*** **:** choix rarement pertinent. De plus au-delà de deux variables numérotées il faut se poser la question d'utiliser plutôt un tableau et des symboles pour les indices. Pour des fonctions numérotées il vaut mieux chercher des noms plus parlants ou condenser les fonctions en une seule avec un ou plusieurs paramètres.

- **E17** le nom d'une variable booléenne ou d'une fonction qui renvoie un booléen doit ***correspondre à l'état VRAI*** pour faciliter sa composition dans des expressions et son test avec les instructions de contrôle. Ex :

```
if(collision(…))
```

E2 Sem2 : Préfixer le ***nom d'une fonction ou d'un symbole exporté*** par le nom du module. Par exemple, si dans le module **`vehicule.c`** on écrit une fonction exportée qui contrôle le déplacement, alors elle devra s'appeler **`vehicule_deplace( ).`** Cela permet de savoir d'où vient chaque fonction. Même chose pour un symbole exporté. Cette règle permet aussi d'éviter les doublons quand on importe (include) plusieurs fichiers en-tête.

Code D *Conventions de documentation* : **D0** : utilisez le temps présent et faites un effort de grammaire et d'orthographe

CL : suivez un cours (gratuit) au **Centre de Langue** pour vous (re)mettre à niveau

D1 Description ***en début de fichier source*** (**.c** et **.h**): indiquer le nom du fichier, une brève description du module, le numéro de version, la date de dernière mise a jour, le nom du responsable du test du module. *Travail en groupe* : ajouter une ligne avec le nom de tous les membres du groupe.

D2 Devant chaque prototype de fonction exportée ***dans le fichier en-tête (.h)***: indiquer brièvement ce que fait la fonction (mais PAS comment elle le fait). Si nécessaire, indiquer le domaine de validité des paramètres. *En cas de paramètre de type tableau* : indiquer si le tableau est modifié par la fonction. Inutile de décrire ce qui est fait avant ou après la fonction documentée.

D3 Dans le code: pas trop de commentaires (dites le *but*, pas le *comment*). Des noms parlants et l'indentation suffisent souvent.

D4 Modèles de structure: indiquer le but de chaque champ et leur domaine de validité si nécessaire.

Code L *Lisibilité du code sur écran et après impression*

Le code n'est pas du texte en langue naturelle comme dans les journaux, les romans, etc... Sa disposition sur la page ou l'écran est essentielle pour sa compréhension.

L00 fournir une impression AVEC les numéros de ligne ; c'est l'éditeur de CODE qui doit le faire, pas vous (c'est possible avec **geany**)

L01 POLICE à chasse FIXE: l'éditeur de CODE utilise normalement une **`police de caractères à chasse fixe`** (exemple : **`courrier New`**) pour respecter les alignements d'instructions sur plusieurs lignes (cf L25).

L1 INDENTATION : Le code doit être indenté ; plusieurs styles d'indentation existent : l'auteur du programme est libre de choisir son style MAIS le style d'indentation doit être ***le même pour tout le code*** d'un programme.

- **L11** faire une indentation pour le corps d'une fonction et pour toutes les structures de contrôle (instruction contrôlée **simple** ou **bloc**)
- **L12** décalage minimum = 2 espaces sinon on ne voit plus la structure du code
- **L13** PAS de décalage supérieur à 4 espaces, PAS 2 indentations par structure de contrôle, sinon le code se décale trop à droite (voir **L2**)

L2 PASSAGES A LA LIGNE et ALIGNEMENT : les expressions complexes doivent et peuvent rester lisibles car le compilateur autorise de passer à la ligne avant la fin de l'instruction signalée par le caractère **;** .

L21 une instruction très longue et peu structurée doit être simplifiée en plusieurs instructions en utilisant des variables intermédiaires.

L22 une instruction longue mais facile à comprendre doit être organisée ***sur plusieurs lignes*** **si elle dépasse la largeur de ce qui est visible dans l'éditeur** ; de plus la limite de la largeur A4 « portrait » est généralement visible dans l'éditeur. Si rien n'est fait l'impression coupe ce qui déborde ou poursuit l'impression sur la ligne suivante mais sans respecter l'indentation. *Le résultat est très pénible à lire, surtout pour la personne qui doit noter la lisibilité du code...* Donc il faut introduire soi-même un ou plusieurs passages à la ligne et aligner la ligne suivante pour rendre l'expression la plus lisible possible comme pour L25. Valable aussi pour appel de fonction. ex :

```
if(nb_robot > 0 && nb_obstacle > 0)
    deplace_robot( tab_robot, nb_robot,
                   tab_obstacle, nb_obstacle);
```

L23 dans le cas de **printf( )** avec une très longue chaîne pour le format, *on ne peut pas insérer de passage à la ligne*, par contre on peut morceler une longue chaîne de format en plusieurs chaînes qui se suivent SANS mettre de virgule entre les chaînes. On peut passer à la ligne entre ces chaînes consécutives. Exemple :

```
printf("on peut passer à la ligne "
       "entre %d chaînes\n", nb_chaine);
```

L24 ajouter des *lignes vides* pour séparer des sections indépendantes d'un fichier (voir ci-dessous Organisation) mais aussi entre chaque définition de fonction et à l'intérieur d'une fonction entre chaque partie réalisant une tâche bien identifiée, au minimum entre les déclarations et le reste.

L25 plusieurs instructions similaires qui se suivent doivent être alignées pour faciliter leur lecture. Exemple : un bloc de déclarations avec initialisation, plusieurs case dans un switch :

```
char accueil[]     =  "bonjour";
char erreur_nom[]  =  "max 8 lettres";
```

```
switch(operateur)
{
  case ADD   : printf("%f\n", v1 + v2);    break;
  case POWER : printf("%f\n", pow(v1,v2)); break;
  default    : printf("erreur!\n");
}
```

Code O *Conventions d'organisation*

O1 Le module **`mon_source.c`** est organisé selon l'ordre suivant:

- **O11** tous les **`#include`**, dans l'ordre: **`< xxx.h >`**, **`" yyy.h"`** , **`"mon_source.h"`**
- **O12** tous les **`#define`** supplémentaires utilisés seulement dans ce fichier
- **O13** description des modèles de structure
- **O14** déclaration de toutes les fonctions *restreintes au fichier* (avec **`static`**)
- **O15** déclaration des éventuelles variables *globales au fichier* (avec **`static`**)
- **O16** **définition** de toutes les fonctions exportées
- **O17** **définition** des fonctions **`static`**

O2 Le fichier **`mon_source.h`** est organisé de la façon suivante:

- le fichier doit adopter la structure suivante (explication au chap 12):

```
#ifndef MON_SOURCE_H
#define MON_SOURCE_H
                /* ici contenu du fichier */
#endif
```

O3 **mon_source.h** contient seulement l'information *exportée* de mon_source.c, selon l'ordre suivant :

- **O31** éventuels **`#include < xxx.h >, " yyy.h"`**
- **O32** éventuels **`#define`** des symboles exportés
- **O33** éventuels **`typedef`** des modèles de structure exportés
- **O34** éventuelles description complète des modèles de structure exportés (sauf pour type opaque)
- **O35** prototypes des fonctions exportées (note: pour une fonction, le mot clef `extern` n'est pas obligatoire)

Code R *Restrictions*

R1 **Pas d'instruction `goto`** : cette instruction peut être remplacée dans 99.99 % des cas par les instructions **`break`, `continue`, `return`** ou par l'utilisation et le test d'une variable booléenne (ayant seulement les valeurs Vrai ou Faux). Le problème avec **`goto`** vient du label de destination qui peut être n'importe où dans la même fonction. L'ordre d'exécution des instructions peut devenir artificiellement compliqué, d'où le nom de "*spaghetti code*" pour ce genre de code. Inutile de dire qu'il est donc plus difficile à comprendre et à mettre au point. L'usage du **`goto`** dans les projets sera pénalisé s'il peut être remplacé par **`break`, `continue`,** ou **`return`**.

R2 **Pas de variable globale à plusieurs fichiers (semestre 2)**: une variable globale est accessible et modifiable partout dans votre code, même dans des fichiers séparés.

Une variable globale est seulement acceptable si elle est déclarée comme une ***constante*** avec le mot clef **`const`**.

L'autre possibilité qui est tolérée *au semestre 2* est d'utiliser le mot clef **`static`** pour limiter la visibilité d'une variable globale au fichier dans lequel elle est déclarée. En dehors de ces deux cas, l'usage d'une variable globale dans les projets sera soit pénalisée, soit autorisée dans un cadre très précis

Remarque : les symboles créés avec #define ne sont pas des variables, au contraire leur usage est encouragé pour éviter de parsemer votre code de valeurs brutes que l'on risque d'oublier de mettre à jour en cours de développement (cf antipattern des *magic numbers*).

R3 **Paramètre de type pointeur**: l'usage des pointeurs réduit la lisibilité du code et introduit plus de risques d'erreurs. Pour cette raison, leur usage est restreint au passage de *tableau* et à la transmission de l'*adresse d'une variable que la fonction désire modifier*. L'usage d'un pointeur est interdit si on veut seulement donner l'accès en lecture à une variable, sans la modifier ; il suffit de transmettre sa valeur pour cela.

Bibliographie et liens

Robert Green and Henry Ledgard. 2011. Coding guidelines: finding the art in the science. *Commun. ACM* 54, 12 (December 2011), 57-63. http://doi.acm.org/10.1145/2043174.2043191 -63. Remarque: cet article utilise un style different pour les noms de variable et de fonction

http://fr.wikipedia.org/wiki/Chasse_(typographie)

Code P *sur le rendu (intermédiaire) d'un projet automne ou printemps*

P1 **Mauvaise compréhension de la donnée / analyse insuffisante** : relire ce qui concerne cette partie / approfondir l'analyse

P2 **Principe d'abstraction** : il manque une ou plusieurs fonctions qui donnent une vue générale sans rentrer dans les détails. De telles fonctions sont utiles pour faciliter la compréhension du code de la même manière qu'une table des matières permet d'avoir une idée globale d'un livre sans le lire en entier. *Ces fonctions sont justifiées même si elles ne sont appelées qu'une seule fois*.

P3 **Principe de réutilisation du code** : votre code comporte plusieurs sections qui réalisent la même tâche sur des variables différentes : définissez une fonction paramétrée qui remplace chaque section de code par un appel de la fonction.

P4 **Principe de séparation des fonctionnalités** : cherchez à ne faire qu'une seule tâche bien identifiée par fonction. MAIS attention à l'excès inverse ! Evitez l'abus de décomposition : une fonction ne doit pas être remplaçable pas un simple opérateur (exemple vu: une fonction qui calcule la différence de 2 nombres...) ou une expression logique simple. L'équilibre n'est pas toujours facile à trouver.

P5 **Utiliser des symboles** : (avec #define ou **enum**) pour remplacer les *magic numbers* dans votre code.
Remarque : un symbole n'est pas une variable globale ; c'est une simple constante.

P6 **Variables intermédiaires** :

- **P61** Si une expression est trop longue il faut qu'elle soit bien organisée, éventuellement sur plusieurs lignes. Sinon, créez des variables intermédiaires.
- **P62** On peut utiliser les paramètres formels comme des variables locales.
- **P63** inutile de créer une variable locale pour récupérer le résultat d'une expression et ensuite faire un `return` de cette variable. Autant faire directement un `return` ***`expression`***.

P7 **structures de contrôle conditionnel / inconditionnel**: l'instruction `switch` + **case** + **break** + **default** est préférable à une profusion de `if + else` lorsque les tests portent sur des valeurs d'une variable entière ou d'un caractère. Dans un **switch**, il n'est pas nécessaire d'avoir un `break` après un `return` puisque cette instruction n'est jamais atteinte.

P8 **Incohérence dans votre architecture logicielle** : un ou plusieurs prototypes de fonctions indiquent une dépendance qui n'apparait pas dans le dessin de votre architecture logicielle.

P9 **Incohérence du prototype de fonction** (rendu intermédiaire): ce prototype de fonction ne permet pas de réaliser le but de la fonction