Conventions de programmation en C++ (sem1&2) 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 **L14**, sont employés pour vous donner un feedback rapide sur le code source rendu pour les projets.

Code E *Conventions d'écriture*

E1 ***Noms* variables, fonctions, types, constantes énumérée et des symboles** (#define) :

- utilisation des majuscules/minuscules:
  - **E11** ***Constante symbolique*** (avec **`#define`** ou avec **`enum`**):
    100% en MAJUSCULES, ex : **`VITESSE_MAX`**

  - **E12** ***variable (y compris avec `const` et `constexpr`), tableau, vector, pointeur, etc), fonction*** :
    100% en minuscule sauf exception pour un nom composé ci-dessous :

    - **E121** : ***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…

  - **E13** ***Type associé à une struct, classe*, *enum***:
    première lettre en Majuscule, les suivantes en minuscules, ex : **`Forme`**

    - **E131** Même variante acceptée pour les noms composés que pour E121

- **E15** ***nom à une lettre*** pour des variables (locales) 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.

- **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(…))
```

Code D *Conventions de documentation*

D0 : dans le rapport ou les commentaires, utilisez le temps présent et faites un effort de grammaire et d'orthographe. Nous acceptons l'anglais.

D1 Description ***en début de fichier source*** (**.cc** et **.h**): indiquer le nom du fichier, le nom et prénom de l'auteur(e), numéro de version.

D2 Devant chaque prototype de fonction en début de fichier: Normalement un choix judicieux de nom de fonction et de nom de paramètre suffit, i.e. il est inutile d'ajouter un commentaire. S'il le faut, indiquer brièvement le BUT de la fonction (mais PAS comment elle l'atteint). Si nécessaire, indiquer le domaine de validité des paramètres. 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 leur nom ne suffit pas.

D5 (sem2) Déclaration d'une classe: le nom d'un attribut devrait suffire à indiquer son but ;
ajouter le domaine de validité en commentaire si nécessaire.

Même chose pour le nom d'une méthode : choisit un nom parlant, adopter la même convention pour getter et setter

La définition des méthodes d'une classe doit être *externalisée* = en dehors de la déclaration de la classe. La motivation de cette convention vient de la programmation modulaire car la déclaration de la classe va dans *l'interface* du module (.h) tandis que la définition des méthodes va dans *l'implémentation* du module (.cc).

Nous tolérons la définition de la méthode si elle tient sur la ligne de déclaration pour deux cas :

- un constructeur limité à une liste d'initialisation
- une méthode getter

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

La disposition correcte du code sur la page ou l'écran est essentielle pour sa bonne 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).

L02 Le nombre maximum de caractère par ligne est 87 = pas de wrapping (geany est paramétrable pour matérialiser cette limite)

L03 Le nombre maximum de lignes par fonction est 40 = taille écran

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**)

Exceptions acceptées au « style unique » : trois styles sont acceptés dans le même code source pour une ***instruction contrôlée simple***

a) sur la même ligne que l'instruction de contrôle.
Ex : `if(x !=0) y = 1/x;`

b) sur la ligne suivant l'instruction de contrôle, avec indentation, sans utiliser d'accolade.
Ex :
```
if(x !=0)
        y = 1/x;
```

c) L'usage d'accolade est une bonne pratique même pour le contrôle d'une seule instruction mais nous ne la rendons pas obligatoire ; cela permet d'écrire des fonctions plus compactes visibles sur une seule page écran.
Ex : ici vous pouvez utiliser votre propre style d'accolade du moment qu'il reste le même pour tout le code (L1):
```
if(x !=0)
{
    y = 1/x);
}
```

L12 Décalage minimum = 2 espaces sinon on ne voit plus la structure du code
Attention danger : votre code sera évalué avec la valeur standard de la tabulation = 4 espaces
Si vous adoptez une valeur plus petite pour TAB votre code risque de dépasser les 87 caractères pour la personne qui note votre code

L13 PAS de décalage supérieur à 4 espaces

L14 PAS 2 indentations par structure de contrôle (une fois pour l'accolade + une fois pour le code), sinon le code se décale trop à droite.
Exemple à ne pas suivre :

```
for(i=0 ; i< MAX ; ++i)
    {
          cout << "cette instruction est doublement indentée") << endl;
    }
```

L15 votre indentation sera évaluée avec geany ; vérifier votre avec geany si vous utiliser TAB dans un autre éditeur de code.

L16 Cas particulier du **`switch`** : on indente chaque **`case`** car ils peuvent avoir leur propre bloc pour déclarer des variables locales.

L17 (sem2) Déclaration d'une classe : NE PAS indenter les mots clef **`public, private, protected`**
Indenter les attributs et méthodes

L18 (sem2) Le contenu d'un **namespace** doit être indenté

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 87 caractères** ; configurez **geany** pour matérialiser cette limite. 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

Valable aussi pour un appel de fonction comme dans cet exemple d'un appel de fonction sur 2 lignes avec alignement des paramètres:

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

L23 dans le cas de **cout** avec une très longue chaîne à afficher, *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 :

```
cout << "on peut découper une chaîne en "
        "deux morceaux avec leurs guillemets doubles" << endl ;
```

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.

L25 plusieurs instructions similaires qui **se suivent sur des lignes consécutives** peuvent être alignées pour faciliter leur lecture.

Exemple1 : un bloc de déclarations avec initialisation,

```
string accueil    =  "bonjour";
string erreur_nom =  "max 8 lettres";
```

Exemple2 : plusieurs **case** dans un **switch** :

```
switch(integer_value)
{ // ADD et POWER sont définis avec enum
  case ADD   : cout << v1 + v2   << endl; break;
  case POWER : cout << pow(v1,v2)<< endl; break;
  default    : cout << "erreur!" << endl;
}
```

Interprétation incorrecte : L25 n'autorise pas d'avoir plusieurs instructions *quelconques* sur la même ligne

Exception : comme illustré dans l'exemple du switch, la sémantique de **break** autorise de l'ajouter à la suite d'une autre instruction

Code O *Conventions d'organisation*

O1 Le fichier source **mon_source.cc** est organisé selon l'ordre suivant:

O11 tous les **#include** dans l'ordre: **< xxx >**, suivi par, au sem2, **" yyy.h" , "mon_source.h"**

O11.1 éventuel `using namespace std ;`

O11.2 [sem2] éventuelle ouverture du ***unnamed namespace***[1] pour contenir les entités *utilisées seulement dans ce fichier*: **O12** à **O15**
En cas d'ouverture, le **unnamed namespace** est refermé après **O15** ; ré-ouverture éventuelle en **O16.3**

Le contenu d'un **namespace** doit être indenté (**L18**)

O12.1 tous les symboles/macros créés avec **#define**

O12.2 tous les symboles créés avec **enum** (éventuellement avec en plus **typedef**)

O12.3 si possible, les variables « constantes globales » avec **constexpr**

O13.1 déclarations anticipées de type de structure *(forward declaration)* et **typedef**

O13.2 description des détails de type structure

O13.3 les variables « constantes globales » utilisant un type structure, avec **constexpr**

O14 **déclaration** de toutes les fonctions appelées seulement dans ce fichier

O15 **[sem2] déclaration** des éventuelles variables *globales au fichier* seulement autorisé dans le ***unnamed namespace***

O16.1 **[sem1] définition** des fonctions; aucun ordre n'est imposé mais respectez celui que vous avez choisi en **O14**

O16.2.0 [sem2] éventuelle ouverture du ***namespace nommé*** définissant les fonctions **O16.2.1**
En cas d'ouverture, le ***namespace nommé*** est refermé après **O16.2.1**

O16.2.1 [sem2] définition des fonctions dont la liste est donnée dans l'interface du module ; respecter l'ordre du fichier .h

O16.3 [sem2] en cas de fonctions présentes en **O14**, ré-ouvrir le ***unnamed namespace*** pour la **définition** de ces fonctions

[1] Cette approche est recommandée par rapport à l'usage de **static** *en dehors de tout bloc* (**static** a longtemps été *deprecated* mais ne l'est plus depuis C++11)

O2 **[sem2]** Le fichier source **mon_source.h** est structuré de la façon suivante

Par convention le fichier en-tête (header) a une extension **.h** ;
il est généralement associé à un fichier d'implémentation de même nom : **mon_source.h** est associé à **mon_source.cc**

O21 On doit utiliser un symbole **100% en majuscule** reprenant le nom du fichier en-tête: **MON_SOURCE_H** pour **mon_source.h**

O22 Le fichier doit adopter la structure suivante (explication dans le cours sur le Préprocesseur)

```
#ifndef MON_SOURCE_H
#define MON_SOURCE_H
/* ici contenu habituel d'un fichier en-tête décrit en O3 (header file)*/
#endif
```

O3 **[sem2]** Le fichier source **mon_source.h** est organisé dans l'ordre suivant, à l'intérieur de la structure indiquée en **O22** :

Tout est indiqué comme « éventuels » car un fichier en-tête doit être limité au strict nécessaire pour réduire les dépendances

O31 éventuels **#include < xxx>, " yyy.h"**

O31.1 éventuel **`using namespace std ;`**

O31.2 s'il existe, ouverture du ***namespace nommé*** contenant la déclaration des entités exportées **O32.1** à **O35**
En cas d'ouverture, le ***namespace nommé*** est refermé après **O35**

Le contenu d'un **namespace** doit être indenté (**L18**)

O32.1 éventuels **#define** des symboles/macros exportés (on recommande plutôt d'utiliser des variables **constexpr**)

O32.2 éventuels **enum** et **typedef** des symboles exportés

O32.3 éventuels, si possible, variables « constantes globales » avec **constexpr** exportés

O33.1 éventuels déclarations anticipées de type de structure *(forward declaration)* et **typedef** des modèles de structure exportés

O33.2 éventuelles description complète des modèles de structure exportés (pour les types *concrets*, interdit pour les types *opaques*)

O33.3 éventuelles variables « constantes globales » utilisant un type structure, avec **constexpr**

O34 éventuels prototypes des fonctions exportées indépendantes d'une classe

O35 éventuelle déclaration d'une classe (voir aussi convention **D5**)

Code R *Restrictions*

R1 **Pas d'instruction `goto`** : cette instruction peut être remplacée par les instructions **`break`, `continue`, `return`** ou par l'utilisation et le test d'une variable booléenne (ayant seulement les valeurs Vrai ou Faux).

R2 **Pas de variable globale**: car une variable globale est accessible et modifiable partout dans votre code.
Le **sem2** autorisera seulement des variables **`limitées`** à un fichier ; cela n'est pas autorisé au **sem1**.

OK pour les constantes globales : Une variable globale est encouragée, si elle est déclarée comme une ***constante*** avec le mot clef **`constexpr`**.

Remarque : les symboles créés avec #define ne sont pas des variables. Avec les variables **constexpr** ces constantes permettent d'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 à la transmission de l'*adresse d'une variable que la fonction désire modifier ET si une référence ne peut convient pas*. 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 ou de travailler avec **const** et le passage par référence.

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 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 **Remplacer les *magic numbers*[2]** en utilisant des variables constantes avec **`constexpr`** ou des symboles avec **`#define`** ou **`enum`**
Un magic number est un ***paramètre potentiel du problème*** ; il pourrait changer lorsque le programme évolue.
Un autre cas est celui du choix d'une approximation d'un nombre tel que Pi
Par contre une valeur numérique apparaissant dans des équations ou formules invariantes n'est PAS un magic number.

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.

[2] Lire « constantes numériques non-nommées » et « usages acceptés comme constantes numériques non-nommées » sur https://fr.wikipedia.org/wiki/Nombre_magique_(programmation)