STANDARD DE PROGRAMMATION C++ (2001-02-15)

Pour certains d'entre vous, le standard qui suit couvre des éléments qui ne s'appliquent pas à ce moment-ci. Toutefois, lorsque vous aurez appris les concepts s'y rattachant, il faudra les intégrer à votre code.

1. GÉNÉRALITÉS

1.1 Identification des fichiers

- Les noms de fichiers doivent être significatifs. En particulier, le nom du projet devra identifier clairement le programme puisqu'il servira à donner le nom au programme exécutable. Pour simplifier l'affichage dans Visual C++ et ailleurs, on mettra des noms plutôt courts et sans blanc.

- Le nom d'un fichier source général doit porter le suffixe « .cpp ». Celui contenant la fonction `main` doit porter le nom de base du projet (*nomDuProjet.cpp*).

- Le nom d'un fichier d'en-tête doit porter le suffixe « .h ». Celui contenant la déclaration des variables et autres éléments globaux (types, constantes, etc.) doit porter le nom de base du projet.

  Si on n'a que quelques déclarations simples, on pourra les mettre dans le fichier d'en-tête du projet. Sinon on fera un ou des fichiers d'en-tête supplémentaires qu'on inclura au besoin dans celui-ci.

1.2 Structure d'un fichier source général (.cpp)

Un fichier source général doit contenir les éléments suivants, dans l'ordre (chacun de ces éléments est généralement séparé du précédent par une ligne blanche) :

- Un commentaire donnant le nom du fichier, la description du programme ou des fonctions qu'il contient, le nom des programmeurs et la date de création selon le format suivant :

```
/* fichier.cpp : Description et limites du programme ou fonctions
 *      ... (suite indentée de la description, au besoin)
 * Auteurs  : noms des auteurs
 * Création : date de création (et/ou autres dates importantes)
 */
```

  (voir 1.3) Le commentaire général du programme sera plutôt placé dans le fichier d'en-tête du projet. Par contre, pour les boîtes de dialogue, on commentera bien celles-ci dans leurs fichiers cpp.

- Les `#include` des fichiers d'en-tête standards (entre `< >`) nécessaires, le dernier étant `<stdcpp>`. On doit mettre tous les fichiers d'en-têtes clairement nécessaires et ne pas se fier au fait qu'ils sont peut-être inclus par les fichiers d'en-têtes personnels.

  On ajoutera les fichiers d'en-tête standards qui ne sont pas déjà inclus automatiquement (par exemple, `<vector>`) juste après le `#include "stdafx.h"`. Le fichier `<stdcpp>` sera inclus seulement par le fichier d'en-tête du *projet* (voir plus loin).

- Les `#include` des fichiers d'en-tête personnels (entre `" "`) nécessaires.

  Les `#include` des fichiers d'en-tête personnels seront ajoutés à la fin de la liste déjà générée.

- Le « `using namespace std;` » si nécessaire (mais on pourra laisser les qualifications `std::` dans les en-têtes de fonctions qu'on aura copiés dans le fichier d'en-tête en tant que prototype).

On ne met jamais de `using namespace std` global : au besoin on qualifiera les éléments avec `std::`, (`std::vector`, `std::map`, etc.). Toutefois, dans les fonctions sollicitant de nombreuses qualifications `std::`, on pourra y mettre localement un `using namespace std` ou, mieux, des déclarations `using` locales spécifiques (`using std::deque`, `using std::lower_bound`, etc.) pour éviter le préfixe.

- Si elles ne sont pas dans un fichier d'en-tête, la définition des constantes ainsi que la déclaration des types (voir la section suivante « Structure des fichiers d'en-tête »);

- La définition des variables globales (dans le fichier source principal *nomDuProjet.cpp*).

- La section `#ifdef _DEBUG` / `#endif`.

- Le `namespace` anonyme général et son contenu (au besoin). Ses accolades sont en colonne 1 et on met un commentaire à l'accolade finale, précédé d'une ligne blanche : `} // Fin du namespace anonyme`.

- La définition des variables membres statiques des classes (dans le fichier source de la classe).

- La définition des fonctions (sauf `main`) dans un ordre logique (normalement l'ordre de leur utilisation), chacune étant précédée par un commentaire de la forme suivante (voir exceptions plus loin) :

```
/*****
 * NomDeLaFonction : description de la fonction y compris, au besoin, celle
 *       des paramètres et de la valeur de renvoi...
 */
void NomDeLaFonction()
    ...
```

N.B. On ne met pas de ligne blanche entre le commentaire descriptif et le début de la fonction).

Si la fonction consulte et/ou modifie des variables globales, on notera ces informations ainsi :

```
/*****
 * NomDeLaFonction : description de la fonction y compris, au besoin, celle
 *       des paramètres et de la valeur de renvoi...
 *
 * g_varX : en consultation (et tout autre commentaire pertinent)
 * g_varY : en modification (et tout autre commentaire pertinent)
 * g_varZ : en consultation/modification (et tout autre commentaire pertinent)
 * ...
 */
void NomDeLaFonction()
    ...
```

- Pour les fichiers d'implantation des classes, on mettra des commentaires similaires :

```
/*****
 * ClNomDeLaClasse::NomDeLaFonction : description de la fonction y compris,
 *       au besoin, celle des paramètres et de la valeur de renvoi...
 */
void ClNomDeLaClasse::NomDeLaFonction()
    ...
```

- Si on implante plusieurs classes dans le même fichier source, on regroupera les fonctions par classe de façon à pouvoir, au besoin, facilement les transférer dans des fichiers séparés.

- Il n'est pas obligatoire de commenter les fonctions courtes (ordinaires ou membres) contenant 7 lignes de code ou moins (donc généralement 10 lignes si on compte les accolades et l'en-tête), en autant que le nom soit clairement significatif et descriptif. **Il n'est jamais interdit de commenter.**

Il n'est pas obligatoire de commenter les fonctions associées à des commandes (`On....`) et les `OnUpdate` associées, si le nom de la fonction est assez clair et que la fonction est assez courte.

- Finalement, le cas échéant, la définition de la fonction `main`, sera précédée par le commentaire suivant :

```
/*****
 * PROGRAMME PRINCIPAL
 */
int main()
    ...
```

1.3 Structure des fichiers d'en-tête (.h)

- Les fichiers d'en-tête doivent commencer par un commentaire semblable à celui des autres fichiers sources, décrivant qu'elles sont les déclarations et définitions qu'on y retrouve.

- Si le projet est décomposé en plusieurs fichiers, on aura généralement un fichier d'en-tête pour le projet qui portera son nom (*nomDuProjet.h).* Au besoin, on aura aussi un fichier d'en-tête pour chaque fichier source contenant des fonctions ou des classes spécifiques.

- Le commentaire général sur le programme (voir 1.2) sera placé dans le fichier d'en-tête du projet, ainsi :

```
// MonProjet.h : main header file for the MONPROJET application
//
//      Description et limites du programme
//      ...
//
// Auteurs  : noms des auteurs
// Création : date (et/ou autres dates importantes)
```

- Après le commentaire initial, on trouvera la protection contre les inclusions multiples sous la forme :

```
#if !defined(NOMFICHIER_H_INCLUS)
#define NOMFICHIER_H_INCLUS
...
#endif
```

  N.B. Le `#endif` est à la toute fin du fichier.

- On mettra ensuite, au besoin, chacun des éléments suivants, séparés du précédent par une ligne blanche :

  - la définition des constantes, commentées;
  - les types par énumération;
  - les structures et les classes, chacune suivie de leur fonction `inline` non membre associée (fonction `operator` principalement);
  - d'autres constantes des types tout juste déclarés;
  - la déclaration de variables globales (`extern`);
  - puis les prototypes de fonctions. Ces derniers doivent être complets, avec le nom des paramètres.

```
/* nomfichier.h : Description des éléments de ce fichier
 *       ...
 * Auteurs  : nom des auteurs
 * Création : date de création (et autres dates importantes)
 */

#if !defined(NOMFICHIER_H_INCLUS)
#define NOMFICHIER_H_INCLUS

#include <...>
#include "..."

const int NOM_DE_CONSTANTE= ...;  // Description de la constante
```

```
...

// ...
enum TypeEnum { ENUM1, ENUM2, ENUM3, ...};
...

struct TypeStruct  // ...
    {
    ...
    };

inline bool operator<(const TypeStruct& p_s1, const TypeStruct& p_s2)
    {
    ...
    }
...

/*****
 * ClNom : ...
 *
 *****/
class ClNom
    {
    ...
    };

inline std::ostream& operator<<(std::ostream& p_os, const ClNom& p_n)
    {
    ...
    }
...

const TypeStruct nomVar;  // ...
...

extern type g_nomVar;  // ...
...

void Fct(int p_premier, double& p_s_deuxieme);
...

#endif
```

- Les fichiers d'en-têtes doivent être complets : ils doivent contenir tous les `#include` nécessaires, mais pas plus[1]. Lorsque c'est possible, on utilisera des prédéclarations de types au lieu d'inclure nos fichiers d'en-tête personnels.

- On ne met jamais de `using` : on qualifie les éléments avec `std::` au besoin (même dans les fonctions *inline*).

- Seuls les éléments qui ne sont pas définis ailleurs ont besoin d'être commentés dans le fichier d'en-tête.

1.4 Impression des programmes (si exigée)

L'impression des fichiers sources avec *Listeur* doit être faite de façon à contrôler les sauts de page. Ils ne doivent pas entraver la lecture de la logique. Idéalement un bloc d'instructions doit apparaître en entier sur la même page. De plus, vous devrez faire apparaître le nom du fichier, la date, le nom des auteurs et la pagination.

Ne pas faire imprimer les fichiers créés par *AppWizard* mais non modifiés par la suite (par exemple, le fichier *nomDuProjet.cpp*). De plus, on utilisera l'option de *Listeur* pour ne pas imprimer les fonctions non modifiées (c'est-à-dire dont les `{}` sont en colonne 1, n'oubliez de les déplacer dans toutes les fonctions modifiées).

[1] On devrait pouvoir les compiler sans erreur en leur mettant le suffixe cpp, mais tout retrait d'un include devrait alors donner des erreurs.

1.5 Contenu de la disquette (si exigée)

Dans le dossier principal, on retrouvera tous les fichiers sources (généraux et en-têtes), le programme exécutable et, le cas échéant, les fichiers de projet (*.dsp* et *.dsw*, etc.). Dans le dossier `\Copie`, on mettra une copie du dossier principal (sauf le programme exécutable, si l'espace ne le permet pas). Vous assurez que la disquette est exempte de virus ou vous serez pénalisés si c'est un virus détectable dans les laboratoires.

Vous ajouterez bien sûr le dossier des ressources dans le dossier principal et `\Copie`.

1.6 Remise des travaux (si applicable)

Remettre le tout dans une « pochette » bien identifiée et qui retient bien la disquette (qui doit aussi être identifiée) sans la rendre impropre à l'utilisation !

2. ÉCRITURE DU LANGAGE

2.1 Identificateurs

Règles à respecter pour tous les identificateurs

- On programme en français. (L'exception étant les préfixes de fonction provenant des MFC comme **`On`**`XYZ` et **`OnUpdate`**`XYZ`, ainsi que les fonctions directement reliées à des messages Windows).

- L'identificateur peut être composé de lettres et de chiffres; on utilisera le trait de soulignement seulement dans les constantes, les valeurs des types par énumération, les *define*, et dans les préfixes des variables globales, des variables membres et des paramètres.

- L'identificateur doit décrire ce qu'il représente; il doit être significatif, précis et réaliste. Il faut éviter les noms vagues comme `x2, tempo` et leurs variantes. Des noms comme `Calculer, cpt, MAX, m_ctrl` ne sont pas acceptables mais ils peuvent parfois servir de préfixe dans des identificateurs composés de plusieurs mots. Cependant, même comme préfixe, il faut bannir les mots de sens neutres comme `Traiter, Gerer`, etc.

- Les identificateurs doivent correspondre vraiment à l'objet identifié. Ainsi on ne donnera pas le nom `nbPair` à une variable qui contiendra un nombre quelconque qu'on testerait pour savoir s'il est pair, pas plus que `moyenneVentes` à une variable servant à totaliser des nombres avant le calcul de la moyenne (c'est un total).

- Les abréviations sont acceptables si elles sont claires et communes. Les abréviations permettent de créer des identificateurs plus courts, ce qui est commode, en autant qu'elles ne leur font pas perdre leur clarté. Il faut s'efforcer d'utiliser les abréviations dans leur sens exact, comme `nb` pour nombre et `no` pour numéro. Par ailleurs, les symboles utilisés parfois en abréviation par les systèmes de mesure sont généralement illisibles dans des identificateurs. On n'écrira donc pas `PO_PAR_M`, ni `MIN_PAR_H`, mais plutôt `POUCES_PAR_METRE` et `MINUTES_PAR_HEURE`. (on n'inventera pas d'abréviation non plus, donc on ne fait pas `POUC_PAR_METR`)

- L'orthographe doit être correct : attention aux « s » et aux accents. L'orthographe des mots compris dans un identificateur peut parfois permettre de sauver des articles. Au lieu de `CalculerLesTaxes`, on peut écrire `CalculerTaxes`, qui n'est pas comme `CalculerTaxe` ! Le problème est aussi de se rappeler l'orthographe des identificateurs qu'on n'a pas écrit correctement. Par exemple, si on écrit `nbArticleVendu`**`s`**`,` il est fort probable qu'on soit tenté d'écrire `nbArticle`**`s`**`Vendu`**`s`**, au moins une fois...

Identificateurs des constantes (`const` globales, `enum`, ou membres `static const`)

- L'identificateur doit être écrit en majuscules; on sépare les mots par le trait de soulignement (_).

- On mettra simplement le préfixe `LARGEUR_` pour qualifier les constantes donnant les longueurs maximales de certains éléments (ou les largeurs à l'affichage). On mettra `LARGEUR_MIN_` pour les minimums, si nécessaire.

- Dans les types par énumération portant un nom : quand les identificateurs risquent d'entrer en conflit avec des valeurs d'autres types par énumération ou des constantes, souvent parce qu'ils sont courts ou un peu vagues, on mettra un préfixe rappelant le nom du type (ou une abréviation). Par exemple, `PAIEMENT_CHEQUE` pour le `TypePaiement`, `CA_MODERNE` pour `TypeCourantArtistique`, etc.

Identificateurs des variables

- L'identificateur doit être écrit en minuscules, sauf s'il est composé de plusieurs mots : dans ce cas, on met une majuscule à la première lettre à partir du deuxième mot. Ne jamais y mettre le mot`var` (ni ses variantes).

- Les identificateurs `i, j,` etc. peuvent être utilisés seulement pour les variables entières déclarées dans les boucles `for`. On les évitera si elles ne servent pas d'indices de vecteurs ou tableaux.

- L'identificateur `it` peut être utilisé seulement pour les variables itérateurs déclarées dans les boucles `for`.

- Bien qu'on désire des noms significatifs et précis, on pourra utiliser un nom de variables court et générique s'il est utilisé *très* localement et qu'il est alors suffisamment significatif dans son contexte d'utilisation. C'est ce qui explique l'acceptation de `i`, `j` et `it` des règles précédentes. Par exemple, `car` (pour un caractère quelconque), `p` (pour un pointeur sur un élément d'un vecteur) et `pos` (pour une position à conserver dans un vecteur) seraient parfois acceptables. Par contre, `car1`, `car2` ou `p1`, `p2`, etc., sont inacceptables.

- Les identificateurs des variables booléennes doivent être « positifs » et être utilisés adéquatement : on préférera `while (continuer)` ou `while (!sortir)` à `while (pasFini)` (et jamais `while (sortir)` !!!)

- Lorsque les identificateurs semblent au pluriel, parce qu'ils sont invariables ou se terminent par *s*, on ajoutera le mot `un` (ou `une`) en préfixe, sauf pour les vecteurs (voir le point suivant). Par exemple, `unCours`.[2] On n'ajoute pas inutilement ce préfixe : par exemple, `client` et non pas `unClient`.

- Pour les vecteurs (de base ou `vector`), on mettra un identificateur clairement au pluriel. Si ce n'est pas possible ou pas assez clair, on ajoutera le préfixe `tab`. Par exemple, `clients` identifie clairement un vecteur, mais on mettra `tabCours`, `tabNoClients` (`noClients` ne serait pas clair, on pourrait mettre `numerosClients` cependant). Par contre, on ne mettra jamais le mot `tab`, ni le pluriel, pour identifier les vecteurs de `char` servant à des chaînes de caractères (terminées par des NUL).

- Les pointeurs commencent par un `p` suivi du nom de l'élément pointé qui commencera alors par une majuscule.

- Les identificateurs de paramètres suivent les mêmes règles, mais comportent un préfixe `p_, p_s ou p_es_` selon le type de paramètres. Il n'est pas nécessaire de modifier les paramètres des fonctions courtes générées par les *Wizard.*

---

[2] On pourrait aussi mettre ce préfixe lorsqu'il y a un conflit avec un mot réservé ou déjà défini par le langage ou les bibliothèques standards, mais on préférera choisir un autre identificateur (plus précis ou un synonyme, etc.). Par exemple, pour le `cout` d'un billet, qui serait en conflit avec `cout` pour l'écriture, on choisira `prix`, `coutParBillet`, etc., au lieu de `unCout`.

- Les identificateurs des variables globales suivent les mêmes règles mais comportent le préfixe `g_`.

- Les identificateurs des variables membres des classes suivent les mêmes règles mais comportent le préfixe `m_`.

- Les identificateurs des paramètres de valeur des `template` comportent le préfixe `t_`.

- Le premier caractère après les préfixes (`p_, g_,` etc.) doit être une lettre minuscule (sauf pour les paramètres de type pointeur sur fonction ou fonction objet, qui commencent pas une majuscule comme les fonctions).

Identificateurs de types

- Les `enum` et les `struct` sont nommés `TypeXyz.`

- Les classes sont nommés `ClXyz`, sauf les classes de style MFC (dérivées publiquement) qui commencent simplement par un `C` (donc pas de `l`).

- Les `typedef` sont normalement nommés `TypeXyz`, mais on les nomme `ClXyz` s'ils réfèrent à une classe ou à quelque chose qui pourrait être une classe. Par exemple :

```
typedef string TypeAdresse[NB_LIGNES_ADRESSE];
typedef vector<TypeClient>::iterator ClIterClient;
```

- Les identificateurs des types paramètres des `template` peuvent être nommés `T`, `U`, `V,` etc.

- Les identificateurs des éléments des `struct` et des classes (champs et données membres) ne doivent pas répéter le nom de la `struct` ou de la classe. Par exemple :

```
struct TypeClient
    {
    int numero;         // Et non pas noClient
    string nom;         // Et non pas nomClient
    string nomDeLaMere; // Ici la précision est nécessaire
    ...
```

Identificateurs des fonctions

- L'identificateur doit normalement être composé de plusieurs mots (ou abréviations); la première lettre de chaque mot doit être une majuscule. Le nom doit être aussi long que nécessaire.

- Lorsque la fonction ne renvoie pas de résultat, le premier mot **doit** être un verbe « fort » à l'infinitif (voir plus loin pour les exceptions dans les fonctions d'accès des classes). Dans le cas contraire, le nom de la fonction doit représenter la valeur renvoyée. Dans le cas des fonctions renvoyant un booléen, le nom devrait pouvoir facilement s'enchaîner dans les `if`, `while`, etc.

  Exemples :

```
AfficherCours(g_tabCours[i]);
if (EstNbPremier(nombre))
cout << NomCategorie(athlete.categorie) << ...
AfficherRapportInscriptions();
```

- Le nom des fonctions membres ne doit pas répéter le nom de la classe : `dateAchat.Afficher()`, si `ClDate dateAchat;,` et non pas `dateAchat.AfficherDate()`.

- Pour les fonctions d'accès des classes, le nom de la fonction de modification (*setter*) sera le même que celui de la fonction de lecture (*getter*), mais avec paramètre(s) bien sûr. Par exemple, on pourrait avoir à écrire `date.Annee(date.Annee()+1);`. On ne mettra donc généralement pas de verbe. Par contre, si on fournit simplement une fonction de modification générale, sans fonction de lecture correspondante, on utilisera le

verbe `Changer` en préfixe avec, au besoin, le nom de l'information à modifier (`date.Changer(1,1,2000);, rapport.ChangerInterligne(2);`).

- On n'utilisera le verbe `Obtenir` que dans les cas où il y une saisie des données fournies par un usager. Par défaut, ces fonctions doivent faire les validations : on ne mettra pas le mot `Valide` dans leur identificateur (surtout pas `ValiderXXX` au lieu de `ObtenirXXX`).

  Exemples :
  ```
  ObtenirClient(client);
  int noCours= ObtenirNoCours();
  ```

  À défaut de mieux, on peut utiliser `Obtenir` comme préfixe pour des fonctions du document qui fournissent simplement des données aux vues.

- Les adjectifs `Valide` ou `Existe,` placés après un nom d'élément à valider, nomment bien les fonctions qui ne font que renvoyer un booléen indiquant si un paramètre est « correct », sans lecture ni d'écriture. (Par contre, `ValiderNoCours(...)` est trop vague et n'est pas acceptable.)

  Exemple :
  ```
  if (NoCoursValide(noCours) && ! CoursExiste(noCours))
      ...  // On peut ajouter un cours portant le numéro noCours
  ```

2.2 Présentation des déclarations/définitions

Définitions des constantes

- On essaiera de regrouper les constantes par thème, en séparant les groupes par des lignes blanches. On met le `const` avant le type.

- Les constantes sont toujours commentées, idéalement à droite de la déclaration. Si le commentaire ne peut pas être facilement mis à droite, l'inscrire avant la déclaration. Le commentaire doit indiquer l'utilité de la constante ou ce qu'elle représente (surtout pas sa valeur !).

Définitions des variables

- On présentera une seule variable par ligne avec répétition des types.

- Toute variable peut être commentée, à droite ou au dessus (comme les constantes). On commentera *obligatoirement* les variables globales (à la déclaration **et** à la définition) et celles ne recevant pas d'initialisation à leur déclaration (par contre, on ne met pas d'initialisation inutile). Le besoin de commenter les variables avec initialisation dépendra de la qualité de leur nom et du contexte de leur utilisation...

Déclarations des structures

- On présentera un seul champ par ligne avec répétition des types.

- On commentera, s'il y a des précisions utiles à apporter, la structure globale et les champs. Par exemple :

```
struct TypeRect  // Représente un rectangle toujours placé à l'horizontale
    {
    ClCoord hautGauche; // Coordonnées du coin supérieur gauche
    int hauteur;
    int largeur;
    };
```

- En particulier, on commentera toujours les structures pour les enregistrements de fichiers.

Déclarations des types par énumération

- Les valeurs des types par énumération seront généralement présentées sous la forme :

```
enum TypeEnum { VALEUR1, VALEUR2, ...,                    N.B On met un blanc après {
                VALEURX, VALEURY, ..., DERNIERE_VALEUR };       et un avant }
```

- Lorsque les noms sont très longs, que les valeurs d'initialisation sont spécifiées ou qu'on veut commenter chaque valeur, on pourra mettre **toutes** les valeurs sur des lignes séparées, avec indentation comme pour les structures :

```
enum TypeEnum
    {
    PREMIERE_VALEUR= ...,
    DEUXIEME_VALEUR= ...,
    ...
    DERNIERE_VALEUR= ...
    };
```

- Pour les types par énumération, il ne sera généralement pas nécessaire de commenter, mais ce n'est pas interdit.

Déclarations des classes (non dérivées des MFC)

- La déclaration est semblable à celle des `struct`, mais est précédée d'un commentaire formant un C et suivie des définitions des fonctions *inline* non définies à l'intérieur de la classe. Voici l'ordre normal des déclarations :

```
/*****
 * ClNomClasse : description
 *       ...
 *****/
class ClNomClasse         // : public/private ClClasseDeBase (au besoin)
    : liste d'initialisation (au besoin)
    {
    public :
    enum sans nom (pour définir des constantes numériques)

    déclarations friend nécessaires pour les classes imbriquées

    Autres types imbriqués (enum, struct, class, typedef).

    Variables ou constantes publiques (const static...)

    Constructeurs
    Destructeur
    Fonctions membres non virtuelles
    Nouvelles fonctions membres virtuelles pures
    Nouvelles fonctions membres virtuelles non pures

    /* virtual */ Fonctions virtuelles surchargées
    Fonctions membres statiques

    déclarations friend générales

    protected :
    Même ordre que dans la section private...

    private :
    Même ordre que dans la section public, sauf qu'on terminera
    par les données membres...
    };

fonction inline membre non définie à l'intérieur de la classe
fonction inline non membre
```

- Pour améliorer la lisibilité, on laissera une ligne blanche entre chaque section. De plus, lorsque c'est possible, à l'intérieur d'une section, on regroupera les déclarations selon un ordre logique (fonctions d'accès, fonctions de modifications, etc.).

- On ne commentera pas les prototypes de fonctions. Les commentaires apparaîtront seulement lorsqu'on définira la fonction dans le fichier d'implantation. Les fonctions `inline` devraient être assez claires pour ne pas avoir besoin d'être commentées, mais ce n'est pas interdit, en particulier celles définies à l'extérieur de la classe.

- Les données membres doivent être commentées de façon générale (dans le commentaire initial) ou individuellement.

- Il n'y aura jamais de données membres publiques. Il faudra donc modifier la position des déclarations ajoutées par *ClassWizard* pour les variables associées aux contrôles.

Il n'est pas nécessaire de modifier l'ordre, ni de commenter les éléments ajoutés automatiquement (par *Add Member Functions*, etc.) pour les classes dérivées des MFC, mais il faut toujours mettre les données privées.

Définitions des fonctions `inline`

- Les fonctions `inline` définies à l'intérieur des classes sont obligatoirement écrites sur la ligne de la déclaration, donc très courtes et simples (on laisse un blanc après `{` et avant `}`). Si ce n'est pas le cas, on peut quand même faire des fonctions `inline`, si elles sont tout de même assez courtes, mais on les définira à l'extérieur de la classe. On ne mettra jamais de fonctions virtuelles `inline` (implicite ou explicite) sauf si elles sont vides (en particulier, on pourra mettre ainsi les destructeurs virtuels des classes de base).

Les templates

- Dans les `template`, on fait ce qu'on peut pour respecter les règles déjà indiquées...

2.3 Présentation générale

Longueur des lignes

- Il est important que les lignes soient complètement visibles à l'écran et à l'impression. Dans Visual C++, on se limitera donc à 95 colonnes, visibles en mode 800x600 et facilement imprimables.

Lignes blanches

- Des lignes blanches seront insérées pour améliorer la lisibilité du programme. On en mettra généralement une, avant (et après) chaque fonction, instruction de contrôle ou groupe d'instructions reliées à une étape logique de l'algorithme. Si un commentaire précède une fonction, une instruction de contrôle ou un groupe d'instructions, on met la ligne blanche avant le commentaire. On ne met jamais de ligne blanche avant une ligne contenant seulement la fin d'un bloc (`}`)[3], ni après une ligne contenant le début d'un bloc (`{`).

Commentaires

- Les commentaires superflus doivent être évités. En particulier, on évitera la simple répétition des noms de variables ou de constantes ainsi que les explications imprécises, incorrectes ou redondantes. Il faut donc que le commentaire apporte une information pertinente et utile. Les commentaires doivent permettre de répondre aux questions qu'on se pose et non pas soulever de nouvelles questions. La plupart du temps, ils sont nécessaires.

---

[3] Pour les `namespace`, un commentaire suivra l'accolade, donc on peut (et on devra) laisser une ligne blanche.

- Les commentaires décrivant les instructions doivent généralement être alignés avec le niveau d'imbrication. Un commentaire simple peut aussi apparaître à droite des instructions s'il ne dépasse pas la colonne maximum.

- On laissera un blanc après les symboles `//` ou `/*`, et avant le `*/`.

- Les commentaires débuteront par une majuscule : si un commentaire s'étend sur plusieurs lignes et qu'il est placé à droite des instructions, on indentera d'un blanc à partir de la deuxième ligne.

```
// La prochaine ligne sera la suite de ce commentaire, mais n'est placée à droite
// des instructions donc on n'a pas mis de blanc supplémentaire initial.
int justePourCommenter= 0;  // Par contre, ici on aura la suite d'un commentaire
                            //  placé à droite, donc on a ajouté un blanc (mais
                            //  on n'en met pas plus d'un).
```

- Les commentaires à droite des instructions et des déclarations seront alignés (autant que possible).

- On évitera les abréviations dans les commentaires.

2.4 Choix des instructions

- Les boucles `for`*`ever`*`-break` ne seront utilisées que si `while` et `do-while` ne conviennent pas, donc quand la sortie de la boucle ne se trouve pas aux extrémités. On ne met qu'**une** seule sortie avec `if-break`, et elle doit être au niveau principal d'imbrication du `for`.

- Les boucles à compteur seront toujours traduites par des `for`. Si la variable de contrôle doit être testée en dehors de la boucle, on la déclare juste avant, en l'initialisant, et on laisse la première partie du `for` vide, en laissant un blanc.

```
int indiceClient= 0;

for ( ; indiceClient < soldes.size() && soldes[indiceClient] < 0.0; ++indiceClient)
```

- La variable de contrôle des boucles `for` ne doit pas être modifiée à la fois par la partie incrémentation et dans le corps de la boucle. S'il faut parfois la modifier dans le corps, on y met toutes les modifications.

- Les seules variables manipulées dans les parties initialisation et incrémentation d'un `for` seront celles qu'on retrouve dans la condition.

- On passera les paramètres d'entrée par « référence constante » lorsque c'est possible et qu'il s'agit de paramètres de types n'ayant pas d'opération de copie rapide. En particulier, on passera la plupart des variables de type structure ou classe (y compris les `std::string`) par référence constante, mais pas les itérateurs (🪟 ni les `CString`). Inversement, on évitera de faire des fonctions qui renvoient des valeurs de types n'ayant pas de copie rapide (on utilisera plutôt un paramètre de sortie).

2.5 Présentation des instructions

- Si possible, on déclarera les variables juste avant leur première utilisation (*j.i.t : just in time*) en les initialisant directement. Selon la logique de l'algorithme, il faut parfois déclarer la variable avant son initialisation dans un `if` (mais il faut d'abord penser à `?:`), un `switch` ou une boucle, pour pouvoir l'utiliser après.

- Dans les tests d'expressions longues ou complexes donnant des résultats simples, on écrira la partie simple avant l'opérateur relationnel et la partie complexe après lui (souvent la constante avant et le calcul après).

  🪟 En particulier : `if (IDOK == AfxMessageBox(...))`.

- Les tests d'intervalles seront présentés dans l'ordre mathématique *`min <= valeur && valeur <= max`* (avec les opérateurs relationnels adéquats selon le cas). Pour tester les valeurs hors intervalle, on met la même notation qu'on inverse avec l'opérateur `!` (voir l'exemple ci-après).

- Une instruction trop longue pour tenir sur une ligne doit être coupée à un endroit facilitant sa compréhension. La partie coupée est ramenée au niveau d'indentation ou plus loin si c'est plus logique. En particulier, si on coupe une condition, on aligne sa suite sur la condition plutôt que sur l'instruction. On coupera idéalement juste avant un opérateur.

```
while (MIN_AUTOMOBILES_PAR_HEURE <= cptAutomobilesObservees
       && cptAutomobilesObservees <= MAX_AUTOMOBILES_PAR_HEURE)

while ( ! (MIN_AUTOMOBILES_PAR_HEURE <= cptAutomobilesObservees
           && cptAutomobilesObservees <= MAX_AUTOMOBILES_PAR_HEURE))
```

- Les tests de classification générale des caractères seront toujours faits avec les fonctions de `<cctype>` (`isdigit`, `isalpha`, etc.) quand c'est possible.

- Les instructions et les blocs d'instructions (`{...}`) seront décalés d'une tabulation de 4 colonnes. Dans les blocs d'instructions, les instructions doivent être alignées sous l'accolade d'ouverture. Par exception, les accolades des `namespace` ne sont pas indentées afin de pouvoir y mettre les éléments avec l'indentation normale.

  On laisse aussi en colonne 1 les accolades des éléments générés par *AppWizard* et qu'on n'a pas modifiées (on déplace les accolades lorsqu'on les modifie).

- On ne met pas de blanc après les parenthèses ouvrantes ni avant les parenthèses fermantes, sauf pour les `!` dans certaines conditions (voir plus loin).

- Les parenthèses entourant la condition des structures de contrôle (`while, if, for`) sont précédées d'un blanc, sauf dans le cas du *forever* qu'on écrit directement `for(;;)`.

- Aucun blanc ne doit être inséré entre le nom d'une fonction et la parenthèse encadrant ses paramètres, autant dans les déclarations qu'aux appels.

- Le `&` notant les références (dans les paramètres) sera collé sur le type et suivi d'un blanc. Même principe pour les pointeurs (`*`). On ne modifiera pas nécessairement les déclarations ajoutées par les outils de Visual C++ qui laissent un blanc avant et après le `&` (et le `*`).

- On ne laisse aucun blanc autour du point et de l'opérateur d'indirection `->`.

- On laisse toujours un blanc après les points-virgules et les virgules, mais pas avant (deux exceptions : `for(;;)` et `for ( ; ... ; ...)`).

- Des blancs doivent être insérés autour des opérateurs relationnels et logiques. S'ils aident à la lisibilité, on peut aussi en mettre autour des opérateurs arithmétiques. Les blancs sont donc toujours en symétrie autour de ces opérateurs (un de chaque côté, ou aucun des deux côtés).

```
if (NOMBRE_MIN_COURS <= nbCours && nbCours <= NOMBRE_MAX_COURS)
```

- Les opérateurs d'affectation (**`=`**, **`+=`**, etc.) suivent directement la variable qui reçoit la valeur, mais sont suivis d'un blanc (asymétrie).

```
double salaire= tauxHoraire * nbHeuresTravaillees;

totalCommissions+= commission;
```

- On ne mettra que les parenthèses qui améliorent la lisibilité, ou qui sont nécessaires pour la syntaxe ou l'ordre d'évaluation. On mettra toujours les parenthèses lors de l'utilisation des opérateurs de manipulations de bits et de l'opérateur `sizeof`.

```
double salaireTotal= salaireDeBase + totalVentes*(1+TAUX_COMMISSION);

bool decouvert= (valeur == valeurCherchee);   N.B. Facultatives ici, selon le goût...

return valeur == valeurCherchee;              N.B. On évite de faire return (...);

mois= (date >> DECAL_MOIS) & MASQUE_MOIS;     N.B. Obligatoire par standard seulement

Write(&enr, sizeof(enr));  N.B. Ou Write(&enr, sizeof(TypeClient));
```

- On entoure de parenthèses, la condition dans les opérateurs conditionnels (`?:`).

- Toujours mettre des parenthèses lors de l'insertion d'expressions avec `<<`.

```
cout << "Prix total : " << (prix+taxe) << " $.\n";

cout << "Il y a " << nbLivres << " livre" << ((nbLivres > 0) ? "s" : "")
     << " en inventaire.\n";
```

- Sauf dans le cas des sauts de lignes multiples, chaque saut de ligne à l'écriture sera montré par un changement de ligne dans le programme.

```
cout << "Prix avant taxe : " << prix << " $.\n"
     << "Taxe à payer    : " << taxe << " $.\n\n"
     << "Prix total : " << (prix+taxe) << " $.\n";
```

- On utilisera les opérateurs `++` et `--` de façon préfixée, quand on n'utilise pas la valeur de l'expression.

- Pour faciliter la lecture de certaines expressions longues ou complexes, on laissera un blanc de part et d'autre de l'opérateur `!`. Dans le cas d'expressions simples, les blancs ne seront pas obligatoires.

```
if ( ! (expression complexe...))
if (!variableBool)                                N.B. Ou  if ( ! variableBool)
if (prix >= PRIX_MIN_TAXABLE && !achatHorsTaxe)   N.B. Ou  if (... && ! achatHorsTaxe)
```

2.6 Écriture des constantes

On utilise le préfixe `0x` (`x` minuscule) mais les suffixes `F` et `L` (majuscules). On écrit le bon nombre de chiffres hexadécimaux selon le type désiré (4 pour les `short`, 8 pour les `long` en ajoutant alors le `L`). Par exemple `0x1234`, `0x000012EFL`, `1.23F`, etc.

3. Structures de contrôle

Bloc d'instructions

- On fait un bloc même lorsqu'on a une seule instruction, si celle-ci occupe plus d'une ligne (ou est précédée d'un commentaire).

```
{                               {
instruction1;                   instruction qui
instruction2;                   prendrait plus d'une ligne;
...                             }
}
```

- Exceptionnellement, on ne mettra des blocs dans les `case` des `switch` que si l'on doit y déclarer des variables locales. On mettra alors le `break` en dehors du bloc (car théoriquement le `break` sort d'un bloc...). Ces cas

devraient être très rares, car on évite de mettre beaucoup d'opérations dans les `switch` (il vaut mieux appeler des fonctions).

- Pour les choix multiples utilisant des `if-else-if`, on ne met pas de bloc (voir plus loin).

- Si le bloc d'une structure de contrôle occupe plus d'un écran (35 lignes environ), un commentaire suivra l'accolade de fermeture et indiquera l'instruction de contrôle complète qui se termine (on indiquera aussi s'il s'agit d'un `else`). Cependant il est très souvent préférable de modulariser afin d'éviter cette possibilité.

```
        ...
        } // Fin du else du if (decouvert)
    ...
    } // Fin du while (!sortir)
```

`if` simple

```
if (condition) instruction;   N.B. Permis seulement si une seule instruction très simple
```

```
if (condition)                N.B. Dans les cas « normaux »
    instructionOuBloc1;           (même s'il n'y a pas de else)
else
    instructionOuBloc2;
```

- On essaie d'écrire les blocs le plus long dans le `else`, particulièrement si un des blocs est beaucoup plus long que l'autre; s'ils sont de tailles semblables on mettra la condition la plus logique ou la plus simple.

- On ne met pas de `else` quand le bloc vrai se termine par un `return`. Par contre, pour les `return` dépendant d'une condition, on fera un `return` *`condition`*`;` (si le résultat est booléen) ou on emploiera l'opérateur conditionnel (`return (`*`condition`*`) ? ... : ...;`) dans les autres cas (donc pas de `if`).

- On ne compare jamais une variable booléenne avec les constantes `true` ou `false`. On écrira donc `if (cond)` au lieu de `if (cond == true)` et `if (!cond)` au lieu de `if (cond == false)`.

`if` utilisés pour alternatives multiples (quand `switch` n'est pas utilisable)

```
if (condition1)
    instructionOuBloc1;
else
if (condition2)
    instructionOuBloc2;
...
else
    instructionOuBloc3;
```

```
if (condition1) instruction1;
else
if (condition2) instruction2;
...
else
    instruction3;
```

- Dans les cas où le dernier `else` ne pourrait servir qu'à tester un cas connu, qu'on pourrait mettre dans un `if`, on fera ce `if` et le dernier `else` sera un `assert(false)`.

- Par souci d'efficacité, il faut mettre les cas les plus probables dans les premiers `if`.

`switch`

- Dans les cas où le `default` ne pourrait servir qu'à tester un cas connu, qu'on pourrait mettre dans un `case`, on fera ce `case` et le `default` sera un `assert(false)`. (Donc, on s'efforcera, quand c'est possible, de mettre un `assert(false)` dans `default`, en listant tous les cas connus dans des `case` individuels).

- On peut avoir des `case` « vide » sans `break` (ni `return`) normalement, mais on commentera clairement les cas (extrêmement rare) où il y a du code sans `break` ni `return`.

- Quand au moins un des `case` nécessite plus d'une instruction, on présente tous les `case` en mettant les instructions en dessous (décalées) plutôt qu'à droite.

```
switch (expression)
    {
    case valeur1 : instr1; break;
    case valeur2 : instr2; break;
    ...
    default :      instr3; break;
    }
```

```
switch (expression)
    {
    case valeur1 : return expression1;
    case valeur2 : return expression2;
    ...
    default :      return expression3;
    }
```

```
switch (expression)
    {
    case valeur1 :
    case valeur2 :
        instruction;
        instruction;
        break;

    case valeur3 :
        {            N.B. Accolades si le bloc contient des déclarations
        instruction;
        ...
        }
        break;

    case valeur4 :
        instruction;
        // On poursuit avec les instructions prévues pour la valeur5
    case valeur5 :
        instruction;
        ...
        break;
    ...

    default :
        instruction;   N.B. Peut être un assert(false)...
        ...
        break;
    }
```

Structures itératives

- Quand les éléments d'un `for` n'entre pas sur une ligne, on les écrit chacun sur une ligne séparée.

```
while (condition)
    instructionOuBloc;
```

```
do
    instructionOuBloc;
while (condition);
```

```
for (int var= début; condition; ++var)
    instructionOuBloc;
```

```
int var= début;

for ( ; condition; ++var)
    instructionOuBloc;
```

```
for (int var= début;
     condition;
     ++var)
    instructionOuBloc;
```

```
for(;;)
    {
    instruction;
    ...
    /***/
    if (condition) break;
    /***/
    instruction;
    ...
   }
```

```
for(;;)
    {
    instruction1;
    ...

    /***/
    if (condition) break;
    /***/

    instruction2;
    ...
    }
```

- Pour l'instruction nulle, on utilisera un bloc vide (`{}`) et on commentera au besoin.