STANDARD DE PROGRAMMATION C++ (version pour 007, 2006-10-23)

1. GÉNÉRALITÉS

1.1 Identification des fichiers

Les noms de fichiers doivent être significatifs. Cependant, pour simplifier l'affichage dans Visual Studio et éviter un certain nombre d'autres problèmes, on mettra des noms plutôt courts, sans espace et sans accent. Le nom d'un fichier source général doit porter le suffixe *.cpp*.

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 et des fonctions qu'il contient, le nom des programmeurs et la date de création (ou autres) 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)
 */
```

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

- Le `using namespace std;`.

- La définition des constantes.

- La déclaration des types `enum` et `struct`.

- Des constantes des types `enum` et `struct` tout juste déclarés, si nécessaires.

- La définition des fonctions de lecture et écriture regroupées par type.

- La définition des autres 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 au prochain point) :

```
/*****
 * 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.

- Il n'est pas obligatoire de commenter les fonctions courtes 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 parfaitement significatif et descriptif. **Malgré cette règle, il n'est jamais interdit de commenter si on peut ajouter des informations.**

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

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

1.3 Impression des programmes (si exigée)

L'impression des fichiers sources avec *Listeur* doit être faite après la vérification des sauts de page seulement. Ils ne doivent pas entraver la lecture de la logique. Il est possible d'ajouter des sauts de page avec `////`. Idéalement un bloc d'instructions doit apparaître en entier sur la même page. De plus, vous devez faire imprimer l'en-tête et le pied de page avec le nom du fichier, la date et l'heure, le nom des auteurs et la pagination.

1.4 Contenu de la disquette ou CD (si exigée)

Dans le dossier principal, on retrouvera une copie des fichiers, au minimum le fichier source et le programme exécutable. Dans un dossier nommé `\Copie`, on mettra une copie des mêmes éléments, sauf le programme exécutable, si l'espace ne le permet pas. Vous devez vous assurer que la disquette ou CD est exempte de virus ou vous serez pénalisés si c'est un virus détectable dans les laboratoires.

1.5 Remise des travaux ou envoi par courriel

Veuillez bien identifier les travaux, disquette ou CD. Si vous remettez une disquette ou un CD avec une copie imprimée, remettez le tout dans une « pochette » bien identifiée qui retient bien la disquette ou le CD. Si vous remettez un programme par courriel, ne pas envoyer autre chose que le fichier source et le programme exécutable : ne jamais envoyer tout le dossier, même compressé !

2. ÉCRITURE DU LANGAGE

2.1 Identificateurs

Règles à respecter pour tous les identificateurs

- On programme en français. On met les signes diacritiques (accents, cédille, etc.) dans les identificateurs et dans les commentaires. On évite de faire des identificateurs simplement différenciés par ces signes.

- 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 et dans les préfixes 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` 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, Gérer`, 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_MÈTRE` et `MINUTES_PAR_HEURE` (on n'inventera pas d'abréviation non plus, donc on ne fait pas `POUC_PAR_MÈTR`).

- L'orthographe doit être correcte : 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 écrits incorrectement. Par exemple, si on écrit `nbArticleVendus,` il est fort probable qu'on soit tenté d'écrire `nbArticle`**`s`**`Vendus`, au moins une fois...

Identificateurs des valeurs constantes (`const` et `enum`)

- 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 : quand les identificateurs ne sont pas clairs hors contexte ou 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 quelconque mais simple). Par exemple, `PAIEMENT_CHÈQUE` 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 de contrôle entières déclarées dans les boucles `for.` Si elles ne servent pas simplement à contrôler les itérations ou comme indices de vecteurs, donc si la variable représente quelque chose de précis qu'on utilisera dans la boucle, on mettra un nom plus significatif, particulièrement si la boucle est longue.

- Les identificateurs des variables booléennes doivent être « positifs » : on préférera donc `while (continuer)` ou `while (!sortir)` à `while (pasFini)`.[1]

- 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, on utilise simplement `client` et non pas `unClient`.

- Pour les vecteurs, 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 `numérosDesClients` cependant).

- Les identificateurs de paramètres suivent les mêmes règles, mais comportent en plus un préfixe `p_`, `p_s_` ou `p_es_` selon le type de paramètres. Le premier caractère après le préfixe doit être une lettre minuscule.

---

[1] Il ne faut jamais utiliser des booléens dans un sens inverse de la logique de leur nom qui, par exemple, demanderait d'écrire `while (sortir)` ou `while (!continuer)`.

[2] On pourrait aussi mettre ce préfixe lorsqu'il y a un conflit avec un mot réservé ou déjà défini par le langage ou la bibliothèque standard, mais on préférera choisir un autre identificateur (plus précis ou synonyme, etc.). Par exemple, pour le coût d'un billet, `coût` serait trop semblable à `cout` pour l'écriture, mais on choisira `prix`, `coûtParBillet`, etc., et non pas `unCoût`.

Identificateurs de types

- Le nom des `enum` et des `struct` commencent par le préfixe `Type.`

- Les identificateurs des éléments des `struct` ne doivent pas répéter le nom de la `struct.` Par exemple :

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

Identificateurs des fonctions

- L'identificateur doit normalement être composé de plusieurs mots (ou abréviations connues); 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. Pour les fonctions renvoyant un résultat, le nom de la fonction doit normalement représenter la valeur renvoyée, si c'est la valeur principale déterminée par la fonction. Dans certains cas, ce sera plutôt la valeur d'un paramètre de sortie qui déterminera l'action principale de la fonction et donc le nom qu'elle devrait porter.

  Exemples :
```
AfficherCours(g_tabCours[i]);
cout << NomCatégorie(athlète.codeCatégorie) << ...
AfficherRapportInscriptions();
```

- Dans le cas des fonctions renvoyant un booléen, le nom devrait pouvoir facilement s'enchaîner dans les `if`, `while`, etc.

  Exemples :
```
while (CoursExiste(noCours)) ...
if (EstNbPremier(nombre)) ...
```

- 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 donc pas inutilement le mot `Valide` dans leur identificateur.

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

- 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 écriture. Par contre, `ValiderNoCours(...)` est faux ou trop vague et n'est pas acceptable.

  Exemple :
```
if (NoCoursValide(noCours) && ! CoursExiste(noCours))
    ...
```

2.2 Présentation des déclarations/définitions

Définitions des constantes

- Si possible, regrouper les constantes par thème, en séparant les groupes par des lignes blanches.

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

Définitions des variables

- On déclarera une seule variable par ligne, avec l'indication du type.

- Toute variable peut être commentée, à droite ou au dessus, comme les constantes. On commentera *obligatoirement* les variables ne recevant pas d'initialisation à leur déclaration, mais 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... Dans le doute, il vaut mieux les commenter.

Déclarations des structures

- On présentera un seul champ par ligne, avec l'indication du type.

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

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

Déclarations des types par énumération

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

```
enum TypeÉnumération { VALEUR1, VALEUR2, VALEUR3, ...,      // On met une espace après {
                       VALEURX, VALEURY, ..., VALEURZ };    //  et une 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 TypeÉnumération
    {
    PREMIÈRE_VALEUR= ...,   // Commentaires au besoin...
    DEUXIÈME_VALEUR= ...,
    ...
    DERNIÈRE_VALEUR= ...
    };
```

- Pour les types par énumération, il n'est généralement pas nécessaire de commenter, si les identificateurs sont bien choisis, mais ce n'est pas interdit.

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 90 colonnes (exceptionnellement 95), ce qui est imprimable avec *Listeur* et facile à comprendre pour un humain !

Lignes blanches

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

Commentaires

- On utilise normalement `//` pour les commentaires. Les commentaires `/* */` sont réservés à des cas très précis.

- 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.

- 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 une espace 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 le texte, et non le `//`, d'une espace à partir de la deuxième ligne.

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

- 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 ce doit être une instruction au niveau principal du `for`, et non pas dans une autre structure imbriquée.

- 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 une espace.

```
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 modifiées dans les parties initialisation et incrémentation d'un `for` seront celles qu'on retrouve dans la condition.

- Lorsque c'est possible, on passera les paramètres d'entrée par « référence constante » pour les types `struct`, `string` et les autres types classes (`ifstream`, etc.). On évitera aussi de faire des fonctions qui renvoient des valeurs de ces types : on utilisera plutôt un paramètre de sortie. Par contre, les valeurs des types de base (`int`, `char`, `double`, etc.) et des types `enum` seront toujours passées normalement et seront renvoyées par les fonctions quand c'est possible.

2.5 Présentation des instructions

- Si possible, on déclarera les variables tout juste avant leur première utilisation en les initialisant directement à la valeur désirée. Selon la logique de l'algorithme, il faut parfois déclarer la variable avant son initialisation qui se fait dans un `if`, 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. Par exemple :

```
if (0.0 == CalculerPrixNet(montant, tauxEscompte, taxesÀPayer))
    ...
```

- 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 juste avant un opérateur et idéalement entre des sections semblables.

```
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 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.

- On ne met pas d'espace après les parenthèses ouvrantes ni avant les parenthèses fermantes, sauf pour les `!` dans certaines conditions (voir prochaine règle).

- Pour faciliter la lecture de certaines expressions longues ou complexes, on laissera une espace de part et d'autre de l'opérateur `!`. Dans le cas d'expressions simples, ces espaces 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)
```

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

- Aucune espace ne doit être insérée entre le nom d'une fonction et la parenthèse ouvrante 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'une espace.

- On ne laisse aucune espace autour du point (accès aux champs des `struct`).

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

- Des espaces doivent être insérées autour des opérateurs relationnels et logiques. Si elles aident à la lisibilité, on peut aussi en mettre autour des opérateurs arithmétiques. Les espaces sont donc toujours en symétrie autour de ces opérateurs (une de chaque côté, ou aucune 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'une espace (ce n'est pas symétrique).

```
double salaire= tauxHoraire * nbHeuresTravaillées;

totalCommissions+= commission;
```

- On mettra seulement les parenthèses qui améliorent la lisibilité, ou qui sont nécessaires pour la syntaxe ou l'ordre d'évaluation. Par principe, on mettra toujours des parenthèses lors des insertions d'expressions avec <<.

```
double salaireTotal= salaireDeBase + totalVentes*(1.0+TAUX_COMMISSION);

bool trouvé= (valeur == valeurCherchée);      N.B. Facultatives ici, selon le goût...

return valeur == valeurCherchée;              N.B. Et non pas  return (...);

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

cout << "Les livres ont " << (nbPagesTotales/nbLivres) << " pages en moyenne.\n";
```

- Chaque écriture d'un saut de ligne sera suivie par un changement de ligne dans le programme, sauf qu'on change simplement de ligne dans le cas des sauts de lignes multiples.

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

- On utilise les opérateurs ++ et -- avant les variables à modifier.

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. On mettra alors le `break` en dehors du bloc, car en principe le `break` sort simplement d'un bloc... Ces cas devraient être très rares, car on évite de mettre beaucoup d'opérations dans les `switch` (il vaut alors mieux y appeler des fonctions).

- Pour les choix multiples utilisant des `if-else-if`, on ne met pas de bloc aux `else`, sauf au dernier si nécessaire (voir plus loin).

- Si le bloc d'une structure de contrôle est très long, par exemple plus de 40 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`). Cette situation devrait être rare, car il est préférable de modulariser afin d'éviter cette possibilité.

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

`if` simple

```
if (condition)
    instructionOuBloc1;
else
    instructionOuBloc2;
```

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

- On ne met pas de `else` quand le bloc vrai se termine par un `return`.

- Pour renvoyer le résultat d'une condition (booléen), on fera un `return` *`condition`*`;` (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 choix 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 peut mettre dans un `if`, on fera ce `if` et le dernier `else` servira pour un `assert(false)`.

- Par souci d'efficacité, il faut mettre les cas les plus probables, s'ils sont évidents, 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.

- 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;
        ...
        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.

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

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

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

    instruction2;
    ...
    }
```

```
for(;;)
    {
    instruction1;
    ...
    /***/
    if (condition) break;
    /***/
    instruction2;
    ...
   }
```