1. Présentation

Ce document a pour but de définir les règles et conventions générales pour l'écriture de programmes en langage C.

Il constitue un standard conforme à la norme ANSI langage C, et suppose que le lecteur possède une connaissance minimale du langage.

Il est le document technique de référence pour l'exécution des revues de codes.

2. Organisation et fabrication des fichiers source

Una application C possède 3 unités d'organisation :

• le fichier
• la fonction
• le bloc

Les fichiers source contiennent l'ensemble des fonctions et des données de l'application.

2.1. Nom des fichiers

Le nom d'un fichier source doit refléter la fonction d'usage réalisée par le fichier.
Il est composé d'un nom simple ou d'une expression nominale dont les noms sont reliés par "souligné" (_).
Dans le cas des fichiers MS-DOS, la limitation de la taille du nom des fichiers à 8 caractères impose des abréviations, mais celles-ci doivent être le plus explicite possible; pour cela, supprimer de préférence les voyelles, quand après coupures le nom est encore trop long.

Exemple :
`apprentissage.h;` sous MS-DOS, `APPRENT.H`
`reaction_thermique.c;` sous MS-DOS, `RCT_THRM.C`

2.2. Types de fichiers

Un fichier source peut être de type différents, et selon le type porter l'extension ".h" ou ".c" :

- DEFINITION, d'extension ".h" : ensemble de directives pour le préprocesseur, de définitions de types de données propres à une APPLICATION, à l'exclusion de toute fonction;

- APPLICATION, d'extension ".c" : ensemble de fonctions et de données constituant une application;

- DEPENDANCE, d'extension ".c" : ensemble de fonctions et de données portant sur le même thème et au service privé d'une APPLICATION;

- INTERFACE D'UTILITAIRE, d'extension ".h" : ensemble de directives pour le préprocesseur, de définitions de types de données et de prototypes de fonctions définies dans l'IMPLEMENTATION correspondante;

- IMPLEMENTATION D'UTILITAIRE, d'extension ".c" : ensemble de fonctions réutilisables dans les APPLICATIONs, référencées dans l'INTERFACE correspondante.

Chaque programme exécutable donne lieu à la création d'au moins un fichier de type APPLICATION, constituant le fichier source principal de l'application et contenant au minimum la fonction main( ).

2.3. Composition d'un fichier d'extension ".c"

2.3.1. Fichier APPLICATION et IMPLEMENTATION D'UTILITAIRE

Un fichier source de type APPLICATION se compose, dans l'ordre indiqué :

• d'un en-tête
• de directives d'inclusion de fichiers ".h" standards
• d'une directive d'inclusion du fichier ".h" de DEFINITION propre à l'application
• éventuellement, des directives d'inclusions de fichiers ".c" de DEPENDANCEs
• de déclarations de fonctions prototypes (références en avant - "forward"- ou externes)
• de déclarations et d'inittialisation de variables globales (réduites au strict nécessaire)
• de la définititon de la fonction main()
• de définitions de fonctions autres que main()

Un fichier source de type IMPLEMENTATION D'UTILITAIRE se compose, dans l'ordre indiqué :

• d'un en-tête
• d'une directive d'inclusion du fichier ".h" D'INTERFACE propre à l'UTILITAIRE
• de définitions de fonctions autres que main().

L'en-tête d'un fichier source de type APPLICATION ou IMPLEMENTATION D'UTILITAIRE comporte les rubriques suivantes, dont seules les deux dernières rubriques sont facultatives :

```
/ ********************************************************
  *
  * NOM             : <nom>.c
  * TYPE            : APPLICATION ou UTILITAIRE
  * SUJET           : <descr. sommaire de la fct. d'usage>
  *
  * AUTEUR          : <nom de (des) l'auteur(s)>
  * VERSION         : <numéro de version>
  * CREATION        : <date de création>
  * DER. MODIF.     : <date de dernière modification>
  *
  * ACCES SRC       : <chemin du code source>
  * ACCES OBJ       : <chemin du code obj pour UTILITAIRE>
  * ACCES EXEC      : <chemin de l'exec pour APPLICATION>
  * FABRICATION     : <cmde de compil.ou nom de Makefile>
  *
  * LIMITES         : <limites d'utilisation >
  * CONTRAINTES     : <contraintes d'implémentation>
  *
  ******************************************************** /
```

Exemple :

```
/ ********************************************************
  *
  * NOM             : video.c
  * TYPE            : UTILITAIRE
  * SUJET           : gestion video d'un ecran quelconque
  *
  * AUTEUR          : JPP
  * VERSION         : V1.1
  * CREATION        : 08/10/90
  * DER. MODIF.     : 15/11/90
  *
  * ACCES SRC       : /usr/local/src/video.c
  * ACCES OBJ       : /usr/local/obj/video.o
  * FABRICATION     : cc -c video.c -lcurses
  *
  * CONTRAINTES     : C ANSI, curses
  *
  ******************************************************** /
```

2.3.2. Taille des fichiers APPLICATION et fichiers DEPENDANCEs

Chaque fichier source de type APPLICATION doit comporter au maximum 1000 lignes exécutables et contient une ou plusieurs fonctions.
Chaque fonction doit comporter au maximum 50 lignes exécutables.

Si un fichier source de type APPLICATION atteint la limite des 1000 lignes du fait d'un grand nombre de fonctions, il convient de le fractionner en plusieurs fichiers, dont un principal, de type APPLICATION, et les autres annexes, de type DEPENDANCE, regroupant des fonctions par thèmes.
La rubrique CONTRAINTES de l'en-tête de l'APPLICATION doit alors donner la liste des DEPENDANCEs utilisées.

Pour chaque fichier de type DEPENDANCE, il convient de se demander dans quelle mesure il pourrait constituer un UTILITAIRE, et dans l'affirmative l'organiser en conséquence.
Si tel n'est pas le cas, un fichier de type DEPENDANCE doit contenir, dans l'ordre :

• un en-tête
• des définititons des fonction autres que main()

L'en-tête doit être documenté de la façon suivante :

```
/ ********************************************************
  *
  * NOM              : <nom>.c
  * TYPE             : DEPENDANCE de <nom fich. principal>.c
  * THEME            : <critère de regroup. des composants>
  *
  * AUTEUR           : <nom de (des) l'auteur(s)>
  * VERSION          : <numéro de version>
  * CREATION         : <date de création>
  * DER. MODIF.      : <date de dernière modification>
  *
  * CONTRAINTES      : <contraintes d'implémentation>
  *
  ******************************************************** /
```

2.4. Composition d'un fichier d'extension ".h"

Un fichier source de type DEFINITION comporte, dans l'ordre indiqué :

• un en-tête
• une directive d'inclusion exclusive
• des définitions de constantes symboliques
• des définitions de macro fonctions
• des définitions de types de données

Un fichier source de type INTERFACE D'UTILITAIRE comporte, dans l'ordre indiqué :

• un en-tête
• une directive d'inclusion exclusive
• des définitions de constantes symboliques
• des définitions de types de données
• les prototypes des fonctions définies dans l'IMPLEMENTATION (références externes)

L'en-tête d'un fichier source de type DEFINITION ou INTERFACE D'UTILITAIRE est de la forme suivante :

```
/* $En-tete : <nom>.h  <num.version> <date>  <nom auteur> */
```

La prolifération des fichiers inclus (`#include`), et le fait de leurs imbrications possibles, représente un risque d'inclusions redondantes, donnant lieu à des "warnings" de redéfinition en phase de pré-compilation, voire à des erreurs en phase de compilation.

La directive d'inclusion exclusive a pour but d'éviter les inclusions redondantes.
Elle consiste à encadrer le contenu du fichier ".h" par la définition conditionnelle d'une constante reprenant le nom du fichier, en majuscules, sans son extension, et préfixée par un double "souligné" (__) :

Exemple pour video.h :

```
#ifndef   __VIDEO
#define   __VIDEO
<contenu de l'include video.h>
#endif
```

2.5. Composition d'une fonction

Une fonction comporte, dans l'ordre indiqué :

• une signature
• un en-tête
• un bloc de définition.

La signature est de la forme :

```
ClasseResultat TypeResultat  NomFonction(listeParametres)
```

La `ClasseResultat` peut être omise, et la fonction est alors `extern` (par défaut).

Chaque fonction fait l'objet d'un en-tête tel que présenté ci-dessous :

```
/*
 * ENTREES     : <param. en entrée>
 * SORTIES     : <param. en sortie>
 * E/S         : <param. en entrée/sortie>
 * RESULTAT    : <résultat>
 * DESCRIPTION : <traitement réalisé>
 *
 * GLOBALES    : <var. globales utilisées>
 *
 */
```

`RESULTAT` ne figure pas pour une fonction de type `void`.
`ENTREES, SORTIES` ou `E/S` peuvent ne pas figurer si le fonction ne possède pas ce type de paramètres.
GLOBALES permet de repérer le couplage d'une fonction par variable(s) globale(s).

2.6. Composition d'un bloc

Un bloc se compose des éléments suivants, dans l'ordre indiqué :

• {
• des références "extern" (essentiellement pour les fonctions)
• des déclarations variables locales
• une liste d'instructions
• }

Un bloc de définition d'une fonction est un bloc.
Une liste d'instructions peut contenir un ou plusieurs blocs.
Un bloc peut être vide.

3. Règles de construction des identificateurs

3.1. Objectifs

Ce paragraphe a pour objectif d'augmenter la compréhensibilité des programmes en se rapprochant le plus possible d'expressions en langage naturel.

Exemples :

```
if (PossedeAuMoinsUnClient(clients))
{    ptrClient = RechercherMeilleurClient(clients) ;
     ptrClient->compte = ptrClient->compte + bonus ;
     clients.bilan = positif
}
else
     clients.bilan = negatif ;


PositionAtteinte =
               ExtraireDernierePosition(trajetRobot) ;
```

3.2. Généralités

Un identificateur est un nom de donnée (constante ou variable), ou de fonction.

Un identificateur doit être uniquement composé de lettres, de chiffres et de "souligné" (_).

Les identificateurs débutant par le caractère "souligné" (_) sont réservés au système, et à la bibliothèque C.

Les identificateurs débutant par un double "souligné" (__) sont réservés constantes symboliques (#define ...) privées manipulées par les fichiers de type UTILITAIRE.

Il est déconseillé de différencier deux identificateurs uniquement par le type de lettre (minuscule/majuscule).

Les identificateurs doivent se distinguer par au moins deux caractères, parmi les 12 premiers, car pour la plupart des compilateurs seuls les 12 premiers symboles d'un nom sont discriminants.
La taille maximale d'un identificateur ne doit pas excéder 32 symboles.

Les mots clés du langage sont interdits comme identificateurs.

Il convient également d'éviter de construire des identificateurs correspondants :
- soit à des noms de langage : fortran, pascal, asm, c++... qui sont parfois réservés par le compilateur;
- soit à des mots clé d'autres langages; ce qui poserait problème, en cas d'interfaçage à ces langages.

Exemple de mots clé :

```
.C++ :      class     delete    friend
            inline    new       operator
            overload  public    private
            template  this      virtual ...
```

3.2. Identificateur de variable

Un identificateur de variable est un nom principal suffisamment éloquent, éventuellement complété par :
- une caractéristique d'organisation ou d'usage;
- un qualificatif ou d'autres noms.

Les différents mots constituant un identificateur sont repérés en mettant en majuscule la première lettre d'un nouveau mot.

Un identificateur de variable globale commence par une majuscule.
Un identificateur de variable locale commence par une lettre minuscule.

Certaines abréviations sont admises quand elles sont d'usage courant : nbre, deb, prem, prec, suiv, max, min, sup, inf, longr, largr, hautr, lst , val ...

Les lettres i, j, k utilisées seules sont usuellement admises pour les indices de boucles.

Il est conseillé d'utiliser le préfixe ptr pour nommer une variable de type pointeur.

Exemples :

```
Distance, DistanceMax, ConsigneCourante,
etatBoutonGaucheSouris,  LstDePieces, longrSup
uneFileDAttente, nbreDEssais, PtrTeteLstDePièces,
ptrZoneTensionsActives.
```

3.2. Identificateur de constante

L'identificateur d'une constante symbolique obéit aux mêmes règles sémantiques qu'un identificateur de variable.

L'identificateur d'une constante symbolique définie à l'aide de la directive `define` s'écrit en majuscules, avec séparation de ses composants par un "souligné" (_).

Exemples :

```
#define   VAL_MAX_VITESSE     130
#define   FREINAGE            3
```

L'identificateur d'une constante défini avec le modificateur `const` se construit comme celui d'une variable.

Exemples :

```
const   float   Pi = 3.1416 ;
const   float   valMaxVitesse = 130 ;
```

3.3. Identificateur de type de donnée

Un identificateurs de type de donnée non scalaire (tableau, structure ...), défini ou non avec `typedef`, obéit aux mêmes règles sémantiques qu'un identificateur de variable, sans qualificatif dans la plupart des cas, et commence par une majuscule.

Exemples :

```
struct Date, Individu, FormeDeSignal, AbonneAEvenement
```

3.4. Identificateur de fonction

Un identificateurs de fonction est construit à l'aide d'un verbe, et éventuellement d'éléments supplémentaires :
- une quantité;
- un complément d'objet;
- un adjectif représentatif d'un état .

Il commence par une majuscule pour une fonction de classe `extern` (exportable en dehors de son fichier de définition), et par une minuscule pour une fonctions de classe `static` (non exportable ou privée).

Le verbe peut être au présent de l'indicatif ou à l'infinitif.

Exemples:

```
void Ajouter(...)
static    void sauverValeur(...)
```

L'adjectif représentatif d'un état concerne surtout les fonctions booléennes.

Exemples :

```
EstPresent(...), EstVide(...)
```

La quantité peut, le cas échéant, enrichir le sens du complément.

Exemples :

```
PossedeAuMoinsUnClient(...)
ViderAMoitieLeReservoir(..)
```

Par cohérence avec les habitudes en langage C, les identificateurs de fonction peuvent être exceptionnellement des noms.

Exemple :

```
Acquisition(), TransmissionOK()
```

3.5. Identificateur de macro-fonction

Par cohérence avec les habitudes en langage C, usuellement l'identificateur d'une macro-fonction est construit comme celui d'une constante symbolique, mais utilise nécessairement des parenthèses.

Exemples :

```
#define   LN()            printf("\n")
#define   CLRSCR()        system("clear") /*sous Unix*/
#define   VAL_ABS(x)      ((x) < 0 ? -(x) : (x))
#define   DIM_MENU(tabChoix)  sizeof(tabChoix) /  \
                                  sizeof(char *)
```

Mais aussi :

```
# define ALLOUER_ELEM (nbElem, typElem) \
                      (struct typElem *)calloc(nbElem,\
                                sizeof(struct typElem))
```

3.6. Paramètres d'une fonction

Les identificateurs des paramètres d'un fonction sont construits comme les identificateurs de variables locales; ils commencent, notamment par une minuscule.

L'ordre de définition des paramètres doit respecter la règle suivante :

```
NomFonction(parametrePrincipal, listeParametres)
```

où `parametrePrincipal` est la donnée principale sur laquelle porte la fonction, la `listeParametres` ne comportant que des données secondaires, nécessaires à la réalisation du traitement réalisé par la fonction.

Exemple :

```
Ajouter(BlocMesures mesures, float uneMesure )
```

La sémantique de cette fonction est d'ajouter une mesure a un bloc de mesures (qui est la donnée principale) et non, d'ajouter un bloc de mesures à une mesure.

4. Règles de programmation en C

4.1. Objectifs

La syntaxe du langage C est très concise et, sans précaution, on risque facilement d'obtenir des programmes illisibles.

Ce paragraphe précise les règles d'écriture des programmes à adopter afin d'en assurer la lisibilité, donc de faciliter les tâches de correction et de maintenance des logiciels.

4.2. Recommandations pour l'utilisation des commentaires

Les commentaires servent à documenter un fichier source afin d'aider à la compréhension des programmes.

4.2.1. Types et formes de commentaires

On distingue plusieurs types de commentaires :
- ceux d'en-tête de fichier ou de fonction;
- ceux de description du rôle d'une inclusion, d'une constante symbolique ou d'une variable;
- ceux de description d'un traitement;
- ceux de description d'un bloc.

L'usage des tabulations permet des alignements verticaux des commentaires, indépendamment des éléments commentés, et permet la modification des textes des commentaires sans avoir à ajuster ces alignements.

La forme et le contenu des commentaires d'en-têtes sont décrits aux § 2.3, 2.4 et 2.5.

Un commentaire relatif à une inclusion, une donnée ou un traitement peuvent revêtir deux formes :
- à la suite de l'inclusion, de la donnée ou du traitement, s'il y a suffisamment de place sur la ligne;
- avant l'inclusion, la donnée ou le traitement s'il n'y a pas suffisamment de place ou si le commentaire porte sur plusieurs éléments.

Dans le premier cas, son début doit être séparé de l'élément commenté par au moins une tabulation, ou un espace s'il n'y a pas assez de place, et sa fin doit être alignée sur la tabulation la plus proche du bord d'une page de listage.

Dans le second cas, le commentaire doit se présenter comme pour l'en-tête d'une fonction, avec alignement par rapport au début de l'objet du commentaire.

Il est interdit de placer un commentaire au milieu d'une instruction.

Exemples :

```
#include  <stdio.h>          /* pour getchar() et EOF   */

/*
 * pour les mem. partagees de consigne et de suivi
 */

#include  <sys/ipc.h>
#include  <sys/shm.c>

#define   NBRE_MAX_CLIENTS    20   /* limite indepassable*/

/*
 * variables globales
 */

Booleen   Fini = FAUX ;            /* signale fin totale */
int       descFichMesures = -1 ;   /* constamment accede'*/

     /*
      * reinitialise le pointeur de fichier
      * lit un enregistrement et le transforme en
      * chaine de caracteres avant de l'afficher
      * ou indique la fin du fichier
      */

     lseek(fd, OL, 0) ;
     nbOctLus = read(fd, buffer, T_MAX_BUFFER) ;
     if (nbOctLus == -1)
          ErreurSys("Pb de lecture de fichier") ;
     else if (nbOctLus != 0)
               buffer[nbOctLus] = FIN_CHAINE ;
          else
               finFichierAtteinte = VRAI ;
```

Un commentaire de description de bloc se place en tête de bloc.
Il commence sur la même ligne que "{", s'étend sur une ou plusieurs lignes, selon la même forme que le deuxième des deux cas précédents.

Exemple :

```
     for (i = 0; i < NB_ELEM; i++)
     {    /*
           * somme ponderee des valeurs
           */
          somPonderee = tabElem[i].valeur * tabElem[i].poids ;
     }
```

4.2.2. Contenus des commentaires

Un commentaire doit aider à la compréhension d'un programme et apporter une plus value dans ce sens :
- en jouant le rôle de cartouche d'un fichier ou d'une fonction;
- en justifiant l'existence d'une inclusion;
- en décrivant le pourquoi, le rôle, d'une donnée;
- en résumant les étapes ou l'idée générale d'un traitement;
- en délimitant une zone de code à maintenir;
- en localisant un traitement dépendant d'une machine ou d'un système.

Il faut éviter les commentaires redondants du type :

```
i = i + 1 ;    /* on incremente i                    */
```

4.3. Recommandations pour l'utilisation des espaces

- les mots clés suivis d'une expression entre parenthèses sont séparés de la parenthèse ouvrante par un espace :

```
for (...
if (...
return (...
```

- dans les déclarations ou les appels de fonction, le nom de la fonction est par contre immédiatement suivi de la parenthèse ouvrante :

```
nomFonction(...)
```

Cette règle s'applique également pour les macro-fonctions.

- les éléments d'une liste d'arguments sont séparés par une virgule suivie d'un espace :

```
main(argc, argv, envp)
```

- il n'y a pas d'espace pour encadrer les opérateurs ( ), [ ], ., ->

```
getchar(unCaractere)
unTableau[index]
uneDate.jour
unPointeurDeStructure->nomDuChamp
```

- il n'y a pas d'espace entre un opérateur unaire et son opérande

```
* unPointeur ++     /* deconseille'                  */
*unPointeur++       /* recommande'                   */
```

Cette règle concerne les opérateurs `*  &  -  !  ~  ++  --  (cast).`

- les opérateurs binaires et d'affectation sont entourés d'un espace de part et d'autre

```
a=b*(c+d)          /* deconseille'                  */
a = b * (c + d)    /* recommande'                   */
```

- les opérateurs d'affectation sont alignés à l'intérieur d'un bloc d'instructions

```
uneVariable = ...              /* deconseille'        */
uneAutreVariable = ...
uneVariable         = ...      /* recommande'        */
uneAutreVariable    = ...
```

- le terminateur d'instruction ";" est précédé d'un espace

```
instruction;         /* deconseille'                */
instruction ;        /* recommande                  */
```

4.4. Constantes

4.4.1. Notation des constantes numériques

Pour l'écriture d'une constante :

- hexadécimale : 0xFFFF est plus lisible que 0XFFFF
- entière longue : 1L est plus lisible que 1l.
- flottante : 10e5 est plus lisible que 10E5

4.4.2. Utilisation des constantes symboliques

Aucune valeur ne doit apparaître dans les expressions ou déclarations.
Toutes les constantes sont des constantes symboliques définies par :

```
#define   NOM_CONSTANTE  valeur    /* commentaire */
```

La définition doit être accompagnée d'un commentaire explicatif dès que nécessaire.
Si une constante est utilisée dans plusieurs fichiers, un fichier d'inclusion de constantes doit être créé.

Une constante symbolique qui n'est plus utilisée doit être libérée avec undef :

```
#undef    NOM_CONSTANTE
```

Il est plus dans l'esprit du C ANSI d'utiliser le modificateur `const` pour définir une constante symbolique.

4.5. Variables et types

4.5.1. Déclarations de variables

Par souci d'homogénéité, les déclarations doivent être faites dans l'ordre :

```
extern
register
static
[auto]
```

Toutes les variables doivent être déclarées de façon explicite, en séparant d'une tabulation leur classe, leur type et leur identificateur.

Exemples :

```
static    int  compteur ;
char      *AdrDebut ;
```

La description d'un type structuré (struct, union) doit respecter la forme suivante :

```
struct    <NomStruct>
{
     <premier champ> ;
     ...
     <dernier champ> ;
} ;
```

Le nom de la structure (struct, union) commence par une majuscule.
Le nom de chaque champ de la structure commence par une miniscule.

Les dimensions des tableaux doivent être définies par des constantes symboliques qui seront notamment utilisées comme bornes de parcours du tableau.

Exemple :

```
struct    AdressePostale
{
     int  numero ;
     char nomDeRue[LONGUEUR_MAX] ;
     long codePostal ;
     char localite[LONGUEUR_MAX] ;
} ;

int  descLignesSerie[NB_LIGNES_SERIE] ;
```

Il est préférable d'utiliser un type énumératif au lieu d'un type entier si la logique intrinsèque le permet.
Ceci assure une meilleure compréhensibilité et surtout une meilleure fiabilité du programme.

Exemple :

```
enum  Semaine
       {
          LUNDI = 1, MARDI, MERCREDI, JEUDI,
          VENDREDI, SAMEDI, DIMANCHE
       } uneSemaine ;
```

Il est préférable d'utiliser un type énumératif au lieu d'un type booléen si la logique le permet :

Exemple :

```
enum EtatBouton     {APPUYE, RELACHE} etatCommande ;
```

est plus clair que l'utilisation des valeurs VRAI et FAUX d'un type "booléen".

Les déclarations de variables doivent suivre un ordre logique et être classées par groupes de données en relation.

Exemple :

```
int  descLignesSerie[NB_LIGNES_SERIE] ;
int  numLigneSerieCourante = -1 ;
char nomLignesSerie[T_MAX_NOM_FICH] ;
```

4.5.2. Initiialisations de variables

Une variable, y compris de type pointeur, ne doit être utilisée qu'après avoir été initialisée ou affectée.
Dans le cas contraire, des résultats imprévisibles et non reproductibles risquent de se produire.

Il est conseillé d'initialiser les variables globales à leur déclaration.

Exemple :

```
float     x = 0 ;
struct    LstDemandes    *premDemande = NULL ;
float     *ptrVal = &x ;
```

Se limiter à une déclaration avec initialisation par ligne.

Exemples :

```
     /*
      * incorrect
      */
long unEntierLong, unAutreEntierLong = 0 ;
     /*
      * correct
      */
long unEntierLong ;
long unAutreEntierLong = 0 ;
     /*
      * encore correct
      */
long unEntierLong ,
     unAutreEntierLong = 0 ;
```

Déclarer une variable par ligne permet de bien commenter chaque variable.

Exemple :

```
char selection ,          /* choix de l'utilisateur   */
     typeReponse = 'O' ; /* oui (O) par defaut       */
```

Une variable statique est généralement initialisée au moment de la compilation.
Dans ce cas, l'initialisation ne peut se faire qu'avec des formes constantes, évaluables à la compilation.

```
/*
 * interdit
 */
static    long uneVariable = nomFonction(i) ;
```

L'initialisation d'un tableau ou d'une stucture (ou d'une union) peut tenir sur une ligne ou sur plusieurs lignes.
Dans le second cas, chaque ligne doit être indentée à l'aide d'une tabulation et contenir une seule valeur.

Exemples :

```
/*
 * NB_REP peut etre sup. a 3
 */
static    int  reponses[NB_REP] = { -1, 0, 1} ;

char *Menu[] =
{
     " Ajouter element" ,
     " Supprimer element" ,
     " Modifier element " ,
     " Quitter "
} ;
```

4.5.3. Visibilité des variables

Pour assurer la modularité des programmes et minimiser les couplages par variables globales, il faut utiliser au maximum les variables locales, combinées avec l'appel de fonction et le passage de paramètres.
Les variables globales ne sont à utiliser qu'en cas de nécessité absolue.

Lorsque la visibilité des variables est limitée au strict minimum, il en résulte une meilleure allocation des ressources (qui sont restituées au plus tôt).

Même si les déclarations de variables peuvent être éparpillées selon le bon vouloir du programmeur, la lisibilité des programmes sera grandement améliorée en déclarant en tête de bloc les données qui doivent respectivement être visibles à l'intérieur de celui-ci.

4.5.4. Definition de types

La definition de type est un moyen de classification des variables utile pour rendre un programme plus compréhensible.

Il est recommandé de définir des types distincts pour des données appartenant à des ensembles logiquement indépendants.

L'association de fonctions à un type de donnée est la règle privilégiée pour l'identification des données et des fonctions des fichiers de type UTILITAIRE.
Chaque fichier de type INTERFACE D'UTILITAIRE doit définir les types de données qu'il manipule par utilisation systématique de l'instruction `typedef`, et les constantes symboliques dont il a besoin.

Exemple :

```
/* $En-tete : pile_entier_long.h  V1.1  10/11/93  JPP    */

#ifndef   __PILE_ENTIER_LONG
#define   __PILE_ENTIER_LONG

#define   __TAILLE_MAX_PILE   100

#ifndef   __BOOLEEN
#define   __BOOLEEN
typedef   enum { FAUX, VRAI} Booleen ; /* FAUX = 0 ...    */
#endif

typedef   struct
          {
               long      cadre[__TAILLE_MAX_PILE] ;
               short     sommet ;
          }  PileDEntierLong ;

extern    void      CreerPile(PileDEntierLong *) ;
extern    void      Empiler(PileDEntierLong *, long) ;
extern    long      Depiler(PileDEntierLong *) ;
extern    short     SommetPile(const PileDEntierLong *) ;

#endif /* __PILE_ENTIER_LONG */
```

L'utilisation des types de base est, a priori, réservée pour la définition de types synonymes.

Exemple :

```
typedef   float     Cordonnee ;
```

4.5.6. Conversion de types

Dans une expression "a op b", le type du résultat est celui de l'opérande de plus haut niveau hiérarchique.
La hiérarchie croissante de niveau est la suivante :

```
signed --> unsigned
short --> int --> long --> float -- >double
```

Les opérandes sont convertis dans le type de plus haut niveau, préalablement à l'exécution de l'opération.
La conversion implicite peut amener des problèmes :

- extension de signe variable selon les systèmes,
- conversion du bit de signe en bit de valeur,
- troncature.

Il est donc conseillé de rendre les conversions de type explicites (opérateur "cast") et de toujours opérer entre grandeurs de même type.
Ces conversions doivent être limitées aux conversions de type simple et à certaines opérations sur pointeurs (notamment dans l'utilisation de la fonction `malloc()`).

Exemples :

```
float     unReel ;
int       unEntier ;
unReel    = (float)unEntier ;
```

4.6. Instructions

4.6.1. Indentation du code

L'indentation du code facilite la lisibilité de la structure de contrôle et des niveaux d'imbrication.

L'indentation doit respecter les règles suivantes :
- indenter en utilisant la touche Tabulation (insertion de TAB dans le fichier) de façon à pouvoir redéfinir aisément la largeur d'une indentation. Autrement dit, on n'indente pas par un certain nombre de caractères SPACE;

- après "{", indenter à droite (sauf après switch);

- avant "}", indenter à gauche;

- indenter selon le format suivant :

```
`


/*
```

```
 * commentaire <instruction de boucle>
 */
<instruction de boucle>
{
     <corps de boucle>
}

/*
 * commentaire <instruction de test>
 */
if <condition de test>
{   /*
     * commentaire du <then>
     */
     <corps de test vrai>
}
else
{   /*
     * commentaire du <else>
     */
     <corps de test faux>
}

/*
 * commentaire <instructions de test>
 */
if <condition de test>
{   /*
     * commentaire du <then>
     */
     <corps de test vrai>
}
else if <condition de test>
{   /*
     * commentaire du <then>
     */
    <corps de test vrai>
}
else
{   /*
     * commentaire du <else> du dernier if
     */
     <corps de test faux>
}
```

```
/*
 *commentaire <instruction de switch>
 */
<instruction de switch>
{
<cas 1> : /* commentaire du <cas 1>                */
     <corps de cas 1, avec une inst. par ligne>
...
<cas n> : /* commentaire du <cas n>                */
     <corps de cas n, avec une inst. par ligne>
default : /* commentaire de "autres cas"           */
     <corps de "autres cas">
}
```

Les lignes de plus de 80 caractères de longueur sont segmentés et chaque segment indenté d'une ou plusieurs tabulations sur la ou les lignes suivantes.

Une ligne blanche doit séparer chaque structure de la programmation structurée comme illustré ci-dessus.

Pour une structure de contrôle étendue sur plus de 20 lignes, la dernière accolade de fin de bloc doit être suivie d'un commentaire de fin portant sur le type de la structure de contrôle.

Exemple :

```
<instruction de switch>
{
<cas 1> :
     <corps de cas 1>
...
<cas n> :
     <corps de cas n>
default :
     <corps de "autres cas">
} /* fin switch */
```

Cette présentation permet de faire apparaître explicitement l'algorithme utilisé par une fonction, qui n'aura pas besoin d'être documenté par ailleurs dans un document de conception détaillée séparé du source.

4.6.2. Utilisation des boucles

L'utilisation des instructions `continue` et `break` est à éviter à l'intérieur du corps d'une boucle.

Deux types de boucle sont nécessaires :

- la boucle avec un nombre d'itérations connu a priori : boucle `for`
- la boucle avec un nombre d'itérations inconnu a priori :
    - si au moins une exécution du corps de la boucle est obligatoire :
    boucle `do-while`
    - si le passage dans le corps de la boucle est facultatif : boucle `while`.

Pour une boucle `for`, les trois champs (initialisation, la condition, modification) doivent être remplis et porter sur une seule et même variable de comptage.
La variable de comptage ne doit pas être modifiée dans le corps de la boucle.

Exemples :

```
for (i = 0; i < 10; i++)                /* correct   */
for (i = 0, j = 0; i < 10; i++, j++)    /* incorrect */

for (i = 0; Ok == VRAI & i < 10 ; i++)  /* incorrect */
```

dans ce dernier cas, il faut utiliser une boucle `while`.

```
for (i = 0; i < 10; i++)
{
     ...
     i = i + 2 ;                        /* incorrect */
     ...
     if ( Ok == VRAI)
          i = 10 ;                      /* incorrect */
     ...
}
```

L'utilisation de variables significatives permet de lever toute ambiguïté.

Exemple :

```
unIndex = 0 ;
for (indexDeParcours = 0; indexDeParcours < 10;
          indexDeParcours++)       /* correct        */
{
     <corps de la boucle for>
     unIndex ++ ;
}
```

La boucle `while` peut se présenter sous deux formes distinctes :

- le "tant que faire" :

```
while (condition)
{
     <corps de la boucle "tant que faire">
}
```

- le "faire tant que" :

```
do
{
     <corps de la boucle "faire tant que" >
}
while (condition)
```

La valeur initiale de la variable de contrôle doit être connue avant l'entrée dans la boucle :
- soit par une initialisation explicite placée avant la boucle,
- soit par un commentaire indiquant l'état initial de cette variable de contrôle.

Une boucle infinie sera codée à l'aide d'une boucle `while (1)` ou `while (VRAI)` et non pas avec `for ( ; ; )` :

```
while (VRAI)
{
     <corps de la boucle infinie>
}
```

4.6.3. Conditions de tests multiples

En cas de choix multiples portant sur des égalités à des constantes, l'utilisation du `switch` doit être préférée à celle de `elsif`.

Dans un aiguillage (`switch`), l'instruction `continue` est interdite et l' utilisation de `break` est limité aux fins de corps des `case` du `switch`.

En fin du `switch`, le cas `default` est obligatoire, quitte à commenter éventuellement qu'il est : `/* sans objet dans le contexte */`.

4.6.4. Expressions de test

Dans le cas d'une expression composée faisant appel à une affectation, l'affectation doit être séparée du test.
On peut combiner cependant affectation et test à l'aide de l'opérateur de liste.

Exemples :

```
while ( (c = getchar()) != EOF)         /* incorrect */
{
     ...
}

while (c = getchar(), c != EOF)         /* admis     */
{
     ...
}

c = getchar() ;                         /* correct   */
```

```
while ( c != EOF)
{
     ...
     c = getchar() ;
}

if (fd = open(nomFich, 0_RDWR), fd != -1) /* correct */
```

Il est recommandé de ne pas utiliser l'égalité ou l'inégalité stricte dans des comparaisons de réels au risque de créer des boucles infinies du fait des troncatures.

Dans le cas d'expressions de test longues portant sur plusieurs lignes, les opérateurs logiques (&, |, ||, &&, etc.) doivent apparaître en début de ligne.

Exemple :

```
while ( temperatureCourante < SEUIL_TEMP
          && acquiqitionFinie == FAUX
          && cmdeArret != APPUYE )
{
     ...
}
```

4.7. Fonctions et macro fonctions

4.7.1. Définition de fonction

La définition d'une fonction est faite de la manière suivante :

```
classeFonct typeResultat  NomFonction(listeParametres)
     /*
      * <en-tete de la fonction>
      */
{
     <references a d'autres fonctions>

     <declarations de variables locale>

     <corps de la fonction>
}
ou
classeFonct typeResultat  *NomFonction(listeParametres)
     /*
      * <en-tete de la fonction>
      */
{
     <references a d'autres fonctions>

     <declarations de variables locale>

     <corps de la fonction>
```

```
}
```

Si la fonction ne retourne pas de valeur, elle est déclarée `void`.
Si une fonction retourne un résultat, elle doit posséder une instruction `return` et une seule en fin de corps.

Chaque paramètre de la fonction est déclaré explicitement.
Quand un paramètre ne doit pas être modifié dans le corps de la fonction, sa déclaration doit être précédé de `const`.

L'utilisation de l'opérateur ternaire "? :" est limitée à des retours de fonctions pour lesquels il présente certains avantages de concision.

Exemples :

```
static    Booleen   pilePleine(const PileDEntierLong
                                          *pile)
{
     return ((pile->sommet == __TAILLE_MAX_PILE) ?
                                        VRAI : FAUX) ;
}

void      CreerPile(PileDEntierLong *unePile)
{
     unePile->sommet = 0 ;
}
```

4.7.2. Déclaration de fonction prototype

Une fonction prototype est une référence en avant (forward) nécessaire pour éviter les erreurs de compilation.

Dans la déclaration d'une fonction prototype, il faut donner des noms aux paramètres à chaque fois que l'on veut définir la fonction référencée avec les mêmes noms de paramètres.

Exemples :

```
extern    float     calculerPrixTTC(float somme,
                                     float tauxTVA) ;
extern    void      echanger( int *, int *) ;
```

4.7.3. Macro-fonctions

Il est recommandé de limiter l'utilisation des macro fonctions à l'implémentation de fonctions simples.
Une macro fonction doit se limiter le plus possible à une seule ligne de code, sinon comporter 5 lignes au maximum.

Les paramètres d'une macro-fonction doivent toujours être mis entre parenthèses dans le corps de cette macro-fonction.

Exemple :

```
#define   CARRE(x)  ((x) * (x))
```

En l'absence de parenthèses, `CARRE(2 + 3)` produirait 2 + 3 * 2 + 3, soit 11.

L'utilisation de l'opérateur ternaire "? :" est limitée à l'expression de macro-fonctions où il présente certains avantages de concision.

Exemple :

```
#define   MAX(a,b)  ((a) < (b) ? (b) : (a))
```

5. Restriction d'utilisation des constructions C

5.1. Règles sur les opérateurs

5.1.1. Opérateurs d'incrémentation

Les opérateurs ++ et -- sont interdits quand ils sont succeptibles de créer des effets de bord contraires à l'évaluation des expressions.

Exemples :

```
x = x++ ;                          /* interdit       */
unTableau[x] = ++x ;               /* interdit       */
x = y++ = 0 ;                      /* interdit       */
nomFonction(x, ++x) ;              /* interdit       */
```

Les opérateurs d'incrémentation sont interdits en arguments effectifs d'un appel à une macro.

Exemple :

```
#define   max(a,b)  ((a) > (b) ? (a) : (b))

max(p++,q++)   /* interdit car p ou q serait          */
               /* incremente deux fois                */
```

Les incrémentations et décrémentations de variables (++ ou --) se font sur des éléments simples et lorsque la variable n'est pas réutilisée dans l'expression.

5.1.2. Opérateur virgule

L'utilisation de l'opérateur "," dans une expression entre crochets ([ ]) ou dans un argument de fonction est interdite dans la mesure où elle laisse croire à l'utilisation d'un tableau à n + 1 dimensions ou respectivement à une fonction à n + 1 paramètres.

```
x = unTableau[k = n + 1, 2 * k] ;       /* interdit  */
nomFonction((i = 1, i + 2)) ;           /* interdit  */
k = n + 1 ;                             /* correct   */
x = unTableau[2 * k] ;
```

5.1.3. Opérateurs sur bits

Pour réaliser un masquage (mise à 0) ou un marquage (mise à 1) de certains bits, il est conseillé de générer des masques dynamiques.

Exemple :

```
b = b & 0xFFEF ;/* incorrect - préjuge b sur 16 bits */

b = b & ~(1  <<  5) ;    /* correct                  */
```

Il est conseillé de parenthéser les opérateurs sur bits, dès qu'ils entrent dans des expressions afin d'éviter les problèmes de précédence.
Ceci concerne : & | ~ ^ >> << .

Exemple :

```
(status & MASK) != SET   /* necessite des ( )        */
```

5.2. Allocation mémoire et taille des types de données

Lors de l'utilisation d'une fonction d'allocation mémoire, le retour "NULL" doit impérativement être testé : il indique une impossibilité d'allocation.

Il est impératif de libérer la mémoire allouée dès qu'elle n'est plus utilisée par :

```
free(unPointeur) ;  /* le pointeur est "en l'air"  */
unPointeur = NULL ; /* on le fixe                  */
```

Des allocations suivies de désallocations conduisent au morcellement de la mémoire.
Il est conseillé d'organiser allocations et désallocations à la manière d'une pile LIFO et d'utiliser la fonction `realloc( )` pour modifier la taille d'une zone précédemment allouée.

La gestion dynamique de la mémoire étant un processus fréquemment mis en œuvre en C, il faut définir des macro-fonctions appropriées inspirées des exemples ci-après:

```
#define   Malloc(type)   (type *)malloc(sizeof(type))
#define   Calloc(n,type) (type *)calloc(n,sizeof(type))
```

La taille d'une structure de donnée ne doit jamais être estimée / calculée par le programmeur. Il ne faut pas préjuger de la taille d'un `int`, elle varie de 2 à 4 octets selon les machines et les compilateurs.

L'opérateur `sizeof` permet de s'affranchir de la connaissance de la taille de int et des des tailles des structures de données manipulées : il doit donc être utilisé partout où son emploi se justifie.

5.3. Règles sur les pointeurs

Seules les affectations suivantes sont autorisées dans un pointeur :
- `NULL;`
- une fonction du type Malloc(type) ou Calloc(n, type);
- un autre pointeur.

Deux pointeurs doivent être de type cohérent avant d'entreprendre une opération sur ceux- ci. D'une manière générale, afin de permettre une meilleure compréhension d'un source, il est souhaitable que les opérations d'alignement de pointeur -cast- soient explicites.

Exemple :

```
int  *unPointeur ;
if (unPointeur == 0) ...            /* deconseille' */

int  *unPointeur ;
if (unPointeur == (int *)NULL) ...  /* recommande'  */

int  *uneAdresse = (int *) 0xFF1A0 ; /* recommande'  */
```

A l'exclusion des pointeurs de tableaux, un pointeur vers un autre pointeur est déconseillé lors d'une déclaration.

Les comparaisons sur les pointeurs devraient se limiter aux relations == et !=, les inégalités ayant une interprétation limitée.

Une variable automatique n'ayant de sens que dans le bloc où elle a été définie, il ne faut pas garder un pointeur sur une telle valeur et en utiliser le contenu une fois sorti du bloc.
Si une fonction retourne un pointeur sur un élement d'un certain type, et si l'on veut conserver la valeur retournée, il faut déclarer l'élément retourné de classe de stockage `static.`

```
struct    S    *f()
 {
    static    struct    S    resultat ;
    /* construction de résultat */
    return ((struct S *) &resultat) ;
 }
```

5.4. Utilisation des macro-instructions

L'utilisation de la même variable en paramètre d'appel de plus d'une macro-fonction est interdit dans une ligne d'instruction.

```
/*
 * interdit
 */
variableResultat = PREMIERE_MACRO(uneVariable) +
                   DEUXIEME_MACRO(uneVariable)
```

On ne peut, en effet, pas préjuger dans quel ordre le compilateur exécutera les deux macro-instructions, ce qui peut induire des effets de bord non désirés sur la variable uneVariable.

5.5. Ordre de stockage des octets, alignement des données

L'ordre de stockage des octets composant une entité "C" varie en fonction des machines : l'exemple typique reste la différence entre les processeurs Intel et les processeurs Motorola qui inversent l'ordre poids forts - poids faibles sur les octets d'adresse paire - impaire.

Il ne faut donc manipuler une entité "C" que par des pointeurs ou par valeur, sans dissocier les composantes élémentaires qui la constitue.

Les alignements des données dépendent des machines.
Par exemple, l'écriture binaire d'une structure dans un fichier n'est pas portable.
Les lectures et / ou écritures dépendent, en effet, de la taille des objets décrits dans la structure, de l'alignement, de l'ordre des octets qui varient en fonction du processeur et du compilateur.

Exemple :

```
typedef         struct
                {
                     int unEntier ;
                     char unTableauDeCaracteres[5] ;
                     long unEntierLong ;
                } MaStructure ;
#define         DIMENSION 6
MaStructure     unTableau[DIMENSION] ;

(void) fwrite((char *) MaStructure,sizeof(MaStructure),
                                   DIMENSION, unFichier) ;
```

n'est pas portable, car le fichier généré sur une première machine ne pourra pas être relu sur n'importe quelle autre machine.

5.6. Codage des caractères

UNIX System V utilise actuellement le code ASCII 7 bits.
Il faut donc utiliser les fonction de `ctype.h` pour s'adapter à un autre code que l'ASCII 7 bits.

Exemple :

```
/*
 * non portable en EBCDIC
*/
if (unCaractere >= "a" && unCaractere <= "z")
/*
 * portable
*/
if (islower(unCaractere))
```

6. Fabrication des exécutables

L'exécutable d'une APPLICATION composée de DEPENDANCEs peut être constitué :

- soit par des inclusions de fichiers DEPENDANCES dans le fichier APPLICATION, et compilation globale du tout;
- soit par compilation séparée des différents fichiers et édition de liens.

Comme les DEPENDANCES utilisent les définitions et références du fichier de DEFINITION associé à l'APPLICATION :

- la première solution est possible en cours de développement, à condition de respecter l'ordre d'inclusion suivant : fichier de DEFINITION, puis fichiers DEPENDANCEs;
- la seconde est conseillée en phase de fabrication finale, à condition d'inclure dans chaque DEPENDANCE le fichier de DEFINITION.

Si une APPLICATION utilise des UTILITAIREs, elle doit inclure les fichiers d'INTERFACEs des UTILTAIREs, et son exécutable doit être construit par compilation séparée et édition de liens avec les objets (.o) des IMPLEMENTATIONs des UTILITAIRES

Toute compilation séparée doit être réalisée à l'aide de `make`.