partie 3 : Les classes de mémorisation

MOOC Langage C INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

Outre son nom, toute variable du langage C possède deux attributs fondamentaux : son type et sa classe. Le type est prédéfini par le langage ou fabriqué par l'utilisateur à l'aide de `typedef`. Nous avons déjà vu quelques exemples d'emploi de `typedef`, et nous y reviendrons dans le chapitre sur les structures. La classe concerne :

— la portée, associée aussi au fait que la variable peut être importée ou qu'elle est exportable, et qui représente la région du programme pour laquelle la variable est définie,

— la visibilité, qui indique dans quelles régions du programme source la variable est utilisable (nous verrons la différence avec la portée),

— la durée de vie, qui est associée, en langage C, avec l'implantation de la variable en mémoire.

Parlons d'abord de la portée et de la visibilité des variables en généralisant ce que nous avons vu pour les fonctions.

1 Portée et visibilité des variables

1.1 Variables externes et variables internes

Une variable définie hors de tout bloc est dite *externe* (dans le sens, justement, d'externe à tout bloc). Une variable définie à l'intérieur d'un bloc, donc d'une fonction, est dite locale ou *interne* à ce bloc. Dans ce cas, rappelez-vous que la définition d'une variable doit se faire en tête de bloc, avant les instructions.

1.2 Niveau d'une variable

Puisque tout bloc est enfermé entre accolades, le niveau d'un bloc est donné par la différence entre le nombre d'accolades ouvrantes et le nombre d'accolades fermantes rencontrées depuis le début du fichier source jusqu'au bloc considéré (reportez-vous à la figure ci-après). Les variables externes et les définitions de fonction sont donc de niveau 0.

1.3 Portée d'une variable

Si la variable est interne, elle porte normalement sur tout le bloc où elle est définie. Pour une variable externe, il faut imaginer que le compilateur travaille en une seule passe. La portée de la variable commence donc à l'endroit où elle est définie dans le fichier pour s'étendre jusqu'à la fin du fichier (voyez

l'exemple de la variable `Niv_Zero` dans la figure suivante). Aussi, une variable de niveau 0 déclarée en tête d'un fichier est-elle souvent qualifiée de globale au fichier.

1.4 Visibilité d'une variable

Comme le montre la figure 1, la redéfinition locale d'une variable (la réutilisation du même nom pour une définition dans un bloc de niveau plus élevé) masque cette variable, sans en altérer la portée. Dans cette figure, `Pi` et `Vi` représentent la portée et la visibilité de la variable numérotée `i` en commentaire. La variable `Niv_Zero` montre l'exemple d'une variable de niveau 0 définie entre deux fonctions.

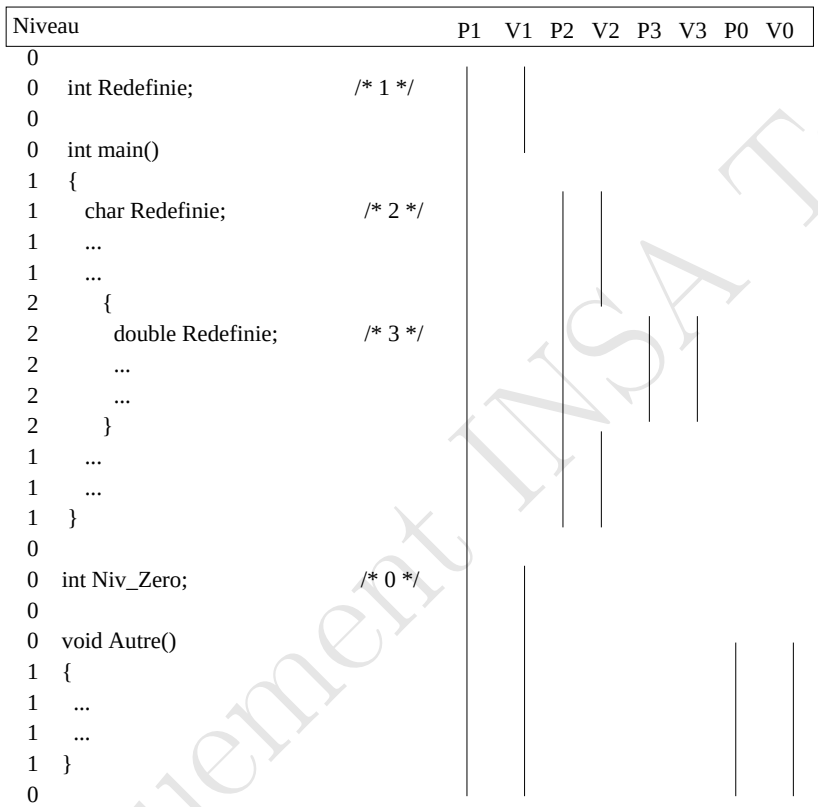

FIGURE 1 – Portée et visibilité

Dans l'exemple de la figure, la variable externe `Redefinie` (un entier) porte sur la totalité du fichier mais elle est par exemple invisible dans la fonction `main()` où elle est redéfinie. En revanche, elle est utilisable dans `Autre()`.

IMPORTANT ! Malgré l'exemple donné, sachez qu'il faut proscrire la réutilisation d'un même nom pour divers usages. Si le compilateur ne se trompe pas, le programmeur, lui, a toutes les (mal)chances de se prendre les pieds dans ses définitions. Trouvez donc à des objets informatiques différents des noms différents et évocateurs. La lisibilité du programme ne peut qu'en bénéficier et la facilité de maintenance qu'en être améliorée.

1.5 Modification de la portée d'une variable externe

Grâce aux déclarations (opérations qui ne réservent aucune place en mémoire, à l'inverse des définitions), on augmente la portée d'une variable ou d'une fonction.

1. À l'intérieur d'un même fichier : une variable de niveau 0 peut avoir sa portée augmentée en amont de sa définition grâce à une déclaration commençant par le mot réservé `extern`. Cette déclaration peut aussi bien être faite en niveau 0 qu'à l'intérieur d'un bloc (déclaration locale, dans ce cas) :

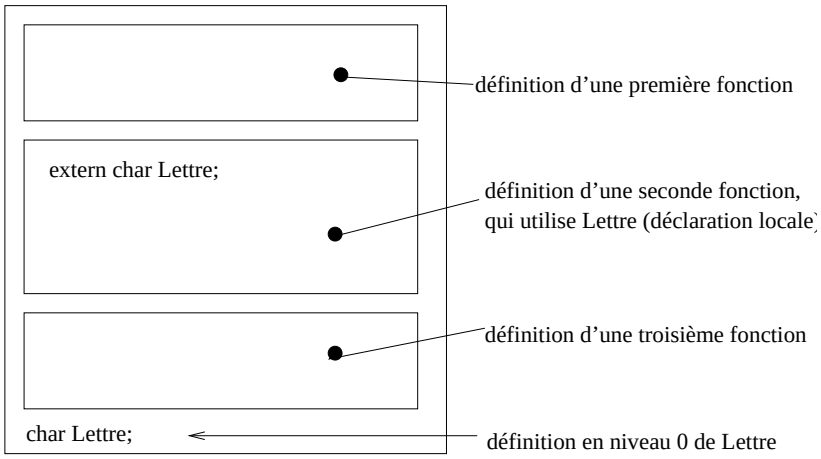

FIGURE 2 – Cas où l'utilisation précède la définition

En pratique, on préfère usuellement définir des variables globales, en tête du fichier, ce qui évite la complication inutile de la figure 2, puisque la définition vaut déclaration. Dans une définition, il est donc interdit d'utiliser le mot `extern`.

2. Import d'une variable de niveau 0 : il est souvent intéressant de fractionner un programme en plusieurs fichiers sources, chacun étant compilé séparément. Le programme final est construit (monté) par l'éditeur de liens. Il est donc parfois nécessaire qu'une variable de niveau 0, définie dans un premier fichier, puisse porter aussi sur un second fichier, où elle est utilisée. Il faut alors qu'elle soit déclarée dans ce second fichier afin qu'il soit compilable isolément. On dit souvent aussi que le second fichier doit importer cette variable. Pour cela, on utilise, ici encore, une déclaration commençant par le mot `extern`. Cette déclaration peut aussi bien être placée en niveau 0 que localement à un bloc. Notez que seules les variables de niveau 0 peuvent être importées.

   Dans l'exemple qui suit (figure 3), l'objet `Xport` est défini global dans le `Fichier_1` et implicitement exportable comme nous le verrons. Il est déclaré explicitement `extern`, donc importé, dans le `Fichier_2` (la déclaration est globale, dans l'exemple choisi) :

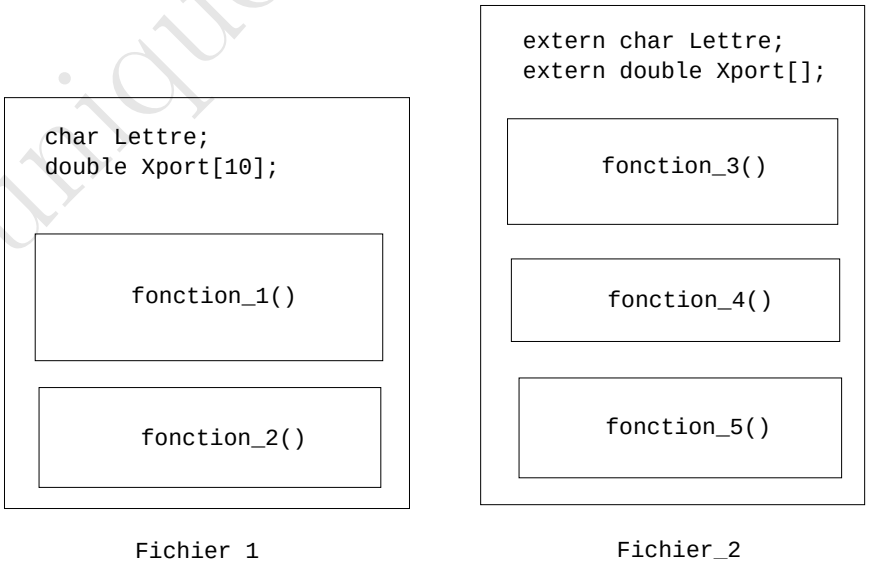

FIGURE 3 – Import - Export

Quand le compilateur traite le `Fichier_1`, il réserve la zone de stockage nécessaire au type défini (10 variables de type double). Quand il compile le `Fichier_2`, il prend simplement connaissance du

fait que `Xport` est l'adresse d'un tableau de réels `double`. Peu importe la taille, puisque le langage C ne la vérifie pas !

```
extern double Xport [];          /* declaration */
```

En conséquence, il peut y avoir autant de déclarations de la même variable (pourvu qu'elles soient toutes identiques, avec le mot `extern`) dans un ensemble de fichiers sources destinés à former un programme, mais on doit n'y trouver qu'une seule définition, et sans le mot `extern`. C'est à ce mot que le compilateur fait la différence entre définition et déclaration.

1.6 En résumé

La définition apparaît à l'endroit où la variable est créée ; on lui affecte de la mémoire et, optionnellement, une valeur (initialisation). Dans l'exemple précédent :

```
double Xport [10];                /* definition */
```

La déclaration consiste en énoncer le nom et la nature de la variable sans aucune réservation en mémoire :

```
extern double Xport [];          /* declaration */
```

2 Durée de vie des variables

2.1 Les variables internes automatiques

Les variables internes (donc de niveau supérieur ou égal à 1) sont, par défaut, qualifiées d'automatiques. Leur durée de vie est alors réduite au temps d'exécution de la fonction car elles sont créées à l'appel de la fonction dans la pile associée au programme (même si elles sont de niveau supérieur à 1), et détruites à la sortie de la fonction. Ce sont donc des variables dynamiques, éphémères. Elles doivent être initialisées à chaque appel de la fonction, sinon leurs valeurs sont imprévisibles.

2.2 Variables externes

Ces variables (de niveau 0) sont conservées en mémoire vive : leur durée de vie est celle du programme. On les qualifie de variables permanentes.

2.3 Variables internes statiques

Si l'on définit une variable interne en utilisant le mot réservé `static`, elle reste locale à son bloc de définition, mais avec une mémorisation permanente en mémoire. Le fait qu'une variable locale soit de classe `static` implique alors que :

— sa portée n'est pas altérée, et reste réduite au bloc où elle est définie, elle n'est donc pas exportable,

— sa durée de vie est celle du programme.

Une variable locale statique est créée au moment du chargement du programme, en même temps que les variables globales (dans le même segment de mémoire, sous UNIX). La valeur d'une telle variable statique interne est donc conservée d'un appel de la fonction à l'autre (comme en FORTRAN). Si dans le fichier source on initialise une variable locale statique lors de sa définition, cette valeur est attribuée une seule fois, au moment de la création de la variable. Par exemple, prenons la fonction suivante :

```
void Afficher (char Chaine[])
{
  static int Curseur = 0;

  Curseur = Curseur + printf ("%s", Chaine);
  printf ("\n  - Valeur de Curseur : %d\n", Curseur);
}
```

Si le programme appelant contient les trois lignes :

```
Afficher  ("1234");
Afficher  ("Bonjour");
Afficher ("Fini");
```

voici ce que l'on aura à l'exécution :

```
1234
  - Valeur de Curseur : 4
Bonjour
  - Valeur de Curseur : 11
Fini
  - Valeur de Curseur : 15
```

Dans ce programme, nous avons tiré parti du fait que la fonction `printf()` renvoie le nombre de caractères qu'elle a affichés. Observez que la variable interne statique `Curseur` n'est mise à zéro qu'une fois.

2.4 Variables externes statiques

Dans un fichier source, le fait de définir une variable de niveau 0 en utilisant le mot réservé `static` limite simplement sa portée à ce fichier. Une variable de niveau 0 est donc exportable par défaut, et privée si elle est définie `static`. Ce dernier cas implique alors que :

— sa portée est réduite au fichier courant selon la règle déjà vue,

— sa durée de vie continue à être celle du programme.

2.5 Fonctions statiques

Les fonctions sont des objets de niveau 0. La classe `static` s'applique à elles aussi, comme pour les variables externes. Une fonction définie en utilisant le mot réservé `static` n'est donc visible qu'à l'intérieur du fichier où elle est définie. Elle est privée à ce fichier. Elle n'est pas exportable. L'intérêt de la classe `static` pour les variables globales et les fonctions réside essentiellement dans le fait de pouvoir protéger des variables ou des fonctions sensibles, ou d'interdire l'accès à des fonctions ne faisant que des traitements partiels, très incomplets considérés isolément. Cette idée de privacité des objets d'une classe est généralisée en programmation objet, par exemple dans le langage C++, qui englobe le C ANSI. Supposons que nous ayons accès à la fonction `Publique()` (sic) et que celle-ci ait le graphe d'appel suivant (figure 4) :

Seule la fonction `Publique()` est exportable, donc accessible à d'autres fichiers sources. Ce n'est pas le cas des fonctions `Privee()`, `Perso()` et `Retraite()` qui sont des ressources de la fonction `Publique()`.

2.6 Variables et segments mémoire d'un processus Unix

Sous Unix, chaque processus qui s'exécute se voit attribuer un certain nombre de segments mémoire, chacun pouvant stocker des variables de différentes natures. À titre d'exemple, la figure 5 présente un résumé de l'emplacement mémoire des variables en fonction de cette nature.

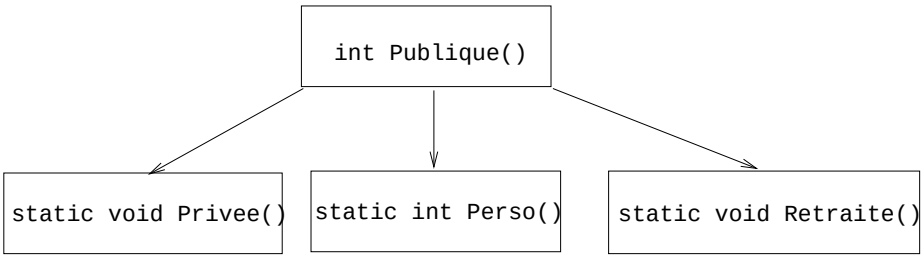

FIGURE 4 – Trois des quatre fonctions sont privées

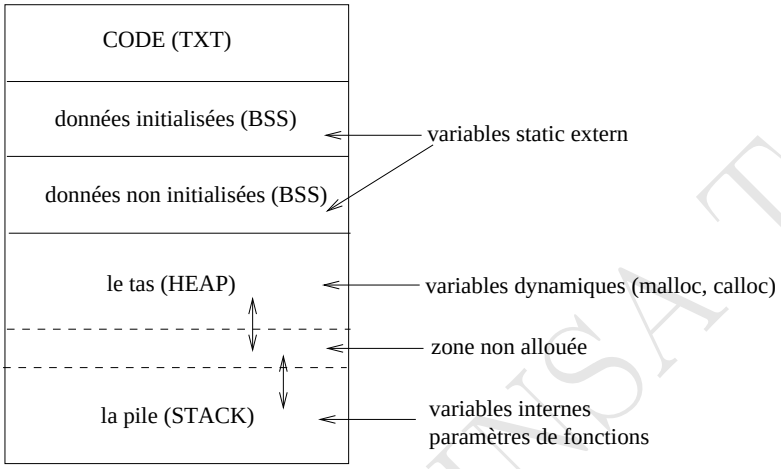

FIGURE 5 – Espace mémoire d'un processus sous Unix

2.7 Exemple comportant plusieurs fichiers sources

Voici un exemple de trois fonctions appelées par la fonction `main()`. La première est placée dans le même fichier que la fonction principale. La seconde et la troisième sont définies dans des fichiers séparés. La première et la seconde agissent sur une variable globale, un tableau, par effet de bord. Elles sont donc sans argument explicite. La troisième agit sur le même tableau dont l'adresse lui est passée en argument, explicitement.

```
/* Classes de memorisation.*/

/******************* FICHIER 1 ************************/
/* Definition de variables globales */
double Vecteur [] ={ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
                     8.0, 9.0, 10.0 };
int N;  /* Argument global des fonctions Un()
           et Deux(), il est evalue par main() */

/*************************
 * Definition de main ()  *
 *************************/
int main(void)
{
  /* prototypes des fonctions externes au bloc
  (et definies apres) ou importees */

  void Un (void);
  extern void Deux (void);
  extern void Trois (double [], int);
  extern int printf ( char *, ...);    /* Biblio standard */
  extern int scanf (char *, ...);      /* Biblio standard */
```

```
  do
  {
    printf ("\n Entrez svp un nb compris entre 1 et 10 : ");
    scanf ( "%d", &N );                /* Effet de bord sur N */
  }
  while ( N < 1 || N > 10 );

  /* Appels des fonctions */

  Un ();    /* Pas d'argument, effet de bord sur Vecteur[] */
  Deux ();  /* Pas d'argument, effet de bord sur Vecteur[] */
  Trois (Vecteur, N);                 /* Deux args, pas d'effet de bord */
  Deux ();
  Trois (Vecteur, N);
  return 0;
}

/*************************
 * Definition de   Un ()  *
 *************************/
  /* Double la valeur des N premiers arguments du
     tableau Vecteur[] */

void Un (void)
{
  int Ind;                           /* Variable interne a Un() */

  for ( Ind = 0 ; Ind < N ; Ind++ )
    Vecteur [Ind] *= 2.0 ;
}

/*******************************************************/
/******************** FICHIER 2 ************************/

/* Declaration des variables importees */
extern double Vecteur [];
extern int N;

/*************************
 * Definition de  Deux () *
 *************************/

/* Incr\'emente les N premiers elements du
   tableau Vecteur[].
   N, un int, et Vecteur[], un tableau de double,
   sont des arguments pass\'es globalement. */

void Deux (void)
{
  int Rang = 0 ;

  while ( Rang < N )
    Vecteur [Rang++] += 1.0 ;
}

/*******************************************************/
/******************** FICHIER 3 ************************/

/* Pas de variables importees */
```

```
/* Prototype des fonction importees */
extern int printf ( char *, ...);    /* Biblio standard */
extern int putchar ( char );

/*************************
 * Definition de Trois () *
 *************************/

/* Affiche les Premiers elements de Matrice[] */

void Trois ( double Matrice [], int Premiers)
{
  int Pas;

  for ( Pas = 0 ; Pas < Premiers ; Pas++ )
    printf ( "\n %8.4lf", Matrice [Pas] );
  putchar ('\n');
}

/*********************************************************/
```

Remarquez, dans cet exemple, comment sont déclarées les fonctions importées ou externes au bloc. Les règles sont les mêmes que pour les variables, à la différence que le mot réservé `extern` est facultatif car toutes les déclarations de fonctions sont de classe externe par défaut en langage C. Pour améliorer la lisibilité de vos programmes, nous vous recommandons alors de ne déclarer explicitement `extern` que les fonctions importées d'un autre fichier. Dans cet exemple, plusieurs effets de bord ont été signalés. Revenons donc au passage des arguments à une fonction. Les arguments peuvent lui être passés soit explicitement, par une liste de paramètres, comme pour `Trois()`, soit par l'intermédiaire de variables globales, comme pour `Un()` et `Deux()`. Cette dernière manière de faire est en général mauvaise pour diverses raisons, dont nous présentons les plus importantes :

- Le nom de ces variables est unique, imposé.
- On ne peut écrire ces fonctions en les détachant de leur environnement.
- La compilation séparée des fonctions implique d'inclure systématiquement la déclaration (`extern`) des variables globales importées.
- Il est difficile, voire impossible, de réaliser des tests isolés de ces fonctions.
- Les fonctions ne sont pas directement réutilisables dans d'autres programmes.
- La lisibilité des programmes est affectée. L'examen des appels ou définitions des fonctions ne renseigne pas sur les arguments utilisés.
- La programmation est faite à coup d'effets de bord. Chaque appel de fonction risque de modifier des variables globales, mais on ne sait pas lesquelles, sauf commentaires très bien faits et permanents, puisque le programme n'est pas auto-explicatif.

Tout ceci rejoint l'idée maîtresse classique : modularisez et structurez ! Les grands maîtres, DIJKSTRA, WIRTH, ARSAC, et bien d'autres, l'ont déjà dit.

3 Variables de la classe `register`

Il s'agit de la dernière classe de mémorisation, que nous présentons séparément car la mémorisation n'est pas faite en mémoire centrale. Elle n'est applicable qu'à des variables locales ou à des paramètres formels de fonctions.

3.1 Variables internes de la classe `register`

Quand on utilise le mot réservé `register` dans la définition d'une variable interne, ceci permet de suggérer au compilateur qu'il serait souhaitable de placer cette variable dans un registre du processeur. Comme l'accès aux registres est bien plus rapide que l'accès à la mémoire, les performances de traitement sont améliorées, surtout si cette variable est placée à l'intérieur d'une boucle parcourue un grand nombre de fois. La taille des registres d'une machine étant imposée, vous ne pouvez appliquer cette classe qu'à des variables de taille inférieure ou égale à celle des registres. Le nombre des registres d'une machine disponibles est limité. Seules les premières demandes de création de variables `register` sont satisfaites. Pour les autres, le mot `register` est ignoré. D'une machine à l'autre, il peut donc y avoir de grandes différences de vitesse de traitement pour un programme demandant de nombreux registres au processeur. On ne peut pas préfixer par `&` une variable de classe `register` car un registre n'a pas d'adresse en langage C. Voici un exemple pour évaluer l'amélioration de performance dans un cas très favorable :

```
int main()
{
  register int Ind;

  for ( Ind = 1 ; Ind <= 10000000 ; Ind ++)
  {}
}
```

La mesure du temps d'exécution, avec et sans le mot `register`, montre une amélioration de `40%` sur nos machines.

3.2 Paramètres de fonctions passés par les registres

Au lieu de passer les arguments par la pile du programme (par défaut), le programmeur peut demander que les arguments soient passés par des registres, comme on le fait couramment en assembleur, ceci afin d'améliorer aussi la vitesse d'exécution. Comme précédemment, une telle demande n'est satisfaite que si un registre est disponible, sinon (et sans avertissement) le paramètre est passé par la pile.

Exemple :

```
int Fct ( register int X , register char Y)
{
  register Z;        /* ou  register int Z */
  /* Corps de la fonction */
}
```

4 Initialisation des variables

L'initialisation d'une variable peut se faire au moment de la définition en faisant suivre le nom de la variable par le signe = puis la valeur souhaitée :

```
int Heure = 12;
char Lettre = 'A',
Symbole,       /* Symbole pas initialise */
Signe  = '+',
Table [12];    /* Table pas initialisee */
char * Ptr = Table;            /* Declaration de Table doit preceder */
```

En l'absence d'initialisation explicite :

— les variables externes et statiques sont mises à zéro,

— les variables automatiques et `register` peuvent avoir n'importe quelle valeur.

Cependant, nous vous recommandons de ne jamais compter sur les initialisations par défaut. Faites donc toujours vos initialisations de manière explicite. Le programme n'en sera que plus lisible.

L'initialisation des variables statiques et externes peut être faite par le compilateur. Si vous avez spécifié une valeur initiale pour une variable automatique ou `register`, il est bien évident qu'elle ne sera attribuée qu'au moment de la création de la variable, donc durant l'exécution, au moment de l'appel de la fonction. Comme une variable automatique n'est initialisée qu'à l'appel de la fonction, il est permis de faire cette initialisation par un appel de fonction comme le montre l'exemple ci-dessous pour `Lettre`. Ceci est évidemment interdit pour une variable de niveau 0 ou statique. Cependant, pour simplifier la lecture des programmes, nous vous recommandons de toujours décomposer définition et action quand celles-ci font intervenir un appel de fonction :

```
char Lettre;                /* Variable locale */
Lettre = getchar();         /* Action */
```

est donc préférable à :

```
char Lettre = getchar();
```

Les objets de la famille aggregate, c'est à dire les tableaux, les unions (redéfinition d'une même allocation de mémoire pour des variables différentes) et les structures (entendez par là les records du langage Ada, nous en reparlerons au chapitre **??**) s'initialisent à l'aide d'accolades, sauf pour la chaîne de caractères où les accolades sont facultatives. Exemples :

```
int Entiers [] = {1, 2, 3, 4, 5};  /* globales */
int DeuxDim [4] [3] =
{
  { 1, 2, 3 },
  { 8, 9, 0 },
  { 0, 1, 0 },
  { 3, 2, 1 }
};

void Une_Fonction()
{

  /* Tableaux locaux */
  double Vecteur [] = { 3.12, 14.32, -1.04e-17 };
  char Chaine[] = "D.G.E.\n A.E.I\n G.I.I.\n";

  /* Pointeur local */
  char * Ptr = "Une autre chaine...";
  /* ... */
}
```