Taurre

Zeste de savoir

Les identificateurs en langage C

12 août 2019

Table des matières

Un identificateur peut être défini comme un nom permettant de désigner, de faire référence à une entité du langage. Un exemple d'identificateur bien connu est le nom d'une variable ou d'une fonction. Toutefois, un identificateur est plus qu'un simple nom et c'est ce qui est exposé dans ce cours.

Je tiens à remercier Maëlan et Marc Mongenet pour leur relecture attentive de ce cours et leur aide dans son amélioration.

1. Première approche

Un identificateur est un nom qui permet de désigner une entité du langage. Oui, mais quelles entités ? La norme C11 en différencie huit types[1] (en comptant les membres de structures, unions ou énumérations à part), à savoir les identificateurs :

1. d'objet ;
2. de fonction ;
3. d'étiquette de structure, union ou énumération, qui correspond au nom que vous donnez à votre structure, union ou énumération ;
4. de membre de structure, d'union ou d'énumération ;

5. de définition de type (`typedef`) ;
6. d'étiquette, utilisée pour l'instruction de saut `goto` ;
7. de macro ;
8. de paramètre de macrofonction.

En C, un objet est une zone mémoire pouvant contenir des données[2].

Afin d'illustrer cette énumération, voici un code déclarant un identificateur pour chacune des entités présentées ci-dessus.

```c
#define
    identificateur_de_macro(identificateur_de_parametre_de_macro)

struct identificateur_d_etiquette_de_structure {
        int identificateur_de_membre_de_structure;
};

typedef int identificateur_de_definition_de_type;

void
identificateur_de_fonction(void)
{
        int identificateur_d_objet;

identificateur_d_etiquette:
        ;
}
```

Je ne parlerai pas des identificateurs de macro et de paramètre de macrofonction dans la suite du tutoriel, ces derniers n'existant plus après traitement du code par le préprocesseur.

2. Portée, espaces de noms et masquage

Vous avez peut-être remarqué que j'ai utilisé le terme *déclaration* dans la présentation, ce n'est pas anodin, il s'agit d'un concept fondamental du langage C permettant la création d'identificateurs.

1. ISO/IEC 9899 :201x, doc. N1570, avril 2011, § 6.2.1, al. 1, p. 35.
2. *Ibid.*, § 3.15, al. 1, p. 6.

2.1. La notion de portée

Une déclaration déclare un identificateur, c'est-à-dire qu'elle le rend utilisable, visible pour la suite du programme. On dit qu'une déclaration confère une *portée* à l'identificateur, c'est-à-dire une portion du programme où il sera utilisable. Il existe quatre types de portée[3] :

1. au niveau d'un bloc ;
2. au niveau d'un fichier ;
3. au niveau d'une fonction ;
4. au niveau d'un prototype.

Cependant, je n'aborderai pas la portée au niveau d'un prototype dans la suite de ce cours, étant donné le peu d'intérêt de cette dernière.

2.2. Au niveau d'un bloc

Une portée au niveau d'un bloc signifie qu'un identificateur est utilisable, visible de sa déclaration jusqu'à la fin du bloc dans lequel il est déclaré. Ainsi, dans le code suivant, L'identificateur `n` ne peut pas être utilisé dans le bloc de la fonction `g()` car il a une portée limitée au bloc de la fonction `f()`.

```
1  void
2  f(void)
3  {
4          int n = 10;
5  }
6
7
8  void g(void)
9  {
10         n = 20; /* Incorrect */
11 }
```

De même, le code suivant est erroné car au moment de la déclaration de l'identificateur `p`, l'identificateur `a` n'est pas encore déclaré, il est donc utilisé en dehors de sa portée.

```
1  int *p = &a; /* Incorrect */
2  int a = 10;
```

2.3. Au niveau d'un fichier

Une portée au niveau d'un fichier signifie qu'un identificateur est utilisable, visible de sa déclaration jusqu'à la fin du fichier dans lequel il est déclaré. Pour obtenir un identificateur ayant une portée au niveau d'un fichier, il est nécessaire de le déclarer en dehors de tout bloc, par exemple comme ceci.

```
1  int n;
2
3  void
4  f(void)
5  {
6          n = 10;
7  }
8
9  void
10 g(void)
11 {
12         n = 20;
13 }
```

Dans ce code, l'identificateur `n` a une portée au niveau du fichier et peut par conséquent être aussi bien utilisé dans la fonction `f()` que dans la fonction `g()`.

2.4. Au niveau d'une fonction

Une portée au niveau d'une fonction signifie qu'un identificateur est utilisable, visible dans toute la fonction où il est déclaré et ce, peu importe la position de sa déclaration. Cette portée est propre aux identificateurs d'étiquette utilisés par l'instruction de saut `goto`.

```
1  int
2  main(void)
3  {
4          int n = 0;
5
6  test:
7          if (!n) {
8                  goto dix;
9          } else {
10                 goto fin;
11         }
12 dix:
13         n = 10;
14         goto test;
15 fin:
```

```
16          return 0;
17  }
```

Comme vous le voyez, les identificateurs `dix` et `fin` peuvent être utilisés avant leur déclaration, car ils ont une portée au niveau de la fonction `main()`.

2.5. La notion d'espace de noms

Le concept d'*espace de noms* n'est pas évident à définir, mais est par contre très facile à comprendre à l'aide d'un exemple. Sachez tout d'abord qu'il existe quatre espaces de noms[4] :

1. un dédié aux identificateurs d'étiquettes ;
2. un dédié aux identificateurs d'étiquettes de structures, unions ou énumérations ;
3. un dédié aux identificateurs de membres de structures ou unions ;
4. un dédié à tous les autres identificateurs.

i

Avant la normalisation du langage en 1989, les champs de structures ou d'unions ne disposaient pas forcément d'un espace de noms distinct. Cela explique pourquoi certaines structures de la bibliothèque standard préfixent le nom de leur champ (c'est le cas de la structure `tm` définie dans l'en-tête `<time.h>` par exemple).

Ensuite, comme convenu, voici un exemple.

```
1   int
2   main(void)
3   {
4           struct test {
5                   int test;
6           };
7           struct test test;
8
9           goto test;
10  test:
11          test.test = 10;
12          return 0;
13  }
```

Comme vous le voyez, il y a quatre identificateurs déclarés avec le nom `test` :

1. un identificateur d'étiquette de structure (`struct test`, ligne 4) ;
2. un identificateur de membre de structure (`int test`, ligne 5) ;
3. un identificateur d'objet (`struct test test`, ligne 7) ;

4. un identificateur d'étiquette (`test:`, ligne 10).

Ces quatre identificateurs ont tous une portée au niveau du bloc de la fonction `main()`. Ce code ne pose pourtant aucun problème, tout simplement parce que ces derniers appartiennent à quatre espaces de noms différents. Tout risque de confusion est évité de par :

— le contexte d'utilisation de l'identificateur (l'instruction `goto` attend un identificateur d'étiquette) ;
— l'utilisation de mots-clés (`struct`, `union` ou `enum` pour désigner l'identificateur d'étiquette d'une structure, union, ou énumération) ;
— l'utilisation d'opérateurs (l'opérateur `.` ou `->` pour accéder aux membres d'une structure et/ou d'union) ;
— la syntaxe de la déclaration (par exemple les deux points suivant la déclaration d'un identificateur d'étiquette).

2.6. La notion de masquage

Une règle importante à retenir est qu'il n'est pas possible de déclarer deux identificateurs de même nom et de même espace de noms dans la même portée[5]. Ainsi, le code suivant est incorrect car il déclare deux identificateurs d'objet `x` dans le même espace de noms et dans la même portée.

```
1  int
2  main(void)
3  {
4          int x;
5          int x; /* Incorrect */
6
7          return 0;
8  }
```

Maintenant, que se passe-t-il lorsque l'on déclare deux identificateurs de même nom et de même espace de noms, mais dans des portées différentes ? Autrement dit, que se passe-t-il dans ce cas ci ?

```
1  #include <stdio.h>
2
3  int n = 10;
4
5
6  int
7  main(void)
8  {
9          int n = 20;
10
11         printf("%d\n", n);
```

```
12          return 0;
13  }
```

En fait, dans une telle hypothèse, c'est l'identificateur ayant la portée la plus faible qui sera privilégié. On dit qu'il *masque* celui ou ceux ayant une portée plus élevée[6] (en l'occurrence celui ayant une portée au niveau d'un fichier). Je dis : « celui ou ceux », car les identificateurs déclarés dans un sous-bloc ont une portée plus faible que ceux déclarés dans le bloc supérieur.

```
1   #include <stdio.h>
2
3   int n = 10;
4
5
6   int
7   main(void)
8   {
9           int n = 20;
10
11          if (n == 20) {
12                  int n = 30;
13
14                  printf("%d\n", n);
15          }
16          return 0;
17  }
```

Dans cet exemple, il y a trois identificateurs d'objet portant tous les trois le nom `n` :

1. le premier a une portée au niveau du fichier ;
2. le second au niveau du bloc de la fonction `main()` ;
3. et le troisième au niveau du bloc du `if`.

L'identificateur ayant une portée au niveau du fichier est donc masqué par celui ayant une portée au niveau du bloc de la fonction `main()`, qui est lui-même masqué par celui ayant une portée au niveau du bloc du `if`. Si l'on exécute ce petit programme, il affichera donc `30`.

Notez que le masquage n'opère qu'une fois l'identificateur de portée plus faible déclaré. Ainsi, dans cet exemple :

```
1   #include <stddef.h>
2   #include <stdio.h>
3
4   int x;
5
6
7   int
```

```
 8 main(void)
 9 {
10         size_t x[sizeof x] = { sizeof x };
11
12         printf("%zu %zu\n", x[0], sizeof x);
13         return 0;
14 }
```

L'expression `sizeof x` utilisée pour déterminer la taille du tableau `x` va être évaluée en utilisant l'identificateur ayant une portée au niveau du fichier, le tableau n'étant pas encore déclaré à ce moment. Toutefois, la seconde expression `sizeof x`, utilisée pour initialiser le premier membre du tableau va, elle, utiliser l'identificateur ayant une portée au niveau du bloc de la fonction `main()`, ce dernier étant désormais déclaré.

3. Liaisons et définitions

Dans le chapitre précédent, nous avons entre autres vu que les identificateurs étaient confinés à une portée et que cette dernière ne pouvait s'étendre au delà d'un fichier. Cependant, si cela s'arrêtait là, il ne serait pas possible d'utiliser des objets ou des fonctions d'autres fichiers. Autrement dit, l'exemple ci-dessous serait incorrect et il serait nécessaire de n'utiliser qu'un seul fichier source, ce qui serait assez peu commode.

— autre.c

```
1 int
2 f(void)
3 {
4         return 1;
5 }
```

— main.c

```
1 int f(void);
2
3 int
4 main(void)
5 {
6         f();
7         return 0;
8 }
```

3. IISO/IEC 9899 :201x, doc. N1570, avril 2011, § 6.2.1, al. 2, p. 35.
4. *Ibid.*, § 6.2.3, al. 1, p. 37.
5. *Ibid.*, § 6.2.1, al. 2, p. 35.
6. *Ibid.*, § 6.2.1, al. 4, p. 36.

3.1. La notion de liaison

Heureusement, il existe une solution : la notion de *liaison*. Chaque identificateur peut disposer d'une liaison qui peut être de deux types : externe ou interne[7]. Grâce à cette notion, il est possible de considérer un groupe d'identificateurs comme faisant référence à un même objet ou à une même fonction. En fait, elle permet de préciser que :

— tous les identificateurs avec liaison *externe* d'un même *programme* font référence au même objet ou à la même fonction[8] ;
— tous les identificateurs avec liaison *interne* d'un même *fichier* font référence au même objet ou à la même fonction[8].

Ainsi, si je reprends l'exemple donné au début de ce chapitre et que l'on considère que tous les identificateurs de fonction `f()` ont une liaison externe, on peut en déduire qu'en fait, ils font tous référence à la même fonction : celle du fichier `autre.c`.

— autre.c

```c
int
f(void)
{
        return 1;
}
```

— main.c

```c
int f(void);

int
main(void)
{
        f(); /* Retournera 1 */
        return 0;
}
```

La même logique peut être appliquée pour une liaison interne, mis à part que le regroupement se limite à un fichier. En conséquence, dans l'exemple ci-dessous, si l'on considère tous les identificateurs de fonction `f()` comme ayant une liaison interne, tous ceux situés dans le fichier `main.c` font référence à la fonction de ce fichier, alors que celui du fichier `autre.c` fait référence à celle située en son sein.

— autre.c

```c
int
f(void)
{
```

```
4         return 1;
5 }
```

— main.c

```
1  int
2  f(void)
3  {
4          return 2;
5  }
6
7
8  int
9  main(void)
10 {
11         f(); /* Retournera 2 */
12         return 0;
13 }
```

3.2. Conditions d'attribution

Maintenant que vous connaissez la notion de liaison, il reste encore à déterminer dans quelles conditions cette dernière est attribuée à un identificateur. En fait, la présence d'une liaison et son type sont déterminés par la position de la déclaration de l'identificateur ainsi que par l'utilisation des mots-clés `extern` et `static`. Concrètement, cela se détermine suivant les règles exposées ci-dessous.

Un identificateur de fonction ou d'objet ayant une portée au niveau d'un fichier a une liaison externe[9] [10], sauf si sa déclaration est précédée du mot-clé `static`, auquel cas il a une liaison interne[11].

```
1  int a;          /* Liaison externe */
2  static int b;   /* Liaison interne */
3
4  void f(void);          /* Liaison externe */
5  static void g(void);   /* Liaison interne */
```

Un identificateur d'objet déclaré à l'intérieur d'un bloc n'a pas de liaison sauf s'il est précédé du mot-clé `extern` (voyez la règle suivante)[12].

```
1 {
2         int a;  /* Pas de liaison */
3 }
```

Un identificateur d'objet ou de fonction dont la déclaration est précédée du mot-clé `extern` a une liaison externe sauf si une déclaration du même identificateur la précède, auquel cas il a la même liaison que ce dernier[13].

Dans le cas où une déclaration d'un identificateur de fonction n'est précédée, ni du mot-clé `static`, ni du mot-clé `extern`, le mot-clé `extern` est implicitement ajouté[14].

Ces règles peuvent paraître quelque peu indigestes, aussi, voici un exemple illustrant chacune de ces dernières.

```c
/*
 * « a » est un identificateur d'objet déclaré en dehors de tout bloc.
 * Il a donc une liaison externe.
 */
int a;

/*
 * « b » est un identificateur d'objet déclaré en dehors de tout bloc.
 * Sa déclaration est précédée du mot-clé « static ».
 * Il a donc une liaison interne.
 */
static int b;

/*
 * « c » est un identificateur d'objet déclaré en dehors de tout bloc.
 * Sa déclaration est précédée du mot-clé « extern ».
 * Aucune déclaration du même identificateur ne le précède.
 * Il a donc une liaison externe.
 */
extern int c;

/*
 * « f » est un identificateur de fonction.
 * Sa déclaration n'est pas précédée du mot-clé « extern » ou « static ».
 * Dès lors, il faut faire comme si elle était précédée du mot-clé « extern ».
 * Aucune déclaration du même identificateur ne le précède.
 * Il a donc une liaison externe.
 */
void f(void);

/*
 * « g » est un identificateur de fonction.
```

```
33  * Sa déclaration est précédée du mot-clé « static ».
34  * Il a donc une liaison interne.
35  */
36 static void g(void);
37
38 /*
39  * « h » est un identificateur de fonction.
40  * Sa déclaration est précédée du mot-clé « extern ».
41  * Aucune déclaration du même identificateur ne le précède.
42  * Il a donc une liaison externe.
43  */
44 extern void h(void);
45
46
47 int
48 main(void)
49 {
50         /*
51          * « a » est un identificateur d'objet déclaré à
    l'intérieur d'un bloc.
52          * Sa déclaration est précédée du mot-clé « extern ».
53          * Il existe déjà une autre déclaration de celui-ci avec
    liaison externe.
54          * Il a donc une liaison externe.
55          */
56         extern int a;
57
58         /*
59          * « b » est un identificateur d'objet déclaré à
    l'intérieur d'un bloc.
60          * Sa déclaration est précédée du mot-clé « extern ».
61          * Il existe déjà une autre déclaration de celui-ci avec
    liaison interne.
62          * Il a donc une liaison interne.
63          */
64         extern int b;
65
66         /*
67          * « c » est un identificateur d'objet déclaré à
    l'intérieur d'un bloc.
68          * Sa déclaration n'est pas précédée du mot-clé « extern ».
69          * Il n'a donc pas de liaison.
70          */
71         int c;
72
73         /*
74          * « d » est un identificateur d'objet déclaré à
    l'intérieur d'un bloc.
75          * Sa déclaration est précédée du mot-clé « extern ».
76          * Aucune déclaration du même identificateur ne le précède.
```

```
77          * Il a donc une liaison externe.
78          */
79         extern int d;
80
81         /*
82          * « g » est un identificateur de fonction.
83          * Sa déclaration n'est pas précédée du mot-clé « extern ».
84          * Dès lors, il faut faire comme si elle était précédée du
    mot-clé « extern ».
85          * Il existe déjà une autre déclaration de celui-ci avec
    liaison interne.
86          * Il a donc une liaison interne.
87          */
88         void g(void);
89
90         return 0;
91 }
```

Le mot-clé `static` ne peut être utilisé, pour modifier la liaison d'un identificateur, qu'en dehors de tout bloc et ce, aussi bien pour les identificateurs d'objet que les identificateurs de fonction[15] [16].

3.3. La notion de définition

Je vous ai dit que la notion de liaison permettait de grouper des identificateurs et de les considérer comme faisant référence au même objet ou à la même fonction. Je vous ai également dit que tous les identificateurs avec liaison *externe* d'un même *programme* font référence au même objet ou à la même fonction et que tous les identificateurs avec liaison *interne* d'un même *fichier* font référence au même objet ou à la même fonction. Cependant, il y a un corollaire qui découle de ces deux règles : il ne peut exister qu'*un seul objet* ou qu'*une seule fonction* qui puisse être référencé par le groupe d'identificateurs.

Au fond, c'est assez logique. Prenez l'exemple ci-dessous, l'identificateur de fonction `f()` déclaré dans le bloc de la fonction `main()` a une liaison interne. Cependant, laquelle des deux fonctions désigne-t-il ? La première ? La deuxième ? Les deux ?

```
1 static int
2 f(void)
3 {
4         return 1;
5 }
6
7 static int
8 f(void)
```

```
{
        return 2;
}


int
main(void)
{
        static int f(void);

        f();
        return 0;
}
```

Il est impossible de le dire, il faudrait qu'il n'existe qu'une seule fonction ou, dit plus formellement, qu'il n'y ait qu'une seule *définition* de la fonction `f()`. Qu'est-ce qu'une définition ? C'est ce que nous allons voir tout de suite.

3.3.1. Les identificateurs de fonction

Une définition d'un identificateur de fonction est une déclaration qui comporte le corps de la fonction[17]. Autrement dit, dans le code ci-dessous, le premier élément est une déclaration de l'identificateur de fonction `f()` alors que le deuxième est une définition de l'identificateur de fonction `f()`, car il comporte le corps de celle-ci.

```
/* Déclaration */
int f(void);

/* Définition */
int
f(void)
{
        return 1;
}
```

3.3.2. Les identificateurs d'objet

Une définition d'un identificateur d'objet est une déclaration qui alloue l'objet qu'il référence[17]. Vous voilà bien peu avancé me direz-vous... Heureusement, il y a une règle simple et absolue pour différencier une déclaration et une définition d'un identificateur d'objet : une déclaration d'un identificateur d'objet, en dehors de tout bloc, comportant une initialisation est une définition[18]. Dans tous les autres cas, il s'agit d'une déclaration.

```c
int a;          /* Déclaration */
static int b;   /* Déclaration */
extern int c;   /* Déclaration */
int d = 10;     /* Définition */
```

Cependant, il y a une (petite) subtilité : les déclarations d'identificateurs d'objet, en dehors de tout bloc, à l'exception de celles précédées du mot-clé `extern`, sont appelées des *définitions potentielles*. Et, dans le cas où un fichier comprend une ou plusieurs définitions potentielles d'un identificateur d'objet mais aucune définition de cet identificateur, une définition est implicitement incluse au début du fichier avec un initialiseur valant zéro[19].

Rassurez-vous, nous allons revoir cela en douceur. Avant toute chose, il est nécessaire de bien différencier une déclaration, une définition potentielle et une définition d'un identificateur d'objet. Pour ce faire, voici un exemple simple.

```c
/*
 * Cette déclaration ne comporte pas d'initialisation.
 * Elle n'est pas précédée du mot-clé « extern ».
 * Il s'agit donc d'une définition potentielle.
 */
int n;

/*
 * Cette déclaration comporte une initialisation.
 * Il s'agit donc d'une définition.
 */
extern int n = 10;

/*
 * Cette déclaration ne comporte pas d'initialisation.
 * Elle n'est pas précédée du mot-clé « extern ».
 * Il s'agit donc d'une définition potentielle.
 */
static int n;

/*
 * Cette déclaration ne comporte pas d'initialisation.
 * Elle est précédée du mot-clé « extern ».
 * Il s'agit donc d'une déclaration.
 */
extern int n;
```

Ensuite, reprenons cette règle pas à pas à l'aide du code ci-dessous.

```c
int n;

int
main(void)
{
        return n;
}
```

Comme vous le voyez, nous avons un fichier comprenant une définition potentielle de l'identificateur d'objet `n`, mais aucune définition de cet identificateur. Ce que dit l'obscure règle que je vous ai présentée auparavant, c'est que dans le cas où un fichier comprend une ou plusieurs définitions potentielles d'un identificateur mais aucune définition de cet identificateur (ce qui est le cas de notre fichier), une définition est implicitement inclue au début de ce fichier avec un initialiseur valant zéro. Autrement dit, appliquée à notre exemple, cela donne ceci.

```c
/* Définition implicite */
int n = 0;
int n;

int
main(void)
{
        return n;
}
```

3.3.3. Formalisation de l'interdiction

Maintenant que nous avons vu la notion de définition, il m'est possible de formaliser ce que je vous ai dit au début de la présentation de cette notion : il ne peut exister qu'un seul objet ou qu'une seule fonction qui puisse être référencée par un groupe d'identificateur. Ou, dit de manière plus formelle :

— il ne peut y avoir qu'une seule définition d'un même identificateur avec liaison externe dans tout le programme[20] ;
— il ne peut y avoir qu'une seule définition d'un même identificateur avec liaison interne dans un même fichier[21] ;

i

Certains compilateurs (gcc pour ne citer que lui) sont par défaut capables de gérer certains cas de définitions multiples. Sachez cependant qu'il s'agit d'une extension non standard. Dans le cas de gcc, il est possible de désactiver cette extension en utilisant l'option `-fno-common`.

Le code ci-dessous est donc incorrect car il comporte plus d'une définition avec liaison interne de l'identificateur `n`.

```c
static int n = 10;
static int n = 20;


int
main(void)
{
        return n;
}
```

De même, le code qui suit est faux car il existe plus d'une définition avec liaison externe de l'identificateur `n` dans tout le programme (n'oubliez pas la définition implicite !).

— autre.c

```c
int n = 10;
```

— main.c

```c
int n;

int
main(void)
{
        return n;
}
```

Notez enfin que si un identificateur apparaît dans un fichier avec à la fois une liaison externe et interne, le résultat est indéterminé[22].

— autre.c

```c
int
f(void)
{
        return 1;
}
```

— main.c

```c
int f(void);

static int
f(void)
{
        return 2;
}


int
main(void)
{
        f();
        return 0;
}
```

Dans cet exemple, l'identificateur `f()` du fichier `main.c` a à la fois une liaison externe et interne. Il est donc impossible de dire à quelle fonction il fait référence.

3.4. En bref

Que retenir de ce chapitre si ce n'est qu'il est affreusement théorique et complexe ? En fait, il est possible d'en déduire une méthode générale afin de partager des variables ou des fonctions entre plusieurs fichiers source.

Étant donné que les fichiers d'en-têtes sont très souvent inclus dans plusieurs fichiers (pensez à ceux de la bibliothèque standard par exemple), ces derniers ne doivent contenir que des déclarations. En effet, si ce n'est pas le cas, vous allez vous retrouver avec des définitions multiples (explicites ou implicites) et, dès lors, rencontrer des erreurs lors de la compilation.

Les fichiers source, quant à eux, recueillent donc les définitions. Ainsi, lorsque vous souhaitez utiliser une ou plusieurs variables ou fonctions définies dans un autre fichier, vous incluez le ou les fichiers d'en-tête comprenant leurs déclarations dans les fichiers source où vous souhaitez les utiliser. Cette méthode a l'avantage d'éviter d'avoir à réécrire toutes les déclarations dans chaque fichier.

L'exemple ci-dessous illustre ce qui vient d'être exposé.

— autre.h

```c
#ifndef AUTRE_H
#define AUTRE_H

extern int n;           /* Déclaration */
extern void setn(int);  /* Déclaration */

#endif /* !AUTRE_H */
```

— autre.c

```c
/* Inclusion des déclarations */
#include "autre.h"

/* Définition potentielle */
int n;

/* Définition */
void
setn(int a)
{
        n = a;
}
```

— main.c

```c
/* Inclusion des déclarations */
#include <stdio.h>

#include "autre.h"

int
main(void)
{
        printf("%d\n", n); /* 0 */
        setn(99);
        printf("%d\n", n); /* 99 */
        return 0;
}
```

7. ISO/IEC 9899 :201x, doc. N1570, avril 2011, § 6.2.2, al. 1, p. 36.
8. *Ibid.*, p. 36, § 6.2.2, al. 2.
9. *Ibid.*, p. 37, § 6.2.2, al. 5.
10. *Ibid.*, § 6.2.2, al. 5, p. 37.
11. *Ibid.*, § 6.2.2, al. 3, p. 36.
12. *Ibid.*, § 6.2.2, al. 6, p. 37.
13. *Ibid.*, § 6.2.2, al. 4, p. 37.
14. *Ibid.*, § 6.2.2, al. 5, p. 37.
15. *Ibid.*, § 6.2.4, al. 3, p. 38.
16. *Ibid.*, § 6.7.1, al. 7, p. 110.
17. *Ibid.*, § 6.7, al. 5, p. 108.
18. *Ibid.*, § 6.9.2, al. 1, p. 158.
19. *Ibid.*, § 6.9.2, al. 2, p. 158.
20. *Ibid.*, § 6.9, al. 5, p. 155.
21. *Ibid.*, § 6.9, al. 3, p. 155.
22. *Ibid.*, § 6.2.2, al. 7, p. 37.

4. Les noms

Nous allons à présent terminer notre tour d'horizon des identificateurs avec un sujet plus léger et plus simple : le nom des identificateurs.

4.1. Caractères utilisables

Un nom est composé d'une suite de lettres et de chiffres. Oui, mais quelles lettres et quels chiffres ? La liste exhaustive nous est donnée par la norme[23].

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 _
```

Sachez qu'un nom ne peut pas commencer par un chiffre, il doit obligatoirement débuter par une lettre ou par un *underscore*[24].

4.2. Noms réservés par le langage

Nous savons désormais de quels caractères peuvent être composés nos noms. Cependant, tous les noms ne sont pas utilisables. En effet, certains sont réservés par le langage C lui-même et ne sont donc pas disponibles[25].

```
auto            if              unsigned
break           inline          void
case            int             volatile
char            long            while
const           register        _Alignas
continue        restrict        _Alignof
default         return          _Atomic
do              short           _Bool
double          signed          _Complex
else            sizeof          _Generic
enum            static          _Imaginary
extern          struct          _Noreturn
float           switch          _Static_assert
for             typedef         _Thread_local
goto            union
```

Il est à noter que certaines implémentations réservent aussi les mots `asm` et `fortran`. Il est donc également préférable de les éviter.

4.3. Noms réservés par la bibliothèque standard

À côté des noms réservés par le langage lui-même, il y a ceux réservés par la bibliothèque standard. En fait, tous les noms de fonctions (par exemple `printf()`) ou de variables (par exemple `errno`) utilisés par celle-ci sont à éviter, même si vous n'incluez pas l'en-tête les utilisant. Renseignez-vous sur les différents en-têtes pour obtenir les noms qu'ils emploient.

En plus de cela, la bibliothèque standard réserve certains types de noms dans des portées particulières. Ainsi, sont interdits :

— les noms commençant par un *underscore* et une lettre majuscule ou commençant par deux *underscores* et ce, peu importe leur portée[26] ;
— les noms commençant par un *underscore* et ayant une portée au niveau d'un fichier[26].

Afin de bien cerner cette interdiction, voici un petit code d'exemple.

```c
/* Interdit */
#define _HELLO

/* Interdit */
#define __HELLO

/* Interdit */
#define __hello

/* Interdit car il a une portée au niveau d'un fichier */
#define _hello

/* Interdit pour les mêmes motifs */
struct _structure {
        /* Permis car il s'agit d'un membre de structure */
        int _membre;
};


int
main(void)
{
        /* Permis car il a une portée au niveau d'un bloc */
        int _variable;

        /* Interdit car « auto » est un mot-clé réservé du langage
           */
        int auto;

        return 0;
}
```

Remarquez enfin que dans le cas de l'en-tête `<errno.h>`, les noms de macro commençant par un `E` et un chiffre ou une lettre majuscule ne doivent pas non plus être employés[27] de même

pour l'en-tête `<signal.h>` et les noms de macro commençant par `SIG` ou `SIG_` et une lettre majuscule[28].

Voilà qui termine mon exposé sur les identificateurs. J'espère que vous y voyez désormais plus clair et que vous jonglez avec les portées et les liaisons.

23. ISO/IEC 9899 :201x, doc. N1570, avril 2011, § 6.4.2.1, al. 1, p. 59.
24. *Ibid.*, § 6.4.2.1, al. 2, p. 59.
25. *Ibid.*, § 6.4.1, al. 1., p. 58.
26. *Ibid.*, § 7.1.3, al. 1, p. 182.
27. *Ibid.*, § 7.5, al. 4, p. 205.
28. *Ibid.*, § 7.14, al. 4, p. 265.