
C est ce que l'on a coutume d'appeler un langage typé. Le type des données manipulées est déterminé au moment de la compilation déjà. Un certain nombre de types de données sont prédéfinis par le langage : ce sont les **types de base**. Ces types de base peuvent être utilisés pour composer d'autres types, formés de collections d'identificateurs de types de base: il s'agit alors de **types dérivés**. Il est également possible à l'utilisateur de définir de nouveaux types, comme dans la majorité des langages typés.

5.1 *Types de base*

C implémente quelques types de base, dont les caractéristiques peuvent dépendre de l'implémentation sur une machine. Pour **chaque implémentation**, il est possible de connaître les détails par l'examen du fichier `<limits.h>`, qui spécifie l'implémentation utilisée. Un exemple d'un tel fichier est donnée ci-après. Il s'agit d'une implémentation pour un Macintosh basé sur un CPU de la famille 32 bit Motorola 680X0 ($X > 2$).

Pour un utilisateur de microcontrôleurs ou de DSP, ce fichier peut s'avérer extrêmement important; de manière plus générale, toute personne travaillant à bas ou très bas niveau, et devant éventuellement interfacer son programme avec du langage d'assemblage aura intérêt à posséder la table correspondant à l'unité centrale de son choix en mémoire !

```
#define CHAR_BIT          8
#define MB_LEN_MAX       1

#define SCHAR_MIN        (~127)
#define SCHAR_MAX        127
#define UCHAR_MAX        255
#define CHAR_MIN         (~127)
#define CHAR_MAX         127

#define SHRT_MIN         (~32767)
#define SHRT_MAX         32767
#define USHRT_MAX        0xFFFF

#if __SC__                /* THINK C++ */

#define INT_MIN          (~2147483647)
#define INT_MAX          2147483647
#define UINT_MAX         0xFFFFFFFF

#else                      /* THINK C */

#if __option(int_4)
#define INT_MIN          (~2147483647)
#define INT_MAX          2147483647
#define UINT_MAX         0xFFFFFFFF
#else
#define INT_MIN          (~32767)
#define INT_MAX          32767
#define UINT_MAX         0xFFFF
#endif
#endif

#define LONG_MIN         (~2147483647L)
#define LONG_MAX         2147483647L
#define ULONG_MAX        0xFFFFFFFFL
```

Ainsi, sur une machine 64 bit, certaines de ces définitions (par exemple `INT_MAX`) peuvent changer pour refléter les possibilités étendues du CPU.

5.1.1 *Type void*

Le terme `void` signifie que l'identificateur correspondant n'a pas de type associé. (`void` = rien). Le type `void` est un type important en C; en particulier, il sert à définir une procédure, qui en C (par opposition à PASCAL, notamment) est une fonction qui ne retourne «rien».

5.1.2 *Type short, int, long*

Ces trois types définissent tous trois des entiers de taille pouvant différer selon les machines. Le langage (dans un souci d'efficacité relativement à la machine utilisée) ne spécifie pas la dimension de ces trois types, non plus que leurs dimensions relatives, si ce n'est par la relation

$$\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

`sizeof()` est un opérateur C permettant de calculer la taille en bit d'une variable ou d'un type. On rencontre fréquemment, sur les machines 32 bit actuelles, les valeurs `short` = 16 bit, `int` = 32 bit, et `long` = 64 bit. Sans autre précision, `short`, `int` et `long` sont munis d'un signe. Il est possible, pour ces trois types de base, de spécifier `unsigned short`, `unsigned int` ou `unsigned long`.

5.1.3 *Type float, double, long double*

Ces trois types implémentent tous un type de variable réel en virgule flottante. Là encore, la précision de ces trois types n'est pas un objet de spécification du langage; il en va des nombres en virgule flottante comme des entiers. Les nombres réels non pourvus d'un signe (`unsigned float`, `unsigned double`, `unsigned long double`) ne sont pas implémentés.

5.1.4 *Type char*

Le type `char` est un cas particulier de nombres entiers. En ce sens, il est entièrement compatible avec le type `int` avec lequel il peut être converti de manière implicite. Le type `char` représente le jeu de caractères disponible sur le système considéré : USASCII sur un système d'exploitation de type MS-DOS ou UNIX, EBCDIC sur un système comme MVS ou RPG, etc...

Dans la très grande majorité des cas, une variable de type `char` occupe un byte (ou octet), soit 8 bits.

Un type `char` peut être muni d'un signe ou non. Dans ce cas, il s'agit d'un `signed char`, ou d'un `unsigned char`. Par défaut, le type `char` est muni d'un signe, pour correspondre à l'implémentation des caractères USASCII à 7 bits, le huitième bit étant non significatif (de cette manière, le type `char` correspond à un type **byte** (non implémenté), qui serait un entier codé sur 8 bit).

5.1.5 *Type enum*

Un type énuméré déclare un ensemble de constantes entières symboliques appelées énumérateurs. Les éléments du type énuméré diffèrent d'éléments constants en ce sens qu'ils n'occupent aucune place mémoire. Il est donc illégal de vouloir calculer l'adresse d'un énumérateur, car cette adresse n'existe pas. On peut associer des valeurs à chaque énumérateur si on le désire. Par défaut, le premier énumérateur prend la valeur 0, et chaque énumérateur prend la valeur du précédent augmentée de 1. Au besoin, on peut imposer des valeurs explicitement à certains (ou à tous) les énumérateurs.

```
enum { false, true }; // false = 0, true = 1
enum { false, fail = 0, true, pass = 1 }; // false == fail == 0
// true == pass == 1
```

Il est possible de définir un nom pour une énumération. Ce nom tient lieu de nom de type. L'avantage est que les mécanismes de contrôle de type du compilateur C peuvent dès lors être utilisés pour garantir que l'on associe des valeurs correctes aux variables. Ainsi :

```
enum ProcessStatus { idle, waiting, running };
enum Boolean { false, true };

enum ProcessStatus Etat;
enum Boolean VraiFaux;

VraiFaux = 0; // Faux, Boolean = int
VraiFaux = waiting; // Faux Boolean = ProcessStatus
Etat = idle; // Correct.
```

Un type énuméré peut être converti implicitement en un entier, mais pas l'inverse:

```
VraiFaux = 0; // Faux, Boolean = int
int EtatEntier = VraiFaux;
// Ok, car conversion implicite int = enum
```

L'utilisation de types énumérés est également intéressante dans l'optique de la documentation du programme.

5.1.6 *Type booléen*

Par contraste à beaucoup de langages, C ne supporte pas le type booléen. Ceci a généré beaucoup d'incompatibilités entre divers programmes, chacun ayant défini un type booléen qui lui était propre.

```
#define boolean int
#define bool int
#define FALSE 0
#define TRUE 1
```

```
typedef unsigned char    bool;
typedef enum    bool_t
{
    false,
    true
}    boolean;
```

Et ainsi de suite. La définition dans le standard ANSI d'un type booléen comme type standard est actuellement en discussion (type `bool`). Dans l'intervalle, il faudra continuer à vivre avec le fait qu'un entier non nul (ou toute quantité pouvant être convertie en un entier, et dont la valeur après conversion est non nulle) représente en C une valeur logique TRUE, alors que la valeur opposée est représentée par une valeur nulle. Ainsi :

```
int    tstVal = 1000;
while (!(tstVal--));    // Boucle effectuée 1000 fois, tstVal étant
                        // décrémenté à chaque itération.
```

En prévision de la standardisation que l'on peut penser imminente, il est néanmoins prudent de se fixer comme règle que :

- La valeur logique FALSE est représentée par un entier `== 0`.
- La valeur logique TRUE est représentée par un entier `== 1`.
- Une valeur entière différente de 0 ou de 1 donne un résultat logique potentiellement indéterminé.

Ces règles devraient faciliter un éventuel portage ultérieur, si un type booléen est effectivement introduit dans le standard.

Noter que de nombreux constructeurs ont défini un type booléen dans leurs systèmes de développement, ainsi Microsoft, qui définit divers types "propriétaires" dans le cadre de Visual C++, tels que `BOOL`.

5.1.7 *La directive register*

La directive `register` peut être spécifiée pour toute quantité susceptible de trouver de la place dans un registre du CPU considéré. Elle permet de dire au compilateur que, pour des raisons d'optimisation, on désire conserver cette variable dans un registre du CPU plutôt qu'en mémoire centrale.

Cette directive est à considérer comme une indication, et non comme une obligation faite au compilateur. S'il en a la possibilité, le compilateur conservera cette variable dans un registre, mais il ne donne aucune garantie à cet effet. De plus, n'oublions pas que seuls certains types d'objets sont susceptibles de prendre place dans un registre CPU !

En fait, cette directive est un anachronisme, et subsiste plus pour des raisons historiques, où l'optimisation manuelle était nécessaire pour faire tourner un programme convenablement. Dans le cas de compilateurs spécialisés (microcontrôleurs, DSP), cette directive peut être utile pour contrôler la présence de certaines variables dans des registres du CPU.

Attention néanmoins au fait que C n'impose nullement, et ne permet aucunement d'imposer, le registre dans lequel sera stockée telle ou telle variable!

5.1.8 *La directive auto*

La directive `auto` est appliquée par défaut aux identificateurs C. Un objet de type `auto` voit sa mémoire allouée **automatiquement** à chaque fois qu'il est nécessaire de le générer. Si sa déclaration est accompagnée d'une clause d'initialisation, cette initialisation est répétée à chaque fois.

```
int      toto;
auto    int      toto;      // Equivalent à la ligne précédente
int      unObjet = 10;
auto    int      unObjet = 10;      // Equivalent à la ligne précédente
```

5.1.9 *La directive static*

La directive `static` permet d'allouer de la mémoire à une variable une fois pour toutes. Corollairement, si une clause d'initialisation accompagne la directive `static`, elle ne sera effectuée qu'une fois, lors de l'allocation mémoire. La différence avec `auto` est particulièrement importante dans le cas de variables locales à une fonction. Des variables `auto` seront réinitialisées à chaque nouvel appel de la fonction, alors que des variables `static` seront initialisées une fois pour toutes et garderont par la suite la dernière valeur qu'on leur aura affectée.

```
#include <stdio.h>
```

```
void      testAutoStatic()
{
    auto    int unAuto = 10;
    static  int unStatic = 20;
    printf("AutoValue = %d\n", unAuto);
    printf("StaticValue = %d\n", unStatic);
    unAuto = unAuto + 1;
    unStatic = unStatic + 1;
}
```

```
void      main()
{
    testAutoStatic();
    testAutoStatic();
}
```

```
AutoValue = 10
StaticValue = 20
AutoValue = 10
StaticValue = 21
```

Une variable `static` déclarée au niveau global d'un module de compilation n'est pas

visible à l'extérieur du module de compilation.

5.1.10 *La directive extern*

La directive `extern` permet de signaler au compilateur qu'une variable existe, mais n'est pas déclarée à l'intérieur du module de compilation actuel. Par défaut en C, tout identificateur déclaré au niveau global d'un programme, s'il n'est pas explicitement accompagné de la directive `static`, sera visible par d'autres modules de compilation.

Attention ! Une variable déclarée `extern`, mais accompagnée d'une clause d'initialisation sera convertie en `auto` sans autre forme de procès, ce qui risque de créer des conflits d'identificateurs lors de l'édition de liens.

```
extern      int      uneVarExterne;
    // Pas de place memoire reservee, uneVarExterne a
    // ete definie ailleurs, et sera resolue a l'edition
    // de liens

extern      int      uneAutreVarExterne = 10;
    // uneAutreVarExterne va etre definie dans le module
    // de compilation actuel !!!
```

5.1.11 *La directive volatile*

Cette directive permet d'éviter que le compilateur ne se livre à des optimisations sur des expressions où figure cette variable. On indique par cette directive au compilateur que la valeur de cette variable peut changer en dehors du contrôle du programme. Cette directive est utile en conjonction avec une programmation de bas niveau, où certains paramètres manipulés correspondent à des registres d'organes périphériques, par exemple.

```
extern  const  volatile short  usartControlRegister;
```

5.2 Structures et unions

5.2.1 Structures

A l'instar de PASCAL au moyen de la construction RECORD ... END, C permet de construire des types composés au moyen de la syntaxe

```
struct [identificateur] { ... };
```

En PASCAL, on représenterait un type Point (incarné par ses coordonnées cartésiennes x et y) par l'enregistrement suivant :

```
TYPE Coordinate = RECORD
    x, y : REAL;
END;
```

De manière similaire, en C, on peut écrire :

```
struct Coordinate
{
    float    x, y;
};
```

Dans la structure Coordinate ci-dessus, les composantes x et y sont appelées des **membres** (*members*). Chacun des membres de la structure peut être d'un type différent (ainsi qu'en PASCAL).

Une structure définit un type de variable que l'on peut **instancier**:

```
struct Coordinate { float x, y; } cxy1, cxy2;
```

cxy1 et cxy2 sont des **instances** de structure Coordinate.

Pour accéder aux membres d'une structure, la syntaxe est la suivante :

```
cxy1.x = 0; // assigne la valeur 0 au membre x de cxy1
cxy2.y = 1; // assigne la valeur 1 au membre y de cxy2
```

Cette syntaxe est quasiment identique à celle de PASCAL pour les RECORD. En ce sens, PASCAL s'est largement inspiré de la syntaxe C, en l'améliorant dans certains cas.

Il est possible d'initialiser une structure globalement. Ainsi, pour définir une instance de type Coordinate ayant les valeurs x = 3.1415926 et y = 6.4, on utilisera la syntaxe suivante :

```
Coordinate cxy3 = { 3.1415926, 6.4 };
```

Notons que la liste d'initialisation peut être incomplète. Les éléments manquants seront remplacés par des zéros. Ainsi :

```
Coordinate cxy3 = { 3.1415926 };
```

initialise `cxy3.x` à 3.1415926, et `cxy3.y` à 0.

Notons que l'on peut assigner à une structure la valeur d'une autre de même type :

```
Coordinate cxy4 = { 1.0, 2.5 };  
Coordinate cxy5 = cxy4;
```

`cxy5` représente une copie, bit à bit, de `cxy4`.

La structure est une construction essentielle en C.

5.2.2 Structures imbriquées

N'importe quel type de données peut être membre d'une structure. C'est en particulier vrai pour une structure elle-même. Les structures peuvent être de ce fait imbriquées :

```
struct full_date {  
    struct time {  
        int hour, minute, second;  
    };  
    struct date {  
        int day, month, year;  
    }  
    int day_of_week;  
};
```

5.2.3 Champs de bits

Il est possible, au sein d'une `struct`, de définir des champs de taille inférieure ou égale à celle occupée par un entier. Ceci permet de compacter de manière plus efficace l'information, généralement au détriment de la vitesse d'accès.

```
struct date_type  
{  
    unsigned int    day,  
                  month,  
                  year;  
};  
  
struct compact_date_type  
{  
    unsigned int    year : 11;    // 2 ** 11 == 2048  
    unsigned int    day  : 5;     // 2 ** 5  == 32  
    unsigned int    month: 4;     // 2 ** 4  == 16  
};  
  
sizeof(date_type) == 3 * sizeof(int);  
sizeof(compact_date_type) == 3; // 20 bit / 8
```

Dans le petit exemple ci-dessus, la différence de taille entre les deux `struct` atteint 3 / 12 sur une machine 32 bit (`sizeof(int) == 4`), bien que l'information véhiculée reste la même. En revanche, l'accès à un entier est incomparablement plus rapide que l'accès à un champ de bit qu'il faut reconverter en un entier avant de pouvoir effectuer n'importe quelle opération sur la quantité qu'il représente.

Un champ de bit ne peut pas dépasser la taille d'un entier. Ceci introduit une dépendance de la machine dans le programme généré. Ainsi, la construction suivante peut ne pas compiler sans erreurs suivant le type de machine utilisé :

```
struct    some_type
{
    unsigned int    field1 : 17;        // 17 > 16 !
    unsigned int    field2 : 10;
    unsigned int    field3 : 5;
};
```

Cette structure utilise 1 entier de 32 bit sur une machine de 32 bit ou plus, mais ne compile pas sur une machine pour laquelle les entiers sont représentés par des quantités de 16 bit.

Les champs de bit sont souvent utilisés pour la programmation de bas niveau ou la programmation système. Dans le cas de programmation de bas niveau, par exemple sur des microcontrôleurs ou des registres de circuits périphériques (*Universal Serial Asynchronous Receiver Transmitter* USART, *Parallel Input Output* PIO, etc), cette possibilité est bien utile pour décrire les registres de contrôle de ces chips très spécialisés, et évite d'avoir trop recours à du code assembleur au sein du langage C. Un programme de contrôle d'un USART donné écrit en C est réutilisable sur un autre processeur, alors qu'écrit en assembleur, ce ne serait plus le cas.

L'exemple suivant donne un exemple de description de registre d'état d'un USART fictif, comportant trois bit significatifs, plus une indication de la vitesse de transmission actuellement en vigueur, codée sur 4 bit (16 vitesses possibles) :

```
struct    usart_status_register
{
    unsigned    sendRegisterEmpty : 1;        // LSB
    unsigned    receiveRegisterFull : 1;
    unsigned    parityError : 1;
    unsigned    baudRate : 4;
    unsigned    unused : 1;        // MSB, pas obligatoire...
};
```

```
...
volatile struct    usart_status_register    usart_SR;
```

```
if (usart_SR.receiveRegisterFull)
{
    if (!usart_SR.parityError)
    {
        // Read next byte from USART
    }
}
```

```

    }
else
    {
    perror("Parity error reading from USART");
    exit(1);
    }
}

```

Lors du compactage de la structure formée par l'ensemble des champs de bit définis, les restrictions suivantes sont à observer absolument, sans quoi on risque de ne pas aboutir au résultats escomptés :

- Les champs sont implémentés dans l'ordre défini dans la struct. Le compilateur n'effectue en principe pas de réorganisation des champs¹.
- Un champ ne peut pas chevaucher deux entiers. Si c'est le cas, le champ considéré est aligné sur un entier immédiatement consécutif en mémoire, laissant de ce fait des bits inutilisés.
- Un champ de longueur 0 force l'alignement du champ suivant sur l'entier immédiatement consécutif en mémoire.
- L'allocation des bit se fait en commençant par les bit de poids faible, en allant vers les bit de poids fort.
- Il n'y a pas de différence fondamentale entre la manipulation de champs de bit et la manipulation d'entiers, ce qui implique que toutes les opérations licites avec des entiers peuvent être appliquées aux champs de bit.

5.2.4 *Le type union*

Une union est un cas particulier de structure, que l'on peut également rencontrer sous une forme légèrement différente en PASCAL, avec les RECORD à variantes. L'espace mémoire consacré à une union correspond à la place mémoire nécessitée par la plus grande de ses données membres. On ne peut donc affecter de valeur qu'à l'une de ses données à la fois.

Un analyseur lexical, par exemple, selon le type d'une constante donnée, devra mémoriser une valeur de type char, int, long, ou double. Il n'y a que l'une de ces valeurs qu'il doit être utilisée, si bien qu'il faut utiliser une structure comme çà-dessous :

```

struct    TokenValue
{
    char    charVal;
    int     intVal;
    long    lVal;
    double  dVal;
};
printf("%d\n",sizeof(TokenValue));// affiche 20 (Macintosh Symantec C++)

```

1. Si c'était le cas, on ne pourrait pas toujours garantir que la configuration d'un registre externe soit correctement transcrite par le compilateur.

serait gaspiller inutilement de la place mémoire. Le type `union` résoud ce problème :

```
union TokenValue
{
    char    charVal;
    int     intVal;
    long    lVal;
    double  dVal;
};
printf("%d\n", sizeof(TokenValue)); // affiche 10 (Macintosh Symantec C++)
```

Dans cette nouvelle notation, `charVal`, `intVal`, `lVal`, et `dVal` possèdent tous la même adresse, au lieu d'occuper des places mémoire séquentielles, comme dans l'exemple précédent.

Notons qu'il est possible, en C, de définir plusieurs unions dans le cadre d'une même structure (en PASCAL, la partie variante d'un RECORD doit obligatoirement être la dernière).

Il n'y a pas de possibilité, dans une union, de connaître quelle est la variante utilisée par une instance particulière. Si cette connaissance est nécessaire, il appartiendra au programmeur de l'implémenter :

```
enum Tokentype {
    CHAR,
    INT,
    LONG,
    DOUBLE
};

struct Token {
    enum Tokentype Type;
    union Tokenvalue
    {
        char    charVal;
        int     intVal;
        long    lVal;
        double  dVal;
    }
};
```

5.3 *Types pointeurs*

Un pointeur est une variable qui contient l'adresse d'un objet en mémoire. Une utilisation typique de pointeurs, connue en d'autres langages comme Pascal, est leur utilisation dans le cadre de listes chaînées. Un pointeur est généralement utilisé dès que l'on alloue de la mémoire en cours d'exécution du programme.

Comme en PASCAL, un pointeur est associé à un type de données, et il n'est pas sans autre possible de faire pointer un pointeur sur un autre type de données. Cette déclaration de type spécifie comment doit être interprété le contenu de la mémoire sur lequel pointe la variable pointeur; par corollaire, elle indique également quelle place mémoire est réservée par l'objet sur lequel on pointe.

La notation pour définir un pointeur est de préfixer l'identificateur par l'opérateur *. Les deux notations suivantes sont équivalentes :

```
char    *charPointer;  
char*   charPointer;
```

La seconde notation est plus logique, car le type du pointeur est bien char*. La première notation comporte néanmoins certains avantages par rapport à la seconde. La raison peut être indiquée au moyen des exemples suivants:

```
char*    cPtr, cPtr2;  
        // cPtr est un pointeur, cPtr2 est un caractère.  
        // La différence n'est pas évidente, optiquement  
  
char     *cPtr, cPtr2;  
        // Par cette notation, la différence est mieux mise  
        // en évidence.  
  
char*    cPtr;  
char     cPtr2;  
        // Cette notation supprime toute ambiguïté !
```

5.3.1 *Initialisation d'un pointeur*

Pour initialiser un objet, on peut utiliser l'opérateur adresse (&) pour obtenir l'adresse d'un objet du même type que le type de données sur lequel pointe le pointeur. Ainsi :

```
int      someInteger = 1024;  
int      *someIntPtr = &someInteger;
```

On peut également initialiser un pointeur avec un autre pointeur du même type :

```
int          *anotherIntPtr = someIntPtr;
              // == &someInteger.
```

Une erreur souvent commise est d'écrire, pour obtenir le résultat ci-dessus :

```
int          *yetAnotherIntPtr = &someIntPtr;
              // Erreur, int* = int**
```

On fait pointer un pointeur sur des entiers (`yetAnotherIntPtr`) sur l'adresse d'un pointeur sur des entiers, soit un pointeur sur un pointeur sur des entiers. La syntaxe correcte pourrait être:

```
int          **yetAnotherIntPtr = &someIntPtr; // OK
```

C'est une erreur que d'assigner à un pointeur une valeur immédiate, ou la valeur d'un pointeur sur un autre type d'objet.

```
int          i = 1000;
int          *ip = i;           // Faux
```

1000 serait considéré comme une adresse mémoire, ce qui n'a que très peu de chances d'être le cas. De même :

```
int          i = 1000, *ip = &i;
double       *dp = ip;         // Faux
```

La raison de l'erreur est qu'un entier et un double ne prennent pas forcément la même place en mémoire. Considérer qu'une variable de type double réside à l'adresse d'une variable entière peut avoir des conséquences imprévisibles.

Ceci ne signifie pas qu'il est impossible pour un programmeur de convertir un pointeur sur un entier en pointeur sur un double. Mais comme cette conversion n'est pas sûre, est potentiellement dangereuse, le programmeur doit faire cette conversion explicitement, au moyen d'une coercion de type (*type casting*).

5.3.2 Opérations arithmétiques sur des pointeurs

Il est possible de modifier la valeur d'un pointeur en lui additionnant ou soustrayant des valeurs entières. De fait, toutes les opérations arithmétiques élémentaires peuvent être effectuées sur des pointeurs, celui-ci étant considéré comme un `unsigned int`. Ces valeurs correspondent à des objets sur lesquels on pointe, et non pas à des valeurs décimales discrètes. Incrémenter un pointeur de 2 signifie que l'on augmente la valeur de l'adresse qu'il contient de $2 * \text{la taille des variables sur lesquelles le pointeur pointe}$. Ceci rejoint la correspondance entre un pointeur et un tableau, que nous discuterons plus loin.

5.3.3 Création et destruction de variables dynamiques

A l’instar de PASCAL, on utilise fréquemment les pointeurs pour accéder à des variables créées en cours d’exécution du programme, c’est-à-dire des variables dynamiques.

Traditionnellement, en C, la création de variables en cours d’exécution du programme se fait au moyen de routines de librairie (donc non comprises dans le langage même). Ces routines (définies dans `<stdlib.h>`) sont les suivantes :

- `void* malloc(size_t espace_a_allouer);`
`malloc()` alloue dynamiquement une zone mémoire de dimension `espace_a_allouer`. Il retourne un pointeur sur cet espace mémoire, qui est de type `void*` (pointeur universel).
- `void* calloc(size_t nbr_elements, size_t taille_element);`
Comme `malloc()`, mais réserve `nbr_elements` fois la place indiquée par `taille_element`. Utile pour la réservation de tableaux. En principe identique à `malloc(nbr_elements * taille_element)`
- `void* realloc(void *ptr, size_t nouvel_espace);`
Permet de changer la taille mémoire réservée par un pointeur donné (`ptr`). Le contenu existant de la zone mémoire considérée reste inchangé. Si `ptr == NULL`, `realloc()` se comporte de manière identique à `malloc()`.
- `void* free(void *ptr);`
Libère la zone mémoire précédemment réservée à l’adresse `ptr`. Attention ! Cette zone doit avoir été réservée au moyen de l’une des routines `malloc()`, `calloc()` ou `realloc()`. Deux appels successifs de `free()` avec un pointeur sur la même adresse ont un résultat indéterminé (en pratique, une “plantée” spectaculaire du programme !).

```
int*      unPointeurEntier;
int       uneValeur;
uneValeur = 4;
unPointeurEntier = malloc(sizeof(int));
// Allocation de la place mémoire
// nécessaire pour un entier
*unPointeurEntier = 2 * uneValeur; // Assignation d’une valeur
cout<<*unPointeurEntier<<endl;    // Logiquement, devrait écrire 8
// sur le terminal.
free(unPointeurEntier);          // Libération
```

5.3.4 Pointeurs sur des chaînes de caractères

Le type de pointeurs le plus fréquemment utilisé en C est le pointeur sur des caractères, ceci parce que les chaînes de caractère sont implémentées par ce moyen. Par convention, une chaîne de caractères se termine par un caractère nul (`'\0'`). Ainsi, le code suivant permet de calculer la longueur d’une chaîne de caractères :

```
int       strlen(const char *string)
```

```

{
int      result = 0;
while (*string++ != '\0') ++result;
return result;
}

```

Pouvez-vous expliquer en détail ce fragment de code? Par contraste, que fait le fragment de code suivant :

```

int      stringLength(const char *string)

{
for (int result = 0; *string++; result++);
return result;
}

```

5.3.5 *Pointeur sur une structure*

Un pointeur sur une structure est fréquemment utilisé dans n'importe quel langage, entre autres pour implémenter des listes chaînées. La syntaxe d'accès aux membres de la structure se fait logiquement en déréférençant le pointeur, puis en accédant au membre désiré normalement, ainsi que dans l'exemple suivant :

```

struct  Coordinate  {  float  x, y; };
Coordinate*  cPtr;
cPtr = new Coordinate;
(*cPtr).x = 1.0;      // Accès au membre x
(*cPtr).y = 2.0;      // Accès au membre y
// Les parenthèses (*cPtr) sont requises pour des
// raisons de précedence d'opérateurs

```

Cette syntaxe est néanmoins lourde et sujette à des erreurs. C offre de ce fait la possibilité d'accéder aux membres d'une structure pointée de manière plus directe au moyen de l'opérateur ->.

```

cPtr->x = 1.0;      // Accès au membre x
cPtr->y = 2.0;      // Accès au membre y

```

5.3.6 *Pointeur void**

Tout pointeur peut être converti (implicitement ou explicitement) en un pointeur void* (littéralement, en un pointeur sur rien). Un pointeur void* est un pointeur dit générique.

Du fait que le compilateur n'a pas de moyen d'interpréter le contenu de la position mémoire sur laquelle pointe un pointeur void*, il n'est pas possible de déréférencer un tel pointeur, non plus qu'il n'est possible d'effectuer des opérations arithmétiques sur ce pointeur.

```

int*      iPtr;

```

```
int      uneValeur;
uneValeur = 1024;
void*    vPtr;
iPtr = &uneValeur;           // Ok
vPtr = iPtr;                 // Ok, conversion implicite
cout<<*iPtr<<endl;          // Ok, devrait écrire 1024
cout<<*vPtr<<endl;          // Erreur à la compilation; on ne peut
                             // déréférencer un pointeur void*

iPtr = iPtr + 2;             // Ok pour la compilation
vPtr = vPtr + 1;            // Erreur à la compilation; pas
                             // d'opérations arithmétiques sur
                             // des pointeurs void*
```

5.3.7 La valeur NULL

En PASCAL, on utilise la valeur spéciale NIL pour désigner un pointeur non initialisé, c'est-à-dire ne pointant sur aucun objet particulier. PASCAL gère ses pointeurs en cours d'exécution, et assure qu'un pointeur non initialisé aura cette valeur particulière NIL. C, fidèle à sa philosophie qui veut qu'un programme compilant correctement doit être laissé à l'entière responsabilité du programmeur, ne comprend aucune gestion des pointeurs en cours d'exécution, si bien qu'il n'existe pas d'équivalent à NIL.

Il est néanmoins possible de signaler qu'un pointeur n'est pas initialisé en lui affectant la valeur NULL. C'est alors au programmeur qu'il incombe de vérifier si un pointeur est ou non initialisé, et d'affecter la valeur NULL à un pointeur qui n'est plus utilisé pour pointer sur quelque chose. La valeur NULL est compatible avec tout type de données, et le langage garantit qu'un pointeur généré par `new` ou `delete` aura une valeur différente de NULL.

5.4 *Types tableau*

Un tableau est une collection d'éléments de type identique. Les éléments individuels ne sont pas nommés, mais peuvent être indexés par leur position relativement aux autres membres de la collection.

Un tableau est composé d'une spécification de type, d'un identificateur, et d'une dimension. Le type définit le type commun à tous les éléments. L'identificateur donne une identité unique au tableau. La dimension indique quel est le nombre d'éléments compris dans ce tableau. Cette dimension doit pouvoir être déterminée au moment de la compilation, et doit être une constante; de plus, il doit s'agir d'un entier positif non nul.

```
extern  int      arraySize;
int     otherArraySize = 20;
const  int      constantArraySize = 10;

char    arrayOne[20];                // Ok.
char    *arrayTwo[constantArraySize]; // Ok.
int     arrayThree[otherArraySize];
// Faux, pas une constante
double  arrayFour[arraySize];
// Faux, pas une constante
```

Pour les habitués du langage PASCAL, signalons une erreur très communément commise. En C, tous les tableaux commencent par l'indice zéro. Ainsi, un tableau de dimension 10 est composé des éléments allant de 0 à 9. Le segment de programme suivant compilera correctement, mais produira des erreurs à l'exécution:

```
int     intArray[10];
intArray[1] = 1;
intArray[2] = 2;
intArray[3] = 3;
intArray[4] = 4;
intArray[5] = 5;
intArray[6] = 6;
intArray[7] = 7;
intArray[8] = 8;
intArray[9] = 9;
intArray[10] = 10;
```

La syntaxe correcte, et recommandée pour ce genre d'exercice, serait plutôt:

```
#define  intArraySize = 10;
int     intArray[intArraySize];

intArray[0] = 0;
intArray[1] = 1;
intArray[2] = 2;
```

```
intArray[3] = 3;
intArray[4] = 4;
intArray[5] = 5;
intArray[6] = 6;
intArray[7] = 7;
intArray[8] = 8;
intArray[9] = 9;
```

Rappelons que C n'effectue aucun contrôle de valeur lors de l'exécution. Un programme qui compile sans erreur n'implique en aucune façon qu'il soit correct. Ainsi, il est légal d'utiliser n'importe quelle expression ayant une valeur entière pour indexer un tableau, mais la responsabilité de contrôle des limites du tableau est laissée entièrement entre les mains du programmeur.

Il est possible d'initialiser explicitement un tableau en lui fournissant une liste de valeurs séparées par des virgules. Dans ce cas, il n'est pas indispensable de fournir une dimension au tableau: le compilateur peut la calculer lui-même. Si on fournit une dimension, cette dernière peut être plus grande que le nombre de valeurs initiales fournies, mais pas plus petite.

```
int    arrayOne[] = { 0, 2, 4, 6, 8 };
        // Tableau de dimension 5

int    arrayTwo[10] = { 0, 2, 4, 6, 8 };
        // Tableau de dimension 10
        // les 5 premières valeurs sont
        // initialisées
```

On utilise souvent les tableaux pour former des chaînes de caractère. Pour donner une valeur initiale à une chaîne de caractère, on peut soit donner une liste de caractères individuels séparés par des virgules, comme pour les entiers ç-dessus, soit donner une chaîne de caractères constante, délimitée par des ".

```
char str1[] = { 'E', 'I', 'N', 'E', 'V' }
char str2[] = "EINEV";
```

Attention: `str1` ne occupe 5 bytes, alors que `str2` en occupe 6! Pouvez-vous dire pour quelle raison?

Contrairement à PASCAL ou MODULA, il n'est pas possible de copier un tableau dans un autre tableau à l'aide de l'opérateur d'assignation `=`¹. Il est nécessaire de construire une boucle qui copie les éléments individuels. Pour que ceci soit possible, il faut bien sûr encore que cet opérateur d'assignation soit défini pour le type de base du tableau. Pour la même raison, il n'est pas possible d'initialiser un tableau à l'aide d'un autre tableau.

1. Il est par contre possible de définir un membre d'une structure de type tableau, et de copier une structure dans une autre équivalente, ce qui copie le tableau par effet de bord. Cette construction est artificielle, et peu pratique à l'usage, la très grande majorité des tableaux, en C comme dans tout langage d'ailleurs, étant générés lors de l'exécution. Ces derniers ne pouvant bien sûr pas être copiés par cet artifice, et doivent être recopiés "à la main". La pratique consistant à définir un tableau de dimension "suffisante" en supposant qu'il n'y aura jamais besoin de l'étendre dans le cadre d'une application réelle dénote souvent un manque d'expérience de la réalité d'un programme distribué commercialement.

5.5 Tableaux multidimensionnels

C ne prévoit de syntaxe particulière pour les tableaux multidimensionnels. Ainsi,

```
int    intArray[4][3];
```

définit un tableau bidimensionnel, ou matrice. La première dimension est le nombre de lignes du tableau, la seconde le nombre de colonnes. Comme les tableaux unidimensionnels, on peut initialiser des tableaux bidimensionnels :

```
int    intArray[4][3] = {
                                {0, 1, 2},
                                {3, 4, 5},
                                {6, 7, 8},
                                {9, 10, 11},
                                };
```

Il est possible d'écrire plus simplement, encore que moins clairement :

```
int    intArray[4][3] = { 0,1,2,3,4,5,6,7,8,9,10,11};
```

Mais attention! écrire :

```
int    intArray[4][3] = { { 0 }, { 3 }, { 6 }, { 9 } };
```

initialise le premier élément de chaque ligne de `intArray`, alors que les autres sont initialisés à zéro. Par contre :

```
int    intArray[4][3] = { 0, 1, 2, 3 };
```

initialise les trois premiers éléments de la première ligne, et le premier élément de la seconde ligne, alors que les éléments restant sont initialisés à zéro.

En PASCAL ou en Ada, on peut indexer un tableau bidimensionnel par une paire de valeurs entières séparée par une virgule. Souvent, les programmeurs ayant une expérience de ces deux langages tombent dans le piège, et utilisent une syntaxe identique en C. Ils écriront de ce fait :

```
intArray[1, 2];    // au lieu de intArray[1][2]
```

Cette expression, malheureusement, compilera correctement en C aussi bien qu'en PASCAL, mais elle évaluera des résultats fort différents. En PASCAL, on adressera effective-

ment le second élément de la première rangée, en admettant que l'on ait déclaré, comme c'est plus ou moins l'usage en PASCAL,

```
VAR intArray = ARRAY[1..ROWS, 1..COLUMNS] OF integer;
```

En C, cette expression indexe la troisième ligne de `intArray`, et résulte en un pointeur sur l'élément no zéro de cette troisième ligne. En effet :

```
intArray[1, 2];
```

est considérée comme une expression virgule, dont la valeur résultante est la dernière valeur de la liste, ce qui donne :

```
intArray[2];
```

`intArray[2]` est un pointeur sur un tableau unidimensionnel d'entiers... Une indexation multidimensionnelle nécessite une paire de crochets (`[]`) pour chaque index que le programmeur désire spécifier.

Par extension, on peut définir des tableaux à N dimensions. Un tableau quadridimensionnel serait défini de la manière suivante :

```
int tableauQuadra[10][20][30][40];
```

5.6 *Relations entre pointeurs et tableaux*

La notation utilisée pour adresser un tableau est étroitement liée à la notion de pointeur. Ainsi :

```
char    charArray[10] = "abcdefghi";
charArray[0] == *charArray == 'a';
charArray[1] == *(charArray + 1) == 'b';
charArray == &charArray[0];
char    *charPtr = charArray;
```

Déclarer un tableau en lieu et place d'un pointeur a pour simple effet de définir le tableau comme un pointeur constant, et de réserver la place nécessaire au stockage des éléments de ce tableau. Il n'est pas permis de déplacer l'origine d'un tableau. Ainsi :

```
*charArray++;    // Erreur
*charPtr++;      // OK.
```

Cette équivalence peut être mise en évidence par la boucle suivante :

```
for (int index = 0; index < 10; index++)
    if ((charArray[index] != *(charPtr + index)) ||
        (charPtr[index] != *(charArray + index)))
        printError("Il y a un petit détail que je n'ai pas compris... ");
```

`printError()` ne fait bien sûr pas partie du langage C, mais devrait être implémenté d'une manière ou d'une autre.

5.7 Définitions de types de données

La définition de nouveaux types de données par l'utilisateur se fait au moyen du mot-clé `typedef`. L'instruction suivante

```
typedef char AlphaScreen[80][25];

AlphaScreen    DEC_VT220Screen;
```

pourrait être utilisée pour définir un type écran alphanumérique, sous forme d'un tableau bidimensionnel; de manière analogue, un écran graphique VGA (640 * 480) avec des couleurs codées au moyen d'entiers pourrait se définir par :

```
typedef int VGAScreen[640][480];

VGAScreen      PCScreen;
```

D'une manière générale, `typedef` est suivi de la définition du type, puis d'un identificateur qui sera utilisé comme nom de ce nouveau type.

On utilisera fréquemment `typedef` en conjonction avec la construction `struct`, pour alléger l'écriture un peu lourde de `struct` :

```
typedef struct {
    double  real;
    double  imag;
} Complex;

Complex z1, z2;
```

`typedef` permet également de définir des synonymes de type. La définition de synonymes n'a pour but que de documenter de manière plus précise un programme. Ainsi, un processus UNIX livre un mot d'état entier comme résultat (Exit status word); au lieu de le déclarer comme `int`, il est plus clair pour le lecteur de le déclarer comme `ExitStatusWord`. Ceci se fait simplement au moyen de la déclaration suivante:

```
typedef int      ExitStatusWord;
```

Un synonyme peut également être utilisé lorsque le programmeur désire cacher l'implémentation réelle, soit pour des raisons de portabilité, soit pour éviter des manipulations intempestives par un utilisateur.

5.8 *Test*

1. Comment pourrait-on tester de quelle manière une implémentation du langage C interprète les types `char`, `short`, `int` et `long` ?
2. Proposer une implémentation d'une fonction retournant la longueur d'une chaîne de caractères (`char*`).
3. Le code suivant est-il correct ?

```
const int x = 100;
int*      ptrEntier;
ptrEntier = &x;
*ptrEntier = 10;
```

Si ce code vous paraît faux, dites pourquoi, et où le compilateur C protestera. Eventuellement, trouvez une manière de contourner le problème et critiquez cette manière.

4. Peut-on définir un tableau dont les cases contiennent soit des `char`, soit des `int`, soit des `float` ?
5. Le code suivant est-il compilable, et si non, pourquoi ?

```
int      x = 199;
int*     xPtr;
void*    vPtr;
xPtr = &x;
*xPtr = *xPtr - 1;
vPtr = xPtr;
*vPtr = *vPtr - 1;
```