

## VIII Structures enregistrements, définition de nom de types

### 1 Structures enregistrements

Nous avons vus précédemment la notion de tableau qui permet de rassembler dans une même variable une collection d'éléments de même type auxquels on accède par leur indice. Il peut aussi être utile de manipuler des collections d'objets de types différents, ce qui n'est pas possible dans un tableau ordinaire. On utilise alors une variable "structure", qui joue un rôle similaire à celui des enregistrements (**record**) de ADA. La variable structure à un nom, chacun des éléments qu'elle contient est appelé "champs" et possède un nom de champs. On utilise la notation pointée pour accéder à un champ particulier d'une variable structure. Exemple :

```
struct client {
    char nom[25]; // 1er champ
    int numero; // 2ème champ
    double compte; // 3ème champ
};

struct client c11, c12; // désormais "struct client" est un type connu

// on peut déclarer des variables à la fin de la définition de la structure :
struct bidule{
    int machin;
    float truc;
} b1, b2; // b1 et b2 sont de type struct bidule

int main() {
    printf("entrez le nom du 1er client ");
    scanf("%s", c11.nom);
    printf("entrez le numéro du 1er client ");
    scanf("%d", &c11.numero);
    printf("entrez la valeur du compte du 1er client ");
    scanf("%f", &c11.compte);

    // initialiser à vide le 2ème client
    c12.nom[0] = 0; // chaîne vide
    c12.numero = 0;
    c12.compte = 0.0;

    // soustraire 10 euros au compte du client 1
    c11.compte = c11.compte - 10.0;

    //

    return 0;
}
```

Note : les noms de structure et de champ sont gérés dans un espace de noms différents de celui des autres identifiants. On peut donc avoir une variable avec le même nom qu'un champ sans problème d'ambiguïté.

**Structures "récurives".** Une structure ne peut se contenir elle-même, sous peine d'engendrer une récursion infinie, donc pas de :

```
struct recurse {
    int truc;
    char machin;
    struct recurse bidule; // impossible , récursion infinie
};
```

Par contre une structure peut tout à fait contenir un pointeur vers une variable du même type structure (ou d'un autre type structure, évidemment), ce qui est nécessaire pour implanter des collections d'objets dits "chaînés" (listes chaînées, arbres, certaines représentations de graphes, etc).

Exemple de déclaration d'une liste chaînée simple d'entiers :

```
struct maillon {
```

```

int valeur; // valeur à mémoriser dans ce maillon de la liste
struct maillon * suivant; // pointeur vers le maillon suivant
};

```

## 2 Définition de nom de types

Le mécanisme présenté ci-dessus présente le léger défaut d'introduire un nom composé pour les types structures, ainsi dans l'exemple précédent le nom de type est `struct client` et pas simplement `client` comme on aurait pu s'y attendre. D'une manière générale, il est de toute façon utile de pouvoir nommer des types de données à son gré, pour faciliter la compréhension du code. Ainsi une application manipulant des données relatives à un réseau électrique pourrait utiliser des types comme "voltage", "intensite" et "puissance". L'utilisation de noms clairs, permettrait notamment de détecter plus facilement des erreurs de programmation comme par exemple ajouter une intensité à un voltage. Le langage C supporte cette fonctionnalité en permettant la création de noms de types, sans toutefois avoir un contrôle de types strict à la ADA ou à la C++, ce qui en limite l'efficacité : ce n'est pas le compilateur qui vérifiera que vous n'additionnez pas des intensités à des voltages.

Un nouveau type se construit toujours à partir des types pré-existants, éventuellement composés en tableau, en structure ou union. La définition du nouveau type s'écrit comme une déclaration de variable de ce type précédée du mot-clé `typedef`. Le nom qui servirait de nom de variable sans `typedef`, devient un nom de type. Par convention, on écrit souvent en majuscules ces noms de types, pour les distinguer des noms de variables :

```
typedef description_du_type NOM_DU_TYPE;
```

Exemples :

```
typedef double INTENSITE;
typedef double VOLTAGE;
// INTENSITE et VOLTAGE sont des synonymes de double

```

```
VOLTAGE v1, v2; // deux variables de type VOLTAGE

```

```
typedef char CHAINE[80];

```

```
CHAINE ch1; // ch1 est un tableau de 80 caractères

```

```
typedef struct client {
    char nom[25]; // 1er champ
    int numero; // 2ème champ
    double compte; // 3ème champ
} CLIENT; // CLIENT est un synonyme du type "struct client"

```

```
CLIENT c11, c12; // c11 et c12 sont de type CLIENT

```

A noter que le nouveau nom de type n'est disponible qu'à partir de l'instruction suivante, notamment lors de l'usage de structures contenant des pointeurs sur elles-mêmes :

```
struct maillon {
    int valeur;
    struct maillon * suivant; // et non pas MAILLON, encore inconnu ici
} MAILLON; // nouveau nom de type

```

## IX Modularité et portée des variables

### 1 Modularité

De même que le découpage d'un programme en plusieurs fonctions sert à en améliorer la lisibilité, il est aussi souvent utile de découper une application en plusieurs fichiers qui isolent des fonctionnalités relativement indépendantes : on parle de modularité. D'une manière générale, un module est donc un ensemble de fonctionnalités que l'on veut rassembler dans un fichier, et qui ne constitue pas un programme complet (en particulier il n'y a pas de fonction `main`). L'idée est de compiler le module, et de pouvoir ré-utiliser ces fonctionnalités dans d'autres programmes, sans avoir à les réécrire. Les modules constituent donc des bibliothèques de fonctionnalités.

Le module peut donc mettre donc à disposition des programmes "clients" différents "objets" C : types, variables, fonctions. Pour réaliser certaines de ses fonctions il peut aussi utiliser des types, variables et fonctions

qui resteront privées, inaccessibles de l'extérieur du module. En C, le support de la modularité est limité au découpage en fichiers, à la possibilité de compiler ces fichiers séparément et à une gestion minimaliste des noms des variables, types et fonctions utilisés dans ces fichiers pour les rendre visibles de l'extérieur ou, au contraire, privés.

## 2 Organisation d'un module

Idéalement un module se présente généralement en deux fichiers : l'en-tête (*header*) d'extension `.h`, et le corps du module d'extension `.c`. Dans le cas où le module offre seulement des types à partager, il n'y a pas besoin de fichier corps.

**En-tête de module.** Dans l'en-tête on va regrouper tous les éléments qui doivent être visibles de l'extérieur : types proposés par le module, déclarations préliminaires des fonctions proposées, et éventuellement déclaration de variables partagées par le module. Les déclarations de variables seront précédées du mot-clé `extern`.

L'en-tête ne sera pas compilé, mais sera inclus dans les programmes "clients" du module, par une directive `#include`. Attention, un fichier d'en-tête peut se retrouver inclus plusieurs fois indirectement, par exemple on inclus deux modules A et B qui ont eux-mêmes inclus tous les deux le module C : C se retrouve inclus deux fois ce qui génère souvent des erreurs de compilation. Par conséquent il faut donner des directives de compilation conditionnelle afin de n'inclure le module qu'une seule fois (voir exemple plus bas). Ces directives ont besoin d'un identifiant pour le module : par convention nous utiliserons le nom du fichier d'en-tête en remplaçant l'extension `.h` par `_h`.

**Corps du module.** Le fichier corps est d'extension `.c` et ne doit pas contenir de fonction `main`. On y trouve les définitions complètes des fonctions (leur code), la déclaration des variables partagées (sans `extern` cette fois), et tous les objets privés au module : déclaration de types, variables et fonctions privées du module préfixées par le mot-clé `static`. Attention, une variable déclarée au niveau du fichier (hors de toute fonction) sans être préfixée par `static` est visible à l'extérieur du module, ce qui peut entraîner un conflit de nom si une autre variable de même nom existe dans un autre fichier ! Le fichier corps va aussi inclure le fichier d'en-tête du module, ce qui permettra au compilateur de vérifier la correspondance des déclarations préliminaires de fonction avec leur définition.

Le corps du fichier n'est pas inclus dans le programme "client", il est compilé séparément en demandant au compilateur de ne pas construire un exécutable mais simplement un fichier dit "objet" d'extension `.o` lequel sera "lié" lors de la compilation du programme client pour obtenir un exécutable.

Compilation du corps de module : `gcc -c module.c`

On obtient un fichier `module.o`.

**Programme client.** Le programme "client" du module se contente d'inclure le fichier d'en-tête :

```
#include "module.h"
```

On le compile en ajoutant le fichier objet :

```
gcc -o client client.c module.o -Wall
```

## 3 Exemple de module

On traite ici un exemple de module fictif qui regroupe toutes les possibilités offertes avec des types, variables et fonctions partagés et d'autres privés. On appelle ce module `exmod`, avec donc un fichier d'entête `exmod.h` et un fichier corps `exmod.c`. Noter que le nom du fichier en-tête est utilisé dans sa directive de compilation conditionnelle.

**Fichier d'en-tête :** fichier `exmod.h`.

```
// directives de compilation conditionnelle pour inclure une seule fois le module
#ifndef exmod_h
#define exmod_h

// type partagé
typedef struct {
    int a, b;
} EXSTRUCT;

// variable partagée
extern int a;
```

```
// déclaration préliminaire de fonction (extern n'est pas obligatoire pour
// les fonctions)
int truc(int x);

// directive de fin de compilation conditionnelle
#endif
```

**Fichier corps :** fichier `exmod.c`.

```
// inclure l'entête du module pour vérification
#include "exmod.h"

// type privé
typedef struct {
float x, y;
} STRUCTPRIVEE;

// variable partagée (sans extern ici !)
int a;

// variable privée
static int b;

// définition fonction
int truc(int x) {
return toto(x);
}

// fonction privée
static int toto(int n) {
return 2*n;
}
```

## 4 Portée des variables

Comme vu dans la section II, une variable peut être masquée temporairement par une autre de même nom dans un bloc imbriqué. Elle n'est alors plus accessible par son nom, bien qu'elle continue d'exister. On appelle "portée" l'étendue (ou les étendues, pas forcément contiguës) du code où une variable est accessible : elle a été déclarée, et elle n'est pas masquée.

Principales règles en matière de portée des variables :

- la portée d'une variable ne dépend pas de l'exécution du code, elle est entièrement déterminée statiquement, lors de la compilation (donc il suffit de lire le code pour la déterminer).
- un nom de variable désigne la variable déclarée sous ce nom dans le bloc englobant le plus proche, à défaut de la trouver dans un bloc englobant, elle doit être déclarée au niveau du fichier ;
- une déclaration au niveau du fichier (hors de tout bloc), préfixée par **extern** désigne une variable déclarée au niveau fichier dans un autre module ;
- **extern** dans une fonction indique simplement que la variable est déclarée en dehors de la fonction : elle peut l'être dans le même module ;
- les paramètres formels d'une fonction se comportent comme des variables locales ;

Attention, C a des règles par défaut dangereuses (ex : une variable non déclarée est considérée comme de type `int`). Les règles précédentes sont pour une part des conventions de bonnes programmation. L'essentiel est illustré dans l'exemple suivant, présentant deux modules, `m1` et `m2`, d'une application fictive :

**module :** m1.c

```
extern int i; // i vient de m2
int j;       // j global à m1 et m2
double f();  // f() n'est pas définie
// => il faut qu'elle le soit dans m2
// c'est donc une déclaration "extern"

static int g() { // g() est privée à m1
    // il masque dans m1 le g() global de m2
    return 1;
}

int h(double x) { // h globale à m1 et m2
    double y;
    extern int i,j; // externes à la fonction
    // (i de m2, j de m1)

    y = f(x); // f() de m2,
    // x le paramètre passé à h()

    return g(); // g() de m1
}
```

**module :** m2.c

```
int i; // global à m1 et m2
static int j; // local à m2,
// ce j masque dans m2 le j global de m1

double f() { // définition de f()
    // f() globale à m1 et m2
    return 2.3;
}

int g() { // global à l'application
    // mais masqué dans m1
    extern int j; // celui de m2

    return (int) f() + j;
}
```

Notez dans l'exemple que le même nom de fonction `g()` et le même nom de variable `j` sont utilisés dans les deux modules. C'est possible grâce au mot-clé `static` qui restreint un nom à être privé à un module, et donc permet l'emploi du même nom dans d'autres modules. Sans `static`, il y aurait conflit de nom et on ne pourrait pas lier les deux modules dans une application. Cette technique est très utile : quand on écrit un module on ne sait pas forcément à l'avance avec quels autres modules il va être compilé, par conséquent tout ce qui peut être mis en privé avec `static` doit l'être, afin d'éviter de possibles futurs conflits de noms.

## X Utilitaires

Un certain nombre de bibliothèques (ou librairies — de l'anglais *library*) livrées en standard avec le compilateur étendent les capacités du langage de base en fournissant des fonctions utilitaires et accessoirement des types et des variables. Cette section présente une sélection de bibliothèques et de fonctionnalités.

### 1 Mode d'emploi

Pour utiliser une bibliothèque, il faut inclure son fichier d'en-tête :

```
#include <entete.h>
```

Dans la suite de la section les fonctions sont présentées en donnant leur déclaration préliminaire (type de valeur retournée, nom de fonction, type et nombre des paramètres). Pour utiliser les fonctions il faut donc transformer cette déclaration en un appel de fonction.

### 2 Entrées/sorties standard : `stdio.h`

**FILE\* fopen(const char\* filename, const char\* mode);**

ouvre un fichier de nom `filename` et retourne un pointeur de fichier ou le pointeur nul si échec. Le mode d'ouverture est précisé par une chaîne :

- `"r"` : lecture
- `"w"` : écriture (le fichier est effacé s'il existe)
- `"a"` : écriture avec positionnement pour ajout en fin
- `"r+"` : lecture + écriture
- `"w+"` : lecture + écriture (le fichier est effacé s'il existe)
- `"a+"` : lecture + écriture avec positionnement pour ajout en fin

**int fflush(FILE\* stream);**

force la copie des données sur `stream` (ouvert en écriture) en vidant les tampons provisoires et retourne 0 si succès, EOF si erreur. `fflush(0)` vide tous les tampons actuellement ouverts en écriture.

**int fclose(FILE\* stream);**

ferme le fichier (après avoir fait un flush si en écriture) et retourne 0 si succès, EOF si erreur.

**int fprintf(FILE\* stream, const char\* format, ...);**

Convertit les données (selon les indications de la chaîne format) et les envoie sur le flux de sortie stream, et retourne le nombre de caractères sortis, ou une valeur négative si erreur. Un format de conversion consiste en :

- un caractère % suivi de :
- un caractère optionnel parmi :
  - sortie alignée à gauche
  - + sortie alignée à droite
  - espace* le + d'un nombre positif est remplacé par un espace
  - 0** la sortie est complétée à gauche par des 0
- nombre optionnel indiquant la largeur minimale de sortie : si spécifié par \*, la valeur est donnée par le prochain argument non utilisé.
- point optionnel . qui précède la précision demandée (voir après).
- précision optionnelle : si le caractère de conversion est **s**, nombre max de caractère de la chaîne à sortir, si le caractère est dans {eEf}, nombre de chiffres après le point décimal, pour {gG}, chiffres significatifs, pour un entier, nombre minimum de chiffres à sortir. Si donné par \*, la valeur est donnée par le prochain argument non utilisé.
- caractère optionnel modificateur de type :
  - h** short
  - l** long
  - L** long double
- caractère de conversion du prochain argument
  - d,i** int, en décimal
  - o** int, en octal
  - x,X** int, en hexadécimal
  - u** int, en décimal non signé
  - c** char
  - s** char\* (chaîne)
  - f** double, au format [-]mmm.ddd
  - e,E** double, au format [-]m.ddddd(e|E)(+|-)xx
  - g,G** double
  - p** void\*, comme une adresse numérique
  - %** sort un signe %

**int printf(const char\* format, ...);**

équivalent de `fprintf(stdout, f, ...)` : sort sur la sortie standard (écran).

**int sprintf(char\* s, const char\* format, ...);**

comme `fprintf`, mais la sortie est envoyée dans une chaîne. Le nombre de caractères (hors '\0') est retourné.

**int fscanf(FILE\* stream, const char\* format, ...);**

Effectue une lecture avec conversion, lisant dans `stream` selon les indications de `format`. Retourne le nombre d'arguments remplis (lus), ou EOF si fin de fichier ou erreur avant toute conversion. Chacun des arguments qui suit le format, doit être un pointeur. Un format de conversion consiste en :

- des espaces ou tabulations, qui seront sautés
- des caractères ordinaires qui doivent correspondre à ceux du flux d'entrée et qui seront ignorés (à éviter!)
- une spécification de conversion débutée par le caractère %
- \* (optionnel) qui indique de passer le prochain argument
- une largeur de champ de lecture, optionnelle
- caractère optionnel modificateur de type :
  - h** short
  - l** long
  - L** long double

- caractère de conversion pour le prochain argument
  - d** argument int\* requis, en décimal
  - i** int\* requis, décimal, octal ou hexa
  - o** int\* requis, en octal
  - x** int\* requis, en hexadécimal
  - u** unsigned int\* requis, en décimal non signé
  - c** char \* requis ; les espace ne sont pas sautés
  - s** char\* (chaîne terminée par 0) ; les espaces stoppent la saisie,
  - e,f,g** float\*, valeur flottante

**int scanf(const char\* format, ...);**

Equivalent à `fscanf(stdin, f, ...)` : lit depuis l'entrée standard (clavier).

**int sscanf(char\* s, const char\* format, ...);**

comme `fscanf()`, mais lit depuis la chaîne `s`.

**int fgetc(FILE\* stream);**

Retourne le prochain caractère de `stream`, ou EOF si fin de fichier ou erreur.

**char\* fgets(char\* s, int n, FILE\* stream);**

Lit des caractères depuis `stream` pour les copier dans `s`, s'arrête quand `n-1` caractères sont copiés, ou si un `'\n'` est copié, ou si fin de fichier ou erreur. En l'absence d'erreur, `s` est une chaîne correcte. Retourne 0 si fin de fichier ou erreur, ou sinon `s`.

**int fputc(int c, FILE\* stream);**

Ecrit `c` dans `stream`. Retourne `c`, ou EOF si erreur.

**char\* fputs(const char\* s, FILE\* stream);**

Ecrit `s` dans `stream`. Retourne non-négatif si succès, ou EOF si erreur.

**int ungetc(int c, FILE\* stream);**

Simule le "renvoi" de `c` (qui doit être un caractère), dans le flux d'entrée `stream` de telle sorte qu'il sera retourné par la prochaine lecture. Un seul caractère peut être ainsi renvoyé. Retourne `c`, ou EOF si erreur.

**int fread(void\* ptr, int size, int nobj, FILE\* stream);**

Lit (au plus) `nobj` objets de taille `size` depuis le fichier associé à `stream` et les copie en mémoire à l'adresse `ptr`. Retourne le nombre d'objets lus (`feof` et `ferror` peuvent être testés). Note : les `int` peuvent être longs.

**int fwrite(const void\* ptr, int size, int nobj, FILE\* stream);**

Ecrit dans `stream`, `nobj` objets de taille `size` copiés depuis le tableau `ptr`. Retourne le nombre d'objets écrits. Note : les `int` peuvent être longs.

**int fseek(FILE\* stream, long offset, int origin);**

Positionne le fichier associé au flux `stream` sur la position `origin + offset`. Trois valeurs sont possibles pour `origin` :

- début de fichier pour `SEEK_SET`,
- position courante pour `SEEK_CUR`,
- fin de fichier pour `SEEK_END`.

Retourne non-zero si erreur.

**long ftell(FILE\* stream);**

Retourne la position courante dans le fichier associé au flux `stream`, ou -1 si erreur.

**void rewind(FILE\* stream);**

Equivalent à `fseek(stream, 0L, SEEK_SET)` ;

**int feof(FILE\* stream);**

Retourne vrai (c'est à dire !=0) si l'indicateur "fin de fichier" a été positionné pour le flux `stream` (il faut avoir tenté une lecture en fin de fichier pour qu'il le soit).

**int ferror(FILE\* stream);**

Retourne non-zero si l'indicateur d'erreur est positionné pour `stream`.

### 3 Traitement de chaînes : string.h

**int strlen(const char\* cs);**

Retourne la longueur de la chaîne `cs`.

**char\* strcpy(char\* s, const char\* ct);**

Copie ct dans s (et retourne s). Attention! pas de test si s est assez grand pour contenir ct!

**char\* strncpy(char\* s, const char\* ct, int n);**

Copie au plus n caractères de ct dans s, complète s avec 0 si possible (et retourne s). Attention à vous d'assurer que s peut contenir s+1 caractères, pour le marqueur de fin.

**char\* strcat(char\* s, const char\* ct);**

Concatène ct à s (et retourne s). Attention! pas de test si s est assez grand pour contenir ct!

**char\* strncat(char\* s, const char\* ct, int n);**

Concatène au plus n caractères de ct à s (et retourne s). Attention! pas de test si s est assez grand pour contenir ct!

**int strcmp(const char\* cs, const char\* ct);**

Compare cs avec ct, selon l'ordre du jeu de caractères utilisés. Retourne une valeur entière à tester :

- négative si cs < ct (cs avant ct),
- zéro si cs est égal à ct,
- positive si cs > ct (cs après ct).

**int strncmp(const char\* cs, const char\* ct, int n);**

Compare les n premiers caractères au plus de cs et ct. Même sémantique que strcmp().

**void\* memcpy(void\* s, const void\* ct, int n);**

Copie brute de mémoire : copie n octets de l'adresse ct vers l'adresse s and retourne s. Risque d'erreurs si l'intersection zone de destination / zone origine n'est pas vide.

## 4 Nombres aléatoires : stdlib.h

Attention le générateur aléatoire standard du C ne possède pas de très bonnes propriétés statistiques, il est recommandé d'utiliser une librairie spécialisée (plusieurs exemples comme SVID sont disponibles sur internet) pour les utilisations sérieuses.

**void srand(unsigned int seed);**

Initialise le générateur pseudo-aléatoire, ce qui doit typiquement être fait une seule fois dans le programme, à son démarrage. Le paramètre **seed** est la graine du générateur : elle lui donne son point de départ. Lorsque deux exécutions utilisent la même graine, les nombres générés seront les mêmes, ce qui peut être utile pour mettre au point le programme. Une fois la mise au point terminée, on utilise souvent comme graine l'heure obtenue par l'appel à la fonction **times(0)**.

**int rand(void);**

Retourne un entier pseudo-aléatoire dans l'intervalle  $[0, RAND\_MAX]$ . On peut typiquement convertir cet entier en flottant dans l'intervalle  $[0, 1[$  par l'expression : **rand() / (1.0 + (double) RAND\_MAX)**.

## XI Extras

On présente ici quelques constructions peu fréquemment utilisées

### 1 Unions

Il arrive qu'on souhaite pouvoir stocker dans une variable "struct" des données telles que leur "format" (nombre, taille et type de données élémentaires) dépend lui-même de la donnée à stocker (on parle d'enregistrement avec partie variable). Par exemple un logiciel de dessin peut mémoriser des formes géométriques dessinées par l'utilisateur : dans le cas d'un carré, on peut le stocker par les 2 paires de coordonnées (coin en haut à gauche et coin en bas à droite), mais pour un cercle il faut fournir la coordonnée du centre et son rayon... On pourrait alors obtenir un gain de place en utilisant la même zone de stockage mémoire soit pour la 2ème paire de coordonnées, soit pour le rayon, selon le type de figure. C'est ce qui est rendu possible avec une variante de "struct" appelée "union" :

```
struct coord {
    int x, y; // une paire de coordonnées à l'écran
};
```

```
struct figure {
    int genre; // 0 = carré, 1 = cercle
    struct coord c1; // 1ère coord : point haut gauche si carré, centre si cercle
    union extra{
```

```

    struct coord c2; // 2nd coord si carré
    int rayon; // rayon si cercle
} extra; // données complémentaires
};

int main() {
    struct figure f;

    // utilisation de f comme un carré
    f.genre = 0;
    f.c1.x = 10;
    f.c1.y = 50;
    f.extra.c2.x = 40;
    f.extra.c2.y = 120;

    // on change d'avis, f doit stocker un cercle
    f.genre = 1;
    f.extra.rayon = 15; // les données stockées dans f.extra.c2 sont écrasées
}

```

Dans cet exemple il faut comprendre que la structure utilise le même espace mémoire (celui de `union extra`) pour ranger soit la paire de coordonnées `c2` soit la donnée `rayon`. Donc en aucun cas on ne peut mémoriser les deux en même temps, c'est pourquoi il est généralement indispensable d'ajouter un champ qui mémorise quelle est la configuration qui a actuellement cours : c'est le rôle du champ `genre` dans l'exemple. On peut éventuellement se dispenser de ce champ, à ses risques et périls : notez que le langage C ne contrôle pas la cohérence des accès aux champs d'une union (on pourrait écrire dans `extra.c2` et relire dans `extra.rayon`)!... Nous ne détaillons pas davantage l'utilisation des unions, qui sont réservées à des utilisations très particulières et qui sont rendus grandement obsolètes dans les "langages à objets" comme C++.

## 2 Variables "statiques" rémanentes

On a vu que le mot-clé `static` servait à indiquer qu'une variable ou fonction est privée à un module. Les concepteurs de C ont maladroitement utilisé le même mot-clé pour un autre concept, celui de variable "rémanente" :

Une variable `static` déclarée dans une fonction est certes privée à cette fonction (elle l'aurait été même sans `static`), mais de plus elle est allouée dans la zone statique de la mémoire, comme si elle avait été déclarée au niveau du fichier. N'étant pas détruite à la fin du bloc fonction, elle garde sa valeur entre deux appels de la fonction, contrairement aux variables locales habituelles! Si elle est initialisée lors de sa déclaration, cette initialisation est effectuée une seule fois, en début de programme. Cette astuce permet par exemple de compter combien de fois une fonction est appelée dans un programme.

```

void truc() {
    static int cpt = 0; // variable rémanente
    // initialisée à 0 une seule fois, en début de programme

    cpt += 1; // exécuté à chaque appel
    printf("truc() vient d'être appelée pour la %d ème fois\n", cpt);
}

```

## 3 Mots-clés `const` et `volatile`

Les mots-clés `const` et `volatile` sont des "qualificateurs de types" introduits dans le C90 : ils sont placés en préfixe d'un type ou d'une variable pour préciser le comportement du compilateur.

**Mot-clé `const`.** Les variables du type préfixé par `const` sont considérées comme constantes : le compilateur interdira les affectations à ces variables. Par contre on peut (et on devrait) les initialiser à leur création : c'est a priori la seule façon de leur attribuer une valeur, quoiqu'il soit aussi possible d'utiliser une conversion dans un type non constant mais dans ce dernier cas on peut légitimement se demander l'intérêt de lever une interdiction qu'on a soi-même posée.

Au contraire d'une constante du pré-processeur (voir section 2.vi) qui sont de simples "copiés-collés" de littéraux, les constantes définies avec `const` sont effectivement *stockées* en mémoire comme les variables : elles

sont donc typées et possèdent une adresse. Pour que la protection contre l'écriture soit effective, il faut donc qu'elle s'étende aussi à l'éventualité de modifier la constante indirectement par son adresse. On ne pourra donc affecter l'adresse d'une constante `const` que dans un pointeur sur un objet constant (attention, gcc ne fait que signaler un "warning" et pas une erreur si on affecte dans un pointeur ordinaire). Le fait que le pointeur lui-même puisse ou pas être constant est complètement indépendant.

Exemples :

```
const int a = 12; // a est une constante entière
int const aa = 12; // autre écriture possible aa est une constante entière
int b = 5;

printf("\%d\n", a); // ok : lecture
a = 3; // INTERDIT : écriture

int *p1 = &a; // WARNING : p1 pointe une variable entière, pas une constante

const int *p2 = &a; // ok, p2 est un pointeur sur constante entière
int const *pp2 = &a; // ok, autre écriture possible: pp2 est comme p2
*p2 = 3; // INTERDIT : p2 est un pointeur de constante

p2 = &b; // ok, même si b n'est pas une constante
*p2 = 6; // INTERDIT car p2 est un pointeur sur constante (bien que b non constant)

int * const p3 = &b; // notez la place de const juste avant p3
*p3 = 6; // pas de problème c'est l'adresse qui est constante, pas le contenu
p3 = &b; // interdit : l'adresse dans p3 est constante
```

**Mot-clé volatile.** Rarement utilisé, le qualificateur `volatile` indique au compilateur qu'il ne doit pas optimiser la variable en question. En particulier le compilateur n'utilisera pas de copie de la valeur de la variable même si elle ne semble pas avoir été modifiée depuis l'obtention de la copie.

Il est généralement utilisé lorsqu'une variable est mise à jour par un autre processus que le programme (adresse associée à un périphérique matériel, ou partagée entre plusieurs processus), auquel cas le compilateur ne pouvant pas savoir quand la variable est effectivement modifiée, il ne doit pas chercher à optimiser son traitement.

## 4 Mot-clé enum

Le mot-clé `enum` ne définit pas réellement un type énuméré à la ADA, mais crée seulement des littéraux d'une manière proche de `#define`.

```
enum LETTRES{
    A, B, C, D='D', E, F, G=48, H
} v1;
```

La variable `v1` est de type entier de même que tous les symboles A à H, qui ont chacun une valeur entière : A est par défaut mis à 0, les suivant prennent les valeurs entières suivantes, donc B vaut 1, et C vaut 2. On peut fixer aussi une valeur, comme ici D qui vaut la valeur du caractère 'D' dans le code de caractères utilisé. E et F prendront alors les valeurs suivantes donc respectivement celles des caractères 'E' et 'F'. De même G prend la valeur 48 et H == 49.

On peut par exemple utiliser ces symboles (sans apostrophes ni guillemets, ce ne sont ni des chaînes ni des caractères) dans une structure `switch` pour améliorer la lisibilité, tout comme on pourrait le faire avec des constantes littérales du pré-processeur.

Par contre la portée de ces symboles est celle de leur déclaration, et pas forcément celle du fichier comme pour les constantes du pré-processeur.

## 5 Pointeurs de fonctions

Le code d'une fonction étant stocké en mémoire, il a une adresse. On peut récupérer cette adresse et réaliser un appel de fonction.

Pour récupérer l'adresse il faut déclarer un pointeur de fonction : on donne le type de retour de la fonction, le nom du pointeur précédé d'une étoile comme d'habitude mais le tout entre parenthèses, suivi entre parenthèses

de la liste des types des paramètres attendus (liste vide mais parenthèses obligatoires si c'est une fonction sans paramètres).

Lors de l'appel on dépointe le pointeur, toujours entre parenthèses, et on passe les paramètres comme pour une fonction normale.

Si plusieurs fonctions sont de même type, on peut stocker leurs adresses dans un tableau comme dans l'exemple (à compiler avec la librairie mathématique : `gcc -o test test.c -lm`) :

```
#include <stdio.h>
void bonjour() { printf("Bonjour\n"); } // une fonction

extern double fabs(double); // 3 membres de la librairie math.h
extern double sin(double);
extern double cos(double);

int main() {
    // declaration et initialisation des pointeurs de fonctions
    void (* pf)() = &bonjour;
    double (*funtab[3])(double) = {&fabs,&sin,&cos};

    // utilisations des pointeurs
    (*bonjour)(); // affiche message de bienvenue
    int a; float x;
    printf("entrez un entier entre 0 et 2 et un flottant:");
    scanf("%d %f", &a, &x);
    printf("%f\n", (*(funtab[a]))(x)); // affiche la valeur de la fonction choisie

    return 0;
}
```

Le même principe permet à la librairie `stdlib.h` d'offrir une fonction de tri générique :

```
void *qsort(const void *base, int nmem, int size, int (*compar)(const void *, const void *));
```

Tri par l'algorithme du "quick sort", du tableau d'adresse `base`, de taille `nmem` éléments, chaque élément étant lui-même de taille `size` (obtenu par l'opérateur `sizeof`). Le tri nécessite qu'on lui passe l'adresse d'une fonction de comparaison permettant d'obtenir l'ordre relatif entre deux éléments du tableau : cette fonction prend deux pointeurs sur les objets à comparer, et renvoie -1, 0 ou 1 selon que le premier est avant, égal à ou après le second. Les pointeurs paramètres de la fonction de comparaison sont génériques (`void*`), car ils sont passés par `qsort()` qui ne peut pas connaître le type qui va lui être soumis. A l'intérieur de votre fonction de comparaison il suffit de convertir les pointeurs génériques en pointeurs typés, afin de pouvoir accéder aux éléments du tableau.

```
int triEntierDecroissant(const void *p1, const void *p2) {
    // ma fonction de tri, pour des entiers, donne l'ordre décroissant
    const int *e1 = (const int *) p1;
    const int *e2 = (const int *) p2;

    if (*e1 > *e2)
        return -1; // *e1 et *e2 sont déjà dans l'ordre décroissant
    if (*e1 < *e2)
        return 1; // *e1 doit être après *e2
    else
        return 0; // egalite
} // triEntierDecroissant
```

```
int main() {
    int tab[5]={12, 3 ,5 , 7, 4};

    // appel du tri avec ma fonction de comparaison de 2 elements du tableau
    qsort((void *) tab, 5, sizeof(int), &triEntierDecroissant);

    // affichage
```

```
int i;
for (i = 0; i < 5; i++)
    printf("%d\n", tab[i]);

return 0;
}
```