Compléments sur le langage C

I. Ferrané et M.C. Lagasquie

1er décembre 2016

Table des matières

1 Les prérequis

Ce document fait suite au document “Bases sur le langage C” étudié en première partie de cours. Il vous faut donc connaître les notions de base du langage C.
Ce document a été bâti en s'appuyant essentiellement sur les ouvrages suivants : [Dri90, Del92, Got93, RS94].

2 Compléments sur les types de données

Dans cette section, seront abordées les notions de structures et de pointeurs.

2.1 Les structures

Dans la présentation des bases du langage, un seul type de données composées avait été décrit : les tableaux. Or, il existe un autre type de données composées : les structures. À la différence des tableaux qui regroupent des données de *même type*, une structure permet de regrouper des données de *types différents*.

2.1.1 Définition d'un type de structure

Il s'agit de définir un modèle de données que l'on pourra utiliser ensuite comme type pour définir des variables.

```
struct nom_modele
      {
         type1 champ1;
         type2 champ2;
         ...
         typeN champN;
       } ;
```

On pourra ainsi utiliser le type `struct nom_modele` pour définir des variables de type structure.
Chaque élément ou *champ de la structure* est défini par son type et son nom.
Exemple :

```
/* définition d'un modèle de donnée regroupant 2 */
/* entiers et une chaîne de caractères           */

struct Date  {
         int jour;
         char mois[10];
         int annee;
     };
```

On peut aussi définir un type spécifique en tant que synonyme d'un type existant en exploitant l'opérateur `typedef` (voir la section suivante).

2.1.2 Définition d'un synonyme de type : `typedef`

Il s'agit plutôt de la possibilité de renommer un type puisque celui-ci doit être décrit au moment de la définition.

```
typedef description_type nom_type;
```

Exemple :

```
/* définition d'un type nommé type_date et */
/* correspondant au modèle de donnée       */
/* struct Date                             */

typedef struct Date  {
```

```
int jour;
char mois[10];
int annee;
}      type_date;
```

Le type `type_date` est donc un synonyme du type `struct Date` et les deux peuvent être utilisés indifféremment pour la définition de variables.
Remarque : L'opérateur `typedef` fréquemment utilisé avec les structures peut également s'utiliser pour définir d'autres types de données.

2.1.3 Définition d'une variable correspondant à une structure

Ceci n'est possible qu'une fois que le modèle ou le type de la structure a été défini. La définition d'une variable entraîne l'allocation mémoire de la zone nécessaire pour stocker les données correspondantes.

```
/* définition d'une variable de nom Jour_J et contenant*/
/* 3 champs : jour, mois et annee                      */
struct Date Jour_J;
```

ou

```
type_date Jour_J;
/* dans la mesure où le type type_date a été défini */
/* comme ci-dessus                                  */
```

On peut aussi définir en même temps le type et la variable (mais pas en passant par un `typedef`) :

```
/* définition d'un modèle de donnée regroupant 2 */
/* entiers et une chaîne de caractères           */

struct Date  {
         int jour;
         char mois[10];
         int annee;
     } Jour_J;
```

2.1.4 Initialisation d'une variable correspondant à une structure

Comme pour les tableaux, il faut spécifier la liste des valeurs de chaque élément.
`struct nom_modele nom_var = {val_chp1, ..., val_chpN};`
Exemple :

```
struct Date Noel = {25, "décembre", 2000 };
type_date Nouvel_an = {1, "janvier",2001};
```

On peut aussi définir en même temps le type et la variable et faire l'initialisation de la dite variable (mais toujours pas en passant par un `typedef`) :

```
/* définition d'un modèle de donnée regroupant 2 */
/* entiers et une chaîne de caractères           */

struct Date  {
         int jour;
         char mois[10];
         int annee;
     } Noel = {25, "décembre", 2000 };
```

En dehors de l'initialisation faite au moment de la définition de la donnée, il est nécessaire d'accéder à chaque champ individuellement.

2.1.5 Accès aux champs d'une structure.

Avec une variable définie comme une structure, on utilise la notation pointée.

```
nom_variable.nom_champ
```

Exemple :

```
Jour_J.jour= 14;
strcpy(Jour_J.mois, "juillet");
Jour_J.annee = 2000;

/* pour attribuer une valeur à une chaîne de caractères  */
/*  (=tableau de caractères) il faut utiliser la         */
/* fonction strcpy définie dans la bibliothèque string.h */
```

2.1.6 Affectation

Contrairement au tableau qu'il faut traiter élément par élément, il est possible d'affecter la valeur d'une structure à une autre structure.
Exemple :

```
type_date Autre_jour;
...
Autre_jour = Jour_J;
```

2.1.7 Tableaux de structures

Un tableau est un ensemble d'éléments de même type. À partir du moment où un type de structure a été défini, il peut être nécessaire de regrouper plusieurs structures dans un même tableau, par exemple s'il est besoin d'effectuer des traitements similaires sur plusieurs structures. Un élément du tableau sera désigné par son indice, et partant de là on pourra accéder à l'un de ses champs.
Exemple :

```
struct Date Dates_cles[10];
/* définition d'un tableau de 10 structures */
int i, nb_dates_mars = 0;
int jour_mars[10] = {0,0,0,0,0,0,0,0,0,0};

/* recherche des dates concernant le mois de mars et */
/* memorisation des jours pour effectuer un traitement */
/* ulterieur                                          */
for(i=0; i < 10; i++)
     {
        if( strcmp(Dates_cles[i].mois, "mars") == 0)
            {
               jour_mars[nb_dates_mars] = Dates_cles[i].jour;
               nb_dates_mars ++;
            }
        ...
     }
```

2.1.8 Imbrication de structures

Une structure peut admettre un champ qui soit lui même une structure. Seul cas impossible, lorsque le champ doit être du même type que celui de la structure en cours de définition. Dans ce cas particulier, celui des structures récursives (dites aussi auto-référentielles), il faut passer par un champ qui soit un pointeur sur une donnée du type en cours de définition (voir sections 2.2 page suivante et 5 page 26).
Exemple :

```
/* définition d'un autre type de structure */
struct Personne
   {
     char nom[20];
     char prenom[20];
     struct Date ne_le;
   };

/* définition et initialisation d'une variable */
/* comportant une structure imbriquée             */
struct Personne P1 = {"Durand", "Paul", {12, "mai", 1980}};

/* exemple d'acces au champ d'une structure imbriquée */
if(P1.ne_le.annee >1970)
        printf("%s %s a moins de 30 ans", P1.prenom, P1.nom);
```

2.1.9 Exercice

Une promotion compte 25 étudiants maximum. Un étudiant se caractérise par son nom, son prénom, sa date de naissance, son numéro de carte d'étudiant et l'ensemble des moyennes obtenues dans les 6 modules de la formation.

1. Donner les définitions de données nécessaires pour représenter ces informations.
2. Donner la partie de code qui une fois connues toutes les informations concernant les étudiants donne l'identité du major de la promotion.

2.2 Pointeurs (aspect statique)

2.2.1 Généralités

Le langage C est un langage de haut niveau c'est-à-dire qu'il permet de programmer de manière structurée. Cependant, conçu initialement pour l'écriture du système d'exploitation UNIX, ce langage dispose également de nombreuses fonctionnalités dites de "bas niveau" permettant d'agir directement sur certaines ressources de la machine (mémoire, registres, ...). La notion de pointeur permet donc de manipuler des données à partir de l'adresse mémoire où elles sont stockées durant l'exécution du programme.
Remarque : ici, nous abordons l'utilisation des pointeurs dans un aspect uniquement statique de la gestion mémoire. L'aspect dynamique sera traité dans la section 5 page 26.

Organisation de la mémoire Lorsqu'un programme est exécuté, une zone de la mémoire centrale lui est attribuée momentanément. Les données nécessaires au programme sont alors stockées dans cette zone qui correspond à un ensemble d'octets. Un octet étant la plus petite unité adressable, chaque octet de la zone mémoire associée au programme peut être repéré par son adresse.

Allocation statique d'une donnée Lors de la définition d'une donnée (variable simple, tableau, variable de type structure, pointeur, ...) une zone mémoire lui est associée de façon à ce que la valeur de celle-ci puisse être mémorisée (=stockée). Le nombre d'octets constituant cette zone dépend du type de la donnée (voir la présentation des bases du langage). Par exemple, un caractère occupera 1 seul octet alors qu'un réel pourra en occuper 4, 8 ou plus suivant la machine et qu'il s'agit d'un réel simple précision (`float`) ou double précision(`double`). L'adresse de la zone ainsi allouée correspond en fait à l'adresse du premier octet de la zone.

Qu'est-ce-qu'un pointeur Un pointeur est une variable destinée à contenir l'adresse d'une donnée. Une adresse est codée sur 4 octets (ou plus suivant le type de machine). Par conséquent, lors de la définition d'un pointeur, seulement les octets nécessaires pour stocker l'adresse sont alloués.
ATTENTION ! Si un pointeur est destiné à permettre l'accès à une chaîne de caractères par exemple, en aucun cas, la zone nécessaire pour stocker la chaîne n'est allouée en même temps. Ceci doit faire l'objet d'une action spécifique (allocation statique ou dynamique de la chaîne).
Accéder à une donnée par l'intermédiaire de son adresse pose 2 problèmes :

— disposer de la bonne adresse,
— savoir de combien d'octets est constituée cette donnée.

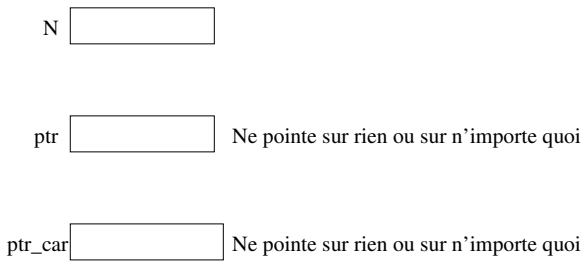
Le premier relève de la séquence d'instructions écrites qui peut, s'il y a un problème, amener à une erreur sur l'adresse de la donnée. Le second dépend du type de la "donnée pointée". Aussi un pointeur est associé au type de la donnée pointée.

Définition d'un pointeur

```
type_donnée_pointée * nom_ptr;
```

Le symbole * marque ici la définition d'une variable qui sera considérée comme un pointeur sur le type de donnée indiqué.
Exemple :

```
int N ;          /* N est une variable correspondant à un */
                 /* entier => allocation de 2 ou 4 octets */
                 /* ou plus suivant le cas                */

int * ptr ;      /* ptr est une variable correspondant   */
                 /* à un pointeur d'entier => allocation */
                 /* de 4 octets (ou plus) pour stocker   */
                 /* l'adresse d'une donnée entière       */
                 /* NE POINTE SUR RIEN ou SUR N'IMPORTE  */
                 /* QUOI, sa valeur ne DOIT PAS être     */
                 /* utilisée directement on doit lui     */
                 /* affecter l'adresse d'un entier avant */

char * ptr_car; /* ptr_car est une variable             */
                 /* correspondant à un pointeur de       */
                 /* caractère =>  allocation des  octets */
                 /* pour stocker l'adresse d'une donnée  */
                 /* de type caractère  MAIS NE POINTE    */
                 /* SUR RIEN ou SUR N'IMPORTE QUOI;      */
                 /* sa valeur ne DOIT PAS être utilisée  */
                 /* directement on doit lui affecter     */
                 /* l'adresse d'un caractère avant       */
```



Définition de 3 variables : N (entier), ptr (pointeur d'entier), ptr_car (pointeur de caractère)

Valeur nulle pour un pointeur : `NULL`.
`NULL` est une valeur prédéfinie dans le fichier stdio.h et qui indique qu'un pointeur ne pointe sur rien. Pour programmer proprement et éviter les très nombreuses erreurs dues à la manipulation des pointeurs, il est plus prudent d'initialiser tout pointeur défini avec cette valeur (ce n'est pas toujours fait automatiquement). Un accès à une donnée à partir d'un pointeur ne pointant sur rien (= dont la valeur est `NULL`) provoque une erreur d'exécution qui donne lieu au message *segmentation fault - core dumped*. Sinon, un pointeur non initialisé contient n'importe quoi et on peut très bien retomber par hasard sur une donnée du programme. Dans ce cas, le programme peut fonctionner malgré l'erreur et les "effets néfastes" peuvent ne survenir que plus tard. Il est alors très difficile d'en retrouver la cause.
Exemple d'initialisation :

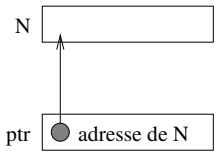
```
int * ptr = NULL ;
char * ptr_car = NULL;
```

2.2.2 Opérateurs liés aux pointeurs

Deux opérateurs sont à la base de la manipulation des pointeurs. Il s'agit de l'opérateur d'adressage et de l'opérateur d'indirection.

Opérateur d'adressage : `&` Il permet d'obtenir l'adresse mémoire d'une variable. Ainsi on peut affecter le résultat de cette expression à un pointeur. Dans ce cas, et uniquement dans ce cas, le lien entre le pointeur et la donnée pointée est établi.
Exemple :

```
int N;
int *ptr = &N;
/* dans ce cas ptr est défini, les octets qui */
/* le représentent sont alloués et l'adresse  */
/* de la variable N y est stockée. ptr peut   */
/* directement être utilisé.                  */
```



Définition de 2 variables : N (entier) et ptr (pointeur d'entier) initialisé à l'adresse de N

Opérateur d'indirection : `*` Il permet d'accéder à une donnée via son adresse (= accès indirect). Ainsi appliqué à un pointeur il permettra au programme d'aller chercher la donnée pointée, à condition bien sûr que le pointeur contienne une adresse valide. L'utilisation de cet opérateur sur un pointeur ne pointant sur rien (valeur égale à `NULL`) provoque une erreur d'exécution (erreur d'adressage) signalée par le message *segmentation fault - core dumped*.
Exemple :

```
int N = 20;
int *ptr = &N;

printf("valeur directe de N = %d", N);
printf("valeur de N obtenue par indirection = %d", *ptr);

/* dans les 2 cas la valeur 20 est affichée */
```

Autres opérations sur les pointeurs

Affectation : on peut affecter une valeur à un pointeur. Il doit s'agir soit de la valeur `NULL`, soit d'une valeur correspondant à une adresse du même type que celui du pointeur.
Exemple :

```
int N;              /* variable entière */
float Y ;           /* variable flottante */
int * ptr_N = &N;     /* pointeur d'entier initialisé à */
                    /* l'adresse de N => pointe sur N */
int * autre_ptr;
float *ptr_Y = &Y;  /* pointeur de flottant      */
                    /* initialisé à l'adresse de */
                    /* Y => pointe sur Y         */
autre_ptr = ptr_N;  /* affectation au pointeur   */
```

```
                        /* autre_ptr de la valeur de */
                        /* ptr_N c'est-à-dire de l'  */
                        /* adresse de N => pointe    */
                        /* aussi sur N               */
autre_ptr = ptr_Y;     /* erreur : les 2 pointeurs  */
                        /* ne sont pas du même type  */
                        /* int * et float *          */
```

Comparaison : on peut comparer 2 pointeurs (`==`, `!=`, `>`, `<`, `>=`, `<=`) à condition :

- qu'ils soient de même type,
- et qu'ils correspondent à des adresses d'éléments appartenant à la même zone mémoire (même tableau - voir section 2.2.3 page suivante).

Différence : on peut faire la différence de 2 pointeurs à condition :

- qu'ils soient de même type,
- que chacun corresponde à l'adresse d'éléments appartenant à la même zone mémoire (même tableau - voir section 2.2.3 page suivante).

Le résultat obtenu correspond alors au nombre d'éléments (et pas au nombre d'octets) qui séparent les 2 adresses.

ATTENTION !!! Les opérations telles que la différence et le décalage (évoqué ci-dessous) tiennent compte implicitement de la taille de la représentation mémoire des données pointées. Ainsi, une différence de 1 correspondra à 1, 2, 4, 8 octets ou plus suivant que les données pointées sont de type `char`, `short`, `long` ou `float`, `double` ou encore d'un type défini par le programmeur.

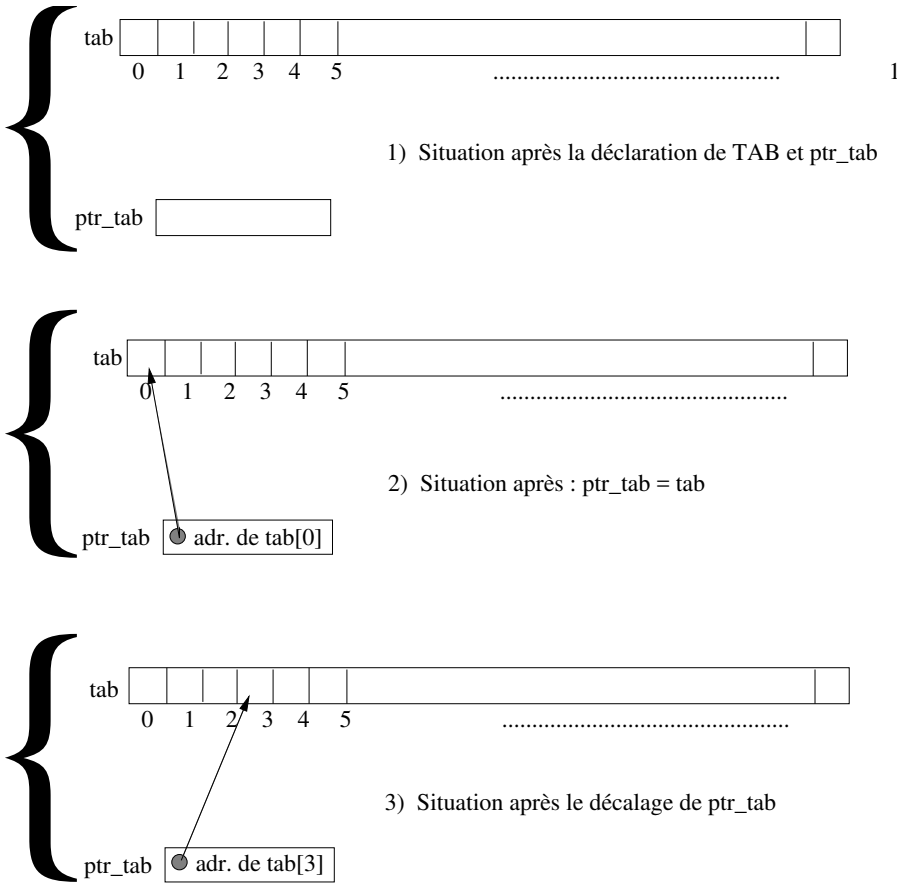
Décalage : il s'agit d'ajouter où de retrancher une valeur entière à un pointeur pour obtenir l'adresse d'une donnée de même type que celle pointée par le pointeur et située dans la même zone mémoire car seul un tableau permet d'accéder à des éléments mémorisés consécutivement (voir section 2.2.3 page suivante).

Il faut d'abord que le pointeur contienne une adresse valide car effectuer un décalage n'a de sens que s'il s'agit d'accéder à un élément précédent ou suivant celui désigné par le pointeur. La valeur entière à ajouter ou à retrancher correspond donc au nombre d'éléments dont il faut se décaler pour accéder à l'élément recherché. Elle ne correspond pas au nombre d'octets.

Par conséquent, il faut d'abord que le pointeur contienne une adresse valide.

Exemple :

```
int tab[20];           /* allocation de 20 entiers avec */
                       /* tab = &tab[0]                 */

int * ptr_tab;           /* allocation de 4 octets   */
                       /* pour stocker une adresse */

ptr_tab = tab;         /* ptr_tab pointe sur le premier */
                       /* élément du tableau de nom tab */

ptr_tab = ptr_tab + 3; /* ptr_tab pointe sur le        */
                       /* quatrième élément du tableau */
                       /* de nom tab                   */
```



Utilisation des pointeurs En langage C, dès que l'on veut réaliser une application un peu évoluée, il est indispensable de savoir manipuler des pointeurs. En effet, ceux-ci jouent un rôle important dans :

- l'accès à un élément de tableau,
- le passage et la modification de paramètres dans une fonction (voir les sections 3.4 page 15 et 3.4 page 15),
- l'accès à une donnée allouée dynamiquement (voir la section 5 page 26)

2.2.3 Pointeurs et tableaux

Historiquement, la notion avec les crochets (`[` et `]`) n'existait pas pour les tableaux. On utilisait donc les pointeurs pour parcourir les tableaux. Les opérations de comparaison, différence et décalage des pointeurs sont donc une rémanence de cette époque-là et ne sont plus guère utilisées de nos jours. Il est toutefois important de comprendre comment cela fonctionnait et c'est donc l'enjeu de la section courante.

Complément sur les tableaux Toute définition de tableau (voir la présentation des bases du langage), qu'il soit à une ou à plusieurs dimensions, donne lieu aux actions suivantes :

- allocation du nombre d'octets nécessaires pour stocker l'ensemble des éléments du tableau. La taille de la zone ainsi allouée correspond au nombre d'éléments maximum du tableau (taille) multiplié par le nombre d'octets occupé par un élément (`sizeof(type_element)`) ;
- définition d'une valeur de pointeur (= adresse) constante, donc non modifiable, égale à l'adresse du premier élément de la zone allouée (= adresse du premier octet). Cette valeur constante est représentée par le nom du tableau seul (sans indice).

Quel que soit le tableau et le type des éléments qu'il contient, si son nom est `T` alors `T` représente la même valeur que `&T[0]`.
Exemple :

```
char tab_car[10];
/* allocation de 20*1 = 20 octets            */
```

```
/* tab_car représente l'adresse du premier   */
/* élément => tab_car est égal à &tab_car[0] */

float res[5];
/* allocation de 5*4 = 20 octets          */
/* res  représente l'adresse du premier   */
/* res est égal à &res[0]                 */
```

Mode d'accès aux éléments d'un tableau Le mode d'accès par indiçage présenté dans la partie de ce document concernant les tableaux (voir la présentation des bases du langage), est une manière simple d'accéder à un élément. Cependant, il faut savoir que cette notation est systématiquement traduite, lors de la compilation, en une notation faisant intervenir un décalage d'adresse. Cela consiste à partir de l'adresse du premier élément représentée par le nom du tableau et à lui ajouter une valeur entière pour obtenir l'adresse de l'élément recherché. L'opérateur d'indirection `*` permet ensuite d'obtenir l'élément situé à cette adresse.
Soit `tab` le nom d'un tableau de type quelconque et `i` une valeur d'indice possible :

`tab[i]` est converti en `*(tab+i)`

`tab` étant l'adresse du premier élément, `tab+i` est l'adresse du `i+1`ème élément. Par conséquent, `*(tab+i)` correspond à l'élément situé à l'adresse `tab+i` soit le `i+1`ème élément du tableau (comme `tab[i]` !)
ATTENTION !!! Lors du décalage d'un pointeur, il n'y a AUCUNE vérification de la validité de la nouvelle adresse. Si celle-ci est en dehors du tableau auquel appartient l'élément pointé, on a une adresse invalide qui conduira à un moment donné à une erreur d'exécution ; Il faut donc être EXTRÊMEMENT vigilant lorsque l'on effectue ce type d'opérations.

2.2.4 Pointeurs et structures

On peut définir un pointeur sur des données de tout type y compris sur des structures (voir section 2.1 page 1) et même sur des pointeurs.
Exemple, en utilisant les structures `Date` et `Personne` définies en section 2.1 page 1 :

```
struct Date *ptr_date ;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le    */
/* pointeur ptr_date ne pointe sur aucune   */
/* donnée valide                            */

struct Personne * ptr_pers;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le    */
/* pointeur ptr_pers ne pointe sur aucune   */
/* donnée valide                            */

struct Date une_date = {14, "Juillet", 2000};
struct Personne une_personne= {"Pierre", "Dubois", {12, "Avril", 1980}};
/* définition et allocation des  octets nécessaires */
/* pour stocker une première donnée correspondant à */
/* une structure date et une autre correspondant à  */
/* une structure personne                           */

ptr_date = &une_date ;
/* le pointeur reçoit l'adresse d'une donnée */
/* ptr_date pointe sur la donnée une_date    */

ptr_pers = &une_personne;
/* le pointeur reçoit l'adresse d'une donnée  */
/* ptr_pers pointe sur la donnée une_personne */
```

Accès au champ d'une structure via un pointeur Si le type de donnée associé à un pointeur (= type de la donnée pointée) est une structure, et que bien sûr le pointeur a reçu l'adresse d'une variable du type de données en question alors le pointeur permet d'accéder indirectement aux champs de la structure pointée. Deux notations sont possibles et totalement équivalentes :

`nom_ptr->nom_champ` La flèche représentée par la combinaison des symboles `-` (moins) et `>` (supérieur) indique que l'on accède indirectement au champ spécifiée de la donnée pointée par le pointeur.

`(*ptr_nom).nom_champ` La notation pointée ne peut être utilisée que si la partie gauche correspond à une variable de type structure. Pour obtenir cela à partir d'un pointeur, il faut d'abord appliquer l'opérateur d'indirection `*` au pointeur ; le parenthésage est obligatoire. On obtient ainsi la donnée pointée. On peut donc ensuite utiliser la notation pointée pour accéder au champ recherché.

Exemple :

```
/* affichage de la valeur de la donnée une_date de 3 façons */
/* différentes                                               */
printf(" une date = %d %s %d", une_date.jour, une_date.mois, une_date.annee);
printf(" même chose = %d %s %d",ptr_date->jour, ptr_date->mois, ptr_date->annee);
printf(" encore la même chose = %d %s %d", (*ptr_date).jour, (*ptr_date).mois,
          (*ptr_date).annee);

/* affichage de la date de naissance de la donnée */
/* une_personne de 3 façons différentes           */
printf(" une date = %d %s %d", une_personne.ne_le.jour, une_personne.ne_le.mois,
          une_personne.ne_le.annee);
printf(" même chose = %d %s %d", ptr_pers->ne_le.jour, ptr_pers->ne_le.mois,
          ptr_pers->ne_le.annee);
printf(" encore la même chose = %d %s %d", (*ptr_pers).ne_le.jour, (*ptr_pers).ne_le.mois,
          (*ptr_pers).ne_le.annee);
```

Structures récursives ou auto-référentielles Le seul moyen d'avoir une structure récursive c'est-à-dire qui admet au moins un champ correspondant à une donnée du même type que la structure en cours de définition est d'utiliser un pointeur sur le type en question.
Exemple :

```
struct membre_famille                /* erreur */
 {         char nom[20];
           char prenom[20];
           int age;
           struct membre_famille pere, mere; } ;

   /* n'est pas possible car un type tel que          */
   /* struct membre_famille ne peut être utilisé      */
   /* qu'une fois complètement défini                 */
   /* il faut donc utiliser un pointeur               */

struct membre_famille                /* correct */
 {         char nom[20];
           char prenom[20];
           int age;
           struct membre_famille *ptr_pere, *ptr_mere; } ;
```

Ce type de structure est utilisé comme élément de base d'une structure plus complexe où des données de même type sont liées les unes aux autres. Ceci est présenté dans la partie concernant la gestion dynamique de la mémoire par l'utilisation des pointeurs (voir section 5 page 26).

2.2.5 Exercices

Énoncé 1 *Reprendre l'exercice sur les structures en utilisant la notation avec les pointeurs.*

Énoncé 2 *Soit la structure suivante :*

```
struct membre_famille
 {          char nom[20];
            char prenom[20];
            int age;
            struct membre_famille *ptr_pere, *ptr_mere; } ;
```

Ecrire un programme qui permet de créer les 3 personnages suivants : TOTO, 10 ans, fils de EGLANTINE X née Y, 30 ans, et de ARTHUR X, 35 ans
On souhaite maintenant connaître les enfants d'un individu donné. Proposez une nouvelle structure et récrivez le programme créant TOTO et ses parents.
On apprend maintenant que TOTO a été adopté et que ses parents biologiques sont MARGUERITE Y, 25 ans, et ALPHONSE X, 25 ans. Sans recréer les personnages déjà existants, écrivez le programme mettant à jour la filiation de TOTO.

3 Fonctions et passages de paramètres

Un vrai programme se réduit rarement à un seul "bloc de code" (hormis dans les premiers exercices de TP). Dès que l'on développe une application un peu complexe, il est nécessaire voire indispensable de structurer un programme en plusieurs "blocs de code". En effet d'un point de vue méthodologique, il est plus simple que résoudre plusieurs "petits problèmes" qu'un seul "gros". Un programme est alors décomposé en *sous-programmes*, chacun d'eux pouvant (et devant) être validé séparément. L'écriture de sous-programmes se justifie dès qu'il s'agit de réaliser des tâches distinctes éventuellement répétées à plusieurs occasions au cours du programme et/ou appliquées à des données différentes appelées alors paramètres.
Par exemple, lors de la présentation des bases du langage, on a écrit des programmes en C dans lesquels il y avait certaines répétitions (des bouts de code que l'on retrouvait quasiment inchangés). Par exemple :

```
int x, y, z, aux;
...
if (x > y) { aux = x ; x = y,; y = aux ; }
if (y > z) { aux = y ; y = z,; z = aux ; }
if (x > y) { aux = x ; x = y,; y = aux ; }
```

Les parties `then` de ce code sont identiques aux variables près. D'où l'idée de les factoriser en un sous-programme.
En langage C, la notion de sous-programme correspond uniquement à la notion de *fonction*. En plus de la fonction `main` obligatoire et des fonctions standards prédéfinies (`printf`, `scanf`, ...), le programmeur peut donc être amené à définir ses propres fonctions.

Difficulté de l'utilisation de sous-programmes/fonctions Puisqu'il s'agit ici de ne pas avoir un programme monolithique impossible à debboguer ou à maintenir, il faut donc savoir "découper" son programme en diverses parties, chaque partie étant écrite et testée à part puis intégrée à l'ensemble pour former le programme complet.
Cela facilitera :

- le travail en équipe,
- le debbogage,
- la lecture et la maintenance du code ainsi produit (en particulier lors d'un travail en équipe).

Pour que cette technique soit utilisable, il faudra bien-sûr pouvoir définir les interfaces entre ces diverses parties. C'est là le rôle des paramètres (voir section 3.4 page 15).

3.1 Définition d'une fonction

Celle-ci est constituée de 2 parties, l'*entête* ou *prototype* et le *corps*, en respectant le schéma général suivant :

```
/* en-tête de la fonction (prototype) */
{
    /* corps de la fonction (déclaration de variables locales et instructions) */
}
```

Les définitions de fonctions se trouvent les unes à la suite des autres après les directives (voir la présentation des bases du langage) et d'éventuelles déclarations de variables et/ou de fonctions. L'ordre n'a, a priori, pas d'importance, les problèmes de référence en avant (utilisation d'une fonction avant sa *définition*) pouvant être résolus par la présence des *déclarations* de fonctions qui vont être utilisées avant toute définition de fonction. En langage C, il n'est pas possible d'imbriquer les définitions de fonctions. Elles sont toutes définies au même niveau.

3.1.1 Prototype d'une fonction

Une fonction est un sous-programme qui renvoie une seule valeur et qui s'applique aux données du programme qui lui sont transmises au moment de l'appel (demande d'exécution de la fonction). L'ensemble de ces informations se retrouvent dans l'entête de la fonction :

```
type_val_retour nom_fonc(liste_typée_de_paramètres)
```

Type de la valeur de retour Cela correspond au type de la donnée que renvoie la fonction à la partie de programme qui l'a appelée. Ce type est soit un type scalaire prédéfini (voir la présentation des bases du langage), soit un type défini par le programmeur (avec `typedef` – voir section 2.1 page 1), soit une structure (voir section 2.1 page 1), ou encore un pointeur sur un type donné (voir section 2.2 page 4).

Bien que la notion de procédure n'existe pas explicitement, une fonction peut ne pas renvoyer de valeur. Ceci doit se traduire par un type de valeur de retour correspondant à `void` (ce qui indique clairement l'absence de données) et non par une omission du type. En effet, lorsque le type de la valeur de retour est omis, alors la fonction est considérée par le compilateur comme renvoyant un entier (`int`). Un manque de rigueur dans la programmation pourra donc entraîner des problèmes de conflits de types ou de double définition lors de la compilation.

ATTENTION !!! En langage C une fonction ne peut pas renvoyer un tableau, ni une autre fonction.

Nom de la fonction C'est l'identificateur qui doit être utilisé pour appeler la fonction et demander son exécution. Dans ce cas, il sera systématiquement accompagné de parenthèses même vides lorsqu'il n'y a pas de paramètres à fournir.

ATTENTION !!! Une utilisation sans parenthèse entraîne la prise en compte de l'adresse mémoire de l'implantation de la fonction. Cela représente une valeur de *pointeur de fonction*. Cette notion n'est pas abordée dans ce document.

Liste typée de paramètres Les paramètres spécifiés au moment de la définition de la fonction s'appellent des paramètres formels. Chaque paramètre est défini par son type et son nom. Il y a autant de couple *type nom* que de paramètres. Si la fonction n'admet pas de paramètre la liste typée est remplacée par `void` ou par rien (liste vide). Un paramètre correspond à une donnée dont la valeur provient de et/ou est transmise à la partie appelante. À ne pas confondre avec les variables locales à la fonction.

ATTENTION !!! L'entête d'une fonction NE SE TERMINE PAS par un point-virgule car il y a la définition du corps qui suit.

3.1.2 Corps d'une fonction

Il s'agit d'un bloc d'instructions dans lequel on retrouve les définitions des données nécessaires au traitement effectué par la fonction suivies des instructions correspondant à ce traitement. Les données définies dans le corps correspondent aux variables locales à la fonction. Ce sont des variables de travail qui n'existent plus une fois l'exécution de la fonction terminée. À ne pas confondre avec les paramètres formels.
Si la fonction est censée renvoyer une valeur, alors le corps de la fonction doit comporter une instruction `return` (ou plusieurs lorsqu'il y a des alternatives, une seule étant alors exécutée) :

```
return (expression);
```

L'expression est évaluée et la fonction s'arrête en communiquant la valeur obtenue à la partie appelante.
Si la fonction ne renvoie pas de valeur alors le corps de la fonction peut contenir (facultatif) une instruction `return` simple :

```
return ;
```

Exemple :

```
/* Définition d'une fonction déterminant la valeur maximum */
/* de 2 valeurs passées en paramètre et représentées       */
/* respectivement par les paramètres formels A et B        */
/* A et B représentent les valeurs qui seront données      */
/* au moment de l'appel de la fonction                     */

int max (int A, int B)
{  if (A > B) return A;
   else return B;
}

/* Définition d'une fonction déterminant la valeur maximum */
/* de 2 valeurs lues au clavier et représentées            */
/* respectivement par les variables locales C et D         */
/* cette fonction n'admet pas de paramètres formels        */

int max_bis (void)       /* ou int max_bis ()  */
{  int C, D;       /* variables locales */

   /* saisie des données */
   printf("Donner 2 valeurs : ");
   scanf("%d %d", &C, &D);
   /* renvoi de la valeur max */
   if (C > D) return C;
   else return D;
}

/* Définition d'une fonction affichant la valeur maximum   */
/* de 2 valeurs lues au clavier et représentées            */
/* respectivement par les variables locales C et D         */
/* cette fonction n'admet pas de paramètres formels et     */
/* ne renvoie pas de valeur.                               */

void max_ter (void)      /* ou void max_ter ()  */
{  int C, D, E;      /* variables locales */

   /* saisie des données */
   printf("Donner 2 valeurs : ");
   scanf("%d %d", &C, &D);
   /* calcul de la valeur max */
   if (C > D) E = C;
   else E = D;
   /* affichage du résultat */
   printf("Valeur max = %d\n", E);
   return;      /* ou pas d'instruction return  */
}
```

3.1.3 Cas particulier de la fonction `main`

Nous rappelons juste ici le cas particulier de la fonction `main` déjà traité lors de la présentation des bases du langage.

Le proptotype normalisé de cette fonction est :

```
int main(int argc, char * argv[])
```

avec `argc` le nb d'arguments placés sous forme d'une chaîne de caractères dans le tableau `argv` (donc `argc` donne la taille effective de `argv`).

3.2 Déclaration d'une fonction

La déclaration d'une fonction reprend l'entête de la fonction telle qu'elle est donnée dans la définition de la fonction, seuls les noms des paramètres formels peuvent être omis, mais pour des questions de lisibilité il est préférable de les laisser.
ATTENTION !!! La déclaration d'une fonction DOIT SE TERMINER PAR un point-virgule (;) :

```
type_val_retour nom_fonc(liste_typée_de_paramètres);
```

La présence de la déclaration d'une fonction permet au compilateur de considérer que, dans ce qui suit, la fonction est connue et peut être utilisée conformément aux indications rappelées dans la déclaration. Il peut ainsi vérifier le type de la valeur de retour et le type et le nombre des paramètres fournis lors de l'appel. On parle de prototypage. Le contrôle de la validité d'un appel est un des apports non négligeables de la normalisation du langage C (Norme Ansi – voir [Dri90]). Avant cela, aucun contrôle n'était effectué et les problèmes survenaient seulement au moment de l'exécution.
Exemple des déclarations des fonctions définies dans la section 3.1 page 11 :

```
int max(int, int) ;
int max(int A, int B);
int max_bis(void);      /* ou int max_bis();    */
void max_ter(void);     /* ou void max_ter();   */
```

Si une fonction doit être utilisée avant sa définition (référence en avant) ou si la définition se trouve dans un autre fichier source (compilation séparée), il est obligatoire d'introduire la déclaration de la fonction avant son utilisation.

3.3 Appel d'une fonction

Un appel de fonction est une expression. On doit donc le retrouver dans une instruction, à l'endroit du programme où l'on a besoin que le traitement réalisé par la fonction soit effectué.
Un appel est constitué du nom de la fonction accompagné de la liste des valeurs représentant les paramètres. Ces valeurs peuvent être des constantes, des variables ou des expressions. L'ordre dans lequel elles sont données et leur type doivent être conforme à la définition de la fonction. Ceci est contrôlé lors de la compilation.
Les paramètres donnés lors de l'appel sont appelés les paramètres effectifs (ou arguments), par opposition aux paramètres formels figurant dans la définition de la fonction.
Exemples d'appels :

```
int val_max, nb1, nb2 ;

/* pour que l'appel suivant rende un résultat cohérent,   */
/* il est indispensable que nb1 et nb2 aient reçu         */
/* des valeurs au cours d'instructions précédemment       */
/* exécutées                                              */
...
val_max = max(nb1, nb2);

/* on peut aussi donner une valeur entière comme */
/* paramètre effectif                            */
nb2 = max(nb1, 35);

/* la fonction est définie sans paramètres donc pas */
/* de paramètres effectifs lors de l'appel          */
nb1 = max_bis();
```

Évaluation d'un appel de fonction Au moment de l'exécution d'une instruction, les expressions qu'elle contient sont évaluées. C'est le cas de tout appel de fonction. L'évaluation d'un appel de fonction se fait selon le principe suivant :

1. Évaluation de chaque paramètre effectif.

2. Création des variables correspondant aux paramètres formels.
3. Recopie de la valeur de chaque paramètre effectif dans le paramètre formel correspondant.
4. Création des variables locales.
5. Exécution du corps de la fonction.
6. Eventuellement, récupération de la valeur de retour.
7. Arrêt de la fonction et suppression des variables correspondant aux paramètres formels et aux variables locales.

3.4 Focus sur le passage de paramètres

Les paramètres sont destinés à échanger des données entre un sous-programme et son programme appelant. Cette section est destinée à expliciter comment ce passage de paramètre est effectué en programmation quel que soit le langage utilisé.

3.4.1 Définitions

En programmation informatique, il existe 3 types de passage de paramètres entre un sous-programme et son programme appelant :
— par valeur,
— par référence (ou par adresse),
— par copie-recopie.

Vocabulaire Tout d'abord rappelons quelques termes :
— *Paramètres formels :* il s'agit des paramètres apparaissant dans la description du sous-programme.
— *Paramètres effectifs :* il s'agit des paramètres/arguments fournis lors de l'appel du sous-programme.
Un exemple pour fixer les idées :

```
int f (int a, int b)
{
    return a + b ;
}

main()
{
    int i, j, x ;
    ...
    i = 10 ;
    j = 40 ;
    x = f(i,j) ;
...
}
```

Dans cet exemple, `a` et `b` sont les paramètres formels de la fonction `f`, alors que `i` et `j` sont les paramètres effectifs/arguments fournis lors de l'appel de `f` par la fonction `main`.

Passage de paramètre par valeur Il y a copie de la valeur du paramètre effectif dans le paramètre formel. Ce qui signifie que, même si le paramètre formel est modifié lors de l'exécution du sous-programme, le paramètre effectif restera inchangé. C'est la règle qui est *systématiquement* appliquée en langage C (voir section 3.3 page précédente).
Exemple :

```
int f (int a)
{
    a++ ;
    return a ;
}
```

```
main()
{
    int i, x ;
    ...
    i = 10 ;
    x = f(i) ;
    ...
}
```

Quand on déroule le programme à l'aide d'un tableau de situation (c'est-à-dire un état des différentes variables au cours de l'exécution), on obtient :

Variables	Valeurs en fc des étapes d'exécution
`i`	10
`x`	11
`a`	10 11
Valeur retournée par `f`	11

Notons que `i` n'a pas changé de valeur même quand `a` a été incrémentée.

Passage de paramètre par référence On établit un lien entre paramètre effectif et paramètre formel, de telle sorte que toute modification du paramètre formel est automatiquement répercutée sur le paramètre effectif : le paramètre formel occupe la même place mémoire (est à la même adresse) que le paramètre effectif. Ce mode de passage *n'existe pas en langage C.*
Exemple dans un autre langage (le langage PASCAL) :

```
VAR i : integer ;

PROCEDURE p (VAR y : integer) ;
    (* C'est le mot-clé VAR qui indique
       le mode de transmission du paramètre *)
BEGIN
    y := y + 1 ;
END ;

BEGIN
    ...
    i := 10 ;
    p(i) ;
    ...
END.
```

Quand on déroule le programme à l'aide d'un tableau de situation, on obtient :

Variables	Valeurs en fc des étapes d'exécution
`i`	10 11
`y`	voir `i`

Passage de paramètre par copie-recopie Il y a copie de la valeur du paramètre effectif dans le paramètre formel, puis, à la fin de l'exécution du sous-programme, il y a recopie de la valeur du paramètre formel dans le paramètre effectif. Ce qui signifie que si la valeur du paramètre formel a été modifiée durant l'exécution du sous-programme alors, *à la fin du sous-programme*, la valeur du paramètre effectif sera elle-aussi modifiée. Ce type de passage de paramètre *n'existe pas en langage C* mais en ADA pour certains cas.
Exemple : Reprenons l'exemple donné précédemment en section 3.4.1 en considérant qu'il s'agit d'un passage par copie-recopie. Quand on déroule le programme à l'aide d'un tableau de situation, on obtient :

Variables	Valeurs en fc des étapes d'exécution
i	10 11
y	10 11

(le lien entre `i` et `y` est mémorisé lui-aussi, pour pouvoir faire ensuite la recopie)

Remarque importante : s'il n'y a pas d'effet de bord [1] alors le passage par référence correspond exactement au passage par copie-recopie.

3.4.2 Mise à jour de données au travers d'un passage de paramètre en langage C

En langage C, le passage par référence et le passage par copie-recopie étant interdits, comment peut-on mettre à jour des données à travers un passage de paramètre ?
L'idée est la suivante : puisque le seul passage autorisé est par valeur, alors il faut passer une valeur qui permettra d'atteindre la zone à modifier, par exemple l'adresse de cette zone en mémoire (et c'est ainsi qu'on découvre une nouvelle utilisation des pointeurs en langage C).
Rappelons donc que la syntaxe à utiliser est la suivante :

- soit la donnée `d`, l'adresse de cette donnée se note `&d`,
- soit une adresse `a`, le contenu de cette adresse (c'est-à-dire la donnée qui est à l'adresse `a` en mémoire) se note `*a`,
- la déclaration d'une variable de type adresse d'une donnée se note :

`type_de_la_donnée_adressée * nom_variable_adresse`

Reprenons alors l'exemple cité pour le passage par référence en section 3.4.1 page précédente et traduisons-le en langage C :

```
void p (int * y)  /* y est donc l'adresse de la zone à changer        */
{
    *y = *y + 1 ; /* on modifie cette zone en passant par son adresse */
}

main()
{
    int i  ;
    ...
    i = 10 ;
    p(&i) ;    /* appel de p avec l'adresse de i puisque  */
               /* c'est i que l'on veut modifier          */
    ...
}
```

Quand on déroule le programme à l'aide d'un tableau de situation, on obtient :

Variables	Valeurs en fc des étapes d'exécution
i	10 11
y	adresse de `i`

Cas particulier : passage d'un tableau en paramètre Le nom du tableau représente une valeur constante correspondant à l'adresse du premier élément. Par conséquent, on ne peut pas lui appliquer l'opérateur d'adressage `&`. D'une part parce qu'un nom de tableau à la différence d'un élément de tableau n'est pas une variable et que, de toutes façons, il représente déjà une adresse.
Lorsque, lors de l'appel, on passe le nom d'un tableau (= adresse d'un élément), du côté de la définition de la fonction on doit avoir un pointeur sur le type des éléments du tableau. On dispose de 2 notations possibles et équivalentes pour définir un paramètre formel correspondant à un tableau.

1. Un *effet de bord* est un effet secondaire obtenu "en plus" lors de l'évaluation d'une instruction. C'est très souvent le cas quand on modifie des variables globales à l'intérieur d'un sous-programme (voir des exemples dans les sections 3.5.2 page 20 et 3.5.3 page 21).

Si le paramètre effectif est le nom d'un tableau à une dimension contenant des éléments d'un type donné, représenté ici par `type_element`, on pourra définir le paramètre correspondant (nommé ici `ptr_tab`) comme suit :

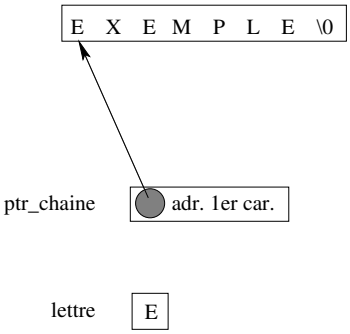
```
type_element * ptr_tab
```

ou bien

```
type_element ptr_tab[]
```

Exemple :

```
/* COMPTE_OCCURRENCE_LETTRE : fonction qui renvoie le     */
/* nombre d'occurrences d'une lettre dans une chaîne      */
/* de caractères => renvoie un entier                     */
/* ptr_chaine : adresse du premier caractère de la chaîne */
/* lettre : caractère à rechercher dans la chaîne         */

int compte_occurrence_lettre (char * ptr_chaine, char lettre)
{ int indice_lettre, nb_occ = 0;

  for(indice_lettre = 0 ; ptr_chaine[indice_chaine] != '\0'; indice_chaine++)
        /* on parcourt la chaîne caractère par caractère  */
        /* et on s'arrête lorsqu'on rencontre le marqueur */
        /* de fin de chaîne de caractères \0              */
     if(ptr_chaine[indice_chaine] == lettre)
           nb_occ ++;        /* on incrémente le compteur  */
                             /* lorsqu'il s'agit de la     */
                             /* bonne lettre               */
  return(nb_occ); /* on renvoie la valeur calculée */
}
```



Etat des paramètres formels après l'appel
compte_occurence_lettre ("EXEMPLE", 'E')

Pour un tableau à 2 dimensions on pourra définir le paramètre correspondant (nommé ici `ptr_tab`) comme suit :

```
type_element * ptr_tab[]
```

ou bien

```
type_element ptr_tab[][dim]
```

où `dim` est le nombre d'éléments maximum de la seconde dimension.
Exemple :

```
#include <stdio.h>
#define MAX_LG 20
#define MAX_COL 10         /* définition de constantes symboliques     */
                           /* représentant le nombre maximum de lignes */
                           /* et de colonnes pour une matrice          */
```

```
/* LIRE_MATRICE : demande les dimensions réelles de la matrice et */
/* procède à la saisie de chaque élément                          */
/* ptr_mat : représente la matrice (= adresse premier élément)    */
/* ptr_lg : représente l'adresse de la variable qui mémorisera le */
/*                 nombre de lignes réels de la matrice           */
/* ptr_col : représente l'adresse de la variable qui mémorisera le*/
/*                 nombre de colonnes réels de la matrice         */
/* la fonction mettant à jour le nombre de ligne et de colonnes il*/
/* est nécessaire de faire du passage par adresse                 */

void lire_matrice (int ptr_mat[][MAX_COL],   int *ptr_lg, int *ptr_col)
{  int i, j;

   printf("dimensions de la matrice (20x10 max): ");

   /* contrôle de la validité du nombre de lignes */
   do
   { printf("ligne = ");
     scanf("%d", ptr_lg); /* pas besoin de & puisqu'on a  */
                          /* déjà l'adresse               */
   } while (*ptr_lg <0 || *ptr_lg >= MAX_LG);

   /* contrôle de la validité du nombre de colonnes */
   do
   { printf("colonne = ");
     scanf("%d", ptr_col);
   } while (*ptr_col <0 || *ptr_col >= MAX_COL);

   /* saisie de chaque élément */
   for(i=0; i< *ptr_lg ; i++)
   { printf("ligne %d\n", i+1);
     for(j=0; j < *ptr_col; j++)
       scanf("%d", &ptr_mat[i][j]);

   /* ptr_mat[i][j] represente un element (= un entier) de  */
   /* la matrice correspond à *(*(ptr_mat + i) +j)          */
   /* &ptr_mat[i][j] peut s'écrire &(*(*(ptr_mat + i) +j))  */
   /* & annulant le premier * on a aussi *(ptr_mat + i) + j */
   }
}

/* ECRIRE_MATRICE : rapelle les dimensions réelles de la matrice et */
/* affiche chaque élément ligne par ligne                           */
/* ptr_mat : représente la matrice (= adresse premier élément)      */
/* nb_lg : représente la valeur connue du nombre de lignes          */
/* nb_col : représente la valeur connue du nombre de colonnes       */
/* la fonction ne modifiant pas ces valeurs, le passage par valeur  */
/* suffit                                                           */

void ecrire_matrice (int ptr_mat[][MAX_COL], int nb_lg, int nb_col)
{ int i, j;

  printf("dimensions de la matrice %d x %d\n", nb_lg, nb_col);

  for(i=0; i< nb_lg ; i++)
  {
```

```
      for(j=0; j < nb_col; j++)
      printf("%d ", ptr_mat[i][j]);
      printf("\n");
  }
}

void main (void)
{ int matrice[MAX_LG][MAX_COL];
  int nb_lg, nb_col;

  lire_matrice(matrice, &nb_lg, &nb_col);
  ecrire_matrice(matrice, nb_lg, nb_col);
}
```

3.4.3 Conclusion sur le passage des paramètres

En langage C, le seul passage de paramètre utilisé est le *passage par valeur* (la valeur du paramètre effectif est recopiée dans le paramètre formel).
Il faut savoir quand même qu'il existe aussi, dans d'autres langages, d'autres modes de passage de paramètres (par référence ou par copie/recopie).
D'autre part, pour éviter les effets de bord, il faut faire très **ATTENTION, lors de la manipulation, a fortiori de la modification,** de variables globales dans un sous-programme !

3.5 Exercices sur les passages de paramètres

3.5.1 Exercice 1

Question 1 Soit le programme suivant, donner le tableau de situation correspondant.

```
int i;
int a[2];
void p (int x)
{
i = i + 1;
x = x + 2;
}

main()
{
a[0] = 10;
a[1] = 20;
i = 0;
p(a[i]);
}
```

Question2 Reprenons le programme précédent avec les modifications suivantes :
- *void p (int * x)* à la place de *void p (int x)* ;
- **x = *x + 2* à la place de $x = x + 2$;
- l'appel de *p* devient alors : *p(&a[i])*.

Donner le tableau de situation correspondant.

Question3 Reprenons le programme précédent et imaginons qu'il puisse s'agir d'un passage de paramètre par copie-recopie. Donner le tableau de situation correspondant.

3.5.2 Exercice 2

Question 1 Soit le programme suivant, donner le tableau de situation correspondant.

```
int i;
int b[2];
void q (int x)
{
i = 0;
x = x + 2;
b[i] = 10;
i = 1;
x = x + 2;
}

main()
{
b[0] = 1;
b[1] = 1;
i = 0;
q(b[i]);
}
```

Question 2 Reprenons le programme précédent avec les modifications suivantes :

- *void q (int * x)* à la place de *void q (int x)* ;
- **x = *x + 2* à la place de $x = x + 2$;
- l'appel de q devient alors : *q(&b[i])*.

Donner le tableau de situation correspondant.

Question 3 Reprenons le programme précédent et imaginons qu'il puisse s'agir d'un passage de paramètre par copie-recopie. Donner le tableau de situation correspondant.

3.5.3 Exercice 3

Soit le programme suivant, donner le tableau de situation correspondant.

```
int y;
int i;
int z;
int f (int x)
{
y = y + 1;
x = x + 1;
return(x + y);
}

main()
{
y = 10;
i = 1;
z = f(i) + y;
}
```

3.5.4 Exercice 4 : Cas d'appels multiples

Soit l'exemple suivant, donner le tableau de situation correspondant :

```
int i, j;
int p (int y, int x)
{
int z;
z = x;
z++;
return y + x;
}

main()
{
i = 10;
j = 5;
i = p(i,j);
i = p(i,j);
}
```

3.5.5 Exercice 5 : Cas particulier des tableaux en C

Soit l'exemple suivant, donner le tableau de situation :

```
#define N 10
void init_tab (int TF[N])
{
int i;
for (i = 0; i < N; i++)
        TF[i] = 0;
}

main()
{
int TE[N];
init_tab(TE);
}
```

3.5.6 Exercice 6 : Ecriture de fonctions

Énoncé 3 *Soit le tableau $T[N]$ d'entiers, écrire une fonction qui calcule la moyenne des valeurs de T.*

Énoncé 4 *Soit le tableau $T[N]$ d'entiers, écrire une fonction qui calcule la moyenne, le max et le min des valeurs de T.*

Énoncé 5 *Soit le tableau $T[N]$ d'entiers, écrire une fonction qui effectue un décalage circulaire à gauche de T. Exemple : T avant l'appel contient $\{v_1, v_2, \ldots, v_n\}$ et après l'appel T contient $\{v_2, \ldots, v_n, v_1\}$.*

Énoncé 6 *Écrire une fonction qui renvoie dans un tableau t_3 la somme indice à indice des valeurs de 2 autres tableaux t_1 et t_2. Exemple : avec $t_1 = \{1, 2, 3, 4\}$ et $t_2 = \{4, 3, 2, 1\}$, on aura $t_3 = \{5, 5, 5, 5\}$.*

Énoncé 7 *Soit $Mat[M][N]$ une matrice contenant les N notes de M élèves. Écrire une fonction qui renvoie dans un tableau l'indice des élèves qui sont reçus (moyenne $\geq$ à 10). Utiliser le résultat de l'énoncé 3.*

Énoncé 8 *Écrire une fonction qui renvoie le code ASCII d'un caractère.*

Énoncé 9 *Écrire une fonction qui encode une chaîne de caractères par un tableau d'entiers en utilisant le résultat de l'énoncé 8.*

Énoncé 10 *Écrire une fonction qui renvoie un "booléen" indiquant si une valeur X donnée apparaît dans un tableau $T[N]$.*

Énoncé 11 *Écrire une fonction qui renvoie :*
- *un "booléen" indiquant si une valeur X donnée apparaît dans un tableau $T[N]$,*

— *un entier donnant la (première) position de X dans T si X est dans T.*

Énoncé 12 *Écrire une fonction qui remplace la première occurrence de la valeur X par la valeur Y dans un tableau $T[N]$ (on sait que X apparaît dans T).*

Énoncé 13 *Écrire une fonction qui remplace la dernière occurrence de la valeur X par la valeur Y dans un tableau $T[N]$ (on sait que X apparaît dans T).*

Énoncé 14 *Idem l'énoncé 12, mais on ne sait plus si X apparaît dans T. Il faut donc renvoyer un "booléen" indiquant si le remplacement a eu lieu.*

Énoncé 15 *Écrire une fonction qui remplace toutes les occurrences de la valeur X par la valeur Y dans un tableau $T[N]$ (on ne sait pas si X apparaît dans T). On renverra le nombre d'occurrences remplacées.*

Énoncé 16 *Idem l'énoncé 15, mais en renvoyant en plus la liste sous la forme d'un tableau des indices de ces occurrences.*

4 Compilation séparée en langage C

On a vu lors de la présentation des bases du langage C la manière de construire un exécutable correspondant à un fichier source.
Malheureusement, dès qu'on développe un gros projet en C, pour respecter une contrainte de modularité (c'est-à-dire faire des petites unités de code facilement maintenables et liées par une certaine sémantique), il arrive très vite que le programme à compiler soit réparti dans plusieurs fichiers sources et plusieurs fichiers "header" (voir un exemple en section 4.1). La méthode présentée initialement n'est donc plus tout à fait exploitable.
Il faut en effet penser à compiler *chaque* fichier source et à faire une édition de lien qui prend en compte *tous* ces fichiers pour créer un exécutable. On parle alors de compilation séparée.
Ce mécanisme peut être automatisé en exploitant une commande spécifique du système d'exploitation UNIX, la commande `make` qui exploite à son tour un fichier texte spécial, dénommé en général `Makefile`.

4.1 Exemple de la modularisation d'un programme

On va considérer l'exemple d'un programme consistant à :
— à construire une boite à outils fournissant un type spécifique `MonTypeBAO` et les fonctions permettant de manipuler ce type `initMonTypeBAO`, `majMonTypeBAO`, `printMonTypeBAO`, `removeMonTypeBAO`,
— à utiliser ce type pour une application particulière `appli`.
Si on ne respecte pas totalement la contrainte de modularité et que l'on met tout dans un unique fichier source `source.c`, on pourrait donc avoir :

```
#include <stdio.h>

typedef ... MonTypeBAO ;

MonType initMonTypeBAO(...)
   {
    ...
   }

MonType majMonTypeBAO(...)
   {
    ...
   }

void printMonTypeBAO(...)
   {
    ...
```

```
   }

void removeMonTypeBAO(...)
   {
    ...
   }

void appli(MonTypeBAO truc)
   {
    ...
   }

int main()
   {
    MonTypeBAO machin ;
    appli(machin) ;
   }
```

Dans ce cas-là, il sera très simple de créer l'exécutable correspondant puisqu'il suffira de taper dans un shell :

```
gcc source.c
```

Maintenant, si on veut pouvoir utiliser notre "boite à outil" avec une seconde application, il va falloir la "séparer" de la première application. Cela se fait en transformant le fichier source `source.c` en trois fichiers :

— le fichier header `bao.h` qui va contenir le type et les prototypes des fonctions de la boite à outils et dont le but est d'être inclus dans tout fichier utilisant le type `MonTypeBAO` :

```
typedef ... MonTypeBAO ;

MonType initMonTypeBAO(...) ;
MonType majMonTypeBAO(...) ;
void printMonTypeBAO(...) ;
void removeMonTypeBAO(...) ;
```

— le fichier source `bao.c` qui va contenir les fonctions de la boite à outils et qui inclut donc le fichier header contenant le type :

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

MonType initMonTypeBAO(...)
   {
    ...
   }

MonType majMonTypeBAO(...)
   {
    ...
   }

void printMonTypeBAO(...)
   {
    ...
   }

void removeMonTypeBAO(...)
   {
    ...
   }
```

— le fichier source `appli1.c` correspondant à la première application et incluant aussi le fichier header contenant le type :

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

void appli(MonTypeBAO truc)
   {
    ...
   }

int main()
   {
    MonTypeBAO machin ;
    appli(machin) ;
   }
```

Pour créer l'exécutable, il faudra alors taper les commandes suivantes, les unes après les autres, dans un shell :

```
gcc -c bao.c
gcc -c appli1.c
gcc bao.o appli1.o
```

Il sera alors possible de créer la seconde application sur le même modèle que la première dans le fichier `appli2.c` :

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

...

int main()
   {
    ...
   }
```

et d'obtenir l'exécutable correspondant avec les commandes suivantes (la recompilation de la boite à outils étant inutile puisqu'on ne l'a pas modifiée) :

```
gcc -c appli2.c
gcc bao.o appli2.o
```

Dès qu'on manipule ainsi plus de 2 ou 3 fichiers, cela peut devenir très vite compliqué. La commande `make` et le fichier `Makefile` peuvent alors vous faciliter la tâche.

4.2 Format du fichier `Makefile`

Le fichier `Makefile` est un fichier texte qui se travaille donc sous un éditeur de texte quelconque et qui contient des directives de compilation utilisables par la commande `make` (voir section 4.3 page suivante). Basiquement, il s'agit d'une liste d'instructions séparées les unes des autres par une ligne vide, chaque instruction ayant la forme suivante :

entité-à-construire : liste des éléments servant à la construction <RC>
<TAB>commande unix permettant de construire l'entité <RC>

avec **<RC>** représentant le retour à la ligne et **<TAB>** représentant le caractère de tabulation.

Exemple Reprenons l'exemple de la section précédente, le fichier "Makefile" contiendra alors les lignes suivantes permettant de créer les fichiers objet correspondant à chaque fichier source ainsi que le fichier exécutable (appelé ici `appli1`) :

bao.o : bao.c bao.h <RC>
<TAB> gcc -c bao.c <RC>

<RC>
appli1.o : appli1.c bao.h <RC>
<TAB> gcc -c appli1.c <RC>
<RC>
appli1 : appli1.o bao.o <RC>
<TAB> gcc -o appli1 appli1.o bao.o <RC>

Remarque : la syntaxe d'un fichier `Makefile` peut être bien plus complexe que l'exemple donné ici. Le lecteur intéressé trouvera sans mal de la documentation sur ce sujet.

4.3 La commande `make`

La commande `make` permet de gérer les compilations des programmes en langage C. Elle nécessite la création d'un fichier `Makefile` (voir section 4.2 page précédente).
Son utilisation est la suivante (voir d'autres options de cette commande en utilisant le manuel en ligne) : dans une fenêtre "shell" taper la commande `make entité-à-construire`.
L'appel de la commande `make` créera alors l'entité-à-construire en faisant les compilations qui s'imposent et uniquement celles-là.
Dans l'exemple donné en sections 4.1 page 23 et 4.2 page précédente, on peut donc lancer 3 types de commande :

- `make bao.o`, ce qui correspondra à la création du code objet lié au fichier `bao.c`,
- `make appli1.o`, ce qui correspondra à la création du code objet lié au fichier `appli1.c`,
- `make appli1`, ce qui correspondra à la création du code exécutable lié à la fonction `main` du fichier `appli1.c` (et donc aussi, a fortiori, à la création des codes objet `bao.o` et `appli1.o`, s'ils n'existaient pas déjà ou s'ils n'étaient pas à jour).

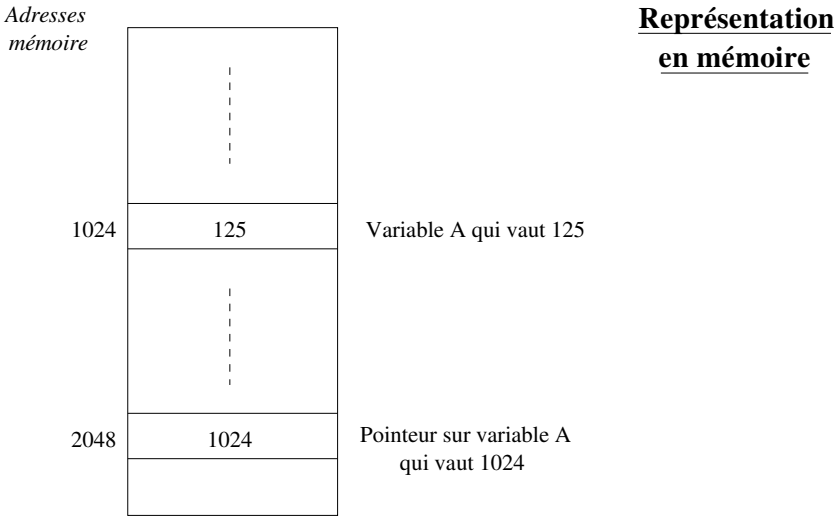
Ainsi, dans la plupart des cas, on ne lance la commande `make` que sur des entités exécutables.

5 Les pointeurs en langage C (aspect dynamique)

Dans la présentation des bases du langage C, nous avons décrit l'utilisation des pointeurs dans un aspect de gestion statique de la mémoire. Ici, après quelque rappels, nous nous intéresserons surtout à l'utilisation des pointeurs pour la gestion dynamique de la mémoire.

5.1 Définition

Un *pointeur* = variable dont la valeur correspond à une adresse.



5.2 Syntaxe en langage C

Les pointeurs correspondent à un type spécial : le type *pointeur*.

- en zone déclarative, on écrit : `type_que_l_on_veut * p;` (par exemple : `int * p` et alors `p` est un pointeur sur un entier),
- dans le corps du programme, on utilise soit `p` pour manipuler l'adresse, soit `*p` pour manipuler la valeur pointée (le symbole `*` est appelé un constructeur).

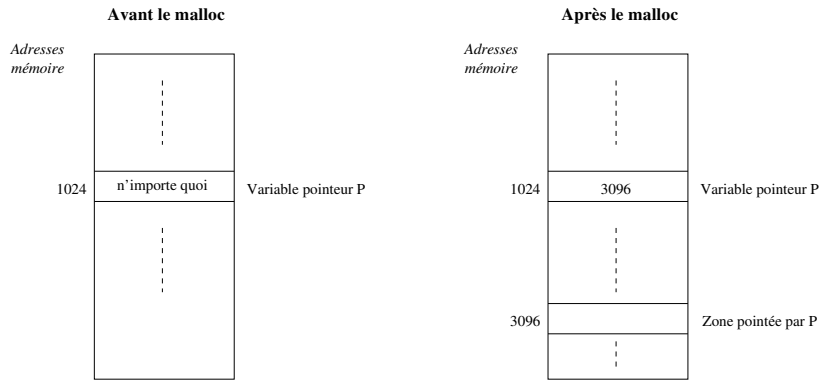
Opérations autorisées En langage C, il existe 6 opérations possibles sur les pointeurs.

Les 4 premières ont déjà été décrites lors de la présentation des bases du langage :

- L'affectation :
  `p = q ;` (`p` et `q` doivent être des pointeurs sur un même type),
  `p = NULL ;` (`NULL` est l'élément nul pour le type pointeur),
  `p = &x ;` (`x` est une variable de type quelconque noté `T` et `p` est une variable de type pointeur sur le type `T`) (le symbole `&` sert à obtenir l'adresse d'une variable).
- L'addition et la soustraction d'un pointeur avec un entier (on parle aussi de *décalage*) :
  `p = q + 1 ;` (`p` et `q` doivent être des pointeurs sur un même type et sur une même entité mémoire [2]),
  `p = q - 10 ;` (`p` et `q` doivent être des pointeurs sur un même type et sur une même entité mémoire).
- La soustraction de 2 pointeurs de même type :
  `i = p - q ;` (`p` et `q` doivent être des pointeurs sur un même type et sur une même entité mémoire et `i` est un entier).
- La comparaison de 2 pointeurs de même type :
  `p > q` (`p` et `q` doivent être des pointeurs sur un même type et sur une même entité mémoire).

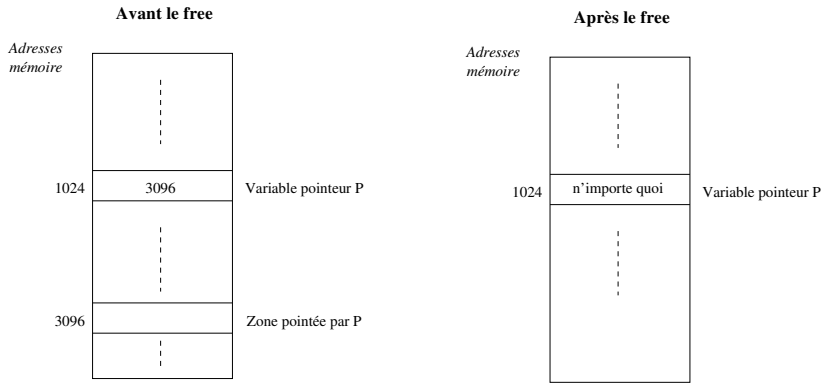
Les deux autres opérations sont destinées à permettre la gestion dynamique de la mémoire :

- La création dynamique (réservation et mise à jour) :
  `p = (type_de_p) malloc(taille_zone_pointée)` avec `p` une variable de type pointeur.
  Cette fonction C de la bibliothèque `stdlib.h` (voir annexe A page 38) permet de réserver une zone mémoire pour la valeur pointée par `p` et met à jour le pointeur `p` avec l'adresse de cette zone.



  Remarque : si, après l'exécution de cette instruction, le pointeur résultat est égal à `NULL`, cela signifie que la mémoire est saturée et qu'on ne peut plus faire d'allocation dynamique.
- La suppression dynamique :
  `free(p)` avec `p` une variable de type pointeur.
  Cette fonction C permet de libérer la zone mémoire réservée pour la valeur pointée par `p`.

2. C'est-à-dire sur un même tableau.



Attention : il arrive qu'après un `free(p)`, on puisse encore accéder à l'ancienne zone pointée par `p` (cela dépend de la machine et de son gestionnaire mémoire), c'est donc une source importante d'erreur ; un conseil : après `free(p)`, faire `p = NULL`.

Avantages L'utilisation des pointeurs permet une gestion dynamique de la mémoire et une souplesse d'implémentation.

Inconvénients Le temps de calcul devient parfois important car les zones allouées sont rarement attenantes l'une à l'autre et il faut alors prévoir un mécanisme permettant de :

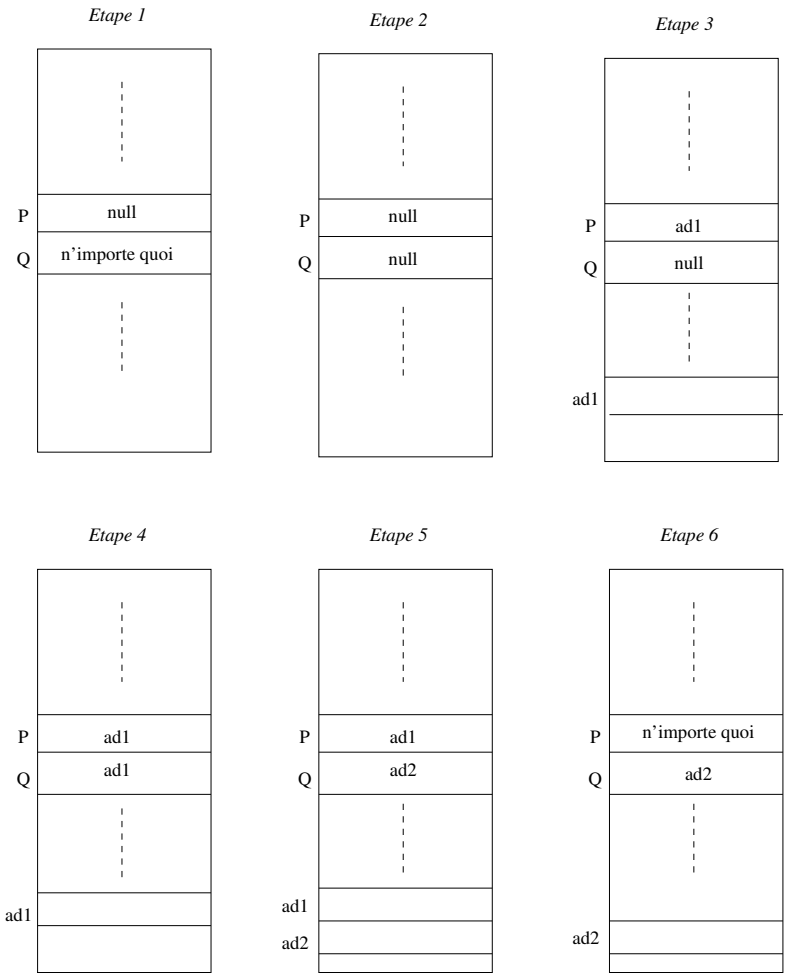
— de ne pas perdre l'accès à ces zones,
— de pouvoir passer d'une zone à l'autre.

5.3 Exemples de base sur les pointeurs

1. Soit le programme suivant :

```
...
#include <stdio.h>
...
int * P, * Q ;
...
P = NULL ;                          /* étape 1 */
Q = P ;                             /* étape 2 */
P = (int *) malloc(sizeof(int)) ;   /* étape 3 */
Q = P ;                             /* étape 4 */
Q = (int *) malloc(sizeof(int)) ;   /* étape 5 */
free (P);                           /* étape 6 */
...
```

Chaque étape est représentée par un schéma de la mémoire, et le déroulement du programme donne la chose suivante :



2. Soit le programme suivant :

```
int * p, * q ;
p = (int *) malloc(sizeof(int)) ;
printf("Après malloc : %d \ n", *p);
q = p ;
printf("Après q = p : %d et %d \n", *p, *q);
*p = 15 ;
printf("Après maj : %d et %d \n", *p, *q) ;
free(p) ;
printf("Après free : %d \n",*q);
```

Ici, le résultat obtenu à l'impression est :

```
Après malloc : Valeur1_quelconque
Après q = p : Valeur1_quelconque et Valeur1_quelconque
Après maj : 15 et 15
Après free : 15 /* ATTENTION: erreur possible (dépend de la machine~!) */
        /* En effet, q pointe sur une zone mémoire qui a été libérée~! */
```

3. Soit le programme suivant :

```
/* définition de nouveaux types */
typedef int zone  ;       /* zone sera le type int */
typedef zone * ptrzone ;  /* ptrzone sera le type pointeur sur zone */
void maj_zone_pointee(ptrzone p, zone val)
    {
     *p = val ;
```

```
}
int main()
{
ptrzone p ;
p = (int *) malloc(sizeof(int)) ;
printf("Avant maj : %d\n",*p);
maj_zone_pointee(p,10);
printf("Après maj : %d\n",*p);
}
```

Ici, le résultat obtenu à l'impression est :

```
Avant maj : Valeur_quelconque
Après maj : 10
```

Remarque : cette fonction permet de modifier une zone par l'intermédiaire de son adresse. C'est ainsi que l'on procède dans les langages où seul le passage par valeur est autorisé pour modifier des valeurs lors de l'appel d'un sous-programme (ex : le langage C).

5.4 Exemples sur les pointeurs + les structures

1. Soit le programme suivant :

```
typedef struct et_cellule {
   int valeur ;
   int * suivant ;
} cellule ;  /* cellule est une structure à 2 champs */
             /*  (entier, pointeur sur entier)       */
             /* cellule et struct et_cellule sont des synonymes */
int main()
{
   cellule c ;
   c.valeur = 10 ;
   c.suivant = (int *) malloc(sizeof(int)) ;
   *(c.suivant) = 11 ;
   printf( "%d %x %d ", c.valeur, c.suivant, *(c.suivant));
}
```

Ici, le résultat obtenu à l'impression est :

```
10 Adresse 11
```

2. Soit le programme suivant :

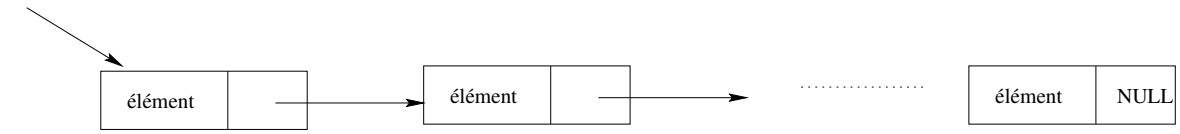
```
typedef struct et_cellule {
   int valeur ;
   struct et_cellule * suivant ;
   } cellule ;  /* cellule est une structure à 2 champs */
                /*  (entier, pointeur sur cellule)      */

typedef cellule * ptrcellule ; /* ptrcellule est un pointeur sur une cellule */
             /* cellule et struct et_cellule sont des synonymes */
             /* cellule *, struct et_cellule * et ptrcellule sont des synonymes */
int main()
   {
   ptrcellule p ;
   p = (ptrcellule) malloc(sizeof(cellule)) ;
   p->valeur = 10 ;
   p->suivant = (ptrcellule) malloc(sizeof(cellule)) ;
   p->suivant->valeur = 11 ;
   p->suivant->suivant = (ptrcellule) malloc(sizeof(cellule) ;
```

```
    p->suivant->suivant->valeur = 12;
    printf( "%x, %d, %x, %d, %x, %d ", p, p->valeur, p->suivant,
            p->suivant->valeur, p->suivant->suivant, p->suivant->suivant->valeur);
    }
```

Remarque : la notation `->` est une manière d'accéder aux champs d'une structure à partir d'un pointeur sur cette structure. `p->valeur` est donc strictement équivalent à `(*p).valeur`.
Ici, le résultat obtenu à l'impression est :

```
Adresse0 10 Adresse1 11 Adresse2 12
```

3. Représentation d'une pile en dynamique. Une pile est un pointeur sur la dernière cellule empilée :

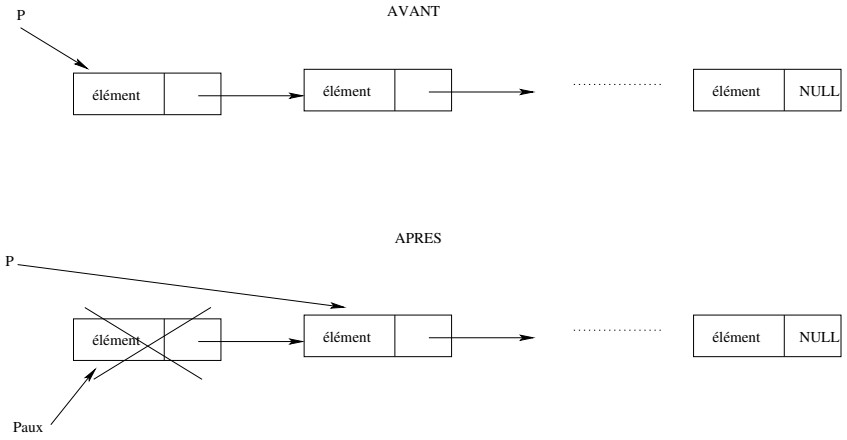



```
 typedef  struct cel {
    int info;
    struct cel *  suiv;
    } cel ;      /* cel = structure à 2 champs (entier, pointeur sur cel) */

typedef struct  cel *  pile ;              /* pile = pointeur sur une cel */
                                  /* cel et struct cel sont des synonymes */
                          /* cel *, struct cel * et pile sont des synonymes */
void creer_pile ( pile * p)
    {                                  /* création d'une pile à vide */
    *p = NULL ;
    }
int pile_vide (pile p)
    {              /* vérifier si une pile est vide (renvoie 0 si non */
                   /*    et une valeur diff. de 0 si oui)  */
    return (p == NULL);
    }
int sommet_pile (pile p, int * pcoderr)
    {                  /* récupérer la valeur en sommet d'une pile */
    *pcoderr = 0 ;    /* pcoderr = adresse du code d'erreur */
                      /*    (code  = 1 si Nok, 0 sinon)     */
    if (pile_vide(p))  {
        printf("erreur") ;
       *pcoderr = 1;
       }
    else return( (*p).info) ;
    }
void depiler (pile * p,  int * x, int * pcoderr)
    {                        /* supprime et renvoie le sommet de pile */
    pile  paux;
    *pcoderr = 1 ;                    /* pcoderr = idem sommet_pile */
    if (pile_vide(*p))  printf("erreur") ;
    else  {
       paux = *p ;
       *x = (*(*p)).info ;
       *p = (*(*p)).suiv ;
       free(paux);
       *pcoderr = 0 ;
       }
    }
void empiler (pile * p, int x, int * pcoderr)
```

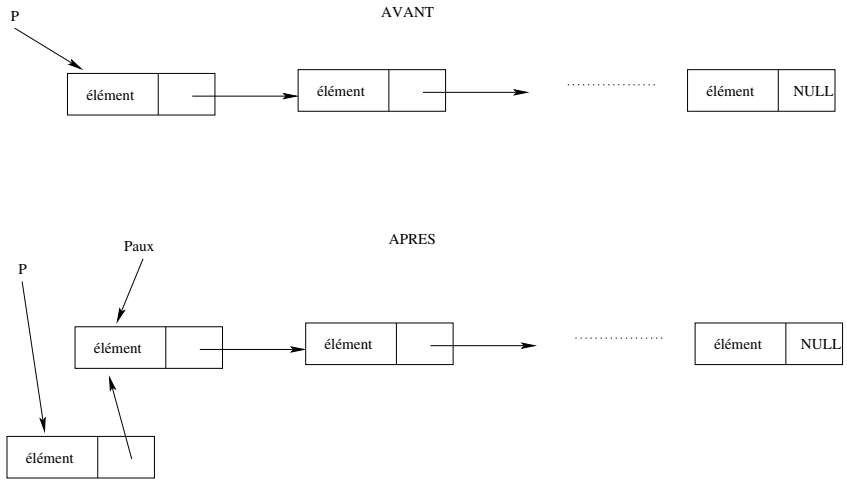
```
{                                        /* rajoute une valeur à la pile */
pile paux ;
*pcoderr = 1 ;                           /* pcoderr = idem sommet_pile */
paux = *p ;
*p = (pile)malloc(sizeof(struct cel));
if (*p == NULL)  printf("erreur") ;
else {
   (*(*p)).info = x ;
   (*(*p)).suiv = paux ;
   *pcoderr = 0 ;
   }
}
```

Le fonctionnement de *depiler* et de *empiler* est décrit par les figures suivantes

Fonctionnement de "depiler"



Fonctionnement de "empiler"



5.5 Conclusion sur les pointeurs

Les pointeurs sont des variables dont la valeur est une adresse. Ils servent à faire de l'allocation dynamique de mémoire et à "simuler" un passage de paramètre par référence (voir section 3.4 page 15).

5.6 Exercices sur les pointeurs

5.6.1 Exercice 1

En utilisant le type défini pour l'exemple de la pile, considérons le code suivant :

```
pile * p ;
x = (*(*p)).info ;
x = (*p).info ;
x = p->info ;
x = (*p)->info ;
```

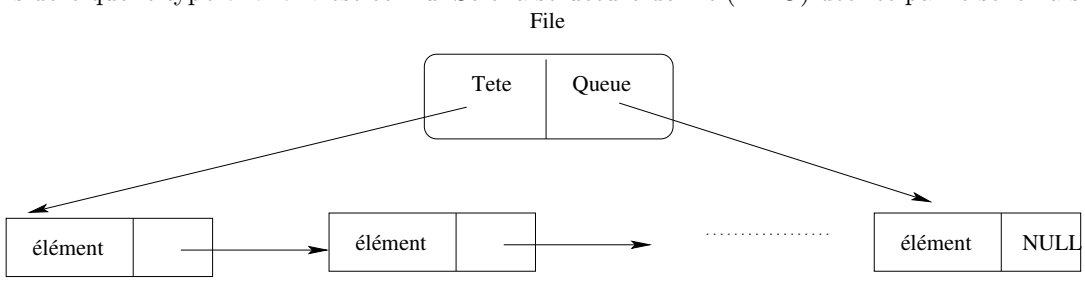
Dîtes si ces expressions sont correctes et si oui à quoi correspondent-elles (type de la variable `xx`) ?

5.6.2 Exercice 2

Réécrire l'exemple de la pile en utilisant la notation `->` et en considérant que les fonctions `creer_pile`, `empiler` et `depiler` renvoient la pile mise à jour (il va donc bien-sûr falloir modifier le prototype de ces fonctions).

5.6.3 Exercice 3

On considère que le type `element` est connu. Soit la structure de file (FIFO) décrite par le schéma suivant.



Proposez un type dynamique en langage C alliant les structures et les pointeurs permettant de simulant une file. Ecrivez en C les fonctions permettant :

- de créer une file vide,
- de vérifier si une file est vide ou pas,
- d'ajouter une nouvelle valeur en tête de la file,
- d'ajouter une nouvelle valeur en queue de la file,
- d'enlever la valeur de la tête de file,
- d'enlever la valeur de la queue de file.

Qu'en déduisez-vous ?

6 Manipulation des fichiers de données

Les entrées-sorties se font soit sur les fichiers standards (`stdin`, `stdout`, `stderr`), soit sur des fichiers de données enregistrés sur le disque. Il s'agit alors de fichiers textes (suite de caractères ASCII) ou de fichiers binaires (suite de blocs d'octets correspondant par exemple à des enregistrements).
Toutes les fonctions d'entrées-sorties se trouvent dans la bibliothèque standard stdio.h. Ne sont données ici que les déclarations associées aux principales fonctions. Voir une liste plus complète en section A page 38.

6.1 Notion de descripteur de fichier

Pour manipuler un fichier de données dans un programme C, il faut utiliser un pointeur sur un type prédéfini dans stdio.h : le type `FILE`.

```
/* pointeur necessaire a l'utilisation */
/* d'un fichier de données             */

FILE * ptr_fic;
```

Un élément de type `FILE`, appelé *descripteur de fichier*, est une structure contenant les informations relatives au fichier auquel il est associé (mode d'ouverture du fichier, type du fichier, position courante, ...).
Le programmeur n'a pas accès directement au contenu de cette structure. Il ne peut manipuler le fichier que via le pointeur associé dans la mesure où celui-ci a reçu l'adresse du descripteur. Ce pointeur est appelé *nom interne* du fichier.

6.2 Ouverture d'un fichier : la fonction `fopen`

Avant de pouvoir être utilisé, tout fichier de données doit être ouvert. Pour cela, il faut d'abord préciser le nom externe du fichier. Il s'agit d'une chaîne de caractères correspondant au nom ou au chemin d'accès caractérisant le fichier. Il faut également préciser le mode d'ouverture souhaité. À partir de ces paramètres, la fonction `fopen` procède à l'ouverture du fichier et renvoie soit l'adresse du descripteur associé au fichier ouvert (signe que l'ouverture s'est bien passée), soit `NULL` si l'ouverture n'a pu être effectuée.

```
FILE * fopen (char* NOM, char* MODE) ;
```

Attention!! Il faut TOUJOURS tester la valeur du pointeur après ouverture du fichier et avant toute opération sur celui-ci. En effet, si l'ouverture ne s'est pas faite pour une raison ou une autre (fichier inexistant, problèmes de droits d'accès, ...) la manipulation du pointeur entraînera une erreur d'adressage puisque la valeur du descripteur vaudra `NULL`.
Spécification du mode d'ouverture :

- `"r"` ouverture en lecture uniquement et `"r+"` en lecture/écriture. Si le fichier n'existe pas alors il y a erreur d'ouverture.
- `"w"` ouverture en écriture uniquement et `"w+"` en lecture/écriture. Si le fichier n'existe pas alors il est créé ; sinon son contenu est effacé.
- `"a"` ouverture en écriture uniquement et `"a+"` en lecture/écriture. Si le fichier n'existe pas alors il est créé ; sinon l'écriture se fait en fin de fichier (l'ancien contenu est conservé).

Attention!! Pour les opérations de lecture/écriture sur un même fichier, il faut gérer la position courante pour ne pas écrire sur les données à lire.

6.3 Fermeture d'un fichier de données : la fonction `fclose`

Lorsque le fichier, représenté par son nom interne, n'est plus utilisé par le programme, il doit être fermé.

```
int fclose (FILE* PTR_FIC) ;
```

6.4 Entrées-Sorties sur un fichier texte

On retrouve des fonctions similaires à celles utilisées pour les entrées-sorties standard. Elles admettent un paramètre supplémentaire précisant le nom interne du fichier sur lequel l'opération de lecture ou d'écriture doit s'effectuer.
On dispose de fonctions traitant :

- un seul caractère : `fgetc` et `fputc`,
- une chaîne de caractères : `fgets` et `fputs`,
- des entrées-sorties formatées : `fscanf` et `fprintf`.

6.5 Entrées-Sorties sur un fichier binaire

Les fonctions d'entrées-sorties ne sont pas les mêmes que pour un fichier texte. Il s'agit de transférer des blocs d'octets de même taille depuis (lecture), ou vers (écriture) le fichier ouvert. Pour cela on a besoin, en plus du nom interne du fichier (pointeur sur `FILE`), de l'adresse du premier bloc, de la taille d'un bloc et le nombre de blocs transférés simultanément.
La lecture et l'écriture d'un ou plusieurs blocs se font respectivement grâce aux fonctions `fread` et `fwrite`.

6.6 Autres fonctions nécessaires à la manipulation des fichiers

La bibliothèque stdio.h dispose également de fonctions permettant de contrôler les flux de données.
La fonction `feof` indique si la fin de fichier a été atteinte ou non en renvoyant respectivement 1 ou 0.
À toute opération de lecture ou d'écriture sur un fichier donné est associé un buffer fonctionnant normalement en mode bloc ; c'est-à-dire que celui-ci mémorise la donnée à transférer et le transfert n'est effectué que lorsque le buffer est plein.
La fonction `fflush` permet de courcircuiter ce mécanisme en provoquant le transfert immédiat des données.
D'autres fonctions permettent d'agir sur le buffer en changeant ses caractéristiques : `setbuf`, `setvbuf`.
Certaines fonctions permettent notamment de connaître et de modifier les informations relatives à la gestion de la position courante : `ftell`, `fseek`, `rewind`.

Pour connaître les spécifications des fonctions dont le nom est indiqué, ci-dessus, vous pouvez consulter l'annexe de ce document (voir section A page 38).
Exemple :

```
#include <stdio.h>

void main (void)
{ FILE * ptr_fic;
  char mot[11];
  int nb;

  /*ouverture en lecture du fichier de nom mon_fic */
  ptr_fic = fopen("mon_fic", "r");

  /* pour la suite ptr_fic désigne le fichier */
  if(ptr_fic != NULL)
  {
  /* ouverture OK => lecture de la 1ere ligne */
         fscanf(ptr_fic, "%10s %d", MOT, &NB);
         while(!feof(prt_fic))
         {
         /* traitement a faire sur tout le fichier */
                .............
                /* lecture ligne suivante */
               fscanf(ptr_fic, "%10s %d", MOT, &NB);
         }
         fclose(ptr_fic); /* fermeture du fichier */
  }
  else
  {
  /* message sur la sortie standard erreur */
         fprintf(stderr, "Probleme d'ouverture");
         /* sortie du programme */
         exit(1);
  }
  /* autres instructions */
  ................
 }
```

Références

[Del92] Delannoy (Claude). – *Exercices en langage C.* – Eyrolles, 1992. ISBN : 2-212-08251-7.

[Dri90] Drix (Philippe). – *Langage C norme ANSI : vers une approche orientée objet.* – Masson, 1990. ISBN : 2-225-81912-2.

[Got93] Gottfried (Byron S.). – *Programmation en C : cours et problèmes.* – Ediscience international, Mc Graw Hill, 1993. ISBN : 2-7042-1230-9.

[RS94] Rigaud (Jean-Marie) et Sayah (Amal). – *Programmation en langage C.* – Cépaduès, 1994.

Index pour les compléments sur le langage C

A Quelques bibliothèques et fonctions standards

Vous trouverez ici la description de quelques bibliothèques et fonctions standard couramment utilisées en langage C. Cette section complète la section 6 page 33 et la description de la bibliothèque stdio.h faite lors de la présentation des bases du langage.

A.1 Fichier stdio.h : Fonctions standards d'entrées-sorties

Cette librairie a déjà été évoquée lors de la présentation des bases du langage. Nous donnons ici quelques suppléments concernant la gestion des fichiers de données.
Attention : `char *` représente soit un tableau de caractères (= adresse du premier élément), soit un pointeur de caractère, c'est à dire une variable destinée à contenir ou contenant déjà l'adresse d'un caractère. Dans ce cas, il n'y a pas forcément de tableau "derrière" !

Manipulation de fichiers de données

`FILE * fopen(char*,char*)` Ouvre le ficher représenté par la première chaîne de caractère suivant le mode spécifié par la seconde chaîne de caractères et renvoie `NULL` si l'opération a posé des problèmes.

`int fclose(FILE*)` Ferme le fichier désigné par le paramètre fourni et renvoie 0 si la fermeture s'est correctement déroulée.

`int feof(FILE*)` Indique si la fin du fichier désigné par le paramètre fourni est atteinte en renvoyant une valeur non nulle, 0 sinon.

`int fseek(FILE*,long int, int)` Décale la position courante[3] du fichier désigné par le 1er paramètre du nombre d'octets spécifié par le 2ème paramètre à partir de l'adresse donnée en 3ème paramètre.

`long int ftell(FILE*)` Renvoie la position courante du fichier.

`void rewind(FILE*)` Déplace la position courante au début du fichier.

Lecture / Écriture sur les fichiers de données Ces opérations se font toujours à partir de la position courante. Celle-ci est ensuite déplacée du nombre d'octets correspondant aux données traitées.

`int getc(FILE*)` Lit un caractère simple dans le fichier désigné par le paramètre fourni.

`int putc(char,FILE *)` Écrit le caractère dans le fichier.

`int fscanf(FILE*,char*...)` Lit des données dans le fichier selon le format donné par la chaîne format.

`int fprintf(FILE*,char*...)` Écrit des données dans le fichier selon le format spécifié par la chaîne format.

`char* fgets(char*,int,FILE *)` Lit une chaîne d'une longueur donnée dans le fichier désigné par le second paramètre ; la chaîne est stockée à l'adresse fournie en premier paramètre.

`int fputs(char*,FILE *)` Écrit la chaîne donnée en paramètre dans le fichier indiqué.

`int fread(char*, int, int,FILE *)` Transfère des données d'un fichier de données binaire (= suite d'octets) dans la zone désignée par le premier paramètre. Les blocs d'octets à transférer ont une taille donnée par le deuxième paramètre, le troisième précisant le nombre de blocs à transférer.

`int fwrite(char*,int,int,FILE *)` Transfère, vers le fichier de données, des blocs d'octets dont la taille est donnée par le deuxième paramètre et le nombre par le troisième. L'adresse de la zone où se trouvent les données à transférer correspond à l'adresse fournie comme premier paramètre.

A.2 Fichier stdlib.h

Cette librairie a déjà été évoquée lors de la présentation des bases du langage. Nous donnons ici quelques suppléments concernant la gestion dynamique de la mémoire.
`void *` correspond à un pointeur générique c'est-à-dire à un pointeur dont le type est déterminé ultérieurement en fonction du type du paramètre passé. `size_t` est un type prédéfini dans cette bibliothèque et correspond à tout type d'entiers.

3. La position courante correspond à l'adresse du prochain octet qui fera l'objet d'une opération de lecture ou d'écriture.

`void *malloc(size_t)` Alloue un emplacement de la mémoire centrale correspondant au nombre d'octets indiqué en paramètre et renvoie l'adresse du début de la zone allouée (pointeur générique `void *` donc conversion nécessaire en pointeur sur le type de donnée allouée).

`void *calloc(size_t, size_t)` Alloue la place mémoire nécessaire pour stocker un tableau correspondant au nombre d'éléments indiqué par le premier paramètre, chaque élément ayant la taille indiquée par le second paramètre. Renvoie l'adresse du début de la zone allouée (pointeur générique `void *` donc conversion nécessaire en pointeur sur le type de donnée allouée).

`void *realloc(void *, size_t)` Change la taille de la zone désignée par l'adresse fournie en paramètre en réallouant le nombre d'octets indiqué par le second paramètre. Renvoie l'adresse du début de la zone allouée car celle-ci peut avoir changé suite à un problème de place. Attention s'il s'agit de réallouer la zone correspondant à un tableau, le second paramètre doit correspondre au nombre total d'octets de la nouvelle zone et non pas la taille d'un élément comme pour la fonction `calloc`.

`void free(void *ptr)` Libère la zone mémoire débutant à l'adresse indiquée.

B Synthèse sur les types de données pointeurs et structures

Le tableau ci-joint donne la syntaxe de quelques-unes des expressions les plus courantes avec les opérateurs et les types associés.

Déclaration	Type variable déclarée	Expressions autorisées	Types des expressions autorisées
type_x * *P* ;	*P* est un pointeur sur *type_x*	*P*	pointeur sur *type_x*
		*P	*type_x*
		&*P*	pointeur sur pointeur de *type_x*
type_x X ;	*X* est une variable de type *type_x*	*X*	*type_x*
		&*X*	pointeur sur *type_x*
struct etiq { *type1* c_1 ; *type2* c_2 ; } *S* ;	*S* est une structure à 2 champs appelée *struct etiq*	*S*	*struct etiq*
		$S.c_1$	*type1*
		$S.c_2$	*type2*
		&*S*	pointeur sur *struct etiq*
struct etiq { ...} * *P* ;	*P* est un pointeur sur une structure de type *struct etiq*	*P*	pointeur sur *struct etiq*
		*P	*struct etiq*
		&*P*	pointeur sur un pointeur sur *struct etiq*
		$(*P).c_1$	*type1*
		P-> c_1	*type1*
		$(*P).c_2$	*type2*
		P-> c_2	*type2*
typedef *struct etiq* { ...} *type_st*; *type_st S* ; typedef *type_st* * *ptr_st*; *ptr_st P* ;	*S* est une variable de type *struct etiq* mais aussi de type *type_st*. *P* est une variable de type pointeur sur une *struct etiq* mais aussi sur une variable de type *type_st*.	*S*	*type_st* ou *struct etiq*
		*P	*type_st* ou *struct etiq*
		$S.c_1$	*type1*
		$(*P).c_1$	*type1*
		P-> c_1	*type1*
		$S.c_2$	*type2*
		$(*P).c_2$	*type2*
		P-> c_2	*type2*
		P	pointeur sur *type_st*, pointeur sur *struct etiq*, *ptr_st*
		&*S*	pointeur sur *type_st*, pointeur sur *struct etiq*, *ptr_st*
		&*P*	pointeur sur pointeur sur *type_st*, ou sur pointeur sur *struct etiq*, ou sur *ptr_st*

C Un dernier exercice : un secrétariat médical

Énoncé 17 *On veut implémenter une base de données de secrétariat médical sachant que :*

- *il y a plusieurs médecins éventuellement de spécialité médicale différente (pédiatre, oto-rhino, ...),*
- *chaque médecin est une personne avec une spécialité,*
- *chaque médecin a sa liste de clients* (dans un premier temps, nous poserons comme hypothèse qu'il n'y a pas d'intersection avec les autres médecins !),
- *chaque client est une personne avec un dossier médical,*
- *chaque dossier médical est une liste de maladies,*
- *chaque maladie est représentée par son nom, la date du diagnostic de la maladie et l'état courant du malade par rapport à la maladie (guéri, évolution +, évolution -).*

Nous considèrerons que nous avons à notre disposition les types :

```
typedef struct Date  {
       int jour;
       char mois[10];
       int annee;
       }     date;
```

```
typedef struct Personne  {
       char nom[20];
       char prenom[20];
       struct Date ne_le;
       }     personne;
```

1. *Définir les types suivants :*
   - `maladie`,
   - `dossier_médical`,
   - `client`,
   - `liste_clients`,
   - `medecin`,
   - `secretariat`.
2. *Écrire les fonctions suivantes :*
   - `init_secretariat`,
   - `init_client`,
   - `init_dossier_medical`,
   - `client_identique`,
   - `ajout_medecin_secretariat`,
   - `ajout_maladie_dossier_medical`,
   - `modif_maladie_dossier_medical`,
   - `suppr_client_de_liste_clients`.
3. *Si on accepte qu'un même client puisse apparaître sur plusieurs listes de clients sans duplication des informations le concernant,*
   (a) *Que devient le type* `client` *?*
   (b) *Comment écrivez-vous les fonctions :*
      - `appartient_client_a_une_liste_clients`,
      - `nb_occurrences_client_a_un_secretariat` *?*
   (c) *Comment modifiez-vous les fonctions :*
      - `init_client`,
      - `client_identique`,
      - `suppr_client_de_liste_clients` *?*