CHAPITRE

6

Mieux articuler le langage C

6.1 Les types composés & personnalisés

Jusqu'à présent, nous avons déclaré des variables à l'aide des types simples du langage : entier, flottant, caractère. Dans l'élaboration d'un algorithme pour résoudre le polynome d'ordre n, $P_n(x) = \sum_i a_i x^i$, nous avons constaté la difficulté de sauvegarder tous les coefficients $\{a_i\}$ de manière efficace, sans déclarer de nouvelles variables pour chacun (voir le programme Horner.c). Les variables de **type composé** permettent de construire des structures de données plus aptes à nos besoins.

1. Les tableaux
   On appellera **tableau** une variable qui représente dans l'espace mémoire de l'ordinateur un bloc ou volume subdivisé en cellules identiques ; ces cellules sont identiques les unes aux autres, elles ont le type affecté au tableau lors de sa déclaration :
   Syntaxe : $<$ *type* $>$ tableau $[N]$;
   où N est une expression ou constante entière. Par exemple, la déclaration
   ```
   float x[10]
   ```
   crée un tableau de 10 éléments réels, contigus dans l'espace mémoire. L'accès à chaque élément se fait en évoquant sa position dans le tableau :

Élément 1 $\rightarrow x[0]$
Élément 2 $\rightarrow x[1]$
$\vdots$
Élément 10 $\rightarrow x[9]$

Comme pour les variables, l'allocation mémoire est elle aussi typée, et respecte la procédure d'initialisation des variables, soit : auto (pas initialisée), static ou extern (initialisée à zéro).

Exemples de tableaux : tableau.c

Dans le fichier tableau.c on trouve les déclarations suivantes :

$\vdots$

```
/* Note : tableau pas initialise explicitement mais global, donc toutes ses
entrées contiennent 0. */
int Tab3[8] ;

/* Debut fonction main */
int main(void) {

int i, j=10;
char Tab2[4]={'t','e','s','t'};/* liste de constantes*/
                               /* caracteres */
int Tab1[8] ={1,2,3,4,5,6,7,8};/* tableau initialise */
                           /* avec constantes entieres */
```

$\vdots$

Le résultat de ces déclarations est toujours la création d'une table de dimension égale à un admettant un seul coefficient comme argument : la longueur de la table est spécifiée par la valeur de la constante ou expression entière apparaissant entre crochets ('[ ]') au moment de la déclaration.

Schématiquement on a la représentation suivante de l'espace mémoire alloué à la table, soit pour le cas de *tab1* :

En d'autres termes, le nom 'tab1' sert d'étiquette référant au début de la table. L'indice [entre crochets] sert alors à indiquer un déplacement de tant d'unités relatif au début de la table.

La déclaration d'une table peut inclure plusieurs dimensions, auxquelles correspondront autant d'indices. Ainsi

```
int Tab1[4][4] = {1,2,3,4,5,6,7,8};
```

aurait pour conséquence de créer une table à deux dimensions et d'initialiser les 8 premières entrées, c'est-à-dire Tab1[0][0 à 3] puis Tab1[1][0 à 3]. Le reste de la table comprendra des zéros. En général : le dernier indice d'un tableau est toujours le premier courant lors de l'affectation de valeurs à l'espace mémoire occupé par le tableau.

Autres exemples : voir programmes racines.c (cf. TD3), Horner.c

2. Structures (mot-clef **struct**)

   Les structures sont une version sophistiquée des tableaux : elles acceptent des *champs*, lesquels ont un *identifiant* (nom) chacun.

   Syntaxe : < *type* > nom { <type1> nom1 ; < type2 > nom2 ; ... ; } [*identifiant*] ;

   Chaque champ fait l'objet d'une déclaration. Pour accéder aux différents champs d'une structure, il suffit d'évoquer le nom de la structure suivi du champ comme ceci :

   structure nom, champ nom1 → nom.nom1

   Remarque : le champ d'une structure peut être de n'importe quel type, régulier ou composé. Ainsi une structure peut inclure une deuxième structure dans sa déclaration.

   Exemples de structures : struct.c

   Une structure 'vecteur' de deux champs pourrait être déclarée comme ceci

   ```
   struct complexe { float reel; float im };
   struct vecteur { struct complexe composante; float
   norme } V;
   ```

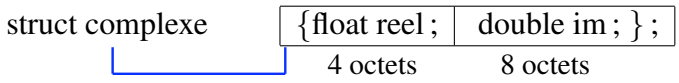
   Dans ce qui précède il y a en fait deux déclarations : une première définie la structure 'complexe' de deux champs (reel et im). La seconde crée la structure vecteur qui *inclue* déjà la nouvelle définition. Ainsi on peut créer le champ nommé 'composante' de la structure vecteur. Un deuxième champ (nommé norme) est déclaré réel ('norme').

   La référence aux champs d'un vecteur V respecte la logique de la syntaxe :

   V.norme → champ norme de V
   V.composante.reel → champ reel des composantes de V.

Restriction : la déclaration d'une structure ne peut inclure elle-même comme champ. Par contre, nous verrons plus tard comment y faire référence par l'utilisation d'un **pointeur**.

Schématiquement on a la représentation suivante de l'espace mémoire alloué à la structure complexe :



Le volume occupé en mémoire est, comme pour les tableaux, la somme des entrées multipliée par leur volume respectif, ici une entrée correspond à un champ de la structure. Un champs occupe l'espace mémoire correspondant à sa déclaration, indépendamment de celui des autres champs : ici le champ im occupe un volume deux fois plus grand que celui du champ réel. Le volume total est, dans cet exemple, de 4 + 8 = 12 octets.

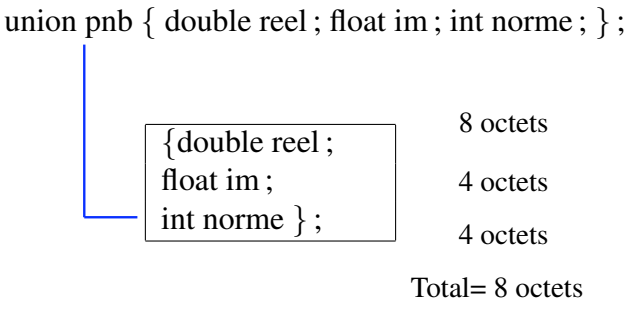
Autres exemples : voir le fichier struct.c.

3. Les unions (mot-clef **union**)
   Les unions sont comme les structures, mais cette fois un seul des champs est actif à la fois, c'est-à-dire que le dernier appel active le champ nommé, rend les autres inopérant.

   Syntaxe : union < *nom* > { < type 1> champ1 ; < type2 > champ2 ; ... } [*identifiant*] ;

   Remarque : l'espace mémoire réservé à une union est celle de son champ le plus grand ; par opposition à la structure, qui occupe un espace égal à la somme de celle de ses champs (qui sont tous actifs en même temps).

   Schématiquement on a la représentation suivante de l'espace mémoire alloué à l'union :



   Voir le programme Union.c pour un exemple spécifique.

Il existe aussi des **champs de bits** comme type composé : ils sont identiques aux structures mais permettent de définir la dimension d'un champ 'bit à bit'. Ils ne nous intéresseront pas davantage dans le présent cours.

Le C permet de définir deux types **personnalisés**, où l'utilisateur a une certaine liberté d'action. Il s'agit de

1. Définition de type avec **typedef**

   Tout simplement un utilitaire du langage permettant à l'utilisateur de définir ses propres termes. La portée d'une déclaration **typedef** respecte celle par défaut des variables. Il ne faut pas utiliser d'allocation mémoire en conjonction avec typedef.

   Syntaxe : typedef $<$ type $>$ *identifiant* ;

   Exemple :

   ```
   typedef int Entier;
   ```

   permet dans la suite du programme de déclarer, avec le mot Entier, des variables de type entier (int), c'est-à-dire par exemple

   ```
   Entier i,n;
   ```

2. l'énumération (mot-clef **enum**)

   L'énumération est en quelque sorte un aide-mémoire. Une énumération permet de faire une liste de noms, par exemple, et d'associer à chacun une valeur entière.

   Syntaxe : enum nom { nom1 [= entier1] , nom2 [= entier2], ... } ;

   Note : si aucune valeur n'est donné, alors l'initialisation se fait de gauche à droite en commencant par 0, puis 1, 2 .. jusqu'au dernier champ (nom).

   Exemple : voir le programme Enumeration.c.

6.2 Formats d'entrées/sorties en C

En général les données manipulées par un utilisateur ne sont pas distribuées arbitrairement, ni sur l'espace disque de l'ordinateur, ni au cour de l'exercice de visualisation de ces données : il est par exemple utile de savoir comment mettre en page les données pour les rendre lisibles à l'écran - effectivement c'est ce que fait le logiciel qui gère ce texte, sous le doigté balourd de votre humble serviteur...

Le but de ce chapitre est de coucher sur une base plus solide les notions de mise en page du C rencontrées au hasard de nos programmes précédents. Nous connaissons les trois fichiers standards d'entrée/sortie (*stdin/stdout*), ainsi qu'un fichier spécial de messages d'erreur (*stderr*). Nous les revisitons à tour de rôle.

6.2.1 Saisir des données formattées sur entrée standard

En mode ligne de commandes, l'entrée standard est tout simplement le clavier ou la souris utilisé/e. Lorsque l'on exécute un programme C, tel une commande Unix, et que celle-ci requiert de saisir un paramètre, le programme se met à balayer la ligne de commandes, jusqu'à rencontrer un caractère spécial, habituellement représenté par une flèche ($\boxed{\hookleftarrow}$), c'est-à-dre le 'retour du chariot' qui enregistre la commande et permet de continuer l'exécution du programme.

Une des fonctions du C qui gère la lecture de la ligne de commandes s'appelle `scanf` - nous l'avons déjà rencontrée à plusieurs reprises : elle admet la syntaxe suivante

int scanf(" expressions de formats", adresse1, adresse2 ..) ;

où à chaque expression de format doit correspondre une adresse : la fonction fait un balayage (`scan`) de la ligne de commandes, et affecte à chaque paramètre rencontré une adresse mémoire, qui, bien sûr, doit être celle d'une des variables de notre programme. Scanf est une fonction de type entier : elle retourne comme valeur le nombre d'affectations effectuées, c'est-à-dire le nombre de champs lus dont la valeur est passée à une variable : on peut donc faire l'affectation de `scanf` à une variable entière.

Les expressions de formats sont spécifiques au type de l'adresse vers laquelle est affecté le paramètre saisi en entrée. Le format se distingue par le caractère '%' qui le précède.

Format	Type de la variable saisie
%i	entier (int)
%li	entier long (long int)
%hi	entier court (short int)
%f	flottant (float)
%c	caractère (char)
%s	chaîne de caractères
%d	décimal

TAB. 6.1 – principaux formats admis par scanf

L'adresse d'une variable s'obtient en précédant le nom de la variable du symbole & . Un appel de `scanf` permettant de lire une variable entière k serait

```
scanf( ``%i '' , &k );
```

par ailleurs le nombre de variables affectées par `scanf` est arbitraire : voici un appel permettant de lire un entier, une variable réelle double précision, un caractère, un mot - avec un seul appel de la fonction

```
scanf( ``%i %lf %c %s '' , &k, &x, &car, &texte );
```

Pour mémoire : on dira que le symbole & *opère* sur la variable, ou qu'il **pointe** vers cette variable. Nous reviendrons là-dessus plus tard.

Dans le dernier exemple il est à noter que les espaces vides qu'on retrouve entre les expressions de formats **ne** sont **pas** reconnus par `scanf`. Par contre, les différents paramètres saisis seront eux distingués par des espaces vides.

Si par ailleurs nous désirions utiliser un symbole particulier pour délimiter les paramètres saisis, nous aurions toute la latitude de le faire avec `scanf`. Par exemple, il est courant de noter l'heure sous le format suivant

hh :mm :ss

pour heure, minutes et secondes. Une saisie du temps dans ce format se ferait comme ceci :

```
scanf( `` %i : %i : %i '', &h, &m, &s );
```

si chacune des variables h, m, s est de type entier. Tous les symboles sont permis comme délimiteur de paramètre, et, en particulier, un délimiteur pourrait inclure du texte ou une constante numérique : l'utilisateur sera contraint d'utiliser ce même délimiteur, quel qu'il soit.

Scanf admet quelques caractères spéciaux, qui permettent de contrôler encore davantage les formats de saisie. Notons :

– les crochets [ ]

Ils permettent de définir les caractères lus ou ignorés en entrée. Par exemple

```
scanf( "%[a-z,A-Z]'' );
```

amènerait `scanf` à lire toutes les lettres (majuscules et minuscules) tapées sur la ligne de commande, jusqu'à rencontrer un symbole d'une autre nature, tel un espace vide, ou le caractère de changement de ligne '\n'. Pour s'assurer qu'une variable caractère (char s ;) ne prenne qu'une lettre en entrée, plustôt qu'un symbole spécial tel un point de ponctuation (; :. ? etc), alors on pourrait appeler

```
scanf( "%[a-z,A-Z]'', &s );
```

Il n'y a pas d'ambiguïté puisque les lettres sont des constantes caractères du C, donc de même type que s.

Si nous désirions lire au plus n lettres à l'écran, nous pourrions spécifier ceci en écrivant

```
scanf( "%n[a-z,A-Z]'', &s );
```

Notez que 's' se verra affecter de la première lettre rencontrée, donc la première affectation permise, alors que scanf continuera de lire la ligne de commande **jusqu'à** ce que le symbole rencontré **ne** soit **plus** une lettre.

Clarifions : avec le format ci-haut, et n = 5, si l'utilisateur entre au clavier

```
aBcD eF ...
```

alors le caractère s = a, première affectation permise ; `scanf` poursuit son balayage, lit BcD et s'arrête là car l'espace ' ' n'est pas une lettre. Si nous avions une déclaration char mot[2], alors

```
scanf( "%n[a-z,A-Z] %s'', &s, mot );
```

donnerait pour la même chaîne de caractères entrée les affectations

s = a ; mot = eF .

Comme pour l'énumération de caractères acceptés en lecture, il est possible d'identifier un ou des caractères de terminaison de lecture, par une négation : elle se fait en insérant le symbole '^' devant la liste de caractères :

```
scanf( "%n[^a-z,A-Z] %s'', &s, mot );
```

Cet appel de `scanf` aurait pour conséquence que n'importe quelle lettre de l'alphabet indiquerait une fin de lecture. Cette instruction peut-etre utile si l'on sait que le texte d'interêt est précédé de caractères particuliers indésirables.

– Le caractère étoile $*$
Il permet de lire un caractère, sans faire l'affectation à une adresse. Par exemple

```
scanf( "%i %*c %i", &m, &s, &h );
```

Cette déclaration ferait en sorte que la variable caractère s ne se verrait pas affectée par la lecture en entrée du premier caractère, par contre les entiers h et m seraient affectés d'une valeur donnée par l'utlisateur.

– Opérateur %n
Cet opérateur permet de compter le nombre de caractère lus par `scanf` **avant** que %n n'apparaisse dans l'expression de formats. Par exemple, pour obtenir le nombre de lettres passées par l'utilisateur, on pourrait faire

```
scanf( "%[a-z,A-Z]%n %s", &s, &scancount, texte );
```

suivant les déclarations de variables tel

```
int scancount=0; char s = ' ', texte[100];
```

La variable entière 'scancount' se verrait affectée de la valeur retournée par %n.

6.2.2 Écrire des données formattées sur sortie standard

La fonction C `printf` joue pour les sorties sur fichier standard le même rôle que la fonction `scanf` pour les entrées.

Syntaxe : int printf(" expression de formats ", variable1, variable2 ..) ;

Les principaux formats de `printf` sont ceux acceptés par `scanf`, voir Table 1. Nous avons également vu une liste exhaustive en TD pour ces deux fonctions, prière de s'y référer.

Comme pour `scanf`, les variables évoquées doivent correspondre une à une avec les formats mentionnés, en nombre et type. On peut effectuer un changement de type d'une variable par `cast`, pour les besoins de la mise en page, comme ceci (admettons une variable déclarée réelle float x ;)

```
printf( `` %i '', (int)x );
```

nous donnerait, en sortie, la partie entière de x, sans pour autant en changer le type (x demeure une variable réelle dans le programme).

Il est à prendre bonne note que les variables sont évoquées par nom, sans opérateur, dans l'appel de `printf`. C'est bien la valeur préalablement affectée à ces variables que nous désirons envoyer à l'écran.

La correspondance avec `scanf` ne s'arrête pas aux formats et syntaxe générale :

– les symboles de séparation de `scanf` deviennent maintenant du texte envoyé à l'écran avec `printf` :

```
printf( `` Ceci est du texte et %i un entier et
      puis %f un flottant .. ``, j, x );
```

– Tout caractère est permis à l'intérieur des guillemets ; pour générer le caractère '%' sans le confondre avec un symbole de format, il suffit de le **doubler** dans l'expression, ainsi :

```
printf( `` Ceci est du texte %%i ...'' );
```

donnerait à l'écran

Ceci est du texte %i ...

– Les symboles spéciaux peuvent aider à la mise en page : \n cause un changement de ligne, \t un espace horizontal (tabulation), \v un espace vertical, sont les deux ou trois plus importants symboles. Se rapporter au cours #3 pour d'autres caractères spéciaux.

6.2.3 Lire et écrire des données non-formattées sur e/s standards

Il est possible de saisir ou d'envoyer à l'écran des données sans pour autant en spécifier le ou les formats :

– la saisie non-formattée se fait avec *getc* (caractère par caractère) et *gets* (une chaîne de caractères complète).
– l'envoie à l'écran se fait par les fonctions *putc* (un caractère à la fois) et *puts* (toute la chaîne de caractères).

La syntaxe de chaque fonction est très similaire, elles retournent toutes des valeurs entières : la syntaxe dans chaque cas est

```
int getc(); int gets( nom_chaîne );
int putc( nom ); int puts ( nom_chaîne );
```

L'appel de chaque fonction est par ailleurs différent : le résultat de *getc* doit être affecté à un entier, en principe, mais si cet entier correspond à une lettre ou symbole de la table ASCII, alors l'affectation peut se faire directement à une variable caractère, ce qui est souhaité ici.

Exemples d'utilisation :

on supposera les déclarations char x, tab[50] ;

1. `x = getc();`
2. `gets( tab );`
3. `putc( x ), putc(tab[1]);`
4. `puts( tab );`

Dans chaque cas les fichiers visés sont *stdin* et *stdout* pour les appels *get* .. et *put* .. respectivement.

Notez que la lecture de caractères non-formattés permet d'inclure tous les caractères de la table ascii, et pas seulement ceux qui apparaissent sur le clavier de l'utilisateur. On aura entre autres accès aux caractères spéciaux \n, \t, etc, ainsi qu'aux caractères identifiés par des **constantes symboliques** du C. À titre d'exemple notons

– NULL : adresse d'une variable mise à zéro ;
– EOF : "end-of-file", caractère indiquant la fin d'un fichier.

Ces deux exemples nous serviront plus loin lorsque nous écrirons dans des fichiers autres que les standards prédéfinis.

6.3 Les pointeurs : définition, propriétés

Problème : comment optimiser l'utilisation de l'espace mémoire lors de l'exécution d'un programme ?

⟶ avec des définitions de tableaux, tel

x[10000]

l'utilisateur doit deviner à l'avance la grandeur maximale de son tableau pour éviter des pertes de données. Une fois déclaré, le tableau existe pour la durée du bloc d'instructions ou du programme.

→ mauvaise gestion si le tableau x ne contient pas toujours ici 10000 éléments

→ cas d'exception possible difficile à prévoir
→ corriger, reprendre le calcul, éditer le fichier : pas efficace ..

Tous ces problèmes disparaissent si l'allocation d'espace mémoire peut se faire de façon dynamique au cours de l'exécution du programme.

Par ailleurs, le programme gagnerait en efficacité si nous pouvions affecter directement des valeurs à des variables, sans passer par l'intermédiaire d'une variable, voire une fonction : c'est par exemple ce que nous cherchons à faire par l'appel à des fonctions avec arguments.

6.3.1 Le pointeur

Le **pointeur** est cet outil qui garde en mémoire l'adresse d'une variable. Cette adresse, c'est-à-dire la valeur affectée au pointeur, est absolue dans toutes les fonctions du programme, car elle représente un volume physique de la mémoire de l'ordinateur.

Syntaxe de déclaration d'un pointeur

< type > * < *identifiant*>

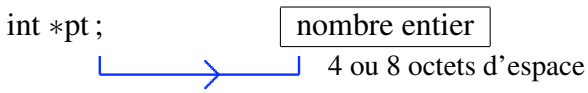
ici le type est un type régulier, composé ou autre du langage.
Exemple : les déclarations

```
int *pt; struct complexe *z;
```

créent un pointeur vers un entier, pt, et une structure complexe,z. Comme l'entier n'a pas été spécifié, le pointeur n'a pas été initialisé, il ne pointe pour le moment vers rien.

Il est à noter que la déclaration d'une structure peut inclure un pointeur vers elle-même : un pointeur n'est pas la même chose qu'une variable.

Déclaration d'un pointeur : schéma



6.3.2 Initialisation d'un pointeur.

Pointeur → affectation d'une adresse mémoire. Elle s'exprime par

a) le mot NULL (adresse vide)

b) une chaîne de caractères (constante du langage)

c) l'adresse d'une variable déclarée

d) nom de tableau ou fonction : ils sont, en réalité, *eux-mêmes des pointeurs*. Le nom d'un tableau est un pointeur vers le début (première case) de ce tableau.

Remarque : l'initialisation doit être conséquente avec le type de la déclaration du pointeur.

Remarque 2 : le pointeur lui-même occupe un espace mémoire, il est distinct de la variable ou objet vers lequel il pointe.

6.3.3 Opérateur &

Un opérateur spécial représenté par le *et* logique, &, est utilisé pour obtenir l'adresse mémoire d'une variable. La syntaxe est simplement le nom de la variable précédé de '&'.

Syntaxe : & < *identifiant* >

6.3.4 L'allocation d'espace mémoire

L'allocation d'espace mémoire peut se faire de manière dynamique avec la fonction `malloc`.

- déclaration d'un pointeur $\neq$ existence de l'objet pointé
- `malloc` : réserve une zone mémoire et renvoie un pointeur au début de cette zone.

  Syntaxe : malloc (sizeof(*objet*))

  ici `sizeof()` tient lieu d'une expression entière du C exprimant le volume en **octets** d'espace mémoire requis.

  Par exemple l'instruction :

  ```
  char *pt = malloc( sizeof(``texte .. '') );
  ```

  crée un pointeur *pt* qui a la dimension d'une chaîne de caractères *texte* ; le pointeur est initialisé par l'adresse du texte, mais **ne** contient **pas** cette chaîne.

  Rappel : les chaînes de caractères sont des constantes du langage, définies lors de l'exécution du programme. Pour une meilleure visibilité, il convient de les déclarer en début de programme ou bloc d'instructions, au même titre que les déclarations de variables.

6.3.5 Libérer l'espace mémoire

Pour **libérer** l'espace mémoire réservé avec *malloc*, il faut évoquer la fonction `free`. Donc :

`malloc` ⟶ nouveau bloc dans la mémoire RAM de l'ordinateur
`free` ⟶ libère cet espace.
Syntaxe : free(<*identifiant*>) ;

6.3.6 La valeur pointée

La valeur pointée, i.e. qui réside à l'adresse indiquée comme valeur du pointeur, s'obtient par le caractère étoile :

Syntaxe : * < *identifiant* > ;

Par exemple : on déclare un pointeur vers un entier, i, ainsi qu'une variable entière, k, par

```
int *i, k = 2;
```
initialisation de i : `i = &k;`
sauvegarde d'un entier : `*i = 5;`
résultat : `k = 4 * ( *i );`
est la même chose que : `k = 4 * k;`

6.3.7 Notation arithmétique des pointeurs

Supposons la déclaration d'un tableau

```
int tab[10];
```

Pour accéder à la $i^{ième}$ case de ce tableau nous connaissons la syntaxe

```
tab[i+1];
```

Or, tab réfère au début du tableau : donc en principe pour atteindre la $i^{ième}$ case il suffit de se déplacer *relativement à la case départ* de i+1 espace d'entiers : c'est ce qui est exprimé dans la notation arithmétique suivante

```
*(tab + i+1 );
```

L'expression entre accolades rondes (..) exprime l'adresse, l'opérateur $*$ fera lui la lecture de la valeur sauvergardée à cette adresse.

6.3.8 Conversion des pointeurs

– un pointeur vers un objet de type *void* peut être converti à souhait, sans cast.
– pour lire sa valeur (ie son adresse) : faire l'affectation seulement
Par exemple :

```
char *ch ; float *ft, f ;
void *pt ;
```

L'opération d'affectation : `pt = ch;`
donne à pt l'adresse d'un caractère. Par contre : `ft = pt;`
ici pt donne sa valeur à un pointeur vers un flottant, mais
`ft = (float *)ch;`
donnera exactement le même résultat, du fait de l'initialisation de pt. Si l'on essaie de lire (par variable de type flottant f) à l'adresse pt
`f = *(float *)pt;`
avec un cast (conversion flottant) : ça va, mais sans conversion : NON. Le cast convertit pt en pointeur flottant, puis prends sa valeur par opérateur $*$: cast explicite, le résultat est préservé dans f.

6.3.9 Pointeurs et fonctions

Nous allons ici construire un algorithme pour additionner deux entiers, afin d'illustrer l'utilisation des pointeurs dans les appels de fonctions.

Fonction PLUS

```
/* d\'eclaration de fonc plus */
int plus ( int i, int *j ) {
*j = *j + 2 ; i = *j + i ;
return i ; }
main () { /* fonc principale (main) */
int x = 10, y = 20, resultat ;
resultat = plus( x, &y ) ;
exit(0) ; }
```

À l'exécution de ce programme, les variables x,y sont initialisées à 10 et 20, respectivement, suivant leur déclaration dans le *main*.

L'astuce ici est que les valeurs numériques de x et y sont utilisées pour initialiser les variables *locales* i, j de PLUS mais par ailleurs : j est un *pointeur*, il pointe vers l'adresse mémoire qui coincide avec la variable y de la fonction *main* : c'est bien $\&y$ qui est passé en argument de PLUS.

Suivant l'appel de PLUS, nous avons comme affectation

resultat = 22 ; et y = 12, mais x = 10, toujours.

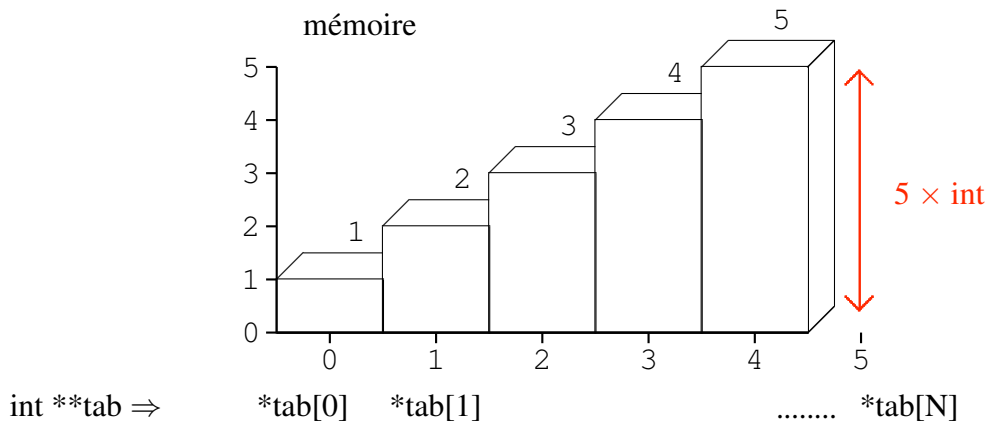
Les pointeurs peuvent donc être utilisés pour définir des fonctions à valeurs multiples, par exemples pouvant modifier plus d'une variables à la fois ou encore permettant de vérifier des résultats internes (à la fonction).

6.3.10 Applications de pointeurs

Application 1 : construction d'une matrice triangulaire

Nous désirons construire une matrice triangulaire - et non pas rectangulaire $m \times n$, afin de sauver jusqu'à 50% d'espace mémoire (voir schéma ci-dessous).

Fig. 1 : Table triangulaire : 'pointeur de pointeurs'



Pour définir une table de dimensions $m \times n$, donc rectangulaire, nous aurions comme déclaration

int tab[m][n] ;

avec le désavantage possible que la table ainsi définie requiert $m \times n$ fois int d'espace mémoire **fixe**. Pendant l'exécution du programme, cet espace n'est plus disponible pour les calculs divers, qui doivent donc attendre que la mémoire devienne disponible (on imaginera que m et n sont grands, très fréquemment le cas dans les calculs numériques intensifs).

Il est plus intelligent de **n**'occuper **que** l'espace mémoire essentiel à chaque étape de l'exécution de notre programme. On utilisera *malloc* et les pointeurs à cette fin.

La déclaration d'une table multi-dimensionnelle correspond à définir un pointeur vers une liste de pointeurs de dimensions réduite d'une unité. Ainsi une table à 2 dimensions se déclare par l'instruction

int **tab ;

soit un 'pointeur vers un pointeur', une manière de lire la notation '**'. Il reste à réserver l'espace mémoire de chaque pointeur. On se rappelle que *malloc*

retourne comme résultat une adresse, celle-ci peut être utilisé pour initialiser un pointeur.

Notre algorithme pourrait se lire comme ceci :

1. déclarer un pointeur de pointeurs vers des entiers : int **tab
2. déclarer un entier (= i) qui servira de compteur
3. définir un entier N qui est le nombre désiré de colonnes (c'est-à-dire de pointeurs int *)
4. Avec malloc : réserver N pointeurs vers des entiers, initialiser tab.
5. Ensuite : pour chaque pointeur définir un bloc mémoire avec malloc. L'entier i sera utilisé pour faire un balayage des colonnes.
6. Insérer des valeur numériques dans la table, ou autres.
7. Effacer la table avec *free()* : il s'agit de procéder inversement aux opérations faites avec malloc.

De la Fig. 1 on déduit qu'à chaque colonne il faut un volume d'entiers correspondant à son cardinal, soit i entiers pour la i^{ieme} colomne.

Matrice triangulaire : pseudo-code

Pour transcrire notre algorithme en pseudo-code, on pourrait procéder comme ceci :

Bibliothèques système requises : stdlib.h, stdio.h
Entier N = nombre de pointeurs d'entiers : #define N 10

int main() {
Déclaration : table à deux dimensions int **tab ;
entier int i ;

Réserver l'espace mémoire pour les pointeurs d'entiers : avec cast
tab = (int **)malloc(N*int *) ;

Créer chaque pointeur l'espace mémoire, schéma simple soit
Pour (i = 0, tant que i $<$ N) {
tab(i) = (int *)malloc((i+1)*int)) ; i = i + 1 ; }
Opérations sur la table tab[i][j] maintenant possibles : exemple d'initialisation
tab[1][2] = 1 ;
Libérer l'espace mémoire : procéder inversement à l'allocation c'est-à-dire
Pour (i = 0, tant que i $<$ N) {
free (tab[i]) ; i = i + 1 ; }

Finalement, libérer le pointeur de pointeurs :
free(tab) ;

terminer l'exécution du programme
exit(0) ; }
Programme : voir Table.c

Application 2 : balayage d'une chaîne de caractères

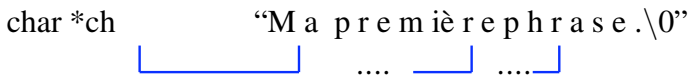
On désire compter le nombre de caractères non-nuls d'une chaîne de caractères. On supposera pré-définie une chaîne de caractères, soit par exemple

char *ch = "Ma première phrase." ; int i,j=0 ;

où nous avons défini une variable pointeur 'ch' et des entiers i, j qui serviront de compteurs. Nous comptons + 1 chaque fois que la chaîne comprend un symbole (n'importe lequel) différent de l'espace vide ou du caractère de terminaison '\0'.

1. ch est initialisé déjà : prendre sa valeur avec l'opérateur * ;
2. additionner + 1 à ch tant que la valeur lue n'est pas égale à '\0' ;
3. additionner + 1 à i si *ch n'est pas vide, c'est-à-dire un espace vide.

Un schéma illustrant le principe du balayage :



Le pseudo-code ressemblerait à ceci :

Inclure les bibliothèques système requises : #include stdlib.h, stdio.h

int main() {
Déclarer un pointeur vers une chaîne de caractères (initialiser), ainsi qu'un 2ième :
char *tp, *ch = "Ma Premiere phrase." ;
Déclarer un compteur entier (initialisé à zéro) : int i = 0 ;

Pour mémoire : initialiser tp = ch ;

Balayage : tant que(*ch n'égale pas '\0') {
si(*ch n'égale pas ' ') i = i + 1 ; (incrémenter le compteur)
incrémenter le pointeur : ch++ ;

}

Renvoyer des résultats à l'écran :
- imprimer la chaîne
- résultat i
- longueur totale de la chaîne : ch - tp ;
}

Application 3 : pointeur vers une fonction

Il peut être avantageux d'utiliser un pointeur vers une fonction pour permettre une plus grande flexibilité au moment de l'exécution d'un programme. Un pointeur vers une fonction doit être du même type que la valeur retournée par la fonction, et doit, en même temps, reconnaître tous les arguments de la fonction.

Par exemple, un pointeur fp vers une fonction de type 'float' et de deux arguments, l'un réel (float), l'autre entier (int), serait déclaré ainsi :

```
float (*fp)(float, int) ;
```

et nous pourrions l'initialiser avec la fonction *puiss(x, n)* déjà rencontrée :

```
fp = puiss ;
```

comme pour tout pointeur. Le nom d'une fonction, à la manière d'un tableau, est un pointeur fixe essentiel pour la transmission des arguments. Un programme complet qui fait usage d'un pointeur vers une fonction est donné ci-bas.

L'utilité d'un programme de la sorte serait de pouvoir donner le nom d'une fonction lié au programme, à la ligne de commandes, au moment de lancer le logiciel. Par exemple, on imagine remplacer 'puiss' dans l'exemple donné ici par des fonctions telles 'sin', 'cos', 'exp', etc, de la biliothèque mathématiques (entête du programme comprendra MATH.H).

Au prochain chapître, nous verrons comment passer des arguments à un programme à la ligne de commandes, comme il est fait pour les commandes Unix avec options.

```
/* Exemple d'utilisation d'un pointeur vers une fonction */

#include <stdio.h>

/* Patron d'une fonction avec pointeur vers une autre
    fonction en argument */
float resultat( float, int, float (*)(float, int) ) ;
```

```
float puiss ( float x, int n )
{
  if( n == 0 ) return ( 1. ) ;
  return ( x * puiss( x, n-1 ) ) ;
}

/* Debut fonction main */

int main() {

  /* declaration de variables : fp est pointeur vers une
      fonction reelle de 2 args */
  int n; float x, y; float (*fp)(float, int) ;

  /* Saisie de deux parametres, x et n */
  printf( " Donner les deux parametres x et n < 10 \n " ) ;
  scanf( "%f %d", &x, &n ) ;

  /* Exemple d'initialisation du pointeur ; la fonction puiss
     est deja connue (voir plus haut) */
  fp = puiss ;

  /* Appel d'une fonction avec la fonction puiss ou fp en
     argument : */
  y  = resultat( x,n, puiss ) ;

  printf( "%s %f\n", "fp Resultat y = ", y ) ;
  return 0;
}

/* Cette fonction peut accepter toute reference a une fonction
   de type float */

float resultat( float x, int n, float (*fonction)(float, int) )
{
float y = fonction( x, n ) ;  return y ;
}
```

6.4 Créer une bibliothèque

6.4.1 Second regard sur la compilation

Nous avons vu à plusieurs reprises que les programmes doivent inclure une référence aux fonctions du C déjà codé et que nous désirons utiliser. Il s'agissait dans chaque cas d'inclure une référence à la bibliothèque contenant les instructions de la fonction en question (stdlib, math, stdio, et autres).

En réalité les directives au pré-processeur que nous avons utilisées ne font références qu'aux patrons des fonctions visées. Par exemple lorsque nous avons écrit

```
#include <math.h>
```

il s'agissait pour le pré-processeur d'insérer un fichier système, math.h, lequel ne fait que donner tous les patrons des fonctions mathématiques usuelles.

Les fonctions elles-mêmes sont déjà compilées et sauvegardées dans un fichier binaire, la bibliothèque. Ce qui permet leur exécution par notre programme, c'est l'édition de lien entre notre programme et cette bibliothèque, au moment de générer l'exécutable.

Cette opération du compilateur est souvent invisible, le lien est créé par défaut. Ceci est généralement le cas pour les bibliothèques standards du C, mais pas toujours (cela pourra par exemple ne pas être vrai avec un compilateur ancien).

Dans le cas spécifique où l'utilisateur désire créer sa propre bibliothèque, il devra préciser le nom et l'emplacement de cette bibliothèque. Voyons tout d'abord comment créer une bibliothèque de manière générale.

6.4.2 Compiler les objets de fonctions

Imaginons que nous ayons deux fonctions, disons EXPO et FX, toutes deux décrites dans des fichiers texte expo.c et fx.c. Ces fichiers ne contiennent **que** les instruction relatives aux fonctions ; notre programme est composé dans un autre fichier, disons Int.c.

Il faut dans un premier temps compiler expo et fx, dans générer d'exécutable. En effet, une fonction seule ne peut être exécutée, et le compilateur ne peut pas, conséquemment, générer un exécutable. C'est l'option '-c' de compilation qui permet de gérer cette situation :

```
ligne de commandes>  g++ -c -DN=10 expo.c fx.c
```

va générer les fichiers binaires contenants les objets compilés, expo.o et fx.o. Par exemple le fichier expo.c contient les instructions

```
float pn[N+1] ;

float expo( float x ) {  float y; int j ;

  /* Definir les coefs de la serie de Taylor pour
      l'exponentielle : */

  pn[0] = 1.0 ;
  for(j = 1; j <=N ; j++ ) { pn[j] = pn[j-1] / j ;  }

/* Calculer exp(x) : */
  y = pn[N] ; for(j = N-1 ; j >= 0 ; j = j - 1 )
                      { y = pn[j] + y*x ; }  return y ; }
```

(Le fichier fx.c ferait un autre calcul, mais est de forme similaire ; il est omis pour simplifier.) Comme la constante symbolique N apparaît dans la définition d'un tableau global, il est nécessaire dévoquer le compilateur avec la définition appropriée (-DN=10).

6.4.3 Archiver ses fonctions

Lorsque nous avons tous les objets nécessaires, il faut les archiver dans une seule bibliothèque. Pour cela nous évoquerons l'utilitaire AR d'Unix. Il admet un grand nombre d'options, toutefois nous ne désirons que créer une bibliothèque nouvelle et y inclure deux fichiers objets. La commande

```
ligne de commandes>  ar -r libint.a expo.o fx.o
```

va créer un fichier libint.a qui comprendra les deux objets (fichiers '.o').

Avant de pouvoir faire le lien avec un exécutable, il faudra créer un fichier texte contenant tous les patrons de nos fonctions (ici, expo et fx). Ce fichier, 'mes_patrons_de_fonctions.h', sera à include par directive au pré-processeur au haut du fichier concerné (voir l'exemple plus loin).

6.4.4 Édition de liens

Ne reste plus qu'à relier notre bibliothèque libint.a à un exécutable en compilant un programme, disons Int.c. Pour cela il faut spécifier la localisation et le nom de la bibliothque :

```
ligne de commandes>  g++ -o int.x -L. -lint  Int.c
```

Cette commande créera l'exécutable int.x ; l'option -L indique au compilateur d'inclure le répertoire local (représenté par le point) dans la recherche de fichiers ; et l'option -l va causer un lien entre l'exécutable et la bibliothèque 'int' (le compilateur ajoutera lib et l'extension .a au nom indiqué).

Un exemple de programme pouvant être lié à notre bibliothèque libint.a est donné ici.

```
/* Inclure les fichiers publics sur le systeme */

#include <stdlib.h>
#include <stdio.h>

#ifndef N
#define N 10
#endif

/* Fichier local cree par l'utilisateur :
   noter l'usage des guillemets.           */

#include "mes_patrons_de_fonctions.h"

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

  float a, b, x ; int i;

    a = atof(argv[1]) ; b = atof(argv[2]) ; }

  /* Faire l'integrale : definir h, et m */
  { int m = N ; float h = (b-a)/m, sum = 0. ;
    x = a ;
    for( i = 0 ; i < m ; i++ ) {
      sum +=  expo(x) * h ;
      x = x + h ;   }

/* Ecrire le resultat a la ligne de commande : */
 printf( " Integrale de exp(x) = %f\n", sum ) ;
  }
```

```
  return 1;
}
```