Cours “Algorithmique et Structure des Données”

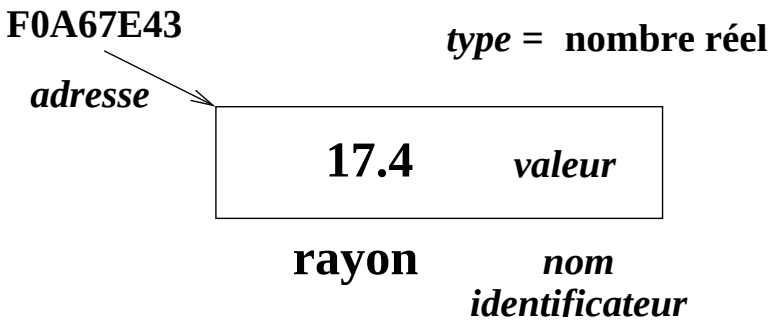
Licence L2 Informatique, semestre S3

Année 2019-2022

Variable : rappel

Les **variables** servent à représenter les données, les résultats, etc...

Une **variable** correspond à :



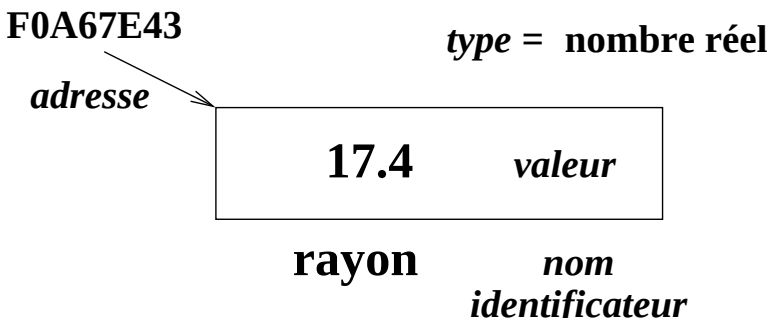
- *(pour l'homme)* un objet, une quantité
  - qui reçoit un **nom** ou **identificateur**,
  - qui est d'un certain **type** (selon la nature de l'information),
  - qui peut prendre des **valeurs** ;
- *(pour la machine)* un emplacement mémoire
  - caractérisé par une **adresse** (lieu où la variable est stockée)
  - et une taille (place utilisée) qui dépend du *type* de la variable.

Nom ou pointeur

La plupart du temps, on manipule les variables en utilisant leur **nom**.

Toutefois, comme on l'a constaté pour les passages de paramètres à des fonctions ou procédures, par exemple, il peut être utile de manipuler **directement** les **pointeurs**.

Si l'on reprend l'exemple, on pourra considérer :



- la variable réelle **rayon** dont la valeur est **17.4**
- un pointeur **ptr** sur un réel, qui pointera sur *rayon*, et dont la valeur sera **F0A67E43** (cette valeur est un **entier**, écrit ici en *hexadécimal*)

Déclaration et affectation pour un pointeur

Il n'est cependant pas nécessaire (voire même impossible...) de connaître explicitement l'adresse à laquelle sera allouée la variable rayon.

On peut néanmoins déclarer une variable de nom **ptr**, avec, comme type, un "pointeur_sur_un_réel", et lui affecter la valeur de l'adresse de rayon.

Dans ce cours, on utilisera la syntaxe du **langage C** pour les pointeurs.

```
float rayon = 17.4;
float *ptr;      // déclaration d'un pointeur_sur_un_réel

ptr = &rayon;    // affectation de l'adresse de rayon au pointeur
```

Précisions sur les adresses

DONC : un pointeur est une **adresse**.

Dans le codage informatique, une adresse est celle d'un octet.
Or la variable réelle rayon est codée sur plusieurs octets

$$\Rightarrow$$

la valeur du pointeur **ptr** sera l'adresse du *premier octet* de l'espace mémoire occupé par la variable rayon.

Cette valeur est un nombre entier ; toutefois **WARNING**
un pointeur n'est PAS de *type* `int`.

(voir suite de cette section)

L'opérateur d'indirection

Si l'on accède à une variable par un pointeur sur son adresse, on peut retrouver la **valeur** de la variable, en utilisant l'opérateur d'indirection *****

```
float rayon = 17.4;
float *ptr;      // déclaration d'un pointeur_sur_un_réel
ptr = &rayon;    // affectation de l'adresse de rayon au pointeur
printf(``valeur pointée par p = %f'',*ptr);   // valeur pointée
```

L'instruction `printf()` affichera **17.4** qui est la *valeur pointée* par **ptr**.

Réservation d'un espace mémoire

Lorsqu'un pointeur est déclaré, sa *valeur par défaut* est (quel que soit le type pointé), une constante symbolique notée **NULL**. On peut initialiser le pointeur en lui affectant l'adresse d'une variable (cf. ci-dessus).

On peut aussi affecter directement une valeur à ***ptr**, mais il faut d'abord réserver un espace mémoire de taille adéquate. L'adresse du premier octet de cet espace sera la valeur de **ptr**.

Cette réservation d'espace mémoire s'appelle l'**allocation dynamique**.

La fonction malloc

En langage C, l'allocation dynamique se fait avec des fonctions de la bibliothèque `stdlib.h` (à inclure, donc). La plus courante est :

`malloc(nombre_d'octets)`

Cette fonction retourne un **pointeur générique de type** `void *` pointant vers un objet de taille nombre_d'octets mais de type non identifié a priori.

Pour spécifier le type, il faut faire un `cast`, i.e. une conversion explicite.

Exemple :

```
int *p = NULL;
p = malloc(sizeof(int));
*p = 27;
```

ou, mieux :

```
p = (int*)malloc(sizeof(int));   // type spécifié par un cast
*p = 27;
```

Les fonctions : `calloc` et `realloc`

La fonction `calloc` est une variante de `malloc` et s'appelle avec deux arguments (à la différence de `malloc`) :

```
calloc(nombre_d'objets , taille_des_objets)
```

outre la réservation d'un espace mémoire, elle initialise chaque objet à zéro.

La fonction `realloc` permet de ré-allouer au même pointeur (voire même à un autre) un espace mémoire de taille différente :

```
realloc(nom_du_pointeur , nombre_d'octets)
```

WARNING : si la nouvelle taille requise est supérieure à la taille de l'espace déjà réservé, il se peut que **la valeur du pointeur change**.

Exemples et précisions

Supposons que l'on veuille réserver un espace mémoire pour 12 entiers :

```
ptr = (*int) malloc(12*sizeof(int));
```

Si l'on avait voulu faire la même chose, mais en s'assurant que l'espace réservé ait été nettoyé (le "c" vaut pour *clear*), i.e. que les entiers sont initialisés à zéro :

```
ptr = (*int) calloc(12, sizeof(int));
```

Et si l'on veut, ensuite, réduire à 8 entiers la taille de l'espace mémoire réservé :

```
ptr = (*int) realloc(ptr, 8*sizeof(int));
```

Pour toutes ces fonctions, en cas d'échec (ex. pas assez de place mémoire), la valeur **NULL** est renvoyée au pointeur demandant l'allocation.

Indispensable `free` pour libérer l'espace

Après utilisation, il est **indispensable** de libérer l'espace mémoire que l'on avait alloué, en faisant appel à la fonction `free` :

```
free(nom_du_pointeur)
```

WARNING : si une erreur se produit lors d'un appel à realloc alors on perd le lien vers l'espace mémoire initialement réservé ⇒ on ne pourra plus le libérer ! *Solution :* utiliser un pointeur auxiliaire lors de la ré-allocation.

Pour plus d'informations sur ces fonctions, on peut consulter, par exemple, le site web :
https ://en.wikipedia.org/wiki/C_dynamic_memory_allocation
ou les excellents cours proposés en ligne par OpenClassrooms

Incrémentation et décrémentation

Bien qu'un pointeur ne soit pas de type `int`, son type est **discret**, i.e. il existe un *successeur* et un *prédécesseur* à toute valeur d'un pointeur.

WARNING

Le successeur n'est pas *la valeur entière +1*, mais :
la valeur entière + le nombre d'octets sur lequel est codé le type pointé.

Exemples (avec des variantes possibles, selon la taille des codages) :
`ptr++`
vaut l'ancienne valeur de **ptr** + 4 s'il s'agit d'un pointeur_sur_un_entier.
`ptr--`
vaut l'ancienne valeur de **ptr** - 8 s'il s'agit d'un pointeur_sur_un_réel.

Additions et soustractions

On peut aussi ajouter et soustraire des pointeurs entre eux (utilisation délicate...), sous réserve qu'ils pointent sur des objets de même type, ou des pointeurs avec des constantes.

La même règle que ci-dessus est respectée : la *constante* **1** représente une unité de codage du type pointé (*exemple :* 4 pour un pointeur_sur_un_entier).

`ptr + i`
prend pour valeur entière celle de **ptr** à laquelle s'ajoute **i * sizeof(type1)** si **ptr** est un pointeur_sur_type1.

`p - q`
prend pour valeur entière celle de **(p - q) * sizeof(type2)** si **p** et **q** sont tous deux de type pointeur_sur_type2.

Comparaisons

Les opérateurs relationnels et logiques sont aussi applicables à des pointeurs, afin de faire des comparaisons et d'évaluer des tests, mais toujours avec les mêmes restrictions :

les pointeurs intervenant dans l'expression logique doivent toujours **pointer sur des objets de même type**.

Pointeurs et tableaux 1D

Un tableau 1D correspond en fait à un pointeur vers son premier élément :
tab équivaut à &tab[0], l'adresse du premier élément.
Ainsi, on peut parcourir les éléments d'un tableau en passant d'un pointeur au suivant : l'incrémentation **ptr++** revient à se déplacer vers l'espace mémoire situé "taille_des_éléments" plus loin.

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
void main()
{
  int *ptr;
  printf("\n lecture de tab, selon l'ordre croissant des indices:\n");
  for (ptr = &tab[0]; ptr <= &tab[N-1]; ptr++)
  printf(" %d \n",*ptr);
}
```

WARNING : le test NE peut PAS être : `ptr < &tab[N]` car tab[N] n'existe pas !

Quelques différences importantes

Toutefois, un tableau **n'est PAS** un pointeur

i.e. les manipulations autorisées sur les pointeurs ou sur les tableaux sont *différentes* :

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse,
  par exemple : `p = &i;` ou `p = tab;`
- un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut **pas écrire** `tab++;`).

Pointeurs et structures : opérateur →

Si **ptr** est un pointeur sur une structure, on peut accéder à un champ de la structure pointée par l'expression :

(*ptr).champ

Les **parenthèses sont indispensables** car l'opérateur d'indirection * a une priorité plus élevée que l'opérateur de champ de structure.

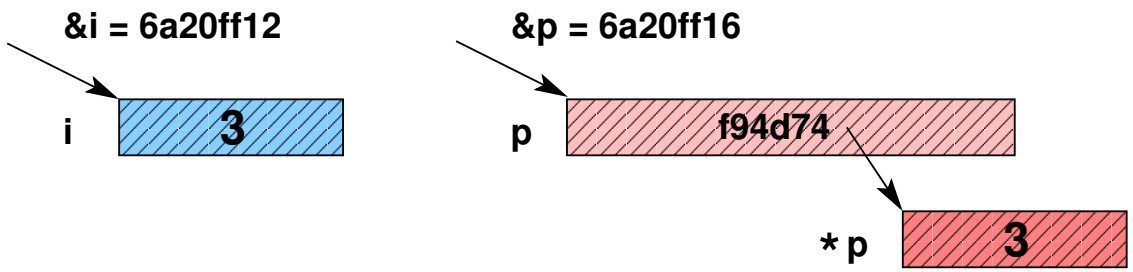
Cette notation peut être simplifiée grâce à l'opérateur **pointeur de champ** de structure, noté → et donc l'expression précédente est strictement équivalente à :

ptr→champ

L'instruction *p = i

CAS 1 : quels sont les effets de l'instruction ***p=i** ?

```
int i = 3;
int *p;
p = (int*)malloc(sizeof(int));
*p = i;
```



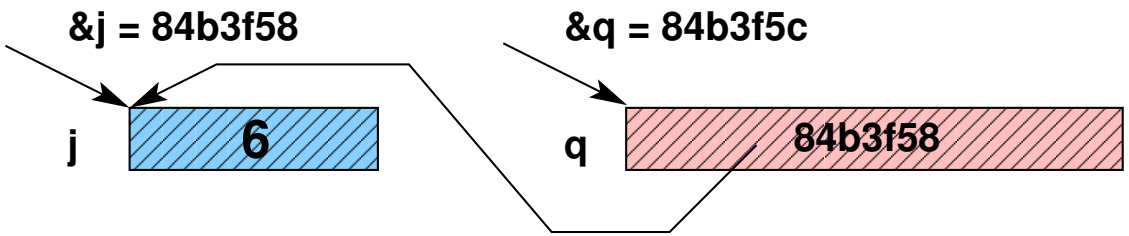
adresse de i = 0x6a20ff12 valeur de i = 3
avec p seulement déclaré, adresse de p = 0x6a20ff16 valeur de p = NULL
après allocation dynamique, valeur de p = 0xf94d74 [N.B. zone mémoire différente]
valeur de *p, AVANT affectation = 0 . . . voire même. . . *n'importe quoi !*
valeur de *p, APRES affectation = 3

*p a la **même valeur** que i MAIS la valeur de p n'est PAS l'adresse de i
⇐ La valeur 3 est stockée à DEUX endroits différents de la mémoire (**recopie ; clonage**) : l'endroit pointé par &i et aussi celui pointé par p.

L'instruction q = &j

CAS 2 : quels sont les effets de l'instruction **q=&i** ?

```
int j = 6;
int *q;
q = &j;
```



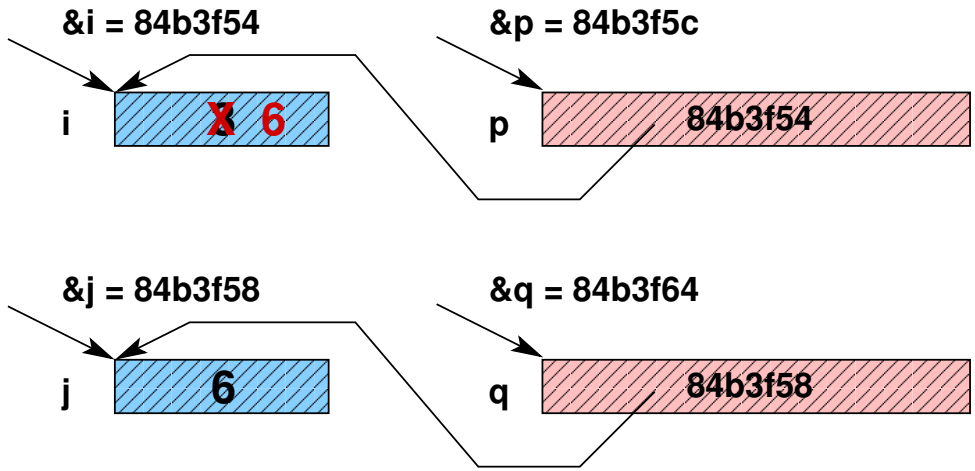
adresse de j = 0x84b3f58 valeur de j = 6
avec q seulement déclaré, adresse de q = 0x84b3f5c valeur de q = NULL
APRES affectation : valeur de q = 0x84b3f58 valeur de *q = 6

Cette fois-ci, on constate que : la valeur de q EST l'adresse de j.
Il en résulte (nécessairement !) que la valeur pointée *q EST la valeur de j.
En effet : ce 6 n'est stocké qu'à UN SEUL endroit dans la mémoire,
endroit vers lequel pointent à la fois &j et q.
j et *q sont **identiques**, pas des clones (contrairement au cas 1).

L'affectation *p = *q

CAS 3 : quels sont les effets de l'affectation ***p = *q** ?

```
int i = 3, j = 6;
int *p, *q;
p = &i;
q = &j;
*p = *q; // affectation
```



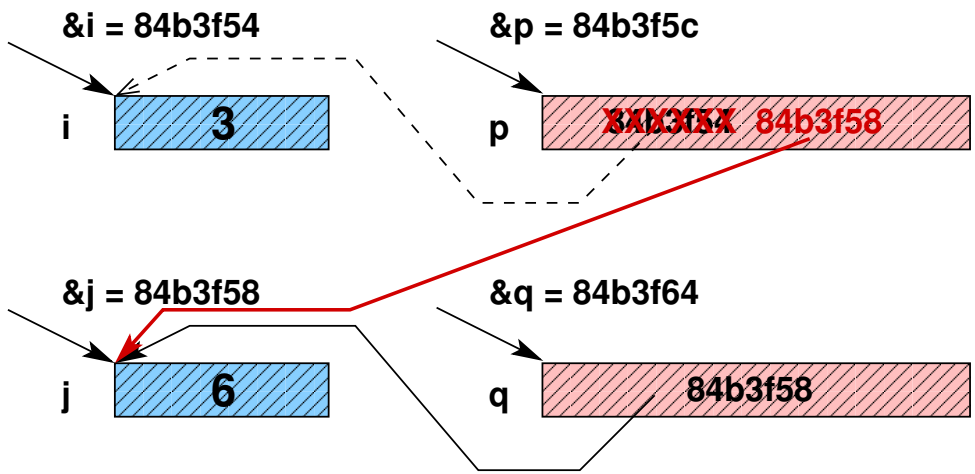
AVANT l'affectation de pointeurs, valeurs pointées : $*p = 3$ et $*q = 6$
APRES l'affectation de pointeurs, valeurs pointées : $*p = 6$ et $*q = 6$

Mais comme p pointait sur i, alors *p n'était autre que la valeur de i ...
$\Rightarrow$ **la valeur de i** a AUSSI été **modifiée** !

L'affectation p = q

CAS 4 : quels sont les effets de l'affectation **p = q** ?

```
int i = 3, j = 6;
int *p, *q;
p = &i;
q = &j;
p = q; // affectation
```



AVANT l'affectation de pointeurs, valeurs pointées : $*p = 3$ et $*q = 6$
APRES l'affectation de pointeurs, valeurs pointées : $*p = 6$ et $*q = 6$

Au niveau des valeurs pointées, mêmes effets que dans le cas 3 ; pourtant :

1. ici, la valeur de i n'a PAS été modifiée ;
2. en revanche la **valeur de p** a changé : elle est devenue l'adresse de j (comme q) et non plus celle de i.

Inconvénients des tableaux statiques

Les tableaux étudiés précédemment étaient **statiques** : ils devaient être *surdimensionnés* pour que la réservation de place mémoire se fasse correctement lors de la déclaration de la variable de type tableau.

Outre le gâchis de place mémoire, les tableaux statiques ont (au moins) deux inconvénients :

- on ne peut pas créer de tableaux dont la taille est une variable du programme,
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'auraient pas toutes le même nombre d'éléments.

Les tableaux dynamiques

L'usage de pointeurs et l'allocation dynamique va permettre de résoudre ces problèmes en créant des **tableaux dynamiques**.

```
#include <stdlib.h>
void main()
{
  int n;
  int *tabdyn;

  ...// instructions diverses, dont lecture ou affectation de n

  tabdyn = (int*)malloc(n * sizeof(int));   // allocation dynamique d'un
                // espace mémoire pour tableau de n éléments exactement

  ...// instructions concernant le tableau

  free(tabdyn);   // libération de la place mémoire allouée
}
```

Tableaux multidimensionnels dynamiques

Un tableau 2D *statique* est un tableau de tableaux, déclaré sous la forme :

`int tab[p][n]`

On peut le voir comme un pointeur vers un pointeur, et donc :

- `tab` équivaut à `&tab[0][0]`, l'adresse du premier élément
- pour tout i de 0 à $p-1$, `tab[i]` équivaut à `&tab[i][0]`

Pour rendre dynamiques ces tableaux, on utilise des **pointeurs de pointeurs** :

```
void main()
{ int i, p, n;
  int **tab;                                    // double indirection
  scanf("%d %d", &p, &n) ;
  tab = (int**) malloc(p * sizeof(int*));     // tableau 2D : p lignes de tableaux
  for (i = 0; i < p; i++)
      tab[i] = (int*) malloc(n * sizeof(int)); // n cases dans chaque tableau 1D
      ...  // instructions agissant sur le tableau
  for (i = 0; i < p; i++)
      free(tab[i]); // libération de chaque tableau 1D, i.e. de chaque ligne
  free(tab);     // libération de la 1ère colonne, donc de l'ensemble du tableau
}
```

Chaîne ou tableau de caractères

Une chaîne de caractères est un tableau 1D de caratères dont la taille est éminemment variable

$\Rightarrow$ l'utilisation de pointeurs et d'allocation dynamique est recommandée.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{ int i;
  char *chaine1, *chaine2, *res, *p;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++, p++)
      *p = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++, p++)
      *p = chaine2[i];
  printf("%s\n",res);
}
```

Que fait ce programme ?

Concaténation de chaînes

Tableau de pointeurs

Très utilisés pour manipuler des chaînes de caractères, les tableaux de pointeurs permettent de stocker des chaînes de dimensions différentes.

Exemple : on veut créer un tableau des noms des jours de la semaine `char *tabJour[] = {"Lundi","Mardi","Mercredi","Jeudi","Vendredi","Samedi","Dimanche"} ;`

Chaque élément `tabJour[K]` du tableau est de type *pointeur sur caractère*.

```
#include <stdio.h>     #include <stdlib.h>     #include <string.h>
#define NKar 100     #define NJ 7
char ch[NKar], *tabJour[NJ] ;
void main()
{ for (i = 0 ; i < NJ ; i++)
  { fgets(ch,NKar,stdin); // saisie de chaîne sécurisée
    printf("%s\n", ch);
    k =strlen(ch);
    printf("%d\n", k);
    tabJour[i] = (char*) malloc((k)*sizeof(char));
    strncpy(tabJour[i], ch, k);
    printf("tabJour[%d] = %s\n", i, tabJour[i]);
  }
}
```

Que fait ce programme ?

Remplissage du tableau des noms des jours de la semaine