# IN103 : Sujet du TP3.

Groupe ESIEE, Denis BUREAU, mars 2006.

**Attention !** Le sujet peut être modifié jusqu'à la veille du TP.

## 1 Les objectifs

Être capable de réaliser des programmes en C manipulant les pointeurs et utilisant l'allocation dynamique ; perfectionner son utilisation du debugger.

## 2 Notions vues en cours

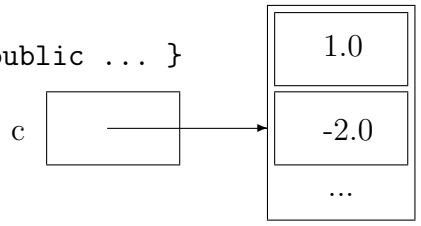
- nouveaux opérateurs : `type *`, `* adresse` , `& variable`
- typage des pointeurs, pointeur générique, pointeur de pointeur, pointeur de struct (`->`),
- valeurs des pointeurs, NULL, pointeurs à ne pas déréférencer
- Équivalence pointeurs ↔ tableaux, Arithmétique des pointeurs
- Allocation dynamique

## 3 Compléments sur l'allocation dynamique (*à parcourir*)

### A) Comparaison avec Java

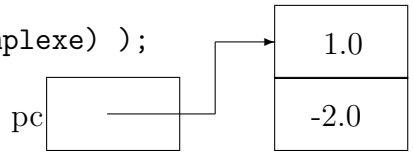
- En java :

```
public class Complexe { private double re, im; public ... }
Complexe c = new Complexe( 1.0, -2.0 );
```




- En C :

```
typedef struct { double re, im; } Complexe;
typedef Complexe * PdeComplexe;
PdeComplexe pc = (PdeComplexe)malloc( sizeof(Complexe) );
pc->re = 1.0;   pc->im = -2.0;
```




### B) Types simples

- 
```
typedef char * PdeCaractere;
PdeCaractere pc = (PdeCaractere)malloc( sizeof(char) );
typedef int * PdEntier;
PdEntier pe = (PdEntier)malloc( sizeof(int) );
```

- `malloc` renvoit `NULL` s'il échoue (par exemple si le système n'a pas trouvé suffisamment de place contiguë en mémoire); il serait donc plus sage de toujours faire suivre
  l'instruction `p = malloc( ... );`
  des instructions `if (p==NULL) { printf( "message d'erreur\n" ); exit(1); }`
- pas de *Garbage Collector* en C; on doit donc désallouer/libérer/rendre la mémoire au système :
  `free( pc );` libère la mémoire pointée par `pc` (ne modifie pas `pc`)
  `pc` doit avoir été initialisé par un `malloc`
  `*pc` est ensuite interdit, mais `pc=malloc(...);` autorisé
- pour utiliser toutes ces possibilités, il faut inclure `<stdlib.h>`.

### C) Tableaux dynamiques

- taille fixe, mais choisie à l'exécution (équivalents aux tableaux java);
- `int N;  printf( "N ? " );  scanf( "%d", &N );`
  `PdEntier pe = (PdEntier)malloc( N * sizeof( int ) );`
  on peut ensuite utiliser `pe[0]` à `pe[N-1]`
- `free( pe );` désalloue tout le tableau d'un coup
- cas particulier : 1 caractère = 1 octet
  `typedef char * PdeCaractere;`
  `#define MAX 20`
  `const PdeCaractere pc = (PdeCaractere)malloc( MAX+1 );`
  équivalent à `char pc[MAX+1];`

### D) Tableaux dynamiques multi-dimensionnels

- impossible directement; créer un tableau dynamique mono-dimensionnel et une fonction d'accès à plusieurs indices
- Exemple :
  `typedef int * Matrice;`
  définir ou saisir le nb de lignes (`NL`) et le nb de colonnes (`NC`)
  `Matrice m = (Matrice)malloc( NL*NC*sizeof(int) );`

| NL \ NC | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | 5 |
| 2 | 6 | 7 | 8 |
| 3 | 9 | 10 | 11 |
| 4 | 12 | 13 | 14 |

- `m[4][1]` pour accéder à la $2^{ème}$ case de la $5^{ème}$ ligne ?  NON, car :
  `m[4][1] = *(m[4]+1) = *( *(m+4) + 1 ) = *(4+1) = *5`
- `Matrice acces( Matrice tab, int nbL, int nbC, int numL, int numC )`
  `{ /* test des indices */ return tab + numL*nbC + numC; }`
  on peut donc écrire `*acces( m, 3, 4, 1 ) = -13;`

# 4 Traduction en C

*(pour se familiariser avec la nouvelle syntaxe, relire notamment le **paragraphe 3.B ci-dessus**)*

Écrire un programme qui contient la traduction en C de l'algorithme suivant :

```
Soit PdeReel le type "Pointeur de Reel".
Dans le programme principal :
Declarer r du type Reel.
Declarer pr et pr2 du type PdeReel.
Faire pointer pr et pr2 sur r.
Initialiser r a 5.4 en passant par pr.
Afficher r en passant par pr2.
```

**Rappel :** La commande de compilation/édition de liens est : `mcc -lm traduction.c -o traduction`

# 5 Retour sur l'exercice 5 du TP2 : `minimum.c`

*(pour utiliser l'allocation dynamique de tableaux, relire notamment le **paragraphe 3.C ci-dessus**)*

1. Recopier l'exercice du précédent TP et remplacer le type tableau de MAX réels par un type pointeur de réels, malgré tout appelé `TabReels` .
2. Conserver telle quelle la fonction `minimum` qui prend en paramètre un `TabReels` et le nombre d'éléments utiles, et qui retourne la plus petite valeur présente dans le tableau.
3. Modifier le `main()` pour qu'il demande la taille du tableau, alloue la mémoire, saisisse le nombre de valeurs nécessaires pour remplir le tableau, appelle la fonction, et affiche le résultat.

# 6 Retour sur l'exercice 2 du TP2 : `Rationnel.c`

*(pour passer des paramètres par adresse et pour utiliser l'allocation dynamique de structures, relire notamment le **paragraphe 3.A ci-dessus**)*

Lire le paragraphe 6.6 ci-dessous pour comprendre comment le `main()` appellera les différents sous-programmes.

1. Recopier l'exercice du précédent TP et conserver la définition d'un type `Rationnel` composé de 2 entiers `num` et `den` , et la compléter avec la définition du type `PtrRat` (pointeur de `Rationnel`).
2. Modifier les deux fonctions `rationnel2()` et `rationnel1()` pour que d'une part, elles allouent dynamiquement le `Rationnel` qu'elles doivent créer/initialiser, et que d'autre part, elles retournent l'adresse de ce `Rationnel`.
   **Rappel :** Chacune de ces 2 fonctions doit être utilisée à chaque fois que c'est possible (notamment, l'une appelle l'autre).
3. Modifier la procédure `affiche` pour qu'elle prenne en paramètre d'entrée non plus un `Rationnel`, mais son adresse.
4. Modifier la fonction `ajoute` pour qu'elle prenne en paramètres d'entrée non plus deux `Rationnel`, mais leurs adresses. D'autre part, elle créera dynamiquement le `Rationnel` résultat dont elle retournera l'adresse (*et pourtant aucun* `malloc()` *dans cette fonction !?*).
5. **Attention !** C'est la transformation la plus difficile ; bien lire tous les mots.
   Transformer la fonction `saisit` en procédure qui fournit en paramètre de sortie l'adresse du rationnel alloué dynamiquement (et dont la valeur a été saisie). La saisie sera précédée par l'affichage du message (chaîne de caractères ou `char *`) passé en paramètre d'entrée.
6. Modifier le `main()` pour tenir compte de tous ces changements :
   - déclaration des 3 pointeurs
   - 2 appels de `saisit()`
   - appel d' `ajoute()`
   - affichages (exemple d'affichage souhaité : `1/3 + 1/6 = 9/18` )
   - libération des 3 zones de mémoire allouées

# 7 Tri de mots

*(pour utiliser l'allocation dynamique de chaînes de caractères)*

1. Écrire un programme qui permettra de saisir des mots, puis qui les affichera par ordre alphabétique. Pour cela, déclarer un type `String` (juste un pointeur de caractère) et le type `Boolean` (par énumération).

2. Écrire une procédure `tri` qui prend en paramètres un tableau de `String` (juste un pointeur de `String`) et le nombre de mots présents dans le tableau.
   **Rappel :** Un tableau étant désigné par l'adresse de sa première case, on peut considérer qu'un paramètre tableau est forcément un paramètre de sortie, d'où la possibilité de modifier les valeurs du tableau dans la fonction de tri.

   - La méthode de tri proposée est celle dite du "tri à bulles". Elle consiste à faire un "parcours total" du tableau tant que le dernier "parcours total" a effectué au moins une modification.
   - Un "parcours total" du tableau consiste à examiner toutes les paires d'éléments consécutifs du début à la fin du tableau, et à échanger ces deux éléments s'ils ne sont pas dans le bon ordre.
   - Pour savoir quand s'arrêter, il faut évidemment mémoriser durant chaque "parcours total" s'il y a eu ou non au moins un échange.
   - Compte tenu de la nature des données à échanger ici, on pourra se contenter d'échanger des pointeurs (de caractères) et ainsi éviter de recopier à chaque fois toute la chaîne de caractères.

3. Écrire un programme principal qui déclare un tableau statique de 4 `String` en l'initialisant avec 4 mots non classés par ordre alphabétique. Appeler ensuite la procédure de tri, puis afficher les 4 mots dans l'ordre du tableau, désormais trié.

4. Écrire maintenant une procédure `saisitMots` qui va devoir fournir 2 paramètres en sortie : l'adresse du tableau de `String` que la procédure aura alloué et le nombre de mots qui auront été saisis et stockés dans le tableau.
   Cette procédure commencera par demander à l'utilisateur combien de mots il veut saisir, allouera la mémoire nécessaire pour contenir le bon nombre de pointeurs de caractères, puis demandera à l'utilisateur de saisir les mots un par un (dans un tableau temporaire dimensionné au maximum à 100 caractères), puis allouera juste la place nécessaire pour stocker chaque mot dans le tableau.

5. Modifier le programme principal pour remplacer le tableau statique de 4 `String` par un simple pointeur de `String` qui sera initialisé grâce à l'appel de `saisitMots()`.

$\longrightarrow$ Avez-vous remarqué que vous avez utilisé un `char***` ?

# 8 Travail personnel

## 8.1 Retour sur l'exercice 6.2 du TP2 : `matrice.c`

*(pour utiliser l'accès à un tableau bidimensionnel alloué dynamiquement)*

- Recopier l'exercice du TP précédent et le modifier pour que l'utilisateur puisse choisir le nombre de lignes de la matrice, d'où allocation dynamique.
- Attention ! L'accès à un tableau dynamique bidimensionnel ne se fait pas simplement. En effet, la fonction `malloc()` alloue un certain nombre d'octets qui représentent donc un tableau unidimensionnel.
- Nous pouvons le considérer comme un tableau bidimensionnel uniquement si nous effectuons nous-même les calculs d'adresse en fonction du nombre de colonnes (par exemple, s'il y a 5 colonnes, le $2^{ème}$ élément de la $3^{ème}$ ligne va se trouver dans la case numéro $2 \times 5 + 1 = 11$ ).
- Le meilleur moyen consiste à définir une fonction d'accès qui retourne l'adresse de la case mémoire concernée en fonction du tableau, du nombre de colonnes, et des 2 indices habituels d'un tableau bidimensionnel (relire notamment le ***paragraphe 3.D ci-dessus***).

## 8.2 Retour sur l'exercice 6.3 du TP2 : `eratosthene.c`

1. Le crible d'Ératosthène est une méthode pour découvrir les nombres premiers. Il consiste à mettre tous les entiers (inférieurs ou égaux à une certaine `LIMITE`) dans une grille, puis à rayer méthodiquement tous les multiples des nombres non encore rayés.

2. Exemple :

```
debut:  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
raye2:  2  3  /  5  /  7  /  9  / 11  / 13  / 15  / 17  / 19  / 21  / 23  / 25
raye3:  2  3     5     7     /    11    13     /    17    19     /    23    25
raye5:  2  3     5     7          11    13          17    19          23     /
raye7:  ...
```

3. On constate alors qu'il ne reste plus que des nombres premiers dans la grille.
   En informatique, il est toutefois difficile de "rayer" un nombre. On choisit donc de déclarer un tableau de booléens dont chaque case représentera le nombre entier correspondant à son indice : si la valeur de la case est `true`, c'est que l'entier correspondant n'est pas rayé. Et rayer un nombre signifiera donc passer la case correspondante à `false`. Pour ne pas compliquer l'écriture de ce programme, on laissera volontairement inutilisée la case d'indice 0.

4. Définir le type `Crible` pour le tableau de booléens.

5. Écrire une procédure `initV` qui initialise correctement les éléments du tableau passé en paramètre, pour signifier qu'a priori, tous les entiers à partir de 2 sont candidats à être premiers. La `LIMITE` est également passée en paramètre.

6. Écrire une procédure `raye` qui applique la méthode décrite ci-dessus en mettant `false` dans le tableau passé en paramètre pour chaque case correspondant à un nombre qu'elle doit "rayer" (inférieur ou égal à la `LIMITE` également passée en paramètre).
   Remarque : il est inutile de parcourir TOUS les nombres jusqu'à la `LIMITE`.
   Question : quand peut-on arrêter la boucle principale ?

7. Écrire une <u>fonction</u> `prepare` qui applique la méthode du crible d'Ératosthène (en appelant les procédures `initV` et `raye`) au tableau de booléens qu'elle commencera par allouer, et qui retourne l'adresse de ce tableau. La `LIMITE` est également passée en paramètre (attention à la case supplémentaire due au zéro).

8. Écrire une procédure `affiche` qui affiche l'intégralité des nombres premiers figurant dans le tableau passé en paramètre (jusqu'à la `LIMITE` également passée en paramètre), à raison de 10 par ligne.

9. Écrire un `main` qui déclare un tableau de booléens, demande quelle action on veut faire (voir ci-dessous), puis "prépare" le tableau et effectue l'action, ceci tant qu'on ne désire pas arrêter le programme.
   Pour choisir l'action, on saisira un nombre de 0 à 3 : 0 pour sortir de la boucle et arrêter le programme, 1 pour appeler la procédure `affiche` (saisir la limite), 2 pour tester si un nombre est premier ou non (saisir ce nombre), 3 pour trouver le plus grand nombre premier inférieur ou égal à un nombre (saisir ce nombre). On supposera que l'utilisateur n'entre que des nombres entiers.

10. Exemple d'affichage (option 1) :

```
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229
...
```