Programmation C
Licence SIRI

Dr Nelson **SAHO**

Institut de Formation et de Recherche en Informatique

10 janvier 2022

Contenu

1 Les pointeurs

Adresse et valeur d'un objet

- Toute variable utilisée dans un programme est stocké quelque part en mémoire. Cette mémoire est constitué d'octets identifiés de manière univoque par un numéro qu'on appelle *adresse*.
- Pour retrouver une variable, il suffit de connaître l'adresse de l'octet où il est stockée ou du moins l'adresse de premier octet.
- C'est le compilateur qui fait le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois il est parfois très pratique de manipuler directement une variable par son adresse.

Adresse et valeur d'un objet (suite)

- On appelle *Lvalue* (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :
  - son adresse, c'est-à-dire l'adresse mémoire à partir de laquelle l'objet est stocké ;
  - sa valeur, c'est-à-dire ce qui est stocké à cette adresse.
  - Dans l'exemple suivant,

```
int i, j;
i = 3;
j = i;
```

Adresse et valeur d'un objet (suite)

- Si le compilateur a placé la variable *i* à l'adresse 4831836000 en mémoire, et la variable *j* à l'adresse 4831836004, on a :

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

- Deux variables différentes ont des adresses différentes. L'affectation $i = j;$ n'opère que sur les valeurs des variables. Les variables *i* et *j* étant de type *int* sont stockés sur 4 octets. Ainsi, la variable *i* est stockée sur les actets d'adresse 4831836000 à 4831836003.

Adresse et valeur d'un objet (suite)

- L'opérateur & permet d'accéder à l'adresse d'une variable. Toutefois, &i n'est pas une Lvalue mais une constante : on ne peut pas figurer &i à gauche d'une opérateur d'affectation.
- Pour pouvoir manipuler des adresse, on doit donc recourir à un nouveau type d'objets, **les pointeurs**.

Notion de pointeurs

- Un pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

```
type *nom-du-pointeur;
```

où *type* est le type de l'objet pointé.

- Cette déclaration déclare un identificateur, *nom-du-pointeur*, associé à un objet dont la valeur est l'adresse d'un autre objet de type *type*.

Notion de pointeurs (suite)

- L'identificateur*nom-du-pointeur* est donc un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.
- La valeur d'un pointeur est toujours un entier (éventuellement un entier long), mais le type d'un pointeur dépend du type de l'objet vers lequel il pointe.
- Un pointeur sur un objet de type *char* donne l'adresse de l'octet où ce caractère est stocké. Par contre, un pointeur sur un objet de type *int*, donne l'adresse du premier des 4 octets où l'objet est stocké.

Notion de pointeurs (suite)

- Soit l'exemple ci-après :

```
int i = 3;
int *p;

p = &i;
```

On se trouve dans la configuration suivante :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

Notion de pointeurs (suite)

- L'opérateur unaire d'indirection $*$ permet d'accéder à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, $*p$ désigne la valeur de i.

```
main () {
    int i = 3;
    int *p;
    p = &i;
    printf("*p = %d \n", *p);
    /*Imprime *p = 3
}
```

Notion de pointeurs (suite)

- Dans ce programme, les objets *i* et **p* sont identiques : ils ont les mêmes adresses et valeurs.

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

- Toute modification de **p* modifie *i*.
- On peut manipuler à la fois *p* et **p* dans un programme.
- **Exercice** : Comparer les deux programmes ci-après :

Notion de pointeurs (suite)

Programme 1

```
main () {
     int i = 3, j = 6;
     int *p1, *p2;
     p1 = &i;
     p2 = &j;
     *p1 = *p2;
}
```

Programme 2

```
main () {
     int i = 3, j = 6;
     int *p1, *p2;
     p1 = &i;
     p2 = &j;
     p1 = p2;
}
```

Notion de pointeurs (suite)

- Avant la dernière affectation de chacun de ces programme, on est dans une configuration du type :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Notion de pointeurs (suite)

- Après l'affectation **p1 = *p2 ;* du Programme 1 on a :

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Notion de pointeurs (suite)

- Par contre, l'affectation *p1 = p2 ;* du Programme 2 conduit à la situation suivante :

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

Arithmétique des pointeurs

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérations arithmétiques classiques. Les opérations arithmétiques valides sont :
  - l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
  - la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.
- Notons que la somme de deux pointeurs n'est pas autorisée.

Arithmétique des pointeurs (suite)

- Si *i* est un entier et *p* un pointeur sur un objet de type *type*, l'expression *p + i* désigne un pointeur sur un objet de type *type* dont la valeur est égale à la valeur de *p* incrémentée de *i * sizeof(type)*. Il en va de même que la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation *++* et de décrémentation *--*.
- Si *p* et *q* sont deux pointeurs sur des objets de type *type*, l'expression *p - q* désigne un entier dont la valeur est égale à $(p - q)/sizeof(type)$.

Arithmétique des pointeurs (suite)

```
main () {
     int i = 3;
     int *p1, *p2;
     p1 = &i;
     p2 = p1 + 1;
     printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

- Ce programme affiche *p1 = 4831835984 p2 = 4831835988*

Arithmétique des pointeurs (suite)

- Ce programme celui ci-dessous affiche *p1 = 4831835984 p2 = 4831835992*

```
main () {
    double i = 3;
    double *p1, *p2;
    p1 = &i;
    p2 = p1 + 1;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

Arithmétique des pointeurs (suite)

- Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.
- L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.
- Le programme suivant inprime les éléments du tableau *tab* dans l'ordre croissant puis décroissant des indices.

Arithmétique des pointeurs (suite)

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main () {
    int *p;
    printf("\n Ordre croissant : \n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf(" %d \n", *p);
    printf("\n Ordre decroissant : \n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf(" %d \n", *p);
}
```

Allocation dynamique

- Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection $*$, il faut l'initialiser. Par défaut, la valeur d'un pointeur est égale à la constante *NULL* définie dans *stdio.h*. En général, elle vaut 0. Le test $p == NULL$ permet de savoir si le pointeur p pointe vers un objet.
- On peut l'initialiser par l'affectation à p de l'adresse d'une autre variable. Il est également possible d'affecter directement une valeur à $*p$. Mais pour cela, il faut d'abord réserver à $*p$ un espace mémoire de taille adéquate. L'adresse de cet espace mémoire sera la valeur de p.

Allocation dynamique (suite)

- Cette opération de réservation est une *allocation dynamique*. Elle se fait en C par la fonction *malloc* de la librairie *stdlib.h*. Sa syntaxe est :

```
malloc(nombre-octets)
```

- Elle retourne un pointeur de type *char ** pointant vers un objet de taille *nombre-octets* octets. Pour initialiser des pointeurs qui ne sont pas de type *char*, il faut convertir le type de sortie de la fonction à l'aide d'un *cast*. L'argument *nombre-octets* peut être donné par *sizeof()*.

Allocation dynamique (suite)

- Ainsi, pour initialiser un pointeur d'entier, on peut écrire :

```
int *p;
p = (int*)malloc(4); /*Ou bien*/
p = (int*)malloc(sizeof(int));
```

- Notons toutefois que la deuxième écriture est préférable car elle est portable.
- Le programme ci-après définit un pointeur *p* sur un objet **p* de type *int*, et affecte à **p* la valeur de la variable *i*.

Allocation dynamique (suite)

```
#include <stdio.h>
#include <stdlib.h>
main () {
    int i = 3;
    int *p;
    printf("Valeur de p avant initialisation = %ld\n", p);
    p = (int*)malloc(sizeof(int));
    printf("Valeur de p après initialisation = %ld\n", p);
    *p = i;
    printf("Valeur de *p = %d\n", *p);
}
```

Allocation dynamique (suite)

- Il imprime à l'écran :

```
Valeur de p avant initialisation = 0
Valeur de p après initialisation = 5368711424
Valeur de *p = 3
```

- Avant l'allocation dynamique, on se trouve dans la configuration :

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

Allocation dynamique (suite)

- A ce statde **p* n'a aucun sens. EN particulier, toute manipulation de la variable **p* générerait une violation mémoire, détectable à l'exécution par le message d'erreur **Segmentation fault**.
- L'allocation dynamique a pour résultat d'attribuer une valeur à *p* et de réserver à ce statde un espace-mémoire composé de 4 octets pour stocker la valeur de **p*. On a alors :

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

Allocation dynamique (suite)

- **p* est maintenant défini mais sa valeur n'est pas initialisée. Cela signifie que **p* peut valoir n'importe quel entier (celui qui se trouvait précédemment à cette adresse). L'affectation $*p = i$; a enfin pour résultat d'affecter à **p* la valeur de *i*. A la fin du programme on a donc :

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

Allocation dynamique (suite)

- Il est important de comparer le programme précédant avec :

```
int i = 3;
int *p;
p = &i;
```

qui correspond à la situation :

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Allocation dynamique (suite)

- Dans ce dernier cas, les variables *i* et **p* sont identiques. Toute modification de l'une modifie donc l'autre. Ce n'est pas le cas dans l'exemple l'ayant précédé où *i* et **p* avaient la même valeur mais des adresses différentes.
- Le dernier programme n'a pas nécéssité d'allocation dynamique puisque l'espace mémoire à l'adresse &i est déjà réservé pour un entier.
- La fonction *malloc* permet également d'allouer un espace pour plusieurs objets contigus en mémoire.

Allocation dynamique (suite)

```
#include <stdio.h>
#include <stdlib.h>
main () {
    int i = 3, j = 6;
    int *p;
    p = (int*)malloc(2 * sizeof(int));
    *p = i;
    *(p+1) = j;
    printf("p = %ld \t *p = %d \t p+1 = %ld \t
    *(p+1) = %d\n", p, *p, p+1, *(p+1));
}
```

Allocation dynamique (suite)

- On a ainsi réservé, à l'adresse donné par la valeur de *p*, 8 octets en mémoire, qui permettent de stocker 2 objets de type *int*. Le programme affiche :

```
p = 5368711424  *p = 3  p+1 = 5368711428 *(p+1) = 6
```

- La fonction *calloc* de la librairie *stdlib.h* a le même rôle que la fonction *malloc* mais elle initialise en plus l'objet pointé **p* à 0. Sa syntaxe est :

```
calloc(nb-objets, taille-objets);
```

Allocation dynamique (suite)

- Ainsi, si *p* est de type *int**, l'instruction :

```
p = (int*)calloc(N, sizeof(int));
```

est strictement équivalente à (son emploi est donc simplement plus rapide) :

```
p = (int*)malloc(N * sizeof(int));
for(i = 0; i < N; i++)
    *(p+i) = 0;
```

Allocation dynamique (suite)

- Lorsqu'on fini d'utiliser le pointeur, il faut libérer cette place en mémoire. Ceci se fait à l'aide de l'instruction *free* qui a pour syntaxe :

```
free(nom-du-pointeur);
```

- A toute instruction *malloc* ou *calloc* est associé une instruction *free*.

Pointeurs et tableaux à une dimension

- L'usage des pointeurs en C est, en grande partie, orienté vers la manipulation des tableaux.
- Tout tableau en C est un pointeur constant. Dans la déclaration *int tab[10];*, *tab* est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, *tab* a pour valeur *&tab[0]*. On peut donc utilisé un pointeur initialisé à *tab* pour parcourir les éléments du tableau.

Pointeurs et tableaux à une dimension (suite)

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main () {
    int i;
    int *p;
    p = tab;
    for(i = 0; i < N; i++) {
        printf(" %d \n", *p);
        p++;
    }
}
```

Pointeurs et tableaux à une dimension (suite)

- On accède à l'élément d'indice i du tableau *tab* grâce à l'opérateur d'indexation [], par l'expression *tab[i]*. Cet opérateur d'indexation peut en fait s'appliquer à tout objet p de type pointeur. Il est lié à l'opérateur d'indirection $*$ par la formule

$$p[i] = *(p + i)$$

- Pointeurs et tableau se manipule donc de la même manière. Par exemple le programme précédant peut aussi s'écrire :

Pointeurs et tableaux à une dimension (suite)

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main () {
    int i;
    int *p;
    p = tab;
    for(i = 0; i < N; i++)
        printf(" %d \n", p[i]);
}
```

Pointeurs et tableaux à une dimension (suite)

- Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi,
  - on ne peut pas créer un tableau dont la taille est une variable du programme ;
  - on ne peut pas créer des tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.
- Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Pour créer un tableau à n éléments, variable du programme on écrit :

Pointeurs et tableaux à une dimension (suite)

```
#include <stdlib.h>
main () {
    int n;
    int *tab;
    ...
    tab = (int*) malloc(n * sizeof(int));
    ...
    free(tab);
}
```

Pointeurs et tableaux à une dimension (suite)

- Si on veut initialiser tous les éléments tu tableau à 0 on utilise *calloc* :

$$tab = (int*)calloc(n, sizeof(int));$$

- Les éléments de *tab* seront manipulés avec l'opérateur d'indexation [] exactement comme dans les tableaux.

Pointeurs et tableaux à une dimension (suite)

- Les deux principales différences entre un tableau et un pointeur sont :
  - un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par une affectation d'une expression d'adresse ;
  - un tableau n'est pas une Lvalue ; il ne peut donc 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 tableaux à plusieurs dimensions

- Un tableau à deux dimensions est, par définition, un tableau de tableau. Il s'agit en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

$$int\ tab[M][N];$$

- *tab* est un pointeur, qui pointe vers un objet lui-même pointeur d'entier.
- *tab* a une valeur constante égale à l'adresse du premier élément du tableau, $\&tab[0][0]$.

Pointeurs et tableaux à plusieurs dimensions (suite)

- De même $tab[i]$, pour i entre 0 et $M-1$, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice i, qui est $\&tab[i][0]$.

- Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

Pointeurs et tableaux à plusieurs dimensions (suite)

- On Déclare un pointeur sur un objet de type *type* $*$:

```
type **nom-du-pointeur;
```

- De même un pointeur qui pointe sur un objet de type *type* $**$ (équivalent à un tableau à trois dimensions) se déclare par :

```
type ***nom-du-pointeur;
```

- **Exercice** : Créer avec un pointeur de pointeur une matrice de k lignes et n colonnes à coefficients entiers.

Pointeurs et tableaux à plusieurs dimensions (suite)

```
main () {
    int k, n;
    int **tab;
    tab = (int**)malloc(k * sizeof(int*));
    for (i = 0; i< k; i++)
        tab[i] = (int*)malloc(n * sizeof(int));
    ...
    for (i = 0; i< k; i++)
        free(tab[i]);
    free(tab);
}
```

Pointeurs et tableaux à plusieurs dimensions (suite)

- La première allocation dynamique réserve pour l'objet pointé par *tab* l'espace mémoire correspondant à k pointeurs sur des entiers, correspondant aux lignes de la matrice.
- Les allocations dynamiques suivantes réservent pour chaque pointeur $tab[i]$ l'espace mémoire nécessaire pour stocker n entiers.
- Si l'on désire que les éléments du tableaux soient initialisés à 0 il faut utiliser *calloc* en lieu et place de *malloc* dans la boucle *for* : $tab[i] = (int*)calloc(n, sizeof(int))$;

Pointeurs et tableaux à plusieurs dimensions (suite)

- Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des ligne $tab[i]$.
- Par exemple si l'on veut que $tab[i]$ contienne exactement $i + 1$ éléments, on écrit :

```
for (i = 0; i< k; i++)
    tab[i] = (int*)malloc((i + 1) * sizeof(int));
```

Pointeurs et chaînes de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type *char*, se terminant par le caractère *nul* "\0". On peut donc manipuler toute chaîne de caractère avec un pointeur de type *char*.
- On pourra définir une chaîne de caractère par :

  ```
  char *chaine;
  ```

  et on peut faire des affectations comme *chaine = "ceci est une chaine" ;* et toute opération valide sur les pointeurs, comme l'instruction *chaine++ ;*.

Pointeurs et chaînes de caractères (suite)

- Le programme suivant impri,e le nombre de caractères d'une chaîne sans compter le caractère nul.

```
main () {
 int i;
 char *chaine;
 chaine = "ceci est une chaine";
 for (i = 0; *chaine != '\0'; i++)
 chaine++;
 printf("nombre de caracteres = %d\n", i);
}
```

Pointeurs et chaînes de caractères (suite)

- Dans la librairie *string.h*, la fonction *strlen* de syntaxe *strlen(chaine)*, où *chaine* est un pointeur sur un objet de type *char*, renvoie la longueur d'une chaîne de caractères.

Pointeurs et chaînes de caractères (suite)

- Le programme suivant, en se servant des pointeurs, réalise la concaténation de deux chaînes de caractères.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main () {
    int i;
    char *chaine1, *chaine2, *res, *p;

    chaine1 = "chaine ";
    chaine2 = "de caracteres";
```

Pointeurs et chaînes de caractères (suite)

```
    res = (char*)malloc((strlen(chaine1) + strlen(chaine2))
    * sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
        *p++ = chaine1[i];
    for (i = 0; i < strlen(chaine2); i++)
        *p++ = chaine2[i];
    printf("%s\n", res);
}
```

Pointeurs et chaînes de caractères (suite)

- On remarquera l'utilisation d'un pointeur intermédiaire *p* qui est indispensable lorsqu'on fait des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de *res*, on aurait évidemment "perdu" la référence sur le dernier caractère de la chaîne. Par exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main () {
int i;
char *chaine1, *chaine2, *res;
chaine1 = "chaine ";
```

Pointeurs et chaînes de caractères (suite)

```
    chaine2 = "de caracteres";
    res = (char*)malloc((strlen(chaine1) +
    strlen(chaine2)) * sizeof(char));
    p = res;
    for (i = 0; i < strlen(chaine1); i++)
        *res++ = chaine1[i];
    for (i = 0; i < strlen(chaine2); i++)
        *res++ = chaine2[i];
    printf("\nnombre de caracteres de res = %d\n",
    strlen(res));
}
```

Pointeurs et chaînes de caractères (suite)

- imprime la valeur 0 puisque *res* a été modifié au cours du programme et pointe maintenant sur le caractère nul.

Pointeurs et structures

- Contrairement aux tableaux, les objets de type structure en C sont des Lvalues. Ils possèdent une adresse correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Ainsi, le programme ci-après crée, à l'aide d'un pointeur, un tableau d'objets de type structure.

```
#include <stdio.h>
#include <stdlib.h>
struct eleve {
     char nom[20];
     int date;
};
```

Pointeurs et structures

```
typedef struct eleve *classe;
main () {
    int n, i;
    classe tab;
    printf("\nnombre d'eleves de la classe = ");
    scanf("%d", &n);
    tab = (classe)malloc(n * sizeof(struct eleve));
    char *chaine1, *chaine2, *res;
    for (i = 0; i < n; i++) {
        printf("\nSaisie de l'eleves numero %d\n", i);
        printf("\nnom de l'eleve = ");
```

Pointeurs et structures (suite)

```
        scanf("%s", &tab[i].nom);
        printf("\n date de naissance JJMMAA = ");
        scanf("%d", &tab[i].date);
    }
    printf("\n Entrer un numero ", i);
    scanf("%d", &i);
    printf("\n Eleve numero %d :", i);
    printf("\n nom = %s", tab[i].nom);
    printf("\n date de naissance JJMMAA = %d\n", tab[i].date
    free(tab)
}
```

Pointeurs et structures (suite)

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

$$(*p).champ$$

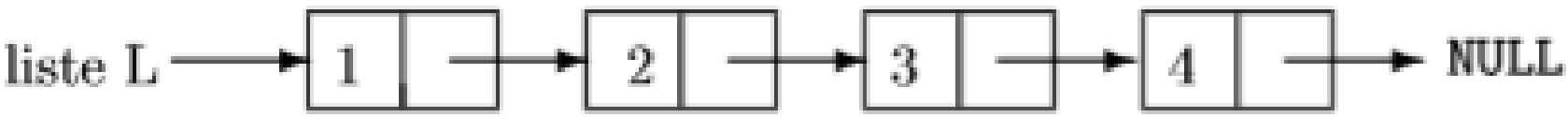
- L'usage de la parenthèse est ici indispensable car l'opérateur d'indirection $*$ a une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté $->$. Alors on écrira simplement

$$p->membre$$

Pointeurs et structures : structures auto-référencées

- Ainsi, dans le programme précédant, on peut remplacer $tab[i].nom$ et $tab[i].date$ respectivement par $(tab + i) -> nom$ et $(tab + i) -> date$.
- On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle. Cette représentation permet de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutesfois ; cette représentation, dite contiguë, impose que la taille maximale de la liste soit connue à priori (on a besoin du nombre d'élément du tableau lors de l'allocation dynamique).

Pointeurs et structures : structures auto-référencées (suite)



- Pour résoudre ce problème, on utilise une représentation chaînée : l'élément de base de la chaîne est une structure appelée *cellule* qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide NULL. La liste est alors définie comme un pointeur sur le premier élément de la chaîne.

Pointeurs et structures : structures auto-référencées (suite)

- Pour représenter une liste d'entiers sous forme chaînée, on crée le modèle de structure *cellule* qui a deux champs : un champ *valeur* de type *int*, et un champ *suivant* de type pointeur sur une *struct cellule*. Une liste sera alors un objet de type pointeur sur une *struct cellule*.

```
struct cellule {
    int valeur;
    struct cellule *suivant;
}

typedef struct cellule *liste;
```

Pointeurs et structures : structures auto-référencées (suite)

- Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un endroit à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

```
liste insere(int element, liste Q) {
    liste L;
    L = (liste)malloc(sizeof(struct cellule));
    L->valeur = element;
    L->suivant = Q;
    return (L);
}
```

Pointeurs et structures : structures auto-référencées (suite)

- Le programme suivant crée une liste d'entiers et l'imprime à l'écran :

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

struct cellule {
    int valeur;
    struct cellule *suivant;
}

typedef struct cellule *liste;
```

Pointeurs et structures : structures auto-référencées (suite)

```
:

liste insere(int element, liste Q) {
    liste L;
    L = (liste)malloc(sizeof(struct cellule));
    L->valeur = element;
    L->suivant = Q;
    return (L);
}
```

Pointeurs et structures : structures auto-référencées (suite)

```
:

main () {
    liste L, P;
    L = insere(1, insere(2, insere(3, insere(4, NULL))));
    printf("\n impression de la liste : \n");
    P = L;
    while (P != NULL) {
        printf("%d \t", P->valeur);
        P = P->suivant;
    }
}
```

Pointeurs et structures : structures auto-référencées (suite)

- On utilisera également une structure auto-référencée pour créer un arbre binaire :

```
struct noeud {
    int valeur;
    struct noeud *fils_gauche;
    struct noeud *fils_droit;
}

typedef struct noeud *arbre;
```

Merci pour votre attention.
Commentaires ? Questions ?