

Pointeurs, allocation dynamique



Rappels

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Rappels

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Le type d'un pointeur est fonction du type de la valeur pointée.

Déclaration d'un pointeur sur un entier

```
int* pt;
```

Rappels

Les adresses sont considérées comme des types particuliers appelés **pointeur**.

Le type d'un pointeur est fonction du type de la valeur pointée.

Déclaration d'un pointeur sur un entier

```
int* pt;
```

- opérateur & qui délivre l'adresse d'une variable
- opérateur * qui délivre la valeur pointée par un pointeur

```
int b = 5;  
int* pt = &b;  
printf("b vaut %d et est à l'adresse %p\n", b, &b);  
printf("b vaut %d et est à l'adresse %p\n", *pt, pt);  
int **p = &pt
```

Arithmétiques des pointeurs

Opérations valides :

- pointeur `ptr` + entier `n` est un pointeur du même type que `ptr` dont la valeur vue comme un entier vaut

$$\text{ptr} + n * \text{sizeof}(*ptr)$$

Arithmétiques des pointeurs

Opérations valides :

- pointeur `ptr` + entier `n` est un pointeur du même type que `ptr` dont la valeur vue comme un entier vaut

$$\text{ptr} + n * \text{sizeof}(*ptr)$$

- affectation,

Arithmétiques des pointeurs

Opérations valides :

- pointeur `ptr` + entier `n` est un pointeur du même type que `ptr` dont la valeur vue comme un entier vaut

`ptr + n * sizeof(*ptr)`

- affectation,
- différence de 2 pointeurs de même type : valeur de type `int`,

Arithmétiques des pointeurs

Opérations valides :

- pointeur `ptr` + entier `n` est un pointeur du même type que `ptr` dont la valeur vue comme un entier vaut

$$\text{ptr} + n * \text{sizeof}(*ptr)$$

- affectation,
- différence de 2 pointeurs de même type : valeur de type `int`,
- comparaison de pointeurs de même type.

Le pointeur void*

`void*` est un pointeur générique compatible avec tous les autres pointeurs.

- Tout pointeur peut être converti en `void*`,

Le pointeur void*

`void*` est un pointeur générique compatible avec tous les autres pointeurs.

- Tout pointeur peut être converti en `void*`,
- Tout pointeur `void*` peut être converti en pointeur sur n'importe quel type,

Le pointeur void*

void* est un pointeur générique compatible avec tous les autres pointeurs.

- Tout pointeur peut être converti en void*,
- Tout pointeur void* peut être converti en pointeur sur n'importe quel type,
- Les opérations suivantes ne sont pas valides avec une variable p de type void* :
 - accès à la valeur pointée *p
 - addition d'un entier p + n
 - différence de pointeurs p1 - p2
 - comparaison de pointeurs p1 == p2, p1 < p2

Relation entre tableaux et pointeurs

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

tab est de type tableau de 10 entiers.

Relation entre tableaux et pointeurs

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

tab est de type tableau de 10 entiers.

“Tout” appel à l’identificateur tab dans une expression

- est converti en **pointeur constant** sur entier,
- vaut l’adresse du premier élément du tableau tab[0].

Relation entre tableaux et pointeurs

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

tab est de type tableau de 10 entiers.

“Tout” appel à l’identificateur tab dans une expression

- est converti en **pointeur constant** sur entier,
- vaut l’adresse du premier élément du tableau tab[0].

excepté avec l’opérateur sizeof et avec l’opérateur &

```
printf("Taille mémoire du tableau: %d\n",sizeof(tab));  
printf("Taille du tableau\n: %d",sizeof(tab) / sizeof(int));  
printf("%d %d\n",sizeof(tab[0]), sizeof(&tab[0]));  
printf("%d %d\n",sizeof(*&tab), sizeof(*&tab[0]));
```

Conséquences de la conversion automatique pour les tableaux

- Le tableau n'est pas considéré dans son intégralité :

```
int t1[10];  
int t2[10];  
/*t1 = t2; pas possible*/
```

Conséquences de la conversion automatique pour les tableaux

- Le tableau n'est pas considéré dans son intégralité :

```
int t1[10];  
int t2[10];  
/*t1 = t2;  pas possible*/
```

- le tableau est converti en pointeur constant, on ne peut le modifier :

```
int *p;  
int t[10];  
/* t = p;          interdit   */  
p = t;             /*  valide   */
```


Comment passer malgré tout un tableau en paramètre d'une fonction ?

Comment passer malgré tout un tableau en paramètre d'une fonction ?

Le passer par adresse :

```
void fonction(int (*p)[10]){  
    int i;  
    for(i=0; i<10; i++)  
        printf("%d", (*p)[i])  
    (...)  
}
```

```
int t[10];  
fonction(&t);
```

Opérateur d'indexation

`tab[i]` est l'élément d'indice `i` du tableau `tab`

Il est équivalent à

$$*(tab + i)$$

en effet, `tab + i` est l'adresse de l'élément d'indice `i` du tableau.

Opérateur d'indexation

`tab[i]` est l'élément d'indice `i` du tableau `tab`

Il est équivalent à

$$*(tab + i)$$

en effet, `tab + i` est l'adresse de l'élément d'indice `i` du tableau.

- l'opérateur d'indexation `[]` ne sert qu'à la lisibilité des programmes

Opérateur d'indexation

`tab[i]` est l'élément d'indice `i` du tableau `tab`

Il est équivalent à

`*(tab + i)`

en effet, `tab + i` est l'adresse de l'élément d'indice `i` du tableau.

- l'opérateur d'indexation `[]` ne sert qu'à la lisibilité des programmes
- l'opérateur d'indexation peut être utilisé pour n'importe quel pointeur

```
int t[10];  
int* p;  
p = &t[4]; // ou p = t+4
```

Opérateur d'indexation

`tab[i]` est l'élément d'indice `i` du tableau `tab`

Il est équivalent à

`*(tab + i)`

en effet, `tab + i` est l'adresse de l'élément d'indice `i` du tableau.

- l'opérateur d'indexation `[]` ne sert qu'à la lisibilité des programmes
- l'opérateur d'indexation peut être utilisé pour n'importe quel pointeur

```
int t[10];  
int* p;  
p = &t[4]; // ou p = t+4
```

- l'addition `tab + i` est commutative donc `tab[i]` peut aussi s'écrire `i[tab]`.

Passage de tableau en paramètre

La conversion automatique des tableaux fait que l'adresse du tableau est passée en paramètre des fonctions :

```
void afficheTableau(int tab[], int N){  
    int i;  
    for(i=0; i<N; printf("%d\n",tab[i]), i++);  
    printf("%d\n", sizeof(tab));  
    tab = &i;  
}
```

peut s'écrire

```
void afficheTableau(int* tab, int N){  
    ...  
}
```

Tout se passe comme si le tableau était passé par adresse dans la fonction.

Tableau multidimensionnel

```
int t[10][20];
```

- le compilateur alloue une zone mémoire permettant de stocker de manière contiguë 10 tableaux de 20 entiers,
- toute référence à `t` est convertie en pointeur de tableaux de 20 `int`.

Tableau multidimensionnel

```
int t[10][20];
```

- le compilateur alloue une zone mémoire permettant de stocker de manière contiguë 10 tableaux de 20 entiers,
- toute référence à `t` est convertie en pointeur de tableaux de 20 `int`.

Dans le passage d'un tableau multi-dimensionnel à une fonction, il est nécessaire d'indiquer la taille des dimensions à l'exception de la première.

En effet l'adresse de `t[i][j]` est

(adresse de `t`) + (i*20 + j) * sizeof(int)

```
void fonction(int tab[][20]){  
    ...  
}
```

Qu'affiche le programme suivant ?

```
int i;  
char t[7] = {'b','o','n','j','o','u','r'};  
for(i=6; i>=0; i--)  
    printf("%c", (6-i)[t] + 'A' - 'a');  
printf("\n");
```

Classification de la mémoire d'un processus

Lors de l'exécution d'un programme en C, 3 zones de mémoires distinctes sont utilisées :

- une zone contenant les **variables statiques**,
- une zone contenant les **variables automatiques**,
- une zone contenant les **variables dynamiques**.

Classification de la mémoire d'un processus

Lors de l'exécution d'un programme en C, 3 zones de mémoires distinctes sont utilisées :

- une zone contenant les **variables statiques**,
- une zone contenant les **variables automatiques**,
- une zone contenant les **variables dynamiques**.

Ce découpage n'est pas spécifique au langage C.

Mais en général la classe d'une variable est définie par le langage.

Variables statiques

On connaît à l'avance la taille des données.

La mémoire est réservée dès le début du programme.

Variables statiques

On connaît à l'avance la taille des données.

La mémoire est réservée dès le début du programme.

Rendre une variable statique :

- la déclarer en dehors de toute fonction,

Variables statiques

On connaît à l'avance la taille des données.

La mémoire est réservée dès le début du programme.

Rendre une variable statique :

- la déclarer en dehors de toute fonction,
- ajouter le mot clef `static` à la déclaration

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

Variables statiques

On connaît à l'avance la taille des données.

La mémoire est réservée dès le début du programme.

Rendre une variable statique :

- la déclarer en dehors de toute fonction,
- ajouter le mot clef `static` à la déclaration

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


Variables statiques

On connaît à l'avance la taille des données.

La mémoire est réservée dès le début du programme.

Rendre une variable statique :

- la déclarer en dehors de toute fonction,
- ajouter le mot clef `static` à la déclaration

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

Propriétés des variables statiques :

- la durée de vie des variables statiques est globale,
- les variables définies à l'intérieur d'une fonction ne sont visibles qu'à l'intérieur de cette fonction.

Variables automatiques

Variables déclarées à l'intérieur d'une fonction sans modificateur de classe mémoire (ou avec `auto`).

Variables automatiques

Variables déclarées à l'intérieur d'une fonction sans modificateur de classe mémoire (ou avec `auto`).

Propriétés des variables automatiques :

- durée de vie : supprimée à la fin du bloc d'instructions dans lequel elle est déclarée,
- les variables ne sont visibles qu'à l'intérieur du bloc dans lequel elles sont déclarées.

Variables automatiques

Variables déclarées à l'intérieur d'une fonction sans modificateur de classe mémoire (ou avec `auto`).

Propriétés des variables automatiques :

- durée de vie : supprimée à la fin du bloc d'instructions dans lequel elle est déclarée,
- les variables ne sont visibles qu'à l'intérieur du bloc dans lequel elles sont déclarées.

Ces variables sont gérées par une **pile**.

Variables dynamiques

La gestion de la mémoire pour les variables dynamiques est gérée par le programmeur.

Les variables dynamiques sont allouées dans une zone mémoire appelée **tas**.

Variables dynamiques

La gestion de la mémoire pour les variables dynamiques est gérée par le programmeur.

Les variables dynamiques sont allouées dans une zone mémoire appelée **tas**.

Avantages et inconvénients du tas :

- la taille du tas est beaucoup plus importante que celle de la pile,
- la gestion du tas est plus complexe que celle de la pile.

Variables dynamiques

La gestion de la mémoire pour les variables dynamiques est gérée par le programmeur.

Les variables dynamiques sont allouées dans une zone mémoire appelée **tas**.

Avantages et inconvénients du tas :

- la taille du tas est beaucoup plus importante que celle de la pile,
- la gestion du tas est plus complexe que celle de la pile.

L'utilisation du tas est donc conseillé pour la gestion de gros ensembles de données.

Gestion de la mémoire dynamique

Il n'existe pas de mot clef pour déclarer une variable dans l'espace dynamique.

Gestion de la mémoire dynamique

Il n'existe pas de mot clef pour déclarer une variable dans l'espace dynamique.

2 fonctions fondamentales de la librairie `stdlib` :

- `malloc` : réserve la mémoire dans le tas,
- `free` : libère la mémoire dans le tas.

Gestion de la mémoire dynamique

Il n'existe pas de mot clef pour déclarer une variable dans l'espace dynamique.

2 fonctions fondamentales de la librairie `stdlib` :

- `malloc` : réserve la mémoire dans le tas,
- `free` : libère la mémoire dans le tas.

Propriétés des variables dynamiques :

- accessibles uniquement par leur adresse dans le tas,
- existent jusqu'à ce que la mémoire soit libérée.

Allocation de mémoire dans le tas

```
void* malloc(size_t taille)
```

- Demande à l'OS de réserver `taille` octets dans le tas ;
- si échec, renvoie l'adresse `NULL`,
- sinon réserve une zone mémoire contiguë et renvoie l'adresse du début de la zone.

Allocation de mémoire dans le tas

```
void* malloc(size_t taille)
```

- Demande à l'OS de réserver `taille` octets dans le tas ;
- si échec, renvoie l'adresse NULL,
- sinon réserve une zone mémoire contiguë et renvoie l'adresse du début de la zone.

Attention : aucun message d'erreur en cas d'échec.

Allocation de mémoire dans le tas

```
void* malloc(size_t taille)
```

- Demande à l'OS de réserver `taille` octets dans le tas ;
- si échec, renvoie l'adresse `NULL`,
- sinon réserve une zone mémoire contiguë et renvoie l'adresse du début de la zone.

Attention : aucun message d'erreur en cas d'échec.

Exemple de tableau dynamique :

```
int taille = 5;
int i;
int* tab = malloc(taille * sizeof(int));
if(tab == NULL) ; //instruction adéquate
for(i=0; i<taille; i++) tab[i] = i;
```

Autres fonctions d'allocation

```
void* calloc(size_t nb_blocs, size_t taille)
```

Essaie d'allouer `nb_blocs` de taille `taille`.

En cas de succès, la mémoire est initialisée à 0.

Autres fonctions d'allocation

```
void* calloc(size_t nb_blocs, size_t taille)
```

Essaie d'allouer `nb_blocs` de taille `taille`.

En cas de succès, la mémoire est initialisée à 0.

```
void* realloc(void* ptr, size_t taille)
```

Modifie la taille de la zone pointée par `ptr`.

Le contenu est entièrement conservé en cas d'allocation d'une taille plus grande.

Libération de la mémoire

```
void free (void* p)
```

Permet de libérer explicitement la zone mémoire pointée par p.

Libération de la mémoire

```
void free (void* p)
```

Permet de libérer explicitement la zone mémoire pointée par p.

En théorie, pas utile de libérer la mémoire à la fin du programme car le système s'en charge à la fin de l'exécution.

En pratique, il vaut mieux le faire.

Libération de la mémoire

```
void free (void* p)
```

Permet de libérer explicitement la zone mémoire pointée par p.

En théorie, pas utile de libérer la mémoire à la fin du programme car le système s'en charge à la fin de l'exécution.

En pratique, il vaut mieux le faire.

Logiciel pour détecter les fuites mémoires (entre autres problèmes) :
valgrind.

Un programme inutile

```
for(i=0; i<taille; i++)  
    *((int*) malloc(sizeof(int))) = i;  
...  
//free(???)
```

Un programme inutile

```
for(i=0; i<taille; i++)  
    *((int*) malloc(sizeof(int))) = i;  
...  
//free(???)
```

Morale : l'adresse des variables dynamiques doit être conservée jusqu'à leur libération.

Tableau dynamique à 2 dimensions

```
int largeur = 7;
int hauteur = 3;

int** tableau = malloc(hauteur * sizeof(int*))
for(i=0; i<hauteur; i++)
    tableau[i] = malloc(largeur * sizeof(int));
for(i=0; i<hauteur; i++)
    for(j=0; j<largeur; j++)
        tableau[i][j] = 0;

/***** libération mémoire *****/
for(i=0; i<hauteur; i++)
    free(tableau[i]);
free(tableau);
```

Tableau dynamique de types structurés

```
struct toto{  
    ...  
}
```

```
(struct toto)* tableau = malloc(taille * sizeof(struct toto));
```

```
...
```

```
free(tableau);
```

Listes chaînées

Permet de stocker un ensemble de variables de même type :

```
struct element{  
    type_var val;  
    (struct element)* suiv;  
}
```

Listes chaînées

Permet de stocker un ensemble de variables de même type :

```
struct element{  
    type_var val;  
    (struct element)* suiv;  
}
```

Propriétés des listes chaînées :

- taille dynamique, donc (de préférence) implémentée dans le tas,
- pas d'opérateur d'indexation (accès à un élément en $O(n)$),
- adaptée pour les fonctions récursives car une sous-liste est aussi une liste.

Listes chaînées

Possibilités pour faire référence à une liste :

- ① utiliser le premier élément de la liste,
- ② utiliser un pointeur sur le premier élément

Listes chaînées

Possibilités pour faire référence à une liste :

- ① utiliser le premier élément de la liste,
- ② utiliser un pointeur sur le premier élément

La deuxième solution est préférable car elle permet

- de définir une liste vide par l'adresse NULL indépendamment du type des variables contenues dans la liste,
- le champ suiv d'un élément de la liste est du type liste, ce qui facilite la récursivité.

Listes chaînées

Possibilités pour faire référence à une liste :

- ① utiliser le premier élément de la liste,
- ② utiliser un pointeur sur le premier élément

La deuxième solution est préférable car elle permet

- de définir une liste vide par l'adresse NULL indépendamment du type des variables contenues dans la liste,
- le champ suiv d'un élément de la liste est du type liste, ce qui facilite la récursivité.

```
struct element{  
    int val;  
    (struct element)* suivant;  
};  
typedef struct element Element;  
typedef Element* Liste;
```

Listes chaînées : quelques fonctions

- Initialization d'une liste :

```
Liste l = NULL;
```

Listes chaînées : quelques fonctions

- Initialization d'une liste :

```
Liste l = NULL;
```

- Ajout d'un élément au début de la liste

```
Element* elt = malloc( sizeof(Element) );
```

```
elt->val = XXX;
```

```
elt->suiv = l;
```

```
l = elt;
```

Listes chaînées : quelques fonctions

- Initialization d'une liste :

```
Liste l = NULL;
```

- Ajout d'un élément au début de la liste

```
Element* elt = malloc( sizeof(Element) );
```

```
elt->val = XXX;
```

```
elt->suiv = l;
```

```
l = elt;
```

- Nombre d'éléments d'une liste

```
int taille(Liste l){
```

```
    l == NULL? return 0 : return 1+taille(l->suiv);
```

```
}
```

Listes chaînées : quelques fonctions

- Initialization d'une liste :

```
Liste l = NULL;
```

- Ajout d'un élément au début de la liste

```
Element* elt = malloc( sizeof(Element) );  
elt->val = XXX;  
elt->suiv = l;  
l = elt;
```

- Nombre d'éléments d'une liste

```
int taille(Liste l){  
    l == NULL? return 0 : return 1+taille(l->suiv);  
}
```

- Libérer liste

```
void libere(Liste l){  
    if(l != NULL){  
        libere(l->suiv); free(l);  
    }  
}
```