

Pointeurs

Marie Pelleau

`marie.pelleau@univ-cotedazur.fr`

Basé sur les transparents de Jean-Charles Régin

Pointeurs

- Les pointeurs sont très utilisés en C
- En général, ils permettent de rendre les programmes
 - plus compacts
 - plus efficaces
- L'accès à un objet se fait de façon indirecte

Pointeurs

```
char c /* alloue de la mémoire pour c */  
c = 'a'; /* met 'a' dans la case mémoire correspondant à c */
```

- On ne peut pas décider de l'emplacement mémoire (adresse) de c
- On ne peut pas changer l'adresse de c

```
char *p; /* pointeur représenté avec une * */  
p = &c; /* la valeur d'un pointeur est une adresse */
```

Déréférencement : opérateur *

Permet d'interpréter le contenu de la case située à l'adresse du pointeur

```
char c2;  
c2 = *p; /* lecture de la case située à l'adresse du pointeur */  
*p = 'z'; /* écriture dans la case située à l'adresse du pointeur */
```

Pointeurs

```
char c;  
c = 'a';
```



```
char *p;  
p = &c;
```



```
char c2;  
c2 = *p;
```



Pointeurs

```
char c;  
c = 'a';
```



```
char *p;  
p = &c;
```



```
char c2;  
c2 = *p;
```



```
*p = 'z';
```

Pointeurs

- `px = &x`
Si je modifie `*px` alors je modifie `x` (même case mémoire concernée)
- `px = &y`
Si je modifie `*px` alors je modifie `y` (même case mémoire concernée)
- `px = &bidule`
Si je modifie `*px` alors je modifie `bidule` (même case mémoire concernée)
- `px` désigne l'objet pointé
- `*px` modifie l'objet pointé

Pointeurs

Types des pointeurs

- Il faut interpréter le contenu des cases mémoires (2 octets pour un `short`, 4 pour un `int`, big-endian, small-endian, ...)
- On va donc typer les pointeurs pour obtenir ce résultat
- `int *p` veut dire que `*p` sera un `int`, donc `int y = *p` est parfaitement ok

Pointeurs

A quoi cela servent-il ?

- accès direct à la mémoire
- **passage de paramètres**
- partage d'objets
- indirection
- passage par référence
- allocation dynamique

Pointeurs : passage de paramètre

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
  
double calculMoyenne (struct etudiant etud) {  
    int i;  
    int sum=0;  
    for (i = 0; i < 10; i++) {  
        sum += etud.notes[i];  
    }  
    return (double)sum / 10;  
}
```

Que se passe t'il quand on passe un étudiant en paramètre ?

- Il y a création d'une variable locale
 - on réalloue de la mémoire pour cette structure locale
 - la structure est entièrement copiée ! Donc au moins 365 opérations !

Pointeurs : passage de paramètre

Comment éviter cela ?

- On définit un tableau global de structure et on passe l'indice de l'élément dans le tableau (je ne vois pas d'autres solutions sans les pointeurs)
- **GROS défauts**
 - Cela impose des données globales (je ne peux pas passer le tableau ! Sinon copie !)
 - Cela impose une structure de tableau, comment gère-t-on les suppressions/ajouts ?
 - Je dois connaître la taille du tableau au début du programme
- C'est pratiquement injouable

Pointeurs : passage de paramètre

```
struct etudiant {  
    char nom[50];  
    char prenom[50];  
    char adresse[255];  
    int notes[10];  
};  
double calculMoyenne (struct etudiant etud) {  
    int i;  
    int sum=0;  
    for (i = 0; i < 10; i++) {  
        sum += etud.notes[i];  
    }  
    return (double)sum / 10;  
}
```

Solution : on utilise un pointeur sur la structure d'un étudiant

```
double calculMoyenne (struct etudiant *etud) {  
    int i, sum=0;  
    for (i = 0; i < 10; i++) {  
        sum += (*etud).notes[i];  
    }  
    return (double)sum/10;  
}
```

On ne passe que l'adresse mémoire et
on travaille avec cette adresse mémoire :
il n'y a pas de copie locale

Pointeurs

A quoi cela servent-il ?

- accès direct à la mémoire
- passage de paramètres
- **partage d'objets**
- indirection
- passage par référence
- allocation dynamique

Pointeurs : partages d'objets

Pour chaque note on veut connaitre celui qui a la meilleure note

- Premier (meilleure note) en C
- Premier en Système
- Premier en Algorithmique
- Premier en Anglais ...

Comment faire ?

- On fait une copie à chaque fois : mauvaise solution, problème de synchronisation entre les copies
- On utilise des indices pour désigner chaque étudiant : problème demande une gestion sous la forme de tableau des étudiants (avec les inconvénients vu)
- On utilise des pointeurs

Pointeurs : partages d'objets

Pour chaque note on veut connaître celui qui a la meilleure note

- Premier (meilleure note) en C
- Premier en Système
- Premier en Algorithmique
- Premier en Anglais ...

On utilise des pointeurs

```
struct etudiant *pC;  
struct etudiant *pSE;  
struct etudiant *pAlgo;  
struct etudiant *pAnglais;
```

Un même élément peut être partagé !

Pointeurs

A quoi cela servent-il ?

- accès direct à la mémoire
- passage de paramètres
- partage d'objets
- **indirection**
- passage par référence
- allocation dynamique

Pointeurs : indirection

- L'indice d'un tableau est différent du contenu : on fait une indirection
- C'est pareil avec les pointeurs
- Plus petit élément d'un ensemble (`ppelt`)
 - Avec un tableau d'éléments de type `t`, `ppelt` est un indice
 - Avec n'importe quelle structure de données d'éléments de type `t`, `ppelt` est un pointeur de type `t` (`t *ppelt`)

Pointeurs

A quoi cela servent-il ?

- accès direct à la mémoire
- passage de paramètres
- partage d'objets
- indirection
- **passage par référence**
- allocation dynamique

Pointeurs : passage par référence

Comment modifier un paramètre avec une fonction ?

```
double calculMoyenne (struct etudiant *etud) {  
    /* on calcule la moyenne */  
    return (double)sum / 10;  
}
```

On voudrait faire

```
double calculMoyenne (struct etudiant *etud, double moy) {  
    /* on calcule la moyenne */  
    moy = (double)sum/10;  
}
```

Cette solution ne marche pas car moy est copié dans une variable locale

Pointeurs : passage par référence

Un pointeur contient une adresse mémoire

Si on déréférence on touche directement à l'adresse mémoire. C'est ce qu'il nous faut !

```
double calculMoyenne (struct etudiant *etud, double *moy) {  
    /* on calcule la moyenne */  
    *moy = (double)sum / 10;  
}
```

Cette solution marche car on modifie ce qui est à l'adresse mémoire passée en paramètre. C'est l'adresse qui est passée et non plus la valeur

Pointeurs

A quoi cela servent-il ?

- accès direct à la mémoire
- passage de paramètres
- partage d'objets
- indirection
- passage par référence
- **allocation dynamique**

Pointeurs : allocation dynamique

- On ne peut pas toujours connaître la taille d'une structure de données au moment où on écrit le code source
 - Nombre d'étudiants à l'université ? Nombre d'arbres dans un jardin ?
- On peut surestimer mais cela peut créer des problèmes
- Allocation dynamique
 - Comment allouer n éléments, avec n qui sera défini à l'exécution ?
 - Comment définir cette variable dans le programme ?

Pointeurs : allocation dynamique

- Allocation dynamique
 - Comment allouer n éléments, avec n qui sera défini à l'exécution ?
 - Comment définir cette variable dans le programme ?
- **Solution** : on travaille en fait avec des cases mémoires avec les pointeurs
 - on n'a pas besoin que la mémoire soit définie,
 - on a juste besoin de savoir que c'est la mémoire à un certain endroit qui va être utilisée
- On allouera cette mémoire (cela veut dire on réservera cette place mémoire) après quand on connaîtra la taille, en utilisant des fonctions spéciales : `malloc`, `calloc`, `realloc`

Pointeurs : allocation dynamique

- Allocation dynamique
 - Comment allouer n éléments, avec n qui sera défini à l'exécution ?
 - Comment définir cette variable dans le programme ?

```
int *tab; /* tab est un tableau */
int n;
tab = calloc(n, sizeof(int)); /* réserve de la place pour
    n entiers */
/* calloc retourne une adresse mémoire et dit à l'OS que la
    place est réservée à cet endroit */
```

Pointeurs : allocation mémoire

- En utilisant les fonctions standard suivantes, l'utilisateur a la garantie que la mémoire allouée est contigüe et respecte les contraintes d'alignement
- Ces fonctions se trouvent dans le fichier de déclarations `stdlib.h`
 - `void *malloc (size_t taille);`
permet d'allouer `taille` octets dans le tas
 - `void *calloc (size_t nb, size_t taille);`
permet d'allouer `taille × nb` octets dans le tas, en les initialisant à 0
 - `void *realloc (void *ptr, size_t taille);`
permet de réallouer de la mémoire
 - `void free (void *ptr);`
permet de libérer de la mémoire (ne met pas `ptr` à `NULL`)

Pointeur

- Un pointeur est un type de données dont la valeur fait référence (référencie) directement (pointe vers) à une autre valeur
- Un pointeur référencie une valeur située quelque part en mémoire en utilisant son adresse
- Un pointeur est une variable qui contient une adresse mémoire
- Un pointeur est un type de données dont la valeur **pointe vers** une autre valeur.
- Obtenir la valeur vers laquelle un pointeur pointe est appelé **déréférencer** le pointeur.
- Un pointeur qui ne pointe vers aucune valeur aura la valeur **NULL** ou 0
- En Java **TOUT** est pointeur

Pointeur

Comment déclarer un pointeur ?

```
type *nom_pointeur;
```

```
char *p;
```

```
int *p1, *p2;
```

```
struct {int x, y;} *q;
```

```
void *r;
```

```
int *s1[3];
```

```
int (*fct)(void);
```

```
int (*T[5])(void);
```

```
double *f(void);
```

Pointeur

- Adresse d'une variable `x` est désignée en utilisant `&x`;
- Pointeur "universel" : `void *`
- Pointeur sur "rien du tout" : constante entière 0 ou `NULL` (macro définie dans le fichier de déclarations `stdlib.h`)
- Simplification d'écriture : si `q` est un pointeur sur une structure contenant un champ `x`
`q->x` est équivalent à `(*q).x`

Références fantômes !

```
#include <stdlib.h>

int *f1 (void) {
    int a = 3;
    return &a;
    /* warning: function returns address of local variable */
}

int *f2 (void) {
    int *a = malloc(sizeof(int));
    return a;
}

int main (int argc, char *argv[]) {
    int *p1 = f1();
    int *p2 = f2();
    return 0;
}
```

Que se passe-t-il ?

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i = 0;
    char c, *pc;
    int *pi;

    printf("sizeof(char) = %lu, sizeof(int) = %lu,\nsizeof(void
        *) = %lu\n\n", sizeof (char), sizeof (int), sizeof (void
        *));
    printf("&i = %p,\n&c = %p,\n&pc = %p,\n&pi = %p\n\n", (void
        *)&i, (void *)&c, (void *)&pc, (void *)&pi);

    c = 'a';
    pi = &i;
    pc = &c;
    *pi = 50;
    *pc = 'B';
    printf("i = %d, c = %c,\npc = %p, *pc = %c,\npi = %p, *pi =
        %d\n", i, c, (void *)pc, *pc, (void *)pi, *pi);
    return 0;
}
```

```
sizeof(char) = 1, sizeof(int) = 4,
sizeof(void *) = 8
```

```
&i = 0x16f26f81c,
&c = 0x16f26f81b,
&pc = 0x16f26f810,
&pi = 0x16f26f808
```

```
i = 50, c = B,
pc = 0x16f26f81b, *pc = B,
pi = 0x16f26f81c, *pi = 50
```

Que se passe-t-il ?

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int i = 0;
    char c, *pc;
    int *pi;

    printf("sizeof(char) = %lu, sizeof(int) = %lu, \nsizeof(void
        *) = %lu\n\n", sizeof (char), sizeof (int), sizeof (void
        *));
    printf("&i = %p, \n&c = %p, \n&pc = %p, \n&pi = %p\n\n", (void
        *)&i, (void *)&c, (void *)&pc, (void *)&pi);

    c = 'a';
    pi = calloc(1, sizeof(int));
    pc = calloc(1, sizeof(char));
    *pi = 50;
    *pc = 'B';
    printf("i = %d, c = %c, \npc = %p, *pc = %c, \npi = %p, *pi =
        %d\n", i, c, (void *)pc, *pc, (void *)pi, *pi);
    return 0;
}
```

```
sizeof(char) = 1, sizeof(int) = 4,
sizeof(void *) = 8
```

```
&i = 0x16b6af81c,
&c = 0x16b6af81b,
&pc = 0x16b6af810,
&pi = 0x16b6af808
```

```
i = 0, c = a,
pc = 0x12d606740, *pc = B,
pi = 0x12d606730, *pi = 50
```

Passage par référence

En C, le passage des paramètres se fait toujours par valeur

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

```
void swap (int a, int b) {  
    int aux;  
    aux = a;  
    a = b;  
    b = aux;  
}
```

```
int main (int argc, char *argv[]) {  
    int x = 3;  
    int y = 5;  
  
    printf("av : x = %d, y = %d\n", x, y);  
    swap(x, y);  
    printf("ap : x = %d, y = %d\n", x, y);  
  
    return 0;  
}
```

Passage par référence

Les pointeurs permettent de simuler le passage par référence

```
#include <stdio.h>

void swap (int *a, int *b) {
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
}

int main (int argc, char *argv[]) {
    int x = 3;
    int y = 5;

    printf("av : x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("ap : x = %d, y = %d\n", x, y);

    return 0;
}
```


Pointeurs et tableaux

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

void imp (int t[], int lg) {
    int i;
    for (i = 0; i < lg; i++) {
        printf("%d ", t[i]);
    }
    fputc('\n', stdout);
}

void maz (int t[], int lg) {
    int i;
    for (i = 0; i < lg; i++) {
        t[i] = 0;
    }
}

void mess (int t[], int lg) {
    int i;
    t = malloc(sizeof(int) * lg);
    for (i = 0; i < lg; i++) {
        t[i] = 33;
    }
    imp(t, lg);
}
```

```
#define MAX 10

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

    maz(t, MAX);
    imp(t, MAX);

    mess(t, MAX);
    imp(t, MAX);

    return 0;
}
```

Pointeurs et tableaux

- Notions très liées en C
- Le nom du tableau correspond à l'adresse de départ du tableau (en fait l'adresse du premier élément)
Si `int t[10]`; alors `t = &t[0]`
- Si on passe un tableau en paramètre, seule l'adresse du tableau est passée (il n'existe aucun moyen de connaître le nombre d'éléments)
- En fait, l'utilisation des crochets est une simplification d'écriture. La formule suivante est appliquée pour accéder à l'élément `i`
`a[i]` est équivalent à `*(a + i)`

Pointeurs : opérations arithmétique

- Un pointeur contenant une adresse, on peut donc lui appliquer des opérations arithmétiques
 - pointeur + entier donne un pointeur
 - pointeur - entier donne un pointeur
 - pointeur - pointeur donne un entier
- Le dernier point est **FORTEMENT déconseillé** car très peu portable

Pointeurs : opérations arithmétique

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int t1[10], t2[20];

    printf ("t1 = %p, t2 = %p\n", (void *)t1, (void *)t2);
    printf ("t1 + 3 = %p, &t1[3] = %p\n", (void *)(&t1 + 3), (void *)(&t1[3]));
    printf ("&t1[3] - 3 = %p\n", (void *)(&t1[3] - 3));
    printf ("t1 - t2 = %ld\n", t1 - t2);
    printf ("t2 - t1 = %ld\n", t2 - t1);

    return 0;
}
```

```
t1 = 0x16f5fb800, t2 = 0x16f5fb7b0
t1 + 3 = 0x16f5fb80c, &t1[3] = 0x16f5fb80c
&t1[3] - 3 = 0x16f5fb800
t1 - t2 = 20
t2 - t1 = -20
```

Pointeurs : opérations arithmétique

```
#include <stdio.h>

void imp (int t[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf ("%d ", t[i]);
    }
    fputc('\n', stdout);
}

int main (int argc, char *argv[]) {
    int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

    imp(tab, sizeof (tab) / sizeof (tab[0]));
    imp(tab + 5, 5);

    return 0;
}
```

Pointeurs : opérations arithmétique

```
#include <stdio.h>
#include <stdlib.h>
#define M 5

void imp (int t[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", t[i]);
    }
    fputc('\n', stdout);
}

void copie (int t[], int *p, int n) {
    int i;
    for (i = 0; i < n; i++) {
        *p++ = t[i];
    }
}

void saisie (int t[], int n) {
    int i;
    printf("les %d nombre ? ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &t[i]); /* t + i */
    }
}
```

```
int main (int argc, char *argv[]) {
    int *p = calloc(M, sizeof(int));
    int t[M];

    saisie(t, M);
    imp(t, M);
    imp(p, M);

    copie(t, p, M);
    imp(t, M);
    imp(p, M);

    return 0;
}
```

Pointeurs et chaînes de caractères

- Une constante chaîne de caractères a comme valeur l'adresse mémoire de son premier caractère (on ne peut la modifier)
- Son type est donc pointeur sur caractères

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

```
int main (int argc, char *argv[]) {  
    char y[] = "1234";  
    char *x = "1234";  
    char z[10] = "1234";  
    char tc[] = {'1', '2', '3', '4', '\0'};  
  
    printf (" x = %p,  &x = %p\n", (void *)x,  (void *)&x);  
    printf (" y = %p,  &y = %p\n", (void *)y,  (void *)&y);  
    printf (" z = %p,  &z = %p\n", (void *)z,  (void *)&z);  
    printf ("tc = %p, &tc = %p\n", (void *)tc, (void *)&tc);  
  
    return 0;  
}
```

Parcours de chaînes de caractères

```
#include <stdio.h>

void aff (char *s) {
    while (*s) {
        fputc(*s, stdout);
        s++;
    }
    fputc('\n', stdout);
}

int main (int argc, char *argv[]) {
    char s1[] = "bonjour vous !";
    char *s2 = "et vous aussi !";

    aff(s1);
    aff(s2);

    return 0;
}
```


Pointeurs et fonctions

En C, le type pointeur sur fonction existe

- `typedef int (*tpf) (int);`
déclaration de tpf comme étant un type pointeur sur fonction prenant un `int` en paramètre et renvoyant un `int`
- `float (*vpf) (double, char*);`
déclaration de vpf comme étant une variable de type pointeur sur fonction prenant en paramètres un `double` et un `char*` et renvoyant un `float`
- `char *f (int);`
déclaration de f comme étant une constante de type pointeur sur fonction prenant en paramètre un `int` et renvoyant un `char*`

Pointeurs et fonctions

Lorsqu'on définit une fonction, en fait on déclare une constante de type pointeur sur fonction et sa valeur est l'adresse de la première instruction de la fonction

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

```
int max (int a, int b) {  
    return a > b ? a : b;  
}
```

```
int main (int argc, char *argv[]) {  
    printf("%p\n", (void *)printf);  
    printf("%p\n", (void *)&printf);  
    printf("%p\n", (void *)max);  
    printf("%p\n", (void *)&max);  
  
    return 0;  
}
```

Fonctions en paramètre

Il suffit de passer le type du pointeur sur fonction (concerné) en paramètre

```
#include <stdio.h>
#define MAX 10

void imp (int t[], int lg) {
    int i;
    for (i = 0; i < lg; i++) {
        printf("%d ", t[i]);
    }
    fputc('\n', stdout);
}

int maz (int a) {
    return 0;
}

int plus1 (int a) {
    return a + 1;
}
```

```
void modif (int t[], int lg, int (*f) (int)) {
    int i;
    for (i = 0; i < lg; i++) {
        t[i] = f(t[i]);
    }
}

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

    modif(t, MAX, maz);
    imp(t, MAX);

    modif(t, MAX, plus1);
    imp(t, MAX);

    modif(t, MAX, plus1);
    imp(t, MAX);

    return 0;
}
```

Pointeur polymorphe (void*)

```
void swap (int *a, int *b) {  
    int aux;  
    aux = *a;  
    *a = *b;  
    *b = aux;  
}
```

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

```
void swap (void *a, void *b, short taille) {  
    void *aux = malloc(taille);
```

```
    memcpy(aux, a, taille);  
    memcpy(a, b, taille);  
    memcpy(b, aux, taille);
```

```
#if 0  
    /* c'est complètement faux : on déréférence un void * !!! */  
    *aux = *a;  
    *a = *b;  
    *b = *aux;  
#endif  
}
```

Pointeur polymorphe (void*)

```
#include <stdio.h>
#include "swap.h"
#define imp(t, s1, s2, x, s3, y) \
    fprintf(stdout, "%s: %s = " t", %s = " t"\n", s1, s2, x, s3, y)
int main (int argc, char *argv[]) {
    int a = 3, b = 9;
    float x = 13, y = 19;
    double m = 23.45, n = 25.0;
    char c1 = 'a', c2 = 'B';

    imp("%d", "avant", "a", a, "b", b);
    swap(&a, &b, sizeof(int));
    imp("%d", "après", "a", a, "b", b);

    imp("%.2f", "avant", "x", x, "y", y);
    swap(&x, &y, sizeof(float));
    imp("%.2f", "après", "x", x, "y", y);

    imp("%.2f", "avant", "m", m, "n", n);
    swap(&m, &n, sizeof(double));
    imp("%.2f", "après", "m", m, "n", n);

    imp("%c", "avant", "c1", c1, "c2", c2);
    swap(&c1, &c2, sizeof(char));
    imp("%c", "après", "c1", c1, "c2", c2);

    return 0;
}
```

Pointeur polymorphe (void*)

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

void swap (void **a, void **b) {
    void *aux = malloc(sizeof(void *));
    aux = *b;
    *b = *a;
    *a = aux;
}

int main (int argc, char *argv[]) {
    int a = 3, b = 4;
    int *x = &a, *y = &b;
    double *m = malloc(sizeof(double));
    double *n = malloc(sizeof(double));

    fprintf(stdout, "av: a = %d, b = %d, x = %d, y = %d\n", a, b, *x, *y);
    swap(&x, &y);
    fprintf(stdout, "ap: a = %d, b = %d, x = %d, y = %d\n", a, b, *x, *y);

    *m = 3.456;
    *n = 1.2345;
    printf("av: *m = %f, *n = %f\n", *m, *n);
    swap(&m, &n);
    printf("ap: *m = %f, *n = %f\n", *m, *n);

    return 0;
}
```