

Langage C - Pointeurs et mémoire

Système et environnement de programmation

Université Grenoble Alpes

Plan

- 1 Les pointeurs
- 2 Pointeurs et tableaux
- 3 Les chaînes de caractères
- 4 Structures
- 5 Allocation mémoire

Echange de deux entiers

Au dernier cours nous avons vu

```
#include <stdio.h>
void echange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int qd = 42;          int ue = 203;
    echange(qd, ue);
    printf("En %d on parle de %d\n", ue, qd);
    return 0;
}
```

qd et ue ne sont pas échangées par l'appel à la fonction echange

Passage de paramètres

Cela vient du passage de paramètres

- Les paramètres d'une fonction sont passés par valeur (copiées)
- Les variables de la fonctions appelante restent inchangées

Si nous voulons changer des variables de la fonction appelante dans la fonction appelée, il nous faut un moyen

- de désigner l'emplacement en mémoire des variables à changer
- d'accéder à l'emplacement mémoire désigné

Les pointeurs nous apportent tout ça

Adresse d'une variable

Toute variable possède une adresse

- c'est l'emplacement mémoire où elle est stockée
- on l'obtient à l'aide de l'opérateur & (on appelle cela un référencement)

```
int variable=42;
```

```
printf("Variable est un int contenant %d\n"  
      "stocké à l'adresse %p\n",  
      variable, &variable);
```

```
Variable est un int contenant 42  
stocké à l'adresse 0x7ffc334cec04
```

Les pointeurs

Manipulation d'adresse

- Si on veut stocker une adresse dans une variable
- Si on veut passer une adresse en paramètre d'une fonction

On a besoin de types correspondants aux adresses, ces types sont nommés des pointeurs !

Caractéristiques des types pointeur

- Leurs valeurs sont des adresses (emplacements mémoire)
- Ils sont associés au type de l'objet stocké à ces adresses (détermine comment effectuer l'accès à ces emplacements)

Déclaration d'un pointeur

On fait suivre le type de l'objet pointé d'une *

```
int *pointeur;
```

- Le type de pointeur est `int *`
- Le type de l'objet référencé par `pointeur` est `int`
- Le `int` de cet exemple peut être remplacé par tout type valide

On accède à l'objet pointé avec l'opérateur *

- Cela s'appelle un déréférencement
- **ATTENTION** : la notation prête à confusion
 - * est un opérateur unaire préfixé de déréférencement
 - * est un qualificateur de type postfixé
 - * est un opérateur binaire de multiplication

probablement la principale difficulté du C

Exemple

```
#include <stdio.h>
int main() {
    int x = 10;
    int *pointeur;           // type : int *

    pointeur = &x;
    *pointeur = 21;         // déréférencement
    printf("L'entier x contient : %d\n", x);
    x = x*2;                // multiplication
    printf("L'entier pointé contient : %d\n",
           *pointeur);
    return 0;
}
```

L'entier x contient : 21

L'entier pointé contient : 42

Retour sur l'échange

Revenons à notre code qui ne marche pas

```
#include <stdio.h>
void echange(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
    int qd = 42;          int ue = 203;
    echange(qd, ue);
    printf("En %d on parle de %d\n", ue, qd);
    return 0;
}
```

En 203 on parle de 42

Passage de paramètres par adresse

Si le programme principal passe les adresses de `qd` et `ue`

```
#include <stdio.h>
void echange(int *a, int *b) {
    int tmp = *a;    // a contient l'adresse de qd
    *a = *b;        // accès à *a <=> accès à qd
    *b = tmp;
}
int main() {
    int qd = 42;    int ue = 203;
    echange(&qd, &ue); // adresses passées par copie
    printf("En %d on parle de %d\n", ue, qd);
    return 0;
}
```

En 42 on parle de 203

Lecture d'une variable

La lecture d'une variable se fait avec `scanf` qui

- utilise un format similaire à celui de `printf`
- prend comme arguments supplémentaires des adresses de variables, car elle modifie leur valeur

```
#include <stdio.h>
int main() {
    int x;
    printf("Saisissez un entier :\n");
    scanf("%d", &x);
    printf("Entier saisi : %d\n", x);
    return 0;
}
```

Quelques particularités de scanf

Comme pour printf, le premier argument de scanf est son format, il peut contenir

- Des directives pour lire
 - %o, %d, %x un entier en base octale, décimale ou hexadécimale
 - %c un caractère
- Des caractères qui doivent être saisis littéralement
- Un seul espace lit un nombre quelconque de séparateurs

```
int nb;
int Tab[10];

scanf(" %d", &nb);
for (int i=0; i<nb; i++) {
    scanf(" %d", &Tab[i]);
    printf("J'ai lu %d\n", Tab[i]);
}
```

Plan

- 1 Les pointeurs
- 2 Pointeurs et tableaux**
- 3 Les chaînes de caractères
- 4 Structures
- 5 Allocation mémoire

Similarités entre tableaux et pointeurs

Les tableaux sont manipulés par adresse

- Leur nom est similaire à un pointeur sur le premier élément
- L'opérateur [] s'applique à tout pointeur

```
int Tab[] = {1, 2, 3, 5, 7, 9, 11, 13, 17, 0};  
int *pointeur = Tab;
```

```
for (int i=0; pointeur[i] != 0; i++)  
    printf("Tab contient %d en position %d à "  
          "partir du pointeur\n", pointeur[i], i);
```

```
printf("Je déplace le pointeur\n");  
pointeur = &Tab[3];
```

```
for (int i=0; pointeur[i] != 0; i++)  
    printf("Tab contient %d en position %d à "  
          "partir du pointeur\n", pointeur[i], i);
```

Différences

- Le nom d'un tableau est un pointeur constant
- On ne peut pas changer son adresse (celle du premier élément)
- Déclarer un tableau alloué de la mémoire pour les éléments
- Déclarer un pointeur n'alloue rien pour les éléments pointés

```
int Tab[] = {1, 2, 3, 5, 7, 9, 11, 13, 17, 0};  
int *pointeur; // Non initialisé  
  
Tab = &Tab[4]; // Ne compile pas, Tab constant  
Tab[4] = 42; // Ok  
pointeur[4] = 42; // Compile, accès mémoire invalide
```

Tableaux en paramètres

Revenons sur notre exemple correct du cours précédent

```
#include <stdio.h>
// Ici, on passe l'adresse du premier élément
void echange(int Tab[], int i, int j) {
    int tmp;
    tmp = Tab[i];
    Tab[i] = Tab[j];
    Tab[j] = tmp;
}
int main() {
    // Initialisation de taille et valeurs
    int Tab[] = {42, 203};
    echange(Tab, 0, 1);
    printf("En %d on parle de %d\n", Tab[0], Tab[1]);
    return 0;
}
```

Tableaux en paramètres

On peut aussi l'écrire avec un pointeur

```
#include <stdio.h>
// Ici, on passe l'adresse du premier élément
void echange(int *Tab, int i, int j) {
    int tmp;
    tmp = Tab[i];
    Tab[i] = Tab[j];
    Tab[j] = tmp;
}
int main() {
    // Initialisation de taille et valeurs
    int Tab[] = {42, 203};
    echange(Tab, 0, 1);
    printf("En %d on parle de %d\n", Tab[0], Tab[1]);
    return 0;
}
```

Usages

Dans les paramètres formels d'une fonction

- Passer un pointeur ou un tableau est similaire (adresse)
- Un pointeur peut être modifié
- Pour cette flexibilité, le passage d'un pointeur est préféré

On écrira donc

```
void echange(int *Tab, int i, int j) {
```

plutôt que

```
void echange(int Tab[], int i, int j) {
```

Plan

- 1 Les pointeurs
- 2 Pointeurs et tableaux
- 3 Les chaînes de caractères**
- 4 Structures
- 5 Allocation mémoire

Définition

Une chaîne de caractères est un tableau de char contenant le caractère '\0' (fin de chaîne)

```
char bonjour [10];
```

```
bonjour [0] = 'S';
```

```
bonjour [1] = 'a';
```

```
bonjour [2] = 'l';
```

```
bonjour [3] = 'u';
```

```
bonjour [4] = 't';
```

```
bonjour [5] = '\0';
```

```
printf("Voilà ma chaîne : %s\n", bonjour);
```

Représentation compatible avec beaucoup de fonctions du système (ici, printf avec le format %s)

Déclaration

On peut initialiser une chaîne de caractères à la déclaration

```
char bonjour[] = "Salut";
```

Cela reste un tableau, on peut accéder à chaque caractère

```
printf("Premier caractère : %c\n", bonjour[0]);
```

Préciser la taille

On peut aussi préciser explicitement la taille du tableau à la création de la variable

```
char bonjour [20] = "Salut";
```

Si la taille du tableau est insuffisante pour la chaîne stockée on obtient un **warning**

```
exemple_chaine.c:4:20: warning:  
  initializer-string for char array is too long  
    char bonjour[3] = "Salut";  
                    ^~~~~~
```

Débordement

Les chaînes de caractères sont des tableaux

- On peut accéder à une case en dehors de la chaîne.
- Comme pour tous les tableaux, c'est un bug !

`scanf` ne connaît pas la taille du tableau pour le format `%s`

⇒ faille classique, permet une attaque par *buffer overflow*

- remplit la mémoire de données invalides / code malicieux
- dangereux si le programme vulnérable a des privilèges
- faille dans Microsoft IIS 5.0 et Microsoft SQL Server 2000
- cracks de la Xbox, PS2 et Wii

Solution : utiliser `fgets` (pas au programme)

Taille de la chaîne stockée

Une chaîne de caractères contient le caractère '\0' (fin de chaîne)

- Le tableau qui la contient peut être plus grand
- Pour déterminer la taille d'une chaîne on cherche ce '\0'

```
int longueur_chaine(char *s) {
    int i=0;
    while (s[i] != '\0') {
        i++; // i=i+1
    }
    return i;
}

int main() {
    char nom[20] = "Toto";
    printf("La longueur de ma chaîne est %d",
           longueur_chaine(nom));
    return 0;
}
```

Exercice

Ecrire la fonction C suivante qui convertit en majuscules un mot

```
void capitalizeWord(char *word);
```

Plan

- 1 Les pointeurs
- 2 Pointeurs et tableaux
- 3 Les chaînes de caractères
- 4 Structures**
- 5 Allocation mémoire

Type structuré

On peut créer un nouveau type en regroupant d'autres types

- = définition d'objets complexes à partir d'objets plus basiques
- Exemples
 - Des coordonnées cartésiennes avec un couple de rationnels
 - Un instant avec 3 entiers (heures, minutes, secondes)
 - Une date avec deux entiers et une chaîne (jours, mois, année)

```
struct instant {  
    int heures;  
    int minutes;  
    int secondes;  
};
```

Ne fait que créer un nouveau type structuré (pas de variable)

Utilisation

Un type structuré est passé par valeur en paramètre effectif d'une fonction et s'utilise généralement comme un autre type

- Déclaration de variable (structure)

```
struct instant maintenant;
```

- Affectation

```
struct instant une_heure_qd = {1, 42, 0};  
maintenant = une_heure_qd; // Copie complète
```

- Pointeur

```
struct instant *pointeur;  
pointeur = &maintenant;
```

Accès aux champs

- L'opérateur `.` permet d'accéder aux champs d'une structure

```
maintenant.heures = 4;  
maintenant.minutes = 2;  
maintenant.secondes = 42;
```

- Attention à la précedence des opérateurs

```
struct instant *pointeur;  
pointeur = &maintenant;
```

```
(*pointeur).minutes = 0;  
(*pointeur).secondes++;
```

- Sucre syntaxique ($a \rightarrow b \iff (*a).b$)

```
pointeur->minutes = 0;  
pointeur->secondes++;
```

Exemple

```
#include <stdio.h>

struct Prof {
    char nom[42];
    int age;
    float taille;
};

int main() {
    struct Prof g = { "Guillaume", 42, 1.42 };

    printf("Bonjour, je m'appelle %s, je mesure "
           "%gm, l'an prochain j'aurai %d ans\n",
           g.nom, g.taille, g.age);
    return 0;
}
```

Simplification et abstraction avec typedef

typedef permet de donner un nouveau nom à un type existant

- utilisation : `typedef <type> <nom>;`
- exemples

```
typedef unsigned long long naturel; // plus court
typedef int *mon_pointeur;         // plus simple
typedef struct Prof professeur;    // detail masqué
```

```
naturel i = 8000000000;
professeur huard = { "Guillaume", 42, 1.42 };
mon_pointeur p = &huard.age;
```

Souvent utilisé pour masquer le détail d'une structure de données (abstraction)

typedef et structures

On peut combiner un typedef avec une définition de structure

```
typedef struct Prof {  
    char nom[42];  
    int age;  
    float taille;  
} professeur;
```

équivalent à

```
struct Prof {  
    char nom[42];  
    int age;  
    float taille;  
};  
typedef struct Prof professeur;
```

Exemple d'appel de fonction modifiant un struct

```
typedef struct Prof professeur;

void anniversaire(professeur *p) {
    p->age++;
}

int main() {
    professeur g = { "Guillaume", 42, 1.42, 84 };

    printf("%s a %d ans.\n", g.nom, g.age);

    anniversaire(&g);

    printf("L'an prochain il aura %d ans.\n", g.age);
    return 0;
}
```

Plan

- 1 Les pointeurs
- 2 Pointeurs et tableaux
- 3 Les chaînes de caractères
- 4 Structures
- 5 Allocation mémoire**

Allocation mémoire, exemple

Lorsqu'on ne connaît pas la taille d'un tableau

- on souhaite stocker des valeurs saisies par l'utilisateur
- on ne sait pas combien, on compte demander à l'utilisateur
- on aimerait pouvoir choisir la taille d'un tableau à l'exécution

Impossible avec un tableau mais avec un pointeur et un peu d'aide du système. . .

malloc, free

Deux fonctions permettent de gérer de la mémoire fournie par le système durant l'exécution

- `malloc` : renvoie l'adresse d'un nouveau bloc mémoire, ayant pour taille le nombre d'octets passé en paramètre, ou `NULL` en cas d'échec (mémoire disponible insuffisante)
- `free` : libère un bloc alloué par `malloc`

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

Souvent utilisé en conjonction avec l'opérateur `sizeof`

- applicable à une variable ou un type
- renvoie la taille en octet de l'objet passé

Exemple

```
int taille; double *Tab;

printf("Combien de valeurs désirez vous ?\n");
scanf(" %d", &taille);

Tab = malloc(taille * sizeof(double));
if (Tab == NULL) {
    printf("Erreur d'allocation mémoire\n");
    return 1;
}
for (int i=0; i<taille; i++)
    scanf(" %lg", &Tab[i]);
printf("Voici votre tableau : \n");
for (int i=0; i<taille; i++)
    printf("%lg\n", Tab[i]);

free(Tab);
```

Fuites mémoire

TOUJOURS libérer la mémoire allouée par `malloc` lorsqu'elle n'est plus utile

Un programme ne le faisant pas contient des fuites mémoire

- mémoire monopolisée par le programme mais non utilisée
- perdure jusqu'à la fin de l'exécution

Bug très très commun

- ne fait rien planter mais alourdit le système
- "difficile" à détecter (outil `valgrind` peut aider)