SE2A3 - Programmation avancée

CM1 - Pointeurs et Structures

Thibault Liétard
Polytech' Lille

2024

Remerciements

Ce cours est largement inspiré du travail de Walter Rudametkin.

Organisation

15 séances

- 13x4h et 2x2h
- 9 TPs + 3TDs + mini-projet
- Partie théoriques + autoformation

Évaluation

- CC + Projet (50/50)
- Contrôle continu : 4 TPs notés, 3 retenus
- Projet : à réaliser en binômes

Types de données et pointeurs

Rappel

Type	Min	Min form.	Max	Max formule
char	-128	-2^7	+127	$2^7 - 1$
unsigned char	0	0	+255	$2^8 - 1$
short	-32 768	-2^{15}	+32 767	$2^{15} - 1$
unsigned short	0	0	+65 535	$2^{16} - 1$
int (16 bit)	-32 768	-2^{15}	+32 767	$2^{15} - 1$
unsigned int	0	0	+65 535	$2^{16} - 1$
int (32 bit)	-2 147 483 648	-2^{31}	+2 147 483 647	$2^{31} - 1$
unsigned int	0	0	+4 294 967 295	$2^{32} - 1$
long (32 bit)	-2 147 483 648	-2^{31}	+2 147 483 647	$2^{31} - 1$
unsigned long	0	0	+4 294 967 295	$2^{32} - 1$
long (64 bit)	-9.22337×10^{18}	-2^{63}	$+9.22337 \times 10^{18}$	$2^{63} - 1$
unsig. long long	0	0	$+1.844674 \times 10^{19}$	$2^{64} - 1$
long long	-9.22337×10^{18}	-2^{63}	$+9.22337 \times 10^{18}$	$2^{63} - 1$
unsig. long long	0	0	$+1.844674 \times 10^{19}$	$2^{64} - 1$

Espace mémoire

```c
#include <stdio.h>

int main() {
        printf("size of data types in bytes\n");
        printf("char:        %zu\n",sizeof(char));
        printf("short:       %lu\n",sizeof(short));
        printf("int:         %lu\n",sizeof(int));
        printf("long int:    %lu\n",sizeof(long int));
        printf("float:       %lu\n",sizeof(float));
        printf("double:      %lu\n",sizeof(double));
        printf("long double: %lu\n",sizeof(long double));
        printf("void:        %lu\n",sizeof(void));

        printf("\nsize of pointers in bytes\n");
        printf("char *:        %lu\n",sizeof(char *));
        printf("short *:       %lu\n",sizeof(short *));
        printf("int *:         %lu\n",sizeof(int *));
        printf("long int *:    %lu\n",sizeof(long int *));
        printf("float *:       %lu\n",sizeof(float *));
        printf("double *:      %lu\n",sizeof(double *));
        printf("long double *: %lu\n",sizeof(long double *));
        printf("void *:        %lu\n",sizeof(void *));

        return 0;
}
```

size_ofs.c

Espace mémoire

```
size of data types in bytes
char:          1
short:         2
int:           4
long int:      8
float:         4
double:        8
long double:  16
void:          1

size of pointers in bytes
char *:          8
short *:         8
int *:           8
long int *:      8
float *:         8
double *:        8
long double *:   8
void *:          8
```

Sur les pointeurs

```c
#include <stdio.h>

int main() {
        int m,n,k;
        int *p1,*p2,*p3;

        m=22; n=33;
        p1=&m; p2=&n;
        printf("%d %d %d %d\n",*p1,*p2,m,n);

        p3=p1; p1=p2; p2=p3;
        printf("%d %d %d %d\n",*p1,*p2,m,n);

        k=*p1; *p1=*p2; *p2=k;
        printf("%d %d %d %d\n",*p1,*p2,m,n);

        printf("\nPointer addresses\n");
        printf("%p %p %p %p\n",p1,p2,&m,&n);
        printf("%p %p %p %p\n",&p1,&p2,m,n);

        return 0;
}
```

Cas du malloc

```c
void main() {
  int*    x;  // Alloue les pointeurs en mémoire
  int*    y;  // (mais pas les valeurs pointés)

  x = malloc(sizeof(int));
      // Alloue un entier (valeur pointé),
      // et fait pointer x sur cette espace

  *x = 42; // Donne la valeur de 42 à l'espace pointé par x
           // (déréférencer x)

  *y = 13; // ERREUR (SEGFAULT)
           // il n'y a pas d'espace pointé en mémoire

  y = x; // Fait pointer y sur le même espace mémoire que x

  *y = 13; // Déréférence y et assigne 13
           // (espace pointé par x et y)
  free(x); // Libère l'espace alloué
}
```

Les tableaux

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
    //déclaré avec la taille
    int tab[10];

    //déclaré puis allocation dynamique
    int* tab2;
    int n;
    scanf("%d", &n);
    tab2 = (int*) malloc(n*sizeof(int));
}
```

Principe des tableaux

Allocation
Lorsque le tableau est alloué, un espace mémoire est réservé en prenant en compte le **nombre d'éléments** et le **type**.

Pointeur
Dans l'exemple précédent, *tab* et *tab2* sont de type pointeur sur un entier. Il désignent l'adresse de la première case du tableau.

Opération sur les pointeurs
tab[i] $\equiv$ *(tab+i)

tab++ //décale le pointeur d'une "case"

String
Les chaînes de caractères sont des tableaux de caractères.

Exercice d'application

Soit le code suivant

```c
int main(){
    char *argv []={
        "Samuel Beckett",
        "Jean-Baptiste Poquelin",
        "Edmond Rostand",
        "Yasmina Reza",
    };
    char **p = argv;
}
```

Donner la valeur de :

1.	2.
(*p++)[2]	*++*p
(*++p)[2]	*(*++p)
*p[1]++	*p[2]++
*++p[1]	*(++p[2])

individuellement et successivement.

Structures

Définition

Concept
C'est un n-uplet d'informations de types potentiellement divers rangées dans des variables appelées "champs".

Algorithmique

```
01 - type <st> = structure
02 -     champs1 : <t1>
03 -     champs2 : <t2>
04 -     champs3 : <t3>
05 -     ...
06 -     champsn : <tn>
07 - fin
```

Notation en C

Pour déclarer une structure en C, on utilise le mot clé **struct** suivi du désignateur (nom) de la structure.

```
1 struct complexe {
2     float reel;
3     float imag;
4 }
5
6 struct complexe c;
```

Variante avec typedef

On peut définir la structure comme un type (on parle d'alias de type) en utilisant le mot clé **typedef**. Dans ce cas, le désignateur de la structure est superflu, mais il faut donner un nom au type.

```
1 typedef struct {
2     float reel;
3     float imag;
4 } complexe;
5
6 int main(){
7     complexe c;
8 }
```

Structure imbriquées

Un champs d'une structure peut parfaitement être défini par une autre structure ou un pointeur vers une structure.

```
1 typedef struct {
2     float reel;
3     float imag;
4 } complexe;
5
6 typedef struct {
7     char lettre;
8     complexe coord;
9 } point;
```

Accession aux champs

L'opérateur '.' permet d'accéder aux champs d'une structure. C'est l'opérateur de plus forte priorité en C.

```
11 int main(){
12     point p;
13     p.lettre = 'A';
14     p.coord.reel = 2;
15     p.coord.imag = 4;
16 }
```

Cas des champs pointeurs

Si une structure possède un champs de type pointeur sur quelque chose, alors il faudra allouer manuellement l'espace mémoire de ce champs après la déclaration avant de l'utiliser. On utiliser pour cela le **malloc**.

```c
#include<stdlib.h>
typedef struct {
    int* a;
} st;

int main(){
    st x;
    x.a = (int*) malloc(sizeof(int));
    *x.a = 2;
}
```

Cas des structures comme pointeurs

Si une structure est déclarée via un pointeur, alors l'opérateur '**->**' permet d'y accéder en "dépointantën même temps. Il ne faut pas oublier d'allouer manuellement l'espace de la structure.

```
 1 #include<stdlib.h>
 2 typedef struct {
 3     int a;
 4 } st;
 5
 6 int main(){
 7     st* x;
 8     x = (st*) malloc(sizeof(st));
 9     x->a = 2;
10 }
```

Limitations des structures

- Pas de comparaisons (==, !=, >, <, ...)
- Pas d'opérateurs arithmétiques
- Pas de E/S (scanf, printf, ...)
- Pas de support de "deep copy"(pas de copie des valeurs "pointées", seulement les valeurs des pointeurs)
- Attention aux passage des structures dans des fonctions (passage-par-copie des structs, implique SShallow Copy")

Utilisations possibles

- Paramètre de fonction
- retour de fonction
- tableau de structure
- types imbriqués
- ...

This is it ☺