Programmation impérative en C

Cours 4. Gestion de la mémoire et pointeurs

Olivier Baldellon

Courriel : prénom.nom@univ-cotedazur.fr

Page professionnelle : https://upinfo.univ-cotedazur.fr/~obaldellon/

Licence 2 — Faculté des sciences et ingénierie de Nice — Université Côte d'Azur

Sommaire

- Il y aura un QCM en lieu et place de l'amphi à la rentrée.
- sur papier dans l'amphi
- Programme : chapitres 1 à 4

EDT légèrement modifié

- Suite à l'absence de certain intervenant, un groupe de TP aura cours à 15h15 au lieu de 13h15.
- Merci de bien vérifier l'EDT chaque semaine

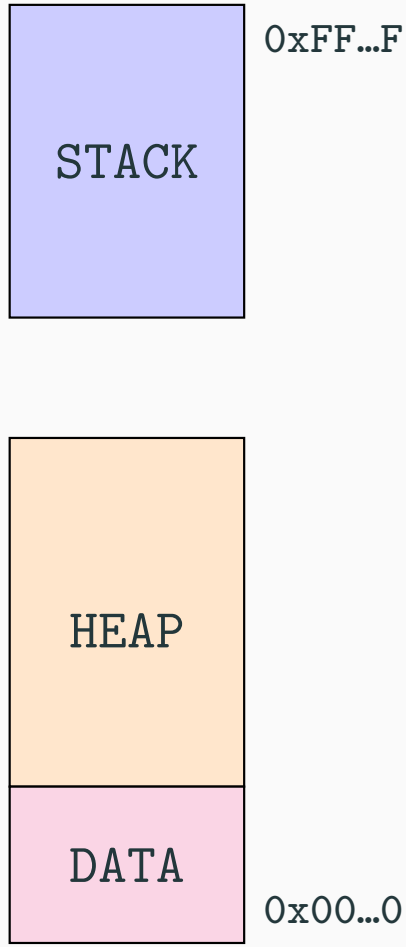
Sommaire

Les trois parties de la mémoire

Chaque exécutable se voit attribuer une mémoire virtuelle de 2^{64} octets.

- 16 exbioctet adressables ($\approx$ 16 exaoctets).
- En pratique, seules certaines de ces adresses sont valides

- La mémoire est partagée en trois.
  - `DATA` : les données
    - contient les instructions du programme
    - contient les variables statiques
    - son organisation est statique
  - `STACK` : la pile
    - contient les variables locales...
    - ... dont la taille est connue à la compilation.
  - `HEAP` : le tas
    - contient les variables créés dynamiquement
    - géré par le développeur.



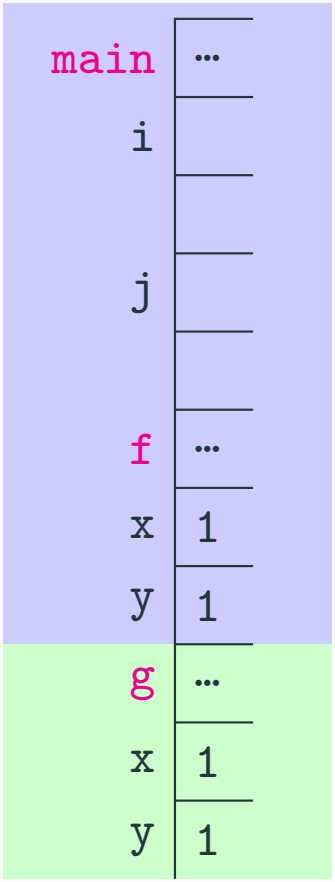
- En pratique, sur 64 bits, les adresses ne vont pas de 0 à $2^{64} - 1$.
  - C'est le système d'exploitation qui décide (Linux, Windows, etc.)
- La mémoire du programme est stockée sur la `RAM`, mais parfois peut-être stockée sur le disque dur s'il n'y a plus de place.
  - Partition `swap` sous Linux : le programme rame alors méchamment !
- Pour déclarer une variable statique, il suffit d'utiliser le mot-clé `static`
  - La variable ne sera pas supprimée à la fin de l'appel de la fonction.
  - Elle est déclarée à `0`.
  - À chaque appel de la fonction, la variable `c` est incrémentée.

*.c

```c
void compteur(void) {
  static int c = 0;
  printf("C = %d\n",c);
  c++;
}
```

Fonctionnement de la pile

- La pile est remplie du haut vers le bas.
  - chaque fonction a accès à un morceau de pile
  - dans lequel sont stockées ses variables
  - on parle de *stack frame*
- Lors de l'appel d'une fonction
  - on empile une nouvelle *frame*
  - pour y stocker les variables de cette fonction
- On recommence ainsi à chaque appel.
  - si le nombre d'appel est trop grand : plantage
  - la taille de la pile est fixée par l'OS (Linux)
  - débordement de pile (*stack overflow*)
- Lorsqu'une fonction retourne
  - on dépile la *frame*
  - la fonction appelante peut continuer son exécution.



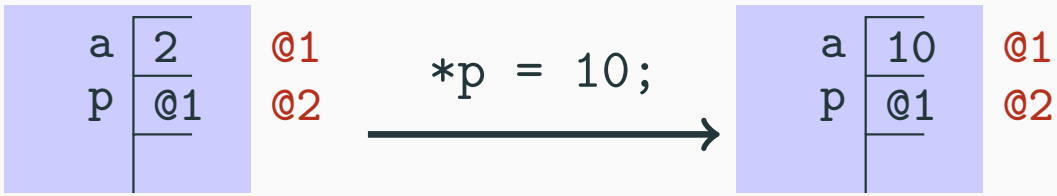
- La gestion de la pile est automatique
  - pas de bug !
  - pas de fuite mémoire
  - pas de gestion mémoire
  - que du bonheur ♥
- Mais la taille de toutes les données doivent être connues à la compilation.
  - On ne peut pas modifier la taille des tableaux
  - On ne peut pas créer de nouvelles données à la volée…
  - …mais uniquement modifier celles déclarées à la compilation.
- Pour dépasser ces limitations on utilise le tas :
  - partie 3

Sommaire

- Un pointeur est un type de donnée représentant une adresse mémoire.

*.c
```c
int a = 2;
int *p;

/* On met l'adresse de a dans p */
p = &a;

printf("a = %d\n",a); /* affiche « a = 2 » */

/* À l'adresse p on écrit 10*/
*p = 10;

printf("a = %d\n",a); /* affiche « a = 10 » */
```

- Dans le code précédant la ligne `*p = 10;`
  - ne modifie pas le pointeur `p` (l'adresse ne change pas)
  - mais modifie la variable `a`



- La notation `&a` représente un pointeur mais `*p` représente une donnée.

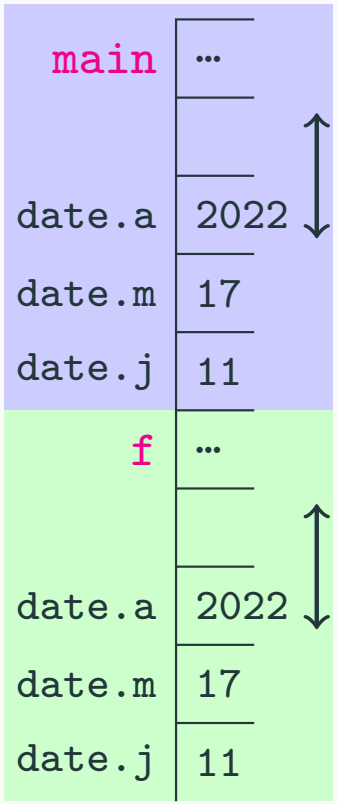
Adresse	Donnée
`&a`	`a`
`p`	`*p`

- Déclaration d'un pointeur vers un entier : `int *p`
  - se lit « p est un pointeur vers `int` »
- Évitez la notation `int* p`
  - syntaxiquement `int*` n'est pas un type.
  - « `int* p1, p2` » est lu par le compilateur « `int *p1, p2` »
  - `p1` est un pointeur mais `p2` un entier !

- Lorsqu'une valeur est passée en argument,
  - elle est copiée dans la nouvelle pile.
  - on parle de passage par valeur.
  - Pour les grosses structures, l'opération peut être coûteuse.

*.c

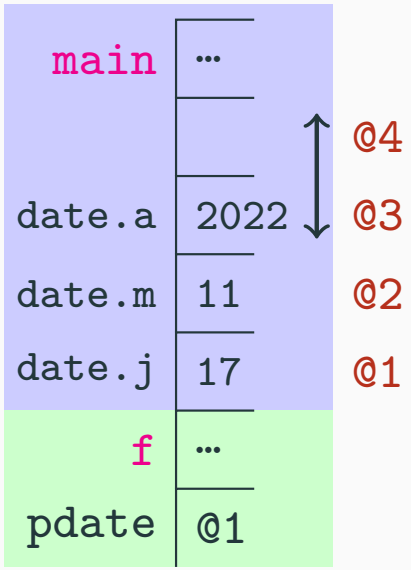
```c
typedef struct {
  unsigned char j, m;
  unsigned short a;
} date_type;

void f(date_type date) {
  …
}

int main(void) {
  date_type date = {17,11,2022};
  f(date);
  return 0;
}
```



- Ici, on se contente de donner le pointeur
  - on parle de passage par paramètre
  - généralement bien plus efficace.
  - Attention, la fonction `f` peut modifier `date` à sa guise

*.c

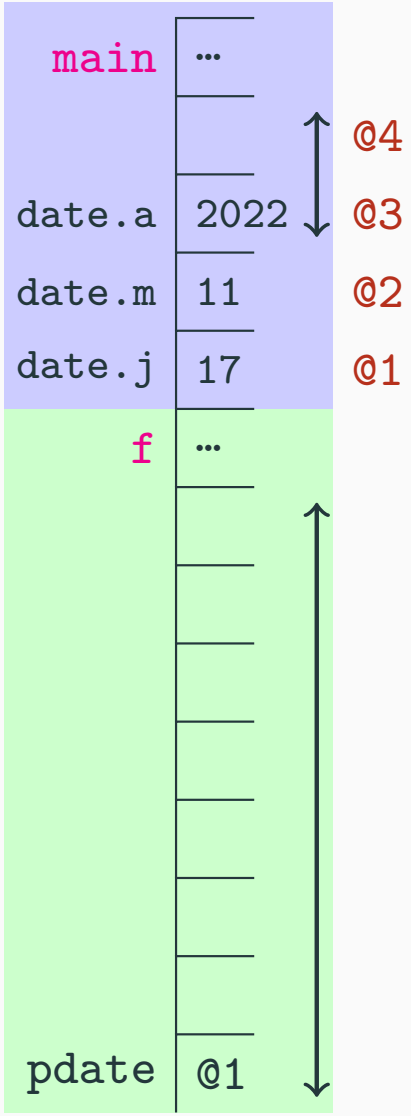
```c
typedef struct {
  unsigned char j, m;
  unsigned short a;
} date_type;

void f(date_type *pdate) {
  …
}

int main(void) {
  date_type date = {17,11,2022};
  f(&date);
  return 0;
}
```



Passage par référence (honnête)

- Dans l'exemple précédent
  - le pointeur faisait 64 bits (8 octets)
  - le gain est ici limité :)
- Mais souvent, la taille du pointeur est très inférieure à la taille des données.
  - sauf les types de base
  - (entiers, caractères et flottants)
- Autre intérêt : pouvoir modifier la variable directement.

```c
int i;
code_erreur = scanf("%d", &i);
```
*.c



Les pointeurs sont des tableaux !

- On peut utiliser les pointeurs pour modifier une unique valeur.
- On utilise aussi les pointeurs pour travailler sur les tableaux :
  - L'adresse correspond à celle de la première case.
  - La longueur du tableau doit être connue par le développeur.
- Il n'y a (presque) pas de différences en C entre tableaux et pointeurs.

*.c
```c
int i = 7;
int *p = &i;
printf("%d == %d\n", *p, p[0]); /* 7 == 7 */
```

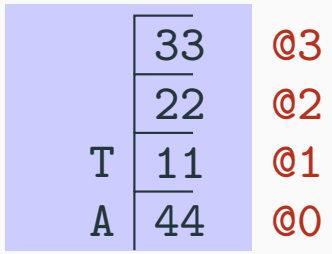
- Ici `T: [11,22,33]` et `p: [22,33]`

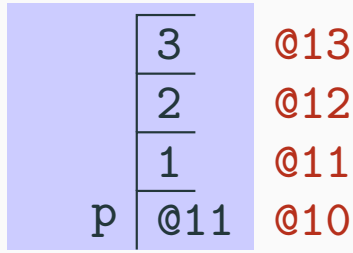
*.c
```c
int *p ;
int T[3] = {11,22,33};
p = T+1; /* p correspond à l'adresse après T */
for (i=0; i<3; i++) printf("%d ",T[i]); /* 11 22 33 */
for (i=0; i<2; i++) printf("%d ",p[i]); /* 22 33 */
```

- Les quatres notations suivantes sont équivalentes :
  - `tab[3] = *(tab + 3) = *(3 + tab) = 3[tab]`

Les pointeurs ne sont pas des tableaux !

- Dans une expression, tableau et pointeur correspondent à des adresses.
  - La valeur d'une variable est le contenu de sa case.
    - `A` vaut `44` (type entier) et `p` vaut `@11` (type adresse)
  - SAUF pour les tableaux : la valeur de `T` est son adresse.
    - `T` vaut `@1`. Mais cette valeur ne se trouve dans aucune case mémoire.
    - `T` et `&T` ont la même valeur.
  - Une variable est associée à une case mémoire
    - cette association ne change jamais.
    - Pour cette raison, l'association entre `T` et `@1` est immuable.
  - Par contre on peut modifier le contenu d'une case mémoire.
    - `p = p+1;` est valide et remplace dans la case `p` la valeur `@11` par `@12`.
    - `T = …` n'a par contre aucun sens. On ne peut pas modifier l'adresse !





Liste des différences entre tableaux et pointeurs

- Déclaration et allocation
  - Tableau : alloué sur la pile. (Les pointeurs pointent souvent vers le tas)
    - `int tab[];` le contenu du tableau est alloué sur la pile ;
    - `char tab[];` idem ;
    - `char *s = ":)";` est allouée dans la partie statique de la mémoire.
    - `int *s;` pas d'allocation (sauf la case pour stocker l'adresse).
  - On peut initialiser le contenu d'un tableau à la déclaration
    - et non celui d'un pointeur (sauf chaîne statique)
    - `int tab[] = {1, 2, 3};` est valide
- On peut modifier l'adresse d'un pointeur (mais pas un tableau)
  - `p = …;` est valide
  - On peut modifier le contenu du tableau `T[i]`, mais pas la valeur de `T`.
- La fonction `sizeof`
  - Pour un tableau, renvoie la taille complète de tous les éléments.
  - Pour un pointeur, affiche la taille d'un pointeur

- Dans une expression, tableau et pointeurs sont tous deux des adresses.
- Les deux signatures suivantes sont strictement équivalentes.
  - `void ma_fonction(int tab[])`
  - `void ma_fonction(int *tab)`
  - dans les deux cas `tab` est un pointeur (et non un tableau)

*.c

```c
void ma_fonction1(int tab[]) {
  printf("%ld", sizeof(tab)); /* tab est un pointeur */
}

void ma_fonction2(int *tab) {
  printf("%ld", sizeof(tab));  /* tab est un pointeur */
}

int main() {
  int tab[] = {1, 2, 3}; /* sizeof(int) = 4*/
  int * p;
  printf("%d", sizeof(tab)); /* affiche 12 (= 3*4)     */
  printf("%d", sizeof(p));  /*  affiche 8 | 8 octets  */
  ma_fonction1(tab);        /*   affiche 8 | valent   */
  ma_fonction2(tab);       /*    affiche 8 | 64 bits */
}
```

- Pour un pointeur de structure la notation `(*pointeur).champs` peut être remplacée par la notation plus agréable `pointeur->champs`.

*.c

```c
typedef struct { unsigned char r, g, b; } couleur;


void afficher(couleur *coul) {
  printf("(%d,%d,%d)\n", (*coul).r, (*coul).g, (*coul).b);
}

void assombrir(couleur *coul) {
  coul->r = coul->r / 2;
  coul->g = coul->g / 2; /* équivalent à (*coul).g = (*coul).g/2 */
  coul->b = coul->b / 2;
}

int main() {
  couleur rougeatre =  {255, 100, 100};
  afficher(&rougeatre);  /* (255,100,100)  */
  assombrir(&rougeatre);
  afficher(&rougeatre);  /* (127,50,50)  */
  return 0;
}
```

- Que se passe-t-il si on ajoute un entier à un pointeur ?
  - pour afficher on utilise `%p` dans le `printf`
  - il faut avant le convertir en pointeur non typé : `(void *)`

*.c
```
int  *p0, *q0;
char *p1, *q1;
q0 = p0+1;
q1 = p1+1;
printf("%p %p\n", (void *) p0, (void *) q0);
printf("%p %p\n", (void *) p1, (void *) q1);
```

- Lorsqu'on incrémente un pointeur de type `truc *`, les calculs se font selon un pas de `sizeof(truc)`.

  - pour `p0` et `q0` : `0x7ffd4586e1b0 0x7ffd4586e1b4`
    - La différence observée est de 4 octets (taille d'un `int`)
  - pour `p1` et `q1` : `0x7ffd4586e1af 0x7ffd4586e1b0`
    - La différence observée est de 1 octet (taille d'un `char`)

- si `truc *p` pointe sur une case d'un tableau de `truc`, `p++` fera pointer `p` sur la case suivante (en dehors d'un tableau : *UB*).

Sommaire

- Travailler sur la pile est très efficace, mais ne permet pas de créer :
  - des objets dont la taille n'est connue qu'à l'exécution
    - Si un utilisateur renseigne son nom, quel en sera la longueur ?
  - des objets dont l'existence n'est connue qu'à l'exécution
    - Si plusieurs joueurs se connectent, combien seront-ils ?
  - des objets dont la taille change.
    - La liste des joueurs a une taille qui peut changer durant le jeu.
- On peut toujours réserver beaucoup de mémoire sur la pile…
  - en espérant que cela suffise.
  - en gâchant beaucoup de mémoire pour rien.
  - bref, la pile n'est pas la réponse à tout.

*.c

```c
#include <stdio.h>
#include <stdlib.h> /* Pour utiliser malloc et free*/

int * intervalle(int a, int b) { /* b exclue */
  int i;
  int n = b-a; /* Nombre d'éléments dans le tableau */
  int *tab = malloc( n * sizeof(int));

  for (i=0; i<n; i++) {
    tab[i] = a+i;
  }
  return tab;
}

int main() {
  int i, *tableau;
  /* On crée tableau dynamiquement*/
  tableau = intervalle(12,15);

  for (i=0; i<3; i++) {
    printf("%d ",tableau[i]); /* affiche 12 13 14  */
  }
  printf("\n");
  /* On n'a plus besoin de tableau : on libére l'espace associé */
  free(tableau);
  return 0;
}
```

- Création

```c
int *tab = malloc( n * sizeof(int));
```
*.c

  - On réserve `n` cases de taille `sizeof(int)`
  - `tab` est un pointeur vers le premier entier.
    - mais en C, un tableau est un pointeur.
  - Les cases de `tab[0]` à `tab[n-1]` sont bien réservées.
    - Ce sera à nous de bien faire attention de ne pas travailler avec des indices invalides.

- Destruction

```c
free(tableau);
```
*.c

  - L'espace sera libérée
  - Utilisable pour une future variable

- En C, pour gérer la mémoire, on passe par les fonctions suivantes
  - `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 d'agrandir la zone mémoire
    - renvoie un pointeur vers la nouvelle zone allouée.
    - si ré-allocation : libère l'ancien pointeur après avoir copié les données.
  - `void free (void *ptr);`
    - permet de libérer la mémoire attribuée à `ptr`.
- Ces fonctions se trouvent dans le fichier de déclarations `stdlib.h`

Sommaire

- C'est tout simple en fait ! Mais il y a de nombreux risques.
  - Fuites mémoires.
    - On oublie de libérer la mémoire
    - Le programme consomme de plus en plus de mémoire RAM
  - Lecture et écriture sur une zone mémoire non valide.
    - Écriture dans un tableau avec des indices trop grands.
    - *Use after free.*
- En cas d'écriture sur une zone non allouée ou non permise :
    - `Erreur de segmentation` ou `segmentation fault`
    - Aucune information supplémentaire (ligne concernée ?)
    - Bon courage pour corriger le problème !

- Attention, ne jamais renvoyer de pointeur appartenant à sa pile !

*.c

```c
#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(void) {
  int *p1 = f1(); /* p1 invalide */
  int *p2 = f2();
  return 0;
}
```

Améliorer un makefile

- On peut ajouter des variables à un Makefile.
  - Comme en *shell*, elle s'utilisent avec un dollar.

Makefile

```
CC = gcc
OPTIONS = -Wall -ansi -pedantic

all: exercice1 exercice2

exercice1: exercice1.c
    $(CC) $(OPTIONS) exercice1.c -o exercice1

exercice2: exercice2.c
    $(CC) $(OPTIONS) exercice2.c -o exercice3
```

- La souris c'est le mal !
- Passer vos fenêtres en pleine écran (ou moitié d'écran : éditeur + sujet)
- Tester vos fonctions dans une autre fenêtre (ne fermez pas l'éditeur !)
- `Alt-<tab>` permet de passer d'une fenêtre à l'autre
- Sous GNU/Emacs
  - `C-c C-c` compile
  - `C-x `` va à la première erreur
  - `C-x C-j` va à la liste des fichiers
  - `F9` alterne entre fichier header et source (cf plus tard)
- Vous n'aimez pas GNU/Emacs ?
  - Personne n'est parfait
  - Apprennez à maîtriser votre éditeur

Merci pour votre attention
Questions

?

Cours 4 — Gestion de la mémoire et pointeurs