4: Mémoire dynamique

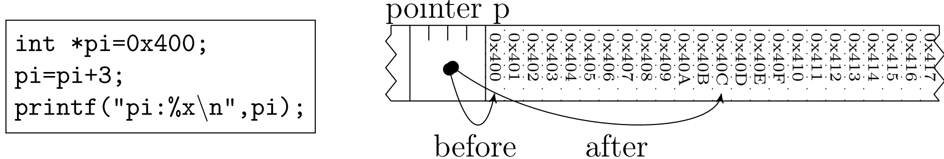
Martin Quinson

- **Logistique ENS:** Portes ouvertes lycée: il faut déjà savoir lesquelles ont lieu (covid?), et quand. Le dpt rembourse un (seul) A/R de train par lycée pour ça. Parlez vous entre anciens collègues.
- **Retours TP2**
  - 'A' et 'a' plus lisibles que les valeurs numériques
  - `if (fich==NULL) return` plus lisible qu'un niveau d'imbrication supplémentaire
  - on n'écrit pas `\};`
- Identifiants locaux / globaux
  - globales statiques : portée réduite
  - locales statiques : durée de vie augmentée
- Pile d'appel, les paramètres sont des variables locales
- Organisation mémoire du processus : les trois segments Data / Tas / Pile (explique les locales statiques)
- Espace d'adressage, la mémoire est un grand tableau
- Pointeur = adresse d'une case
  - Permet de modifier n'importe où en mémoire. C'est la force du C ; c'est la raison d'arrêter le C.
  - Les pièges des pointeurs: une étoile en trop et c'est le drame; l'étoile a 3 significations
  - Le pointeur NULL

I) Pointeurs (suite)

I.1) Arithmétique des pointeurs

- Addition: pointeur + entier : valide.



- – C'est un décalage en case, pas en octets.
  - C'est déroutant, mais c'est parce que les pointeurs ressemblent aux tableaux en C

    `p[i]` $\triangleq$ `*(p+i)`
  - Donc `after = before + sizeof(int)*3` car ce sont des entiers.
- Soustraction: pointeur - entier : valide
  - C'est la même chose, mais en décalant vers la gauche.
- Toutes les autres opérations entre pointeurs et entiers sont invalides (division, multiplication, etc)

I.2) Transtypage (cast en anglais)

- c'est l'art de convertir un type dans un autre. Par exemple `int a = `**(int)**`b`.
  - Quel que soit le type de `b`, le compilateur fait de son mieux pour en faire un entier
- On retrouve des transtypages dans à peu près tous les langages.
- Deux types très differents de transtypages en C.
- Dans tous les cas, le type en C ne dénote pas de la sémantique, mais taille et représentation en mémoire

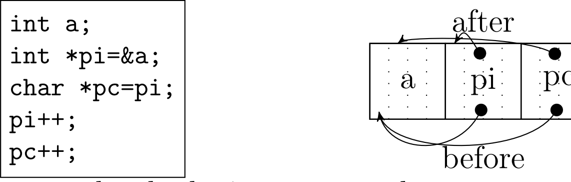
I.2.a) Transtypage de scalaires

- un scalaire est une valeur normale: un entier, une lettre un double.
- Lors d'un transtypage de scalaire, on change la valeur.
- `double d = 5.7;` [5.7]

  `int i = (int)d;` [5]
- Cela change la représentation en mémoire.
  - Dans l'exemple, double représentés en IEEE 754 (partout), sur 8 octets; entier sur 4 octets en 32bits.
- Cela peut induire une perte irréversible de précision
  - Dans l'exemple, on ne retrouvera jamais le 0.4 depuis la variable `i`

I.2.b) Transtypage de pointeurs

- Mémoire inchangée (donc valeur inchangée), mais change la sémantique.
  - Càd, change la façon d'interpréter les pointeurs (arithmétique)



- Dans l'exemple: `pi` se décale de 4 octets car il pointe sur des entiers; `pc` se décale d'un seul octet.

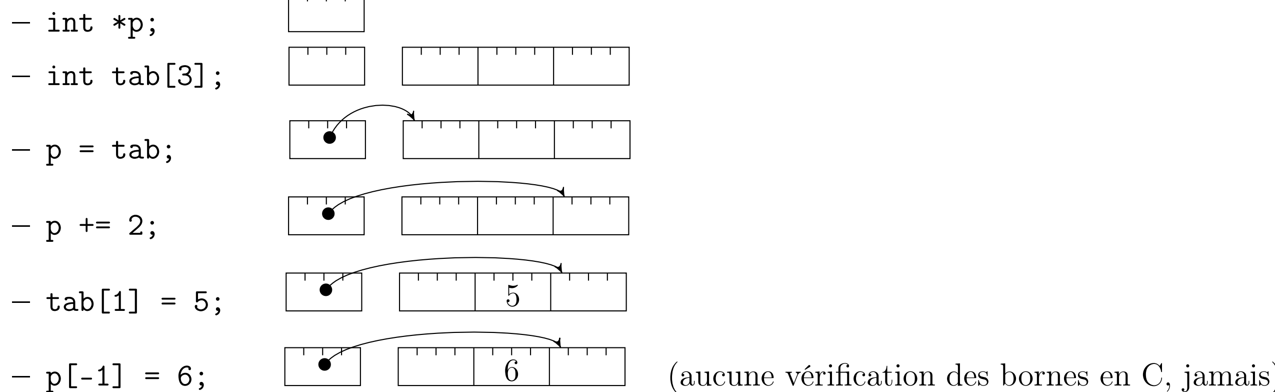
- C'est cohérent avec l'idée que les pointeurs peuvent être utilisés comme des tableaux en C

I.3) Pointeurs génériques: `void*`

- Ce sont des pointeurs dont on ne connaît pas le type.
  - Exemple: les paramètres de `printf` ou `scanf`
  - Exemple: manipulation directe de la mémoire `memcmp`, `memcpy` (si pas overlap), `memmove` (si overlap)
- A l'origine on ne pouvait pas faire d'arithmétique des pointeurs sur ces pointeurs, et c'est assez logique
  - Mais `gcc` l'autorise quand même, en supposant que `sizeof(void)=1` car c'est trop pratique
    - C'est une extension GNU, donc `clang` le fait aussi mais pas `icc` ni `visualC`
  - Oui, la taille du vide n'est pas nulle, mais le vide est de taille 1 :)

I.4) Pointeurs et tableaux

- Tableaux et pointeurs se ressemblent beaucoup en C, sans être la même chose: conversions automatiques
  - C'est une source de complexité très avancée. Les plus curieux liront le tuto "la vérité sur les pointeurs et les tableaux en C" (sur la page du cours) pour comprendre. Les concepteurs du langage ont un regret pour ce point précis : ils ont fait trop compliqué.
- Exemples de code pour comprendre les points communs et différences



- Dans cet exemple, `tab = p` n'aurait aucun sens: `tab` n'est pas une case mémoire modifiable.
  - Il faut voir `tab` comme une valeur numérique, comme 42. On ne change pas la valeur de 42.
  - Chaque fois que le compilateur voit `tab`, il écrit une valeur numérique, l'adresse du début de `tab` dans le segment *Data* qu'il construit (ce qui est joyeux à calculer avec l'ASLR address space layout randomization, mais c'est vraiment hors sujet).
  - C'est aussi pour ça qu'on ne peut pas écrire `tab1 = tab2` Le compilateur ne voit que des adresses.
- Pour bien comprendre la différence entre pointeur et tableau, réfléchissons à la taille mémoire occupée.
  - `p` est un pointeur $\rightsquigarrow$ 4 octets en 32bits; `tab` est un tableau de 3 cases $\rightsquigarrow$ 12 octets en 32 bits
  - Mais piège supplémentaire: `sizeof` appliqué aux tableaux ne compte pas le nombre d'octets de la représentation mémoire (comme sur tout le reste), mais il compte les cases du tableaux (comme l'arithmétique des pointeurs)...
- Donc les pointeurs et les tableaux sont différents.
  - Sauf dans les types de paramètres: `fun(char *p)` $\triangleq$ `fun(char p[3])` $\triangleq$ `fun(char p[])`
  - Là, les pointeurs et les tableaux sont rigoureusement identiques (et la taille du tableau est ignorée)
  - C'est historique, mon ami.
- Les pointeurs de tableaux ne sont pas des tableaux de tableaux (ni des pointeurs de pointeurs)

- si `int tab[3] = { 0, 1, 2};` alors `&tab` est de type `int (*)[3]`, ce qui n'est pas `int **`
- mais c'est compliqué, vous irez voir le doc sur le site si ça vous intéresse

I.5) Les chaînes de caractères

- Il n'y a pas de type `String`, on fait des tableaux de caractères.
- Pas de métadonnées où ranger la taille. Convention: chaînes terminées par le caractère zéro, noté `'\0'`
- Fonctions à connaître: `strlen`, `strcpy`, `strcat`, `strcmp`

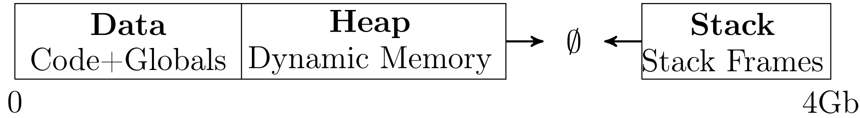
I.6) Pointeurs sur fonction

- C'est utile pour passer des callbacks (préciser un comportement à exécuter quand un événement arrive) ou pour faire du *dynamic dispatch* (associer un code à exécuter à un type de structure, pour faire un pas de plus vers la POO en C. Pas forcément une bonne idée: C++ préférable)
- Définir une variable de type pointeur: il faut des parenthèses.
  - `int (*fun)(void); //` La variable est nommée `fun`, son type est `int(*)(void)`.
  - Si on oublie les parenthèses, `int*` est plus prioritaire, et le résultat n'a pas de sens.
- Affecter une valeur à un pointeur sur fonction: pas besoin d'esperluette
  - `int mafonction(){ return 42; }` // Déclaration d'une fonction
  - `fun =&mafonction;` // La variable `fun` contient maintenant un pointeur vers `mafonction`.
  - `fun = mafonction;` // exactement comme avant. Ce `&` est optionnel
- Invoquer la fonction pointée: naturellement
  - `x = (*fun)()` // La variable `x` vaut maintenant 42
  - `x = fun()` // Exactement comme avant, même si c'est très moche. C'est autorisé car non-ambigu.

II) Mémoire dynamique

II.1) Motivation

- Les tableaux sont de taille statique en C
  - L'écriture `int n; scanf("%d", &n); int tab[n];` est interdite par défaut (ça marche en C99)
  - Donc il faut connaître la taille de tous les tableaux à la compilation
  - C'est très pénible, on voudrait pouvoir faire des tableaux de taille connue seulement à l'exécution
- Solution: on va le faire à la main, en utilisant le tas:
  - On demande des blocs mémoire quand on en a besoin
  - On les rend après usage



II.2) L'interface malloc

- c'est l'approche standard. c'est une bibliothèque; emacs plus hardcore et fait sans ça, `brk()` direc
  - `#include <stdlib.h>`

  - `void* malloc(int size)`: réserve un bloc de `size` octets en mémoire et retourne adresse début
  - `void free(void* p)`: libère (rend) un bloc précédemment alloué
  - `void* realloc(void* p, int size)`: modifie la taille d'un bloc; /!\ cela peut le déplacer
- Exemple simple.
  - `void *A=malloc(12);` [A   ]
  - `void *B=malloc(5);` [A B   ]
  - `free(A);` [B   ]
  - `void *C=malloc(6);` [C B   ]
  - `C=realloc(C,13);` [B C]

II.3) Survivre à `malloc`

- Comme d'habitude en C, il n'y a aucun garde-fou et la moindre erreur mène au SEGFAULT
- Solution 1: **bonnes pratiques** pour éviter les problèmes (et métaphore du notaire: mémoire = terrain)
  - **Règle #1:** on n'accède qu'à des zones réservées
    - usage avant malloc: on achète le terrain avant de construire, svp
    - usage après free: on arrête d'utiliser ce qu'on a vendu, svp
    - symptômes sinon: SEGFAULT si chanceux, corruption mémoire quelque part sinon
      - On ne peut pas être chanceux sur un *use after free*
  - **Règle #2:** à chaque `malloc` correspond un `free`
    - s'il manque un `free`, alors c'est une fuite mémoire *(memory leak)*
      - Le système pense que la mémoire est encore prise et ne peut la réutiliser
      - Quand y'a qu'un leak, ça va. C'est quand il y en a beaucoup que ça pose problème.
      - Ralentissement *(swapping)*, puis malloc retourne `NULL`, puis l'OOM killer de linux intervient
    - Double `free` du même malloc: on vend deux fois
      - Si le bloc n'avait pas été réalloué, c'est une no-op
      - S'il avait été réalloué [dans un autre module], ça le libère dans le dos de l'autre. Pas cool.
      - `int *A = malloc(12); free(A);`
      - `int *B = malloc(12); free(A);` $\rightsquigarrow$ libère `B`.
      - Les mallocs modernes se suicident quand ils détectent ça. Car oui, y'a plusieurs implem de malloc, et même de la recherche en R&D là dedans. La détection se base sur les métadonnées autour. Implémenter son propre malloc est un exercice instructif, même s'il est dur de prendre les vrais de vitesse
- Solution 2: il faut **connaître les outils** pour soigner le mal:
  - `valgrind` pour détecter les erreurs à leur source, et pas seulement les défaillances: un outil formidable, simple d'accès et voit 90% des problèmes (seul le tas est surveillé, pas la pile)
  - `gdb` pour explorer l'état de la mémoire: un outil formidable. Vieux mais parfaitement robuste. Pratique comme un tank soviétique.
  - C'est l'occasion d'introduire un peu de vocabulaire classique
    - faute: bêtise du programmeur
    - erreur: comportement incorrect (ie, comportement proscrit, ou qui ne fait pas partie de la spec)
    - défaillance: comportement observé $\neq$ comportement attendu (là, ça se voit)

III) Structurer son code

- En C, on est libre de tout, y compris de coder proprement
- Il faut tout faire, car le compilateur n'aidera en rien (il ne se plaint pas des fautes de goût)

III.1) Déclarer des structures

- Base de la propreté en informatique: ranger ensemble ce qui va ensemble. En C, c'est des structures
- Voir le code de la structure `point` sur la feuille de pompe. `/!\` au ; final
- Exemple d'usage avec la notation pointée
- Déclaration sur place: `struct point p2 = {4.2, 3.7, 2};`
- Usage en paramètre ou en retour de fonction $\rightsquigarrow$ copie complète, comme d'habitude
- Possibilité de faire des structures récursives.
- Pas d'opérateurs globaux. `memset` et `memcpy` pour mettre à zéro ou copier l'un dans l'autre.

III.2) Modèles d'organisation d'un code C

- L'objectif majeur est la lisibilité.
  - On est pas là pour jouer à l'IOCCC
  - Optimisation = mauvaise idée, ca complique et faut bien comprendre pour être sûr que ça gagne
    - Rule 1: Optimization: don't do it
    - Rule 2 (experts only): Optimization: don't do it yet.
    - Premature optimization is the root of all evil. Knuth.
- Programmer procédural en C
  - C'est la façon historique
  - On organise des modules où toutes les fonctions qui vont ensemble sont dans le même fichier
  - L'état est dans des globales cachées (déclarées globales static dans un module)
  - L'interface est dans un fichier d'entête
  - cette approche est *has been* pour de bonnes raisons (mauvaise encapsulation, impossible d'avoir deux instances du module)
- Programmer orienté objet en C
  - État placé dans une structure. Fonctions prennent un pointeur vers l'instance en premier argument
  - Pas si éloigné du python avec son `self`, si on regarde pas de trop près
  - On peut faire de l'encapsulation et du dispatch (avec des pointeurs sur fonction) assez facilement
  - On peut aussi faire de l'héritage, mais faut savoir arrêter et passer au C++ (ou Rust!)

III.3) Le module point en détail

- Alias de type avec `typedef` pour raccourcir par rapport au très valide `struct point*`
- Constructeur, destructeur, copy constructeur; Des fonctions de manipulation
- Le contenu de la structure et l'implem des méthodes est caché, dans un fichier à part
  - Le compilo n'a pas besoin de connaître la structure car on manipule un *pointeur vers* de taille connue
- Le fichier d'entête donne le prototype de ces méthodes
  - Inclus dans le fichier d'implem pour vérifier que le prototype correspond à la vraie définition

- Inclus dans les fichiers appelants pour annoncer le prototype au compilo
- Attention, il faut protéger contre la double inclusion (crétin de compilo)

III.4) Compilation séparée

- Un seul fichier de 500 000 lignes, c'est dur à lire, naviguer; long à compiler; dur à collaborer
- Mais 5 fichiers séparés, c'est dur à compiler. Il faut savoir quoi recompiler quand. Transitif inclusions.
- Il faut pas essayer de compiler à la main, mais il faut utiliser un outil pour ça
- `make` est parfait pour compiler: il a un arbre et gère tout bien. Mais dur d'écrire un `Makefile`
- `cmake` converti un fichier `CMakeList.txt` à peu près lisible en `Makefile` très bien fait

IV) Conclusion

- Notions pas vues ici: enum, union, bit fields. Mais ça devrait suffire pour la plupart des projets où vous aurez à coder
- L'objectif était surtout de vous faire comprendre comment fonctionne la machine au plus bas niveau, quand on a meme pas de libc pour nous aider
- A partir de là, on peut partir dans plusieurs directions
  - le réseau, à partir de la semaine prochaine, pour découvrir un gros système cohérent
  - le C++, le semestre prochain, pour avoir de l'aide du compilateur pour organiser de gros codes
  - l'architecture, le semestre prochain, pour mieux comprendre comment fonctionne un vrai processeur
  - le système, l'an prochain, pour voir comment fonctionne l'OS, et comment sont implémentés les OS