3: Mémoire statique

Martin Quinson

Épisode précédent

- C est un langage préhistorique, mais les outils aident. IOCCC
  - Les fichiers en C. fopen, fprintf, fscanf.
- Qu'est ce qu'un OS; Les 3 fonctions de base: protéger, adapter l'interface et virtualiser.
- Sécurité informatique. Faut protéger ses infrastructures ; la capacité technique ne donne pas le droit.

I) Identifiants C, portée et durée de vie

I.1) Portée des identifiants locaux et globaux

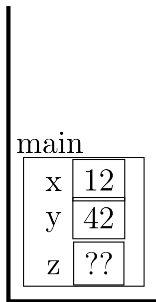
- Pas de différence entre variables et fonctions (application un peu extrême des idées de Von Neumann)
- Les identifiants sont globaux ou locaux. Cela leur donne une visibilité ou portée (scope) et une durée de vie spécifique.
  - Les variables locales sont dans une fonction.
    * Scope: visible depuis le bloc appelant
    * Durée de vie: jusqu'à la fin du bloc
    * Faire des fonctions locales n'est pas très C-ish, même si les compilos modernes acceptent.
  - Identifiants globaux (variables et fonctions)
    * Scope: partout ; Durée de vie : toujours (jusqu'à la fin du programme)
- Étude du programme 1 sur la visibilité
- Oui, on peut faire des locales à un sous-bloc, aussi.
- Non, faire des variables de même nom qui se masquent ainsi n'est pas une bonne idée.
- Pour comprendre, il faut imaginer des variables qui s'empilent, et indicer a par la ligne de déclaration (a_2 pour la globale)

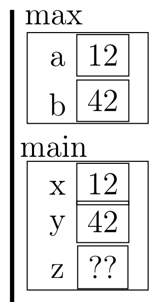
```
(l5)  a:0  b:0
(l9)  a:?  b:0
(l14) a:10 b:10
(l18) a:10 b:0
(l20) a:10 b:15
(l22) a:10 b:0
```

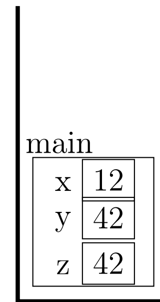
I.2) Paramètres et pile d'appel

- Quel est l'affichage du **programme 2** (fonctions et paramètres) ?
  - Cela inverse les valeurs de `a` et `b` car les paramètres ligne 4 sont inversés. haha, très drôle. En fait, non. C'est pas drôle, il faut pas écrire des pièges à soi-même comme ça.
- La pile d'appel est un endroit de la mémoire où sont créés des **cadres de pile**, un contexte d'exécution pour chaque appel de fonction (récursif ou non). C'est là que vivent les variables locales. Créé à la demande lors de l'appel, puis détruit lors du `return`.
- Les paramètres sont des variables locales (que la bienséance seule interdit de modifier même si le **programme 3** montre que le compilo laisse faire)

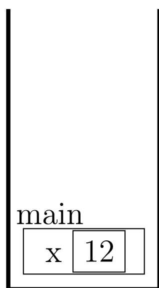
- En C, (tous) les paramètres sont passés par copie.
- Étudions le **programme 4a**

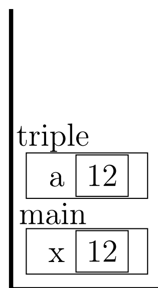


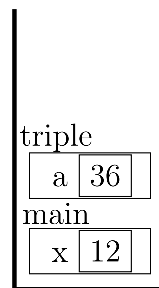


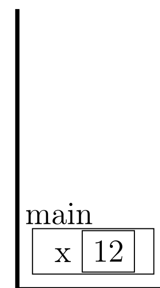


- Attention, le passage par valeur pose parfois des pièges, comme on va voir dans le **programme 4b**
  - On calcule bien 36 dans la fonction, mais cette zone mémoire est effacée en sortie de fonction









I.3) Mot-clé `static`

- Quel est l'affichage du **programme 5** ?
  - La colonne des a est remplacée par des zéros; l'incrément de a en fin de fonction est sans effet puisque la variable est détruite après chaque appel
- Le mot-clé `static` a deux significations:
  - Associé à une variable locale, cela augmente sa durée de vie tandis que sa portée est inchangée. Cf. **programme 6**, qui fait ce à quoi on s'attend: la fonction `nextInt` énumère les nombres entiers
  - Associé à une variable globale, cela réduit sa visibilité au fichier courant. Sa durée de vie est inchangée. C'est l'équivalent Java ou C++ de `private`

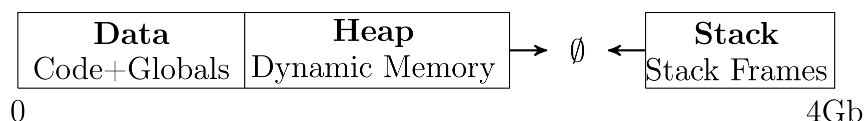
	portée	durée de vie
Fonction	projet	∞
Variable globale	projet	∞
globale `static`	$\approx$ fichier	∞
locale `static`	bloc	∞
locale	bloc	bloc

- Plus précisément, une globale statique est limitée à l'unité de compilation, c'est à dire au .o constitué.
- Il est important ici de comprendre que la portée est définie à la compilation, tandis que la durée de vie est à l'exécution.
- Ces nouvelles connaissances posent la question de *où* sont stockées les locales statiques en mémoire.
  - on dirait une globale : initialisé une seule fois et persistant entre les appels
  - Il faut parler système pour comprendre ce mystère

II) Organisation logique de la mémoire

II.1) Memory layout d'un processus dans l'OS

- Les locales `static` sont troublantes
  - On dirait une globale (initialisé une seule fois, presistante) mais déguisée en locale (visible ici seulement)
  - Pour comprendre, il faut savoir comment est organisée la mémoire du processus.
  - (un processus est un programme en cours d'exécution dans l'ordinateur)
- Il y a trois segments mémoire, de 0 à 2^{32} en 32bits
  - **Data**: Là où se trouvent le code compilé + les variables globales. C'est ce qu'on avait en M99
  - On reparle du **tas** bientôt: c'est là où on fait les `malloc` pour ceux qui connaissent
  - **Pile**: Contient les cadres de piles comme vu la semaine dernière, et comme proposé par l'extension du M99
  - Il y a un trou (une zone inutilisée) entre la pile et le tas, qui peuvent grandir tous les deux. Si collision, alors le processus a épuisé la mémoire disponible.

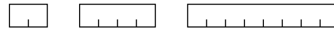


- C'est de l'**adressage virtuel**, car l'OS fait de la médiation sur les cases mémoires lues.
  - A chaque accès mémoire, le programme lit une adresse virtuelle sans savoir où les données sont physiquement en mémoire. Le CPU a une table associative (TLB – translation lookaside buffer) qui à chaque couple $(pid, adresse\ virtuelle)$ associe une adresse physique.
  - Il faut faire efficace pour les performances générales donc fait par matériel: MMU = zone du CPU.
    - De plus la TLB groupe la mémoire par pages, pas des adresses individuelles
  - C'est fondamental pour assurer les fonctions de l'OS vu la semaine dernière: protection des applis; virtualisation mémoire; uniformisation des accès mémoire depuis l'appli
- Cela explique les locales `static`: à la compilation, ce sont des locales, mais le compilateur les place dnas le segment Data, pas dans les cadres de pile.

II.2) Espace d'adressage virtuel

- C'est un autre nom pour la mémoire d'un processus. Elle peut donc être vue comme un grand tableau de case mémoire successives. Chaque case fait un octet sur tous les systèmes (hard et OS) que je connais.
- **Adresse mémoire**: numéro de la case dont on parle, tout simplement. En adressage virtuel, hein.
- *Pourquoi la pile commence à 4Go ?* Car je fais mes dessins en 32bits et que MAXINT=2^{32}=4Go sur cette architecture.
  - En fait, sous linux, la pile commence à 3Go car le dernier Go de l'espace d'adressage est un mapping direct d'une zone de l'OS pour rendre les context switch entre l'appli et l'OS plus efficaces. Mais c'est hors programme.
- *Et si l'ordinateur n'a pas tant de mémoire?* $\leadsto$ virtualisation: des pages swappées sur disque au besoin
- *Contenu d'une cellule*
  - Avec 8 bits on code 256 valeurs. Par exemple $[0, 255]$ ou $[-127, 128]$

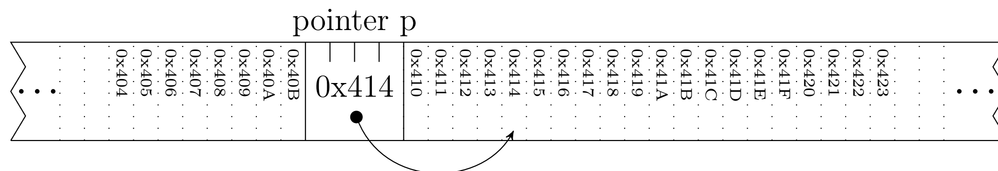
  – Pour manipuler des plus grandes données, on agrege plusieurs cases (lues et interprétées ensemble)
  – 2 cellules: $2^{16} = 65535$; 4 cellules: $2^{32} \approx 4 \cdot 10^{10}$; 8 cellules: $2^{64} \approx 1.8 \cdot 10^{19}$



- *Organisation interne*
  – Il n'y a aucune meta-donnée, ce qui est cohérent avec le langage C: si tu l'as pas fait, y'en a pas.
  – On ne sait donc pas interpréter les données qui se trouvent dans la mémoire
    * comme en M99: on sait pas si 110 est `LDA 10`, ou si c'est la valeur numérique 110.
    * En C, on ne sait même pas si c'est une valeur ou un morceau (l'une des cases) d'une valeur

III) Pointeur

- C'est un mot qui fait peur quand on apprend le C, et c'est une source d'erreur très importante. Peut-être la principale
- Mais à la base, c'est juste une variable numérique contenant une adresse mémoire
- En 32bits, il faut 4 cases pour stocker une adresse



- Pour savoir interpréter une case pointée, il faut connaître le type de donnée stockée à cet endroit

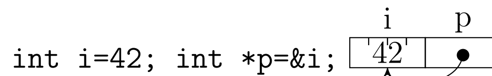
  `char* pc;` a `int* pi;` 42

- On peut pointer vers une case contenant un pointeur. On obtient par exemple `int**`

III.1) Utiliser les pointeurs

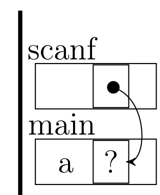
- La valeur numérique des pointeurs n'a aucune valeur sémantique (valeur `0x414` sans importance).
  – Ce qui compte, c'est ce vers quoi ça pointe (c'est un pointeur vers là où la variable `cpt` est stoquée)
  – C'est à quoi sert l'opérateur `&` qui se lit *adresse de*

`int i=42; int *p=&i;`



- On a déjà rencontré ce `&`: dans les paramètres de la fonction scanf. Cela explique comment elle peut modifier les variables dans la fonction appelante:
  – Et c'est pour ça que la fonction a besoin du `%d`: c'est pour savoir comment interpréter ce pointeur.
  – Oui, les pointeurs permettent de modifier la mémoire hors du scope. Mémoire magma informe.

```
int main() {
  int a;
  scanf("%d",&a);
}
```



- On peut maintenant corriger le code de la fonction triple sur la feuille
  – Le paramètre est de type `int*`, on lit et modifie avec `*a`, on appelle avec `&a`

III.2) Pièges classiques avec les pointeurs

- **#1: L'étoile `*` a une sémantique extrêmement lourde.**
  - une étoile en trop ou pas assez dans un programme, et c'est le SEGFAULT (mort du processus)
- **#2: L'étoile a deux sens très différents**
  - =int *p= declares a **pointer variable** `p` which is a pointer to an integer value
  - =*p= is then the **pointed value**, interpreted according to the pointer type
  - (that's actually three meanings when counting $\times$, the multiplication)
  - =int *p; p=12;= selects where it points in memory
  - =int *p; *p=12;= changes the memory in the pointed area
  - Pascal was a bit more reasonable: `INTEGER ^p` vs. `p^` (pas le même ordre au moins)
  - In Java, there is no pointers, but reference to objects are close to that concept

III.3) Le pointeur NULL

- Utile pour dire non initialisé, ou invalide (comme dans la fonction `fopen`)
- Par convention, c'est la valeur numérique 0 (même ça, ça n'est pas dans le langage. Un simple define)
- Pour s'assurer que l'OS fait bien le SEGFAULT attendu, on a qques pages sans permission de r/w au début