

Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC sem7 : pointeur (2) *le retour...*

Questions sur le projet

Un exemple d'usage de pointeur: réseau d'amis

Mémoire centrale: pile (stack) et tas (heap)

Pointeur et allocation dynamique de mémoire

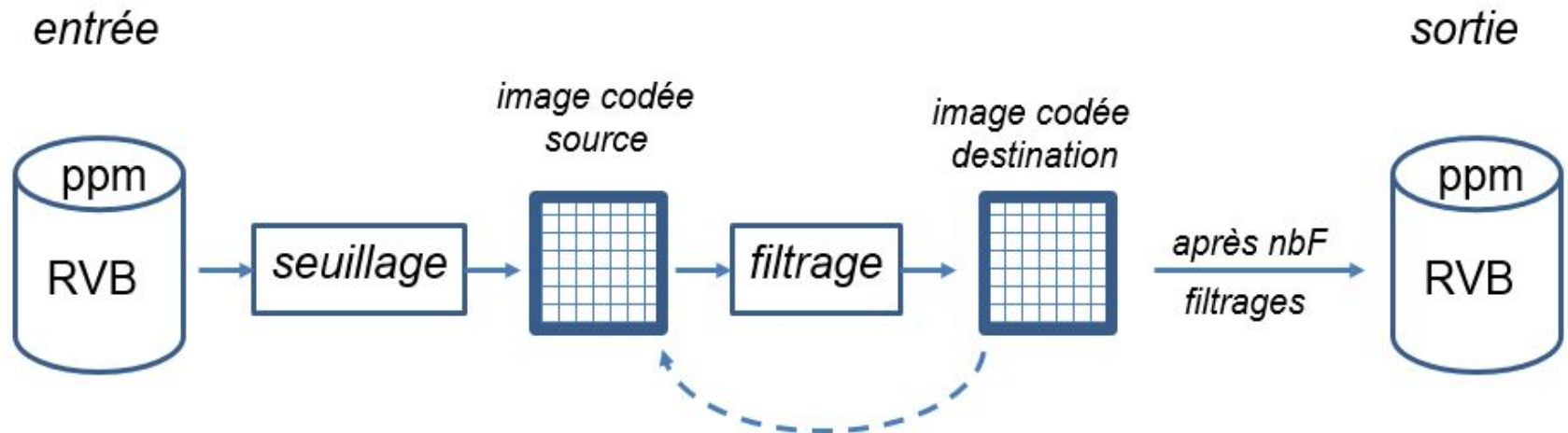
Démo du projet ColoReduce / Questions ?

Erreurs fréquentes

- Ne pas initialiser ses variables
- Déclarer les dimensions des tableaux, array ou vector avec des variables qui n'ont pas encore été lues ou, encore pire, qui n'ont pas été initialisées.

Comportement fréquent

- Ne pas réussir à relire son code avec un oeil neutre et auto-critique
- Ne pas tester ses fonctions *une à une* avant de tester l'ensemble du programme



Un exemple d'usage de pointeur: réseau d'amis

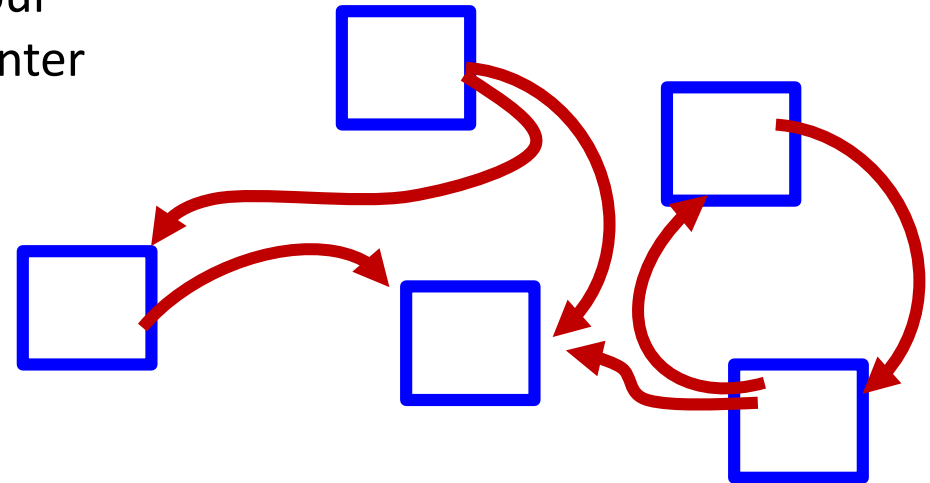
Besoin: définir une structure **Personne** qui contient les informations d'identification et *la liste de ses amis*.

Idée: utiliser la même structure **Personne** pour représenter un(e) ami(e). Permet de représenter un réseau d'amis de manière homogène.

Problème: une structure ne peut pas se contenir elle-même.

Solution: une structure peut contenir une liste de liens (pointeurs) vers d'autres structures.

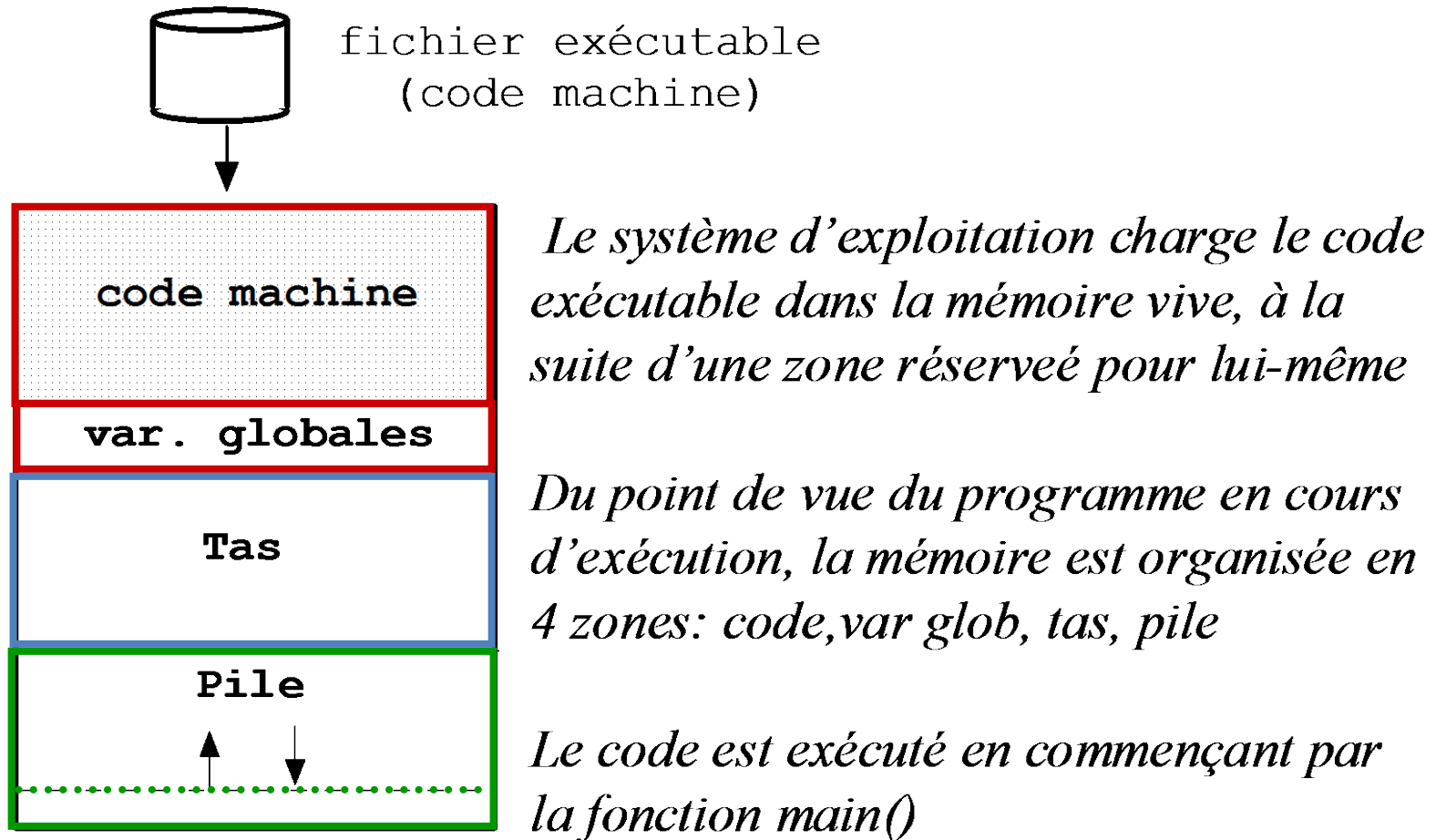
```
struct Personne{
    string nom;
    // ...
};
```



```
struct Personne{
    string nom;
    vector<const Personne*> amis;
};
```

Mémoire centrale: pile (stack) et tas (heap)

Organisation de la mémoire pour l'exécution d'un programme



Rappel: Pile => pour les variables locales à une fonction

- Les 4 zones (*programme*, *variables globales*, *Tas (heap)*, *Pile (stack)*) existent toujours
 - zone immédiatement après le code exécutable :
 - Variables globales
 - Constantes littérales chaînes de caractères: "EPFL"
 - les fonctions travaillent avec la Pile (stack)
 - A **chaque appel** (imbriqué) de fonction, une zone d'**espace mémoire** est prise **au sommet** de la pile



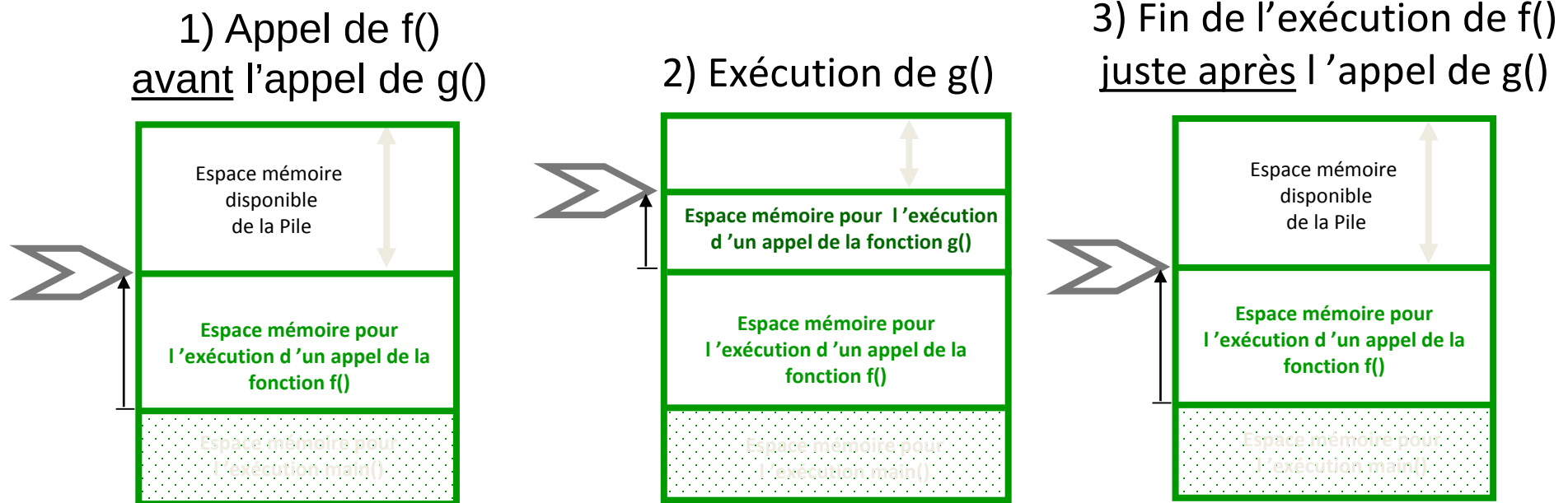
Cette zone réservée sur la pile mémorise :

- les **arguments** (qui initialisent les paramètres formels)
- les **variables locales**
- la **valeur de retour**
- l'adresse de retour de la fonction

main() appelle la fonction f() qui elle-même appelle la fonction g()

```
main()
{
    ...
    f();
    ...
}
void f(void)
{
    ...
    g();
    ...
}
...
```

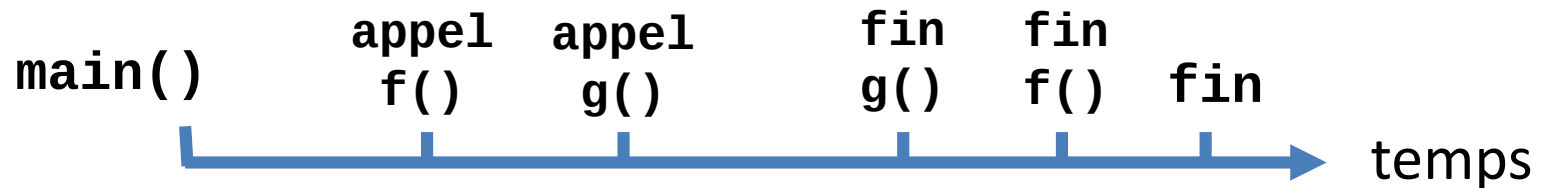
Le **Pointeur de Pile** est mis à jour à chaque appel



Durée de vie: les variables locales n'existent que pendant l'appel

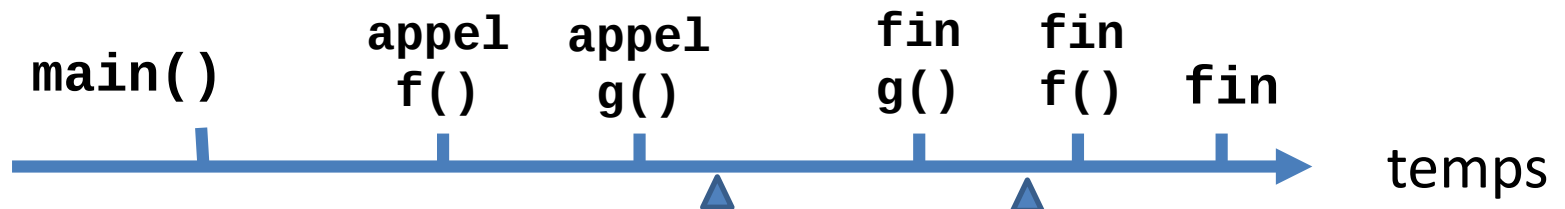
Le tas (heap) offre une durée de vie étendue

Durée de vie sur la Pile: exécution de la fonction où la variable est déclarée



But de la zone mémoire «Tas»:

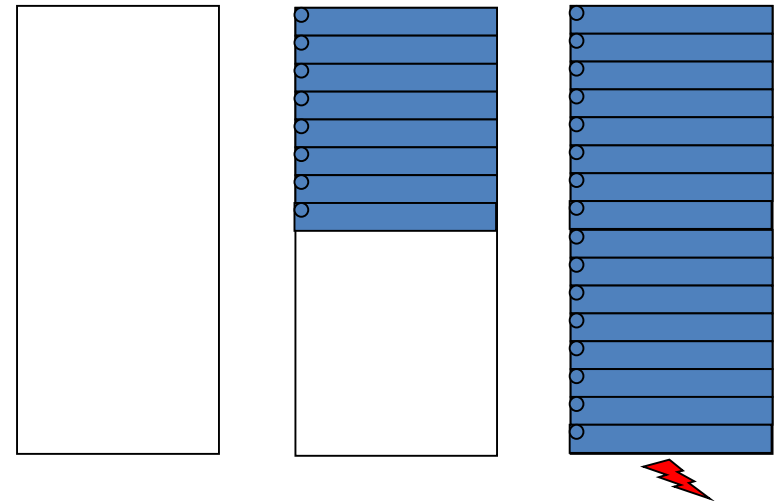
- Se libérer de la durée de vie limitée sur la pile
- Pouvoir **demander de la mémoire librement**, quand on en a besoin pendant l'exécution.
 - Éviter d'avoir à bloquer de manière statique, au moment de l'écriture du programme, de grandes zones de mémoire dans des array ou tableaux à-la-C.
- Pouvoir **libérer la mémoire quand elle n'est plus nécessaire**
- Outil: le pointeur



Le bon usage de l'allocation dynamique

Ne pas oublier de **libérer** la mémoire allouée avec **delete** quand on n'en a plus besoin

Sinon on crée une *fuite de mémoire (memory leak)* qui fait planter le programme lorsqu'il n'y a plus de place dans le Tas (heap).



L'usage des pointeurs à-la-C avec l'allocation dynamique demande une grande rigueur

Alternative à l'allocation dynamique

Il existe en C++ un outil robuste permettant **d'ajuster dynamiquement La quantité de mémoire utilisée** => **vector**

MAIS comme toute variable locale, un **vector** existe seulement pendant la durée d'exécution de la fonction où il est déclaré.

Pour augmenter sa durée de vie, une approche simple est de:

- Déclarer le **vector** dans une fonction de plus haut niveau
 - Ex ci-dessous: dans **f()** au lieu de **g()**
- Passer le **vector** par référence à la fonction **g()**:
 - **void g(vector& v);**
- Ou lui affecter la valeur d'un **vector** renvoyé par **g()**:
 - **vector g();**

