M1103 Amphi 4 : Mémoire

Thomas Nowak

Université Paris-Sud

La dernière fois

Mémoire

Mémoire

- la mémoire vive contient toutes les variables et constantes de nos programmes
- un appel à une fonction crée de nouvelles variables
  - variables locales
  - arguments
- un `return` d'une fonction détruit les variables locales de la fonction
- après avoir recopié la valeur de retour à l'endroit où elle est utilisée

Programme exemplaire

```
int main()
{
    int x = 3;
    int y = x;
    return 0;
}
```

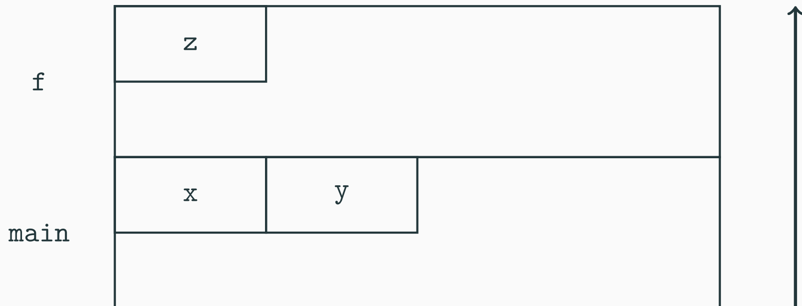
Mémoire lors de l'exécution de `main`

Programme exemplaire avec fonction `f`

```
void f()
{
    int z = 0;
}

int main()
{
    int x = 3;
    int y = x;
    f();
    return 0;
}
```

Mémoire lors de l'appel à `f`



Mémoire lors d'un appel

- tout appel à une fonction vit dans son *propre espace de mémoire*
- le code à l'intérieur de la fonction `f` ne connaît pas les variables `x` et `y`, seulement `z`
- ces espaces de mémoire sont organisés dans l'*ordre temporel* des appels
- ils contiennent les variables locales de la fonction

Arguments

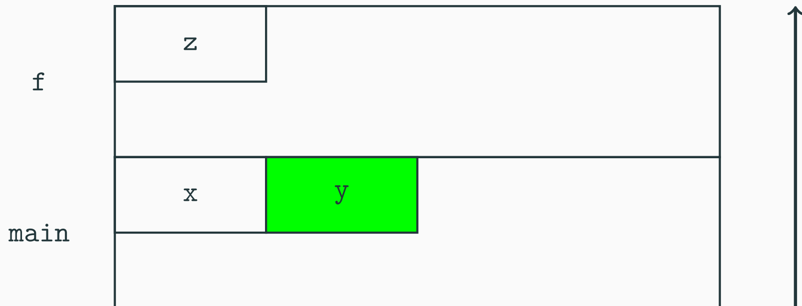
- a priori, il n'y a pas de communication directe entre une fonction appelée et la fonction qui l'a appelée
- communication via des *arguments* et des *valeurs de retour*
- du point de vue d'une fonction, il n'y a pas de différence entre une variable locale et un argument
- la recopie des argument dans l'espace de mémoire de la fonction appelée se fait juste avant l'exécution de la fonction appelée

Fonction `f` avec argument

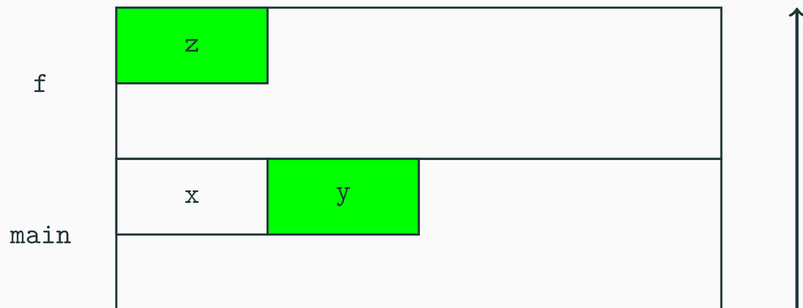
```c
void f(int z)
{
    z = z + 1;
}

int main()
{
    int x = 3;
    int y = x;
    f(y);
    return 0;
}
```

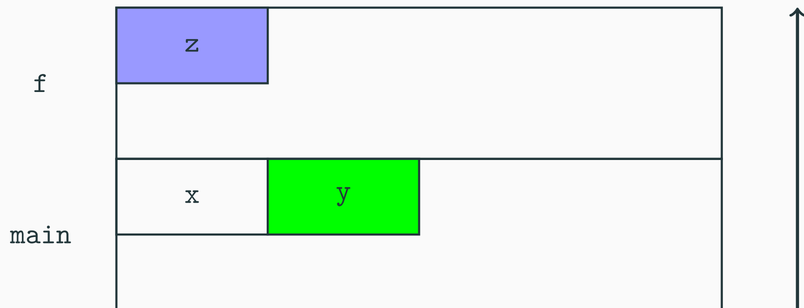
Mémoire juste avant l'appel à `f`



Mémoire juste avant l’appel à `f`



Mémoire lors de l'appel à `f`



Collisions de noms

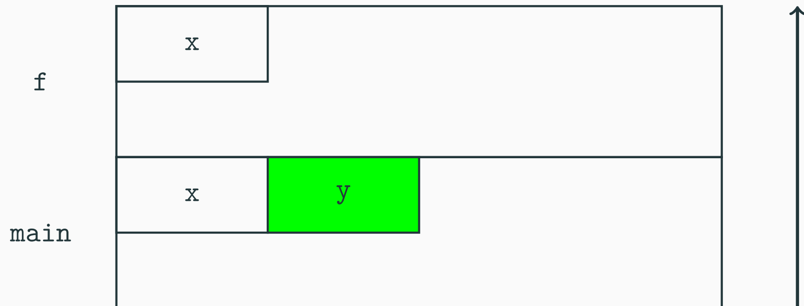
- parce que les fonctions travaillent dans des espaces de mémoire différentes, les collisions de nom ne présentent aucun problème au compilateur
- chaque fonction utilise sa propre variable dans son propre espace
- mais peut provoquer des erreurs de compréhension humaine

Fonction `f` avec collision de nom

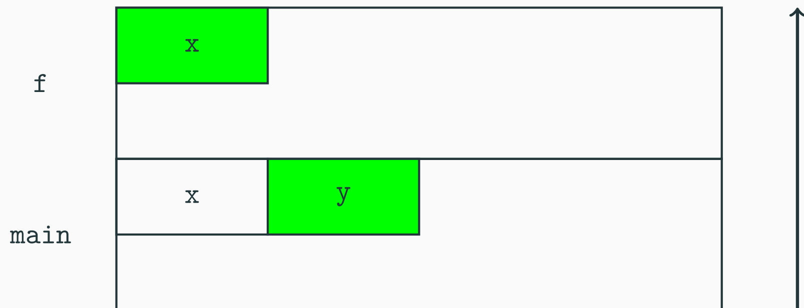
```
void f(int x)
{
    x = x + 1;
}

int main()
{
    int x = 3;
    int y = x;
    f(y);
    return 0;
}
```

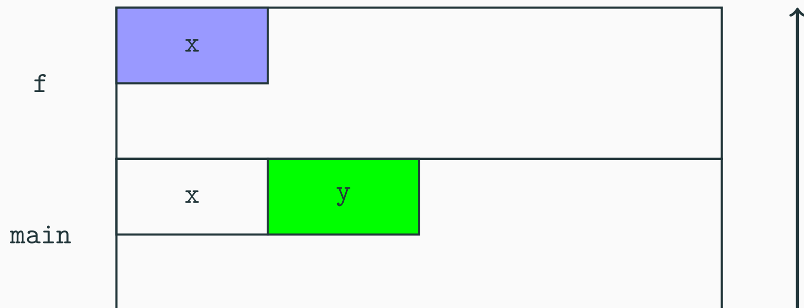
Mémoire juste avant l'appel à `f`



Mémoire juste avant l'appel à `f`



Mémoire lors de l'appel à `f`



Passage par référence

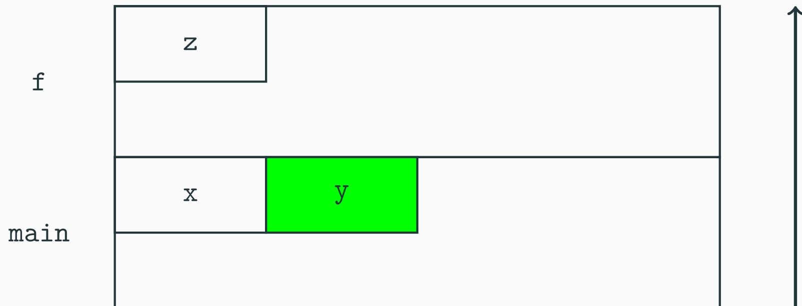
- une référence stocke l'*adresse* d'une variable ou d'une constante
- à chaque fois que la référence est utilisée dans une ligne de code, la variable ou la constante référencée est en fait utilisée à sa place lors de l'exécution
- si on passe une variable par référence à une fonction, la fonction appelée peut directement accéder à la variable originale (parce que l'adresse est connue)

Fonction `f` avec passage par référence

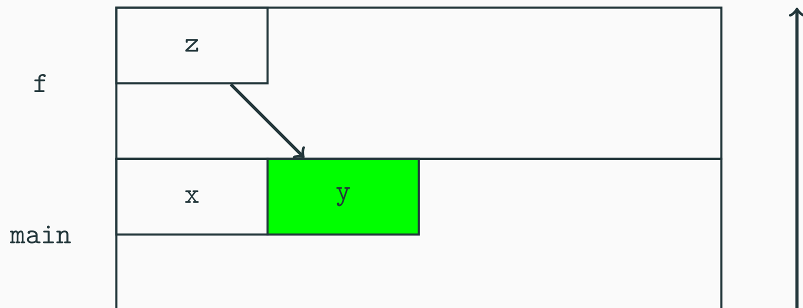
```
void f(int &z)
{
    z = z + 1;
}

int main()
{
    int x = 3;
    int y = x;
    f(y);
    return 0;
}
```

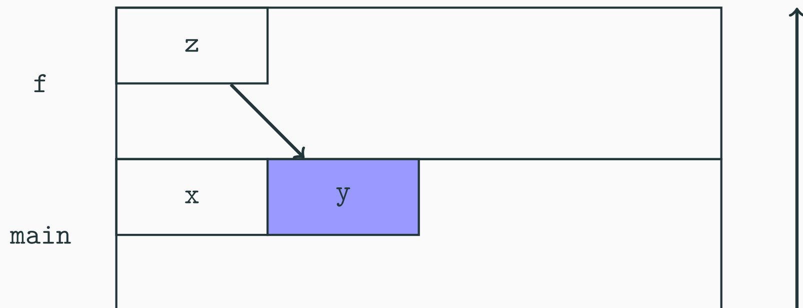
Mémoire juste avant l'appel à `f`



Mémoire juste avant l'appel à `f`



Mémoire lors de l'appel à `f`



Fonctions membres

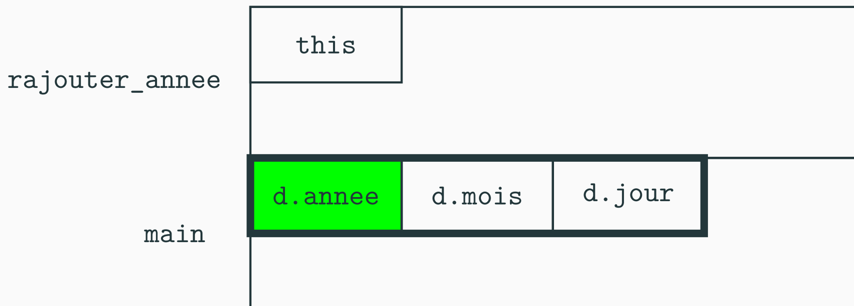
- les appels à des fonctions membres fonctionnent de la même manière
- une exception : elles ont une référence à l'objet sur lequel elles ont été appelées
- l'adresse de l'objet est stockée dans `this`
- implicitement tout nom utilisé qui n'a pas été défini dans le code de la fonction membre est cherché dans l'objet

Fonction membre `rajouter_annee` de la classe `Date`

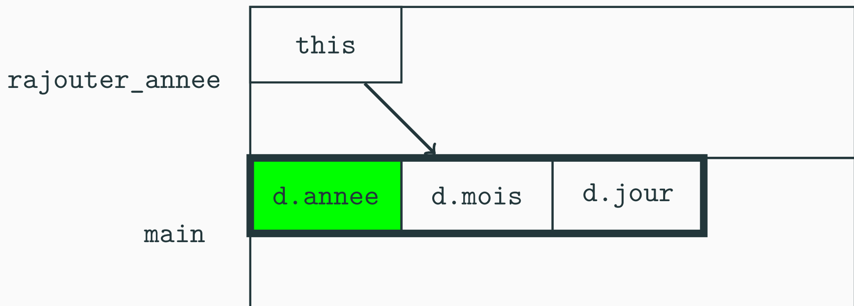
```
int main()
{
    Date d(2018, 11, 28);
    d.rajouter_annee();

    return 0;
}
```

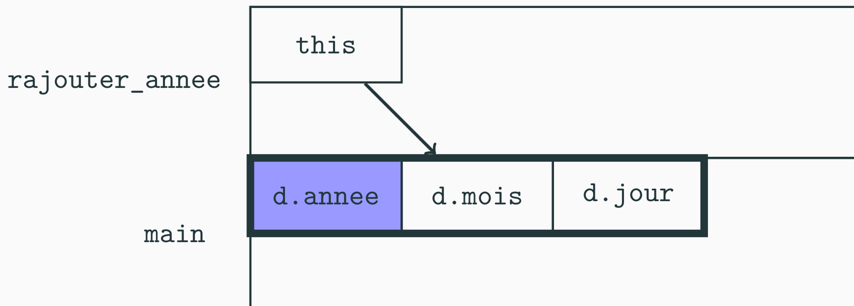
Mémoire juste avant l'appel à `f`



Mémoire juste avant l'appel à `f`



Mémoire lors de l'appel à `f`



Mémoire

- chaque appel à une fonction crée son propre espace de mémoire, avec ses propres variables
- à l'intérieur d'une fonction, un argument n'est qu'une variable supplémentaire
- passage par référence peut changer le contenu de l'espace de mémoire d'une fonction différente
- fonctions membres ont un argument implicite (l'objet courant)

Questions