

Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC sem4 traitée sur 2 semaines: fonction (1)

Les deux grands principes: Abstraction et Ré-utilisation

Notion de fonction: prototype, appel, définition

Appel avec passage d'arguments: par valeur / par référence

Portée et durée de vie des variables locales à une fonction

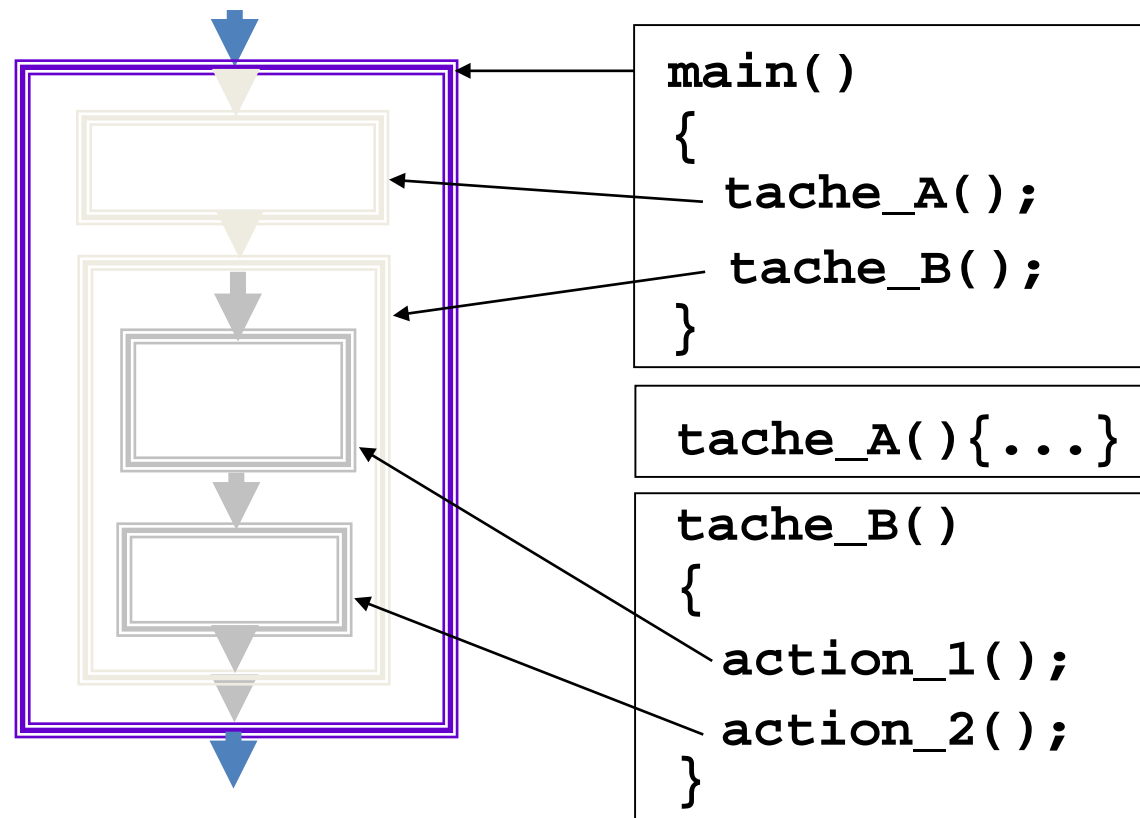
Méthode de conception d'une fonction en 5 étapes

Exemples / Questions

Les deux grand principes: le principe d'abstraction

Conception top-down: un sous-problème = une fonction

- 1) **Principe d'Abstraction**: main() présente l'idée générale de la solution (aux niveaux supérieurs) sans se perdre dans les détails

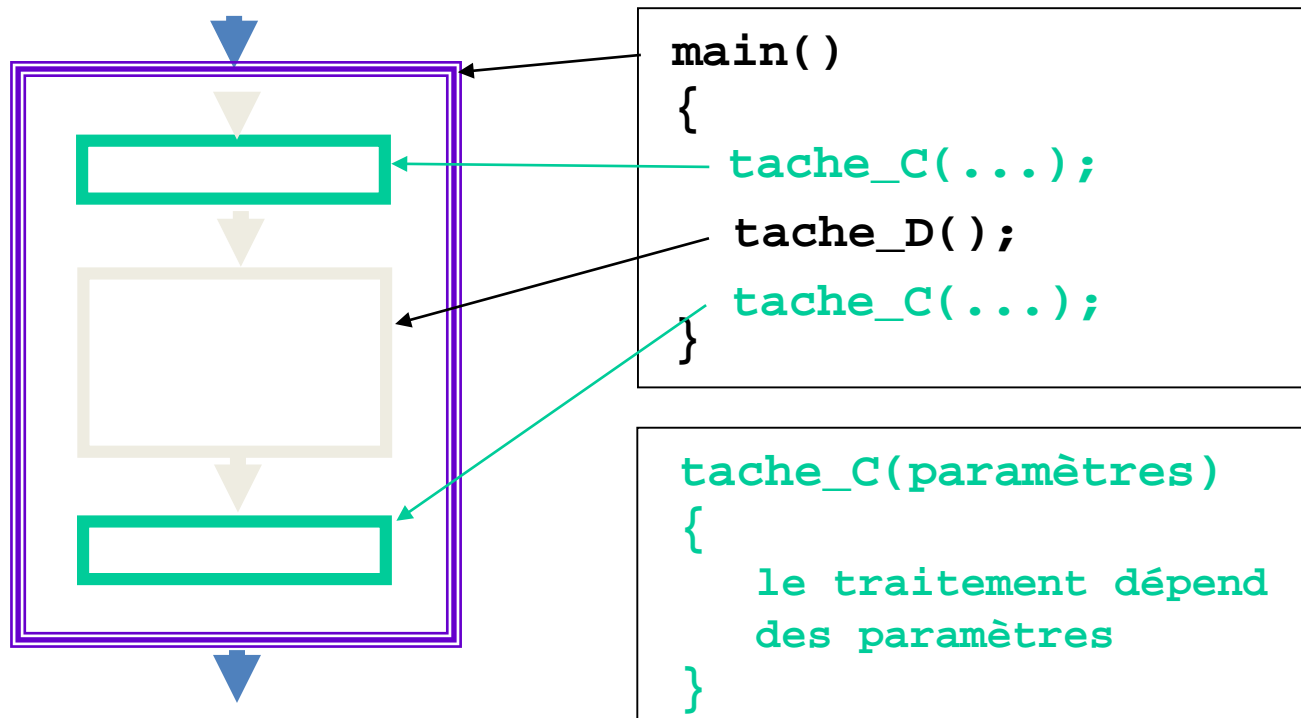


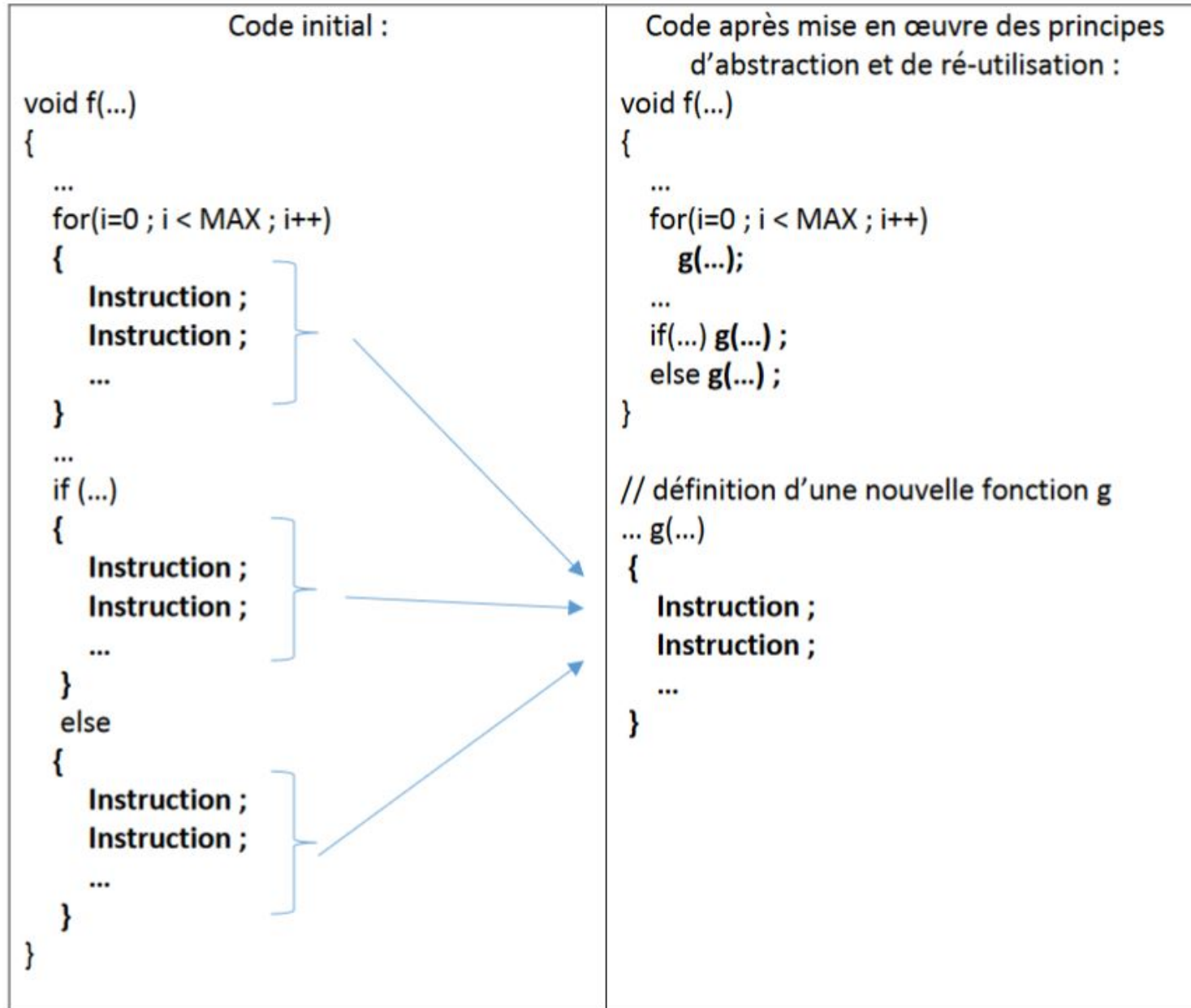
Code initial :	Code après mise en œuvre du principe d'abstraction :
<pre>void f(...) { ... for(i=0 ; i < MAX ; i++) { Instruction ; Instruction ; Instruction ; Instruction ; Instruction ; } ... }</pre>	<pre>void f(...) { ... for(i=0 ; i < MAX ; i++) g(...); ... } // définition d'une nouvelle fonction g ... g(...) { Instruction ; Instruction ; Instruction ; Instruction ; Instruction ; }</pre>

Les deux grand principes: le principe de ré-utilisation

Approche bottom-up: ne pas ré-inventer la roue / éviter le copier-coller

- 2) **Principe de Ré-utilisation** pour réduire l'effort de mise au point et la taille du code en **ré-utilisant du code**.





Fonction: prototype, appel, définition

Le QUOI: But / Résumé / Contrat → déclaration du **prototype**
 Utilisation → **appel**
 Le COMMENT: détails des instructions → **définition**

```
#include <iostream>
using namespace std;

double moyenne(double nombre_1, double nombre_2);

int main()
{
    double note1(0.0), note2(0.0);
    cout << "Entrez vos deux notes : " << endl;
    cin >> note1 >> note2;
    cout << "Votre moyenne est : "
         << moyenne(note1, note2) << endl;
    return 0;
}

double moyenne(double x, double y)
{
    return (x + y) / 2.0;
}
```

prototype

appel

définition

Appel avec passage d'arguments: par valeur / par référence

Le **prototype** et la **définition** montrent les **paramètres formels** de la fonction

L'**appel** évalue les *valeurs effectives*, ou **arguments**, transmis à la fonction

Si passage par **valeur** : la valeur effective initialise le paramètre formel qui se comporte comme une variable **locale** à la fonction

Si passage par **référence**: le paramètre formel est un *alias* (un second nom) de l'argument. Une modification du paramètre formel modifie la variable indiquée comme argument.

Ex: code avec passage par valeur / somme des n premiers entiers

Cas général: qu'est-ce qu'une référence ?

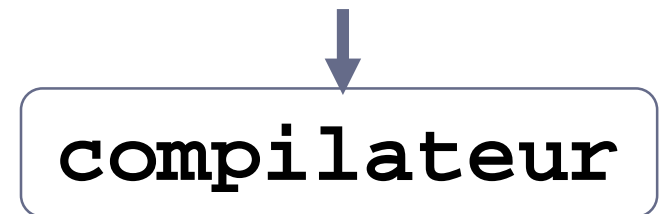
La déclaration d'une **variable** doit définir son **type** et son **nom**, encore appelé **identificateur**. Il est recommandé d'initialiser sa valeur dès la déclaration de la variable

La déclaration d'une **référence** doit indiquer à quelle **variable** elle est associée. La référence agit comme un *alias*, un *nom supplémentaire* de la variable associée.

Pour le compilateur il n'y a qu'un seul emplacement de la mémoire qui est accédé par la variable **max** et par la référence **ref**.

```
int max(100);
int &ref = max;
```

ref
max



adresse



Soyons un peu concret avec le passage par référence...

Une fonction dont le **paramètre formel** est une **référence**, comme ici avec `val` pour la fonction `f`, peut modifier la **variable transmise** comme **argument** à chaque appel de `f`.

Chaque appel transmet à la fonction `f` un moyen pour cette fonction de pouvoir accéder à la variable dont le nom est fourni en argument.

-> il s'agit de **l'adresse** de la variable qui permet d'accéder à la mémoire de manière transparente.

Dans l'exemple les 2 variables `max` et `min` sont modifiées.

```
void f(int &val);

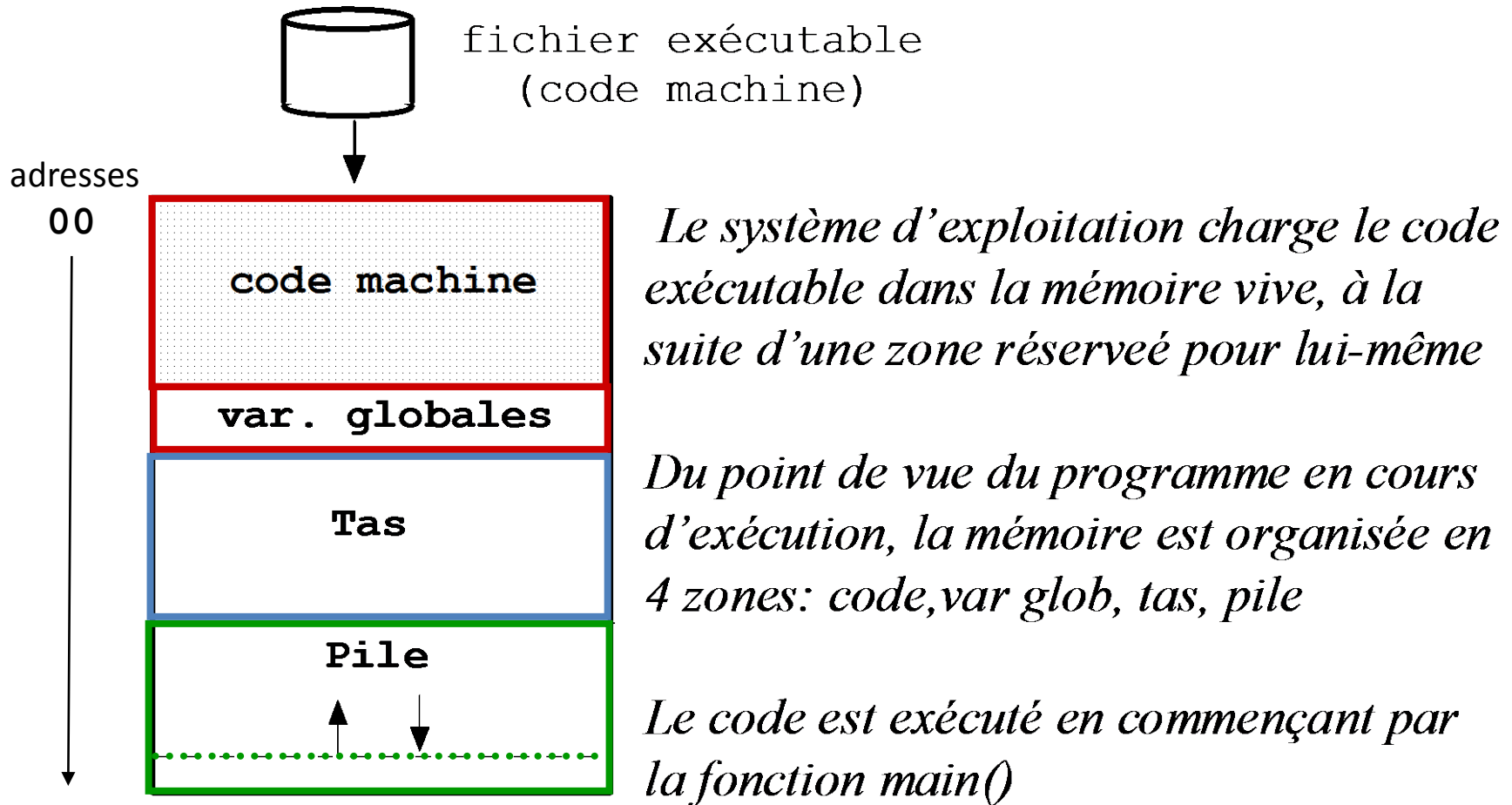
int main()
{
    int max(100);
    f(max);
    cout << max << endl;

    int min(0);
    f(min);
    cout << min << endl;
}

void f(int &val)
{
    val = 33;
}
```

Portée et durée de vie des variables locales à une fonction

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



Les variables locales à une fonction existent dans la pile

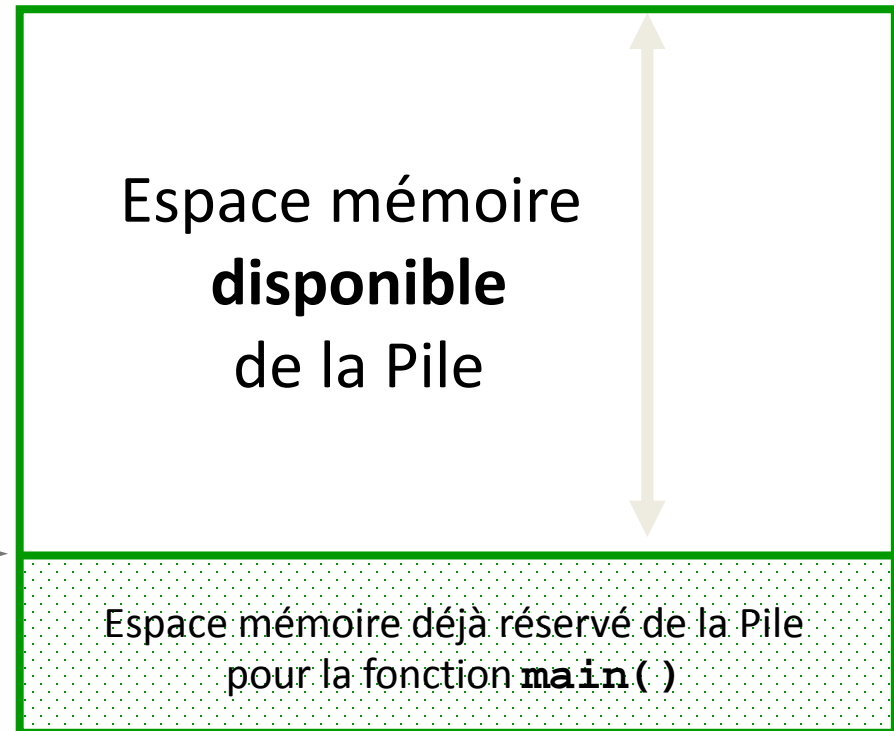
- Les 4 zones (***programme***, ***variables globales***, ***Tas (heap)***, ***Pile (stack)***) existent toujours
 - les **variables globales** et **static** existent dans une zone immédiatement après le code exécutable.
 - les fonctions travaillent avec la Pile (stack)
 - A **chaque appel (imbriqué)** 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

La fonction `main()` occupe la première tranche de la pile

Le **Pointeur de Pile** indique le sommet de la Pile: c'est le début de l'espace mémoire disponible pour le prochain appel de fonction

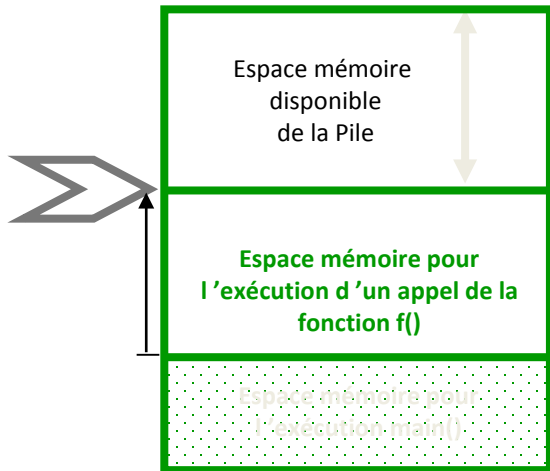


main() appelle la fonction f() qui elle-même appelle la fonction g()

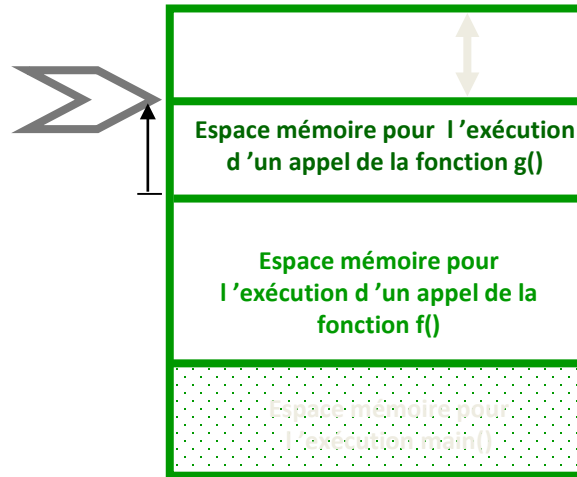
```
main()
{
  ...
  f();
  ...
}
void f(void)
{
  ...
  g();
  ...
}
...
```

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

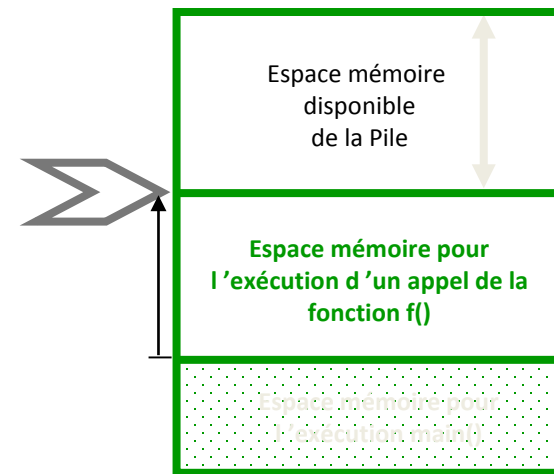
1) Appel de f()
avant l'appel de g()



2) Exécution de g()



3) Fin de l'exécution de f()
juste après l'appel de g()



Portée des variables locales limitée au bloc de la fonction

Les variables **n** et **p** de **main()** sont complètement indépendantes des variables **n** et **p** de **f**.

Elles sont créées au moment de l'appel de la fonction dans laquelle elles sont déclarées.

Chacune dispose donc d'un espace mémoire distinct sur la pile.

Il n'existe aucun lien entre elles.

```
main( )
{
    int n(0), p(0);
    ...
}

void f( )
{
    int n(-1), p(10);
    ...
}
```

A quoi sert un passage par référence `const` ?

un **appel par référence** transmet au moins **l'adresse** de la variable à la fonction.

Un **appel par valeur** transmet la **valeur** de la variable à la fonction.

Pour les type de base il n'y a aucune différence de **performance** entre ces 2 options pour l'exécution de **l'appel de la fonction**.

-> privilégier le *passage par valeur* car plus local (plus sûr).

Par contre, la transmission de la **valeur** d'un type plus lourd **T** (avec un nombre d'octets important) prend plus de temps car les octets sont copiés sur la pile **pour initialiser le paramètre formel**.

Un passage par référence devient alors intéressant en termes de **performances** MAIS, par sécurité, il faut alors ajouter **const** si on veut empêcher la modification de la variable associée à la référence.

Méthode de conception d'une fonction en 5 étapes

- 1) Le QUOI : que doit faire la fonction ? Quel est son but ?
 - Détermine son **nom**
 - Ne PAS se soucier du COMMENT à ce stade
- 2) Avec quelle matière première ? Quels paramètres ?
 - choisir un **nom** qui exprime sa nature/ son utilité
- 3) Pour chaque paramètre: doit-il être modifié ?
 - privilégier le passage par valeur / moins de risques
- 4) La fonction est-elle utilisée dans une expression ? Ex: $y = f(x)$;
 - Renvoie-t-elle une valeur?
 - Si oui, de quel type ? Si non, elle est de type **void**.
- 5) Le COMMENT : on peut commencer l'analyse fine de la fonction quand son prototype est clairement défini = son contrat avec le monde extérieur.