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

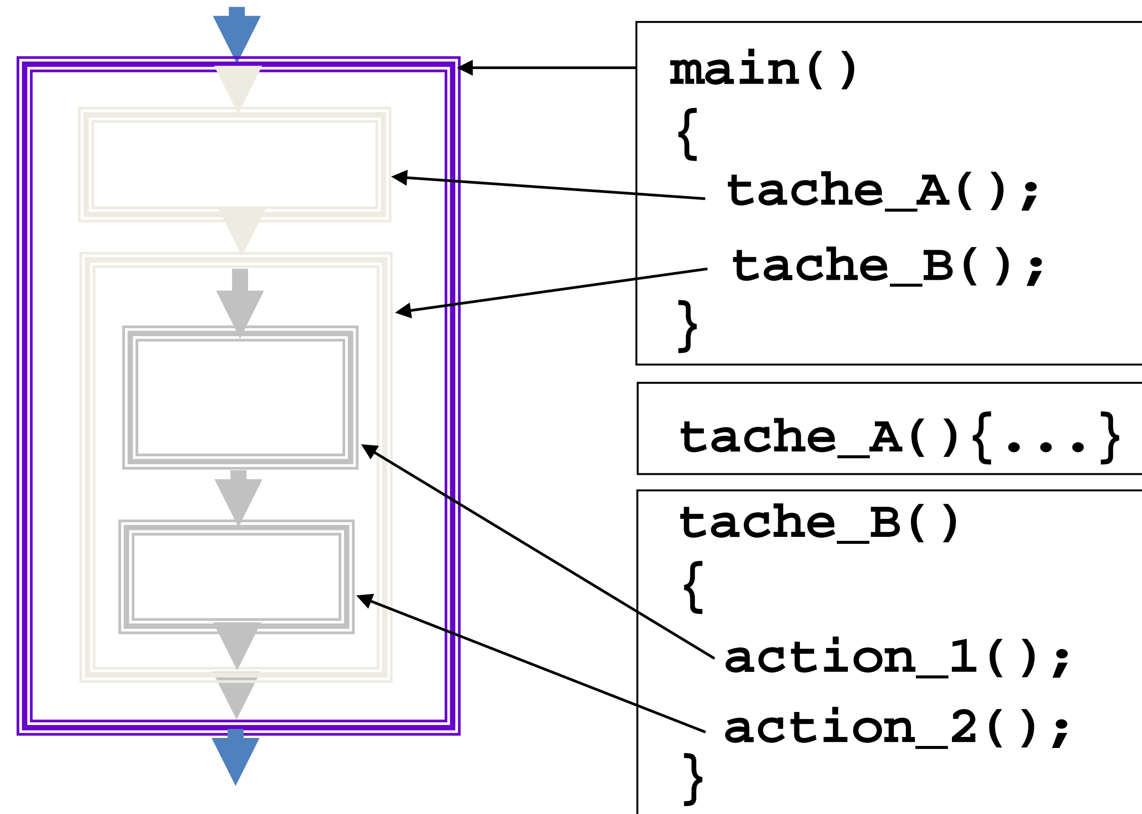
Appel avec passage d'arguments: par valeur / par référence

Portée et durée de vie des variables locales à une fonction: la Pile

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



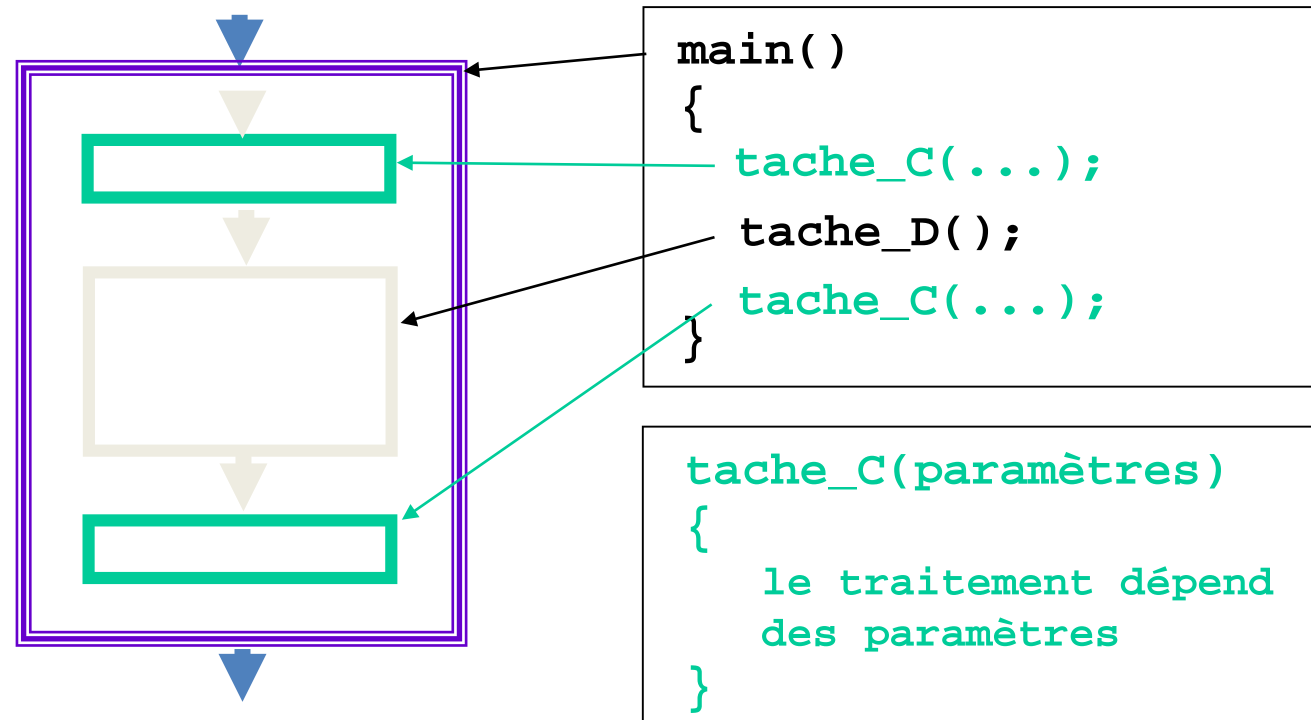
Code initial :	Code après mise en œuvre du principe d'abstraction :

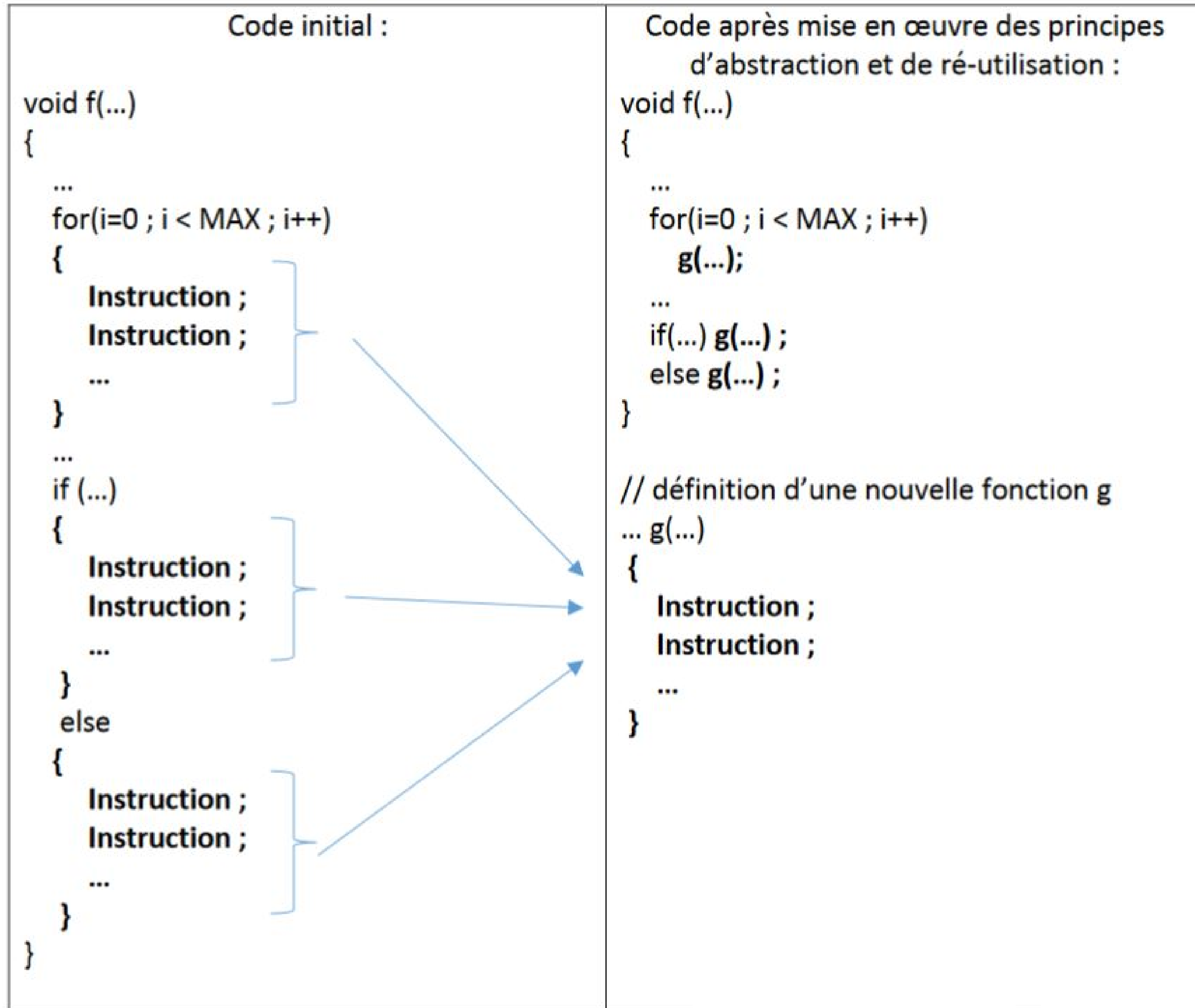
void f(...)
{
  ...
  for(i=0 ; i < MAX ; i++)
  {
    Instruction ;
    Instruction ;
    Instruction ;
    Instruction ;
    Instruction ;
  }
  ...
}
	void f(...)
	{
	  ...
	  for(i=0 ; i < MAX ; i++)
	    g(...);
	  ...
	}
	
	// définition d'une nouvelle fonction g
	... g(...)
	{
	  Instruction ;
	  Instruction ;
	  Instruction ;
	  Instruction ;
	  Instruction ;
	}

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.**




Code initial :
void f(...)
{
...
for(i=0 ; i < MAX ; i++)
{
Instruction ;
Instruction ;
...
}
...
if (...)
{
Instruction ;
Instruction ;
...
}
else
{
Instruction ;
Instruction ;
...
}
}
Code après mise en œuvre des principes d'abstraction et de ré-utilisation :
void f(...)
{
...
for(i=0 ; i < MAX ; i++)
g(...);
...
if(...) g(...) ;
else g(...) ;
}
// définition d'une nouvelle fonction g
... g(...)
{
Instruction ;
Instruction ;
...
}

Fonction: prototype, appel, définition, *passage par valeur,* return

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

Code **moyenne.cc**

```
#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

Valeurs effectives

définition

Paramètres formels

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

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

Si passage par **valeur** :

- **L’appel** évalue les *valeurs effectives*, ou **arguments**, transmis à la fonction
- Chaque *valeur effective* est **convertie** dans le type de son paramètre formel
- Chaque *valeur effective* convertie **initialise** son paramètre formel
- Le paramètre formel se comporte comme une variable **locale** à la fonction

ex: slide précédent

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.

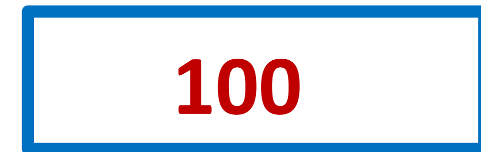
ex: slides suivants

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

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

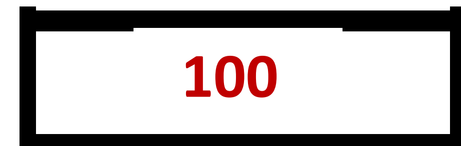
La déclaration d'une **référérence** doit indiquer à quelle **variable** elle est associée. La référence agit comme un *alias*, un *nom supplémentaire* de la variable associée.



compilateur

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`.

adresse



EPFL

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;
}
```

Code **variable_et_reference.cc**

EPFL

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**.

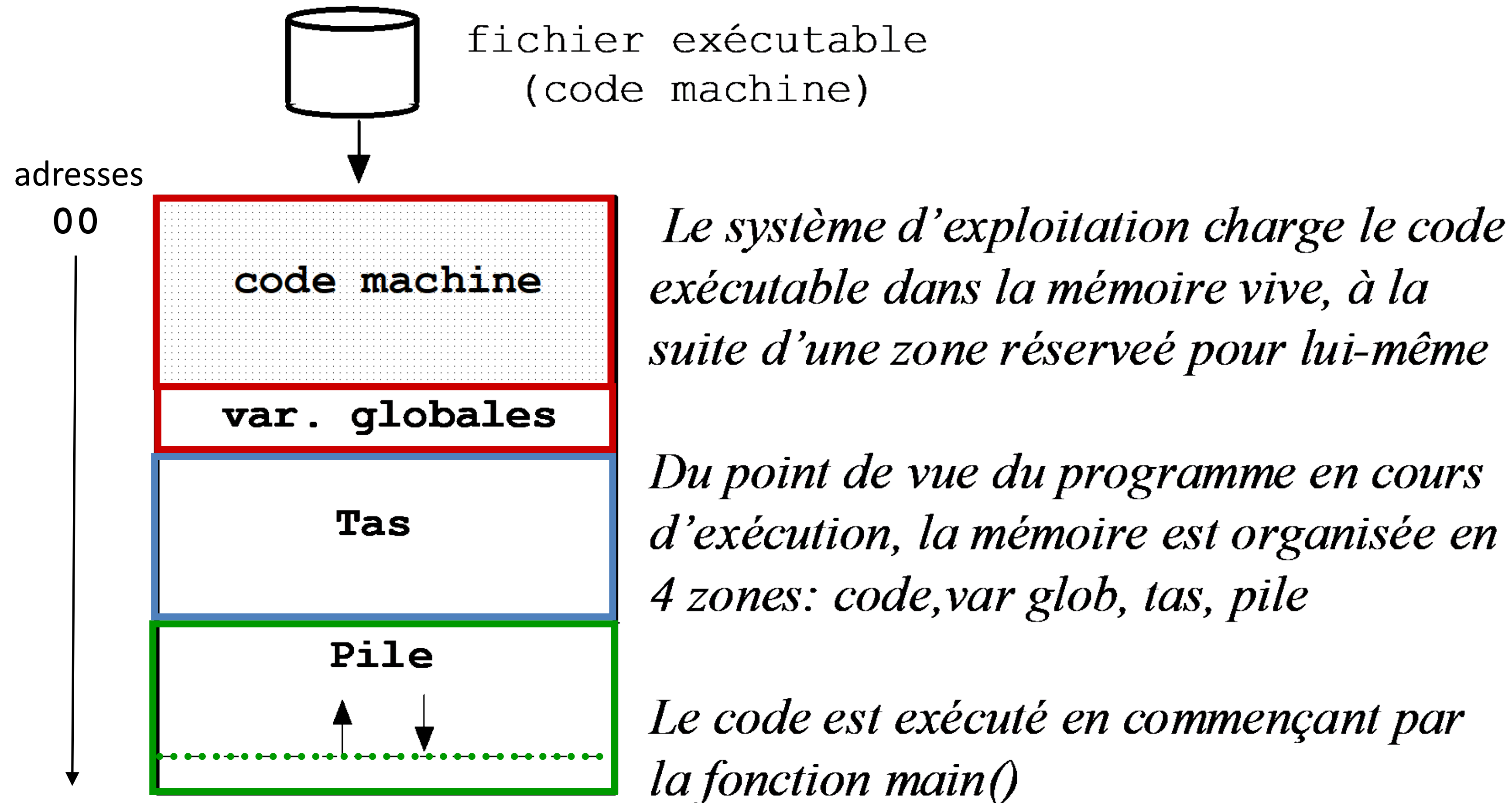
-> privilégier le *passage par valeur* car plus local (plus sûr).

Par contre, la transmission de la **valeur** d'un type **T** plus lourd (par exemple un `vector` avec un nombre d'octets important) prend plus de temps car les octets sont copiés **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.

Code **const_reference.cc**

Portée et durée de vie des variables locales à une fonction



EPFL

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** existent dans une zone immédiatement après le code exécutable.

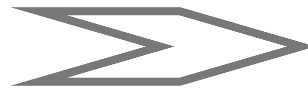
- les fonctions travaillent avec la Pile (stack)
  - A **chaque appel** (imbriqué) de fonction, une zone d **'espace mémoire** est prise **au sommet** de la pile

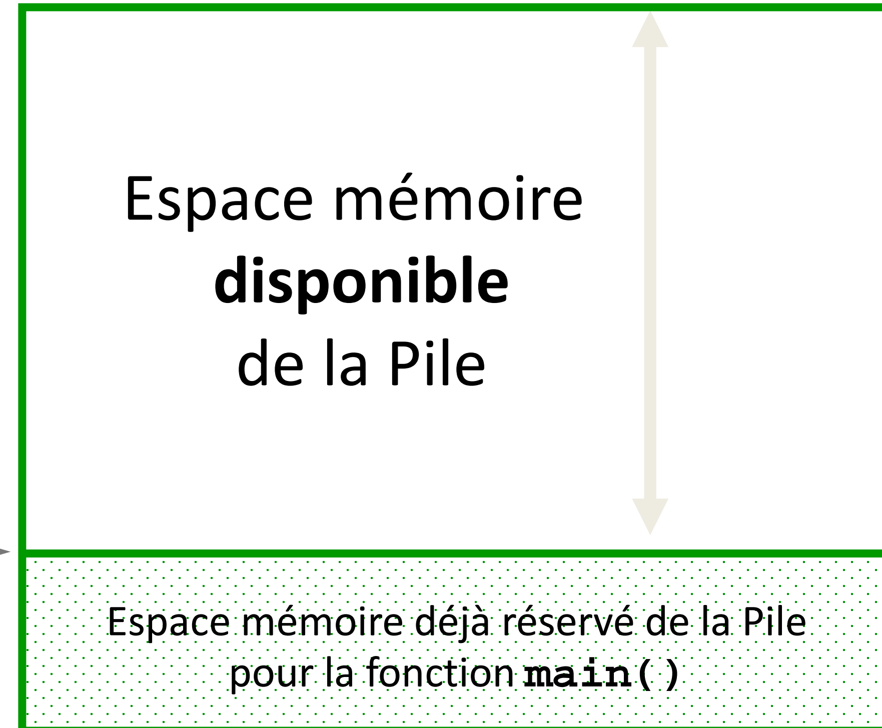
Cette zone réservée sur la pile mémorise :

- 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

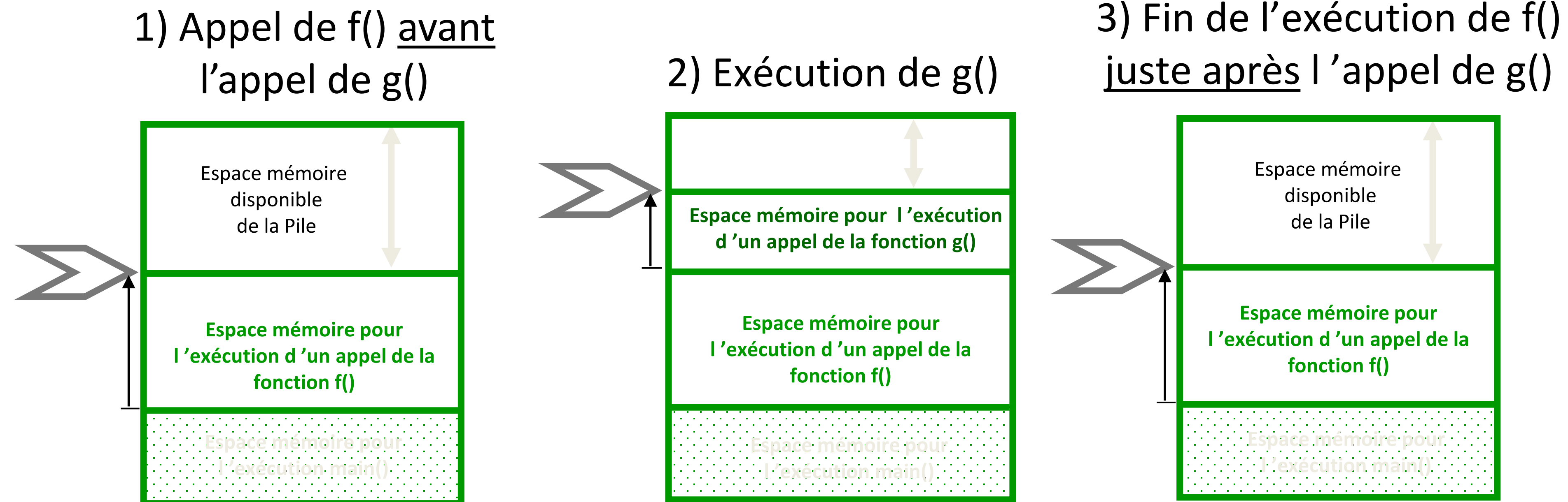




main() appelle la fonction f()
qui elle-même appelle la fonction g()

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

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



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

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(1), p(2);
   …
}

void f()
{
   int n(10), p(20);
   …
}
```

Code **variables_locales_fonctions.cc**