[Aller plus loin] Le memoire en detail

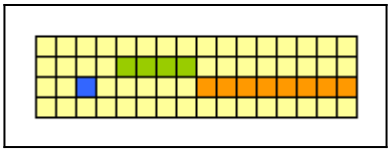
Organisation de la mémoire

Vue d'ensemble

Avant-propos
Les explications donnees dans ce chapitre sont volontairement simplifiees, pour vous aider a comprendre le fonctionnement des fonctions et le passage d'arguments. Mais l'organisation et la gestion de la mémoire est quelque chose de relativement complexe dans les details, en particulier parce que cela va dependre des interactions avec le systeme d'exploitation.

La mémoire peut être vue comme un immense tableau d'octets. Dans ce tableau, l'index est l'adresse mémoire, chaque octet peut donc être identifié de manière unique par son adresse.

Il est possible de représenter les octets sous forme d'un tableau à deux dimensions, pour gagner de la place. Par exemple, dans la figure suivante, chaque octet est représenté par une case, un objet de type `char` sera représenté par une case aussi (en bleu), un objet de type `int` sera représenté par quatre cases (en vert) et un objet de type `double` par huit cases (en orange).



Pour connaitre l'adresse memoire d'un objet (son indice dans le tableau), il faut utiliser l'operateur *address-of* `&` devant un identifiant de variable. Et pour rappel, pour connaitre la taille en memoire d'un objet, il faut utiliser l'operateur `sizeof`.

main.cpp

```
#include <iostream>

int main() {
    const int i { 123 };
    const auto address { &i };
    const auto size { sizeof(i) };
    std::cout << std::hex << std::showbase;
    std::cout << "address: " << address << std::endl;
    std::cout << "size: " << size << std::endl;
}
```

affiche :

```
address: 0x7fff1a3f51ac
size: 0x4
```

Une adresse memoire est souvent representee sous forme hexadecimale.

Attention aux syntaxes

Vous avez deja rencontre un operateur `&` dans les chapitres precedents : l'operateur ET bit-a-bit. Vous verrez egalement que cet operateur peut representer aussi les references (qui sont un type d'indirections, que vous allez voir dans la suite de ce chapitre).

Dans la meme facon, il existe l'operateur `*`, qui peut representer une multiplication, un pointeur nu (un autre type d'indirection) ou un dereferencement (qui correspond a l'operation inverse de *address-of*,

c'est a dire passer d'une adresse a une valeur).

	`&`	`*`
valeur @ valeur	ET bit-a-bit	multiplication
@ valeur	address-of	dereferencement
type @ nom	reference	pointeur

Quand vous lisez un code, il faut donc bien faire attention a ce que represente chaque identifiant (variable, type, fonction, etc).

Creation et destruction des objets

Les deux operations de base sur les objets ont deja ete vue : ce sont la creation et la destruction des objets. Comme vous l'avez deja vu, ces deux operations sont automatiques pour les variables locales, c'est donc l'approche a plus simple pour gerer les objets.

Il est egalement possible de creer et detruire dynamiquement des objets, c'est a dire lorsque la creation et destruction des objets ne peut pas etre determinee a la compilation. C'est ce que fait par exemple `std::vector` lorsque vous changer sa taille lors de l'execution ou `std::string` lorsque vous modifiez le texte.

Ces classes sont interessantes, puisqu'elles permettent de gerer la memoire dynamiquement, en conservant les avantages des variables locales pour gerer la memoire. Ce type de classe est appelle une “capsule RAII”, vous verrez cela plus en detail dans la partie sur la programmation orientee objet.

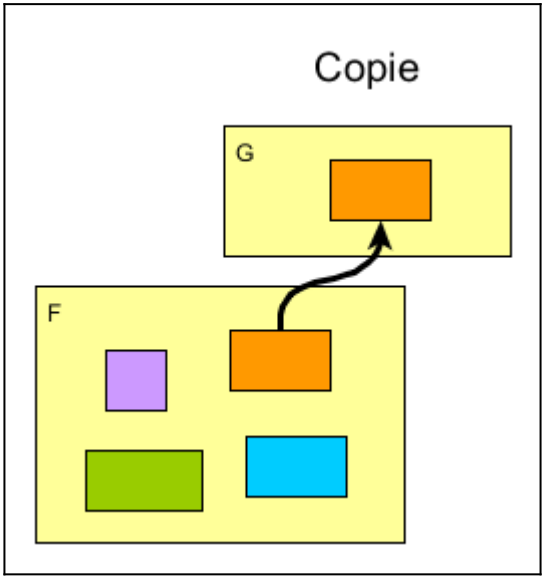
La bibliotheque standard offert de nombreuses classes RAII, et une approche moderne de la programmation C++ privilegera leur utilisation (ou la creation de nouvelles classes RAII dans le cas de fonctionnalites non fournies par la bibliotheque standard) plutot que la gestion manuelle des objets.

Copie d'objets

Chaque objet est donc represente en memoire par un bloc d'octets continues, avec une taille et une addresse. L'adresse est suffisante pour identifier chaque objet de maniere unique, il suffit donc de comparer les adresses memoires de deux objets pour savoir s'ils sont le meme objet ou non.

La copie d'un objet consiste a creer un nouvel objet, identique a un objet existant. Ce nouvel objet sera identique en memoire, sauf qu'il aura une addresse differente.

copie = création d'un objet identique en mémoire. C'est-à-dire que les octets sont identiques, mais avec une adresse différente. On obtient 2 objets valides.



Deux objets copiés sont indépendants. Si on modifie ou supprime l'un, l'autre n'est pas modifié.

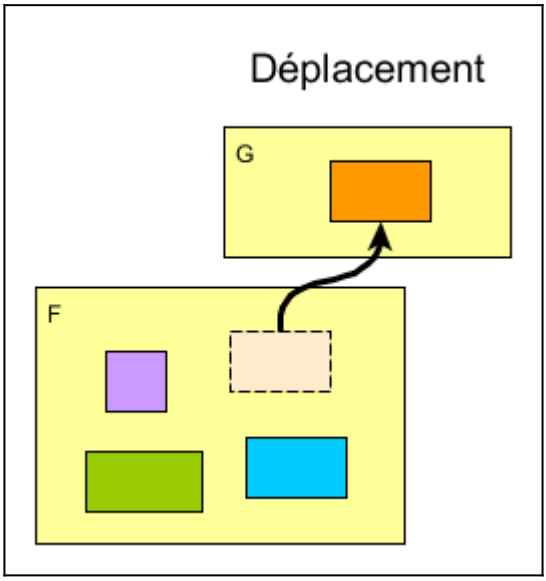
Tous les objets ne sont pas copiables. Les types fondamentaux (int, float, etc) sont par défaut copiable. Pour les objets plus complexes, cela

dépendra de la sémantique que l'on veut donner à sa classe (sémantique de valeur). La majorité des classes de la STL (Standard Template Library) ont une sémantique de valeur. (ce sera détaillé dans la partie POO).

Les déplacements

déplacement = il n'y a qu'un seul objet valide apres le deplacement. La variable initiale est invalide (en fait, elle est dans un etat indetermine, mais valide, qui autorise uniquement l'affectation ou la destruction), la variable de destination contient l'objet.

Vous ne devez plus utiliser la variable initiale apres le deplacement ! Tout ce que vous pouvez faire, c'est affecter un nouvel objet



En pratique, un déplacement peut être une copie déguisée : copie puis suppression de l'objet initial (dans ce cas, le déplacement aura le même coût que la copie). Ça sera le cas des types fondamentaux. Pour des objets plus complexes (par exemple `std::string`, `std::vector`, etc),

toutes les données ne seront pas copiée. (dans ce cas, le déplacement sera beaucoup moins couteux que la copie).

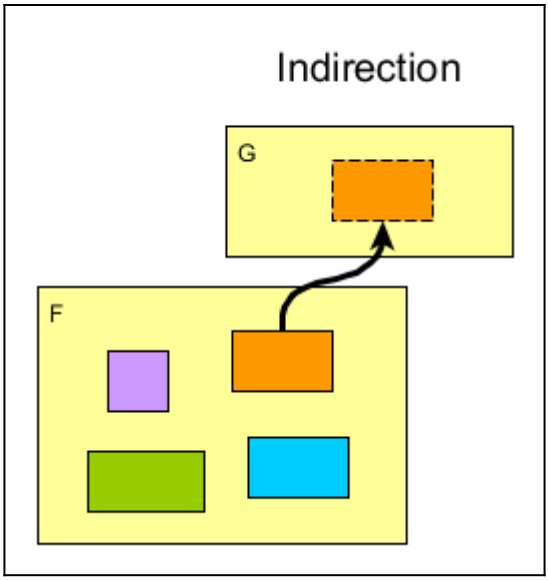
```
void g(int j) {}

void f() {
    int i { 123 };   // non const, le déplacement est une
modification
    g(std::move(i)); // l'objet contenu dans la variable i
est déplacé dans
                     // la variable j de la fonction g, i
devient invalide
}
```

Concerne aussi la sémantique de valeur. Mais indépendant de la copie (c'est-à-dire un objet peut être copiable et movable, copiable et non movable, ou non copiable et non movable)

indirections

Indirection = “objet” qui permet d'accéder à un autre objet à distance. Utiliser une indirection revient à utiliser l'objet qui est lié à l'indirection.



Si A est un objet et si B est une indirection sur A, alors utiliser B revient à utiliser A.

Exemple d'indirection déjà vu : les itérateurs. Un itérateur est une indirection sur un élément d'une collection. Accéder à l'itérateur revient à accéder à l'élément.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = std::begin(v);
    std::cout << (*it) << std::endl;
    (*it) = 5;
    for (auto i: v) { std::cout << i << ' '; }
    std::cout << std::endl;
}
```

affiche :

```
1
```

```
5 2 3 4
```

`it` permet d'accéder au premier élément du tableau.

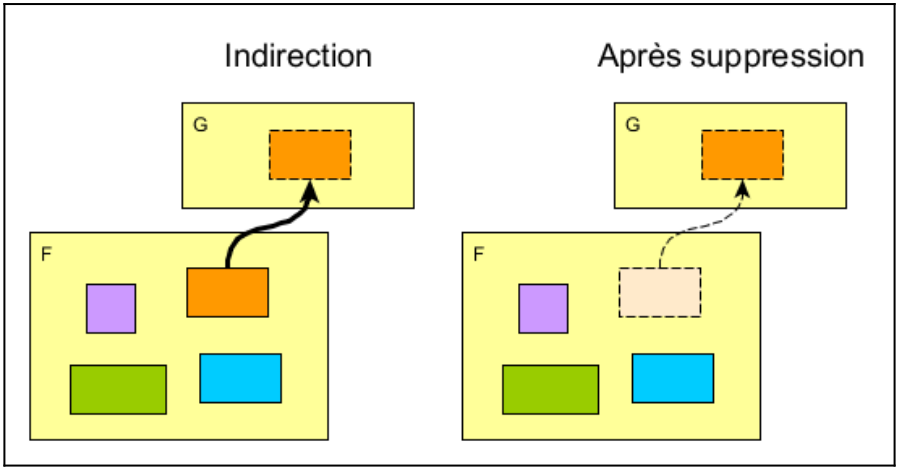
Validité d'une indirection

Le problème de la validité d'une indirection a déjà été abordé pour les itérateurs. Pour rappel :

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v { 1, 2, 3, 4 };
    auto it = std::begin(v);
    std::cout << (*it) << std::endl;
    v.clear();
    std::cout << (*it) << std::endl;
}
```

Après l'appel à `clear`, l'élément correspondant à l'itérateur n'est plus valide et utiliser `it` produite donc un comportement indéterminé.



Et retenez surtout qu'un “undefined behavior” ne produit pas forcément un crash (c'est le cas de ce code, il semblera valide à l'exécution, mais pourra faire n'importe quoi) et ne produira JAMAIS d'erreur de compilation.

Ce qui est valide pour l'itérateur est valide pour toutes les indirections : il faut que l'objet correspond à l'indirection soit valide, sinon l'indirection est invalide et l'utiliser produit un “undefined behavior”.

Types d'indirection

classification :

- peut être null ou non (la référence doit toujours être valide)
- managé ou non = responsable de détruire l'objet (référence = non managé)
- affectable ou non (référence = non affectable)
- cas particulier des itérateurs (indirection sur élément d'une collection)

Il n'y a pas besoin de voir en détail toutes les indirections, certaines sont moins utilisées et seront vues dans les chapitres complémentaires.

- référence (lvalue et rvalue ?)
- `std::reference_wrapper` (affectable)
- itérateur = indirection sur collection
- pointeur = peut être null
- pointeur intelligent = managé (partagé = `std::shared_ptr` ou non = `unique_ptr`), pointeur nu = non managé

Note sur les indirections en C et C++

Même si le C et le C++ sont deux langages différents, ils partagent une origine commune (d'où leur noms qui sont proches et des syntaxes que

l'on peut retrouver dans les deux langages) et surtout, il est courant d'utiliser des bibliothèques écrites en C dans un programme C++.

Malgré cette origine commune, ces deux langages ont des mécanismes très différents, en particulier concernant la gestion de la mémoire. Mais dans ce chapitre, c'est un autre point qui va être abordé : les indirections.

Le langage C propose qu'un seul type d'indirection, les pointeurs null, alors que le C++ en proposent différents types (voire virtuellement une infinité, puisqu'il est possible de définir des classes implémentant des indirections, comme c'est le cas des itérateurs par exemple).

Cela peut sembler être une difficulté du C++ par rapport au C, puisque cela nécessite d'apprendre plusieurs syntaxes et concepts, là où il n'en faut qu'un seul en C. En fait, chaque type d'indirection apporte des fonctionnalités et garanties différentes, ce qui peut simplifier le code. Il est donc important de bien comprendre les spécificités de chaque type d'indirections et de les utiliser correctement.

Au contraire, en C, les pointeurs null seront utilisés comme indirection sur un objet, comme indirection sur un élément d'un tableau ou pourront même représenter un tableau. Il n'est pas possible de savoir ce que l'on peut faire ou non avec un pointeur uniquement en regardant son type (idem pour le compilateur, ce qui veut dire qu'il ne peut rien vérifier), il faut regarder le contexte d'utilisation du pointeur pour savoir comment l'utiliser.

Un dernier point sur les indirections : il est courant de trouver dans des codes C++ (sur internet ou dans de vrais projets) qui sont en fait des codes C (généralement du C++ écrit en utilisant les pratiques du C). Dans ces codes, vous verrez régulièrement des pointeurs null, puisque c'est le seul type d'indirection disponible en C. Cependant, en C++, les pointeurs null sont le type d'indirection qui apportent le moins de garanties et leur utilisation doit donc être limité aux profits des autres types d'indirection (en premier lieu les références).

Faites bien attention d'identifier les codes C++ problématiques et corrigez les si nécessaire.

Fonctions

Pourquoi présenter les indirections dans la partie sur les fonctions ? Les indirections sont utilisables en dehors des fonctions (en particulier les itérateurs), mais elles sont particulièrement intéressantes avec les fonctions.

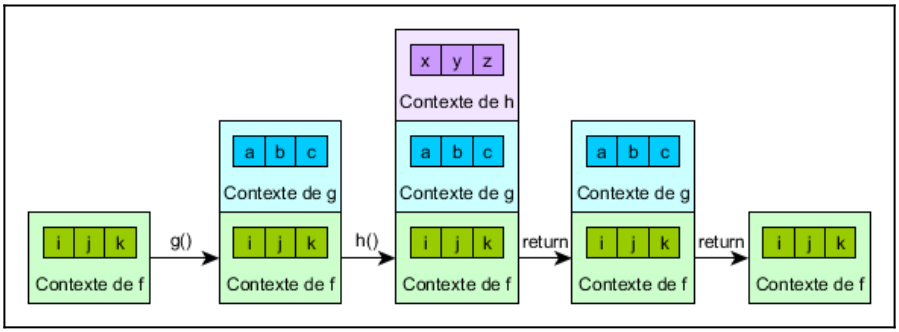
On va se limiter à un type d'indirection : les références.

Pile d'appel de fonction

Explications simplifiées pour comprendre la mécanisme. *Stack* en anglais.

Lorsqu'une fonction est appelée (cas particuler pour la fonction main qui est appelée par le système), cela créé en mémoire un contexte pour cette fonction. Ce contexte contient les informations nécessaires pour l'exécution de la fonction, en particulier la mémoire correspondant aux variables locales de la fonction (ainsi que les paramètres d'appel, que l'on peut considérer comme des variables locales).

```
void f(int i) {
    int j {};
    int k {};
}
```

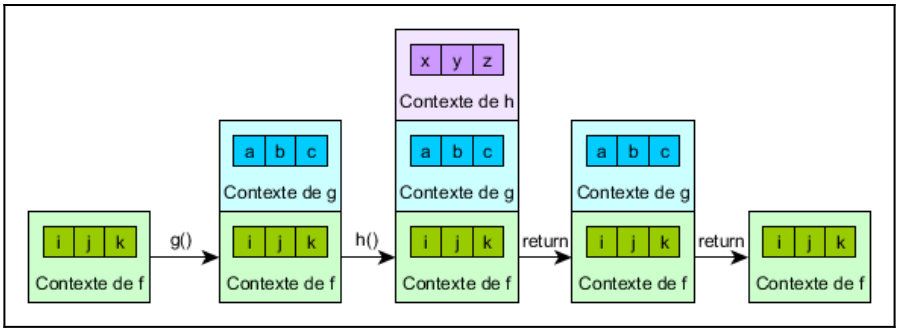


A chaque fois qu'une fonction est appelée, le nouveau contexte “s'empile” sur les précédents (on parle de “Pile d'appel des fonctions”).

```
void h() {
    double x {};
    double y {};
    double z {};
}

void g() {
    bool a {};
    bool b {};
    bool c {};
    h();
}

void f() {
    int i {};
    int j {};
    int k {};
    g();
}
```



Lorsque f est exécutée, la pile contient le contexte d'exécution de f. Après l'appel de g, la pile contient f et g, puis après l'appel de h, la pile contient f, g et h. Après le retour de h, la pile contient à nouveau f et g, et après le retour de g, la pile contient f.

Note : mémoire = très gros tableau d'octets. l'“empilement” est

conceptuel, il n'y a pas de notion de “dessus” et “dessous” dans une mémoire.

Contexte d'une fonction

Le contexte d'une fonction est l'ensemble des variables qu'une fonction peut utiliser.

```
void g() {
    i = 123; // erreur
}

void f() {
    int i {};
    g();
}
```

Exception :

- variables globales, accessibles partout. A éviter (complexifie la lecture du code)
- les littérales (il faut bien qu'elles soient quelque part dans la mémoire. Elles sont dans un espace dédié, le data segment. Et il y a un segment spécifique pour le texte. Mais tout cela est géré en interne par le compilateur, pas besoin de s'en préoccuper).

Taille limite de la Pile

Il existe une limite à la taille de la pile (par exemple 1 Mo avec MSVC). Cela dépend des configurations et du système. Mais si vous essayez de créer un milliard de `int` dans une fonction, cela va produire très certainement un dépassement de pile. (erreur de type “Segmentation fault”).

2 conséquences :

Appeler une fonction consomme un peu de mémoire sur la pile (même si la fonction n'a pas de variable locale). Appelle récursif illimité finira donc pas saturer la pile et produire un crash.

Parfois (souvent) on a besoin d'utiliser plus que 1 Mo de mémoire (penser à une simple image en mémoire, cela peut prendre plusieurs Mo). Il existe un second espace mémoire, le Tas (ou mémoire dynamique), qui contient tout ce qui n'est pas sur la Pile.

Différences Pile/Tas

- Pile = taille très limité, Tas = beaucoup plus de mémoire (en première approximation = infini... en vrai = taille limité, espace libre généré par le système et peut changer avec le temps = besoin de gérer si la mémoire est dispo ou non)
- Pile = généré automatiquement lors des appels et retour de fonction, Tas = à gérer par le code

Cas particulier des objets complexes comme `std::vector` et `std::string`. Ils sont en fait constitués de plusieurs parties :

- une petite partie, de taille fixe, qui sera sur la Pile par défaut
- les données proprement dites (les éléments ou le texte), qui sont sur le Tas
- des indirections depuis la partie sur la Pile vers la Partie sur le Tas

image d'un vector

Des collections plus complexes (`std::list` par exemple) peuvent contenir plusieurs objets différents en mémoire, liés entre eux par des indirections.

En pratique, il n'y a pas de limite au nombre de sous-objets que peut contenir un objet complexe, et il peut y avoir des organisations très complexe en mémoire.

Note sur `std::array` : toutes les données sont sur la Pile, il y aura donc un problème si tableau est trop grand. Il est préférable d'utiliser `std::vector` (ou dynarray) si c'est le cas.

Remarque sur vector et copie/déplacement. Quand on copie, cela copie toutes les données. Quand on move, cela copie simplement la petite

partie sur la Pile : c'est plus performant. (Mais le plus performant reste d'utiliser une indirection).

Déplacer une partie des explications dans un chapitre complémentaire

Passage de valeurs

Contextes de fonctions indépendants = pas possible d'accéder directement à une variable. Comment faire ?

```
void g() {
    i = 123; // erreur
}

void f() {
    int i {};
    g();
}
```

On copie l'objet depuis le contexte de f dans le contexte de g. Il est alors possible d'utiliser cet objet dans g.

La syntaxe est celle déjà vue :

```
// déclaration
void f(TYPE NOM)

// appel
f(VALEUR)  // littérale, expression, variable
```

Par exemple :

```
void g(int i) {
    i = 123;  // ok
}

void f() {
```

```
    int i {};
    g(i);
}
```

L'objet est perdu à la fin de la fonction (sauf si retourné), donc les modifications faites sont perdues.

```
void g(int i) {
    i = 123;  // ok
}

void f() {
    int i {};
    g(i);
    std::cout << i << std::endl; // affiche 0
}
```

différence entre copie et déplacement ? (selon le type de value, rvalue ou lvalue, `std::move`)

retour de valeur

Passage par référence

Indirection sur l'objet de `f` dans `g`. Chaque indirection a une syntaxe spécifique. Pour la référence, 2 types : const ou non.

```
// déclaration
void f(TYPE const& NOM)  // const
void f(TYPE      & NOM)  // non const

// appel
f(VALEUR)  // littérale, expression, variable
```

const = on ne peut pas modifier le paramètre, non const = on peut.

Dans tous les cas, ce n'est pas la référence que l'on manipule, mais c'est en fait l'objet dans `f`. En particulier, si on fait une modification dans `g`, elle sera visible dans `f`.

```
void g(int & i) {  // on modifie i donc non const
    i = 123;  // ok
}

void f() {
    int i {};
    g(i);
    std::cout << i << std::endl; // affiche 123 !
}
```

Note : une indirection prend un peu de place dans la Pile. Environ le même cout qu'un type fondamental (int, float, etc)

Conclusion : que faut-il utiliser ?

- pour éviter la copie = indirection. Si la copie n'est pas couteuse (types fondamentaux) ou si on a besoin d'une copie (si on veut faire des modifications sans les conserver)
- pour modifier un objet = référence non const
- par défaut (si on ne sait pas si la copie est couteuse) = référence constante.

Compatibilité entre paramètres et arguments

- lvalue : variable...
- rvalue : temporaire (littérale, expression)...

- passage par valeur et référence constante : accepte lvalue et rvalue
- passage par référence : accepte lvalue uniquement
- passage par rvalue référence : accepte rvalue uniquement

```
void f(int x);
voif g(int const& x);
void h(int & x);
void i(int && x);
```

```
int x {};  // lvalue
f(x);      // ok
g(x);      // ok
h(x);      // ok
k(x);      // erreur

           // rvalue
f(1);      // ok
g(1);      // ok
h(1);      // erreur
k(1);      // ok
```

rvalue, lvalue, xvalue, prvalue, glvalue...

Valeur par défaut

Selon le type de passage. Idem que les arguments, nécessite correspondance.

```
void f(int x = 123);   // ok, rvaleur vers value
voif g(int const& x = 123);  // ok, rvalue vers ref const
void h(int & x = 123);   // erreur
void i(int && x);    // ?
```

Paramètres de retour

Ne pas retourner une indirection sur un variable locale, qui sera détruite à la fin de la fonction (donc indirection sur variable invalide).

(N)RVO : optimisation pour retour, le plus efficace. + conditions d'utilisation.

```
int g() {
    int i;
```

```
    return i;
}
```

Par valeur, mais pas de copie ici, optimisé. Note : utilisation de déplacement = désactive optimisation, moins performant.

+ cas particulier ds fonctions pure

```
int g(int i) {
    ...
    return i;
}

// alternative à

void g(int & i) {
    ...
}
```

La piles d'appel des fonctions

Pile stocke les informations lorsque l'on entre dans une fonction, dans *Stack frames*. Permet de stocker

- les arguments de fonction
- les variables locales
- divers informations utiles (retour d'appel de fonction)

Pile “dernier entré, premier sorti” (*Last In, First Out*, LIFO). Similaire à une pile de pièces empilées. On peut ajouter une pièce sur le dessus de la pile et retirer la pièce qui se trouve au dessus.

Déroulement d'un programme :

```
void f() {
}

void g() {
    f();
```

```
}

int main() {
    g();
}
```

1. entrée dans main. Pile = main
2. appel de g() - Pile = main-g
3. dans g, appel de f. Pile = main-g-f
4. sortie de f, retour dan g. Pile = main-g
5. sortie de g, retour dans main. Pile = main

Avec des variables locales

```
void f() {
     int j { 456 };
}

void g() {
    int i { 123 };
    f();
}

int main() {
    g();
}
```

Pile :

1. main
2. main - g+i
3. main - g+i - f+j
4. main - g+i
5. main

Il est possible de manipuler directement les informations dans la Pile, mais le fonctionnement exact dépend du système, du compilateur et des options de compilation. Nécessite d'écrire un code spécifique pour chaque cas possibles et de bien connaitre la mécanique interne du fonctionnement des programmes, cela sort du code de ce cours.