

B3 - Bas Niveau C++

Session 1 - Gestion de la mémoire

Par Julien SOSTHENE - 2020

Sommaire

1. Introduction du module
2. Setup
3. Organisation de la mémoire
4. Gestion de la mémoire en C++
5. Smart pointers
6. Move Semantics en C++

1. Introduction du module

1.1 Votre intervenant

- Julien SOSTHENE
- Ingénieur, spécialisé en ingénierie logicielle
- Travaille principalement sur du Jeu vidéo et du Web
- Mentor sur OpenClassrooms
- Très intéressé par le "comment?" et le "pourquoi?"
- Touche à tout!

1.2 Vue d'ensemble

- **Nombre d'heures présentielles : 24**
 - 8h, 8h, 4h, 4h
- **Sujets abordés**
 - Gestion des ressources et move semantics
 - Principes d'héritage avancés
 - Gestion d'exceptions
 - TDD en C++
 - Multithreading

1.3 Evaluation

- 2 Quiz de connaissances
- 1 TP à finir et à rendre en groupes de 2

1.4 Pré-requis

- Notions de C++ basiques
- Connaissances basiques sur le fonctionnement matériel d'un ordinateur
- Notions d'Orienté Objet
- Utilisation d'un debugger

2. Setup

2.1 Environnement de développement

- Editeur : VSCode
- Compilateur : G++
- Debugger : GDB

💡 Si vous avez l'habitude d'autres outils, vous pouvez les utiliser!

2.2 Visual Studio Code

- Installez Visual Studio Code
- Vérifiez/Effectuez l'installation de l'**extension C/C++ officielle**

Linux

- Installez G++ (suite GCC) et GDB si ce n'est pas fait

Windows

- Installez Cygwin avec les paquets `gcc-core` , `gcc-g++` et `gdb` (dernières versions stables)
- Ajoutez le répertoire `\bin` de Cygwin à votre `Path`

Mac

- Xcode doit suffire

💡 Visual Studio Community aura ce genre d'outils par défaut.
Nous optons ici pour une approche plus légère et cross-platform.

2.3 Test

- Relancez VSCode
- Créez un programme basique C++ tel que :

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

- Lancez **Debug** -> **Start Debugging**

- Choisissez `C++ (GDB/LLDB)` puis `g++.exe build and debug active file` dans les menus qui s'ouvrent alors

 **Vous êtes prêts à suivre les manipulations du module!**

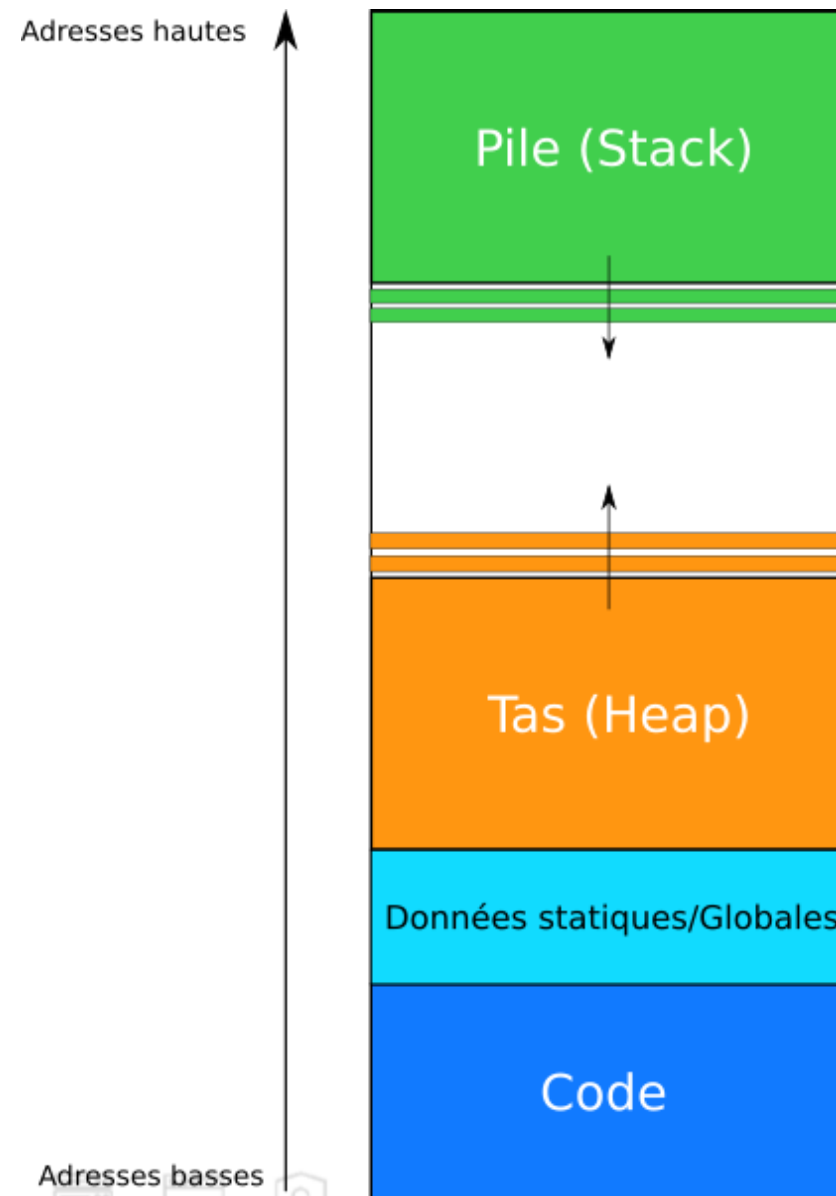
3. Organisation de la mémoire

3.1 Organisation générale

La mémoire est organisée dans un **espace d'adresses virtuelles** contigu

Le code est chargé en mémoire (instructions) ainsi que les données statiques ou globales nécessaires

La **mémoire dynamique** est organisée entre *pile* et *tas*.



- 💡 L'espace d'adresses est **virtuel** et spécifique au processus!
- ⚠️ La pile est à l'envers: elle s'empile **vers le bas**!
- 💡 Le tas s'agrandit vers le haut au fur et à mesure qu'on alloue de la mémoire.

3.2 Fonctionnement de la pile (stack)

La pile fonctionne sur un principe de *frames*.

En C/C++, on a une frame par fonction, qui représente:

- Les arguments de la fonction
 - Les variables locales de la fonction
- 💡 La taille d'une frame est déterminée par le compilateur

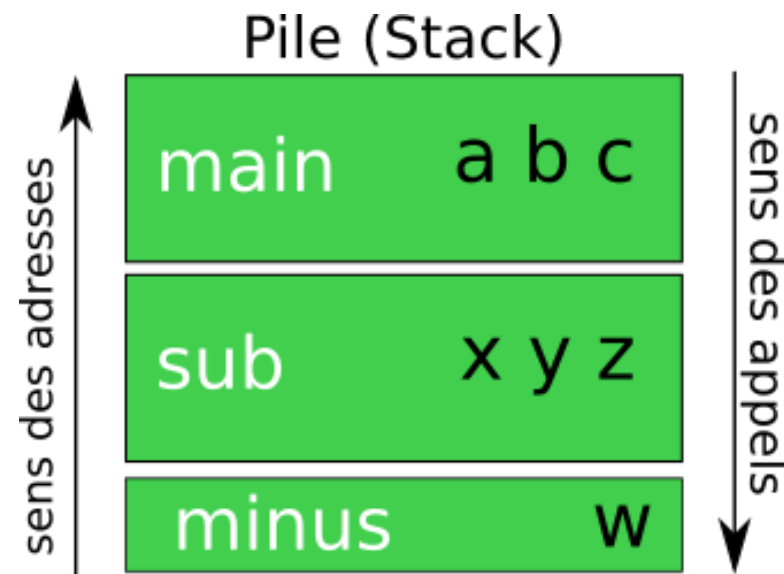


```
#include <iostream>

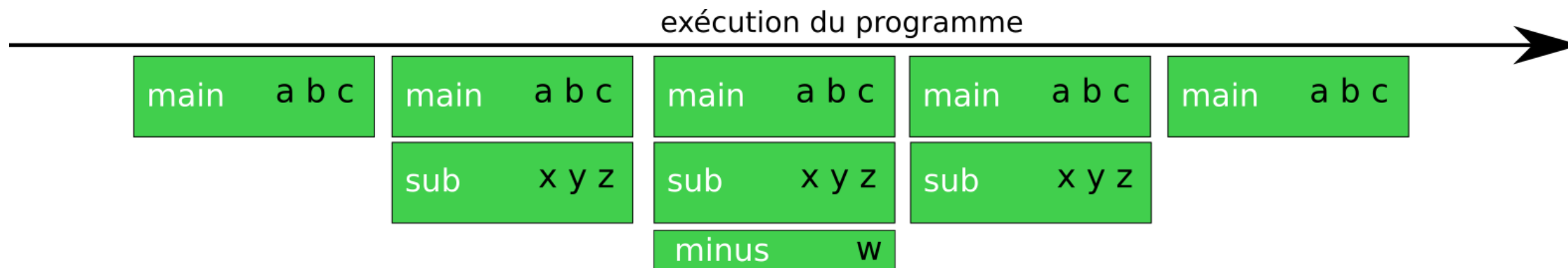
int minus(int w){
    return -w;
}

int sub(int x, int y) {
    int z = minus(y);
    return x + z;
}

int main () {
    int a = 3;
    int b = 4;
    int c = sub(b,a);
    std::cout << "c vaut " << c
                << std::endl;
    return 0;
}
```



Dans ce simple programme, la pile évolue comme suit:



⚠ Le programme peut être largement optimisé par le compilateur. Ici, en mode debug, ce n'est pas le cas!

3.3 Testons ceci avec VSCode et GDB!

- VScode nous affiche le contenu de la frame la plus basse qui correspond au point d'arrêt actuel
- Dans la debug console, nous pouvons aussi utiliser les commandes GDB grâce à

```
-exec <commande gdb>
```

3.4 Quelques commandes GDB utiles

- `info frame` affiche les informations de la frame en cours
- `up` remonte d'une frame dans la pile
- `down` descend d'une frame dans la pile
- `p <variable>` ou `print <variable>` : affiche la valeur d'une variable donnée
- `p &<variable>` affiche l'adresse mémoire de la variable dans la pile
- `x/16x <adresse mémoire>` affiche 16 mots de 4 octets à partir d'une adresse
- `disas <fonction>` ou `disassemble <fonction>` affiche le code assembleur de la fonction

3.5 Observons la mémoire de notre programme!

Placez un breakpoint dans la fonction la plus "profonde" en termes d'appels et observez les différentes frames grâce à GDB!

3.6 ⚠ à retenir

- La taille de la pile est limitée (pour des raisons de cache CPU, de sécurité, de contiguité en mémoire physique)
- La taille du tas est virtuellement aussi grande que votre RAM (nous y reviendrons)

Test 0

Dans notre test précédent, passons une variable d'une fonction à une autre.

Observez les adresses mémoires de l'entier en question dans la frame inférieure et dans la frame supérieure!

⚠ Par défaut, les variables passées en argument "telles quelles" sont copiées en mémoire! ⚠

Test 1

Passez un gros tableau en argument d'une fonction et observez sa frame!

Vérifiez l'adresse mémoire du tableau dans la frame supérieure.

⚠ Certains types, comme les tableaux, sont automatiquement passés *par référence*! ⚠

Elles ne sont alors pas copiées d'une frame à l'autre: seule leur adresse mémoire l'est.

C'est parce qu'un tableau est concrètement un pointeur vers son premier élément.

💡 Si nous modifions une des données de notre tableau dans la fonction, le tableau est modifié dans la frame supérieure



Test 2

- Créez une classe simple et passons une instance en argument, par valeur.
- Observez l'argument et la valeur d'origine dans la frame supérieure

⚠ Les instances de classe peuvent être passées par valeur! ⚠

Leurs valeurs sont copiées d'une frame à l'autre.

- 💡 Si l'objet est gros, il va être lent à copier d'une frame à l'autre.
- 💡 De plus il va prendre beaucoup de place dans une pile limitée!

Test 3

- Quid d'une classe ayant une propriété tableau?
- Faites le test!

⚠ Cette fois nos deux tableaux ont une adresse différente; le tableau a été copié! ⚠

3.7 Passer des arguments par référence

Dans ce cas, mieux vaut passer notre instance *par référence*

Une **référence**, c'est l'adresse mémoire d'un élément de la pile, il suffit d'ajouter **&** :

```
int minus(int& w){  
    return -w;  
}
```

```
minus(a);
```

 Observez à présent la valeur de l'argument depuis GDB!

! Référence vs Pointeur

Attention au vocabulaire!

Le pointeur est la *valeur* d'une référence!

```
int m = 4;  
int *n = &m;  
n = nullptr;
```

- `n` est un pointeur; il peut être `NULL` (adresse)
- `&m` est une référence; il ne peut pas être `NULL` (adresse d'une entité concrète qui existe actuellement)

⚠ Lors d'un passage de référence, on manipule l'objet comme son original et non comme un pointeur.

☞ On accède aux propriétés des références d'objet avec `.` et non `->` !

⚠ Attention:

```
int minus(int& w){
    return -w;
}
```

```
minus(10); //ERREUR à propos d'une "lvalue". A suivre!
```

3.8 Fonctionnement du tas (Heap)

Le tas est un autre espace mémoire à la gestion plus libre

- Il est fourni à la demande à notre programme.
- Les zones attribuées doivent être libérées explicitement
- Il est *virtuellement* aussi grand que la RAM disponible
- La taille à allouer n'a pas à être connue à la compilation du programme
- Le tas entier est libéré à la fermeture du programme.



On considère la classe suivante:

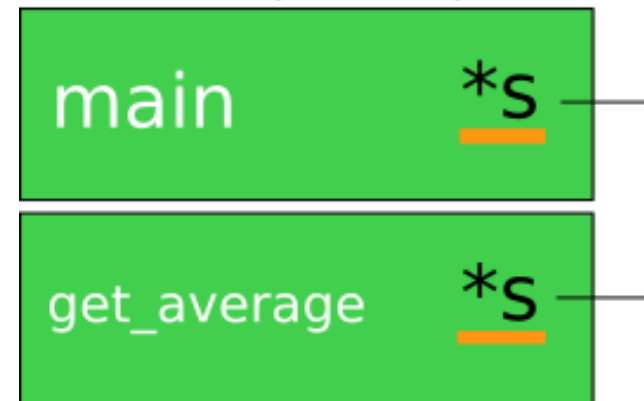
```
class Student {  
    public :  
        int grades[3];  
        Student(int grade1, int grade2, int grade3) {  
            this->grades[0] = grade1;  
            this->grades[1] = grade2;  
            this->grades[2] = grade3;  
        }  
};
```

```
#include <iostream>

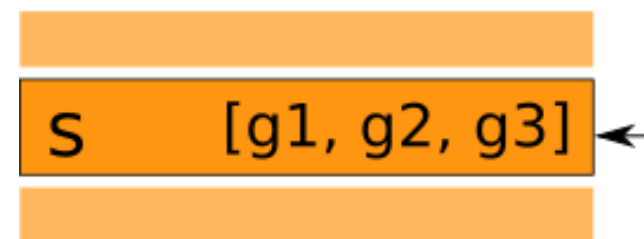
float get_average(Student* s) {
    float result = 0;
    for(int i = 0; i < 3; i++){
        result += s->grades[i];
    }
    return result / 3.0;
}

int main () {
    Student* s = new Student(12,16,14);
    std::cout << get_average(s)
                << std::endl;
    return 0;
}
```

Pile (Stack)



Tas (Heap)



- En C, on alloue de la mémoire du tas avec `malloc` , `calloc` ou `realloc`
- En C++, on le fait avec `new` !
- La valeur reste allouée tant qu'on ne la libère pas
 - La mémoire allouée avec `malloc` se libère avec `free`
 - La mémoire allouée avec `new` se libère avec `delete` (et `delete[]`)

! Danger Zone!

- La mémoire du tas ne se désalloue pas toute seule; pour chaque `new` doit suivre un `delete`, sinon on crée une fuite mémoire! 💧 💧 💧
- Les pointeurs, eux, stockés dans la pile, disparaissent, mais ça ne veut pas dire que la mémoire du tas n'est plus là!
- Si on essaie d'accéder à la mémoire d'un pointeur désalloué, on crée une *Segmentation Fault*, un appel mémoire non autorisé. C'est le crash! 💣

💡 Le système d'exploitation empêche les processus d'avoir accès à la mémoire des autres processus (ou du système même). Vous ne pouvez accéder qu'à de la mémoire que vous avez vous-même allouée.

Sauf quand c'est possible! (Cf. les failles Meltdown et Spectre 🤖)

3.9 Coût du tas

👉 Coût humain: vous êtes responsables de la mémoire! Une erreur est vite arrivée!

Une grande partie des failles de sécurité trouvées aujourd'hui dans les langages système proviennent d'un problème de gestion mémoire



Vulnérabilités communes 🐛

- utilisation après `free` / `delete` (use after free)
- déréférencement d'un pointeur `NULL`
- utilisation de mémoire non initialisée (valable avec la pile, mais plus rare)
- double `free` / `delete` (double free)
- buffer overflow (valable avec la pile)

👉 à chaque `new` ou `malloc`, une petite mécanique s'enclenche autour d'un appel système.

Le programme demande une plage mémoire au système, qui lui réserve la plage et lui fournit un pointeur.



Démonstration

- Créons une fonction `void test_stack()` qui crée juste une variable de type `int`.
- Désassemblons son code grâce à la commande `disas test_stack`
- Créons une fonction `void test_heap()` qui crée juste un pointeur de type `int`.
- Désassemblons son code grâce à la commande `disas test_heap`

⚠ Le passage par un appel système a un surcoût!

- D'un point de vue de performance, la pile est toujours préférable pour les petits objets!
- Le surcoût d'allocation sera moins notable sur les gros objets
- De plus l'accès à la mémoire du tas est souvent plus lente (non contiguë dans la mémoire physique)

4. Gestion de la mémoire en C++

4.1 Programmation orientée objet

En C++, la différence principale de structure est l'approche Objet.

Comme nous l'avons vu, les éléments de la **pile** sont **automatiquement gérés par le compilateur**.

En sortant de la portée d'une méthode, sa frame est automatiquement libérée.

On peut donc considérer que les variables sont nettoyées dès qu'on sort de la portée de leur déclaration



C++ est rétrocompatible avec le C, on peut donc à la fois

- Allouer de la mémoire de tas avec `malloc` et la libérer avec `free`
- ou allouer la mémoire avec `new` et la libérer avec `delete` (préférer celle-ci).

Les deux méthodes traitent de pointeurs.

```
int* i = new int[5]();
...
delete[] i;

SomeClass * sc = new SomeClass(arg1,arg2);
...
delete sc;
```

4.2 Constructeur et Destructeur

En C++, on crée une instance de classe avec `new`, justement, et celui-ci appelle le *constructeur* de la classe!

Sémantiquement, si on doit allouer de la mémoire, c'est l'endroit rêvé! 🌈🦄



Au travail! Créez une classe Student

Créons une classe Student qui aura les propriétés suivantes:

- Un buffer de 30 notes `grades (float *)` (privé)
- Le nombre de notes du semestre `numberOfGrades` initialement à zéro (privé)

On écrira les méthodes suivantes:

- Le constructeur `Student()` qui réservera le buffer `grades` (public)
- `addGrade(float grade)` qui ajoute une note et incrémente le nombre des notes du semestre (public)
- `getAverage()` qui renvoie la moyenne des notes du semestre. (public)

Notre pointeur `grades` est privé,  il est en sécurité  et ne pourra a priori pas être libéré par une autre portion de code.

... A moins de le rendre accessible depuis l'extérieur!

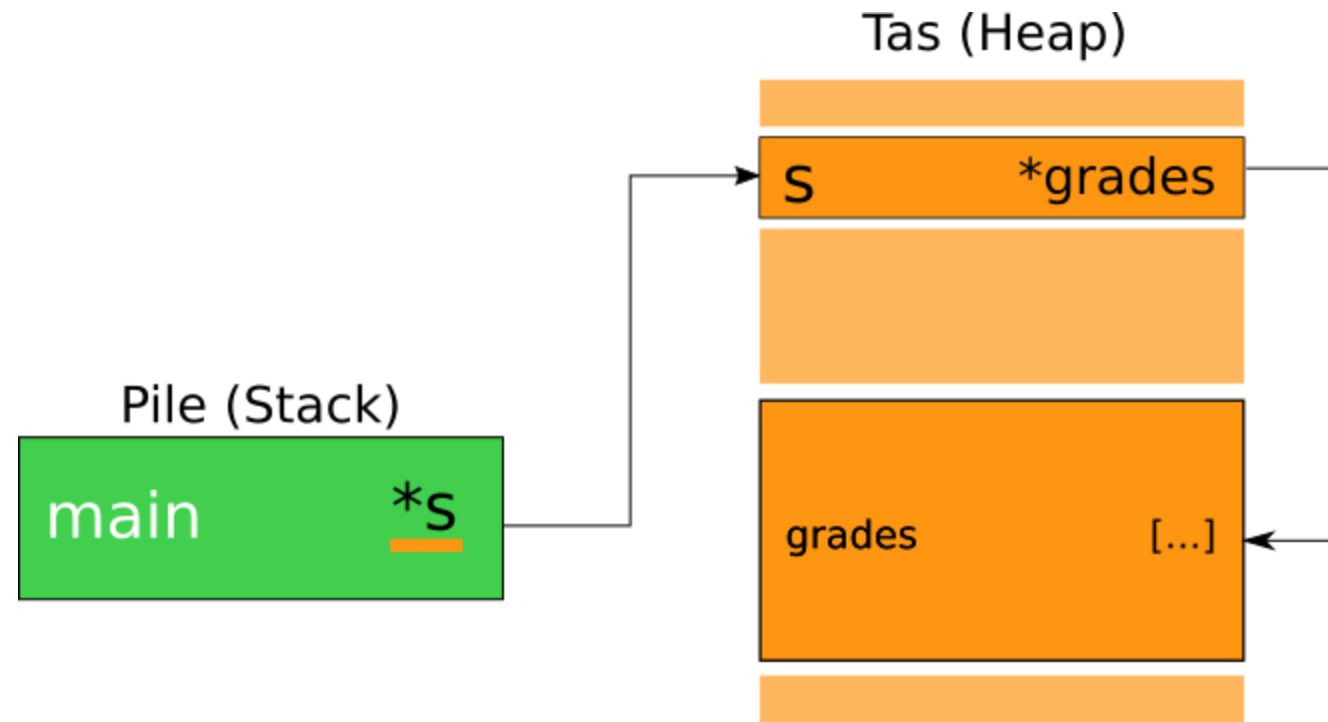
... Mais on a fait une allocation sans la libérer! 🤖



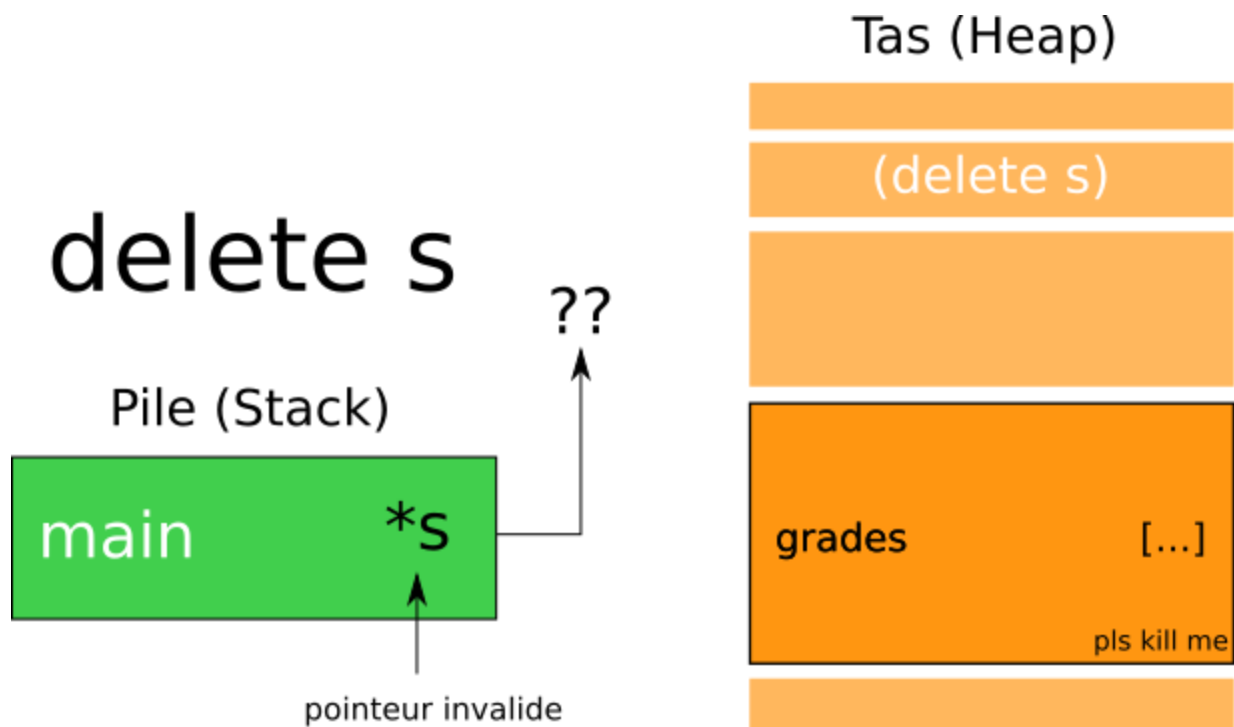
Même en faisant comme suit:

```
Student* s = new Student();  
delete s;
```

On libère un pointeur vers `Student`, mais pas son pointeur interne vers son buffer de notes, qui reste alors alloué.





delete s



Pour pouvoir s'y retrouver (éviter les double free ou les use after free, par exemple), il vaut mieux que la *logique* dicte l'endroit où on appellera `delete`.

Comme notre classe Student a initialisé la mémoire du buffer, il semble logique qu'il soit *responsable* de la libérer.

On dira qu'une instance de Student est le **propriétaire** (owner)  de cette mémoire, il sera donc responsable de la libérer.

S'il n'y a qu'un propriétaire  capable d'accéder à la mémoire et de la libérer, il devient plus facile d'éviter les use after free, les double delete, etc.

? Mais quand libérer la mémoire ?

A priori on a besoin du buffer jusqu'à la disparition de l'instance.

On utilise pour cela le **Destructeur** 🧑‍🔧 de la classe, qui s'écrit `~Student()`.

Créons le destructeur de Student


- Ecrivez la méthode `~Student()` qui libère la mémoire du buffer de notes.

4.3 Appels automatiques du destructeur

Le Destructeur 🛠️ est appelé automatiquement lorsque l'instance de classe doit quitter la mémoire:

- Si l'instance est dans la pile, au moment de la sortie de la portée contenant l'instance (possiblement avant la disparition de la frame!)
- Si l'instance est sur le tas, au moment de l'appel à `delete` de l'instance.

Testons tout ça

- Ajoutez un log sur `cout` dans le destructeur de `Student`  pour notifier sa destruction
- Testez la sortie de portée d'une instance créée sur la pile
- Testez l'appel de delete d'une instance créée sur le tas avec `new`

💡 Conclusion 💡

A priori, allouer la mémoire dans le constructeur et la désallouer dans le destructeur

- permet d'associer systématiquement une allocation à une libération (pas de fuite mémoire 💧, pas de double free 💣)
- permet d'assurer que le pointeur est valide durant toute la durée de vie de l'instance (pas de use after free 💣)

La structure mémoire de Student étant en pratique 8 ou 12 octets (un entier et un pointeur), on peut à présent l'allouer systématiquement sur la pile, sachant que ses données "lourdes" sont automatiquement gérées sur le tas!

4.4 La règle des 3

C++ crée pour nous quelques fonctions "par défaut":

- Le constructeur par copie (Copy Constructor)

```
Student(const Student& otherStudent){...}
```

- L'opérateur d'assignation (Copy Assignment operator)

```
Student& operator=(Student otherStudent)
```

💡 On verra plus tard qu'il existe en réalité plusieurs opérateurs d'assignation

Ils sont construits automatiquement et s'exécutent lorsqu'on tente ce genre d'opération:

```
Student s1; // Constructeur principal  
Student s2(s1); //Copy constructor  
Student s3 = s1; //Copy constructor  
s3 = s2 // Opérateur d'assignation
```

Essayons!

- Créez un `Student s1` par le constructeur par défaut;
- Créez-en une copie avec `Student s2(s1)` ou `Student s2 = s1`
- Testez le programme!
- Essayez en passant par un opérateur d'assignation (et non plus le constructeur de copie)
- Testez également!

Et là, c'est le drame

Des exceptions se déroulent au moment du Destructeur.

L'objet est copié, mais le pointeur reste le même sur chaque copie!

Et quand l'objet sort de la portée... le destructeur s'exécute plusieurs fois pour libérer le même pointeur... 🧑

C'est un double free! Précisément ce qu'on voulait éviter...

Enoncé de la Règle des **3**

Si vous avez besoin d'écrire explicitement l'une de ces 3 fonctions

- Destructeur
- Opérateur d'assignation (Assignment operator)
- Constructeur par copie (Copy Constructor)

⚠ Il faut les écrire explicitement toutes les 3. ⚠

💡 Depuis C++11, la règle des **3** devient même la règle des **5** 💡

Nous y reviendrons un peu plus loin avec la notion de *Move Semantics*

Démonstration

- Ecrivez le constructeur de copie pour copier le buffer sur une nouvelle zone mémoire (utilisez `std::memcpy(to_pointer, from_pointer, size);` se trouvant dans `<cstring>`)
- Ecrivez l'opérateur d'assignation de façon à également copier cette mémoire autre part sur le tas.
- Assurez vous d'avoir conservé le log dans le destructeur!
- Testez les exemples précédents!

😱 Pour l'opérateur d'assignation, on s'aperçoit que 3 Destructeurs sont appelés...

C'est parce que l'argument de l'opérateur est copié (passé par valeur, donc copié dans la pile!) avant d'être assigné.

Il est alors immédiatement détruit. ☁

Notre `Student` est copié... Mais n'avons nous pas une fonction pour ça?

Le Copy Constructor!

C'est ce qu'utilise l'opérateur d'assignation! Donc sa mémoire est *déjà* copiée!

Optimisation

Comme la copie est immédiatement supprimée, on peut simplement inverser les pointeurs avec `std::swap(pointer1, pointer2)`, qui se trouve dans `<utility>`

On évite ainsi une copie mémoire inutile!

Mettez en place le nouvel opérateur d'assignation.

5. Smart Pointers

5.1 Principe

Les *smart pointers* sont des classes C++ de la bibliothèque standard moderne.

Elles servent à initialiser un objet ou un tableau sur le tas sans appel à `new` ni `delete` : c'est cette classe qui se charge d'allouer et libérer automatiquement la mémoire.

Ils se comportent comme des pointeurs dans la syntaxe.

il existe plusieurs types de smart pointers, notamment:

- `unique_ptr`, le pointeur unique
- `shared_ptr`, le pointeur à compteur de références
- `weak_ptr`, le pointeur à validité optionnelle

5.2 Le pointeur unique `unique_ptr`

Ce pointeur alloue automatiquement un espace mémoire pour une instance de classe, et libère automatiquement la mémoire dès qu'on quitte sa portée de définition.

Il ne peut pas être copié (il n'a pas d'opérateur d'assignation ni de copy constructor!), et est seul **propriétaire** des données allouées.

Syntaxe

```
std::unique_ptr<SomeClass> v1 = std::make_unique<SomeClass>(arg1,arg2,...);  
v1->someMethod(); // S'utilise comme un pointeur  
std::unique_ptr<int[]> v30 = std::make_unique<int[]>(30);  
v30[4]; // S'utilise comme un tableau
```


Le pointeur unique dispose des méthodes suivantes:

- `release` qui renvoie l'objet initial et "libère" non pas la mémoire mais l'appartenance
- `reset` qui remplace l'objet sous-jacent par un autre du même type
- `swap` qui échange deux valeurs entre deux `unique_ptr` du même type sous-jacent



 Testons ça avec notre Student!

5.3 Le pointeur compteur de références : `shared_ptr`

Ce pointeur fonctionne de la même façon, sauf qu'il peut être copié et partagé dans le code.

- A chaque fois qu'il est copié, son compteur s'incrémente de 1
- A chaque fois qu'une copie disparaît, son compteur décrémente de 1.
- Quand le compteur atteint zéro, l'objet sous-jacent est détruit.

⚠ Il faut noter que ce mécanisme inclut un surcoût, souvent négligeable.

Syntaxe

`std::shared_ptr` fait partie de `<memory>` .

```
std::shared_ptr<SomeClass> v1 = std::make_shared<SomeClass>(arg1,arg2,...);
v1->someMethod(); // S'utilise comme un pointeur
std::shared_ptr<int[]> v30 = std::make_shared<int[]>(30);
v30[4]; // S'utilise comme un tableau
```

```
std::shared_ptr<SomeClass> v1 = std::make_shared<SomeClass>(arg1,arg2,...);
```

On peut connaître le nombre de références grâce à

- `v1.use_count()`
- `v1.unique()`

 Testons ça avec notre Student!

5.4 Le pointeur à validité optionnelle : `weak_ptr`

`weak_ptr` est un pointeur spécial fait pour fonctionner avec `shared_ptr`, qui n'incrmente ni ne décrément le compteur du `shared_ptr` de base!

On l'utilise quand on veut qu'une référence n'empêche pas la disparition de la ressource.



Créer un `weak_ptr`

```
std::weak_ptr<Student> weak_s;  
// ...  
std::shared_ptr<Student> s = std::make_shared<Student>();  
weak_s = s;
```


Utiliser un `weak_ptr`

Avant d'être utilisé, il faut le retransformer en `shared_ptr` (ce qui n'est pas forcément possible!)

```
if(std::shared_ptr<Student> s2 = weak_s.lock()){  
    s2->addGrade(16.0);  
}  
else{  
    //... Le pointeur a déjà été libéré  
}
```

 **Testons ça avec notre Student!**

6. Move Semantics

6.1 lvalues et rvalues



Vous avez peut-être déjà rencontré des erreurs de compilateur concernant les *lvalues* et les *rvalues*... Mais qu'est-ce que c'est ? ?

En réalité, C++ définit plus de types de valeurs:

- les rvalues,
- les xvalues,
- les glvalues,
- les prvalues,
- les lvalues.

💡 Restons simples: on ne parlera que des lvalues et des rvalues ici.

Définition :

- Une **lvalue** est une valeur qui a une **adresse mémoire identifiable**.
 - Une **rvalue** est une valeur qui n'est pas un **lvalue**.
-  Souvent, une *lvalue* est une valeur qui peut être utilisée à gauche d'une assignation : *lvalue*. Une *rvalue* est une valeur qui ne peut être qu'à droite.
-  Souvent une *rvalue* ne peut plus être utilisée dès la ligne suivante!

Exemple 1:

```
int i = 3;
```

- `i` est une *lvalue*. `3` est une *rvalue*.

Exemple 2 :

```
int i = 3;  
int j = i + 4;  
int k = j;
```

- `i`, `j` et `k` sont des *lvalues*
- `3`, `(i + 4)` sont des *rvalues*
- à la dernière ligne, `j` est une *lvalue* convertie en *rvalue* avant l'assignation.

⚠ une *lvalue* peut être convertie implicitement en *rvalue* mais pas le contraire! ⚠

Exemple 3 :

```
float f = cos(3.14);
```

- `f` est une *lvalue*, `cos(3.14)` est un *rvalue*
- 💡 (la valeur de retour d'une fonction est une *rvalue*).

6.2 Copie vs transfert (move)

Pour l'instant, nous avons créé un constructeur par copie, et un opérateur d'assignation par copie.

Etudions un cas concret impliquant une *rvalue* et voyons ce que ça implique, en créant une fonction qui renvoie un `Student`.

Création d'une fonction retournant un `Student` moyen.

- Ecrivez une simple fonction `createAverageStudent` retournant un `Student` possédant 3 notes, chacune valant `10.0`.
- Ajoutez un log `Constructing Student` dans le Constructeur
- Ajoutez un log `Copy Constructing Student` dans le constructeur par copie
- Vérifiez que vous avez encore le log dans le Destructeur.
- Créez un `Student` `s1` par défaut sur la pile
- Réassignez `s1` avec le résultat de `createAverageStudent`.
- Observez le résultat en console!

😬 On n'aurait pas un peu trop d'appels aux constructeurs et au destructeur? 😬

Explications

- on crée un `Student s1` par le constructeur, réservant une plage de 30 `float`
- `createAverageStudent` crée un autre `Student` par le constructeur, réservant une autre plage de 30 `float`.
- Au moment de retourner cet autre `Student`, on le copie dans la pile (avec le copy constructor!), pour le passer (par valeur!) de la frame de `getAverageStudent` à la frame de `main` avant que celle de `getAverageStudent` ne disparaisse, créant au passage une quatrième plage de 30 `float` sur le tas.

- La frame de `createAverageStudent` est supprimée, le destructeur du 2nd `Student` est donc appelé.
- L'opérateur d'assignation utilise à nouveau le Copy Constructor pour passer le 3ème `Student` à la frame de la fonction `operator=` et crée un quatrième `Student` avec donc une quatrième plage de 30 `float`, qui est sauvegardé dans `s1`.
- Une fois la frame de `operator=` supprimée, on appelle de destructeur du 4ème `Student` ;
- Le `Student` passé depuis `createAverageStudent` disparaît lui aussi, appelant son destructeur.

Au final, on a réservé 4 plages de 30 `float` et opéré deux copies profondes sur une *rvalue*, à savoir une valeur temporaire.

Comme une *rvalue* n'a pas vocation à être réutilisée, on aurait pu utiliser sa plage de 30 `float` directement et transférer sa **propriété** plutôt que de la copier.

Pour cela, on utilise

- Un constructeur de transfert (Move Constructor)
- Un opérateur d'assignation de transfert (Move assignment operator)

... Qui utilisent tous les deux une **référence de rvalue** symbolisée par `type&&`.

Move Constructor

```
Student(Student&& rStudent)
{
    this->numberOfGrades = rStudent.numberOfGrades;
    this->grades = rStudent.grades;
    rStudent.grades = nullptr; // Notez bien ceci
}
```

⚠ Le destructeur de la *rvalue* utilisée sera quand même appelé, il faut donc s'assurer qu'il ne désalouera pas le pointeur tout juste transféré à notre nouveau `Student` !

Sans la ligne

```
rStudent.grades = nullptr;
```

on a un double free...

On transfère la propriété de rStudent à `this`, avec la responsabilité de désallouer la mémoire.

Move Assignment Operator

De la même façon on écrit une nouvelle version de l'opérateur d'assignation:

```
Student& operator=(Student&& rStudent){  
    this->numberOfGrades = rStudent.numberOfGrades;  
    this->grades = rStudent.grades;  
    rStudent.grades = nullptr;  
    return *this;  
}
```

💡 Avec ces 2 définitions supplémentaires, on évite des copies coûteuses et on passe donc de la règle des **3** à la règle des **5**!

⚠️ Dépendant de la situation, on pourra ne pas avoir besoin du Copy Assignment ou du Move Assignment, l'un des deux suffisant dans la plupart des cas.



On redescend à la règle des **4**!

⚠ Dans certaines circonstances, le compilateur peut décider d'éviter la copie ou le transfert entièrement! ⚠

Avec des exemples simples, on risque de souvent tomber dans cette situation!

On désactive ce comportement en ajoutant l'option `-fno-elide-constructors` à GCC.

6.3 Développons le type `string` en partant de `char*`

- Créez une classe `string` ayant une propriété de type `char*`
 - Créez un constructeur qui prend en argument un `char*`
 - Ecrivez la méthode `string operator+(string& other)` qui concatène deux strings
 - Ecrivez les 4 fonctions nécessaires à la gestion optimale des `string` en mémoire:
 - Le destructeur, le copy Constructor, le Move Contructor, le Copy Assignment Operator
-  utilisez `std::strlen` de `<cstring>` pour obtenir la taille d'un `char*`.
-  N'oubliez pas qu'un `char*` finit par un `\0` au delà de sa taille!