



SKL

Piscine C++ - Rush 1

François Pradel francois.pradel@epitech.eu
Raphaël Londeix raphael.londeix@epitech.eu

Abstract: Sujet du rush de la première semaine de piscine C++ / Standard Kreog Library

Table des matières

I	Introduction	2
I.1	Inspiration	2
I.2	Déroulement du rush	2
I.3	Soutenance	2
I.4	Rendu	3
II	Partie 0 : Un peu de lecture	4
II.1	L'encapsulation	4
II.2	Des types et des objets	4
II.3	Implémentation en C	5
II.3.1	<i>object.h</i>	5
II.3.2	<i>raise.h</i>	5
II.3.3	<i>point.h</i>	5
II.3.4	<i>point.c</i>	5
II.3.5	<i>new.h</i>	5
III	Partie 1 : Des objets simples	6
III.1	ex_01 : Creation / destruction d'objets	6
III.2	ex_02 : Création / destruction d'objets, le retour	7
III.3	ex_03 : Additionner / soustraire des objets	8
III.4	ex_04 : Types de base	10
IV	Partie 2 : Conteneurs génériques	12
IV.1	ex_05 : class Array	15
IV.2	ex_06 : class List	16
IV.3	ex_07 : Des bonus !	17

Chapitre I

Introduction

Lisez bien plusieurs fois l'introduction en entier, toutes les consignes sont très importantes. Ce rush nécessite beaucoup de rigueur.

I.1 Inspiration

Ce rush est inspiré des travaux de Axel-Tobias Schreiner, et principalement de son livre *Objekt-orientierte Programmierung mit ANSI-C* (disponible à l'adresse suivante : <http://www.cs.rit.edu/~ats/books/ooc.pdf>). Nous vous encourageons bien évidemment à y jeter un oeil.

Dans la préface, l'auteur nous fait part de son amusement à découvrir que le C est un langage orienté objet à part entière. Cela n'a pas manqué d'attiser notre curiosité, et le résultat en est ce rush. Nous espérons bien évidemment que vous vous amuserez autant que l'auteur !

I.2 Déroulement du rush

Le rush se décompose en trois parties principales.

La partie 0 (dite de lecture) vous présentera quelques concepts qu'il sera indispensable de maîtriser avant de poursuivre le rush. Vous serez aussi évalués en soutenance sur cette partie, ne la négligez pas !

La partie 1 (dite de conception) est obligatoire, et les exercices y sont linéaires (ie. vous devez les faire dans l'ordre). Dans cette première partie nous vous présenterons une conception "objet" en C. Cette conception n'a nulle prétention à être parfaite, mais nous vous demanderons dans cette partie de la comprendre et de l'implémenter.

La partie 2 (dite d'implémentation) est l'apogée de ce rush. À l'aide de l'implémentation objet que nous vous aurons présentée dans la première partie, nous vous demanderons d'implémenter un certain nombre de types de base, de conteneurs et d'algorithmes orientés objet et génériques. Dans cette partie, tous les exercices sont indépendants.

I.3 Soutenance

Lors de la soutenance vous serez évalués sur deux points principaux :

- L'avancement de votre rush. Après avoir rempli la partie 1 vous pourrez donner libre cours à votre imagination dans la partie 2.
- La qualité de votre réflexion. Ce sera à nos yeux le plus important. Le but de ce rush est de vous amener à réfléchir sur un certain nombre de concepts nouveaux pour vous, et qui vous seront précisés durant les deux semaines à venir de piscine. Documentez-vous, posez-vous des questions, et tâchez d'y répondre!

I.4 Rendu

Nous vous demandons de respecter scrupuleusement les noms des dossiers dans lesquels vous rendrez chacun des exercices : ex_01, ex_02, ex_03, ex_04, ex_05, ex_06 et bonus. Ces dossiers doivent se trouver à la racine de votre répertoire de rendu SVN. Aucun de ces dossiers ne doit en aucun cas comporter de sous-dossiers. Si vos dossiers ne portent pas le bon nom ou sont mal placés vous ne serez pas corrigés! Aucune dérogation ne sera accordée, vous êtes prévenus.

Le rendu devra se faire sur le dépôt SVN qui vous est fourni par le Koalab. Vous recevrez les informations de connexion au dépôt de votre groupe au début du rush. Vous trouverez la documentation pour utiliser votre dépôt SVN fourni par le Koalab au même endroit que ce sujet. Nous vous conseillons de la lire scrupuleusement.

Votre dépôt sera fermé en écriture dimanche matin à 10h00. Seuls les fichiers présents sur votre dépôt lors de la soutenance seront corrigés. Aucune dérogation ne sera accordée.

Pour toute question, nous vous invitons à envoyer un message par mail aux auteurs du sujet.

Je vous conseille de relire cette partie à nouveau.



Vos fichiers seront compilés avec les flags `-std=gnu99 -Wall -Wextra -Werror`

Chapitre II

Partie 0 : Un peu de lecture

Dans cette partie nous allons vous présenter quelques concepts de la programmation orientée objet.



Google et Wikipédia sont vos amis !

II.1 L'encapsulation

Le problème dans la création d'un type en C, est que l'utilisateur est totalement libre dans sa manière de l'utiliser. Votre seul moyen de protéger l'utilisateur est de lui fournir un ensemble de fonction pour utiliser ce type. Par exemple, si vous définissez une structure `player_t`, il faudra penser à fournir les fonctions `player_init(player_t*)`, `player_destroy(player_t*)` et ainsi de suite.

Ainsi, pour chaque type différent, vous devez systématiquement coder toutes les fonctions qui gère les *objets* de votre programme. Par exemple, pour le type `player_t`, vous préfixerez toutes les fonctions par `player_`. Le fait de masquer à l'utilisateur la représentation en mémoire d'un type, de ranger toutes fonctions qui s'y applique ensemble s'appelle l'*encapsulation*.

Cette pratique garanti à l'utilisateur une sécurité totale tant qu'il se cantonne à manipuler ces objets avec les fonctions fournies.

II.2 Des types et des objets

Lorsque vous déclarez un type en C, vous devez, pour l'utiliser, faire deux choses de manière systématique. Tout d'abord allouer une zone mémoire pour votre instance, et ensuite l'initialiser. Nous appellerons ce processus la *construction* d'un *objet* ou *instance*. La première étape se fait exactement de la même façon pour tout les types, pour peut que l'on connaisse la taille de ce type. Seule la deuxième étape, dite d'initialisation, est typique de chaque type. Nous appelons la fonction qui initialise un objet un *constructeur*, et celle qui le détruit un *destructeur*.

L'ensemble d'un type et de toutes les fonctions membres (qui agissent sur ce type) est appelé une *classe*. Nous allons décrire au travers de ce rush un des moyens possible de faire de la *programmation orienté objet* en C.

II.3 Implémentation en C

Quoi que l'auteur mentionné ci-dessus puisse en penser, force est de reconnaître que le C n'est pas un langage foncièrement orienté objet. Comment allons-nous donc nous en sortir ? Nous présenterons dans cette section les principales idées sous-jacentes à la conception que nous vous proposons. Charge à vous de concevoir les bouts manquants dans la Partie 1 !

II.3.1 *object.h*

object.h contient le type `Class`. C'est donc le type d'un type (inception), il contient le nom de la classe, sa taille (pour le `malloc`), et des pointeurs sur fonctions, comme le constructeur et le destructeur.

II.3.2 *raise.h*

raise.h contient la macro `raise()` que vous devez utiliser pour gérer tout les cas d'erreur (par exemple, si `malloc()` renvoie `NULL`). Elle prend en paramètre une chaîne de caractères qui indique le type d'erreur. Par exemple :

```
1 if ((ptr = malloc(sizeof(*ptr))) == NULL)
2     raise("Out of memory");
```

Et a pour effet d'afficher ce message sur la sortie d'erreur et de quitter le programme.

II.3.3 *point.h*

point.h présente la *classe* `Point`. Cette description est intrigante, mais *point.c* devrait vous éclairer. La *classe* `Point` est en fait la déclaration externe d'une variable statique qui décrit la *classe*. Vous pouvez maintenant vous arrêter de lire quelques minutes, et méditer cette phrase à l'aide du code fourni !

En fait, puisque `Point` est une variable de type `Class`, elle décrit un type. Nous utiliserons ce type pour pouvoir construire et détruire des *instances* de points.

II.3.4 *point.c*

Ce fichier présente l'*implémentation* de la classe `Point`. En définitive, c'est ici que le constructeur et le destructeur seront codés. Vous trouverez aussi la fameuse variable `Point`, qui contient comme prévu les pointeurs sur fonctions des constructeurs et destructeurs, qui seront utilisés de manière transparente par `new` et `delete`.

II.3.5 *new.h*

Enfin, *new.h* contient les prototypes des fonctions `new()` et `delete()`, qui vous permettront de construire et détruire des objets. À vous de jouer maintenant !

Chapitre III

Partie 1 : Des objets simples

III.1 ex_01 : Creation / destruction d'objets

Fichiers fournis :

Nom	Description
raise.h	Contient la macro <code>raise</code>
new.h	Prototypes pour <code>new</code> et <code>delete</code>
object.h	Définition de la structure <code>Class</code>
point.h	Déclaration de la variable <code>Point</code>
point.c	Implémentation de la classe <code>Point</code>
ex_01.c	main d'exemple

Fichier à rendre :

Nom	Description
new.c	Implémentation des fonctions <code>new</code> et <code>delete</code>

Le but de cet exercice est l'implémentation des fonctions `new()` et `delete()`. Celles ci auront pour effet de construire et de détruire des *objets* de type `Point`. On veut pouvoir écrire un code comme celui-ci :

```
1 #include "new.h"
2 #include "point.h"
3
4 int main()
5 {
6     Object * point = new(Point);
7     delete(point);
8
9     return 0;
10 }
```

La tâche de cette première version de `new()` est d'allouer de la mémoire en fonction de la classe passée en argument, puis d'appeler le *constructeur* de la *classe*, s'il est disponible. De la même manière, la fonction `delete()` devra appeler le destructeur s'il est présent, puis libérer la mémoire.



`malloc(3) memcpy(3)`

III.2 ex_02 : Création / destruction d'objets, le retour

Fichiers fournis :

Nom	Description
raise.h	Contient la macro <code>raise</code>
new.h	Prototypes pour <code>new</code> et <code>delete</code>
object.h	Définition de la structure <code>Class</code>
point.h	Déclaration de la variable <code>Point</code>
vertex.h	Déclaration de la variable <code>Vertex</code>
ex_02.c	main d'exemple

Fichiers à rendre :

Nom	Description
new.c	Implémentation des fonctions <code>new</code> et <code>delete</code>
point.c	Implémentation de la classe <code>Point</code>
vertex.c	Implémentation de la classe <code>Vertex</code>

La précédente version de la fonction `new()` ne permettait pas de passer des paramètres au constructeur. Vous devrez donc fournir une nouvelle version de celle-ci en utilisant les `va_list`. En outre, pour pouvoir répondre à d'autres besoins, une version de `new()` nommée `va_new()` est demandée. Les prototypes des fonctions demandées sont donc :

```
1 Object* new(Class* class, ...);
2 Object* va_new(Class* class, va_list* ap);
3 void delete(Object* ptr);
```



Les constructeurs et destructeurs n'afficheront plus de message, pour cet exercice et les suivants.

Ainsi, il sera possible d'utiliser les fonctions de cette manière :

```
1 #include "new.h"
2 #include "point.h"
3 #include "vertex.h"
4
5 int main()
6 {
7     Object* point = new(Point, 42, -42);
8     Object* vertex = new(Vertex, 0, 1, 2);
9
10    delete(point);
11    delete(vertex);
12    return 0;
13 }
```


Vous devrez créer une nouvelle classe `Vertex` en vous inspirant de la classe `Point`. La structure `Class` contient maintenant une nouvelle *fonction membre* `__str__`, qui renvoie une chaîne de caractère. La macro `str` présente dans `object.h` s'occupe d'appeler cette fonction membre. Vous devrez faire en sorte que le code source suivant :

```
1 printf("point = %s\n", str(point));
2 printf("vertex = %s\n", str(vertex));
```

produise cet affichage :

```
point = <Point (42, -42)>
vertex = <Vertex (0, 1, 2)>
```



`stdarg(3)` `snprintf(3)`

III.3 ex_03 : Additionner / soustraire des objets

Fichiers fournis :

Nom	Description
<code>raise.h</code>	Contient la macro <code>raise</code>
<code>new.h</code>	Prototypes pour <code>new</code> et <code>delete</code>
<code>object.h</code>	Définition de la structure <code>Class</code>
<code>point.h</code>	Déclaration de la variable <code>Point</code>
<code>vertex.h</code>	Déclaration de la variable <code>Vertex</code>
<code>ex_03.c</code>	main d'exemple

Fichiers à rendre :

Nom	Description
<code>new.c</code>	Implémentation des fonctions <code>new</code> et <code>delete</code>
<code>point.c</code>	Implémentation de la classe <code>Point</code>
<code>vertex.c</code>	Implémentation de la classe <code>Vertex</code>

Dans cet exercice, nous allons simplement ajouter deux fonctions membres dans la structure `Class` pour pouvoir additionner ou soustraire des objets. Vous devrez donc adapter vos précédents fichiers `point.c` et `vertex.c` pour qu'ils implémentent les fonctions membres `__add__` et `__sub__`. Nous pourrions par exemple avoir un code comme celui-ci :

```
1 #include "new.h"
2 #include "point.h"
3 #include "vertex.h"
4
5 int main()
6 {
7     Object* p1 = new(Point, 12, 13),
8         * p2 = new(Point, 2, 2),
9         * v1 = new(Vertex, 1, 2, 3),
10        * v2 = new(Vertex, 4, 5, 6);
11
12    printf("%s + %s = %s\n", str(p1), str(p2), str(add(p1, p2)));
13    printf("%s - %s = %s\n", str(p1), str(p2), str(sub(p1, p2)));
14    printf("%s + %s = %s\n", str(v1), str(v2), str(add(v1, v2)));
15    printf("%s - %s = %s\n", str(v1), str(v2), str(sub(v1, v2)));
16
17    return 0;
18 }
```

Et nous aurons donc l'affichage suivant :

```
<Point (12, 13)> + <Point (2, 2)> = <Point (14, 15)>
<Point (12, 13)> - <Point (2, 2)> = <Point (10, 11)>
<Vertex (1, 2, 3)> + <Vertex (4, 5, 6)> = <Vertex (5, 7, 9)>
<Vertex (1, 2, 3)> - <Vertex (4, 5, 6)> = <Vertex (-3, -3, -3)>
```

III.4 ex_04 : Types de base

Fichiers fournis :

Nom	Description
<code>raise.h</code>	Contient la macro <code>raise</code>
<code>bool.h</code>	Définition du type <code>bool</code> et des macros <code>true</code> et <code>false</code>
<code>new.h</code>	Prototypes pour <code>new</code> et <code>delete</code>
<code>object.h</code>	Définition de la structure <code>Class</code>
<code>float.h</code>	Déclaration de la variable <code>Float</code>
<code>int.h</code>	Déclaration de la variable <code>Int</code>
<code>char.h</code>	Déclaration de la variable <code>Char</code>
<code>ex_04.c</code>	main d'exemple

Fichiers à rendre :

Nom	Description
<code>new.c</code>	Implémentation des fonctions <code>new</code> et <code>delete</code>
<code>float.c</code>	Implémentation de la classe <code>Float</code>
<code>int.c</code>	Implémentation de la classe <code>Int</code>
<code>char.c</code>	Implémentation de la classe <code>Char</code>

Nous allons étendre une fois de plus la classe de base et réécrire quelques types natifs du C. Les nouveaux types que vous devez implémenter sont les types `Int`, `Float` et `Char`. Il s'agira comme précédemment de pouvoir additionner et soustraire des objets du même type, mais aussi de pouvoir les comparer. Les opérateurs de comparaison `==` (`__eq__`), `<` (`__lt__`) et `>` (`__gt__`) font donc leur apparition dans la classe de base. Ces trois classes seront utilisées dans la section suivante de façon intensive. Les opérations et comparaisons entre des objets de différents types n'est pas demandé mais pourra être l'objet d'un bonus :).

Par exemple, un objet de type `Int`, `Float` ou `Char` pourra être manipulé comme ceci :

```
1 void compareAndDivide(Object* a, Object* b)
2 {
3     if (gt(a, b))
4         printf("a > b\n");
5     else if (lt(a, b))
6         printf("a < b\n");
7     else
8         printf("a == b\n");
9
10    printf("b / a = %s\n", str(div(b, a)));
11 }
```

Comme d'habitude, des macros sont définies pour faciliter l'appel des fonctions membres.

Chapitre IV

Partie 2 : Conteneurs génériques

Nous avons créé dans la partie précédente des types simples. Cette section va se concentrer sur l'écriture de conteneurs. Un conteneur est un objet capable de contenir tout type d'objet dérivant de la classe de base. Nous allons introduire ici une *classe intermédiaire*, afin de n'ajouter de fonctions membres qu'aux conteneurs. En effet il serait trop lourd d'ajouter toutes les fonctions membres pour toutes les classes, ce que nous avons fait jusqu'à présent. Une classe intermédiaire est simplement une structure contenant une variable de type `Class` et est définie ainsi :

```
1 #include "object.h"
2
3 typedef struct Container_s Container;
4
5 struct Container_s
6 {
7     Class base;
8     void (*newMethod)(Container* self);
9 };
```

Nous avons simplement ajouté un pointeur sur fonction nommé `newMethod` dans la classe `Container`. Vous noterez que cette classe *contient* la classe de base, elle doit donc implémenter toute ses fonctions membres.

Pour définir un conteneur, vous pouvez procéder de cette manière dans le fichier `.c` :

```

1
2 #include "container.h"
3
4 typedef struct MyContainerClass
5 {
6     Container base;
7     int _val;
8 };
9
10 static MyContainerClass _descr = {
11     { /* Container struct */
12         { /* Class struct */
13             sizeof(MyContainerClass),
14             "MyContainer",
15             /* All Class functions here */
16         },
17         &MyContainer_newMethod, /* the new method from class Container */
18     },
19     0, /* members of MyContainer */
20 };
21
22 Class* MyContainer = (Class*) &_descr; /* as usual */

```

Nous avons ici défini une classe `MyContainer`, *dérivant* de `Container`, elle-même dérivant de `Class`. C'est pour cela que la variable `_descr` comprend la déclaration de trois structures imbriquées.

Le dernier concept introduit dans cette section est le concept d'*itérateurs*. Un itérateur permet de parcourir un conteneur, un peu comme un index pour un tableau. Avant, lorsque vous parcouriez un tableau C, vous faisiez quelques chose comme ça :

```

1 int tab[10];
2 unsigned int i;
3
4 i = 0;
5 while (i < 10)
6 {
7     /* do stuff */
8     i = i + 1;
9 }

```

Le problème, est que lorsque vous décidiez de changer de conteneur, par exemple de passer par des listes, il fallait changer tout le code, car la manière de parcourir une liste est totalement différente. Une manière uniformisée de parcourir un conteneur est d'utiliser des itérateurs. En admettant la présence d'une classe `MyContainer`, nous aurions :

```

1 Container* tab = new(MyContainer);
2 Iterator* it;
3 Iterator* tab_end;
4
5 it = begin(tab);
6 tab_end = end(tab);
7 while (lt(it, tab_end)) /* it < tab_end */

```

```
8 {  
9     /* do stuff */  
10    incr(it);  
11 }
```

Vous remarquerez donc que la philosophie est la même, mais l'implémentation du conteneur est complètement masquée. La fonction membre `incr` est exactement identique à l'incrément de la variable `i` dans l'exemple précédant : elle nous permet d'*avancer* à l'élément suivant du tableau. La définition de la classe `Iterator` suit le même raisonnement que pour `Container`. Nous la définissons comme une classe intermédiaire, servant à définir les autres itérateurs, spécifiques à chaque conteneurs.

IV.1 ex_05 : class Array

Fichiers fournis :

Nom	Description
<code>raise.h</code>	Contient la macro <code>raise</code>
<code>bool.h</code>	Définition du type <code>bool</code> et des macros <code>true</code> et <code>false</code>
<code>new.h</code>	Prototypes pour <code>new</code> et <code>delete</code>
<code>object.h</code>	Définition de la structure <code>Class</code>
<code>container.h</code>	Définition de la structure <code>Container</code>
<code>iterator.h</code>	Définition de la structure <code>Iterator</code>
<code>float.h</code>	Déclaration de la variable <code>Float</code>
<code>int.h</code>	Déclaration de la variable <code>Int</code>
<code>char.h</code>	Déclaration de la variable <code>Char</code>
<code>array.h</code>	Déclaration de la variable <code>Array</code>
<code>array.c</code>	Implémentation partielle de la classe <code>Array</code>
<code>ex_05.c</code>	main d'exemple

Fichiers à rendre :

Nom	Description
<code>new.c</code>	Implémentation des fonctions <code>new</code> et <code>delete</code>
<code>float.c</code>	Implémentation de la classe <code>Float</code>
<code>int.c</code>	Implémentation de la classe <code>Int</code>
<code>char.c</code>	Implémentation de la classe <code>Char</code>
<code>array.c</code>	Implémentation de la classe <code>Array</code>

Dans cet exercice, nous vous donnons l'implémentation partielle de la classe `Array`, simulant le fonctionnement d'un tableau standard. Vous devez donc remplir les fonctions contenues dans le fichier `array.c` afin d'obtenir le comportement suivant :

Le constructeur d'un `Array` prend respectivement en argument la taille (`size_t`), le type contenu dans le tableau (`Class*`) et les arguments pour le constructeur de ce type. Nous aurons par exemple :

```
1 Object* tab = new(Array, 10, Float, 0.0f);
```

Ici, `tab` est un tableau de 10 `Float` initialisés à la valeur 0.0f.

La fonction membre `getitem` prend en argument un index (`size_t`) et retourne un objet (`Object*`).

La fonction membre `setitem` prend en argument un index (`size_t`), et les arguments pour construire une instance du type contenu. Par exemple :

```
1 setitem(tab, 3, 42.042f);
```

Nous plaçons dans `tab` un `Float` dont la valeur est 42.042f à l'index 3.

La fonction membre `setval` de l'itérateur de tableau fonctionne de la même manière :

```
1 Object *start = begin(tab);
2 setval(start, 3.14159265f);
```

Nous passons la première case du tableau à la valeur de 3.14159265f



utilisez `va_new()`

IV.2 ex_06 : class List

Fichiers fournis :

Nom	Description
<code>raise.h</code>	Contient la macro <code>raise</code>
<code>bool.h</code>	Définition du type <code>bool</code> et des macros <code>true</code> et <code>false</code>
<code>new.h</code>	Prototypes pour <code>new</code> et <code>delete</code>
<code>object.h</code>	Définition de la structure <code>Class</code>
<code>container.h</code>	Définition de la structure <code>Container</code>
<code>iterator.h</code>	Définition de la structure <code>Iterator</code>
<code>float.h</code>	Déclaration de la variable <code>Float</code>
<code>int.h</code>	Déclaration de la variable <code>Int</code>
<code>char.h</code>	Déclaration de la variable <code>Char</code>

Fichiers à rendre :

Nom	Description
<code>new.c</code>	Implémentation des fonctions <code>new</code> et <code>delete</code>
<code>float.c</code>	Implémentation de la classe <code>Float</code>
<code>int.c</code>	Implémentation de la classe <code>Int</code>
<code>char.c</code>	Implémentation de la classe <code>Char</code>
<code>list.h</code>	Déclaration de la variable <code>List</code>
<code>list.c</code>	Implémentation de la classe <code>List</code>
<code>ex_06.c</code>	main d'exemple

En prenant exemple sur la classe `Array`, vous créerez une classe `List`, permettant de manipuler facilement des listes. Vous pouvez choisir les comportements que vous voulez pour les opérateurs (par exemple l'addition), mais vous devrez justifier vos choix lors de la soutenance (est-il pertinent de pouvoir multiplier deux listes?). Vous êtes libres d'ajouter des méthodes si vous utilisez une classe intermédiaire spécifique aux listes. Nous devons pouvoir utiliser vos listes et la classe `Array` ensemble.

Vous nous rendrez dans votre main l'ensemble des tests nécessaires à montrer le bon fonctionnement de votre classe. Ces tests devront être nombreux et de qualités, et seront évalués en soutenance.

Vous avez dans cette partie le droit de modifier `container.h`.

IV.3 ex__07 : Des bonus !

Sentez vous libre de nous impressionner en :

- Codant des conteneurs supplémentaires (encore des **String**?)
- Ajoutant des classes intermédiaire pour **Array** et pour **List** et qui définissent des fonctions membres spécifique à l'une et l'autre de ces classes :
 - `Array.resize`
 - `Array.push_back`
 - `List.push_front` et `List.push_back`
 - `List.pop_front` et `List.pop_back`
 - `List.front` et `List.back`
- Rendant les précédents conteneurs compatibles avec les types natifs du C (`int`, `float`, `char`)
- Rendant l'utilisation plus sûre (plus de macros? nombre magique au début d'une classe? vérifier les types?)
- Modifiant la conception pour supporter la définition de fonctions membres dans les classes intermédiaires
- Rendant possible de faire des opérations entre des types différents :
 - `3.0f + 2 = 5.0f`
 - `3 * ['a'] = ['a', 'a', 'a']`
 - `2 * "salut" = "salutsalut"`
 - etc.
- Ayant l'air totalement frais après ce rush ;)