

## TP 1

### 1 Passer de C++ à Java ou vice versa !

Il est parfois difficile pour les habitués du **JAVA** de se mettre au **C++**. Probablement parce que les syntaxes se ressemblent alors que les comportements sont très différents. Passer de **C++** à **JAVA** est généralement moins douloureux, une fois que l'on a admis que l'on se place dans une optique résolument plus orientée objet où tous les objets dérivent implicitement d'une super classe de base **Objet**, qu'il n'y a plus de pointeurs et que tout est référence (sauf les variables correspondant aux types primitifs), et que la fonction `main()` doit être une méthode d'une classe de lancement du programme. En **C++**, la compilation se fait à travers une succession d'étapes (précompilation, création des `.o`, édition de lien) qu'il est nécessaire de refaire pour adapter l'exécutable à une nouvelle architecture. En **Java**, la compilation produit un code interprétable par une **JVM** (Java Virtual Machine) qui peut donc être exécuté sur toute machine disposant d'une **JVM**.

Définition en **JAVA** d'une nouvelle classe `LaJavaClasse` dans un fichier `LaJavaClasse.java` (un fichier `.java` par classe, préfixé par le nom de la classe).

```
public class LaJavaClasse
{
    public :
        // Methodes publiques
        // Constructeurs
        LaJavaClasse()
        {
            i=1;
            System.out.println("LaJavaClasse() "+ i + " (constructeur par défaut) ");
        }
        // Methode finalize appelee par le ramasse-miette quand une instance n'est plus utilisee
        finalize() {}// facultatif
    private :
        // Attributs
        int i; // pas d'entiers non signes en Java
}

```

Une instance de `LaJavaClasse` ne peut être créée que dans la zone de la mémoire appelée le tas, puis elle est manipulées par le biais d'une référence, laquelle peut très bien référer sur une autre instance au cours du programme.

```
public class Test
{
    public static void main(String[] args)
    {
        LaJavaClasse jaja;
        jaja = new LaJavaClasse(); //Construction par défaut
    }
}

```

La compilation et l'exécution dans un terminal se font avec les commandes suivantes :  
`javac LaJavaClasse.java Test.java` (création des fichiers `.class`) puis `java Test`.

En C++, une instance d'une classe est généralement créée dans la pile. On peut aussi opter pour une demande d'allocation dynamique dans le tas, sachant qu'il faudra dans ce cas penser à la restitution de l'espace mémoire en fin d'utilisation. L'instance allouée dynamiquement se manipule alors à travers une indirection de type pointeur (la syntaxe de l'indirection est donc plus explicite que dans le cas d'une référence). Dans les deux cas, l'instance est initialisée par appel automatique du constructeur adéquat.

```
#include <iostream>
class LaClasse
{
public :
    // fonctions membres publiques
    // Constructeurs
    LaClasse()
    {
        i=1;
        std::cout << "LaClasse::LaClasse() " << i << " (constructeur par défaut) "
                    << std::endl;
    }
    // Destructeur
    ~LaClasse() {}
private :
    // attributs
    unsigned int i;
};

int main(int argc, const char * argv[])
{
    LaClasse * lolo; // Variable de type pointeur cree sur la pile
                    // (32 ou 64 bits suivant l'architecture)
    lolo=new LaClasse(); //Allocation d'un LaClasse dans le tas,
                    //initialise par default,
                    // et dont l'adresse est retournée dans lolo
    {
        LaClasse lili; //Instance creee sur la pile
                    //et initialisee par appel automatique au constructeur par default
        ....

    } //Appel automatique au destructeur de lili
    ....
    delete lolo; // Appel au destructeur de l'instance pointee par lolo
                // et restitution de l'espace alloue
    return 0;
}
```

La compilation et l'exécution dans un terminal se font avec les commandes suivantes :

```
g++ -Wall -c monprogramme.cpp (compilation)
g++ monprogramme.o (édition de lien)
puis ./a.out.
```

Création de tableaux en Java :

```
int tab[] = new int[5]; // Allocation dynamique dans le tas d'un tableau de 10 int
LaJavaClasse tjav [] = new LaJavaClasse[5];
```

```
for (int i=0;i<5;i++)
    tjav[i]=new LaJavaClasse();
```

Création de tableaux en C++ :

```
LaClasse tab[5]; // Tableau statique de 5 LaClasse contigus
                // crees dans la pile et initialises par default
LaClasse *tab2 = new LaClasse[5]; // Tableau alloue dynamiquement
                // de 5 LaClasse contigus crees dans le tas
                // et initialises par default
...
delete tab2 []; // restitution de l'espace memoire apres appel
                // au destructeur des 5 LaClasse du tableau pointe par tab2
```

## 1.1 A vous de jouer ! En C++ puis en Java (ou vice versa)

Ecrivez une classe `Pixel` contenant 3 entiers. Pour l'instant vous vous contenterez de munir votre classe d'un constructeur par défaut qui met les 3 valeurs à 0, ainsi que d'accesseurs et de modificateurs de chacune des 3 valeurs. Ecrivez ensuite une classe `Image` munie d'une largeur, d'une hauteur et d'un tableau de `Pixels`. Munissez votre classe d'un constructeur qui crée le tableau de `Pixels` en fonction de la largeur et de la hauteur passées en paramètre, ainsi que d'accesseurs et de modificateurs permettant d'accéder et de modifier la valeur d'un `Pixel` dont on donne les coordonnées. Votre classe `Image` sera naturellement dotée d'un destructeur dans le cas de l'implémentation C++.

Dans un programme principal, créez une `Image` (dans la pile pour ce qui concerne l'implémentation C++) et faites une boucle qui modifie les valeurs de tous les `Pixels` de l'`Image` autant de fois que voulu par l'utilisateur. Que constatez-vous en terme de rapidité du `Java` et du `C++` quand le nombre de passage dans la boucle devient important ?

## 2 Foire aux constructeurs et autres babioles

Cet exercice est essentiel et à faire avec soin, pour vous assurer que vous comprenez bien le mécanisme des constructeurs et des destructeurs en C++. Prenez une feuille pour faire la trace du programme, en expliquant le contexte de chaque appel de fonction. Vous remettrez cette feuille à votre chargé de TP afin que nous puissions identifier qui a besoin d'informations supplémentaires sur les notions abordées.

Commencez tout d'abord à effectuer la compilation avec l'option `-fno-elide-constructors` pour être sûr qu'aucune optimisation ne soit effectuée par le compilateur. Supprimez ensuite cette option. Quelle est la nature des optimisations effectuées par le compilateur ? Habituez vous également à ajouter l'option de compilation `-std=c++03` ou `-std=c++11` ou `-std=c++1y` pour vous conformer à la norme 2003, 2011 ou 2014 du C++.

```
#include <iostream>

class LaClasse
{
public :
    //Construction, conversion, affectation et destruction
    LaClasse() : l(0)
    {std::cout << "LaClasse::LaClasse()\n";}
    LaClasse(const LaClasse & lc) : l(lc.l)
    {std::cout << "LaClasse::LaClasse(const LaClasse & )\n";}
    LaClasse(int i) : l(i)
```

```

{std::cout << "LaClasse::LaClasse(int)\n";}
operator bool() const
{std::cout << "LaClasse::operator bool() const\n"; return true;}
~LaClasse()
{std::cout << "~LaClasse::LaClasse()\n";}
const LaClasse & operator=(const LaClasse & c)
{std::cout << "LaClasse::operator=(const LaClasse &)\n"; return *this;}
//Autre fonction membre
LaClasse F(LaClasse);
// Déclaration fonction extérieure amie
friend std::ostream & operator << (std::ostream & os, const LaClasse & lc);
private :
    int l;
};

LaClasse F(LaClasse vv)
{
    std::cout << " in F \n";
    return 8;
}

LaClasse LaClasse::F(LaClasse v)
{
    std::cout << " in LaClasse::F \n";
    return :F(v);
}

std::ostream & operator << (std::ostream & os, const LaClasse & lc)
{
    os << " in ostream << LaClasse "<< lc.l << std::endl;
    return os;
}

// Testez et analysez la séquence d'appels aux fonctions membres
// de LaClasse dans le programme suivant :

int main()
{
    LaClasse c1;
    LaClasse c2=LaClasse();
    LaClasse cc1(c1);
    LaClasse cc2=c1;
    LaClasse cc3=LaClasse(c1);
    LaClasse cv1(5);
    LaClasse cv2=5;
    LaClasse cv3=LaClasse(5);
    LaClasse cv4=(LaClasse)5;
    std::cout << std::endl;
    c1=cc1;
    std::cout << std::endl;
    c2=8;
    std::cout << std::endl;
    if(cv1)

```

```

{
    cv1=F(10);
    cv1=F(c1);
    cv1=c1.F(c2);
}

std::cout << "Tableaux \n";
LaClasse tab[3];
LaClasse *pc=new LaClasse(tab[0]);
delete pc;
std::cout << "Avant de sortir ... \n";
return 0;
}

```

### 3 Un module *String* fait main

Ecrivez une classe *String* permettant la manipulation de chaînes de caractères.

Comme toujours, vous enchaînez des cycles de :

1. développement d'une nouvelle fonctionnalité;
2. compilation;
3. exécution et vérification de son bon fonctionnement.

Vous ne passerez à une nouvelle opération que lorsque la précédente sera validée sur au moins un cas.

Votre classe *String* devra satisfaire (notamment) aux contraintes suivantes :

- Possibilité de construction à partir d'une chaîne de caractères ordinaire (ou chaîne vide par défaut);
- Possibilité de considérer un caractère comme une chaîne d'un seul caractère (constructeur à partir d'un *char*);
- Possibilité de conversion en une chaîne de caractères ordinaire, pour pouvoir utiliser certaines des fonctionnalités de la bibliothèque standard C-ANSI (Attention à bien protéger votre classe *String*!)
- Opérations de concaténation (+) et de comparaison (==, !=, <, >, <=, >=) entre chaînes;
- Accès en lecture (et en écriture si c'est possible) à la *i*ème lettre d'une chaîne (opérateurs []);
- Accès à la longueur d'une chaîne;
- Test d'une chaîne vide;
- Fabrication d'une sous-chaîne (à partir de 2 indices début et fin);
- Construire une chaîne à partir d'une chaîne dans laquelle une sous-chaîne donnée aura été enlevée.

On munira également la classe d'opérations d'entrée/sortie (utilisation des classes *ostream* et *istream* avec surcharges des opérateurs « et » ).