TP 1

1 Anatomie d'une classe

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 pourrez remettre cette feuille à votre chargé de TP afin qu'il puisse identifier les informations dont vous avez besoin.

Commencez tout d'abord par 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`, `-std=c++14`, `-std=c++17` pour vous conformer à la norme 2003, 2011, 2014 ou 2017 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)
  {l=c.l; 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;
}
```

2 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, que tout est référence (sauf les variables correspondant aux

types primitifs), et que la fonction `main()` doit être une méthode statique 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). Si vous voulez placer vos classes dans un package, vous placerez vos fichiers `.java` dans un répertoire portant le nom du package.

```
//package LeJavaPackage; // Pour plonger la classe dans un package
public class LaJavaClasse
{
     // Methodes
     // Constructeurs :
     public LaJavaClasse() // Constructeur par defaut
      {
        i=1;
        System.out.println("LaJavaClasse() "+ i + " (constructeur par defaut) ");
       }
    // Methode finalize appelee par le ramasse-miette quand une instance n'est plus utilisee
    public finalize() {}// facultatif
    // Attributs
    private 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 (par appel à `new`), puis elle est manipulée 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; //Creation d'une reference non liee a un objet
      jaja = new LaJavaClasse(); //Construction et initialisation par defaut
                                 // d'une instance de LaJavaClasse dans le tas
   }
}
```

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` (ou bien `java LeJavaPackage.Test` si les classes sont plongées dans un package).

En `C++`, une variable instance d'une classe est 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 par `new` se manipule alors à travers une indirection de type pointeur (la syntaxe de l'indirection est alors explicite alors qu'elle ne l'est pas 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() // Constructeur par defaut
     {
       i=1;
       std::cout << "LaClasse::LaClasse() " <<  i <<  " (constructeur par défaut) "
                     << std::endl);
     }
   // Destructeur
    ~LaClasse() {}
  private :
    // donnees membres
   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 defaut,
                // et dont l'adresse est retournee dans lolo
  {
     LaClasse lili; //Instance creee sur la pile
                //et initialisee par appel automatique au constructeur par defaut
     ....

  } //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`. Si jamais vous voulez également regarder le résultat de la précompilation vous pouvez utiliser : `g++ -E monprogramme.cpp`

Création de tableaux en `Java` :

```
int tab[] = new int[5]; // Allocation dynamique dans le tas d'un tableau de 5 int
LaJavaClasse tjav [] = new LaJavaClasse[5]; // Allocation dans le tas de 5
                                    // references non liees a des objets
for (int i=0;i<5;i++) //Allocation dynamique successive de 5 LaJavaClasse
   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 defaut
LaClasse *tab2 = new LaClasse[5]; // Tableau alloue dynamiquement
                  // de 5 LaClasse contigus crees dans le tas
```

```
                  // et initialises par defaut
...
delete [] tab2; // restitution de l'espace memoire apres appel
                // au destructeur des 5 LaClasse du tableau pointe par tab2
```

2.1 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 crèe le tableau de `Pixel`s 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'implementation `C++`.

Dans un programme principal, créez une `Image` (dans la pile pour ce qui concerne l'implementation `C++`) et faites une boucle qui modifie les valeurs de tous les `Pixel`s 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 ?