TD 3 - C++ - Polymorphisme

Hugo Bolloré[1]

[1]hugo.bollore@uvsq.fr

Dans ce TD nous utiliserons le langage C++, vous aurez besoin du compilateur *g++* pour compiler les codes donnés par la suite.

Exercice 3.1 Polymorphisme ad hoc

Question 3.1.1 Surcharge d'opérateur

En vous inspirant de l'exemple donné en cours, ajoutez les surcharges d'opérateurs manquantes dans le code suivant :

- L'opérateur * appliqué au type duck doit multiplier la valeur de *first* par la valeur de l'opérande entier,
- L'opérateur >> appliqué au type duck doit effecteur un décalage binaire à droite de *second* d'autant de bits que la valeur de l'opérande entier,
- L'opérateur / appliqué au type duck doit diviser la valeur de *third* par la valeur de l'opérande entier,
- L'opérateur + appliqué au type duck doit additionner la valeur de *fourth* avec la valeur de l'opérande entier.

```
#include <iostream>
using namespace std;

struct duck {
    double first = 16.75;
    short second = 222;
    int third = 525;
    char fourth = '7';
};

ostream& operator<<(ostream& os, const duck& anjou) {
    return os << (char)anjou.first << (char)anjou.second << \
    (char)anjou.third << (char)anjou.fourth << endl;
}

int main()
{
    duck bourgogne;
    bourgogne = bourgogne * 4;
    bourgogne = bourgogne >> 1;
    bourgogne = bourgogne / 5;
    bourgogne += 55;

    cout << bourgogne;

    return 0;
}
```

Question 3.1.2 Surcharge de fonction

Implémentez les fonctions nécessaires pour que le code suivant soit fonctionnel :

- *somme(int, int)* doit retourner la somme des entiers,
- *somme(const char*, const char*)* doit retourner la concaténation des deux chaînes de caractères.

```
#include <iostream>
using namespace std;

int main()
{
    cout << somme(11,11) << somme("v'la", "les flics!");
}
```

Exercice 3.2 Polymorphisme paramétré

Question 3.2.1 Patron de fonction

Écrivez un patron de fonction unique qui applique le chiffrement de César sur un tableau quelque soit le type des éléments contenus.

Le patron de fonction doit s'appeler *cesar* et les paramètres seront les suivants :

1. Le tableau d'éléments de type générique,
2. Le nombre d'éléments du tableau (entier),
3. Le décalage à appliquer (entier);

```
#include <iostream>
using namespace std;

int main()
{
    int li[5]   = {69, 66, 73, 73, 76};
    char lc[5]  = {86, 74, 71, 84, 71};

    cesar(li, 5, 3);
    cesar(lc, 5, -2);

    for (int i = 0; i < 5; i++)
        cout << (char)li[i];
    cout << endl;
    for (int i = 0; i < 5; i++)
        cout << (char)lc[i];
    cout << endl;
}
```

Question 3.2.2 Patron de structure et patron de fonction

Mettez à jour le code précédent pour que les tableaux utilisent les templates. La structure *myArray* devra avoir un template constitué d'un typename et d'un size_t N et les champs suivants :

1. Un tableau de N éléments de type générique.

Vous aurez à mettre à jour la fonction *cesar* qui n'aura plus besoin de son second argument et qui devra prendre en paramètre une structure *myArray* au lieu d'un tableau.

Pour vous aider, voici le code pour initialiser, appeler la fonction *cesar* et afficher les résultats.

```
#include <iostream>
using namespace std;

int main()
{
    myArray<int, 5> li   = {69, 66, 73, 73, 76};
    myArray<char, 5> lc  = {86, 74, 71, 84, 71};

    cesar(li, 3);
    cesar(lc, -2);

    for (int i = 0; i < 5; i++)
        cout << (char)li.array[i];
    cout << endl;
    for (int i = 0; i < 5; i++)
        cout << (char)lc.array[i];
    cout << endl;
}
```