Nom : Ilyass RAMDANI
3A Génie Industriel

[C++] PRALG : Rapport sur le TP Héritage, À l'attention de monsieur Pascal Monasse

Pour ce TP de polymorphisme, je trouve intéressant de présenter étape par étape la construction de mon script et mes apprentissages. J'étoffe donc les points bloquant avec les solutions auxquelles j'ai pu faire appel.

L'un des plus grands apprentissages dans ce TP réside pour moi dans la structuration de l'écriture du code. En effet, outre l'apprentissage concret de la notion d'héritage, le travail fourni pour les précédents TP m'a permis d'acquérir une organisation qui me rend plus productif.

Etape 1 : Création et écriture du fichier fonction.h

J'ai décidé d'écrire tous les types de fonctions au même endroit (dans le fichier fonction.h).

D'un point de vue du workflow, cette manière de procéder me permet de me concentrer par la suite à la création du cpp. En effet, une fois que j'ai écrit toutes les méthodes nécessaires pour chaque classe, je n'ai ensuite qu'à faire des aller-retour entre le .cpp et le .h pour voir les méthodes manquantes et les rédiger. Une fois la méthode rédigée, je la test dans le main. Cela me permet d'être plus structuré, d'aller plus vite dans l'écriture et test du code.

Cela me permet aussi d'avoir de la visibilité sur les différents héritages :

- Fonction → Polynôme → Affine
- Fonction → Trigo

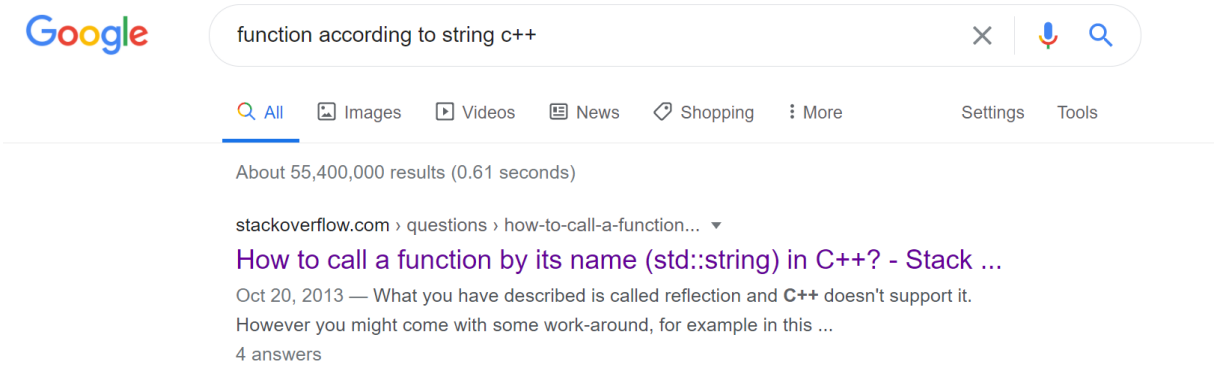
Etape 2 : Ecriture des méthodes classiques dans fonction.cpp (Destructeur, constructeur, operateur ())

L'operateur () fonctionne pour ma fonction affine, dans main : $f1(x) = x + 1$.

L'operateur () fonctionne pour ma fonction polynomiale, dans main : $f2(x) = x^3$.

La difficulté rencontrée dans cette partie étant la construction de l'operateur () pour les fonctions trigonométriques. La fonction trigonométrique étant initialisée avec un string (« cos », « sin », « tan »), il me fallait trouver comment l'associer à la fonction trigo présente dans math.h.

J'ai donc recherché sur Google « Function according to string C++ » qui n'est certainement pas la meilleure formulation.



Mais il faut croire que google a compris ma requête puisque dans le premier lien stackoverflow, la proposition était d'utiliser map de la façon suivante :

```cpp
#include <iostream>
#include <map>

int add(int i, int j) { return i+j; }
int sub(int i, int j) { return i-j; }

typedef int (*FnPtr)(int, int);

int main() {
    // initialization:
    std::map<std::string, FnPtr> myMap;
    myMap["add"] = add;
    myMap["sub"] = sub;

    // usage:
    std::string s("add");
    int res = myMap[s](2,3);
    std::cout << res;
}
```

Je l'ai donc réadapté à notre fonction trigonométrique :

```cpp
float Trigo::operator()(const float x) const{
    map<string, FnPtr> MyMap;
    MyMap["sin"] = sin;
    MyMap["cos"] = cos;
    MyMap["tan"] = tan;
    float res;
    res = MyMap.at(trig)(x);
    MyMap.clear();
    return(res);
};
```

J'introduis FnPtr dans le fonction.h : `typedef float (*FnPtr)(float);`

Après test, L'operateur () fonctionne pour ma fonction trigo, dans main : $f3(x) = \cos(x)$.

Etape 3 : Ecriture de la méthode dérivée

Pour la fonction affine $f(x) = ax + b$, il suffit de retourner la fonction affine constante $f'(x) = a$.

Pour la fonction polynomiale, nous recreons le vecteur des nouveaux coefficients :

Mathématiquement, la dérivée d'un polynome $P(x) = \sum_{i=0}^{n} a_i x^i$ est $P'(x) = \sum_{i=0}^{n-1} a_{i+1}(i+1)x^i$.

Ainsi le vecteur de coefficient pour la dérivée est le vecteur $(a_{i+1}(i+1))$.

```
61  //Derivee
62  Fonction* Polynome::derivee() const{
63      vector<float> coefficientsDerivee;                          // coefficients du polynome derivee
64      vector<float> coefficientsCopie = coefficients;             //copie des coefficients du polynome
65      int i=0;
66      coefficientsCopie.erase(coefficientsCopie.begin());
67      vector<float>::const_iterator it=coefficientsCopie.begin();
68      for (; it!=coefficientsCopie.end(); ++it){
69          coefficientsDerivee.push_back((*it) * (i+1));
70          i++;
71      }
72      return new Polynome(coefficientsDerivee);
73  }
```

Etape 4 : Ecriture de la méthode inverse

En reprenant la méthode de Newton, nous écrivons la méthode inverse. Cette méthode sera générique à tous les descendants de Fonction. Donc nous ne mettons pas de virtual et nous la definissons qu'une fois dans la classe Fonction.

```
13  //Inverse
14  float Fonction::inverse(float y) const{ //xi+1 --> xb and xi --> xa
15      float xb ;
16      float xa =1;
17      float eps = 1;
18      int i=0;
19      Fonction* A = (this)->derivee();
20      while(eps>0.00001 && i<100){
21          xb = xa + (y - (*this)(xa))/(A->operator()(xa));
22          eps = abs(xb-xa);
23          xa=xb;
24          i++;
25      }
26      return xb;
27  }
```

Etape 5 : Généralisons la dérivée à toutes les fonctions

Les méthodes dérivée et inverse fonctionnent pour les fonctions polynomiales et affines. Seulement, elles ne fonctionnent pas pour la fonction trigo par exemple car la dérivée d'une fonction trigonométrique n'est pas une fonction trigonométrique.

Nous devons donc créer une classe Dérivée dotée d'un champ intégrale et écrire la méthode operateur pour calculer la dérivée.

```
//operateur()
float Derivee::operator()(const float x) const{
    const float eps = 0.01;
    float yprime= ((*integrale)(x+eps)-(*integrale)(x-eps))/(2*eps);
    return yprime;
}
```

La méthode dérivée dans la fonction Trigo doit donc retourner une nouvelle fonction de type Dérivée dont le champ intégrale est une copie de l'objet appelé. Nous devons pour éviter ajouter un constructeur virtuel (clone) pour éviter tout soucis de dépendance.

Nous en ajoutons dans chaque classe de la manière simple.

```
Fonction* Derivee::clone()const {
    return new Derivee(*this);
}
```

Notre code fonctionne bien avec différents exemples (cf. main).