TP2 : Introduction au C++ (2)

Préambule

Qu'est-ce que le C++ ?

Le C++ est un langage très populaire qui allie puissance et rapidité. Voici quelques qualités du C++ que nous découvrirons au cours de ces TP :

- Il est très **répandu**. C'est un des langages de programmation les plus utilisés internationalement dans l'industrie et la recherche. Il y a donc beaucoup de documentation sur Internet et on peut facilement trouver de l'aide sur les forums.
- Il est très **rapide** (en temps de calcul) et est donc utilisé pour les applications qui ont besoin de performance (jeux vidéo, outils financiers, simulations numériques ...)
- Il est **portable**, c'est-à-dire qu'un même code source peut être transformé rapidement en exécutable sous Windows, Mac OS et Linux.
- Il existe de nombreuses bibliothèques pour le C++, nous en verrons quelques unes dans ces TP. Les bibliothèques sont des extensions pour le langage car de base, le C++ ne fournit que des outils bas-niveau mais en le combinant avec de bonnes bibliothèques, on peut créer des programmes très puissants.
- Il est **multi-paradigmes**, c'est-à-dire qu'on peut programmer de différentes façons en C++. La plus célèbre est la : **Programmation Orientée Objet (POO)**. Cette technique permet de simplifier l'organisation du code dans les programmes et de rendre facilement certains morceaux de codes réutilisables. Plusieurs TP y seront consacrés.

Quelques liens utiles

N'hésitez pas à les consulter quand vous êtes bloqués ...

- Un polycopié (écrit par K. Santugini)
  htTP://www.math.u-bordeaux.fr/ ksantugi/downloads/CPlusPlus/PolyCxx.pdf
- Quelques sites web :
  http://stackoverflow.com/, http://www.cplusplus.com, http://cpp.developpez.com

Pour la compilation et l'exécution

Pour compiler un fichier main.cc :

```
g++ -std=c++11 -o run main.cc
```

L'exécutable créé ainsi s'appelle run, on le lancera en tapant

```
./run
```

Pour compiler plusieurs fichiers :

```
g++ -std=c++11 -o run TP1.cc main.cc
```

Si l'on souhaite séparer la phase de compilation de la phase d'édition des liens, on exécutera les commandes suivantes :

```
g++ -std=c++11 -c fonctions.cc
g++ -std=c++11 -c main.cc
g++ -std=c++11 -o run fonctions.o main.o
```

Quelques conseils

Bien lire la console quand vous avez des erreurs à la compilation. En effet le compilateur vous donne de nombreuses informations pour vous aider à corriger la syntaxe de votre code. Un exemple :

```
main.cc:20:35: error: expected ';' after expression
    cout << "Hello world!" << endl
                                  ^
                                  ;
1 error generated.
```

Le compilateur nous informe que l'erreur est dans le fichier « main.cc » à la ligne 20 et à la colonne 35. Ensuite il nous informe qu'il attend un « ; » à la fin de la ligne :

```
cout << "Hello world!" << endl
```

N'hésitez pas à **commenter votre code**, cela facilitera la recherche d'erreurs. Pour insérer un commentaire en C++ sur une ligne :

```
// Ceci est un commentaire
```

et sur plusieurs lignes :

```
/* Ceci est un commentaire
sur plusieurs lignes */
```

Pensez à bien **indenter** votre code pour faciliter sa lecture. Avec *Atom* (éditeur qui est conseillé), vous pouvez faire :
Edit > Lines > Auto Indent

Pour ce second TP, de nombreux programmes vous sont donnés, n'hésitez pas à **modifier et/ou enlever** certaines lignes pour voir le rôle de chaque bout de code.

Objectifs

- Structurer son code (.cc, .h, .cpp)
- Les vecteurs
- Lire et écrire des fichiers
- Les pointeurs

Exercices

Exercice 1 - Structurer son code

1. Vous pouvez partir de votre code obtenu à l'exercice 6 du TP 1 ou du code ci-dessous.

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

void euler(double& y, double tn, double dt, double (&f)(double, double))
{
  y += dt*f(tn, y);
}

double f_ty2(double t, double y)
{
  return t*y*y;
}

int main()
{
  cout.precision(15);
  double dt(0.001), Tfinal(1.), y0(-2.), y(y0), tn;
  int N = int(round(Tfinal/dt));
  for (int i = 1; i < N+1; i++)
  {
    tn = i*dt;
    euler(y, tn, dt, f_ty2);
  }

  double y_ext_T_final = 2.*y0/(2. - y0*Tfinal*Tfinal);
  cout << "y'=t*y^2 avec y(0)=" << y0 << endl;
  cout << "y(Tf)=" << y << " and y_exacte(Tf)=" << y_ext_T_final << endl;

  return 0;
}
```

Déplacer les fonctions *euler* et *f_ty2* à la fin du programme. Vous obtenez ce type d'erreur (le message exact varie selon le compilateur) :

```
main.cc:21:22: error: use of undeclared identifier 'f_ty2'
```

2. En effet, afin d'améliorer la correction des erreurs, le C++ impose que toute fonction soit déclarée avant son premier appel. Cependant, la définition peut apparaître plus tard. Laisser les **définitions** des fonctions en bas du fichier mais rajouter au début, à la ligne 4 les **déclarations** :

```
void euler(double& y, double tn, double dt, double (&f)(double, double));
double f_ty2(double t, double z);
```

Cette fois ci la compilation fonctionne car la fonction *main* n'a pas besoin de savoir comment la fonction *euler* fonctionne (son algorithme : sa **définition**) pour l'utiliser (y faire appel) ; elle doit par contre savoir ce qui y rentre et ce qui en sort (en connaître son prototype : sa **déclaration**).

3. Nous allons maintenant pouvoir structurer notre code à la manière classique du C++, c'est-à-dire que nous allons séparer les **déclarations/prototypes** et les **définitions** dans deux fichiers différents. Actuellement, tout notre code est dans le fichier main.cc et nous allons donc créer :

- un header (fichier *.h) qui contiendra les prototypes des fonctions ;
- un fichier source (fichier *.cpp) qui contiendra la définition des fonctions.

Créer deux fichiers nommés respectivement : Fonction.h et Fonction.cpp :
a) Insérer dans le fichier Fonction.h les deux prototypes des fonctions *euler* et *f_ty2* **et rien d'autre ! ! !**
b) Insérer dans le fichier Fonction.cpp les deux définitions des fonctions *euler* et *f_ty2*. Rajouter en haut de votre fichier l'inclusion du fichier ".h" :

```
#include "Fonction.h"
```

c) Enfin dans le main.cc vous pouvez supprimer les prototypes et les définitions des fonctions, à l'exception bien sur de la fonction *main*. Afin d'avoir accès aux fonctions *euler* et *f_ty2* vous devez en haut de votre fichier ajouter (comme pour le fichier .cpp) l'inclusion suivante :

```
#include "Fonction.h"
```

Vous êtes prêts pour la compilation en ajoutant le fichier Fonction.cpp :

```
g++ -std=c++11 -o run main.cc Fonction.cpp
```

4. Regarder votre code c'est important de bien comprendre comment les appels aux différents fichiers sont faits. N'hésitez pas à commenter certaines lignes pour voir les erreurs qui apparaissent. Vous avez remarqué que le fichier qui est inclu est le *.h.

Une bonne structure de code permet d'éviter beaucoup beaucoup d'erreurs !
N'hésitez pas à revenir régulièrement à cet exercice !

Exercice 2 - Les vecteurs

Dans cet exercice nous allons voir les vecteurs qui permettent de contenir différentes variables de même type. Attention en C++ les vecteurs démarrent à l'indice 0 ! Pour les définitions :

```
vector<type> mon_tableau(n); //définitions d'un tableau de taille n
vector<type> mon_tableau(n, xxx); //et dont tous les éléments valent xxx
vector<type> mon_tableau; //et d'un tableau dont la taille est inconnue
```

1. Voici quelques exemples d'utilisation à tester :

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm> // Pour la fonction sort
using namespace std;
int main()
{
  vector<int> x(20); // Un tableau de 20 entiers
  for (int i=0;i<20;i++) {
        x[i]=i*(i+1);
  }
```

```
  int taille_vecteur = x.size() ; // Retourne la taille du tableau x
  x.resize(40); // Le tableau contient 40 éléments
  vector<string> S; // un tableau vide de string
  S.push_back("Pierre"); //Rajoute "Pierre" en fin de tableau.
  S.push_back("Julien"); //Rajoute "Julien" en fin de tableau.
  S.push_back("Paul"); //Rajoute "Paul" en fin de tableau.
  sort(S.begin(),S.end());
  // Les string sont triés par ordre alphabétique
  for (int i=0;i<S.size();++i)
        cout << "Bonjour " << S[i] << endl;
}
```

Attention l'accès aux valeurs d'un tableau par x[i] ne vérifie pas si on déborde ou non du tableau. Pour un accès moins rapide mais qui vérifie les bornes du tableau, il est possible d'utiliser x.at(i).

2. Revenir au code de l'exercice 1. Construire deux vecteurs *temps* et *solutions* qui contiennent respectivement t_n et $z(t_n)$.

Exercice 3 - Écrire dans un fichier

1. Nous allons dans cet exercice sauvegarder les vecteurs t_n et $z(t_n)$ dans un fichier afin de pouvoir, par exemple, tracer la solution. Nous allons pour cela utiliser la librairie "fstream" (file stream). Tester le code suivant sans oublier de regarder le fichier créé.

```
#include <vector>
#include <iostream>
#include <fstream> // Pour pouvoir écrire et lire des fichiers
#include <string>
using namespace std;
int main()
{
  vector<double> vector_constant(10,1.1);
  ofstream mon_flux; // Contruit un objet "ofstream"
  string name_file("mon_fichier.txt"); // Le nom de mon fichier
  mon_flux.open(name_file, ios::out); // Ouvre un fichier appelé name_file
  if(mon_flux) // Vérifie que le fichier est bien ouvert
  {
    for (int i = 0 ; i < vector_constant.size() ; i++)
      mon_flux << vector_constant[i] << endl; // Remplit le fichier
  }
  else // Renvoie un message d'erreur si ce n'est pas le cas
  {
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
  }
  mon_flux.close(); // Ferme le fichier
}
```

2. Construire un fichier formé de 2 colonnes correspondant respectivement à t_n et $z(t_n)$. Tracer ensuite la courbe en utilisant gnuplot :

```
plot "ma_solution.txt" title "solution approchee" with linespoints
```

Ajouter une troisième colonne contenant la solution exacte. Tracer les 2 courbes :

```
plot "ma_solution.txt" using 1:2 title "solution approchee" with linespoints
replot "ma_solution.txt" using 1:3 title "solution exacte" with linespoints
```

Exercice 4 - Lire un fichier

1. À présent nous allons apprendre à lire un fichier. Ce qu'il faut comprendre c'est qu'on lit le fichier en le parcourant au fur et à mesure donc à chaque fois qu'on lit soit une ligne, soit un mot, soit un caractère, le curseur de lecture se retrouve derrière le dernier élément lu. Tester (et comprendre) le code suivant pour la lecture du fichier que nous venons de créer :

```
#include <iostream>
#include <fstream> // Inclusion de "fstream"
using namespace std;

int main()
{
  //Ouvre un fichier
  ifstream mon_flux("ma_solution.txt");

  // 1ère façon de lire : ligne par ligne
  string ligne;
  getline(mon_flux, ligne);
  // On lit la première ligne : 0 -2 -2
  cout << "ligne 1 : " << ligne << endl;
  getline(mon_flux, ligne);
  // On lit la seconde : 0.001 -2 -2
  cout << "ligne 2 : " << ligne << endl; // etc ...

  // 2ème façon de lire : mot par mot (comme cin)
  double nombre1, nombre2;
  mon_flux >> nombre1 >> nombre2;
  // On lit les 2 premiers mots sur la ligne 3 : 0.002 et -1.99999
  cout << "1er mot de la ligne 3 : " << nombre1 << endl;
  cout << "2eme mot de la ligne 3 : " << nombre2 << endl;

  // 3ème façon de lire : caractère par caractère
  char a, b;
  mon_flux.get(a);
  mon_flux.get(b);
  // On lit le premier caractère sur la ligne 3 après -1.99999 : un espace
  // (donc rien ne s'affiche!)
  cout << "premier caractère après premier mot de la ligne 3 : " << a << endl;
  // On lit le caractère après l'espace : -
  // (correspondant au signe de la valeur de la solution exacte)
  cout << "caractère suivant : " << b << endl;
  mon_flux.close();
}
```

2. Reconstruire les trois vecteurs *temps*, *solution_ approchee* et *solution_ exacte* à partir du fichier que vous avez créé à l'exercice précédent. Les deux fonctions *eof* et *good* présentées dans le bout de code suivant devront être utilisées :

```
double a;
//Tant que je ne suis pas à la fin du fichier
while (!mon_flux.eof())
{
  mon_flux >> a;
  if (mon_flux.good())
  {
    // La fonction good() renvoie true si le flux est toujours
    // operationnel (pas de fin de fichier, ni de lecture incorrecte)
    // Cela vérifie donc que la valeur a est bien un double et qu'il
    // est possible de s'en servir dans la suite du code
  }
}
```

Vérifier votre lecture en écrivant un deuxième fichier à partir de ces trois vecteurs. Vous pouvez comparer vos deux fichiers avec la commande *diff*.

Exercice 5 - Les pointeurs

1. L'utilisation des pointeurs est une notion complexe donc accrochez vous ! Les pointeurs sont utilisés dans tous les programmes et ils permettent de gérer très finement ce qui se passe dans la mémoire de l'ordinateur. La mémoire d'un ordinateur est constituée de *cases* (plusieurs milliards sur un ordinateur récent). Il faut donc un système pour que l'ordinateur puisse retrouver les cases dont il a besoin. Chacune d'entre elles possède un numéro unique que l'on appelle son adresse. Donc la variable *mon_ entier* définie par la commande suivante :

```
int mon_entier = 23;
```

a pour adresse *133655* d'après la figure suivante.

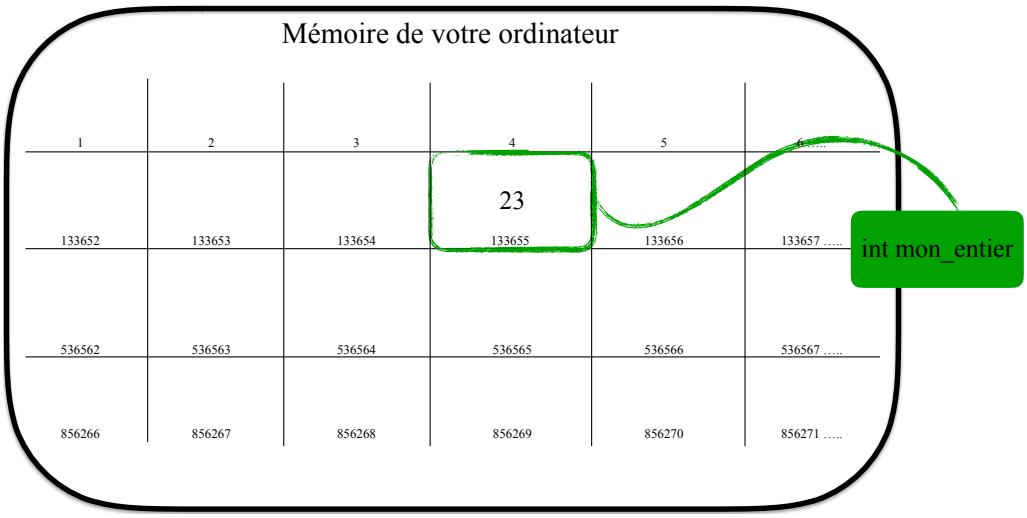

FIGURE 1 – Schéma de la mémoire de votre ordinateur

L'adresse est donc un deuxième moyen d'accéder à une variable. Tester le code suivant.

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

int main()
{
   int mon_entier(23);
   //Affichage de l'adresse de la variable en utilisant &
   //Elle sera donnée en base 16 (d'où la présence de lettres)
   cout << "L'adresse est : " << &mon_entier << endl;
   return 0;
}
```

Nous allons maintenant pouvoir définir ce qu'est un pointeur :

Un pointeur est une variable qui contient une adresse mémoire (généralement celle d'une autre variable).

Pour comprendre comment cela fonctionne rajouter les quelques lignes suivantes :

```cpp
//Ce code déclare un pointeur qui peut contenir
//l'adresse d'une variable de type int.
int *pointeurInt;
//On peut faire de même pour n'importe quel type :
double const *pointeurDoubleConst;
vector<int> *pointeurVector; //etc ...
```

Pour le moment, ces pointeurs ne contiennent aucune adresse connue. C'est une situation très dangereuse. Si vous essayez d'utiliser le pointeur, vous ne savez pas quelle case de la mémoire vous manipulez. Pire : elle contient une adresse quelconque (pas nécessairement "0" !), ce qui est une source très fréquente de bug et de faille de sécurité. Il ne faut donc jamais déclarer un pointeur sans lui donner d'adresse. Pour être tranquille, il faut toujours déclarer un pointeur en lui donnant la valeur 0. En effet, sur la figure ci-dessus, la première case de la mémoire avait l'adresse 1 car l'adresse 0 n'existe pas. Lorsque vous créez un pointeur contenant l'adresse 0, cela signifie par convention qu'il ne contient l'adresse d'aucune case. Corriger le code précédent :

```cpp
//Ce code déclare un pointeur qui peut contenir
//l'adresse d'une variable de type int.
int *pointeurInt(0);
//On peut faire de même pour n'importe quel type :
double const *pointeurDoubleConst(0);
vector<int> *pointeurVector(0); //etc ...
```

Maintenant qu'on a la variable, il est possible de lui affecter une valeur :

```cpp
int mon_entier(23); //Une variable de type int
int *ptr(0); //Un pointeur pouvant contenir l'adresse d'un int
ptr = &mon_entier; //L'adresse de 'mon_entier' est mise dans le pointeur 'ptr'
//On dit alors que le pointeur ptr pointe sur mon_entier
cout << "L'adresse de 'mon_entier' est : " << &mon_entier << endl;
cout << "La valeur de 'ptr' est : " << ptr << endl;
cout << "La valeur est : " << *ptr << endl;
```

Décrire les étapes qui sont effectuées par la machine à la ligne 7.

Avant de voir à quoi cela peut bien servir (non le but n'est pas juste de se compliquer la vie), nous allons faire un petit récapitulatif.

Pour une variable **int nombre** :
- nombre permet d'accéder à la valeur de la variable,
- &nombre permet d'accéder à l'adresse de la variable.

Pour un pointeur **int *pointeur** :
- pointeur permet d'accéder à la valeur du pointeur, c-à-d à l'adresse de la variable pointée,
- *pointeur permet d'accéder à la valeur de la variable pointée.

Donc on a : "nombre égale *pointeur" et "&nombre égale pointeur"... Dans les questions suivantes, nous allons répondre à la question suivante : À quoi cela peut bien servir ?

2. **Une première utilisation : l'allocation dynamique.** Lors de la déclaration d'une variable, le programme effectue deux étapes :
- Il demande à l'ordinateur de lui allouer une zone de la mémoire.
- Il initialise la case avec la valeur fournie. Attention : si rien n'est fourni, il n'y a pas nécessairement de valeur par défaut.

Tout cela est entièrement automatique. De même, lorsque l'on arrive à la fin d'une fonction, le programme rend la mémoire utilisée à l'ordinateur. C'est ce qu'on appelle la libération de la mémoire. C'est à nouveau automatique. Les pointeurs permettent de faire tout ça manuellement.

```
1 int *pointeur(0);//Définit un pointeur pouvant contenir l'adresse d'un int
2 pointeur = new int;//Demande 1 case et renvoie 1 pointeur pointant vers celle-ci
3 *pointeur = 2;//Accède à la case mémoire pour en modifier la valeur
4 delete pointeur;//Libère la case mémoire
5 //Attention le pointeur pointe toujours mais vers une case vide !!!
6 pointeur = 0;//Indique que le pointeur ne pointe plus vers rien
```

Faire un programme demandant l'âge de l'utilisateur et l'affichant à l'aide d'un pointeur. C'est plus compliqué que la version sans allocation dynamique mais la mémoire est contrôlée. Vous devez définir qu'une seule variable qui est votre pointeur !
Rappel : Une case peut être réservée dans la mémoire manuellement avec *new*. Dans ce cas, il ne faut pas oublier de libérer l'espace en mémoire, avec *delete*.

3. **Une deuxième utilisation : choisir parmi plusieurs éléments**
a. Tester le QCM suivant.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7    string repA("Euler"), repB("Runge-Kutta");
8    cout << "Quel schéma en temps souhaitez-vous utiliser ? " << endl;
9    cout << "A) " << repA << endl;
10   cout << "B) " << repB << endl;
11
12   char votre_reponse;
13   cout << "Votre reponse (A ou B) : ";
```

```
  cin >> votre_reponse; //Récupère la réponse de l'utilisateur

  string *reponseUtilisateur(0); //Un pointeur qui pointera sur la réponse

  switch(votre_reponse)
  {
  case 'A':
      reponseUtilisateur = &repA; //Déplace le pointeur sur la réponse choisie
      break;
  case 'B':
      reponseUtilisateur = &repB;
      break;
  default:
      cout << "Ce choix n'est pas valable !" << endl;
      exit(0); //Le programme s'arrête
  }
  //Utilise le pointeur pour afficher la réponse choisie
  cout << "Vous avez choisi la reponse : " << *reponseUtilisateur << endl;
  //Le pointeur n'ayant pas été défini par un "new"
  //il n'est pas nécessaire de libérer la case mémoire
  return 0;
}
```

b. Que se passe t-il si on retire la ligne 28 et que l'on ne répond ni par A ni par B ?
c. L'exemple précédent est intéressant mais n'est pas vraiment utile. En effet, il est possible de juste copier un string sans le passer par un pointeur ! Pour un exemple plus interessant, nous allons reprendre le code de l'exercice 1.
i. Ajouter la fonction suivante dans le code (prototype dans le Fonction.h et définition dans le Fonction.cpp) :

```
double f_y(double t, double y)
{
 return y;
}
```

ii. Modifier votre fonction main pour pouvoir demander à l'utilisateur de choisir entre résoudre $y' = y$ ou $y' = y^2 t$. Pour cela, construire le pointeur suivant :

```
double (*fct)(double, double);
```

et le définir dans un *switch*.

4. **Une troisième utilisation : partager une variable dans plusieurs parties du code.** Cela sera très utile quand nous commencerons la programmation orientée objet (prochain TP).