Programmation pour le calcul scientifique
Année : 2019-2020
Formation : L3 Ingénierie Mathématique

TP3 : C++ : fonctionnalités de base, suite.

1 Arguments de la fonction main.

1. Dans l'exercice du TP précédent sur l'intégration numérique [1], le nom du fichier de résultats ainsi que les pas de temps sont codés « en dur ». On aimerait cependant avoir un peu plus de contrôle sur ces données au moment de lancer le programme. Modifiez le code pour que l'utilisateur donne le nom de fichier de sorties suivi des pas de temps, comme ceci :

```
$> ./monProgramme nomFichier dt1 dt2 dt3 ...
```

   Quelques conseils :
   - Placez la lecture de la ligne de commande dans une fonction qui modifiera une chaîne de caractères et remplira le tableau `dts` contenant les pas de temps.
   - Pensez à l'utilisateur qui ne connait pas le fonctionnement de votre programme. Ce dernier doit donner un exemple de ligne de commande correcte en cas d'erreur.
   - Le tableau `argv` ne contient que des chaînes de caractères « à la C », et non des réels. Vous pourrez utiliser la fonction stod (string to double) issue du standard de 2011. N'oubliez pas l'option `-std=c++11` quand vous compilerez !
2. À l'aide de la documentation de `GNU getopt()`, qui est une bibliothèque C [2], modifiez la fonction de lecture de la ligne de commande pour que celle-ci respecte d'avantage des options standard :
   - `-h` affichera l'aide pour lancer correctement le programme
   - `-o nomFichier` pour donner le nom du fichiers de résultats
   - Les pas de temps seront laissés sans option.

2 Signal de retour.

1. Écrire une fonction main qui ne fait rien d'autre que retourner 0. Compilez et vérifiez que l'éxécution se déroule bien en affichant le code de sortie de la dernière commande. Ceci se fait avec la commande

```
$> echo $?
```

2. Comparez le code de sortie lorsque la seule instruction de votre programme est
   - `return 1;`
   - `exit(EXIT_SUCCESS);`
   - `exit(EXIT_FAILURE);`

1. Téléchargez la correction si nécéssaire
2. Il existe bien sûr d'autres façons de faire qui sont d'avantage dans l'esprit du C++, mais celle-ci est un « classique ».

- `abort();`

`exit`, `EXIT_SUCCESS`, `EXIT_FAILURE` et `abort` sont définis dans la bibliothèque C `stdlib.h`.

Les codes de sorties peuvent être utilisés lorsque vous lancez votre programme de manière automatique avec un script. Vous pouvez alors décider de quelle suite à donner à une éventuelle mauvaise éxécution de votre programme. Pour aller plus loin sur ce sujet, vous pourrez vous renseigner sur la structure de contrôle `try - catch` qui est destinée à gérer les erreurs au sein du code.

3 Fonctions : arguments par défaut.

Soit le programme suivant :

exo3.cpp

```
#include <vector>
#include <iostream>

void initialize(char* argv[], std::vector<std::vector<double>>& data,
                bool readEntries = false, int nArrays = 2) {
  // We suppose argv is large enough

  data.resize(nArrays);
  if (readEntries)
    for (int i=0;i<nArrays;i++)
      data[i] = std::vector<double> (100,std::stod(argv[i]));
  else
    for (int i=0;i<nArrays;i++)
      data[i] = std::vector<double> (100,i);
}


int main(int argc, char* argv[]) {

  std::vector<std::vector<double>> v;

  initialize(argv,v);

  std::cout << "v.size() : " << v.size() << std::endl;
  for (int i=0; i<v.size(); i++)
    std::cout << "v[ " << i << "][0] : " << v[i][0] << std::endl;

  // Do something with this data...

  return 0;

}
```

1. Que fait la fonction `initialize` ? Compilez le programme tel quel et lancez-le.
2. Dans le but de ne créer qu'un seul tableau, l'utilisateur indique dans la fonction `main` :

```
22  initialize(argv,v,1);
```

Que se passe-t-il ? Comment expliquez-vous cela ?

4 `switch` et types non entiers

Le structure de contrôle `switch` n'est compatible qu'avec des types assimilables aux entiers (entiers, caractères, pointeurs). Admettons que l'utilisateur choisisse un solveur d'algèbre linéaire au cours du programme (LU, gradient conjugué, GMRES, etc). On aurait

```
int solverType;
// ... here get input for solverType, whether from std::cin or input file

switch (solverType) {
  case 0:
    // ...
    break;
  case 1:
    // ...
    break;
  case 2:
    // ...
    break;
  default:
    std::cerr << "Unknown solverType: " << solverType << std::endl;
}
```

Sachant que ces blocs d'instructions peuvent être éloignées au sein du code, ce n'est pas très clair quand on définit `solverType` que 0 veut dire gradient conjugué, 1 la factorisation LU, etc. Pour que le code reste lisible sans ajouter trop de commentaires, on aurait envie de faire

```
std::string solverType;
// ... here get input for solverType, whether from std::cin or input file
switch (solverType) {
  case "CG":
  case "conjugate gradient":
    // ...
    break;
  case "LU":
    // ...
    break;
  case "Choleski":
    // ...
    break;
  default:
    std::cerr << "Unknown solverType: " << solverType << std::endl;
}
```

Mais ceci n'est pas valide, puisqu'on teste dans un `switch` une variable de type `std::string`. Il existe plusieurs façons de pallier cela. Une solution consiste à utiliser le type `enum`. Cette fonctionnalité permet de créer de nouveaux types, associés aux entiers, qui ne peuvent prendre qu'un nombre limité de valeurs. Voici un exemple d'`enum` :

```
// --------------------------------
// Unscoped enum:

enum color {
  red, blue, green
  // By default, red is 0, blue is 1, green is 2.
};

// declare a variable of type color:
color myColor = red; // behind the scenes, myColor is an integer equal to 0
```

```
int i = 1;
bool isRed = (i==red); // This is valid, int and color types are compatible

// --------------------------------
// Scoped enum, note the difference with the 'struct' keyword:

enum struct color2 {
  red=20, blue=30, green=40
  // you can specify your own values (also in unscoped enums)
};

// declare a variable of type color2:
color2 otherColor = color2::blue; // we have to indicate the name of the enum
                                  // before the enumerator
                                  // => avoids confusion and multiple definitions
```

Écrivez et testez une structure `switch` où la variable `solverType` sera de type associé à une `enum` (remplacez les instructions commentées`// ...` par un simple affichage à l'écran pour vous assurer de la bonne sélection).

5 Gradient conjugué

S'il vous reste du temps, à partir du programme sur les matrices creuses, implémentez l'algorithme du gradient conjugué pour résoudre un système linéaire $AX = B$ où A est creuse. Créez vous un cas test, par exemple une matrice de l'équation de la chaleur en 1D avec condition de Dirichlet homogène au bord.

Rappel : si $u_i = u(x_i)$, $f_i = f(x_i)$ pour $i \in [\![0, N]\!]$, la relation $\partial_t u - \Delta u = f$ se discrétise en différence finies centrées d'ordre 2 par :

$$\frac{u_i^{n+1}}{\delta t} - \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\delta x^2} = \frac{u_i^n}{\delta t} + f_i.$$

La condition de Dirichlet impose $u_0 = u_N = 0$. Vous pourrez tester avec $u(t = 0, x) = \sin(x)$ pour x dans $[0, \pi]$ et $f = 0$. La solution en tout temps est donnée par $u(t, x) = \mathrm{e}^{-t} \sin(x)$.