© *Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>*

Fiche de TP1 : Programmation structurée en C++

Le but de ce TP est de passer en relief les outils fournis par le C++ , pour structurer les programmes. Pour gagner du temps dans l'écriture des bouts de programmes fournis dans les exercices, nous rappelons que les fichiers sources de ces programmes sont téléchargeables à l'adresse **/doc/apoung/M2_IM**

Thème - 1 *Énumération - Tableaux*

Exercice-1 **:** On considère le programme suivant suivant

Listing 1 – fichier :enumeration.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  enum E{a,b,c=0,d,e};
5  int main(int argc, char** argv)
6  {
7    cout<< " a= " << a <<endl;
8    cout<< " b= " << b <<endl;
9    cout<< " c= " << c <<endl;
10   cout<< " d= " << d <<endl;
11   cout<< " e= " << e <<endl;
12 return 0;
13 }
```

Q-1-1 **:** Qu'affiche ce programme ?

Q-1-2 **:** Modifier le programme en mettant cette fois

Listing 2 – fichier : enumeration.cpp

```
enum E{a,b,c,d,e};
```

Qu'observez-vous ?

Exercice-2 **:**

Q-2-1 **:** On considère le programme suivant

Listing 3 – fichier :tableau_ivalue.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main(int argc, char** argv)
4  {
5    int tab[3]={1,2,3};
6    *(tab + 1) = -1;
7    for(int i=0; i<3; i++){tab++;}
8  return 0;
9  }
```

Ce programme compile-t-il ? justifier votre réponse.

Q-2-2 : On considère le programme suivant, dire s'il compile. Justifier la réponse.

Listing 4 – fichier :tableau_const.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char** argv)
5  {
6    int tab[3]={1,2,3};
7    const int *ptab1 = tab;
8    const int *  const ptab2 = tab;
9    const int ctab[3] = {4,5,6};
10   *(ptab1 + 1) = -1;
11   *(ptab2 + 1) = -1;
12   ctab[1] = -1;
13   for(int i=0; i< 3; i++}{ptab1++ ; ptab2++;}
14   return 0;
15 }
```

Thème - 2 *Fonctions*

Exercice-1 :

On considère le programme suivant

Listing 5 – fichier :tableau_argument.cpp

```
1  #include <iostream>
2  using namespace std;
3  //FONCTION D'AFFICHAGE
4  void afficheTableau(int t[],int size)
5  {
6    for(int i=0; i< size; i++) cout<<t[i]<<",\t";
7    cout<<endl;
```

```
}
//FONCTION D'INCREMENTATION DU CONTENU
void incrementeTableau(int t[]){
   int size = sizeof(t)/sizeof(*t);
  for(int i=0; i< size; i++) { t[i]++;}
}
// EVALUATION
int main(int argc, char** argv)
{
  int tab[3]={1,2,3};

  cout<<" tab avant : \n";
  afficheTableau(tab,3);

  incrementeTableau(tab);

  cout<<" t apres : \n";
  afficheTableau(tab,3);

return 0;
}
```

1. La fonction **incrementeTableau** réalise-t-elle le but visé ?
2. Que se passerait-il si on remplaçait dans la fonction **main**, la fonction **incrementeTableau** par son contenu ?
3. Comment peut-on corriger la fonction **incrementeTableau** pour qu'elle réalise le but visé ?

Exercice-2 : Soit la fonction mathématique f définie par $f(x) = \frac{(2x^2+3)(x^2-1)}{3x^2+1}$.

Q-2-1 : Écrire une fonction **C++** qui retourne la valeur de $f(x)$ pour un point x passé en paramètre.

Q-2-2 : Une approximation de la dérivée f' de la fonction f est donnée en chaque point x, pour h assez petit (proche de 0) par :

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h}$$

Écrire une fonction **C++** qui calcule une approximation de la dérivée f' de f en un point x entré au clavier. On passera la valeur de h en paramètre de la fonction.

Q-2-3 : La dérivée seconde de f est la dérivée de la dérivée. Écrire une fonction qui calcule une approximation de la dérivée seconde de f'' de f en un point x entré au clavier. On passera la valeur de h en paramètre de la fonction.

Q-2-4 : Écrire une fonction **C++** qui détermine le signe de la dérivée seconde de f en fonction de x. On pourra faire un programme principal qui lit x entré au clavier et affiche le résultat.

Q-2-5 : Écrire une fonction **C++** qui donne le choix à l'utilisateur d'afficher la valeur de la fonction f, de sa dérivée première ou de sa dérivée seconde en un point x lu au clavier.

Thème - 3 *Structures-modules*

Exercice-1 **:** Créer une structure **`R2`** pour la gestion des éléments de $\mathbb{R}^2$.

Cette structure devra contenir deux **`double`**, pour l'abscisse et l'ordonnée.

Q-1**-**1 **:** Fournir une fonction **`void init(R2& r, double x=0, double y=0)`** qui initialise la structure en lui assignant comme abscisse et ordonnée respectivement les deux derniers arguments de la fonction.

Q-1**-**2 **:** Sachant que les coordonnées dans la structure peuvent être stockées de plusieurs manières, fournir des fonctions d'accès aux coordonnées :
`double getX(const R2& r), double getY(const R2& r)`
ainsi que des fonctions de modifications :
`void setX(R2& r, double x), double setY(R2&, double y)`. (*Rendre inline celles qui sont nécessaires en le justifiant.*)

On utilisera exclusivement ces fonctions dans l'implémentation de celles qui vont suivre.

Q-1**-**3 **:** Fournir une fonction **`void affiche(const R2& r)`** qui affiche sous la forme **`(abscisse, ordonnée)`** un objet de type R2.

Q-1**-**4 **:** Fournir une fonction **`double norm(const R2& r)`** qui calcule la norme d'un objet de type R2.

Q-1**-**5 **:** Fournir une fonction **`void copy(R2& dest, const R2& orig)`** qui copie le contenu de la variable **`orig`** dans la variable **`dest`**.

Q-1**-**6 **:** En vous servant de l'approche adoptée dans la déclaration de **`copy`**, Programmer des fonction pour la réalisation des opérations standards (**`somme, différence, ajout, retranchement, produit scalaire, produit vectoriel, produit par un scalaire, ...`**), qui rendent possible l'algèbre linéaire sur les objets de type R2.

Q-1**-**7 **:** Démontrer le fonctionnement dans un programme principal.

Exercice-2 **:** Création d'un module

Q-2**-**1 **:** Structurer le code ci-dessus pour une compilation séparée : on créera une fichier **`R2.hpp`** qui contiendra les déclarations et un fichier **`R2.cpp`** qui contiendra les définitions.

Q-2**-**2 **:** Tester cette modification dans un programme principale écrit dans un fichier **`testR2.cpp`**. On pourra utiliser pour cette fin une fichier Makefile pour la compilation (*Voir la fin de cette fiche*).

Q-2**-**3 **:** Pour éviter les collisions de noms, ajouter un espace de nom (*namespace*), a votre développement, et démontrer le bon fonctionnement de l'ensemble dans un programme principale.

Q-2**-**4 **:** Modifier le fichier Makefile de sorte à faire de votre module R2 une bibliothèque.

Thème - 4 *Compilation à l'aide d'un Makefile*

Lorsque le nombre de fichiers d'un projet devient important, il est impensable de compiler "à la main". Tout d'abord pour une raison pratique mais essentiellement à cause des dépendances entre les fichiers sources : modifier certains fichiers peut imposer la compilation de tous les fichiers. Le rôle d'un **Makefile** est de gérer les règles de compilation d'un programme (quel qu'il soit).

Il existe de nombreux outils de génération de **Makefile**
– Outils de développement intégrés,
– **Cmake** **http://www.cmake.org**,
– **autotools (GNU)** **http://www.sourceware.org/autobook**,
– ...
La syntaxe d'un **Makefile** est assez simple et repose sur l'utilisation de règles et de dépendances. Il fonctionne en comparant les dates de création des fichiers. Voici un exemple :

Listing 6 –

```
hello.o : hello.cpp
        g++ -c hello.cpp

hello: hello.o
       g++ -o hello hello.o

clean:
      \rm *.o hello

all: hello
```

Ici, le fichier **hello.o** dépend de **hello.cpp**. La ligne suivante est la règle pour fabriquer **hello.o**. Le fichier (exécutable) **hello** dépend de **hello.o**. Enfin **all** qui est la *cible* par défaut dépend de **hello**. Si on modifie **hello.cpp** les fichiers **hello.o** et **hello** sont générés si on utilise la commande

make

Si on souhaite créer une cible particulière, il suffit de passer son nom en argument :

make clean
make hello.o

Remarque. *Attention, dans un* ***Makefile*** *les espaces sont significatifs ! Ainsi, les dépendances sont à indiquer sur la même ligne que la cible ; les règles viennent sur les lignes suivantes et commencent par une* ***tabulation****. Une ligne blanche marque la fin de l'instruction.*

Voici un autre exemple

Listing 7 –

```
# // EXEMPLE SIMPLE DE MAKEFILE
# // objets
OBJETS    =
# // variables
CXX       = g++
CXXFLAGS  = -g
CXXINC    = -I.
CXXLIBS   = -lm
# // sources
hello :%:%.o $(OBJETS)
        $(CXX) $(OBJETS) $(CXXLIBS) -o $@ $<
```

```
# // suffixes
.SUFIXES: .cpp .o
.cpp.o:
        $(CXX) $(CXXFLAGS) $(CXXINC) -c $<
.PHONY: clean
clean:
        \rm -rf  hello *.o *~  *.backup
# //dependencces
hello.o: hello.cpp
```

Pour ajouter un nouveau fichier (**autreExemple.cpp**), il suffit de remplacer la ligne 13 par

```
autreExemple hello:%:%.o $(OBJETS)
```

puis mettre an oeuvre la dépendance en ajoutant en fin du fichier la ligne

```
autreExemple.o: autreExemple.cpp
```

Ainsi la commande

make autreExemple

aura pour effet de générer l'exécutable **autreExemple**.

Exercice-1 : Créer un fichier **Makefile** pour tous les programmes écrits jusqu' ici ainsi que pour les exercice à venir. Cela signifie notamment que chaque exercice doit avoir son propre répertoire.

Thème - 5 *Brève introduction au débogueur GDB*

Pour utiliser le débogueur **GDB**, il faut tout d'abord compiler son code en mode **debug**, en utilisant l'option **-g**, voir plus haut (Thème 1 de cette fiche).

Supposons que l'exécutable se nomme **mon-exe**. On peut alors rentrer la commande

gdb mon-exe

sous l'invite gdb, on peut exécuter un certain nombre de commandes dont les plus essentielles sont listées ci-dessous

- **run** lance ou relance le programme à déboguer (il est suivi des paramètres de la ligne de commande du programme)
- **c** continue l'exécution
- **s** fait un pas d'une instruction (en entrant dans les fonctions)
- **n** fait un pas d'une instruction (sans entrer dans les fonctions)
- **finish** continue l'exécution jusqu'à la sortie de la sortie de la fonction (return)
- **u** sort de la boucle courante

– **p** affiche la valeur d'une variable, expression ou tableau : **p x** ou **p *v@100** (affiche les 100 valeurs du tableau défini par le pointeur v).
– **where** montre la pile des appels
– up monte dans la pile des appels
– **down** descend dans la pile des appels
– **l** listing de la fonction courante
– **l 5** listing à partir de la ligne 5
– **info functions** affiche toutes les fonctions connues, et **info functions tyty** n'affiche que les fonctions dont le nom contient la chaîne "tyty".
– **info variables** même chose mais pour les variables
– **b main.cpp :100** définit un point d'arrêt en ligne 100 du fichier **main.cpp**
– **b zzz** définit un point d'arrêt à l'entrée de la fonction zzz.
– **d 5** détruit le 5-eme point d'arrêt
– **step** exécute la ligne suivante
– **watch** définit une variable à tracer, le programme va s'arrêter quand cette variable va changer.
– **help** pour avoir plus d'information en anglais.
– **quit** ou **q** quitter le débogueur.

Exercice-1 : Prendre un des programmes écrit précédemment. Ajouter, juste avant la fonction **main** la fonction

Listing 8 –

```
void myexit(){ cout<<"fin du programme"<<endl;}
int main(int argc, char** argv){
```

et dans la fonction **main**, juste avant l'instruction **return**, ajouter la ligne

Listing 9 –

```
  atexit(myexit);
  return 0;
}
```

Attention ! la fonction **atexit** se trouve dans le fichier entête **<cstdlib>**

Compiler le code en mode **debug**.

Sous **gdb** entrer la commande

b myexit

Afficher le contenu du code (commande **l**) et de certaines variables (commande **p**). Tester d'autres commandes.