Fiche de TP1 : Notion de base de C++

Le but de ce TP est de passer en relief les notions de base du C++. 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
http ://www.math.u-psud.fr/~apoung/mywepage/M2/codeTP1.tar.gz

Thème - *1 Rappels sur la compilation*

Comme nous l'avons vu en cours, la compilation se déroule en plusieurs étapes. Cependant, dans la pratique, l'utilisateur n'en perçoit que deux. La compilation proprement dite et l'édition des liens. Supposons donc qu'on veuille compiler un programme contenu dans le fichier (**`fichier.cpp`**). Une commande de compilation s'écrit simplement

```
g++ -c fichier.cpp
```

Si la compilation se déroule correctement, le fichier objet **`fichier.o`** est créé. Reste maintenant à éditer les liens. C'est cette fois la commande

```
g++ -o nom-d-executable fichier.o
```

qui s'en charge.

Notons que ces commandes sont des versions minimales au sens où il est soit possible soit nécessaire d'ajouter des options. Par exemple, pour compiler une version optimisée (de niveau 2) du code, on peut utiliser

```
g++ -c -O2 fichier.cpp
```

Pour générer une version *debug* du programme, on utilise

```
g++ -c -g fichier.cpp
```

De même, pour l'édition des liens il est nécessaire de spécifier les bibliothèques qui ont été utilisées. Par exemple, si on a utilisé la bibliothèque mathématique (utilisation d'un sinus par exemple), il faut utiliser la commande

```
g++ -o nom-d-executable fichier.o -lm
```

Thème - *2 Chaînes de caractères*

Exercice-*1* : *Dans un fichier appelé par exemple* **hello.cpp**, *écrire un programme qui affiche*

```
Hello world!
```

à la console.

Exercice-*2* : *Écrire un programme qui affiche les différents arguments passés à la fonction* **main**. *Par exemple, si l'exécutable s'appelle* **execname**, *la commande*

```
execname 1 deux 3
```

affichera le résultat suivant à la console

```
execname
1
deux
3
```

Pour cet exercice, on rappelle les arguments de la fonction `main` *:*

```
in main(int argc,char** argv)
```

- `argc` *est le nombre d'arguments passés en ligne de commande et,*
- `argv` *est le tableau des chaînes de caractères.*

Exercice-*3* : *Sur les mêmes bases, écrire un programme qui écrit les arguments à l'envers. Cette fois, l'exécution de*

```
execname 1 deux 3
```

affichera à la console,

```
3
deux
1
execname
```

Exercice-*4* : *En suivant les mêmes principes et ceci afin de manipuler encore les chaînes de caractères, écrire cette fois un programme qui écrit chaque argument à l'envers, de sorte à obtenir*

```
emancexe
1
xued
3
```

Proposer deux versions. Une première comptant les caractères de chaque mot et une seconde basée sur la récursivité.

Exercice-*5* : *Afin d'assimiler le lien qui existe entre les* `chars` *et les nombres entiers, on propose maintenant d'écrire un programme qui affiche les arguments en mettant la première lettre de chaque argument en majuscule. Pour cela, on se servira du fait que dans la table des caractères ASCII la distance entre les majuscules et les minuscules est constante. De ce fait, on a*

$'A' - 'a' == 'Z' - 'z'$

Si on écrit

```
execname un deux trois
```

On obtiendra

```
Execname
Un
Deux
Trois
```

Modifiez votre programme pour que

```
execname 1 Deux trois
```

affiche

```
Execname
1
Deux
Trois
```

Thème - *3 Énumération - Tableaux*

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

fichier : enumeration.cpp

```
#include <iostream>

using namespace std;

enum E{a,b,c=0,d,e};

int main(int argc, char** argv)
{
  cout<< " a= " << a <<endl;
  cout<< " b= " << b <<endl;
  cout<< " c= " << c <<endl;
  cout<< " d= " << d <<endl;
  cout<< " e= " << e <<endl;

return 0;
}
```

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

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

fichier : enumeration.cpp

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

Qu'observez-vous ?

Exercice-*2* **:**

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

fichier : tableau_ivalue.cpp

```
#include <iostream>
```

```
using namespace std;

int main(int argc, char** argv)
{
  int tab[3]={1,2,3};
  *(tab + 1) = -1;
  for(int i=0; i<3; i++){tab++;}

return 0;
}
```

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

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

fichier : tableau_argument.cpp

```
#include <iostream>

using namespace std;

//FONCTION D'AFFICHAGE
void afficheTableau(int t[],int size)
{
  for(int i=0; i< size; i++) cout<<t[i]<<",\t";
  cout<<endl;
}

//FONCTION D'INCRÉMENTATION 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 ?*

Q-2-3 : *On considère le programme suivant, dire s'il compile. Justifier la réponse.*

fichier : tableau_const.cpp

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

Thème - *4 Fichiers*

Exercice-*1* **:** *Écrire un programme qui clone le fonctionnement de la commande Unix* **`cat`**. *C'est à dire qui affiche à l'écran le contenu d'un fichier texte. Le nom du fichier texte à afficher est à passer en argument de la ligne de commande.*

Pour cela utiliser la classe **`ifstream`** *et l'objet* **`cout`** *de la bibliothèque standard* `C++` *. Attention, par défaut les espaces sont ignorés. Pour cela utiliser la fonction membre* **`unsetf`** *de la manière suivante :*

```
std::ifstream fin(''nom-du-fichier'');
fin.unsetf(std::ios::skipws);
```

qui obligera **`fin`** *à lire les espaces.*

Exercice-*2* **:** *Écrire maintenant un programme qui écrira dans un fichier des points du graphe de*

$$x \mapsto \sin(x)$$

pour x compris entre $-\pi$ et π. Formater le fichier sous la forme

```
abscisse0 ordonnées0
abscisse1 ordonnées1
abscisse2 ordonnées2
...
```

Visualiser ensuite le résultat à l'aide de **`gnuplot`** *:*
– *entrer la commande* **`gnuplot`**
– *dans* **`gnuplot`** *taper*

```
plot 'nom-du-fichier' w l
```

Exercice-*3* **:**

On considère le fichier généré à l'exercice précédent. Dont le contenu est sous forme d'une matrice à N lignes et M colonnes. Les valeurs N et M étant inconnues, on voudrait le renseigner dans ce fichier (afin de l'ouvrir par exemple sous `Matlab`*). On convient alors de mettre à l'entête du fichier l'information*

```
# ce fichier contient une matrice de taille N, M
```

Pour cela sous l'éditeur de texte, on ajoute une ligne vide dans l'entête du fichier puis on le sauvegarde.

Écrire un programme `C++` *, qui ouvre ce fichier sauvegardé, le lit, et le modifie comme souhaité.*

(On se renseignera sur la fonction **`istream`**::**`seekg`***).*

Thème - *5 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 :

```
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

```
1
2   ## les objets
3
4   OBJETS    =
5
6   ## Les variables
7   CXX       = g++
8   CXXFLAGS  = -g
9   CXXINC    = -I.
10  CXXLIBS   = -lm
11  ## sources
12
13  eds :%:%.o $(OBJETS)
14          $(CXX) $(OBJETS) $(CXXLIBS) -o $@ $<
15
16  ## les suffixes
17
18  .SUFIXES: .cpp
19  .cpp.o:
20          $(CXX) $(CXXFLAGS) $(CXXINC) -c $<
21
22
```

```
23 .PHONY: clean
24 clean:
25         \rm -rf *.o *~  eds
26
27 ## dependencces
28 eds.o: eds.cpp
29
```

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

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

et ajouter à la 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 crées jusqu'à lors et pour les suivants. Cela signifie notamment que chaque exercice doit avoir son propre répertoire.*

Pour aller un peu plus loin, on pourra télécharger le fichier

http ://www.ann.jussieu.fr/~delpino/public/makefile

*qui contient un **Makefile** un peu plus complexe permettant la gestion automatique des dépendances.*

Thème - *6 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*

```
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*

```
  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.*