
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 [**/doc/apoung/M2.IM**](#)

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 possible 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 *Entrée-sortie, sélections, boucles*

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

Exercice-6 : Soit la fonction mathématique f définie par $f(x) = \frac{(2x^2+3)(x^2-1)}{3x^2+1}$.

Q-6-1 : Écrire une fonction **C++** qui retourne la valeur de $f(x)$ pour un point x passé en paramètre.

Q-6-2 : 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-6-3 : 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-6-4 : 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-6-5 : 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.

Exercice-7 : Reprendre si possible (sinon reporter cet exercice pour la fiche TP2) l'exercice précédent en créant une classe **CFunction** pour que les appels se fassent comme suit

```
int main(int argc, char** argv){
    CFunction Cf;           //< instantiation d'un objet
    Cf.setStep(1e-4);       //< valeur du pas @a h
    Cf.setFunc(std::cos);    //< chargement de la fonction @a f
    std::cout<<" f      (0) = "<<Cf.f(0.)<<"\n"
              <<" f\'    (0) = "<<Cf.dxf(0.)<<"\n"
              <<" f\'\'   (0) = "<<Cf.dxxf(0.)<<"\n";

    return 0;
}
```

Exercice-8 : La suite des nombres de Fibonacci est définie de manière récursive par

$$u_0 = 0, \quad u_1 = 1, \quad u_{n+1} = u_n + u_{n-1} \quad \text{pour } n = 1, 2, 3, \dots$$

Par exemple, $u_2 = u_1 + u_0 = 1 + 0 = 1$ et $u_3 = u_2 + u_1 = 1 + 1 = 2$. Les nombres de Fibonacci ont de nombreuses applications et de propriétés. L'une d'elles est que la suite des quotients

$$q_n = u_n / u_{n-1}, \quad n = 2, 3, 4, \dots$$

converge vers le nombre d'or, à savoir $(2 + \sqrt{5})/2 \approx 1.618$.

En codant les nombres de fibonacci et leur quotient respectivement en *long int* et *long double*, écrire un programme qui génère l'affichage suivant.

(Utiliser les manipulateurs **width**, **setw**, **fill**, **precision** accessibles via **#include <iomanip>**)

n	Nombre de Fibonacci	Quotient de Fibonacci
10	55	1.6176470588235294118
20	6765	1.6180339631667065295
30	832040	1.6180339887482036213
40	102334155	1.6180339887498947364
50	12586269025	1.6180339887498948482
60	1548008755920	1.6180339887498948482
70	190392490709135	1.6180339887498948482
80	23416728348467685	1.6180339887498948482
90	2880067194370816120	1.6180339887498948482

Thème - 3 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**).

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  #EXEMPLE DE SIMPLE MAKEFILE
2
3  # les objets
4
5  OBJETS      =
6
7  # Les variables
8  CXX          = g++
9  CXXFLAGS     = -g
10 CXXINC       = -I.
11 CXXLIBS      = -lm
12
```

```

13 # sources
14
15 eds :%:%.o $(OBJETS)
16      $(CXX) $(OBJETS) $(CXXLIBS) -o $@ $<
17
18 # les suffixes
19
20 .SUFFIXES: .cpp .o
21 .cpp.o:
22      $(CXX) $(CXXFLAGS) $(CXXINC) -c $<
23
24
25 .PHONY: clean
26 clean:
27      \rm -rf *.o *~ *.backup
28
29 # dependences
30 eds.o: eds.cpp

```

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 créés jusqu'à lors et pour les suivants. 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 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.