Table des matières

1 Structure du programme, types et compilation

1.1 Fonction principale et fonctions auxiliaires

Commençons par quelques exemples, dans lesquels on voit apparaître la structure de base d'un programme C++ : un préambule, des fonctions auxiliaires, et une (**unique**) fonction principale (fonction nommée `main`). Chacun de ces éléments est optionnel, mais sans une fonction `main`, on ne crée pas d'exécutable. Attention, les procédures n'existent pas en C++ (mais il est possible de contourner cet interdit en écrivant des fonctions de type de retour `void`).

Un exemple de programme :

```
#include <iostream> //bibliotheque de gestion des flux (cout, cin,...)

int main() {
                std::cout<<"Bonjour"<<std::endl;
                return 0;
}
```

(nom du fichier : `Chapitre1/prog01.cpp`*)*

ou, la même chose en plus simple :

```
#include <iostream>
using namespace std; //evite d'avoir a ecrire std:: devant les flux

int main() {cout<<"Bonjour"<<endl;}
```

(nom du fichier : `Chapitre1/prog02.cpp`*)*

Un exemple de programme avec une fonction auxiliaire :

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

double ajoute(double x, double y){return x+y;}

int main() {double x=5, y=6;
                        cout<<ajoute(x,y)<<endl;}
```

(nom du fichier : `Chapitre1/prog03.cpp`*)*

et un exemple d'utilisation des flux de lecture :

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

int main() {int naissance_utilisateur;

        string nom_utilisateur;

        cout<<"Bonjour, quel est ton nom ?"<<endl;

        cin>>nom_utilisateur;

        cout<<"En quelle annee es-tu ne ?"<<endl;

        cin>>naissance_utilisateur;

        cout<<nom_utilisateur<<", tu as eu 20 ans en l'an "
        <<naissance_utilisateur+20<<"."<<endl;}
```

(nom du fichier : `Chapitre1/prog04.cpp`*)*

Exercice 1 (Fonction de lecture)**.** *Donner la sortie du programme suivant.*

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

string lecture() {
        string s;
        cin>>s;
        return s;}

int main() { cout<<"Entrez une chaine"<<endl;
        string s=lecture();
        cout<<"Vous avez entre : "<<s<<endl<<"Tres interessant."<<endl;}
```

(nom du fichier : `Chapitre1/prog05.cpp`*)*

1.2 Variables et types

1.2.1 Types

C++ est un langage dans lequel chaque variable introduite est définie avec son type. Plus précisément, son type est défini avant sa valeur. Les types de base, pour nous, sont `int` (pour les entiers), `double` (pour les réels), `bool` (pour les booléens) et `string` (pour les chaînes de caractères, à utiliser avec la librairie `string`).

Remarque. Le type permet au programme d'allouer la bonne taille de mémoire à chaque variable ; par exemple, pour un pointeur de type `double * p`, la commande `p++` permet de se décaler dans la mémoire du bon nombre de bits pour trouver la variable suivante dans un tableau dynamique.

Il existe bien entendu certaines conversions automatiques de certains types vers d'autres (par exemple des entiers vers les réels) mais, *au moindre doute*, il convient toujours de vérifier qu'une telle conversion existe car le compilateur *ne* donne *jamais* de message d'erreur en cas d'ambiguïté.

Exercice 2 (La conversion double –> int est la partie entière)**.** *Donner la sortie du programme suivant.*

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

int main() {
        double x=2.5;   int n=x;
        cout<<n<<endl;
        x=-4.5; n=x;
        cout<<n<<endl;
        }
```

(nom du fichier : `Chapitre1/prog06.cpp`*)*

Exercice 3. *Écrire, sur le modèle du deuxième programme ci-dessus, une fonction de prototype*

`double distribuer(double x, double y, int z)`

dont le résultat est `(x+y)*z`*. La tester dans un programme.*

Exercice 4. *Que va afficher le programme suivant ?*

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

int main() {cout<<1/3<<endl;
        cout<<1/(double (3))<<endl;}
```

(nom du fichier : `Chapitre1/prog07.cpp`*)*

1.2.2 Variables constantes et statiques

À la déclaration d'une variable, on peut la déclarer constante (avec le mot clé `const`), ce qui empêchera de la modifier dans la suite. On remarque ainsi que la valeur d'une variable étiquetée `const` doit être initialisée dès sa déclaration.

Exercice 5. *Que va afficher le programme suivant ? Pourrait-on y décommenter l'avant dernière ligne ?*

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

const int R=2;

int main() {cout<<R<<endl;
        //R=R+1;
        }
```

(nom du fichier : `Chapitre1/prog08.cpp`*)*

On peut également la déclarer statique (avec `static`), ce qui permet que, même une fois sorti du bloc où elle a été définie, la variable soit gardée en mémoire (et accessible lorsque l'on retourne dans le bloc en question). Il faut faire attention à ne faire trop usages de ces variables afin de ne pas encombrer la mémoire. Une idée d'utilisation parmi d'autres est de compter le nombre d'appels à une fonction donnée.

Exercice 6. *Que va afficher le programme suivant ? Pourrait-on y décommenter l'avant dernière ligne ?*

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

void f() {static int s;
        s=s+1;
        cout<<s<<endl;}

int main() {f(); f(); f();
        //cout<<s<<endl;
        }
```

(nom du fichier : `Chapitre1/prog09.cpp`*)*

1.3 Compilation

1.3.1 Compilation d'un fichier simple

La compilation et la création d'un fichier exécutable, à partir d'un programme autonome (comme celui, par exemple, de l'exercice 1), nommé `mon_programme.cpp`, se fait avec la commande, dans un terminal,

```
g++ mon_programme.cpp -o mon_executable.exe
```

et on exécute ensuite l'exécutable `mon_executable.exe` avec la commande

```
./mon_executable.exe
```

Attention : selon le système d'exploitation et selon le choix de l'environnement de travail, il existe de nombreux compilateurs. Le principe général est le même et les erreurs détectées sont les mêmes : seules la syntaxe des options et la formulation des messages d'erreurs varient. Il est possible la machine `134.157.117.41` en TP d'utiliser [1] également le compilateur `clang`.

1.3.2 Compilation d'un fichier multiple

(Ce paragraphe n'est pas indispensable pour débuter la programmation en C++). Lorsqu'un programme devient volumineux, il est peu rentable de le placer dans un seul fichier : la compilation devient très longue, puisqu'il faut tout recompiler chaque fois ; il est difficile de s'y retrouver lorsque l'on veut le modifier ultérieurement ; et surtout il est impossible d'en réutiliser des morceaux pour une autre application ou pour un échange avec un collègue.

Pour simplifier le travail, on répartit alors les différentes routines dans plusieurs fichiers source (`*.cpp`). Chacun de ces fichiers peut alors être compilé indépendamment, produisant un fichier objet (`*.o`). L'ensemble des fichiers objets est ensuite regroupé par l'éditeur de liens pour former un programme exécutable unique (*.exe, sous Windows).

Comme les fichiers (ou les programmeurs) doivent communiquer entre eux sans avoir à connaître le détail du code d'autrui, il faut pour chaque fichier `*.cpp` au moins un fichier en-tête (dont le nom terminera le plus souvent par `.h` ou `.hpp`) qui contient toutes les déclarations (de classes, de fonctions, de constantes, de variables, etc.) susceptibles d'être utilisées par les autres, ainsi, le plus souvent, qu'un descriptif sommaire des fonctions en commentaires. On n'oubliera pas les fonctions déclarées `inline`, car le compilateur doit les connaître entièrement pour les placer directement dans le code produit.

Dans chaque fichier source `*.cpp` proprement dit, on trouvera généralement une directive d'inclusion du fichier en-tête correspondant. Cette directive s'écrit `#include "mon_entete.hpp"` si le nom du fichier en-tête est `mon_entete.hpp`. Elle est suivie éventuellement d'autres directives d'inclusion, soit pour les librairies standard (`#include <iostream>`, `#include <string>`,...), soit pour les autres fichiers en-têtes du même programme utilisés par le fichier courant. On trouvera ensuite l'implantation des fonctions qui ne sont pas `inline`.

Dans le fichier contenant la fonction principale `main()`, on trouvera toutes les inclusions d'en-têtes nécessaires, suivies par `main()`.

Par exemple, le programme suivant

1. Il suffit de se connecter sur la machine par la commande `ssh 134.157.117.41`, de se placer dans le bon répertoire puis de compiler en utilisant `clang++` à la place de `g++`.

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

const double R=6371e3;

#define pi 3.14

double aire_sphere(double x){return 4*pi*pow(x,2);}

int main() {cout<<"L'aire de la surface de la terre\
 est approximativement "<<
 aire_sphere(R)<<" m^2."<<endl;}
```

(nom du fichier : `Chapitre1/surface_terre.cpp`*)*

appelé `surface_terre.cpp`, compilé par la commande

```
g++ surface_terre.cpp -o surface_terre.exe
```

peut être remplacé par les programmes suivants.

Tout d'abord, le fichier d'en-tête `surface_terre2.h` :

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

const double R=6371e3;

#define pi 3.14

double aire_sphere(double);
```

(nom du fichier : `Chapitre1/surface_terre2.h`*)*

puis le fichier `surface_terre2_aux.cpp` :

```
#include "surface_terre2.h"

double aire_sphere(double x){return 4*pi*pow(x,2);}
```

(nom du fichier : `Chapitre1/surface_terre2_aux.cpp`*)*

et `surface_terre2_main.cpp` :

```
#include "surface_terre2.h"

int main() {cout<<"L'aire de la surface de la terre\
 est approximativement "<<
 aire_sphere(R)<<" m^2."<<endl;}
```

(nom du fichier : `Chapitre1/surface_terre2_main.cpp`*)*

On compile alors avec les commandes suivantes :

```
g++ -c surface_terre2_aux.cpp
```

(qui crée le fichier objet `surface_terre2_aux.o`) puis

```
g++ surface_terre2_aux.o surface_terre2_main.cpp -o surface_terre2.exe
```

Exercice 7. *Sachant que la commande* `#define` *fonctionne par simple copier-coller, que se passe-t-il dans les programmes précédents si l'on remplace la ligne*

```
#define pi 3.14
```

par

```
#define pi 3+0.14
```

Exercice 8 (Utilisation de define et de undef)**.** *Donner la sortie du programme suivant. Peut-on y décommenter la ligne 5 ?*

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

#define chaine "Bonjour"
//#undef chaine
#undef pi

int main() {cout<<chaine<<endl;}
```

(nom du fichier : `Chapitre1/prog10.cpp`*)*

1.4 Les messages d'erreurs

Il est rarissime qu'un code que l'on écrit ne comporte aucune erreur car il n'est pas possible de se concentrer sur tous les aspects à la fois. Il y a plusieurs types d'erreurs : les erreurs de syntaxe que détecte le compilateur, les erreurs de programmation dues à des ambiguïtés de syntaxe (conversion de type implicite mal devinée) et les erreurs de raisonnement dont on ne se rend compte qu'à l'exécution du programme sur des données test.

1.4.1 Les erreurs de syntaxe

Les erreurs les plus fréquentes sont les oublis de parenthèse ou d'accolade, de point-virgule et de déclaration de variables. Tout compilateur digne de ce nom renvoie en général une liste d'erreurs constatées avec un numéro de ligne. **Attention :** seule la première erreur est une erreur de manière certaine. En effet, si une accolade manque, le découpage en bloc d'instructions ne se fait plus correctement et le compilateur peut alors inventer des erreurs supplémentaires. Il conviendra donc de corriger les erreurs une à une en commençant systématiquement par la première et en recompilant.

Des compilateurs différents détectent évidemment les mêmes erreurs mais peuvent renvoyer des messages formulés différemment. Dans une entreprise, le choix du compilateur est le plus souvent dicté par les choix antérieurs de l'équipe : si ce n'est pas votre compilateur favori, commencez par réaliser différents petits programmes en créant vous-même des erreurs *ad hoc* pour apprendre les réponses du compilateur. Par exemple, dans la présente feuille, il est utile d'introduire des erreurs volontaires pour faire connaissance avec `g++`.

1.4.2 Les erreurs de raisonnement ou mauvaises interprétations d'ambiguïtés

Les erreurs de raisonnement donnent lieu à un programme syntaxiquement correct mais qui ne renvoie pas la réponse souhaitée ou bien se met à boucler à l'infini. Afin d'éviter ce type d'erreur souvent difficile à corriger, il convient de bien réfléchir à l'algorithme avant de se mettre à écrire le code (l'esprit est alors occupé à éviter les erreurs de syntaxe). En particulier, il est utile

- d'avoir une idée claire de l'algorithme à utiliser,
- d'avoir listé au préalable tous les cas et sous-cas à envisager,
- de penser son code sous une forme modulaire ("une tâche, une fonction") afin de pouvoir tester chaque morceau séparément.

Si une erreur est néanmoins constatée en utilisant le programme sur un exemple simple que l'on sait traiter à la main, on commencera par tester toutes les fonctions une par une et à afficher par `cout <<... << std::endl;` toutes les variables pour lesquelles on émet des doutes. Il est utile également d'utiliser un `return 0;` dans la fonction `main()` en changeant d'endroit pour localiser la ligne problématique.

2 Boucles, opérations `x+=`, `k++`, `++k`, ...

2.1 Opérations `x+=`, `k++`, `++k`, ...

Si `x, y` sont des entiers ou des réels, `x+=y` est équivalent à l'affectation `x=x+y`. De même, on peut écrire `x*=y`, équivalent de `x=x*y`, ... Si `k` est un entier ou un réel, les opérations `k++`, `++k` l'incrémentent de un. La différence entre ces deux opérations apparaît dans l'exercice suivant.

À strictement parler, aucune des deux opérations `x++` et `++x` n'est nécessaire car il suffit de faire l'incrémentation à la main `x = x+1;` au bon endroit : ce sont surtout de simples raccourcis pour qui les maîtrise et de grosses sources d'erreurs pour qui ne les maîtrisent pas. Il faut absolument les connaître pour être capable de lire le code d'autrui mais les utiliser est loin d'être inévitable !

Exercice 9 (De la différence entre x++ et ++x, visibilité des variables)**.** *Donner la sortie du programme suivant. Pourrait-on décommenter l'avant dernière ligne ?*

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

void f() {static int s;
        cout<<s++<<endl;
        cout<<++s<<endl;}

int main() {f(); f(); f();
        //cout<<s<<endl;
        }
```

(nom du fichier : `Chapitre1/prog11.cpp`*)*

Exercice 10 (De la différence entre x++ et ++x, visibilité des variables)**.** *Donner la sortie du programme suivant. Pourrait-on décommenter l'avant dernière ligne ? Pourrait-on remplacer* `int s;` *par* `const int s;` *?*

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

int s;

void f() {cout<<s++<<endl;
        cout<<++s<<endl;}

int main() {f(); f(); f();
        //cout<<s<<endl;
        }
```

(nom du fichier : `Chapitre1/prog12.cpp`*)*

2.2 Instructions, boucles et disjonctions

Les instructions sont suivies par un point virgule et regroupées par des accolades si nécessaire. Notons que les affectations et modifications intervenant entre des accolades, hors du cadre des variables `static` (qui sont gardées en mémoire mais accessibles uniquement en cas de retour dans le bloc où elles ont été définies), sont indépendantes du reste du programme.

Exercice 11 (Notion de bloc d'instructions)**.** *Que rendent les programmes suivants ? (NB :* `'\n'` *est synonyme de* `endl`*).*

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

int main(){int i=0;
        {int i=5; cout<<++i<<'\n';}
        cout<<i<<'\n';}
```

(nom du fichier : `Chapitre1/prog13.cpp`*)*

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

int main(){{int i=5; cout<<++i<<'\n';}
        cout<<i<<'\n';}
```

(nom du fichier : `Chapitre1/prog14.cpp`*)*

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

int main(){int i=0;
        {const int i=5; cout<<++i<<'\n';}
        cout<<i<<'\n';}
```

(nom du fichier : `Chapitre1/prog15.cpp`*)*

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

int main(){int i=0;
        {cout<<++i<<'\n';}
        cout<<i<<'\n';}
```

(nom du fichier : `Chapitre1/prog16.cpp`*)*

Exercice 12 (Instruction if)**.** *Que rend le programme suivant ? Que se passe-t-il si l'on supprime les accolades autour de* `b=8; c=10;` *?*

```
#include <iostream>

int main(){int a=0, b=1, c=2;
        if (a>b) b=-4;
                else {b=8;c=10;}
        std::cout<<b<<" "<<c<<std::endl;}
```

(nom du fichier : `Chapitre1/prog17.cpp`*)*

Exercice 13 (Boucle for)**.** *Que rend le programme suivant ? Que se passe-t-il si l'on remplace* `++x` *par* `x++` *à la dernière ligne ?*

```
#include <iostream>

int main(){int x=10;
for(int i=0;i<10;i++) {x++;}
        std::cout<<++x<<std::endl;}
```

(nom du fichier : `Chapitre1/prog18.cpp`*)*

Exercice 14 (Boucle while)**.** *Que rend la fonction* `f` *du programme suivant ?*

```
#include <iostream>

int f(int n){
        if (n<0) return 0;
        int r=1;
        while (n>1) r*=n--;
        return r;}

int main(){int a=4;
        std::cout<<f(a)<<std::endl;}
```

(nom du fichier : `Chapitre1/prog19.cpp`*)*

Exercice 15 (Disjonction switch)**.** *Que rend le programme suivant ? Que se passe-t-il si l'on remplace* `f(b--)` *par* `f(--b)` *à la dernière ligne ?*

```
#include <iostream>

int f(int x){
        switch (x)
        {       case 1 : return 1;
                case 2 : return 4;
                default : return 0;}}

int main(){int a=1, b=3;
        std::cout<<f(a)<<" "<<f(b--)<<" "<<b<<std::endl;}
```

(nom du fichier : `Chapitre1/prog20.cpp`*)*

Exercice 16 (Affectation)**.** *Que rend le programme suivant, basé sur l'affectation par la syntaxe par*

`(booleen)? valeur~: autre_valeur`

Peut-on remplacer `max(1,3)` *par* `max(1,3)++` *à la dernière ligne ?*

```
#include <iostream>

int max(int a, int b){return (b<a)?a:b;}

int main(){std::cout<<max(1,3)<<std::endl;}
```

(nom du fichier : `Chapitre1/prog21.cpp`*)*

Exercice 17. *(Conjecture de Syracuse/Collatz) On définit, pour a entier strictement positif, la suite (u_n) par $u_0 = a$ et*

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

C'est une conjecture (qui n'a été ni démontrée, ni infirmée par un contre-exemple) que le nombre

$$N_a = \inf\{n\,;\, u_n = 1\}$$

(dépendant bien entendu de a) est alors fini, quel que soit a.

1) Écrire un programme qui demande à l'utilisateur de taper a et affiche toutes les valeurs de u_n pour $n \leq N_a$, ainsi que N_a.

2) Écrire un programme qui demande à l'utilisateur de taper A et affiche

$$\max\{N_1, \ldots, N_A\}.$$

Exercice 18. *Écrire un programme qui demande à l'utilisateur de saisir un entier N et qui affiche le $N^{ème}$ nombre premier.*

Rappel : un entier $p \geq 2$ est premier s'il n'est divisible par aucun entier compris entre 2 et $\sqrt{p}$.

Exercice 19. *Écrire un programme qui demande à l'utilisateur de saisir deux entiers n, m et qui affiche leur pgcd.*

Rappel : L'algorithme d'Euclide permet de calculer le pgcd $n \wedge m$ de deux nombres n, m suivant les règles suivantes :

$$\begin{cases} n \wedge m = m & \text{si } n = 0, \\ n \wedge m = m \wedge n & \text{pour tous } n, m, \\ n \wedge m = n \wedge r & \text{où } r \text{ est le reste de la division euclidienne de } m \text{ par } n, \text{ pour tous } n \leq m. \end{cases}$$

3 Variables aléatoires en C++

Le programme suivant explique et illustre la génération de variables aléatoires en C++ (sans appel à une bibliothèque dédiée). L'idée est que `random()` rend un entier aléatoire de loi uniforme sur l'intervalle $\{0, \ldots,$`RAND_MAX`$\}$, où `RAND_MAX` est un nombre fixe, de l'ordre de 10^{10}. On divise donc `random()` par `RAND_MAX` pour obtenir des v.a. sur $[0, 1]$. Bien entendu, plusieurs appels successifs à `random()` donnent des v.a. indépendantes. Afin d'éviter les biais, on a recours à une modification de la graine de l'aléa par `srandom(time(NULL))` : l'usage de l'heure de début du programme est l'un des rares moyens d'obtenir un paramètre qui varie à chaque exécution !

```
#include <iostream>
#include <ctime> //gestion du temps, utile pour reinitialiser
        //la graine de l'alea avant la premiere utilisation
#include <cstdlib>

using namespace std;

inline double unif_rand() // "inline" s'utilise, optionnellement,
        //       pour les fonctions courtes
    {return 2*(random()+0.5)/(RAND_MAX+1.0)-1;}

int main() {
        srandom(time(NULL));//on initialise la graine de rand
    for ( int n = 0 ; n < 10 ; ++n ) cout << unif_rand() << endl;}
```

(nom du fichier : `Chapitre1/unif_rand.cpp`*)*

Exercice 20. *Écrire un programme affichant sur la console une variable aléatoire de loi de Cauchy.*

Rappel : Pour U v.a. de loi uniforme sur $[-\pi/2, \pi/2]$, $\tan(U)$ est une variable aléatoire de loi de Cauchy (c'est la méthode de simulation par *inversion de la fonction de répartition*).

Exercice 21. *Écrire un programme demandant à l'utilisateur le choix de deux paramètres réels a, b, renvoie un message d'erreur si $a \geq b$ et affiche sur la console une variable aléatoire de loi uniforme sur $[a, b]$ sinon.*

Exercice 22. *Écrire un programme demandant à l'utilisateur le choix d'un paramètre réel λ et d'un entier n et affichant sur la console, si $\lambda > 0$ et $n \geq 1$, n simulations de variables aléatoires de loi exponentielle de paramètre λ (dans le cas contraire, on affichera un message d'erreur approprié).*

Rappel : Pour U v.a. de loi uniforme sur $[0, 1]$, $-\log(U)/\lambda$ est une variable aléatoire de loi exponentielle de paramètre λ (c'est la méthode de simulation par *inversion de la fonction de répartition*).

Exercice 23. *Écrire un programme demandant à l'utilisateur le choix d'un entier n et affichant sur la console n simulations de variables aléatoires de loi de Wigner, la loi de Wigner étant la loi de support $[-2, 2]$ et de densité*

$$\frac{1}{2\pi}\sqrt{4 - x^2}.$$

Rappel : Pour simuler une v.a. de densité f de support $[a, b]$ et telle que $0 \leq f \leq M$, on utilise une suite $(U_n, V_n)_{n\geq 1}$ de couples indépendants de variables aléatoires telles que pour tout n, U_n, V_n sont indépendantes et de lois uniformes sur respectivement $[a, b]$, $[0, M]$, on pose

$$\tau = \min\{n \geq 1\,;\, V_n \leq f(U_n)\},$$

et la variable aléatoire U_τ suit alors la loi de densité f. C'est la méthode de *simulation par rejet*. Notons que plus M est petit, plus cette méthode est rapide, on a donc intérêt à choisir $M = \|f\|_\infty$.

Exercice 24. *Une variable de Poisson de paramètre λ peut être obtenue de la manière suivante : soit $(U_k)_{k\in\mathbb{N}}$ une famille de v.a. i.i.d. uniformes sur $]0, 1[$. La v.a.*

$$T = \inf\{n \in \mathbb{N}; U_0 \dots U_n \leq e^{-\lambda}\}$$

suit une loi de Poisson. Écrire une fonction qui génère une v.a. de Poisson en utilisant cette propriété.

4 Écriture sur un fichier extérieur

Le processus d'écriture sur un fichier extérieur est présenté dans le programme suivant.

```
#include <iostream>
#include <fstream>//permet de gerer les lectures/ecritures sur
        //  les fichiers exterieurs

using namespace std;

int main() {
        ofstream flux("nom_fichier.dat"); //on cree un flux de sortie
                //  nomme "flux", qui
                //  ecrit, non pas sur la console comme "cout", mais sur
                //  le fichier nom_fichier.dat,
                //  qui est cree, dans le repertoire ambiant,
                //  s'il n'existe pas, et ecrase puis
                //  recree s'il existe deja.
        flux<<34<<endl;//on ecrit 34 sur la premiere ligne
                //  du fichier nom_fichier.dat
        flux<<56<<endl;//on ecrit 56 sur la deuxieme ligne
                //  du fichier nom_fichier.dat
        flux<<77<<endl;//on ecrit 77 sur la troisieme ligne
                //  du fichier nom_fichier.dat
        flux.close();}
```

(nom du fichier : `Chapitre1/flux_sortie.cpp`*)*

Exercice 25. *Recopier le programme ci dessus, le compiler et l'exécuter. Ouvrir ensuite le fichier* `nom_fichier.dat` *avec un éditeur de texte. Modifier le programme de façon à faire figurer aussi la ligne*

```
1 2 3 4 5 6 7 8 9 10
```

à la fin du fichier `mon_fichier.dat`.

Exercice 26. *Écrire un programme qui demande à l'utilisateur le choix d'un entier n et qui crée un fichier nommé* `echantillon_wigner.dat` *sur lequel il écrit, en colonne, n simulations de variables aléatoires de loi de Wigner. Ouvrir ensuite Scilab, et afficher l'histogramme de cet échantillon (pour $n = 1500$), que l'on compare à la densité, avec les commandes suivantes (à écrire dans la fenêtre Scilab).*

```
(on se rend sur le repertoire ambiant avec les commandes Linux)

clf; (permet d'effacer la fenetre graphique si elle contient deja une image)

echant=read("echantillon_wigner.dat",-1,1);

abscisses=-2:0.01:2;

ordonnees=(2*%pi)^(-1)*sqrt(4-abscisses.*abscisses);

histplot(50, echant);

plot2d(abscisses,ordonn'ees,5);

xselect(); (met la fenetre graphique au premier plan)
```

Exercice 27. *Écrire un programme demandant à l'utilisateur le choix de l'affichage sur la console ou de l'écriture sur un fichier et d'un entier n et qui réalise les instructions suivantes :*

- si le choix est l'écriture sur la console, on simule et affiche sur la console $\min(n, 20)$ *réalisations indépendantes de la loi Gaussienne standard,*

- si le choix est l'écriture sur un fichier, on simule et écrit sur un fichier `echant_gauss.dat`*, en colonnes, n réalisations indépendantes de la loi Gaussienne standard.*

On utilisera la méthode de simulation de Box-Müller, décrite ci-dessous, en essayant, si possible, d'utiliser des variables `static` *pour n'avoir à simuler les variables aléatoires U et θ que $n/2$ fois. On comparera avec la densité Gaussienne (qui vaut $\frac{1}{\sqrt{2\pi}}\exp(-\frac{x^2}{2})$ en tout x de $\mathbb{R}$) standard via Scilab.*

Rappel : Pour U, θ variables aléatoires de loi uniforme sur respectivement $[0, 1]$, $[0, 2\pi]$, les variables parties réelles et imaginaires du nombre complexe $\sqrt{-2\log(U)}e^{i\theta}$ sont deux variables aléatoires indépendantes de loi Gaussienne standard.

Exercice 28 (Ajustement de paramètre)**.** *Dans tout cet exercice, α est un réel strictement positif fixé arbitrairement. On placera donc en préambule de son programme (ou dans le fichier d'en tête) la ligne*

`const double alpha =` *mettre ici la valeur que l'on choisit*

(on essaiera, à la fin de l'exercice, de l'ajuster selon un critère qui sera présenté plus loin).

a) Écrire un programme qui demande à l'utilisateur le choix d'une valeur de l'entier n, simule, via la méthode de Box-Müller, des v.a.i.i.d.

$$X_1, \dots, X_n$$

de loi gaussienne standard et affiche, en colonne sur la console, les valeurs de

$$S_k := \frac{X_1^2 + \cdots + X_k^2 - k}{k^\alpha},$$

pour des valeurs de k variant de 1 à n.

b) (Facultatif) Essayer d'ajuster la valeur de $\alpha > 0$ de façon à ce que, pour une grande valeur de n, la valeur de S_n aie tendance à se "stabiliser", i.e. à ne devenir ni trop proche de 0, ni trop grande. NB : Penser au TCL !

c) Vérifier, empiriquement, que lorsque $\alpha = 1$, S_n tend vers une limite que l'on déterminera.

d) Vérifier, en exportant des données et en réalisant un histogramme sous Scilab, que lorsque $\alpha = 0.5$, $\frac{S_n}{\sqrt{2}}$ converge en loi vers une loi gaussienne standard.

5 Optimisation d'un programme

Même si les machines actuelles sont rapides et ont une mémoire suffisante pour la plupart des tâches, de nombreuses situations requièrent d'avoir un code qui s'exécute aussi rapidement que possible (interactions avec un signal en temps réel, présence de données gigantesques, dispositifs embarqués dans une machine aux capacités réduites, etc). Il convient donc de savoir écrire un code *efficace*. Attention néanmoins, il existe différentes optimisations possibles :

— optimisation de l'espace mémoire (rare) : on stocke le moins d'information possible, quitte à recalculer plusieurs fois les mêmes quantités ;
— optimisation du temps d'exécution : on essaie de ne jamais recalculer trop de fois la même quantité et on préfère la stocker en mémoire pour usage futur ;
— optimisation de l'adaptabilité du programme : lorsque l'on sait que le programme est voué à évoluer et à être généraliser, on essaie de prévoir ces évolutions et on écrit un code le plus générique possible afin de n'avoir plus qu'à le compléter et non à le refondre.

Le premier cas reste rare et nous nous focaliserons ici sur le deuxième et le troisième cas.

Une bonne mesure de l'optimisation temporelle se fait, sous Linux, par l'usage de la commande `time` selon l'exemple générique suivant :

```
g++ programme.cpp -o executable.exe

time ./executable.exe
```

et affiche, après le nom du fichier, le temps total d'exécutions (et d'autres informations moins utiles directement). Vous êtes encouragés, lorsque vous hésitez entre plusieurs méthodes, à coder les différentes versions et à mesurer le temps d'exécution de chacune.

Vous pouvez également aller sur le site `http://projecteuler.net/problems` qui proposent une multitude de petits problèmes mathématiques à résoudre par la programmation : les problèmes sont choisis de telle manière qu'il est nécessaire de coder de manière efficace pour avoir la réponse en temps raisonnable sur un PC standard. User et abuser de ce site !

5.1 Règles à respecter

Il est rare de produire un code optimisé dès sa première écriture : néanmoins, si l'on se rend compte que l'on calcule plusieurs fois le même objet, il est utile de le calculer une fois pour toutes et de le stocker. Si on cherche seulement l'existence d'un certain élément dans un tableau, nul besoin de le parcourir en entier si l'élément est présent plusieurs fois : il suffit de s'arrêter à la première occurrence ! Des problèmes proches peuvent avoir des solutions optimales... très éloignées (cf. l'exemple des nombres premiers ci-dessous).

5.2 Optimisation par le compilateur

Le compilateur est souvent capable de multiples optimisations. En particulier, `g++` possède, outre une pléthore d'optimisations ponctuelles, trois niveaux d'optimisations globales.

Cela s'utilise au moment de la compilation par l'ajout de l'une des trois options :

```
g++ -O1 programme.cpp -o executable.exe

g++ -O2 programme.cpp -o executable.exe

g++ -O3 programme.cpp -o executable.exe
```

par ordre croissant d'optimisation.

Attention ! Ce sont des schémas *génériques* d'optimisation *souvent utiles* mais ils n'accélèrent pas le programme à coup sûr ! Pour un même programme, cela peut donner lieu à une nette amélioration comme à un net ralentissement. De plus, face à un programme mal codé au départ, cela n'apporte pas à grand chose.

5.3 Exercices de réflexion

Exercice 29. *Reprendre l'exercice 18 sur le test de primalité d'un nombre premier en essayant d'écrire le programme le plus rapide possible. Comparez avec vos camarades.*

Exercice 30. *Écrire un programme qui écrit dans un fichier* le plus rapidement possible *la liste des nombres premiers de* 1 *à* 1000000*. Donner également la méthode de compilation qui donne le meilleur résultat ainsi que le temps d'exécution mesuré par la commande* `time` *sous Linux : si c'est trop long (plus de dix minutes), réduire à 100000 et/ou réfléchir davantage.* Attention*, la réutilisation de l'exercice précédent n'est pas nécessairement la méthode la plus efficace...*

Exercice 31. *Reprendre l'exercice 17 et vérifier la conjecture de Syracuse pour a variant de* 1 *à 1000000 le plus rapidement possible : vous écrirez dans un fichier l'entier N_a pour chaque a de telle sorte à pouvoir faire ensuite l'histogramme par Scilab. Attention, la réutilisation de l'exercice 17 n'est pas nécessairement la méthode la plus efficace ; prenez garde à ne pas refaire trop de fois la même chose...*