1

L'incontournable Hello world

Nous le retrouvons dans presque tous les ouvrages de programmation. Il vient toujours en tête ! Ce sera aussi pour nous l'occasion de nous familiariser avec l'environnement de développement. Dans ce chapitre, il s'agira tout d'abord d'éditer les fichiers source, à savoir `hello.cpp` et `Hello.java`.

Sur le CD-Rom d'accompagnement figure un éditeur bien pratique pour Windows, que nous pouvons utiliser pour les différentes étapes de la programmation de nos exemples ou exercices : l'éditeur Crimson (voir annexe C). L'environnement de Crimson permet d'éditer les programmes source C++ et Java qui sont des fichiers de texte ASCII, de les compiler et de les exécuter. La compilation et le résultat de l'exécution des programmes peuvent s'afficher dans une fenêtre de travail intégrée à Crimson ; c'est donc l'outil idéal pour cela. Quant aux utilisateurs de Linux, ils ne devraient pas avoir de difficultés à choisir les outils adaptés aux tâches décrites ci-dessous (voir annexe F).

Dans ce chapitre, nous allons aussi introduire le fameux `make` de GNU, qui nous permettra d'automatiser la compilation. Avant de pouvoir utiliser le `make` ainsi que les compilateurs Java et C++ et de les intégrer dans l'éditeur Crimson, il faudra tout d'abord installer ces produits. Ils sont tous disponibles sur le CD-Rom, comme décrit dans l'annexe B. Un autre éditeur que Crimson peut évidemment être utilisé, mais il faudra alors compiler les programmes avec le `make` et les exécuter dans une fenêtre DOS (procédures décrites dans l'annexe B). Un outil Open Source beaucoup plus complet, NetBeans, peut aussi être utilisé (voir les annexes E et F pour son installation et son utilisation à travers des exemples complets).

Nous avons étendu l'exercice traditionnel en ajoutant la date, qui sera imprimée avec notre incontournable « `Hello world` ».

Hello world en C++

La première tâche consiste à introduire le code source dans le fichier `hello.ccp` via l'éditeur Crimson tel que décrit dans l'annexe C.

```
// hello.cpp
#include <ctime>
#include <iostream>

using namespace std;

int main()
{
 time_t maintenant;
 time(&maintenant);

 cout << "Hello world en C++: " << ctime(&maintenant) << endl;
 return 0;
}
```

Avant de passer à la compilation, nous allons examiner ce code dans les détails, en laissant de côté le `main()` et le `cout`, qui seront analysés et comparés entre les deux langages un peu plus loin. Il en va de même pour le :

```
using namespace std;
```

qui identifie un espace de noms, que nous étudierons au chapitre 6. Nous pouvons déjà signaler ici que l'utilisation de `namespace` n'est disponible que dans le Standard C++. Cet emploi est lié à la disparition en Standard C++ des `.h` dans les en-têtes. Si nous avions écrit le code suivant :

```
// hello2.cpp
#include <time.h>
#include <iostream.h>

int main()
{
 time_t maintenant;
 time(&maintenant);

 cout << "Hello world2 en C++: " << ctime(&maintenant) << endl;
 return 0;
}
```

nous aurions obtenu en fait le même résultat, excepté que cette dernière version est plus familière aux programmeurs C et C++ traditionnels ! En le compilant, nous obtiendrions ces remarques de codes dépréciés :

```
C:/MinGW/bin/../lib/gcc/mingw32/3.4.5/../../../../include/c++/3.4.5/backward/
➥iostream.h:31,
                 from hello2.cpp:4:
#warning This file includes at least one deprecated or antiquated header.
```

Le binaire (hello2.exe) serait tout de même généré et correctement exécutable.

Pour en revenir à la première version, les :

```
#include <ctime>
#include <iostream>


using namespace std;
```

vont en effet de pair, et nous les utiliserons tout au long de cet ouvrage. Nous travaillerons presque exclusivement avec les formes sans les .h, c'est-à-dire avec le Standard C++, même si, à de rares occasions, il nous arrivera d'utiliser des .h, c'est-à-dire des fonctions C qui ne sont pas disponibles dans la bibliothèque du Standard C++.

Les #include <ctime> et #include <iostream> sont des fichiers d'en-têtes nécessaires à la définition des classes et des objets utilisés. Si time_t, time et ctime n'étaient pas introduits, la ligne #include <ctime> ne serait pas obligatoire.

Il faut noter que le fichier <ctime> du Standard C++ contient en fait l'include <time.h>, qui est un héritage des fonctions C. Nous reviendrons sur la directive include au chapitre 4, section « Directive include ».

Dans <iostream>, nous avons la définition de cout et de son opérateur <<.

time_t est un type défini dans la bibliothèque ANSI C++ permettant de définir la variable maintenant, qui sera utilisée pour obtenir la date actuelle au travers de time(&maintenant). Nous reviendrons plus loin sur le &, qui doit être ajouté devant la variable car la fonction C time() s'attend à recevoir une adresse. Le résultat sera obtenu au moyen de << ctime (&maintenant), comme nous allons le voir ci-dessous. Le chaînage des opérateurs << est à noter, ce qui permet d'envoyer une série de blocs de données sans devoir les définir sur des lignes séparées.

Le fichier hello.cpp est ensuite compilé par le compilateur C++, accessible par le chemin d'accès PATH décrit dans l'annexe B. Cette compilation se fait de la manière suivante :

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

Le paramètre –o de g++ indique que le nom hello.exe correspond au fichier binaire compilé à partir des fichiers source inclus dans la ligne de commande (ici, hello.cpp). Avec Linux, nous pourrions utiliser :

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

Les fichiers hello.exe ou hello sont les exécutables. Par la suite, nous utiliserons systématiquement l'extension .exe, qui est nécessaire sous DOS. Un programme nommé hello.exe pourra aussi s'exécuter sous Linux, qui possède une propriété d'exécutable pour le fichier. Ce n'est pas le cas sous DOS, où les programmes doivent se terminer avec une extension déterminée comme .bat, .com ou .exe. Lorsque nous exécutons le programme hello.exe, on obtient :

```
Hello world en C++: Mon Mar 31 14:55:20 2008
```

Le fichier `hello.exe`, nommé parfois exécutable binaire, contient le code machine que le processeur pourra directement exécuter grâce aux fonctions du système d'exploitation (DOS ou Linux). Il s'exécutera uniquement sur les machines ayant le même système d'exploitation. Le programme `hello.exe` peut aussi ne pas fonctionner sur un PC doté d'un processeur ancien, car le compilateur n'aura sans doute pas assuré de compatibilité avec les modèles antérieurs, par exemple, au Pentium 586 d'Intel. Pour fonctionner sous Linux, avec d'autres systèmes d'exploitation ou d'autres types de configurations matérielles, il faudra recompiler le programme `hello.cpp` sur ce type de machine. Un programme C ou C++ compilable sur différents supports est communément appelé portable. Nous allons voir que cela se passe bien différemment avec Java.

Hello world en Java

Avant tout, une première remarque s'impose concernant le nom des fichiers. Comme le nom d'une classe commence toujours par une majuscule, il est nécessaire de définir le nom du fichier selon le même principe, c'est-à-dire ici `Hello.java`. Faillir à cette règle est une erreur classique, et la plupart des débutants tombent dans le piège. En effet, si on oublie ce principe, cette classe ne pourra être compilée :

```
import java.util.*;

public class Hello {
   public static void main(String[] args) {
      Date aujourdhui;
      aujourdhui = new Date();
      System.out.println("Hello world en Java: " + aujourdhui);
   }
}
```

En Java, le traitement de la date est vraiment très simple. Nous n'avons plus des variables ou des structures compliquées comme en C ou en C++, mais simplement un objet. `aujourdhui` est ainsi un objet de la classe `Date` qui est créé grâce à l'opérateur `new`. Les détails de la méthode `println` sont expliqués plus loin.

La compilation s'effectue ainsi :

```
javac Hello.java
```

et nous l'exécutons par :

```
java Hello
```

ce qui nous donne comme résultat :

```
Hello world en Java: Mon Mar 31 14:55:44 GMT+02:00 2008
```

Nous remarquons ici que nous n'avons pas de `Hello.exe`. En effet, le processus se déroule différemment en Java : après la compilation de `Hello.java`, avec `javac.exe` (qui est, comme pour C++, un exécutable binaire différent sur chaque système d'exploitation), un fichier compilé `Hello.class` est généré. Pour exécuter `Hello.class`, nous utilisons le

programme java.exe, qui est la machine virtuelle de Java. Notons qu'il faut enlever dans la commande l'extension .class. java.exe trouvera alors un point d'entrée main() dans Hello.class et pourra l'exécuter. Si la classe Hello utilise d'autres classes, elles seront chargées en mémoire part la machine virtuelle si nécessaire.

Le GMT (*Greenwitch Mean Time*) pourrait être différent suivant l'installation et la configuration du PC, la langue ou encore la région. Un CET, par exemple, pourrait être présenté : *Central European Time*.

Une fois que Hello.java a été compilé en Hello.class, il est alors aussi possible de l'exécuter sur une autre machine possédant une machine virtuelle Java, mais avec la même version (ici 1.6) ou une version supérieure. Le fichier Hello.class sera aussi exécutable sous Linux avec sa propre machine virtuelle 1.6. Si la machine virtuelle ne trouve pas les ressources nécessaires, elle indiquera le problème. Dans certains cas, il faudra recompiler le code Java avec une version plus ancienne pour le rendre compatible.

Nous constatons donc qu'en Java, le fichier Hello.class ne contient pas de code machine directement exécutable par le processeur, mais du code interprétable par la machine virtuelle de Java. Ce n'est pas le cas du programme C++ hello.exe, qui utilise directement les ressources du système d'exploitation, c'est-à-dire de Windows. L'application hello.exe ne pourra pas être exécutée sous Linux et devra être recompilée (voir les exemples de l'annexe F).

La machine virtuelle Java – JRE

Dans cet ouvrage, nous devons tout de même mentionner la machine virtuelle JRE (*Java Runtime Environment*) bien que nous allons certainement passer la moitié de notre temps à éditer et compiler des programmes Java (l'autre moitié pour le C++).

Sur notre machine de développement, l'exécutable java.exe se trouve dans le répertoire C:\Program Files\Java\jdk1.6.0_06\bin. Si nous examinons le répertoire C:\Program Files\Java, nous découvrirons un second répertoire nommé jre1.6.0_06 ainsi qu'un sous-répertoire bin qui contient également un fichier java.exe.

Nos amis ou clients à qui nous livrerons la classe compilée Hello.class n'auront pas besoin d'installer JDK, le kit de développement, mais uniquement la machine virtuelle, c'est-à-dire JRE. Sun Microsystems met à disposition différentes distributions de JRE, juste pour exécuter notre Hello.class :

```
"C:\Program Files\Java\jre1.6.0_06\bin\java.exe" Hello
Hello world en Java: Wed Jul 30 13:58:09 CEST 2008
```

et ceci depuis le répertoire C:\JavaCpp\EXEMPLES\Chap01.

Le JDK n'est nécessaire que pour la compilation, c'est-à-dire lorsque nous utilisons la commande javac.exe. Sur une même machine, nous pourrions avoir plusieurs JDK et plusieurs JRE dans le répertoire C:\Program Files\Java (voir annexe B, section « Désinstallation des anciennes versions »).

Erreurs de compilation

L'oubli de la déclaration des ressources est une erreur classique. Si nous effaçons la première ligne (`import java.util.*;`) ou si nous la mettons en commentaire (`//` devant), nous générerons l'erreur suivante :

```
javac Hello.java
Hello.java:5: Class Date not found.
    Date aujourdhui;
    ^
Hello.java:6: Class Date not found.
    aujourdhui = new Date();
                     ^
2 errors
```

Il est à noter la manière claire dont le compilateur nous indique la position des erreurs. Ici, il ne trouve pas la classe `Date`. En ajoutant `import java.util.*`, nous indiquons au compilateur d'importer toutes les classes du *package* (paquet) des utilitaires de Java. Au lieu d'importer toutes les classes du package, nous aurions pu écrire :

```
import java.util.Date;
```

Compiler régulièrement est une très bonne habitude en Java et C++, en écrivant de petits morceaux et avant de terminer son code. Le simple oubli d'un point-virgule en C++ (qui indique la fin d'une instruction) peut entraîner une erreur quelques lignes plus loin et faire perdre inutilement du temps précieux.

Notre premier fichier Makefile

Pour compiler nos programmes Java et C++, nous allons utiliser tout au long de cet ouvrage un outil GNU bienvenu : le `make`. Le fichier `Makefile` est le fichier utilisé par défaut lorsque la commande `make` est exécutée sans paramètre. Cette commande est un héritage du monde Unix (Linux) que nous retrouverons aussi avec NetBeans (voir annexe E, section « Configuration pour le C++ et le make »). Le `Makefile` possède une syntaxe très précise basée sur un système de dépendances. Le `make` pourra identifier, en utilisant la date et l'heure, qu'une recompilation ou une action devra être exécutée. Dans le cas de fichiers Java, si un fichier `.java` est plus récent qu'un fichier `.class`, nous devrons considérer qu'un fichier `.class` devra être régénéré. L'outil `make` permet donc d'automatiser ce processus.

Voici notre tout premier exemple de `Makefile`, soit le fichier `MakefilePremier`, dont nous allons expliquer le fonctionnement. Les `Makefile` sont des fichiers texte ASCII que nous pouvons aussi éditer et exécuter sous Windows avec l'éditeur Crimson (voir annexe C) qui va nous simplifier le travail.

```
#
# Notre premier Makefile
#
all:          cpp java
```

```
cpp:          hello.exe hello2.exe
java:         Hello.class

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

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

hello2.exe:   hello2.o
              g++ -o hello2.exe hello2.o

hello2.o:     hello2.cpp
              g++ -c hello2.cpp

Hello.class:  Hello.java
              javac Hello.java

clean:
              rm -f *.class *.o *.exe
```

Avant d'exécuter ce `Makefile`, il faudrait s'assurer que tous les objets sont effacés. Dans le cas contraire, nous risquons de n'avoir aucun résultat tangible ou partiel, bien que cela resterait correct : uniquement les fichiers objets (`.o`, `.exe` et `.class`), dont les sources respectives ont été modifiées plus récemment, seront recompilés !

Nous aurions pu fournir un fichier `efface.bat` pour faire ce travail avec les commandes DOS :

```
del *.exe
del *.o
del *.class
```

mais nous avons préféré cette version :

```
make clean
```

L'entrée `clean` (nettoyer) va permettre ici d'effacer tous les fichiers `.class`, `.o` et `.exe` (s'ils existent) afin que le `make` puisse régénérer tous les objets et tous les exécutables. La commande `rm` est l'équivalent Linux de la commande DOS `del` et le paramètre `-f` va forcer l'effacement sans demander une quelconque confirmation. Un fichier `efface.bat` est fourni dans chaque répertoire des exemples et des exercices.

Le `make clean` et sa présence dans le `Makefile` sont importants : NetBeans (voir annexe E) en a besoin pour construire ses projets C++.

Mentionnons ce message que le lecteur pourrait rencontrer lors de son travail :

```
make: *** No rule to make target `clean'.  Stop.
```

Il indique qu'il n'y a pas de règle (`rule`) pour le point d'entrée `clean`. Nous devrons alors le rajouter dans le `Makefile` ou vérifier que la syntaxe du fichier est correcte (espaces et marques de tabulations tout particulièrement).

Enfin un premier make effectif

Pour exécuter le fichier `MakefilePremier`, nous devrons entrer la commande :

```
make -f MakefilePremier
```

En navigant dans le répertoire, nous pourrons constater que les fichiers `.o`, `.exe` et `.class` ont été régénérés. L'erreur sur la recompilation de `hello2.cpp` réapparaîtra évidemment (voir ci-dessus).

Si nous avons un nom de fichier `Makefile`, comme c'est le cas dans tous les chapitres, il nous faudra simplement exécuter :

```
make
```

Lorsque la commande `make` est exécutée, le fichier `Makefile` sera chargé et le point d'entrée `all` exécuté. La commande `make` permet aussi de spécifier un point d'entrée :

```
make -f MakefilePremier java
```

Ici, uniquement le point d'entrée `java` serait activé pour compiler la classe `Hello.class`. Les points d'entrée `Hello.class` ou `hello.exe` sont aussi possibles pour un choix plus sélectif.

`all`, `cpp` et `java` sont des actions à exécuter. `hello.exe`, `hello.o` et `Hello.class` sont des fichiers générés par les compilateurs. Il faut être très attentif avec le format des fichiers `Makefile`, car ils sont extrêmement sensibles à la syntaxe. Après les deux-points (`:`), il est préférable d'avoir un tabulateur (TAB), bien que nous puissions avoir des espaces, mais sur la ligne suivante nous avons toujours des tabulateurs. Suivant la grandeur des variables, nous en aurons un ou plusieurs, mais cela dépend aussi de la présentation que nous avons choisie. Après les `hello.exe:` et `hello.o:`, nous trouvons les dépendances et sur les lignes suivantes les commandes.

À la première exécution du `make`, `cpp` et `java` seront activés, car aucun des `hello.o`, `hello.exe` et `Hello.class` n'existe. En cas d'erreur de compilation dans `hello.cpp`, ni `hello.o` ni `hello.exe` ne seront créés.

Passons maintenant à la partie la plus intéressante : si `hello.cpp` est modifié, sa date sera nouvelle et précédera celle de `hello.o`. Les deux commandes `g++` seront alors exécutées.

Dans ce cas précis, la séparation en deux parties `hello.o` et `hello.exe` n'est pas vraiment nécessaire, puisque nous n'avons qu'un seul fichier `hello.cpp`. La commande suivante suffirait :

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

Enfin, que se passe-t-il si nous définissons :

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

`hello.o` sera régénéré si `Hello.java` a changé et même si `hello.cpp` n'a pas été touché. On voit donc que cette dépendance est inutile.

Les trois premières lignes du fichier `MakefilePremier` sont des commentaires. Ils commencent par le caractère # :

```
#
# Notre premier Makefile
#
```

Nous pouvons, par exemple, les utiliser pour éliminer des parties de compilation pendant le développement :

```
all:          cpp #java
```

Ici, uniquement la partie C++ sera compilée.

Nous reviendrons sur les dépendances dues aux fichiers d'en-tête au chapitre 4. Les paramètres `-c` et `-o` y seront expliqués en détail.

Le point d'entrée main()

La fonction `main()` est le point d'entrée de tout programme. Le corps de la fonction située entre les accolades sera exécuté. En Java, il n'est pas possible de définir une fonction indépendante, car elle doit faire partie d'une classe. De plus, elle doit être déclarée `public` et `static`. Nous en comprendrons les raisons plus loin dans cet ouvrage. Contrairement au C++, chaque classe en Java peut posséder son entrée `main()`. Lorsque nous entrons :

```
java Hello
```

la méthode `main()` de la classe `Hello.class` est activée. La classe `Hello` pourrait utiliser d'autres classes possédant aussi un point d'entrée `main()`. Nous verrons plus loin que cette technique peut être employée pour tester chaque classe indépendamment.

Les paramètres de main()

`main()` peut recevoir des paramètres. Lorsque nous entrons une commande DOS telle que :

```
copy hello.cpp hello.bak
```

les fichiers `hello.cpp` et `hello.bak` sont les deux paramètres reçus par la commande `copy`. Nous allons examiner à présent les différences entre les deux langages et la manière de procéder pour tester et acquérir des paramètres. Par exemple, si nous devions programmer la commande DOS `copy` (équivalente à la commande `cp` sous Linux), il faudrait que nous vérifiions les paramètres de la manière qui va suivre.

main() et C++

Le fichier `copy_arg.cpp` peut se présenter sous cette forme :

```
// copy_arg.cpp
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  if (argc != 3) {
    cerr << "Nombre invalide de paramètres" << endl;
    return -1;
  }

  cout << "argv[0]: " << argv[0] << endl;
  cout << "argv[1]: " << argv[1] << endl;
  cout << "argv[2]: " << argv[2] << endl;
  return 0;
}
```

Ce programme C++ peut se compiler avec :

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

ou avec un `Makefile` qui se trouve sur le CD-Rom. Nous pouvons ensuite l'exécuter avec :

```
copy_arg hello.cpp hello.bak
```

Il faut absolument joindre ces deux arguments, sous peine de récolter une erreur (nombre invalide de paramètres). Le résultat sera ainsi présenté :

```
argv[0]: copy_arg
argv[1]: hello.cpp
argv[2]: hello.bak
```

Nous savons déjà que `copy_arg.exe` peut être abrégé en `copy_arg` sous DOS, mais pas sous Linux.

main() et Java

Le fichier `CopyArgs.java` est en fait très similaire si nous l'étudions en détail :

```
public class CopyArgs {
  public static void main (String[] args) {

    if (args.length != 2) {
      System.err.println("Nombre invalide de paramètres");
      return;
    }
```

```
    System.out.println("args[0]: " + args[0]);
    System.out.println("args[1]: " + args[1]);
  }
}
```

Ce programme Java peut se compiler avec :

```
javac CopyArgs
```

ou un `Makefile`, combiné avec la compilation de la version C++ ci-dessus, qui se trouve sur le CD-Rom. La classe `CopyArgs.class` compilée sera exécutée avec la machine virtuelle Java de cette manière :

```
java CopyArgs hello.cpp hello.bak
```

Nous obtiendrons le résultat suivant :

```
args[0]: hello.cpp
args[1]: hello.bak
```

Analyse comparative

La première grande différence est le nombre de paramètres retournés, c'est-à-dire un de plus pour la version C++, où nous recevons (dans `argv[0]`) le nom du programme. Les `strings` existent aussi en Standard C++, mais la fonction `main()` est un héritage du C.

L'instruction `if` viendra au chapitre 3, mais la traduction de :

```
if (argc != 3) {
```

serait « si `argc` n'est pas égal à 3 alors… ».

Les différentes sorties à l'écran, `cout` et `cerr` pour C++, ainsi que `out` et `err` en Java, sont utilisées sur les deux canaux de sorties. Il est possible, plus particulièrement sous Linux, de filtrer ces deux sorties pour différencier les cas normaux (`cout`, `out`) des erreurs (`cerr`, `err`). Nous ne donnerons pas ici d'exemples sur l'utilisation de telles techniques, mais nous garderons cependant l'habitude d'utiliser le canal `cerr`/`err` pour les erreurs.

Observons ce qui se passe avec l'instruction :

```
cout << "argv[0]: " << argv[0] << endl;
```

Le premier « `argv[0]:` » est simplement un texte. Le texte entre guillemets est envoyé au travers de l'opérateur `<<` à `cout`. Ce dernier est défini dans le fichier `iostream`, qui se trouve dans la bibliothèque standard (`std`) du Standard C++. Nous reviendrons plus tard sur ces détails.

Les sorties à l'écran (`cout`) pouvant être chaînées, le terme suivant se trouve être `argv[0]`, `argv` étant un pointeur à une liste de pointeurs, c'est-à-dire la raison du `**` dans la déclaration du `main()`. `argv[0]` va pointer sur l'adresse mémoire où figure le premier paramètre, c'est-à-dire le nom de la commande. Si nous écrivions :

```
cout << "Adresse de argv[0]: " << &argv[0] << endl;
```

nous verrions apparaître sous forme hexadécimale l'adresse mémoire où se trouve en fait le texte « copy_arg.exe ». On peut également concevoir argv comme un tableau de chaînes de caractères, le 0 de argv[0] étant un index dans ce tableau.

Enfin, le endl est une opération aussi définie dans <iostream>, qui consiste à envoyer le caractère de saut de ligne, que l'on représente parfois comme le \n ou \010 (valeur octale).

En Java, nous n'avons pas de chaînes de caractères comme en C++, mais un String. Celui-ci est un objet représentant aussi une chaîne de caractères. length est une opération exécutée par le compilateur qui nous permet de recevoir la dimension de la variable args, qui est un tableau de String identifié avec le []. En C et en C++ nous recevons la variable argc. La dimension de args doit être de 2 dans notre exemple, ce qui nous permet d'obtenir les deux paramètres args[0] et args[1]. L'index commence toujours à 0 dans un tableau. L'index maximal est args.length -1.

L'objet out dans la bibliothèque Java System permet de sortir du texte sur la console au moyen de println(). out, défini dans le package java.lang.System, est un objet statique de la classe PrintStream ; println() est une méthode (fonction) de la classe PrintStream. Nous verrons ces concepts plus loin. Il n'est pas nécessaire d'importer le package System avec une instruction import, car il est reconnu par le compilateur.

Le ln de println() est équivalent au endl du C++. Dans cet exemple Java, nous utilisons l'opérateur + pour associer deux strings avant de les envoyer au travers de println(). Il est aussi possible d'utiliser plusieurs print() et le "\n" comme ici :

```
System.out.print("args[0]: ");
System.out.print(args[0]);
System.out.print("\n");
```

Le return est équivalent en C/C++ et en Java. En Java, nous avons un void main() et nous ne devons pas retourner de valeur. Le return peut d'ailleurs être omis si nous atteignons correctement la fin du programme. Avec un int main() en C++, il est nécessaire de retourner une valeur qu'il est judicieux de définir négativement en cas d'erreur.

Toute démarche doit se faire d'une manière logique. Si nous devions écrire les programmes copy_arg.cpp et Copy_arg.java en totalité, il faudrait vérifier l'existence du fichier entré comme premier paramètre (ici, hello.cpp). Ensuite, si le deuxième paramètre était un fichier existant (ici, hello.bak), il faudrait décider si nous acceptons de l'écraser ou non. Nous pourrions alors demander, avec un oui ou un non, si cela est acceptable ou encore définir un autre paramètre (par exemple, –f) pour forcer l'écriture. Nous aurions alors une commande comme :

```
copy -f hello.cpp hello.bak
```

Comme -f serait optionnel, il faudrait traiter les arguments différemment. Enfin, nous pourrions pousser l'analyse à l'extrême et vérifier si l'espace disque est suffisant avant de commencer la copie. En cas d'erreur, il faudrait effacer la mauvaise copie. Si nous n'avions pas la permission d'écrire, cela affecterait vraisemblablement la logique du traitement des erreurs ! Il y a en effet des cas en programmation où nous découvrons qu'un oubli

dans l'analyse ou la conception (*design,* en anglais) peut entraîner une restructuration complète du code !

> **Note**
>
> Que se passe-t-il sous DOS ?
>
> Si nous exécutons cette ligne d'instruction C++ sous DOS :
>
> `cout << "éàöâê" << endl;`
>
> cela pourrait nous surprendre au premier abord. En effet, les caractères sous DOS sont différents. Les lettres et caractères de la langue anglaise vont apparaître correctement, alors que les lettres avec accents poseront quelques difficultés. Mais comme nous exécuterons nos programmes avec Crimson, c'est-à-dire sous Windows, nous ne rencontrerons pas ce type de problème. Nous reviendrons sur ce point plus loin dans cet ouvrage.

Jouer avec Crimson

C'est sans doute le meilleur moment pour retourner dans l'éditeur Crimson (voir annexe C) et pour refaire quelques-unes des démarches et fonctions usuelles :

- Charger un fichier `.java`, le compiler et l'exécuter.
- Charger un fichier `.cpp` , le compiler et l'exécuter.
- Charger un `Makefile` et l'exécuter avec le `make`.

Constater que nous pouvons aussi charger des fichiers `.bat` (`efface.bat` ou `Makefile.bat`) et les exécuter dans Crimson directement et sans fenêtre DOS.

Résumé

À la fin de ce chapitre, nous savons déjà compiler des programmes Java et C++, bien qu'il ne soit pas encore possible d'écrire un programme digne de ce nom. L'outil `make` de GNU permet d'automatiser le processus lorsque des changements dans le code ont été apportés.

Exercices

Toutes les solutions des exercices de cet ouvrage sont disponibles sur le CD-Rom (voir annexe B, section « Installation des exemples et des exercices »). En programmation, il n'y a pas UNE solution, mais plusieurs. Il vaut cependant la peine de s'accrocher et de faire quelques-uns de ces exercices, voire tous, avant de consulter les solutions.

1. Écrire une classe Java `Bonjour` qui nous sortira un :

   ```
   Bonjour et bonne journée
   ```

2. Écrire le `Makefile` pour cette classe et vérifier qu'une compilation est à nouveau exécutée si le fichier `Bonjour.class` est effacé ou bien si le fichier `Bonjour.java` est modifié avec un autre texte de salutations !

3. Écrire une version de `copy_arg.cpp` et de `CopyArgs.java` qui, lorsque aucun paramètre n'est donné, imprime à l'écran un descriptif des paramètres nécessaires aux programmes.
4. Créer un fichier `execHello.bat` pour exécuter les binaires `Bonjour.class` et `bonjour.exe` des exercices 1 et 2 précédents. Ajouter une pause en fin de fichier et exécuter ce dernier deux fois depuis Crimson. Examiner l'effet de la pause et le résultat obtenu en double-cliquant sur le fichier depuis l'explorateur de Windows.