Programmation C et Java

(programmation et outils associés)

– Travaux Dirigés (**4**) –

Résumé

Le but de ce TD est de vous initier aux outils de la programmation C et Java. Vous apprendrez à utiliser les compilateurs classiques pour faire des programmes en C et en Java ainsi que les principes de base dans ces langages et quelques outils de développement qui pourront vous être utiles.

Table des matières

Avant propos

« *Un peu de programmation éloigne de la logique mathématique ; beaucoup de programmation y ramène.* »
Xavier Leroy (Chercheur en informatique)

La programmation requiert énormément de rigueur, de logique et une certaine compréhension des mécanismes internes des ordinateurs. Mais, outre ces pré-requis indispensables, il faut aussi bien connaître un ou plusieurs langages de programmation. Dans cette feuille d'exercices nous mettrons l'accent sur le langage C et le langage Java qui sont tous deux des langages très courants dans le monde informatique.

1 Le langage C

La première version du langage C a été créée par Denis Ritchie entre 1969 et 1973 dans les laboratoires d'AT&T (Bell Labs). Le but était de disposer d'un langage de programmation de plus haut niveau que l'assembleur pour programmer un système d'exploitation qui soit déployable facilement sur différentes plate-formes. C'est de cette initiative qu'est aussi né le système UNIX.

Le langage C est un langage impératif statiquement typé, l'une de ses caractéristiques majeures est qu'il laisse au programmeur le soin de gérer la mémoire du programme directement (contrairement à beaucoup d'autres langages qui masquent la gestion de la mémoire au programmeur). C'est un langage qui est considéré comme l'un des plus proche de la machine et du langage assembleur, ce qui lui permet d'obtenir des performances que les autres langages ne peuvent pas atteindre mais, en contre partie, demande beaucoup plus de soin et de prudence dans la gestion de la mémoire afin d'éviter les bugs.

```c
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  puts ("Hello World!");
  return EXIT_SUCCESS;
}
```

Listing 1 – Programme « *Hello World !* » en langage C.

1.1 Premiers programmes en C

Nous commencerons par quelques petits programmes pour vous familiariser avec le processus d'écriture et de compilation d'un programme en C.

1.1.1 *Hello World !*

Questions

1. Recopiez le programme représenté sur le listing 1 dans un fichier `hello.c`.
2. Compilez le avec la ligne de commande suivante : `gcc -Wall -Wextra -std=c99 -o hello hello.c`
3. Enfin, exécutez le en faisant : `./hello`
4. Faites un `man gcc` pour y trouver l'explication des différentes options que vous avez tapées.
5. En utilisant les paramètres `int argc` et `char** argv` de la fonction `main`, transformez votre programme pour qu'il prenne un argument optionnel et affiche « `Hello World!` » autant de fois que demandé, comme ceci :

```
sh> ./hello
Hello World!
sh> ./hello 4
Hello World!
Hello World!
Hello World!
Hello World!
```

Fonctions à utiliser : `atoi()`, `fprintf()`

1.1.2 Somme des multiples de 3 et 5

Questions

1. Si on liste tous les nombres naturels en dessous de 10 qui sont multiples de 3 ou 5, on obtient : 3, 5, 6 et 9. La somme de ces multiples est 23. Écrivez un programme qui trouve la somme de tous les multiples de 3 et 5 en dessous d'un nombre donné en argument comme ceci (on utilisera l'opérateur modulo (`%`) du langage C) :

```
sh> ./multiples_3_5 10
The sum of the multiples of 3 and 5 below 10 is 23.
```

2. Donnez la somme des multiples de 3 et 5 en dessous de 1000.
3. Faites en sorte que votre programme donne un message d'erreur lorsqu'aucun argument n'est donné :

```
sh> ./multiples_3_5
multiples_3_5: error: missing argument !
```

Fonctions à utiliser : `atoi()`, `fprintf()`

1.2 Compilation séparée et bibliothèques dynamiques

La compilation séparée est une méthode qui permet de *modulariser* votre programme en rassemblant les fonctions qui le composent en des ensembles cohérents qui pourront éventuellement être utilisés dans d'autres programmes ensuite.

Une bonne modularité d'un programme permet souvent une meilleure compréhension de celui-ci (même si vous ne l'avez pas écrit) et une plus grande facilité à le faire évoluer.

1.2.1 *Hello World !* modulaire

Dans cet exemple introductif, nous allons exagérer intentionnellement l'aspect modulaire. Notre but ici est de vous montrer comment cela marche et non de vous enseigner quand et comment il faut y avoir recours dans le cadre d'un vrai programme.

Questions

1. Commencez par écrire un module `hello` sous la forme d'un fichier d'en-tête (`hello.h`) :

```
#ifndef HELLO_H
#define HELLO_H

void hello (void);

#endif
```

   À quoi sert ce fichier ? Et, quelle est l'utilité des commandes `#ifndef`, `#define` et `#endif`.

2. Écrivez aussi un fichier source (`hello.c`) qui contiendra une fonction `hello()` qui affiche la célèbre phrase. N'écrivez pas de fonction `main()` ! Et compilez le avec la ligne de commande suivante (lisez la page de manuel de `gcc` à propos des options '`-c`' et '`-I`') :

```
sh> gcc -Wall -Wextra -std=c99 -I. -c -o hello.o hello.c
```

3. Écrivez le programme principal dans le fichier `main.c` qui contiendra le fonction `main()` et qui appelera la fonction `hello()` avant de quitter. Compilez le tout avec la ligne suivante :

```
sh> gcc -Wall -Wextra -std=c99 -c -I. -o main.o main.c
sh> gcc -Wall -Wextra -std=c99 -o main main.o hello.o
```

1.2.2 Utilisation d'une bibliothèque dynamique

Dans certains cas, nous auront besoin de fonctions qui ne sont pas par défaut dans le langage C. Il nous faudra recourir à des appels à des bibliothèques dynamiques. C'est ce que nous allons illustrer ici.

Questions

1. Reprenez le programme `main.c` que vous venez d'écrire. Ajoutez y l'affichage de la valeur de racine de 2 (`sqrt(2)`). Compilez-le et exécutez le.
2. Quel est le nom du fichier contenant la bibliothèque de `math` et quel est le lien avec l'option '`-lm`' ?
3. Si vous aviez écrit et compilé votre propre bibliothèque dynamique, à travers quelle option indiqueriez-vous son chemin relatif (`./path/to/lib/`) à `gcc` ?

1.3 Automatisation de la chaîne de compilation

L'ensemble des commandes à taper pour produire le programme exécutable final est tout de même assez contraignant. De plus, il faut se rappeler si l'on a modifié (ou pas) les fichiers sources pour savoir si la compilation est nécessaire ou pas. Heureusement, il existe des outils assez simples qui permettent l'automatisation de cette tâche de compilation et de résolution de dépendances. Ne négligez pas ces outils, si vous êtes dans une situation où vous devez produire rapidement du code, chaque minute que vous pourrez vous épargner sera précieuse.

Les Makefiles sont des fichiers textes qui contiennent les règles de construction de votre programme que vous aurez écrites vous-même. Ils s'appellent `Makefile` par convention et c'est l'outil `make` qui permet de les exécuter.

Le principe d'écriture d'une règle d'un Makefile est toujours le même :

```
cible: dépendances
<tab>   règles de construction
```

La `cible` peut être soit un fichier qu'il va falloir créer (par exemple votre exécutable), soit une étiquette qui vous permettra de déclencher la règle à partir de la commande `make <cible>`.

La commande `make`, lorsqu'elle est invoquée sans argument exécute la première cible. Et, par convention, on appelle toujours cette première cible '`all`'.

Questions

1. Écrivez un Makefile qui regroupe toutes les règles nécessaires pour produire l'exécutable final `main`.
2. Quelles règles sont appliquées si on ne touche qu'à `main.c`, et si on ne touche qu'à `hello.c` ? Quel fichier faudrait-il modifier pour que toutes les règles soient appliquées à nouveau ?
3. Ajoutez une règle `clean` qui efface tous les fichiers produits par la compilation.
4. Ajoutez une règle `help` qui liste toutes les règles dans le Makefile, comme ceci :

```
Makefile usage :
        - make [all]                build the document
        - make clean                erase unuseful files
        - make help                display this help
```

5. Enfin, expliquez à quoi servent les variables suivantes et utilisez les de manière adéquate à l'intérieur de votre Makefile : `CC`, `CFLAGS`, `CPPFLAGS`, `LDFLAGS`.

1.4 Débugguer les programmes

Le débuggage est un processus aussi important que la programmation car les bugs sont souvent inévitables dès que les programmes deviennent un peu compliqués. En langage C, les deux principaux outils de débuggage sont `gdb` et `valgrind`. Ces deux outils sont vraiment complémentaires car ils fournissent des informations très différentes sur le programme étudié.

1.4.1 `gdb`

Cet outil est ce que l'on appelle un débuggeur, il permet de stopper l'exécution du programme et de le suivre en mode pas-à-pas tout en laissant libre accès à toute la mémoire du logiciel.

Questions

1. Compilez un des programmes précédent en ajoutant l'option '`-g`' aux options du `CFLAGS`. Puis, lancez `gdb` sur ce programme en faisant simplement : `gdb <program>`.
2. Faites la commande `help`, puis `help break`.
3. Positionnez un *breakpoint* sur la fonction `main()` de votre programme et exécutez le avec `run`.
4. Faites une exécution pas-à-pas en utilisant la commande `next`.
5. Copiez le programme suivant et trouvez le bug qui cause un *segmentation fault* en suivant le programme avec `gdb` :

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_N 15

int main()
```

```
{
  int array[10];

  for (int i = 0; i < MAX_N; i++)
    array[i] = 1;

  printf ("array[10] = %d\n", array[10]);

  return EXIT_SUCCESS;
}
```

1.4.2 `valgrind`

`valgrind` est un outil qui traque l'utilisation mémoire de votre programme. Comme le langage C requiert de gérer soit même presque tous les aspects de la mémoire, il est extrêmement fréquent de faire des erreurs mémoire lors de la conception de son programme. `valgrind` vous permettra de mettre en évidence ces erreurs, même si vous ne les aviez pas forcément remarquées à premier abord.

Contrairement à `gdb`, `valgrind` n'est pas interactif. Il exécute votre programme dans un environnement virtualisé dans lequel il trace tous les accès mémoire (lectures/écritures). Une fois le programme terminé, il dresse un bilan des erreurs qui ont pu survenir pendant l'exécution de votre logiciel. Notez bien qu'exécuter votre logiciel dans cet environnement virtualisé est entre 5 à 10 fois plus lent que la normale. Prévoyez donc un peu de temps si jamais votre programme fait beaucoup de calculs.

> **Attention**: `valgrind` ne peut vérifier qu'un seul chemin d'exécution à la fois. C'est à dire que lorsque vous testez votre logiciel, la validation de `valgrind` ne vaudra que pour le chemin et les valeurs prises lors de cette exécution. Il est parfaitement possible que des bugs subsistent si vous changez les arguments de départ ou que vous passez par des chemins du code que vous n'avez pas encore exploré.

Principalement, `valgrind` se focalise sur les types de problèmes suivants :

- **Les branchements fait sur des valeurs non-initialisées** : Lorsqu'un test utilise une partie de la mémoire qui n'a pas été initialisée avant.
- **Les lectures/écritures illégales** : Lorsque le programme lit ou écrit à des endroits de la mémoire où il ne devrait pas être.
- **Les fuites mémoire** : Il s'agit de places mémoires qui ont été allouées par le programme mais qui ne sont jamais désallouées et qui restent jusqu'à la fin du programme.

Questions

1. Commencez par prendre l'un des programmes que vous avez réalisé pendant cette session et exécutez-le sous `valgrind` :

   ```
   sh> valgrind <program> <program_options> <program_arguments>
   ```

   Observez la sortie de `valgrind`. Il ne devrait, normalement pas y avoir de problème.

2. Copiez le programme suivant, compilez-le (avec l'option '`-g`') et observez le résultat de `valgrind` lorsque vous exécutez le programme avec un argument et sans argument. Puis, corrigez les bugs du programme.

   ```
   #include <stdio.h>
   #include <stdlib.h>

   int main()
   {
       int age = 10, height;
   ```

```
    if (argc < 2)
       height = 178;

    printf("I am %d years old.\n");
    printf("I am %d cm tall.\n", height);

    return EXIT_SUCCESS;
}
```

3. Mêmes questions avec le programme suivant :

```
#include <stdio.h>
#include <stdlib.h>

typedef struct StudentRec
{
  long int ID;
  char name[20];
} Student;

void printStudent(Student* s)
{
  printf("\n ID : %ld \n Name: %s \n\n", s->ID, s->name);
  return;
}

void readStudent(Student* s)
{
  printf("Enter ID: ");
  scanf("%ld", &s->ID);

  printf("Enter name: ");
  scanf("%19s", s->name);

  return;
}

int main()
{
  Student *temp = NULL;

  readStudent(temp);
  printStudent(temp);

  return EXIT_SUCCESS;
}
```

2 Le langage Java

Le langage Java a été initié par James Gosling, Mike Sheridan, and Patrick Naughton en 1991. Originellement, ce langage devait à créer des applications pour la TV payante par câble (jeux, enregistrement d'émissions, émission interactives, ...), mais, très vite, le langage s'est avéré trop complexe pour les appareils de l'époque. En parallèle, le langage lui-même a été remarqué comme un très bon langage généraliste orienté objet, surtout grâce à ses applications sur le Web via les Java Applets (petits logiciels Java exécuté dans les navigateurs). L'apparition du JavaScript remet actuellement en cause ses fondements

```java
/* This class is the Class HelloWorld meant to be a Java example. */
public class HelloWorld {
    /* This method is the main method of the class HelloWorld */
    public static final void main(String[] args) {
      System.out.println("Hello World !");
    }
}
```

Listing 2 – Programme « *Hello World !* » en langage Java.

Web, mais le langage lui-même est tellement utilisé dans le monde de l'industrie qu'il est devenu un standard pour de très nombreuses applications *server-side* et qu'il continu à être développé activement.

Le but de cette section est de prendre en main les principes de base de Java ainsi que votre environnement de travail pour Java. L'outil principal sera l'IDE Eclipse, mais nous allons d'abord commencer par apprendre à tout faire à la main avant d'aborder l'environnement intégré d'Eclipse.

2.1 Premiers programmes en Java

L'une des originalités de Java vient du fait qu'il doit être compilé mais qu'il ne produit pourtant pas de fichier exécutable *natif*. Il produit un fichier intermédiaire contenant du *bytecode* qui pourra être exécuté sur une machine virtuelle qui est disponible, le plus souvent, avec le compilateur.

Figure 1 – Chaîne de création et d'exécution d'un logiciel Java.

Le fichiers source ont l'extension '`.java`', une fois compilés, ils contiennent du bytecode Java et ont l'extension '`.class`'. Ce sont ces fichiers '`.class`' qui seront chargés dans la machine virtuelle Java pour être exécutés.

2.1.1 *Hello World !*

Commençons par le traditionnel programme *Hello World !*. Le fichier doit impérativement s'appeler `HelloWorld.java` car chaque classe est programmée au sein d'un seul fichier qui porte le même nom.

Questions

1. Commencez par vérifier que les différents outils sont bien là et sous quelle version (compilateur, puis machine virtuelle) :

   ```
   sh> javac -version
   sh> java -version
   ```

2. Recopiez le programme représenté sur le listing 2 dans un fichier `HelloWorld.java`.
3. Compilez le avec la ligne de commande suivante : `javac HelloWorld.java`
4. Enfin, exécutez le en faisant : `java HelloWorld` (ne mettez pas le '`.class`'! On fait référence ici à la classe qu'il faut exécuter et non au fichier).
5. Faites un Makefile qui contienne les règles de compilation et d'exécution du fichier Java.
6. Explorez la documentation Java en ligne pour Java7 [1] (ou dans la version que vous utilisez) et essayez d'y trouver le type de l'objet `java.lang.System.out`.

1. `http://docs.oracle.com/javase/7/docs/api/`

7. Rajoutez des commentaires Javadoc [2] pour la classe '`HelloWorld`' et la méthode '`main`'. Puis, produisez la documentation avec l'utilitaire '`javadoc`'.
8. Évidemment les applications Java ont souvent besoin d'énormément de fichiers '`*.class`' pour s'exécuter. Afin de faciliter la distribution de logiciels Java on utilise des fichiers JAR (Java ARchive). La commande '`jar`' permet de construire ces archives, elle fonctionne de manière très similaire à la commande UNIX '`tar`'.

   Bien que le programme `HelloWorld` ne soit constitué que d'un seul fichier, nous allons créer une telle archive et nous l'exécuterons. Commencez par créer un fichier `Manifest` qui contienne les lignes suivantes :

```
Manifest-Version: 1.0
Created-By: Me!
Main-Class: HelloWorld
```

   La seule ligne vraiment importe est celle qui indique la `Main-Class` qu'il faudra lancer au démarrage du logiciel. Les autres lignes sont uniquement informatives.

   Pour construire une archive complète, il nous faut le Manifest ainsi que l'ensemble des fichiers '`*.class`' qui constituent le logiciel (ici, `HelloWorld.class` suffit) :

```
sh> jar cfmv HelloWorld.jar Manifest HelloWorld.class
added manifest
adding: HelloWorld.class(in = 427) (out= 290)(deflated 32%)
```

   Voilà, vous avez une archive qui est transportable partout. Pour l'exécuter, il suffit de demander à la machine virtuelle de le faire :

```
sh> java -jar HelloWorld.jar
Hello World !
```

2.1.2 Somme des multiples de 3 et 5

Questions

1. Si on liste tous les nombres naturels en dessous de 10 qui sont multiples de 3 ou 5, on obtient : 3, 5, 6 et 9. La somme de ces multiples est 23. Écrivez un programme qui trouve la somme de tous les multiples de 3 et 5 en dessous d'un nombre donné en argument comme ceci (on utilisera l'opérateur modulo (`%`) du langage Java) :

```
sh> java SumMultiples35 10
The sum of the multiples of 3 and 5 below 10 is 23.
```

2. Donnez la somme des multiples de 3 et 5 en dessous de 1000.
3. Faites en sorte que votre programme donne un message d'erreur lorsqu'aucun argument n'est donné :

```
sh> java SumMultiples35
SumMultiples35 : error: missing argument !
```

   Fonctions à utiliser : `Integer.parseInt()`, `System.out.println()`, `System.err.println()`, `System.exit()`.

2.2 Eclipse IDE

Eclipse est LA plate-forme des développeurs Java. C'est un environnement de développement complet, lui-même écrit en Java, qui contient tout ce qu'il faut pour écrire du code Java, le compiler, l'exécuter et le débugguer. Et plus encore si vous ajoutez quelques plug-ins. Son développement a démarré en 2001, initialement soutenu par IBM mais très vite un consortium de grandes entreprises se forme autour du développement de l'outil et prend en main l'évolution de l'IDE.

Notez qu'il existe quelques tutoriaux auxquels vous pouvez vous reporter lorsque vous ne trouvez pas comment faire :

2. Voir : `http://java.sun.com/j2se/javadoc/faq/index.html`

— *Eclipse IDE Tutorial* de Lars Vogel.
`http://www.vogella.de/articles/Eclipse/article.html`

— *Eclipse And Java : Free Video Tutorials.*
`http://eclipsetutorial.sourceforge.net/`

— *Eclipse Documentation* par la Eclipse Foundation.
`http://www.eclipse.org/documentation/`

2.2.1 Création d'un Projet Java

Questions

1. Lancez la commande '`eclipse`'[3].
2. Créez un nouveau projet Java (menu File → New → Project...) dont le nom sera '`HelloWorld`' (nous allons refaire le programme *HelloWorld* de tout à l'heure à travers Eclipse).
3. Créez un package '`hello`' qui contienne une classe '`HelloWorld`' avec une méthode '`public static void main(String[] args)`'.
4. Dans le programme `HelloWorld`, faites afficher un "**Hello World !**" sur la sortie standard.
5. Enfin, voici quelques documentations Java (officielles) auxquelles vous pourrez vous reporter en cas de problème (remplacez le `<X>` par votre version de Java) :
   — Java `<X>` : `http://download.oracle.com/javase/<X>/docs/`.
   — API Java `<X>` : `http://download.oracle.com/javase/<X>/docs/api/`
   — Tutoriels Java : `http://download.oracle.com/javase/tutorial/`
   — Java Tutorials Learning Paths (pour vous aider à vous y retrouver) :
   `http://download.oracle.com/javase/tutorial/tutorialLearningPaths.html`

2.2.2 Quelques autres fonctionnalités d'Eclipse

Questions

1. De la même façon qu'auparavant, rajoutez des commentaires Javadoc pour le package '`hello`', la classe '`HelloWorld`' et la méthode '`main`'. Puis, produisez la documentation à travers Eclipse.
2. Utilisez Eclipse pour positionner un breakpoint[4] sur le début de votre programme '`HelloWorld`'. Puis exécutez votre programme en mode '*debug*'.
3. Utilisez le menu "**Refactoring**" pour renommer un package ou une variable.

2.2.3 Installation de plugins

Eclipse utilise très largement les plug-ins pour pouvoir ajouter des fonctionnalités annexes à l'environnement. Cette section vous montrera comment installer votre premier plugin.

Questions

1. Allez dans le menu '**Help**' → '**Install New Software...**'. Sélectionnez le champs '**Xxx – http ://download.eclipse.org/releases/xxx**' dans **Work with ...**
2. Sélectionnez et installez les plugins suivants :
   — **General Purpose Tools → Marketplace Client**
3. Installez ces plugins et redémarrez Eclipse.
4. En utilisant la '*Marketplace*' (**Help → Eclipse Markeplace**), installez le plugin '**FindBugs**' (par exemple).

3. Attention, le lancement peut prendre plusieurs dizaines de secondes, ne vous impatientez pas !
4. Voir : `http://www.vogella.de/articles/EclipseDebugging/article.html`

3 Git

3.1 Intialiser votre premier dépôt

Questions

1. Initialiser votre configuration globale :

```
sh> git config --global user.name <name>
sh> git config --global user.email <email>
sh> git config --global core.editor emacs
```

   Vérifiez que tout s'est bien passé :

```
sh> git config --list
```

2. Créez votre propre dépôt :
   Créez un répertoire caché à la racine de votre compte : `~/.git-repo/`. Il sera accessible ('`+x`') mais pas lisible ('`-r`'), ce qui va empêcher les utilisateurs de faire un '`ls`' dessus. Il n'y aura plus qu'à choisir un nom aléatoire suffisamment complexe pour le répertoire contenant le dépôt git afin d'empêcher n'importe qui d'accéder à ce répertoire.

```
sh> mkdir ~/.git-repo
sh> chmod a+x,g-rw,o-rw ~/.git-repo
```

   Choisissez ensuite un nom aléatoire avec la commande '`pwgen 10`' comme suit :

```
sh> pwgen 10

iVe7eeteu3 feith2Aegh oShi4quiex so0faefaiH Ahnohquu2o eshirae1oL
eehohRi2ch cee7ooXaiw jo3eik3Ru8 paip1aegeP au7aegeiSh DohToh7owu
...
```

   Par exemple, nous choisirons le premier ('`iVe7eeteu3`') pour l'exemple et on initialise notre dépôt :

```
sh> git init --bare /net/cremi/<user>/.git-repo/iVe7eeteu3
```

3. À présent, on doit créer un clone du dépôt central afin de pouvoir avoir une copie de travail :

```
sh> cd ~/path/to/the/place/you/want/your/wc
sh> git clone ~/.git-repo/<pwd>/ sudoku/
```

4. Si vous désirez vous synchroniser avec votre dépôt central depuis l'extérieur du CREMI, vous pouvez utiliser cette commande :

```
sh> git clone \
ssh://<user>@jaguar.emi.u-bordeaux.fr/net/cremi/<user>/.git-repo/<pwd> sudoku
```

   Une fois cloné, le dépôt n'a plus besoin de l'adresse complète, vous pouvez faire comme à l'acoutumé.

5. Finalement, mettez les droits correctement sur les différents répertoires de votre dépôt central :

```
sh> chmod a+x g-rw o-rw ~/
sh> chmod a+x g-rw o-rw ~/.git-repo/
sh> chmod a+rx ~/.git-repo/<pwd>/
```