UNIVERSITÉ DE MONTPELLIER
UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5

Licence 2 informatique – 2016/2017

HLIN302 – Travaux Dirigés n° 1

Programmation impérative avancée

Alban MANCHERON et Pascal GIORGI

1 Passage de paramètres

Voici un code C++ définissant quatre fonctions qui échangent des valeurs et un programme principal :

```
1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4
5  void ech1(int a, int b) {
6    int c;
7    c = a; a = b; b = c;
8  }
9
10 void ech2(int *a, int *b) {
11   int c;
12   c = *a; *a = *b; *b = c;
13 }
14
15 void ech3(int &a, int &b) {
16   int c;
17   c = a; a = b; b = c;
18 }
19
20 void ech4(size_t u, size_t v, int *t){
21   int c;
22   c = t[u]; t[u] = t[v]; *(t + v) = c;
23 }
24
25 #define PrintVal(v)                    \
26   std::cout << "l" << __LINE__ << ": " \
27             << #v << " = " << v        \
28             << std::endl
29
```

```
30 int main(int argc, char** argv) {
31   int x[10];
32   size_t i, j, k;
33   srand(19);
34   j = rand() % 10;
35   k = rand() % 10;
36   for (i = 0; i < 10; i++) {
37     x[i] = rand() % 100 + 1;
38   }
39   PrintVal(j); PrintVal(k); PrintVal(x);
40   PrintVal(x[j]); PrintVal(x[k]);
41   std::cout << "=====" << std::endl;
42   ech1(x[j], x[k]);
43   PrintVal(x[j]); PrintVal(x[k]);
44   std::cout << "=====" << std::endl;
45   ech2(x+j, &x[k]);
46   PrintVal(x[j]); PrintVal(x[k]);
47   std::cout << "=====" << std::endl;
48   ech3(x[j], x[k]);
49   PrintVal(x[j]); PrintVal(x[k]);
50   std::cout << "=====" << std::endl;
51   ech4(j, k, x);
52   PrintVal(x[j]); PrintVal(x[k]);
53   std::cout << "=====" << std::endl
54             << "Fin du programme"
55             << std::endl;
56
57   return 0;
58 }
```

1. Ce fichier est-il syntaxiquement correct ? Dans le cas contraire, apportez les corrections nécessaires pour pouvoir le compiler.
2. Écrire – en justifiant – ce que l'exécution du programme issu de la compilation du fichier [1] affichera, sachant que les premières lignes affichées sont :

```
l39: j = 3
l39: k = 7
l39: x = 0x7fff500f5440
l40: x[j] = 81
l40: x[k] = 73
```

1. Après éventuelle correction.

2 Retours de fonctions

On considère maintenant le code suivant :

```
1  #include <iostream>
2
3  int min1(int a, int b) {
4    return (a < b ? a : b);
5  }
6
7  int min2(int &a, int &b) {
8    return (a < b ? a : b);
9  }
10
11 int &min3(int &a, int &b) {
12   return (a < b ? a : b);
13 }
14
15 #define PrintVal(v)                    \
16   std::cout << "l" << __LINE__ << ": " \
17             << #v << " = " << v        \
18             << std::endl
19
20 int main(int argc, char** argv) {
21   int x = 2, y = -5;
22   PrintVal(x); PrintVal(y);
23
24   PrintVal(min1(x, y));
25   PrintVal(min2(x, y));
26   PrintVal(min3(x, y));
27
28   PrintVal(min1(2, 3));
29   PrintVal(min2(2, 3));
30   PrintVal(min3(2, 3));
31
32   PrintVal(min1(0, min1(x, y)));
33   PrintVal(min2(0, min2(x, y)));
34   PrintVal(min3(0, min3(x, y)));
35
36   min1(x, y) = 10;
37   min2(x, y) = 10;
38   min3(x, y) = 10;
39
40   PrintVal(x); PrintVal(y);
41   return 0;
42 }
```

1. Ce fichier est-il syntaxiquement correct ? Dans le cas contraire, apportez les corrections nécessaires pour pouvoir le compiler.
2. Écrire – en justifiant – ce que l'exécution du programme issu de la compilation du fichier[2] affichera, sachant que les premières lignes affichées sont :

```
l66: x = 2
l66: y = -5
l68: min1(x, y) = -5
l69: min2(x, y) = -5
l70: min3(x, y) = -5
```

3 Algorithmique et programmation

1. Écrire le corps de la fonction `int Puissance(int x, unsigned int n)` permettant, en utilisant l'algorithme naïf, de calculer la n^{e} puissance de x.
2. Proposez une version récursive équivalente.
3. Quelle est, selon vous, la meilleure solution (vous justifierez votre avis) ?
4. Est-il possible d'améliorer encore ce calcul ? Le cas échéant, obtient-on toujours les mêmes résultats pour toutes les valeurs de x et de n ?

Pour rappel, $x^n = \begin{cases} 1 & n = 0 \\ \prod_{i=1}^{n} x \left(= \underbrace{x \times \cdots \times x}_{n \text{ fois}} \right) & n > 0 \end{cases}$

4 Bonnes pratiques et bons outils

Afin de mettre en pratique les exercices vus en TD, il est fondamental d'avoir une bonne méthodologie et d'utiliser une panoplie d'outils adaptés au développement de tout projet informatique.

2. Après éventuelle correction.

Nous proposons d'illustrer ces bonnes pratiques à travers la description d'outils reconnus pour leur efficacité, qu'il s'agisse du choix de l'éditeur, du processus de compilation, du *debugging* ou de la gestion de l'évolution des programmes.

4.1 Éditeur de code

De nombreux éditeurs de texte permettent d'implémenter vos codes. les plus basiques ne permettent que de saisir du texte, sans aide aucune (*e.g.*, bloc-notes), d'autres proposent pléthore de boutons et de menus afin de vous – paraît-il – faciliter le travail d'édition [3] (*e.g.*, `code::Blocks`).

Toute solution offre ses avantages et ses inconvénients. Les outils rudimentaires sont très légers et permettent d'éditer tout et n'importe quoi, les outils très élaborés sont plus ou moins maniables, lourds et opaques.

Parmi toutes les solutions, il existe un éditeur qui a l'énorme avantage d'être quasi-universel : *Emacs* (*cf* `http://www.gnu.org/software/emacs/` et `http://www.xemacs.org/`). Sa prise en main n'est pas immédiate et il nécessite un investissement personnel indéniable pour son apprentissage, cependant une fois que l'on sais s'en servir, il est difficile de s'en passer tant il est puissant, performant, adaptable, évolutif, ...

> *Trois éditeurs pour les mordus de terminaux texte,*
> *Sept pour les techniciens dans leurs bureaux sous terre,*
> *Neuf pour les étudiants destinés au trépas,*
> *Un éditeur pour le codeur furieux sur son* OS *favoris,*
> *Au pays d'**Emacs** où s'étendent les codes*
> *Un Éditeur pour les programmer tous*
> *Un Éditeur pour les formater*
> *Un Éditeur pour les débugger tous,*
> *Et dans les modes les lier*
> *Au pays d'**Emacs** où s'étendent les codes.*

4.2 Compilation automatisée

Sur votre ENT, vous trouverez un tutoriel sur le `Makefile`. Il s'agit non seulement d'un nom de fichier, mais également d'un mécanisme permettant d'automatiser « intelligemment » le processus de compilation d'un programme en fonction des dépendances entre les différents fichiers le composant.

Dans le cadre de la séance de TP, et en vous référant au tutoriel mis à votre disposition, il vous est demandé de :

- Préparer un fichier `Makefile` simple permettant de compiler les fichiers des 3 exercices précédents.
- Enrichir ce `Makefile` afin de procéder au nettoyage des fichiers binaires et de sauvegarde générés automatiquement (ajout d'une cible `clean`).
- Ajouter une cible permettant de créer une archive au format `tar.gz` ou `tar.bz2` de l'ensemble de votre programme. Il est demandé que cette archive intègre un *timestamp* [4] dans son nom.

4.3 Traces d'exécution

Vous l'aurez compris, le `Makefile` vous permet de compiler facilement, rapidement, sans risque d'erreurs (si le fichier est correct), ... Toutefois, il est fort probable que vos codes ne compileront pas du premier coup, et qu'une fois qu'ils compileront, il demeure des erreurs qu'il vous faudra débusquer.

L'outil `gdb` (*The GNU Project Debugger*, *cf* `http://www.gnu.org/software/gdb/`) vous permet d'explorer votre programme lorsqu'il est exécuté. L'exploration consiste à marquer des points d'arrêts, observer le contenu de la mémoire (tas et pile), ...

Pour pouvoir l'utiliser plus efficacement, il est recommandé de compiler vos programmes avec les options `-g` et `-O0` (*cf* documentation de votre compilateur préféré [`g++`]).

3. De type de solution s'appelle une IDE (*Integrated Development Environment* [Environnement de développement intégré]).
4. « Horodatage » en français.

Pour l'utiliser, il suffit d'exécuter `gdb` (en ligne de commande) avec comme argument le nom de votre programme (*e.g.*, `TD1-1`). Vous obtenez alors un message rempli d'informations que vous devriez lire au moins une fois et une invite de commande

```
prompt$ gdb TD1-1
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from TD1-1...done.
(gdb)
```

Plusieurs commandes sont disponibles, et le plus simple est de consulter la documentation en tapant `help`.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Si vous avez consulté l'aide en tapant `help running`, vous devriez comprendre la capture ci-dessous.

```
(gdb) run
Starting program: /home/user/HLIN302/TD1/TD1-1
Traceback (most recent call last):
  File "/usr/share/gdb/auto-load/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19-
  gdb.py", line 63, in <module>
    from libstdcxx.v6.printers import register_libstdcxx_printers
ImportError: No module named 'libstdcxx'
l39: j = 9
...
Fin du programme
[Inferior 1 (process 15953) exited normally]
(gdb)
```

Pour quitter, il suffit de taper `quit`.

Les commandes de base qui vous seront utiles très rapidement sont `help`, `run` et `quit` que nous venons de voir, mais également `breakpoint`, `print`, `watch`, `list`, `step`, `next`, `continue`, `backtrace` et `clear`.

Il vous est demandé de consulter l'aide de chacune de ces commandes et de les tester sur vos programmes correspondant aux exercices précédents.

Le second outil que vous devriez explorer s'appelle `valgrind` (*cf* http://valgrind.org/). Il permet notamment de détecter des fuites mémoires. N'hésitez pas à l'essayer, il vous fera gagner des heures de développement.

Il existe de nombreux autres outils vous permettant de corriger ou d'améliorer votre code, soyez curieux, lisez les documentations, les forums et testez ceux qui vous semblent prometteurs.

4.4 Gestion de versions et travail collaboratif

Deux principes sont à la base de tout travail en informatique : sauvegarder régulièrement et conserver plusieurs copies de ces sauvegardes.

Les sauvegardes régulières ont pour principal objectif de pouvoir revenir à une version spécifique de nos fichiers (*e.g.*, revenir avant la dernière modification d'un code qui fait que plus rien ne fonctionne, panne du système d'exploitation ou d'alimentation qui fait que tous les fichiers en cours d'édition sont corrompus).

L'intérêt de conserver plusieurs copies est de diminuer le risque de perte des sauvegardes. Il est par ailleurs recommandé que ces copies soient stockées sur différents supports, et idéalement à différent endroits géographiques.

Comme vous l'avez vu précédemment, le `Makefile` vous permet de faciliter les sauvegardes régulières, encore faut-il ne pas oublier de les générer. Pour ce qui est de conserver plusieurs copies, vous pouvez copier vos fichiers de sauvegarde sur des supports amovibles, sur un serveur de stockage distant, ...

Il y a toutefois deux inconvénients majeurs à procéder ainsi. Tout d'abord, cela nécessite beaucoup d'actions manuelles et que sur ce point, l'humain est moins fiable que la machine. Ensuite, un développeur consciencieux, travaillant sur un projet représentant 1MO de données, qui ferait ne serait-ce qu'une sauvegarde par jour sur deux supports différents utiliserait en 1 mois de travail (à raison de 5 jours travaillés par semaine) 22MO.

À l'échelle d'une entreprise cela n'est pas plus raisonnable que de ne pas faire de sauvegarde.

Plusieurs outils permettent à la fois d'automatiser les mécanismes de sauvegarde et des multiples copies (RCS, CVS, SVN, *Mercurial*, GIT, ...).

Ils reposent tous sur trois mêmes concepts :

1. ne conserver que les différences entre deux versions consécutives (sous forme de *patches*)ainsi que la dernière version des fichiers. Ainsi pour revenir à une précédente version, il suffit d'appliquer les *patches* de la version en cours jusqu'à la version désirée ;
2. permettre à plusieurs personnes de travailler sur les mêmes fichiers tout en garantissant que chacun n'écrase pas le travail de ses collaborateurs ;
3. s'assurer que les utilisateurs travaillent sur une copie du projet et non sur un original.

Un des outils récent qui se généralise de plus en plus est GIT (*cf* http://git-scm.com/).

La première chose à faire pour utiliser cet outil est de le personnaliser.

```
prompt$ git config --global user.name Édith Rice
prompt$ git config --global user.email edith.rice@lamer.com
prompt$ git config --global core.editor emacs
prompt$ git config --global alias.co checkout
prompt$ git config --global alias.ci commit
prompt$ cat ~/.gitconfig
```

La commande `git` dispose, à l'instar de `gdb`, d'une aide structurée et intégrée.

```
prompt$ git help
...
prompt$ git help help
...
prompt$ git help config
...
prompt$ git help init
...
```

Il suffit donc de vous placer dans votre répertoire de travail pour ce TP et d'exécuter :

```
prompt$ cd TP1
prompt$ git init
Dépôt Git vide initialisé dans /home/user/HLIN302/TD1/.git/
prompt$ git status
Sur la branche master

Validation initiale

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)

            Makefile
            TD1-1.cpp

aucune modification ajoutée à la validation mais des fichiers non suivis sont
présents (utilisez "git add" pour les suivre)
prompt$ git help add
...
prompt$ git add TD1-1.cpp Makefile
git add TD1-1.cpp Makefile
prompt$ git status
Sur la branche master

Validation initiale

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)

          nouveau fichier: Makefile
          nouveau fichier: TD1-1.cpp
prompt$ git ci -m "Mon premier 'commit'"
  [master (commit racine) 9afe8da] Mon premier 'commit'
 2 files changed, 58 insertions(+)
 create mode 100644 Makefile
 create mode 100644 TD1-1.cpp
prompt$ git status
Sur la branche master
rien à valider, la copie de travail est propre
prompt$
```

Lisez les documentations des commandes `git` suivantes, puis testez-les : `mv`, `rm`, `show`, `log`, `diff` et `tag`.

Pour plus d'informations sur l'utilisation de `git`, vous pouvez également suivre les tutoriels suivants :

- http://bioinfo-fr.net/git-premiers-pas
- http://bioinfo-fr.net/git-usage-collaboratif

Enfin, quand vous aurez appris à vous servir de `git`, sachez que le service informatique de la Faculté des Sciences propose un *GitLab* accessible à l'adresse : http://gitlab.info-ufr.univ-montp2.fr/