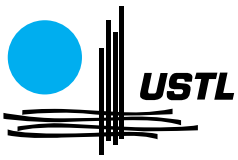

Licence d'informatique
Module de C/Unix

Premier TP de programmation en C

Philippe MARQUET

Révision de novembre 2002

Ce premier TP introductif à la programmation en C va vous familiariser avec :
- la syntaxe du langage ;
- la mise en œuvre du compilateur ;
- le préprocesseur `cpp` ;
- l'utilisation d'un « débogueur/dévermineur/metteur au point » ;
- la compilation séparée et l'utilisation de `make` ;
- divers outils relatifs au dévéloppemnt de programmes en C.

Quelques unes des questions du présent sujet de TP sont inspirées d'un TP proposé par Christian QUEINNEC sur son site VideoC, `http://videoc.lip6.fr`

Vous pouvez copier/coller les sources C et les commandes du présent sujet depuis la version en ligne accessible à `http://www.lifl.fr/~marquet/ens/cu/tpc1.html`

Exercice 1 (Factorielle)

Question 1.1 Écrivez un programme qui calcule la factorielle de 10, et affiche le résultat. On définira une fonction

```
unsigned int
factorielle (unsigned int n)
{
  unsigned int i = 1, res = 1;

  while (i  n)
   res = res * i++;
  return res;
}
```

qui calcule la factorielle de `n` de façon itérative. On placera le source de ce programme dans le fichier source `factorielle.c` ;

Question 1.2 Compilez ce programme (le fichier source) par la commande

```
% gcc -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
```

- on utilise le compilateur C du projet GNU, `gcc` ;
- les options `-ansi` et `-pedantic` précisent que l'on produit des programmes C au standard ANSI C et ce de manière stricte. **Ces options seront systématiquement utilisées pour tous vos développements**, au moins dans le cadre de ce module ;
- de même, vous utiliserez systématiquement les options `-Wall` (signaler tous les warnings) et `-Werror` (considérer les warnings comme des erreurs) ;
- l'option `-g` du compilateur C permet de conserver des informations utiles au dévermineur ;
- l'option `-o` permet de spécifier le nom du fichier exécutable, (`factorielle` ici, `a.out` par défaut).

Si il n'y a pas d'erreur, le compilateur n'affiche rien et génère l'exécutable `factorielle`. Sinon, le compilateur affiche la liste des erreurs qu'il a détectées. Par exemple

Id : tpc1.tex,v 1.8 2001/11/20 18 :00 :20 marquet Exp

```
% gcc -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
factorielle.c: In function 'factorielle':
factorielle.c:6: parse error before 'n'
cc1: warnings being treated as errors
factorielle.c: At top level:
factorielle.c:11: warning: return-type defaults to 'int'
```

indique que dans la fonction `factorielle` du fichier source `factorielle.c`, il y a une erreur. C'est une erreur de syntaxe à la ligne 6 (`while (i n)`) où il manque un opérateur entre `i` et `n`. La seconde erreur est une admonestation (warning) à la ligne 11 (`main()`), où le type retourné par la fonction `main` a été fixé par défaut à `int`. Lisez donc toujours bien les messages d'erreurs. Exécutez ensuite ce programme.

```
% factorielle
362880
```

Exercice 2 (Utilisation d'un dévermineur)
Un dévermineur (ou metteur au point) permet d'exécuter un programme en mode pas à pas et de visualiser l'ensemble de la mémoire du processus correspondant au programme en cours d'exécution. La plupart des processeurs possède un mode pas à pas dans lequel ils appellent automatiquement une routine après l'exécution de chaque instruction machine. Les dévermineurs utilisent ce mode pour exécuter un programme :

- instruction machine par instruction machine (`stepi`, `nexti`);
- ligne de code source par ligne de code source (`step`, `next`);
- point d'arrêt par point d'arrêt (`continue`).

C'est ce dernier mode que l'on utilise le plus fréquemment :

- on place un point d'arrêt (`break`) dans le code source à l'endroit où l'on suspecte que se trouve l'erreur ;
- on démarre l'exécution du programme (`run`) jusqu'à ce point d'arrêt ;
- puis on exécute pas à pas en examinant le comportement du programme.

On peut également examiner le contenu des variables (`print` et `display`) ou même la valeur d'une expression quelconque du langage C.

La pile des appels de fonction est aussi visible (`backtrace`), ainsi que les cadres d'appel de chaque fonction (`frame` et `info frame`).

Le dévermineur que vous allez utiliser est `gdb` (Gnu DeBugger) et plusieurs interfaces graphiques à `gdb` sont disponibles, dont `xxgdb` et `ddd`, ce dernier étant le moins austère.

Question 2.1

1. Placez un point d'arrêt avant l'appel à la fonction `factorielle()`.
2. Exécutez alors le programme en mode pas à pas (`step` pour entrer dans la fonction, `next` saute les appels de fonction).
3. Affichez les valeurs des variables `res` et `i` (`display`) : leurs valeurs sont quelconques car elles n'ont encore pas été initialisées. Les variables locales sont en effet allouées automatiquement dans la pile, qui à cet instant de l'exécution peut contenir n'importe quoi.
4. En continuant à exécuter en mode pas à pas, vous verrez les variables `res` et `i` évoluer au cours des itérations de la boucle.

Question 2.2 Relancez l'exécution du programme (`run`) avec le même point d'arrêt. Modifier la valeur de `n` dans la fonction `main()` :

```
(xxgdb) set n=12
```

et terminez son exécution (`cont`). Le programme a bien calculé la factorielle de 12, et non pas de 10. Recommencez avec `n=13`, et vérifiez que le résultat n'est pas égal à 13 fois le précédent. Les entiers de type `int` ne font bien que 32 bits sur cette machine.

Exercice 3 (Compilation conditionnelle et préprocesseur)
Le préprocesseur permet de sélectionner des zones de code à compiler ou à ne pas compiler selon différentes conditions. Il est ainsi possible de tester l'existence de macros avec les directives `#ifdef/#endif`, et d'inclure ou non le code compris entre ces deux directives. On peut aussi tester la valeur d'une macro avec la directive `#if/#endif`. Ceci s'avère très utile pour fournir des programmes pouvant être compilés sur différentes architectures. Par exemple :

```
#if ARCH = i386
/* code pour un intel i386 */
#endif
#if ARCH = alpha
/* code pour un DEC Alpha */
#endif
```

Question 3.1 Renommez la fonction `factorielle()` en `factorielle_iterative()`. Écrivez la fonction

```
unsigned int factorielle_recursive (unsigned int n);
```

qui calcule la factorielle de `n` de façon récursive. Placez les corps de ces fonctions dans une directive `#ifdef/#endif` de la manière suivante :

```
#ifdef RECURSIVE
unsigned int
factorielle_recursive (unsigned int n)
{
  ...
}

unsigned int
factorielle (unsigned int n)
{
  return factorielle_recursive (n);
}
#else
unsigned int
factorielle_iterative (unsigned int n)
{
...
}

unsigned int
factorielle(unsigned int n)
{
  return factorielle_iterative (n);
}
#endif
```

De cette manière, si la macro `RECURSIVE` est définie, seule la fonction `factorielle_recursive()` sera définie, et l'appel à la fonction `factorielle()` sera un appel à `factorielle_recursive()`. Sinon seule `factorielle_iterative()` est définie, et un appel à `factorielle` sera un appel à `factorielle_iterative()`.

Question 3.2 La macro `RECURSIVE` peut être soit définie dans le fichier source (`#define RECURSIVE`), soit définie au moment de la compilation grâce à l'option `-D nom=definition` du compilateur. Vérifiez le travail du préprocesseur `cpp` en demandant au compilateur de ne faire que la phase de préprocessing :

```
% gcc -E -P factorielle.c | grep -v '^[ ]*$' | more
```

Vous voyez apparaître les déclarations de types, variables et fonctions standard, puis celles contenues dans <stdio.h>, et enfin la définition de factorielle_iterative(), factorielle() et main(). Recommencez en définissant la macro RECURSIVE :

```
% gcc -E -P -D RECURSIVE factorielle.c | grep -v '^[ ]*$'
```

C'est bien maintenant la version récursive qui est définie.

Question 3.3 Compilez la version récursive pour étudier son comportement sous dévermineur (par exemple xxgdb, vous pouvez préférer ddd) :

```
% gcc -D RECURSIVE -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
% xxgdb factorielle
```

1. Placez un point d'arrêt avant le return de factorielle_recursive() et lancez l'exécution (run).
2. Puis avancez de quelques appels récursifs (step).
3. Vous pouvez examiner la pile des appels de fonctions avec la commande bt, et vous placer dans n'importe quel cadre d'appel par la commande frame n où n est le numéro du cadre choisi (à gauche dans l'affichage fait par bt). Par exemple,

```
(xxgdb) print n
$1 = 4
(xxgdb) frame 3
#3  0x804840d in factorielle_recursive (n=7) at factorielle.c:5
(xxgdb) print n
$2 = 7
```

Exercice 4 (Compilation séparée et utilisation de l'utilitaire `make`)
La commande make permet de déterminer automatiquement quelles parties d'un programme sont à recompiler et génère les commandes pour ce faire. Il est très utile pour la compilation séparée, où un programme est constitué de plusieurs fichiers sources.

Question 4.1 Séparez votre programme en deux fichiers sources factorielle.c et main.c. Le premier ne contient que les définitions des fonctions factorielle_iterative(), factorielle_recursive() et factorielle(). Le second ne contient que la fonction main(). Créez un fichier factorielle.h contenant la déclaration de la fonction factorielle:

```
extern unsigned int factorielle(unsigned int);
```

et incluez ce fichier dans main.c :

```
#include <stdio.h>
#include "factorielle.h"
...
```

Remarquez l'utilisation des doubles quotes pour indiquer au compilateur de chercher le fichier d'entêtes factorielle.h dans le répertoire courant en premier. Sans utiliser make, pour compiler le programme, il faut exécuter les commandes suivantes :

```
% gcc -c -Wall -Werror -ansi -pedantic factorielle.c
% gcc -c -Wall -Werror -ansi -pedantic main.c
% gcc -Wall -Werror -ansi -pedantic -o factorielle factorielle.o main.o
```

L'option -c de gcc permet de spécifier au compilateur de ne pas faire l'édition de liens, et de ne générer que les fichiers objets .o.

Question 4.2 Créez un fichier Makefile (ou makefile) avec le contenu suivant :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

factorielle: factorielle.o main.o
        $(CC) $(CFLAGS) -o factorielle factorielle.o main.o
factorielle.o: factorielle.c
        $(CC) -c $(CFLAGS) factorielle.c
main.o: main.c
         $(CC) -c $(CFLAGS) main.c
```

Les lignes correspondantes aux commandes à exécuter commencent par une *tabulation*. Lancez la compilation par la commande `make` (ou `make factorielle` si la première règle de dépendance ne concerne pas `factorielle`). Les trois compilations sont automatiquement exécutées. En modifiant la date du fichier `factorielle.c`, et en recompilant,

```
% touch factorielle.c
% make
```

seule la compilation de `factorielle.c` et l'édition de liens sont exécutées, le fichier `main.c` étant resté inchangé.

En fait, `make` connaît déjà un ensemble de règles de dépendance par défaut, et seule la première dépendance suffit. L'option `-p` de `make` liste la base de données (variables et règles de dépendance) utilisée. Pour vous en convaincre, essayez de supprimer les 4 dernières lignes de `Makefile`.

Dans un `Makefile`, on place aussi une entrée `clean` pour faire le ménage des fichiers intermédiaire et une entrée `realclean` qui ne conserve que le strict nécessaire. Ces entrées ne correspondent pas à des fichiers à construire, on le précise à `make` par la directive `.PHONY` :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

factorielle: factorielle.o main.o
        $(CC) $(CFLAGS) -o factorielle factorielle.o main.o

.PHONY: clean realclean
clean:
        $(RM) factorielle.o main.o
realclean : clean
        $(RM) factorielle
```

Partisan du moindre effort (et de la moindre erreur), on peut rassembler l'ensemble des fichiers `.o` dans une variable `OBJ` et utiliser la variable automatique `$@` :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

OBJ     = factorielle.o main.o

factorielle: $(OBJ)
        $(CC) $(CFLAGS) -o $@ $(OBJ)

.PHONY: clean realclean
clean:
```

```
        $(RM) $(OBJ)
realclean : clean
        $(RM) factorielle
```

On termine alors la séance de TP par :

```
% make realclean
rm -f factorielle.o main.o
rm -f factorielle
```

Exercice 5 (Autres outils)
Devant les difficultés et pièges du langage C, d'autres outils sont aussi a considérer :

- `lint`, dont une version moderne est LCLint (installé sur le réseau enseignement et disponible à `http://lclint.cs.virginia.edu`), fournit nombre d'avertissements sur votre code C.
  Essayez la commande `lclint` sur les sources écrits lors de ce TP.
- `gprof` (GNU profiler) analyse les performances d'une exécution d'un programme. Le programme C doit avoir été instrumenté lors de la compilation (option `-pg` du compilateur). L'exécution du programme produira un fichier `gmon.out`. L'exploitation de ce fichier par `gprof` permet de visualiser l'arbre des appels des fonctions du programme, le pourcentage de temps CPU passé dans chacune des fonctions, etc.
- `nm` liste les symboles présents dans un fichier objet (`.o`) ou un exécutable. Il est ainsi possible d'identifier les variables définies et exportées, les fonctions définies, les fonctions utilisées sans être définies... dans un fichier objet.

Pour terminer

Nous vous demandons (et exigeons pour les projets) de toujours utiliser les options de compilation `-Wall` (signaler tous les warnings), `-Werror` (considérer les warnings comme des erreurs), `-ansi` (le source est du C-ANSI), et `-pedantic` (et seulement du C-ANSI).

Nous exigeons également que vous rendiez vos projets avec un `Makefile`, le correcteur ne doit avoir qu'à exécuter `make` !