SOMMAIRE

Introduction.....	3
Rappels.....	4
Environnement De Programmation.....	10
Gérer les arguments.....	17
Programmation Système.....	20
De la couleur	22
Annexe 1 : Les Langages de Script.....	24
Annexe 2 : Java.....	27
Annexe 3 : Glossaire.....	28

© Copyright 2010 tv <thierry.vaira@orange.fr>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License,

Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover.

You can obtain a copy of the GNU General Public License : write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

INTRODUCTION

Objectifs

Être capable d'éditer un fichier texte et de fabriquer un exécutable simple.

Découvrir l'environnement de programmation sous GNU/Linux.

Estimer les besoins en programmation d'un administrateur.

Manipulations

On va travailler dans l'arborescence suivante :

```
$HOME
|
- tv
  |
  - TPOS9
```

Créer l'arborescence de répertoires :

```
$ mkdir -p $HOME/tv/TPOS9
```

Se déplacer dans l'arborescence de travail :

```
$ cd $HOME/tv/TPOS9
```

RAPPELS

La programmation

La programmation dans le domaine informatique est l'ensemble des activités qui permettent l'écriture des programmes informatiques. C'est une étape importante de la conception de logiciel.

La programmation représente usuellement le codage, c'est-à-dire la rédaction du code source d'un logiciel. On utilise plutôt le terme développement pour dénoter l'ensemble des activités lié à la création d'un logiciel.

Pour écrire des programmes informatiques, on utilise un langage de programmation.

Langage de programmation

Les langages de programmation permettent de définir les ensembles d'instructions effectuées par l'ordinateur lors de l'exécution d'un programme. Il existe des milliers de langages de programmation, la plupart d'entre eux étant réservés à des domaines spécialisés. Ils font l'objet de recherches constantes dans les universités et dans l'industrie.

Environnement de Développement Intégré

Les programmeurs utilisent aussi des EDI (Environnement de Développement Intégré, IDE en anglais) qui leur permettent d'avoir dans une même application tous les composants et outils pour développer des programmes (éditeur de texte, compilateur, outil automatique de fabrication, explorateur, débogueur ou *debugger*, documentation, etc ...).

Un environnement de développement intégré (ou IDE pour *Integrated Development Environment*) est donc un programme regroupant un ensemble d'outils pour le développement de logiciels. On peut également trouver dans un EDI un système de gestion de versions et différents outils pour faciliter la création de l'interface graphique (GUI pour Graphical User Interface) ou IHM (Interface Homme Machine).

Les EDI les plus connus sont : Eclipse, NetBeans, Microsoft Visual Studio, Builder de Borland, Kdevelop, Qt Creator, WinDev, glade, Code::Blocks, ...

Atelier de génie logiciel

On désigne par atelier de génie logiciel (AGL) un ensemble de programmes informatiques permettant eux-mêmes de produire des programmes de manière industrielle.

Alors qu'un compilateur, par exemple, ne peut que traduire un langage informatique vers un autre, la réalisation de programmes, notamment au sein d'équipes nombreuses, requiert bien d'autres activités qui sont généralement couvertes par un AGL.

Fichier source

Les fichiers sources (appelés aussi code source ou tout simplement le code) sont des fichiers textes ASCII. Pour les créer, il suffit donc d'avoir un éditeur de texte (un traitement de texte ne convient pas). L'extension `.c` précisera que c'est un fichier contenant du code source en langage C.

L'éditeur de texte est l'outil indispensable d'un programmeur (et aussi d'un administrateur) : il conviendra de le choisir convenablement bien !

En attendant, les éditeurs de texte sont nombreux :

- `vi` : l'éditeur « historique » des système Unix (fonctionne en mode console), c'est l'un des éditeurs les plus puissants et on a la garantie de le trouver sur tous les OS de type UNIX
- `vim` : Vi Improved (très puissant)
- `emacs` : c'est l'un des éditeurs (avec `vi`) les plus répandus et les plus appréciés des programmeurs (très puissant)
- `kwrite`, `kate`, etc ... : éditeurs graphiques généralistes de l'environnement KDE (très facile)
- le bloc-notes : éditeur de l'environnement Windows (© Microsoft)
- etc ...

Remarque : vim et emacs sont les éditeurs privilégiés des programmeurs et administrateurs. Les autres ne sont que d'aimables plaisanteries !

Il va de soi que le code source n'est pas compréhensible tel quel par le système d'exploitation de l'ordinateur.

Pour exécuter un programme à partir de son code source, il n'y a que deux solutions :

- disposer d'un autre programme, nommé interpréteur, capable de lire le code source et d'effectuer les opérations décrites dans le code source ;
- disposer d'un autre programme, nommé compilateur, capable de traduire le code source en langage binaire, qui sera alors directement exécutable par l'ordinateur.

Les langages compilés et interprétés

Les différents langages informatiques peuvent donc être scindés en deux groupes : les langages compilés et les langages interprétés.

Les programmes interprétés sont fatalement relativement lents, puisque l'interpréteur doit analyser en permanence les fichiers sources pour le traduire en opération à exécuter à la volée. De nos jours, il existe une variante qui est la pré-compilation par l'interpréteur avant exécution.

En revanche, les programmes compilés sont beaucoup plus rapides à l'exécution, puisque la phase d'analyse du code source a été réalisée au préalable et se fait en dehors de la phase d'exécution. Ces programmes peuvent donc s'exécuter nativement, sans être ralenti par l'interpréteur.

Les langages interprétés

Du point de vue du programmeur, les avantages et inconvénients d'un langage interprété peuvent être résumés ainsi :

- Le code source fraîchement écrit peut être testé immédiatement ; il suffit de lancer l'exécution en fournissant le nouveau fichier source à l'interpréteur.
- Pour que le fichier source puisse être utilisé, il faut que l'interpréteur soit également présent. La portabilité est assurée sur une machine différente si le même interpréteur y est disponible.
- Le fichier à transporter est un code source de type texte, généralement petit (quelques kilo-octets). En revanche, la mémoire nécessaire pour l'exécution est plus importante que dans le cas d'un fichier compilé, car l'interpréteur doit aussi y être présent.
- L'exécution du programme nécessite que l'interpréteur analyse chaque instruction, et la traduise. Le processus est donc plus lent.
- Le fichier exécutable est naturellement lisible et modifiable par tout utilisateur. Si le programme a une bonne documentation interne, il se suffit parfaitement. C'est un gros avantage dans l'esprit des logiciels libres.

Remarque : certains interpréteurs modernes (Perl, Python, PHP, etc.) procèdent à une phase de compilation avant de lancer le programme pour permettre une optimisation et donc des gains en rapidité. De toute manière, l'utilisateur/programmeur ne verra pas de phase intermédiaire entre la rédaction du code et son exécution.

Vous trouverez en Annexe 1 une liste des langages de scripts les plus populaires.

Les langages compilés

Un compilateur est un programme informatique qui traduit un langage, le langage source, en un autre, appelé le langage cible. En pratique, un compilateur sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage d'assemblage ou un langage machine. Le compilateur n'est qu'un maillon de la chaîne qui permet de fabriquer un logiciel. En fin de chaîne de fabrication, on obtiendra un code binaire exécutable par une machine.

Les langages compilés les plus connus sont le langage C et le langage C++ (qui permet la programmation objet).

Remarque : pour comprendre le cas « Java », lire l'annexe 2.

Automatisation de tâches

Les administrateurs (ou parfois les programmeurs) ont souvent le besoin d'automatiser des traitements (surveillance, sauvegarde, maintenance, installation, ...). Pour cela, ils peuvent :

- faire des groupement de commandes à partir d'un shell : fréquent
- créer des scripts (avec le shell ou avec un autre langage de script comme Perl, Python, PHP, etc ...) : très fréquent
- créer des programmes utilisant l'API de l'OS : moins fréquent

L'automatisation des tâches est un domaine dans lequel les scripts shell prennent toute leur importance. Il peut s'agir de préparer des travaux que l'on voudra exécuter ultérieurement grâce à un système de programmation horaire (*crontab*) mais on utilise aussi les scripts pour simplifier l'utilisation de logiciels complexes ou écrire de véritables petites applications.

Les scripts sont aussi très utiles lorsqu'il s'agit de faire coopérer plusieurs utilitaires système pour réaliser une tâche complète. On peut ainsi écrire un script qui parcourt le disque à la recherche des fichiers modifiés depuis moins d'une semaine, en stocke le nom dans un fichier, puis prépare une archive tar contenant les données modifiées, les sauvegarde sur une bande, puis envoie un e-mail à l'administrateur pour rendre compte de son action. Ce genre de programme est couramment employé pour automatiser les tâches administratives répétitives.

Les shells proposent un véritable langage de programmation, comprenant toutes les structures de contrôle, les tests et les opérateurs arithmétiques nécessaires pour réaliser de petites applications. En revanche, il faut être conscient que les shells n'offrent qu'une bibliothèque de routines internes très limitée. Nous ferons alors fréquemment appel à des utilitaires système externes.

En résumé, les scripts sont utilisés surtout pour :

- automatiser les tâches administratives répétitives
- simplifier l'utilisation de logiciels complexes
- mémoriser la syntaxe de commandes à options nombreuses et utiliser rarement
- réaliser des traitements simples

Les shell scripts

Un *shell* script est :

- un fichier texte ASCII (on utilisera un éditeur de texte)
- exécutable par l'OS (le droit x sous Unix/Linux)
- interprété par un shell (/bin/bash par exemple)

Remarque : on a l'habitude de mettre l'extension .sh au fichier script.

Exemple d'un shell script :

```
$ ls -l script.sh
-rwxr-xr-x 1 tv tv 38 2010-08-05 11:22 script.sh

$ cat script.sh
#!/bin/bash
echo "je suis un script"
```

Remarque : un script doit commencer par une ligne "shebang" qui contient les caractères '#!' (les deux premiers caractères du fichier) suivi par l'exécutable de l'interpréteur. Cette première ligne permet d'indiquer le chemin (de préférence en absolu) de l'interpréteur utilisé pour exécuter ce script.

Avec un script, il est possible :

- d'exécuter et/ou grouper des commandes
- d'utiliser des variables prédéfinies (\$?, ...), des variables d'environnement (\$HOME, \$USER, ...)
- de gérer ses propres variables
- de faire des traitements conditionnels (if, case) et/ou itératifs (while, for)

ENVIRONNEMENT DE PROGRAMMATION

On vous propose d'utiliser quelques utilitaires simples disponibles sous GNU/Linux.

Programmer avec vim

La configuration de votre éditeur vim est réalisée à partir d'un fichier de configuration vimrc :

```
$ cat /etc/vim/vimrc
```

Vous pouvez ajouter ces lignes dédiées à la programmation avec vim :

```
set sw=3
set ts=3
set si
set et

" Pour eviter le copier/coller avec indentation en cascade
" noautoindent, cindent, smartindent, indentexpr
" setl noai nocin nosi inde=
" Avant de faire un coller de code, taper sur F8
nnoremap <F8> :setl noai nocin nosi inde=<CR>
" Pour revenir en mode autoindentation, taper sur F9
nnoremap <F9> :setl noai! nocin! nosi! inde=<CR>
```

1) Rechercher et donner la signification des options sw et ts de vim.

Site : <http://www.vim.org/>

Liens :

- <http://fr.wikipedia.org/wiki/Vim>,
- http://vim.wikia.com/wiki/Best_Vim_Tips
- http://www.swaroopch.com/notes/Vim_fr

Créer le fichier source hello.c suivant :

```
//Programme hello.c
/*
   auteur : tv
*/
#include <stdio.h>

int main (int argc, char* argv[])
{
    printf("Hello World !\n\n")
    return 0
}
```

Remarque : les deux ';' ont été volontairement retirés !

Compiler ce code source :

```
$ gcc -Wall hello.c -o hello >& .erreurs
```

Éditer avec vim le fichier des erreurs générés par gcc :

```
$ vim -q .erreurs
```

Il existe un certains nombres de commandes pratiques pour faire du développement avec vim :

```
:help Q_qf
:cc [nr] display error [nr] (default is the same again)
:cn[ext] display the next error
:cp[revious] display the previous error
:cl[ist] list all errors
:cf[ile] read errors from the file 'errorfile'
:cq[uit] quit without writing and return error code (to the compiler)
:make [args] start make, read errors, and jump to first error
:gr[ep] [args] execute 'grep' to find matches and jump to the first
one.
:sh[ell] start a shell
:set nu affiche les numéros de ligne
:![command] execute {command} with a shell
```

Par exemple, sans sortir de l'éditeur vim, on peut faire :

```
:!gcc -Wall hello.c -o hello
:!/hello
```

Remarque : la commande ctags sera plus tard aussi une aide précieuse.

Indenter le code source

Il est très important d'avoir un code source parfaitement lisible tout en respectant des règles d'écriture définies dans un projet, une équipe, une entreprise. Pour cela, on peut utiliser la commande `indent` (ou `bcpp` ou l'outil fourni dans son EDI) pour passer son code source à la « moulinette ».

Un script d'indentation automatique :

```
#!/bin/bash
# indent.sh
# Style: standard de codage BTS IRIS (by tv)

if [ $# -eq "0" ]
then
  echo "Usage: $(basename $0) <single_input_file> [-o <output_file>]";
  echo "Usage: $(basename $0) <input_files>"; echo
  exit 1
fi
INDENT="indent"
OPTIONS="-nbad -bap -nbc -bbo -bl -bli0 -bls -ncdb -nce -cp1 -di3 -ndj
-nfc1 -nfca -hnl -i3 -ts3 -npsl -lp -npcs -nprs -saf -sai -saw -nsc
-nsob"
$INDENT $OPTIONS $@

#####
# Explications: man indent #
#####
# Styles pré-définis dans indent #
#####
# indent -gnu (the GNU style) options are : #
# -nbad -bap -nbc -bbo -bl -bli2 -bls -ncdb -nce -cp1 -cs -di2 #
# -ndj -nfc1 -nfca -hnl -i2 -ip5 -lp -pcs -nprs -psl -saf -sai #
# -saw -nsc -nsob #
# indent -kr (The Kernighan & Ritchie style) #
# indent -orig (The style of the original Berkeley) #
#####
```

Remarque : le script est disponible sur le serveur.

Il existe aussi le programme `bcpp` (*beautififier cpp*) qu'on utilisera plutôt avec du code `c++`. Lien : <http://invisible-island.net/bcpp/>

Un exemple de code mal « écrit » :

```
#include <stdio.h>

#define SIX 1 + 5
#define NEUF 8 + 1

int main(){printf("ce qu'on obtient si on multiplie %d par %d: %d\n ",
SIX, NEUF,
        SIX * NEUF);return 0;}
```

On le passe à la « moulinette » :

```
$ ./indent.sh life.c

$ cat life.c
#include <stdio.h>

#define SIX 1 + 5
#define NEUF 8 + 1

int main()
{
    printf("ce qu'on obtient si on multiplie %d par %d: %d\n ", SIX,
NEUF,
        SIX * NEUF);
    return 0;
}
```

Compiler et tester ...

```
$ gcc life.c

$ ./a.out
ce qu'on obtient si on multiplie 6 par 9: 42
```

2) Pourquoi obtient-on ce résultat surprenant ?

Remarque : 42 ! Si ce chiffre n'active pas immédiatement et simultanément tous les synapses de votre cerveau, c'est que vous n'avez jamais entendu parler de "h2g2" et de son auteur Douglas Adams. Car 42 est le chiffre le plus important de l'Univers. C'est tout simplement la réponse à l'Ultime Question de la Vie, l'Univers et Tout Le Reste. Euh, quelle question déjà? Bien, justement, c'est là l'énigme. Nous connaissons la Réponse à l'Ultime Question, celle qui nous permettra d'appréhender enfin la raison d'être et le mode de fonctionnement du monde, mais nous ne connaissons pas cette sacrée Question ! Vous pouvez vérifier en recherchant "the answer to life, the universe and everything" sur google.

Fabrication automatique

L'application est décomposée en plusieurs modules ou fichiers afin d'assurer une programmation modulaire plus compréhensible et une maintenance plus facile.

Les besoins sont :

- une compilation séparée automatisée
- une optimisation du temps et des ressources : ne (re-)compiler que le nécessaire
- une mémorisation des règles de compilation de l'ensemble de l'application

Solution : l'outil `make`

Make est un outil indispensable aux développeurs :

- il assure la compilation séparée automatisée
- il permet de ne recompiler que le code modifié
- il utilise un fichier distinct contenant les règles de fabrication (Makefile)
- il permet d'utiliser des commandes shell (ce qui permet d'assurer des tâches de nettoyage, d'installation, d'archivage, etc ...)
- il utilise des macro-commandes et des variables

Le fichier `Makefile` contient la description des opérations nécessaires pour générer une application :

- les dépendances
- les règles à appliquer lorsque les dépendances ne sont plus respectées

Remarque : Le fichier `Makefile` sera donc lu et exécuté par la commande `make`.

Une **règle** est une suite d'instructions qui seront exécutées pour construire une cible, si la cible n'existe pas ou si des dépendances sont plus récentes. La syntaxe d'une règle est la suivante :

```
cible: dépendances
    commandes
# attention: les commandes sont précédées d'une tabulation
```

La **cible** est généralement :

- le nom d'un fichier qui va être généré par les commandes qui vont suivre, ou
- une action gérée par ces mêmes commandes (par exemple clean, install, etc ...)

Les **dépendances** sont les fichiers ou les règles nécessaires à la création de la cible. Par exemple un fichier en-tête ou un fichier source dans le cas d'une compilation C. Dans le cas d'un fichier, la cible n'est construite que si ce fichier est plus récent que la cible.

Commandes est une suite de commandes shell qui seront exécutées au moment de la création de la cible.

Un fichier Makefile « type » :

```

CC = gcc
RM = rm

# le nom de l'application
TARGET = myApp
# ajouter les autres .o
OBJS = $(TARGET).o
# les .h sont ici dans le répertoire courant
INCLUDES = -I.
# minimum conseillé
WARN = -Wall -Wstrict-prototypes
# les options de compilation
CFLAGS = $(INCLUDES) $(WARN)
# indiquer les librairies x utilisées avec -lx
LIBS =

all:    $(TARGET)

$(TARGET):    $(OBJS)
              $(CC) $(CFLAGS) $(LIBS) $^ -o $@

%.o:    %.c
              $(CC) $(CFLAGS) -o $@ -c $<
              # ou: $(CC) $(CFLAGS) -c $*.c -o $@

clean:
              $(RM) -f *.o

cleanall:
              $(RM) -f *.o $(TARGET)

dep:
              $(MAKEDEP) $(CFLAGS) -Y -s "# Dépendances by 'make dep'" *.c
2>/dev/null
              rm -f Makefile.bak

```

Remarque : pour tester, il vous préalablement un fichier myApp.c

```
$ cp hello.c myApp.c
```

Quelques utilisations possibles à partir de ce Makefile :

```

$ make dep
$ make          : ou make all
$ make clean
$ make cleanall

```

GÉRER LES ARGUMENTS

Un paramètre (ou argument) est au sens large un élément d'information à prendre en compte pour prendre une décision ou pour effectuer un calcul.

Un paramètre est une donnée manipulée par une section de code (programme, shell script, sous-programme, fonction, méthode) et connue du code appelant cette section.

On distingue deux types de paramètres :

- les paramètres d'entrée : donnée fournie par le code appelant au code appelé ;
- les paramètres de sortie : donnée fournie par le code appelé au code appelant .

Un paramètre d'entrée peut être transmis de deux façons :

- passage par copie (aussi appelé par valeur) : le code appelé dispose d'une copie de la valeur. Il peut la modifier, l'information initiale dans le code appelant n'est pas affectée par ces modifications.
- passage par adresse (aussi appelé par référence) : le code appelé dispose d'une information lui permettant d'accéder en mémoire à la valeur que le code appelant cherche à lui transmettre. Il peut alors modifier cette valeur là où elle se trouve, le code appelant aura accès aux modifications faites sur la valeur. Dans ce cas, le paramètre peut aussi être utilisé comme un paramètre de sortie.

Dans certains langages de programmation, il n'est pas possible de définir de paramètres de sortie autrement qu'en utilisant un paramètre d'entrée par son adresse. D'autres langages de programmation (comme le Pascal par exemple) obligent le programmeur à spécifier explicitement si un paramètre est d'entrée, de sortie ou d'entrée-sortie.

Les shell scripts

En programmation shell sous Unix/Linux, on parle de paramètre positionnel. C'est une variable dont le nom est un numéro.

Par exemple, le premier paramètre d'un script s'appelle \$1, le deuxième \$2 et ainsi de suite.

On utilisera les variables prédéfinies suivantes :

```

$# : le nombre de paramètres (arguments)
$* : tous les paramètres à partir de $1 séparés par des espaces
$0 : le nom du script
$1 à $9 : les neuf premiers arguments

```

Remarque : pour accéder aux paramètres suivant, il faut utiliser la commande shift qui décale d'un rang vers la gauche (\$2 dans \$1, ... et \$9 prend la valeur du dixième paramètre)

Créer le script testArg.sh avec vim :

```

#!/bin/bash
echo "nb d'arguments = $#"
```

```

echo "nom du script \"$0\" : $0"
```

```

test -n "$1" && echo "\$1 = $1" || exit 1
```

```

test -n "$2" && echo "\$2 = $2"
```

```

test -n "$1" && echo "les arguments = $@"
```

```

exit 0

```

Ajouter le droit exécution au script :

```

$ chmod u+x ./testArg.sh

```

Tester le script :

```

$ ./testArg.sh bts 2010

```

Le langage C

En programmation C, on parle de paramètre effectif. Il s'agit de la variable (ou valeur) fournie lors de l'appel du programme (ou d'un sous-programme).

Lorsqu'on saisit des arguments après le nom d'un programme, ceux-ci sont passés en paramètres de la fonction principal **main**. La fonction main d'un programme C :

```
int main(int argc, char *argv[]) { ... }
```

Le paramètre `argc` contient le nombre d'arguments (%d pour l'afficher).

Le paramètre `argv` contient l'ensemble des arguments sous la forme de chaînes de caractère : `argv[0]` contient le nom du programme, `argv[1]` contient le premier argument, etc ... (%s pour afficher un `argv[x]`).

Créer le fichier *testArg.c* avec *vim* :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf("nb d'arguments = %d\n", argc);
    printf("nom du programme argv[0] : %s\n", argv[0]);

    for(i=0;i<argc;i++)
        printf("argv[%d] = %s\n", i, argv[i]);

    if (argc < 2)
        return 1;
    return 0;
}
```

Compiler le programme :

```
$ gcc testArg.c -o testArg
```

Tester le programme :

```
$ ./testArg bts 2010
```

3) Quelle est la principale différence constatée entre les deux programmes (shell script et langage C) ?

PROGRAMMATION SYSTÈME

A partir d'un programme en C, il est possible d'exécuter des commandes de plusieurs manières avec :

- l'appel `system`
- les appels `exec`
- les appels systèmes (*system calls*) disponible dans l'API de l'OS (`mkdir` par exemple)

Remarques : les commandes et API system calls correspondent directement à des services offerts par le système d'exploitation. On peut donc les exploiter soit en utilisant la commande correspondante soit par programmation en C :

```
$ man 1 mkdir : la commande mkdir dans le chapitre 1 (les commandes)
$ man 2 mkdir : l'appel système mkdir() en C dans le chapitre 2
ou dans konqueror, taper #mkdir ou man:/mkdir
```

Exemple 1 : l'appel `system()`

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

int main()
{
    printf("Résultat de la commande ps -l :\n");
    system("ps -l");
    printf("Fin\n");
    return 0;
}
```

Exemple 2 : l'appel `execlp()`

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

int main()
{
    printf("Résultat de la commande ps -l :\n");
    execlp("ps", "ps", "-l", NULL);
    printf("Fin\n"); // je ne m'affiche pas, pourquoi ? cf. man execlp
}
```

Remarque : la commande ps permet de lister les processus (un équivalent du Ctrl-Alt-Suppr ou taskmgr.exe sous Windows)

4) Tester les deux programmes précédents et commenter les résultats (arbre des processus).**Exemple 3 : API System Calls**

Écrire un programme testMkdir.c qui utilise l'appel système mkdir() :

```
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    int result;
    mode_t mode = 0750;

    if(argc != 2)
    {
        printf("Usage: %s nom_repertoire\n", argv[0]);
        return 1;
    }
    result = mkdir(argv[1], mode);
    return result;
}

$ gcc testMkdir.c -o testMkdir
```

5) Tester le programme et commenter les résultats obtenus pour les cas ci-dessous :

```
$ ./testMkdir
Usage: ./testMkdir nom_repertoire
$ echo $?
1
```

```
$ ./testMkdir test
$ echo $?
0
$ ls -l
drwxr-x--- test/
```

```
$ ./testMkdir test
$ echo $?
255
```

DE LA COULEUR ...

Pour afficher du texte, on peut utiliser les commandes `echo` ou `printf` :

```
#!/bin/bash
((VALEUR=1))
CHAINE="world"
echo -en "\tUne valeur : $VALEUR\n"
echo -e "\tUn message : hello $CHAINE !!!"
printf "\tUne valeur : %02d\n" $VALEUR
printf "\tUn message : hello %s !!!\n" $CHAINE
```

On peut encore améliorer le résultat en ajoutant de la couleur !!!

Pour cela il faut passer des séquences d'échappement au terminal. Il faut savoir que cette technique de commande existe aussi pour d'autres périphériques (notamment les imprimantes).

Le caractère **ESC** (la touche Echap) a pour valeur ASCII 27 ou 0x1B en hexadécimal ou encore 033 en octal (man ascii). La séquence d'échappement qui permet de mettre du texte en couleur a la structure suivante :

```
ESC[attribut;foreground;backgroundm
```

Exemples :

```
echo -ne "\x1B[1;31;49m"
echo -n "Hello World !!!" : # un texte en couleur
echo -e "\033[0m" : # penser à revenir au mode normal
```

Toutes les possibilités :

```
#!/bin/sh
# Nico Golde <nico(at)ngolde.de> Homepage: http://www.ngolde.de
for attr in 0 1 4 5 7 ; do
    echo
    "-----"
    printf "\t\t- ESC[%s;Foreground;Background -\n" $attr
    for fore in 30 31 32 33 34 35 36 37; do
        for back in 40 41 42 43 44 45 46 47; do
            printf '\033[%s;%s;%sm %02s;%02s ' $attr $fore
        done
    done
    printf '\033[0m\n'
done
done
```

La séquence d'échappement permettent aussi de positionner du texte de la structure suivante :

ESC[positionG

Exemple like-linux :

```
#!/bin/bash
NAME=$(basename $0)
echo -n "Execution du script $NAME"
echo -ne "\x1B[60G"
echo -n "[  " ; echo -ne "\x1B[1;32;49m" ; echo -n "OK" ; echo -ne
"\033[0m" ; echo "  ]"
```

ANNEXE 1 : LES LANGAGES DE SCRIPT

Perl

Le langage Perl a été créé par Larry Wall durant l'année 1988 comme une généralisation d'un logiciel sur lequel il s'appuyait pour extraire des données statistiques à partir de fichiers système. Il s'inspira des concepts du shell Bourne, de sed, de awk, du langage C, ainsi que d'autres sources, pour créer un langage puissant, simple et très portable.

L'implication de la communauté des programmeurs Perl créa une dynamique très importante autour de ce langage qui continue toujours à évoluer. Perl est un langage où les scripts subissent une première phase de compilation avant l'exécution, ce qui améliore sensiblement son efficacité. De plus, il est apparu dans le monde des logiciels libres à un moment où les performances des ordinateurs grand public s'étaient suffisamment accrues pour rendre crédible l'emploi d'un langage interprété pour des tâches non triviales. On le rencontre donc dans des projets scientifiques et industriels de pointe, y compris pour des opérations système de bas niveau.

Un autre domaine particulièrement prisé est celui de la programmation Web où Perl met toute sa puissance à la disposition des scripts CGI, tout en incluant les éléments de sécurité indispensables à toute application Internet (mais si difficiles à garantir dans d'autres langages comme le C).

Tous ces éléments font que Perl est sans doute le langage phare de la programmation par scripts, celui qui lui a apporté crédibilité et large diffusion.

Tcl

Tcl (*Tool Command Language*) est un langage créé à l'université de Berkeley par John Ousterhout, à peu près en même temps que Perl. Il poursuit le développement du langage, ainsi que celui de la bibliothèque graphique Tk qui lui est associée en travaillant chez Sun. Après une dizaine d'années de développement, le langage fut placé sous le contrôle du groupe Tcl Consortium, tandis que son auteur original fonda une société nommée Scriptics, assurant des prestations liées aux développements de logiciels interprétés.

Le langage Tcl est relativement simple d'accès, et extensible par l'intermédiaire de bibliothèques externes (Tk étant la plus connue). La bibliothèque de fonctions disponibles avec Tcl est moins importante que celle qui est livrée avec Perl, mais on peut aisément l'interfacer avec d'autres applications, ce qui en fait un langage intéressant pour automatiser des tâches administratives.

Avec la bibliothèque Tk, développée également par John Ousterhout, on peut facilement développer des applications graphiques pour X-Window. Sa conception est simple, l'aspect de l'interface respecte les standards industriels habituels (look and feel de type Motif), et la programmation est bien plus simple qu'avec les bibliothèques employées généralement dans les programmes C (Xlib, Xt, Xm). Bon nombre de développeurs ont ainsi créé des applications dont le noyau central était écrit en C, mais dont l'interface reposait sur un script Tcl, pour pouvoir disposer de la bibliothèque Tk.

On peut donc dire que Tk est pour une bonne part dans le succès de Tcl ; la preuve en est que cette bibliothèque a été adaptée pour être disponible dans d'autres langages de scripts. On la rencontre ainsi fréquemment avec Perl ou Python.

Python

Python est un langage de script créé en 1990 par le hollandais Guido Van Rossum, dont la dénomination se présente comme un hommage au fameux « Monty Python's Flying Circus ». Très extensible, ce langage présente des aspects plutôt rares comme l'indentation significative, ou une orientation objet complète. On retrouve ainsi quelques caractéristiques qui proviennent du langage Modula (version objet du Pascal) et du C.

Python est apprécié par de nombreux programmeurs, principalement pour sa facilité d'emploi lors de la mise au point des applications. La puissante gestion des exceptions facilite le développement, avec l'assistance d'outils de débogage intégrés.

Ruby

L'apparition du langage Ruby sur les systèmes Unix est relativement récente. Il s'agit d'un langage de scripts issu d'une conception théorique plutôt que pratique, et, en ce sens, il implémente véritablement les concepts de la programmation objet, d'une manière plus aboutie que des langages comme Perl ou même Python.

Développé au Japon par Yukihiro Matsumoto, ce langage a mis du temps à parvenir jusqu'aux programmeurs occidentaux et un certain engouement se fait jour.

Les utilisations les plus courantes de ce langage sont certainement constituées par les applications d'interface avec le Web, où Ruby commence à se poser en concurrent sérieux pour Perl.

PHP

Le langage PHP a été créé par Rasmus Lerdorf en 1994 pour des besoins personnels (*Personal Home Page*). En 1997, le projet devient un travail d'équipe et l'interpréteur est réécrit par Zeev Suraski et Andi Gutmans pour donner la version PHP3, version qui s'est rapidement imposée et devient PHP (Hypertext PreProcessor). La version PHP 4 date de 2000. Elle intègre en mode natif le moteur Zend (société privée créée par Suraskyi et Gutmans). PHP4 s'avère plus rapide, plus fiable et plus complet. Les scripts sont désormais compilés puis exécutés. La dernière version en date est PHP5 (2004) et représente une évolution majeure : nouveau modèle objet, support d'XML (*eXtended Markup Language*) et des services web, base de données locale, ...

PHP est surtout utilisé dans le développement de sites web : utilisé par plus d'un site web sur trois dans le monde.

ANNEXE 2 : JAVA

Java est le nom d'une technologie mise au point par Sun Microsystems qui permet de produire des logiciels indépendants de toute architecture matérielle. Le langage Java est un langage de programmation orienté objet. Java est sous licence GNU GPL depuis novembre 2006.

Un programme java s'exécute dans une machine virtuelle, dite machine virtuelle Java. Il contient le bytecode (code binaire) Java qui est le résultat de la compilation d'un programme écrit en Java par le compilateur Java (javac).

La *Java Virtual Machine* (JVM, en français Machine virtuelle Java) est une machine virtuelle permettant d'interpréter et d'exécuter le bytecode Java. Ce programme est spécifique à chaque plate-forme ou couple (machine/système d'exploitation) et permet aux applications Java compilées en bytecode de produire les mêmes résultats quelle que soit la plate-forme (d'où le slogan « *Compile once, run everywhere* »), tant que celle-ci est pourvue de la machine virtuelle Java adéquate.

Depuis des années, Sun Microsystems appelle Java la « technologie Java » dans son ensemble. En pratique, beaucoup de programmeurs utilisent le mot « Java » pour désigner le langage de programmation, tandis que la plate-forme d'exécution est appelée « JRE » (*Java Runtime Environment*, environnement d'exécution Java) et le système de compilation : « JDK » (*Java Development Kit*) plutôt que « compilateur Java ».

ANNEXE 3 : GLOSSAIRE

Certaines traductions en langue anglaise sont indiquées en italique.

Bogue <i>bug</i>	Défaut de conception ou de réalisation se manifestant par des anomalies de fonctionnement.
Classe	Modèle décrivant un objet informatique.
Compatibilité	Qualité d'un matériel ou d'un logiciel conforme aux règles d'interface d'un système informatique défini, et dont l'introduction n'altère pas les conditions de fonctionnement de ce système.
Donnée <i>data</i>	Représentation d'une information sous une forme conventionnelle destinée à faciliter son traitement.
Débogueur <i>debugger</i>	Programme d'aide à l'élimination des bogues.
Implanter <i>to implement</i>	Installer un logiciel ou un sous-système donné en réalisant les adaptations nécessaires à leur fonctionnement dans un environnement défini.
Implémenter <i>to implement</i>	Réaliser la phase finale d'élaboration d'un système qui permet aux matériels, aux logiciels et aux procédures d'entrer en fonction.
Information	Élément de connaissance susceptible d'être représenté à l'aide de conventions pour être conservé, traité ou communiqué.
Informatique	Science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social. Utilisé comme adjectif : qui se rapporte à l'informatique.
Interface	Jonction entre deux matériels ou logiciels leur permettant d'échanger des informations par l'adoption de règles communes, physiques ou logiques.
Logiciel <i>software</i>	Ensemble des programmes, procédés et règles, et éventuellement de la documentation, relatifs au fonctionnement d'un ensemble de traitement de données.
Matériel <i>hardware</i>	Ensemble des éléments physiques employés pour le traitement de données.
Objet informatique	Unité de programme réutilisable constituée d'un ensemble cohérent et protégé de données et d'opérations.
Pilote <i>driver</i>	Programme de gestion des échanges avec un périphérique. Synonyme : gestionnaire de périphérique.
Processus	Enchaînement et évolution temporelle ordonnés de procédés. Communément, un programme en cours d'exécution.
Système d'exploitation <i>operating system</i>	Logiciel gérant un ordinateur, indépendant des programmes d'application mais indispensable à leur mise en oeuvre.