Séminaires de Programmation Système: Unix et Langage C

Manuel Oriol

Séminaires Hiver 2000

Table des matières

Table des figures

Chapitre 1

Introduction

ceci est l'introduction

Chapitre 2

Introduction à Unix et à la Ligne de Commandes

2.1 Les Caractéristiques d'Unix

Unix est le système d'exploitation des grands serveurs par excellence. Ce système est multitâche, multi-utilisateur, multi-platforme, réparti et sécurisé.

Ce système existe depuis de nombreuses années et ne paraît pas prêt d'être rejeté. De fait, on parle d'Unix pour désigner le produit commercial de SUN mais aussi des Unix pour désigner l'ensemble des systèmes qui correspondent aux spécifications de la norme POSIX. Ceux-ci sont nombreux : UNIX de SUN, AIX d'IBM, NextStep de NEXT, Linux, BSD, MacOSX Server... Le plus étonnant est que même les systèmes d'exploitation personnels annoncés semblent vouloir prendre une base Unix (MacOSX par exemple).

Voyons maintenant les particularités de ce dernier et comment s'y retrouver.

2.2 Prendre un bon départ sous Unix

2.2.1 Le login

Pour commencer à se servir d'une machine qui possède un système Unix il faut avoir un **compte** sur cette machine. Qu'est-ce qu'un compte? En fait, ceci représente une identité sur la machine en question.

La machine Unix vous demande alors de lui fournir deux informations différentes: votre **login** et votre mot de passe. Le login représente alors votre identité et le mot de passe et le moyen de l'identifier. Notez que seul le login apparaît en clair lors de l'entrée de ces informations.

Pour ce qui est de notre cas, un environnement graphique se lance immédiatement. L'environnement graphique n'est qu'une surcouche du système et obéït aux mêmes lois que les autres programmes.

Voici quelques commandes intéressantes:

Commande	**Fonction**
passwd	vous permet de changer votre mot de passe
rlogin *nom_machine*	permet d'utiliser une machine distante
su *nom_utilisateur*	permet d'endosser l'identité d'un autre utilisateur
talk *utilisateur@nom_machine*	permet d'ecrire en direct à un autre utilisateur loggé sur la machine
finger *utilisateur*	permet d'avoir des informations sur un utilisateur (informations présentes dans **.plan**)

2.2.2 Le réseau

Un réseau de stations Unix a la particularité de fournir aux utilisateurs une interface similaire quelle que soit la station (la machine) où ils se connectent. De même pour les fichiers de chacun.

2.2.3 Une fois loggé

Une fois entré dans le réseau sous votre identité vous êtes libres de faire ce que vous voulez avec vos données et vos programmes, du moment que vous respectez les limitations que les autres utilisateurs vous ont fixé pour leur données. Nous expliquerons ce point plus avant par la suite.

2.2.4 Le terminal et la ligne de commande

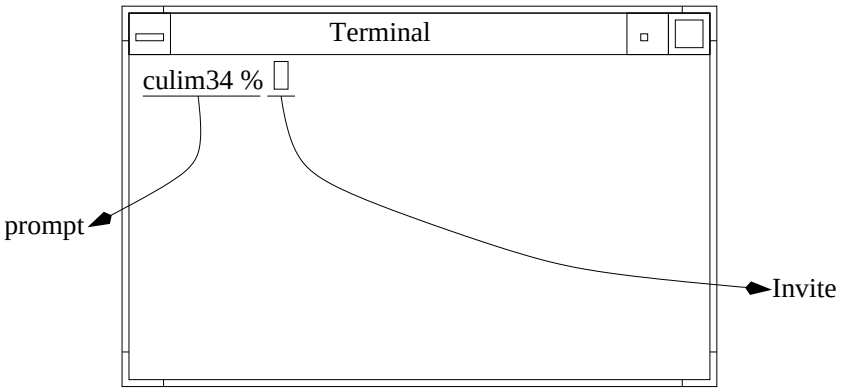

FIG. 2.1 *Exemple de terminal*

Un terminal (cf figure 2.1) est une fenêtre qui va vous permettre de lancer des programmes, de gérer vos fichiers et, plus généralement, d'interagir avec votre machine. On peut identifier le **prompt** qui est un message généré automatiquement et l'**invite** qui vous permet de taper vos commandes.

Exemple d'utilisation: pour lancer netscape (Navigateur et lecteur de mail et de news) tapez simplement *netscape* dans un terminal.

À partir de maintenant la phase de découverte est terminée et nous allons expliquer comment marchent les différents composants et que faire pour utiliser correctement le système.

2.3 Le Système de Fichiers

2.3.1 Généralités

Le système de fichier sous Unix est hiérarchisé. Il peut être représenté par un arbre dont la racine est "/" (voir figure 2.2) et les répertoires sont désignés par leur chemin à partir de la racine (chemin absolu).

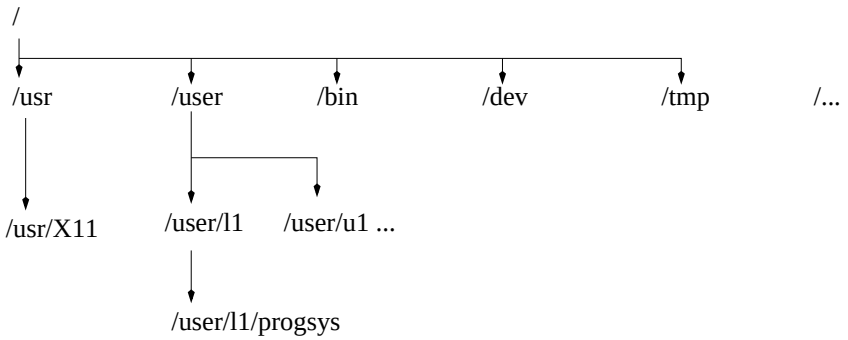

FIG. 2.2 *Exemple d'arborescence de fichiers*

Certaines règles sont toujours vraies dans un interpréteur (voir sous-section 2.4), en voici un petit résumé

- **.** représente le répertoire courant.
- **..** représente le répertoire parent.
- ~/ représente votre racine.
- ~**nomutilisateur** représente la racine de l'utilisateur **nomutilisateur**.

Voici quelques commandes pour se déplacer dans le système de fichiers et le scruter:

Commande	**Fonction**
ls	affiche le contenu du répertoire courant
cd *truc*	change de répertoire pour le répertoire *truc*

2.3.2 Organisation interne

Structure Un fichier est une suite de bytes (caractères) à laquelle est associée un bloc d'informations appelé i-node. Ce bloc est composé de six informations:

- taille en caractères (ou bytes)
- adresse d'un emplacement sur disque
- propriétaire et droits d'accès
- type (fichier ordinaire, répertoire, fichier spécial (device), lien)
- date de dernier accès, écriture et modification
- nombre de liens sur ce fichier

Les i-nodes sont rassemblés dans une table indexée. Un répertoire est un fichier de paires (index d'i-node, nom de fichier).

Network File System (NFS) Le NFS est un système qui permet d'accéder à des fichiers distants de manière transparente. Le principe est simple une machine est serveur pour une partie de ses fichiers et une ou plusieurs machines sont clientes et peuvent utiliser les fichiers en question. Notons qu'une machine peut à la fois être client et serveur...

exemple d'arborescence exportée: /user/l1 qui est stationné sur cuinfs.

Droits d'accès Le résultat de la commande **ls -al** est donné dans la figure 2.3

```
total 9912
drwxr-xr-x   2 oriol    denis         512 Oct  6 14:34 .
drwxr-xr-x   3 oriol    denis         512 Oct  5 18:06 ..
-rw-r--r--   1 oriol    denis        6030 Oct  6 14:32 PATH.eps
-rw-r--r--   1 oriol    denis        1453 Oct  6 14:31 PATH.fig
-rw-r--r--   1 oriol    denis        2057 Oct  6 14:35 TP1.aux
-rw-r--r--   1 oriol    denis       22836 Oct  6 14:35 TP1.dvi
-rw-r--r--   1 oriol    denis        7047 Oct  6 14:35 TP1.log
-rw-r--r--   1 oriol    denis       14358 Oct  6 14:34 TP1.tex
-rw-r--r--   1 oriol    denis     5447916 Oct  6 10:44 xterm.eps
-rw-------   1 oriol    denis     4603904 Oct  6 10:44 xterm.ps
```

FIG. 2.3 *Exemple de résultat de ls -al*

En voici la traduction:

- Le premier caractère indique le type de fichier (**-** pour un fichier standard, **d** pour un répertoire)
- Les trois caractères suivants concernent le possesseur du fichier, ce sont les opérations qu'il se donne le droit de faire **r** pour la lecture (Read), **w** pour l'écriture (Write) et **x** pour l'exécution (eXecute). Les trois caractères suivants représentent la même chose pour le groupe auquel il appartient et les trois derniers sont pour le reste du monde.
- On voit ensuite le nombre de liens sur le fichier puis le login du possesseur, le nom du groupe auquel il appartient, la taille du fichier, la date de dernière modification et, enfin, le nom du fichier ou répertoire.

Notons que les droits en exécution sur un répertoire correspondent à un droit d'ouverture du répertoire, les droits en écriture correspondent à une autorisation de création de fichiers et les droits en lecture correspondent à un droit de visualisation du répertoire.

Voici quelques commandes utiles:

Commande	**Fonction**
chmod	change les droits d'accès sur le fichier la syntaxe est chmod [ugo][+-][rwx] **nom_fic**.
mv *source destination*	change le fichier de place.
cp *source destination*	copie le fichier.
pwd	affiche le répertoire courant.
mkdir *nom*	fabrique un répertoire dans le répertoire courant.

Pour visualiser des fichiers et les éditer vous avez le choix des outils: on peut utiliser des commandes qui marchent en mode texte ou des commandes qui ne marchent qu'en mode graphique. En voici un bref descriptif:

Commande	**Fonction**
more	affiche le contenu d'un fichier
cat	affiche aussi le contenu d'un fichier texte
vi *fichier*	permet de modifier un fichier en mode texte (difficile à utiliser)
xemacs	editeur de fichier en mode graphique

2.3.3 Le quota

Du fait que les capacités de stockage ne sont pas étendues à l'infini les comptes sont limités en espace disponible. Pour connaître vos limitations et l'usage que vous faî tes de votre espace voici quelques commandes utiles:

Commande	**Fonction**
quota	affiche votre espace total e vos limites.
du -k *repertoire*	affiche la place totale prise par le *repertoire*.
df -k	affiche les informations sur les différentes partitions.

2.4 Le Shell et le Noyau Unix

2.4.1 Le noyau (ou kernel)

Le noyau Unix (voir figure 2.4) est un ensemble de petits programmes qui permettent de gérer l'exécution des autres programmes qui tournent sur la station. Ces primitives gérent l'accession à la mémoire et, plus généralement, aux différentes resources de la machine. Le shell ne va faire que des appels à ces primitives.

2.4.2 Le shell

Bases Un terminal est une fenêtre où s'exécute un shell qui fournit une ligne de commande. Ce 'shell' est un interpréteur c'est-à-dire que c'est un programme qui tourne sur la station et qui va traduire chacun des ordres que vous lui donnerez par l'entremise du clavier. Ces ordres sont, eux-même, des programmes ou des agrégats de programmes. Pour trouver un programme appelé, l'interpréteur

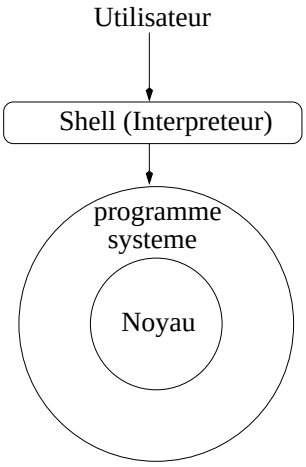

FIG. 2.4 *Couches de programmes sous Unix*

regarde une variable appelée **PATH** (on utilise **$PATH** pour avoir sa valeur) qui représente l'ensemble des répertoires à explorer.
exemple: Résultat de **echo $PATH**

.:/unige/java/jdk1.2/bin:/usr/bin:/user/u1/oriol/bin:/usr/ucb:/sbin

En regardant cet exemple on voit que cette variable est composée de plusieurs répertoires désignés les uns aprés les autres etséparés par **:**. On peut reconnaître le répertoire courant désigné par **.**. Cette liste de répertoires est ordonnée, c'est-à-dire que l'interpréteur cherche d'abord dans les premiers répetoires et s'il n'a pas trouvé la commande demandée il continue avec les suivants.

Par défaut vous utilisez un shell qui est le C-Shell et dont le fichier de configuration est le fichier placé à votre racine appelé **.cshrc** ce fichier est exécuté à chaque fois que vous lancez le C-Shell. Je vous conseille de changer de shell pour le TC-Shell (lancez-le avec **tcsh**) qui a exactement les mêmes fonctionnalités mais qui est plus convivial grâce aux possibilités de sa ligne de commande: on peut compléter automatiquement un nom en appuyant sur *Tab* et on peut récupérer les commandes exécutées précédemment en utilisant les flêches hautes et basses.

Voici quelques commandes simples pour voir les variables, les mettre à jour et comprendre mieux le shell:

Commande	**Fonction**
setenv *nom valeur*	met a jour une variable du shell
printenv [*nom*]	avec un argument affiche l'état de la variable concernée sans argument affiche toutes les variables.
which *nom_cmd*	affiche le chemin absolu vers cette commande.
chsh	change le shell par défaut.
man *nom_cmd*	affiche les informations sur une commande (préférez la version graphique **xman**)
source *nomfichier*	permet d'exécuter le fichier en conservant l'etat des variable après son exécution.

Les expréssions régulières Pour vous faciliter la tâche et automatiser certains traitements le shell utilise des raccourcis: les expressions régulières. Le principe est simple vous écrivez quelque chose et le shell se charge de remplacer votre expression par le ou les noms de fichiers qui correspondent à cette expression. Les deux principaux opérateurs sont ***** et **?**. ***** représente n'importe quelle suite de caractères (la plus grande possible) et **?** représente n'importe quel caractère (mais un seul).

exemple:

- mv * .. fait remonter tous les fichiers du répertoire courant dans son parent.
- mv *.zip .. fait remonter tous les fichiers se terminant par .zip du répertoire courant dans son parent.
- mv *sac* ~/. fait remonter tous les fichiers contenant sac dans leur nom du répertoire courant dans la racine de l'utilisateur.

2.4.3 Notion de processus

Un processus Unix est un programme en train de s'exécuter. Un processus est caractérisé par par un numéro que l'on peut avoir en utilisant la commande **ps -al** et en regardant dans la colonne PID (process ID).

```
F   UID   PID  PPID CP PRI NI   SZ  RSS    WCHAN S TT        TIME COMMAND
8     0   304     1  0  54 20 1496 1024 ar_g_hea S console   0:00 /usr/lib/saf
8   606   440   433  0  46 20 1288 1048 modlinka S pts/3     0:00 /bin/csh
8   606   624   440  0  48 202700014800 ar_g_hea S pts/3     0:18 /unige/ow3/b
8   606   641   624  0  57 2018512 3352 ar_g_hea S pts/3     0:00 (dns helper)
8   606  1117   440  0  49 202332812760 redirmin S pts/3     0:01 ugtool
8   606  1146     1  0  49 201560812792 redirmin S pts/3     0:02 /unige/frame
```

FIG. 2.5 *Exemple de résultat de ps -al*

Un processus Unix a les mêmes droits que son possesseur et seul son possesseur ou le super-utilisateur (root) a le droit de le stopper. Vous l'avez donc compris si on pirate votre compte c'est-à-dire si quelqu'un arrive à faire tourner un processus dont vous êtes le possesseur et à le contrôler il peut faire pas mal de choses à votre compte mais du fait des protections il ne pourra pas toucher à un autre compte du réseau.

Une fois muni du numéro d'un processus qui vous appartient, vous pouvez le tuer (sic!). Pour cela utilisez la commande **kill** avec l'option -9.

En regardant le résultat de **ps -al** on peut voir une multitude de processus tourner. Ceci est possible car Unix est multitâche et donc peut faire tourner plusieurs en même temps (se reporter au cours pour plus de précisions). Pour lancer une exécution en parallèle il suffit de taper un **&** en fin de la ligne de commande normale. Le shell lance alors un processus de plus qui ne fait qu'effectuer la commande demandée laissant l'interpréteur courant poursuivre son exécution (essayez par exemple **netscape &**).

Commande	**Fonction**
ps	affiche les processus qui tournent sur la machine physique
top	affiche les processus avec le pourcentage de temps machine utilisé
kill -9 *numero*	tue le processus ayant le numero *numero*

2.5 Commandes utiles à connaître

Voici quelques commandes que tout le monde connaît et qui peuvent vous être utiles.

Commande	**Fonction**
tar xvf *nom_fic.tar* tar cvf *nom_fic_cible fichiers...*	détarre une archive tar crée une archive
zip,unzip *nom_fic*	compacte et décompacte (compatible Winzip)
gzip,gunzip *nom_fic*	compacte et décompacte
xv *nom_fic*	visualise un fichier image

2.6 Exercices

Voici quelques petits exercices simples que je vous conseille vivement de faire si vous voulez partir du bon pied.

1. Créer un fichier texte avec ecrit quelque chose dedans.
2. Créer un répertoire et y ranger le fichier créé.
3. Copier une image du Web et la visualiser avec **xv**.
4. Explorer mon répertoire (~oriol).
5. Expliquer pourquoi on ne peut accéder à:
   /user/u1/oriol/Sources_ecrits/COURS/TP_Sysop/TP1.

Chapitre 3

Construction de scripts

Pourquoi apprendre à construire des scripts minimaux? Une des réponses possibles est que le développement d'un script en tant que prototype pour une application est rapide et peu couteux. De plus cela permet souvent de réaliser des opérations répétitives sans grand effort et en un minimum de temps.

3.1 Introduction

3.1.1 Qu'est-ce qu'un script?

Un script est une suite de commandes. Dans l'absolu rien ne dit que cette suite de commande doit se trouver dans un fichier; néanmoins, dans la pratique, un script est un fichier exécutable. Ils permettent d'automatiser des opérations répétitives.

3.1.2 Redirection de flots

La première chose à apprendre est comment faire des redirections de flot de données. Le principe est simple: les résultats des commandes telles que **ps** ou **more** est un flot de données (classé ligne par ligne) auquel on peut appliquer plusieurs traitements et que l'on peut rediriger. Chaque commande ayant un input et un output on redirige les flots vers ceux-ci ou vers d'autres fichiers.

Voici les différents opérateurs de redirection:

Opérateur	**Fonction**
$cmd1 \mid cmd2$	dirige l'output de $cmd1$ sur l'input de $cmd2$
$cmd1 > fic1$	dirige l'output de $cmd1$ dans le fichier $fic1$ en écrasant son contenu
$cmd1 >> fic1$	dirige l'output de $cmd1$ dans le fichier $fic1$ et le place à la suite du texte existant
$cmd1 < fic1$	dirige le fichier $fic1$ dans l'input de $cmd1$

3.1.3 Commandes Utiles

Pour utiliser au mieux ces redirections il faut être capable de traiter les flots de données, voici donc quelques commandes et quelques indications pour les

utiliser:

Commande	**Fonction**
grep *expr*	ne garde que les lignes qui matchent avec *expr*
sed *expr*	identique mais peut aussi effectuer des remplacements (voir exemple 3.2.2).
find *dir* -name *expr* -print -exec *com* \;	trouve dans l'arborescence en-dessous de *dir* les fichiers dont le nom matche avec *expr* et exécute *com*

3.2 Constrution de Scripts C-Shell

3.2.1 L'essentiel

les arguments Les arguments sont la suite de caractères qui suivent la commande quand vous l'exécute, ils sont séparés par un espace et voici comment les utiliser:

- Les arguments de la commande sont repérés par **$argv**[**1**] **... $argv**[**i**] (ou **$1 ... $i** en abrégé) où **i** est le i-ème argument.
- **$#argv** (ou **$#** en abrégé) est le nombre d'arguments.
- **$0** est le nom de la commande lancée.
- **$$** est le numéro du processus courant.
- **$argv** (ou **$*** en abrégé) retourne la liste des arguments.

exemple: **ls -al** a un seul argument: **-al** pour le récupérer on utilise **$argv**[**1**] ou **$1**

les variables Les variables sont des données internes qui servent au programmeur dans son programme. En C-Shell voici comment on utilise les variables:

- **$variable** ou **${variable}** retourne la valeur de la variable.
- **$variable**[*indice*] ou *indice* est un entier ou un intervalle de la forme *indice*1-*indice*2 retourne la liste des éléments qui correspondent.
- **$?$variable** substitue 0 si la variable est indéfinie et 1 sinon.
- **set** *variable*=*valeur* affecte la variable *variable* à la valeur *valeur*.
- **unset** *variable* supprime la définition de la variable.
- **setenv** *variable valeur* affecte la variable d'environnement *variable* à la valeur *valeur*.
- **unsetenv** *variable* supprime la définition de la variable d'environnement.
- **set** affiche les valeurs des variables.
- **printenv** affiche les valeurs des variables d'environnement.
- **@** *variable*=*expression* met à jour la valeur de *variable* avec la valeur de *expression*.
- **echo** *truc* affiche la valeur de truc. truc

On voit donc que **argv** n'est qu'une variable comme les autres lors de l'exécution de programme.

les expressions On a vu que l'on pouvait affecter une expression à une variable. Comment construit-on une expression? Le plus simple est de penser à une ligne de calcul et d'utiliser les opérateurs classiques suivants:

Opérateur	**Fonction**
$expr1 + expr2$	fait la somme.
$expr1 - expr2$	fait la différence.
$expr1 * expr2$	fait le produit.
$expr1 / expr2$	fait le quotient.

Pour forcer l'evaluation on inclut l'expression dans des parenthèses ().

Exemple: le programme suivant calcule le produit de 2 valeurs entrées en argument

```
#!/usr/bin/csh

set prod=0

@ prod = $argv[1] * $argv[2]
 echo ``le resultat est ''$prod
```

Remarque : on écrit #!/usr/bin/csh en début de fichier pour que l'interpréteur de commandes lance un C-Shell pour exécuter le programme en question. Cela est nécessaire car, par défaut l'interpréteur le voie comme un script issu de sh.

Opérateurs booléens L'algèbre de Boole (algèbre des propositions vraies ou fausses) nous fournit lui aussi quelques opérateurs qui nous seront utiles plus tard. En voici un petit aper cu:

Opérateur	**Fonction**
$expr1 > expr2$	supérieur.
$expr1 < expr2$	inférieur.
$expr1 >= expr2$	upérieur ou égal.
$expr1 <= expr2$	inférieur ou égal.
$expr1 == expr2$	égal.
$expr1 != expr2$	différent.
$expr1$ && $expr2$	ET logique.
$expr1 \mid\mid expr2$	OU logique.

L'expression $expr1 > expr2$ a alors la valeur 1 (vrai) si la valeur de $expr1$ est supérieure à celle de $expr2$ elle prend la valeur 0 (faux) sinon. et de même pour chacun des opérateurs du premier groupe.

Expliquons plus les opérateurs du deuxième groupe. Le ET logique de $expr1$ et $expr2$ a pour valeur 1 si $expr1$ et $expr2$ ont pour valeur 1. Il a pour valeur 0 sinon. Le OU logique de $expr1$ et $expr2$ a pour valeur 1 si $expr1$ ou $expr2$ ont pour valeur 1. Il a pour valeur 0 sinon (c'est-à-dire s'ils ont tous les deux pour valeur 0).

Algorithmique: quelques structures de contrôle

Les structures de contrôle sont des structures qui permettent de contrôler le dérulement d'un programme et de lui faire subir des modifications par rapport à une exécution linéaire.

if...then...else...endif La structure **if** va permettre de vérifier une condition et d'avoir deux traitements possibles suivant la véracvité de cette condition. La syntaxe est la suivante:

if (*expr_bool*) **then**
execute_si_vrai
else *execute_si_faux*
endif

Exemple d'application: ce programme vérifie un mot de passe (ici en clair 12345678)

```
#!/usr/bin/csh

set passwd=12345678

if ( $1 == $passwd ) then
    echo "ok"
    else echo "refuse"
endif
```

foreach...end La structure de contrôle **foreach** permet d'appliquer un même traitement à une liste de valeurs. En voici la syntaxe:

foreach *variable* (*liste_valeurs*)
suite_de_commandes
end

Le shell va exécuter une la *suite_de_commandes* pour chaque valeur de *liste_valeurs* en affectant la valeur courante de la liste a *variable*.

Exemple d'application: ce programme liste tous les répertoires passés en arguments.

```
#!/usr/bin/csh

set i

foreach i ($argv)
   echo "-----------------------------------------------------------"
   echo $i
   echo "-----------------------------------------------------------"
   ls $i
end
```

while...end Cette structure permet de faire une suite d'opération tant qu'une expression est vraie. La syntaxe est la suivante:

while (*expression*)
suite_de_commandes

end

Exemple d'utilisation:

```
#!/usr/bin/csh

while (1)
   wait
   echo ``penser au cadeau de Melanie''
end
```

Eh oui ce programme ne s'arrête jamais... comment l'arrêter? Avec **kill** ou en faisant **ctrl+C** dans la fenêtre qui exécute le programme.

...et les autres D'autres commandes existent comme **switch**, **repeat**, **break** et **goto**; vous pouvez en trouver la description en regardant dans le **man**.

3.2.2 Exemple d'application

Nous allons prendre un simple exemple d'application: comment faire pour éliminer tous les fichiers portant les noms passés en argument. Appelons cette commande **rmmax** et développons-la. Pour chaque nom passé en argument il faudra appliquer la même suite de commandes: nous utiliserons une boucle **foreach**. Pour chacun de ces noms il faudra alors trouver toutes les occurences du nom dans l'arborescence et leur appliquer un taitement simple (**rm**) nous pouvons donc utiliser **find** pour cela.

On obtient donc le programme suivant:

```
#!/user/bin/csh

set i

foreach i ($argv)
find . -name $i -print -exec rm {} \;
end
```

3.3 Exercices

Lisez le manuel sur ces commandes et pour hacune essayez de savoir ce qu'elles fait.

- find
- sed (difficile)
- grep
- kill
- rlogin
- xterm
- tar
- awk
- top
- telnet

- history
- man
- cd, pushd, popd, dirs, chdir
- ls
- last
- ps
- bg, fg, jobs, stop, notify
- who (am i)
- csh
- touch
- banner
- alias

Chapitre 4

Programmation en C: Découverte

4.1 Généralités

Le C a été développé en même temps qu'Unix et a été utilisé pour programmer son noyau. Le C est donc proche de la machine et de sa représentation interne des variables. Son principal avantage est d'être un langage qui produit des programmes rapides et (facilement) portables sur différentes plateformes. C'est un langage tellement connu et utilisé en industrie qu'il a servi de base à plusieurs langages objets lorsque ceux-ci sont apparus (voir cours de Programmation Objet) au nombre desquels C++, Objective-C (qui ne sont que de simples extensions du C) ou encore Java (qui en reprends partiellement la syntaxe). Il a donné lieu à une spécification : la norme ANSI connue sous le nom de ANSI-C et à une autre norme, la norme POSIX. Sa connaissance (même superficielle) paraît donc actuellement indispensable à tout informaticien.

4.2 Classification du Langage

Le langage C fait partie de la grande famille des langages impératifs (au même titre que Ada, Basic, Fortran, Pascal...) ce qui signifie qu'il exécute chaque ligne de code l'une après l'autre (nous verrons plus avant cette notion de ligne de code).

4.3 Compilation?

4.3.1 Code Source et Compilation

Le **code source** d'un programme est sa forme écrite en clair dans un fichier texte. En effet, à l'inverse des scripts C-shell (cf. Séminaires 1 et 2) les programmes C nécessitent une opération appelée **compilation** pour pouvoir s'exécuter. Cette opération transforme le **code source** en **code exécutable** (cf cours sur le CPU 'code machine'). On parle alors de **langage compilé** plutôt que de **langage interprété**. Ceci pose le problème du langage (cf sous-section

4.3.2). Pour **compiler** plusieurs **compilateurs** différents existent. Celui que je vous conseille est **gcc** (/unige/gnu/bin/gcc) car il est présent sur bon nombre de plateformes et ne vous créera donc que peu de probèmes de compatibilité. Pour avoir le PATH correct vous pouvez rajouter dans votre **.cshrc** la ligne suivante:

```
source /unige/util/env/gnu.csh
```

Commande	**Fonction**
gcc *fichier.c* [-o *fichierSortie*]	compile le programme C du *fichier.c* et place l'exécutable dans *fichierSortie* (par défaut a.out)

4.3.2 Langage de programmation

Un langage de programmation est une convention d'écriture pour pouvoir communiquer avec l'ordinateur. Un autre programme traduit alors le texte de départ pour le rendre compréhensible par la machine. Si ce 'traducteur' tourne pendant l'exécution du programme c'est un **interpréteur**. Si ce 'traducteur' tourne une fois pour toute et produit quelque chose d'executable, alors c'est un **compilateur**. Pour C on a donc un compilateur (par exemple **gcc**) alors que pour le C-shell on a un interpréteur.

4.4 Programmation Simple

4.4.1 Mon Premier Programme

Voici un petit programme qui ne fait qu'une seule chose: écrire quelque chose sur le terminal.

```
#include <stdio.h>

void main() {

printf("Hello");
}
```

Dans ce listing, on peut déjà repérer tois parties:

1. Une ligne indiquant une inclusion de quelque chose appelé **stdio.h**.
2. Une ligne indiquant une partie principale (**main** en anglais) comprise entre accolades.
3. Une ligne exécutant un affichage (**printf**) d'un message "Hello".

La première partie correspond à une inclusion de bibliothèque (nous reverrons ce concept plus avant dans les séminaires) la seule chose à retenir à ce point est que pour utiliser **printf** on doit inclure **stdio.h**.

La deuxième partie est le point d'entrée du programme: elle indique que l'exécution devra commencer par cet endroit-là.

La troisième partie constitue le corps du programme: c'est ce qui va être exécuté en commençant par la première ligne et en suivant l'ordre des lignes. On appelle ces lignes lignes de code. En C, chaque ligne de code doit se terminer par un ;.

4.4.2 Variables et codage

Qu'est-ce qu'un Octet (byte)?

Un octet(voir figure 4.1) est une suite de 8 bits (case '0 ou 1').

0	1	0	0	0	0	0	1

FIG. 4.1 *Représentation en machine du caractère A*

Le codage des caractères (**char**) se fait par l'interprétation d'un byte en caractères (par exemple 'A' = 97 , 'B' = 98 ...) et ainsi les chaines de caractères sont codées comme une suite de bytes. On peut ainsi comprendre que dans ce système il ne peut y avoir que 256 caractères (codés de 0 à 255).

Le codage des entiers (**int**) se fait sur 2 ou 4 octets (sous Unix sur 4) ce qui permet des valeurs entre $-(2^{31})+1$ et 2^{31}.Le codage des entiers longs (**long**) se fait sur 4 octets...

Les types de variables

On voit donc que l'on va avoir une représentation différente suivant le type de donnée. Ceci a amené les concepteurs du langage à définir des types pour les variables (pour les données). On va tout d'abord se limiter aux types dont nous avons parlé précédemment (char, int, long) mais nous reviendrons sur les types de données et sur le typage plus avant dans les séminaires. En théorie les types devraient garantir que l'on ne peut ajouter des caractères à des entiers; néanmoins en C cela n'est pas vrai (tout au plus le compilateur le signale à la compilation). Ainsi C n'est pas un langage fortement typé.

Les variables

Comme en C-Shell, si on veut utiliser une variable, il faut la déclarer avant de l'utiliser. Par exemple si, dans le programme précédent on veut pouvoir afficher 2 fois le message "Hello" on peut procéder de la sorte :

```
#include <stdio.h>

void main() {

char* chaine;
chaine = "Hello";

printf(chaine);
printf(chaine);

}
```

On a défini une chaine de caractère (**char*** délimitée par " et ") portant le nom *chaine* puis on a **affecté** la valeur "Hello" à cette chaine. Puis on a affiché 2 fois de suite cette même chaîne.

Manipulation de Variables Pour manipuler une variable il faut avant tout la déclarer. Ensuite vous pouvez l'utiliser en l'appelant par son nom. Voici quelques exemples d'utilisation :

```
int a;
int b;
a = 2;
b = 3;
a = a + b;

printf(" le resultat est : %d",a);
```

ou encore :

```
char a;

a = 'a';
a = a + 1;
printf(" le resultat est : %c",a);
```

Notez que toutes les vaiables doivent être déclarées au début du main et en même temps (on ne peut pas les définir au milieu des autres lignes de code).

Interaction avec l'Utilisateur Nous allons évoquer deux fonctions : **scanf** et **printf**.

scanf va permettre de lire une entrée au clavier de l'utilisateur et **printf** va permettre d'écrire quelque chose dans un terminal. Leur implantation demande au système de communiquer avec le terminal (appel système). Voici le fonctionnement de **printf** pour afficher une chaîne de caractères :

```
printf(" le resultat est :");
```

Pour afficher une chaîne et un entier a:

```
printf(" le resultat est : %d",a);
```

Notez les conventions de représentation des aractès spéciaux **\n** pour un retour à la ligne, **\b** pour un *backspace*, \" pour " et \\ pour un \.

(a remplacera alors le pourcentage-d par la valeur de a)

scanf va permettre de lire des entrées au clavier. Pour lire un nombre au clavier et le ranger dans la variable entière i voici comment faire :

```
scanf("%d",&i);
```

Voyons comment programmer le jeu le plus bête du monde :

```
#include <stdio.h>

void main() {

int i;

printf("Bonjour, entrez un nombre :\n");
scanf("%d",&i);
i=i+1;
printf("Je joue %d, vous avez perdu, a bientot peut-etre...\n",i);
}
```

Structure de test : if La syntaxe est un peu différente de celle du C-Shell :
if (*test*) **{**
partie de programme exécutée si le test est vrai (sa valeur est différente de 0)
}
else { *partie de programme exécutée si le test est faux (sa valeur est égale à 0)*
}

La structure de contrôle if utilise le même format de test qu'en C-Shell :

Opérateur	**Fonction**
$expr1 > expr2$	supérieur.
$expr1 < expr2$	inférieur.
$expr1 >= expr2$	upérieur ou égal.
$expr1 <= expr2$	inférieur ou égal.
$expr1 == expr2$	égal.
$expr1 != expr2$	différent.
$expr1$ && $expr2$	ET logique.
$expr1 \mid\mid expr2$	OU logique.

Exemple d'utilisation :

```
#include <stdio.h>

void main() {

int i;
int annee_courante=1999;

printf("Bonjour, entrez votre annee de naissance :\n");
scanf("%d",&i);
if (i>annee_courante) {
printf("Vous n'etes meme pas ne!!!");
}
else {
printf("Vous avez %d ans... vous etes deja super vieux, mon vieux!!!\n",1999-i);
}

}
```

4.5 Conclusion

Dans ce séminaire on a présenté quelques éléments du langage C ainsi que quelques exemples. Ceci devrait déjà vous permettre de construire de petits programmes amusants pour tester les mécanismes de compilation et d'exécution du code compilé. Je vous engage vivement à essayer de faire les exercices pour comprendre pleinement la suite des séminaires.

4.6 Exercices

1. Ecrivez le jeu le plus bête du monde et entrez 99999999999. Que se passe-t-il et pourquoi?

2. Faites un petit programme qui ajoute 2 entiers initialisés à 7 et 9 puis affichez la valeur de leur somme (solution dans le cours sur la mémoire primaire, transparent 8).
3. Faites un petit programme qui mutliplie 2 entiers entrés au clavier.
4. Ecrivez un programme qui calcule l'age de votre chien suivant la date entrée (en tenant compte du mois et du jour).
5. Reposez-vous.
6. Ecrivez un petit programme qui teste un mot de passe et donne une information ensuite si ceui-ci est vérifié.
7. Reposez-vous.

Chapitre 5

Programmation en C, les Structures de Programmation

5.1 Types de Données

Dans le précédent séminaire nous avons vu des types de données représentant des entiers et des caractères. Ce petit paragraphe vous présente les **float** (codeés sur 4 octets) qui représentent les nombres en virgule flottante, les **double** (codeés sur 8 octets) représente les nombres à virgule flottante en double précision et les **long double** (codeés sur 12 octets) qui sont des nombres flottants longs.

Lorsque vous affectez un **float** à un **int**, une conversion se fait en tronquant le **float**. Exemple :

```
float a=1;
int b;
a = (a/1230995);
b = a;
```

Après cette opération a=0.00001... et b=0. Il faut donc faire attention aux erreurs qui en découlent et aux erreurs d'arrondis.

5.2 Structurs de Contrôle (Suite)

Après avoir étudié **if** (cf Séminaire 2) nous allons voir les structures telles que **for**, **while** et **case**.

5.2.1 Structure for : structure pour

La structure permet de répéter un certain nombre de fois une partie de programme tant qu'un test n'est pas vérifié. Sa syntaxe est la suivante :

for($expression_1$;$expression_2$;$expression_3$) { *bloc d'instructions* }
suite du programme...

Cette structure a le fonctionnement suivant :

1. on évalue l'$expression_1$
2. on évalue l'$expression_2$ si sa valeur est vraie (différente de 0) on exécute le *bloc d'instructions* puis $expression_3$ si sa valeur est fausse (égale à 0) on sort de la structure **for** et on exécute la *suite du programme...*
3. On passe à létape 2

Schématiquement on aura quelque chose du style :
for(*initialisation*;*test*;*modification de variable de boucle*)...
Exemple :
voici une boucle qui calcule les factorielles successives entre 1 et 15.

```
int i,x;
x=1;

for(i=1;i<15;i=i+1)
 {
    x=x*i;
    printf("factorielle de %d est : %d\n",i,x);
 }
```

5.2.2 Structure while: structure tant que...

La structure **while** a la syntaxe suivante :
while(*expression*) {
bloc d'instructions
} *suite du programme...*
Cette structure a le fonctionnement suivant :

1. On évalue l'*expression*. Si celle-ci a la valeur vraie (différente de 0) alors on exécute le *bloc d'instructions* sinon on sort de la structure **while** et on passe à la *suite du programme....*
2. On passe à létape 1

Exemple :
- Un exemple d'utilisation est en page 26 du cours de language de communication informatique.
- Voici un autre exemple qui demande un mot de passe et le vérifie.

```
#include <stdio.h>

void main(){
int passwd=1;

while (passwd != 12345)
   {
   printf("\n passwd: ");
   scanf("%d",&passwd);
   }
...

}
```

5.2.3 Structure do...while: structure tant que(2)...

La structure **do** a la syntaxe suivante :
do {
bloc d'instructions
}
while(*expression*)
En fait, cette structure est la même que la précédente sauf que le *bloc d'instructions* est au moins exécuté une fois puis l'*expression* est évaluée.

5.2.4 Instruction break

L'instruction **break** permet de sortir d'une structure (**while**, **do**, **for**) ou **switch** en cassant sa structure. Par exemple :

```
#include <stdio.h>

void main(){
int passwd=1;

while(1) { // boucle infinie


if (passwd==12345) //on teste le mot de passe
   {
   break;
   }
  else
   {
   printf("\n passwd: ");
   scanf("%d",&passwd);
   }
}
```

5.2.5 Structure switch: le super if...

La structure **switch** a la syntaxe suivante :
switch (*expression*) {
case $expression\_constante_1$: { *bloc d'instructions 1* }
case $expression\_constante_2$: { *bloc d'instructions 2* }
...
case $expression\_constante_n$: { *bloc d'instructions n* }
default: { *bloc d'instructions n+1* }
}
Cette structure a le fonctionnement suivant :
On évalue l'*expression*. Si sa valeur correspond à $expression\_constante_i$ on exécute *bloc d'instructions i ... bloc d'instructions n+1*. Si sa valeur ne correspond à aucune $expression\_constante_i$, alors on n'exécute que *bloc d'instructions n+1*

Ceci n'est pas très pratique! Ce qui se passe donc en pratique est que l'on utilise **break** en fin de *bloc d'instruction.*

Exemple : qui peut servir pour faire un menu.

```
#include <stdio.h>

void main(){

int choix;

scanf("%d",&choix);

switch (choix){
 case 2 : {//choix de l'entree 2
          printf("%d",2);
          break;
          }

 case 1 : {//choix de l'entree 1
          printf("%d",1);
          break;
          }
 default :{// mauvais choix
          printf("rien");
          }

}
}
```

5.3 Procédures et Fonctions

5.3.1 Concepts et Exemples

Les procédures et les fonctions constituent l'essence de la programmation structurée et sont les outils de préférés du programmeur pour se simplifier le travail. Elle permettent de rendre une tâche abstraite dans la programmation d'un projet un tant soit peu important.

Schématiquement, une fonction est une espèce de sous-programme utile et que vous voulez programmer et que vous utilisez plusieurs fois. Par exemple : vous voulez disposer d'un affichage standard pour deux entiers. Une fonction va se décrire ainsi :

type_de_retour nom_de_la_fonction(*variables_en_entree*){
programme à exécuter
}

Par exemple :

```
void afficher_entier(int a, int b){

printf("les valeurs sont %d et %d\n",a,b);
```

```
}
```

Dans cet exemple le type de retour est **void** qui signifie "rien". Voici le même exemple ou la fonction retourne le premier entier :

```
int afficher_entier(int a, int b){

printf("les valeurs sont %d et %d\n",a,b);
return a;
}
```

On peut remarquer le **return a;** qui signifie que la fonction doit s'arrêter et retourner dans le programme qui l'a appelé en retournant la valeur de a.

Les noms **a** et **b** ne sont visibles que dans la fonction **afficher_entier**. Dans le cas où ces noms aient déjà été utilisés dans le main par exemple ce sont des variables différentes. On utilise la fonction en l'appelant par son nom deux fonctions ne doivent donc pas avoir le même nom.

Exemple :

```
#include <stdio.h>

void afficher_entier(int a, int b){
printf("les valeurs sont %d et %d\n",a,b);
}

int afficher_entier2(int a, int b){
int d;
d=a;
printf("les valeurs a et b sont %d et %d\n",a,b);
return d;
}

void main(){
int b=1,c=2;

afficher_entier(b,c);
c=afficher_entier2(b,c);
afficher_entier(b,c);
}
```

Quand on exécute ce programme compilé on obtient l'affichage suivant :

```
[oriol@linoriol]$ a.out
les valeurs sont 1 et 2
les valeurs a et b sont 1 et 2
les valeurs sont 1 et 1
```

On appelle procédure qui retourne **void** (rien).

Un point intéressant à noter est que main est une fonction comme les autres (ou presque...).

5.3.2 Variables Globales et Variables Locales

Comme nous l'avons dit les variables n'ont de valeur que dans la fonction où elles sont définies, on les appelle **variables locales**. Il existe des variables qui sont définies dans le programme entier, elles sont définies juste après les inclusions de librairies (#include ...). Exemple:

```
#include <stdio.h>

int lastint;

void afficher_entier(int a, int b){
printf("les valeurs sont %d et %d\n",a,b);
lastint=b;
}

int afficher_entier2(int a, int b){
printf("les valeurs a et b sont %d et %d\n",a,b);
lastint=b;
return a;
}

void main(){
int b=1,c=2;

afficher_entier(b,c);
c=afficher_entier2(b,c);
afficher_entier(b,c);
printf("derniere valeur traitee %d\n",lastint);

}
```

Donne comme retour:

```
[oriol@linoriol]$ a.out
les valeurs sont 1 et 2
les valeurs a et b sont 1 et 2
les valeurs sont 1 et 1
derniere valeur traitee 1
```

5.4 Commentaires

Comme on peut le voir dans les listings precedents les programmes deviennent de plus en plus difficiles a relire avec le temps et la taille. Il est donc **très** intéressant de mettre des commentaires dans le listing du programme. Ceux-ci sont alors purement et simplement enlevés lors de la compilation.

Il existe deux manières de marquer des commentaires en C:

```
/* debut des commentaires
...
fin des commentaires */
```

Qui permet d'écrire des commentaires sur plusieurs lignes et:

```
eventuellement code //commentaires jusqu'a la fin de la ligne
```

Ceci pourrait donner sur le listing précédent :

```
#include <stdio.h>

/* variable permettant de retourner le dernier entier
*/
int lastint;

/* fonction permettant d'afficher 2 valeurs
*/
void afficher_entier(int a, int b){
printf("les valeurs sont %d et %d\n",a,b);
lastint=b;
}

/* fonction permettant d'afficher 2 valeurs
*/
int afficher_entier2(int a, int b){
printf("les valeurs a et b sont %d et %d\n",a,b);
lastint=b;
return a;
}
/* commentaire sur le main
*/
void main(){
int b=1,c=2;

afficher_entier(b,c);
c=afficher_entier2(b,c); // on affecte ici pour une bonne raison
afficher_entier(b,c);

// et enfin on affiche la derniere valeur traitee...
printf("derniere valeur traitee %d\n",lastint);
}
```

5.5 Exercices

L'exercice est de fournir un programme qui permette de réaliser des opérations de calcul simple (style calculatrice). Il devra proposer les operations suivantes : addition, multiplication, division et soustraction. Un petit menu permettra de choisir entre ces opérations et on pourra réutiliser le résultat précédemment calculé. Les commentaires pertinents devront se faire très présents dans votre code source. Un rapport sera à me rendre avant le 2 décembre 14h15. Celui-ci devra comporter une explication de votre programme (comment l'utiliser, quelles ont été les difficultés et pourquoi...) et vos listings en annexe (avec leur chemin absolu). Pour toute questions n'hésitez pas à passer au bureau 403, consulter les news ou envoyer des e-mails à **oriol@cui.unige.ch**. Question bonus : implantez aussi un choix pour calculer des factorielles.

Chapitre 6

Pointeurs et Mémoire

6.1 Pointeurs

Les pointeurs sont des entiers qui indiquent la position dans la mémoire de certaines données.

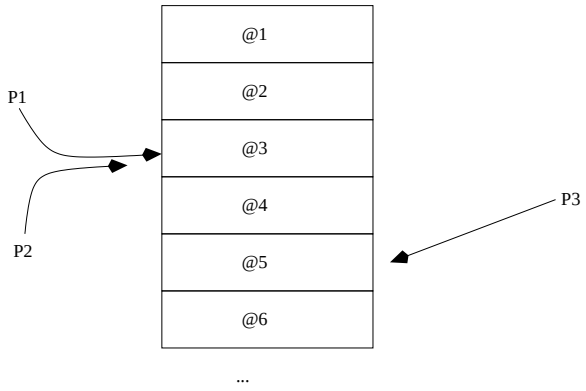

FIG. 6.1 *Les Pointeurs*

Dans la figure 6.1 on peut voir que deux pointeurs différents pointent sur la même case mémoire **@3** ainsi, si la valeur de P1 est la même que celle de P2 alors on peut dire qu'ils pointent sur la même chose. Lorsqu'on veut comparer deux données on va comparer les valeurs pointées par ces pointeurs (s'ils sont différents). Comment déclare-t-on des pointeurs?

Il n'y a que deux choses á connaître:

1. Si i est une variable **&i** est un pointeur sur i.
2. Si p est un pointeur alors ***p** est la valeur pointée par p.

Il s'ensuit que ***&i** est i mais que **&*i** n'a réellement de sens que si i est un pointeur. De la même manière, lorsqu'on déclare un pointeur sur un type de variable on écrit, par exemple, dans le cas d'un entier:

```
int *i;
```

i est alors un pointeur sur un entier (la valeur pointée par i est un int). Sur la figure 6.1 on voit clairement que P3=P2+2 et que donc, si l'on commet une

erreur sur une incrémentation de valeur et que l'on incrémente le pointeur sur cette valeur, alors on peut aboutir à des erreurs d'exécution grave (exemple: Segmentation Fault).

6.2 Passage de Paramètres par Valeur ou par Réfé-rence

Nous allons voir maintenant pourquoi on parle de passage de paramètres par valeur ou par référence. Le passage de paramètre est la manière dont on va passer des arguments d'une fonction. On a deux possibilités: soit on passe directement la valeur d'une variable (exemple: int multiplier(int a, int b)...) soit on passe un pointeur sur la valeur de la variable (exemple: int multiplier (int *a,int *b)...). Dans le premier cas la valeur est copiée et on ne peut plus agir sur celle-ci dans la fonction. dans le deuxième cas c'est la valeur du pointeur qui est copiée et dont on peut agir sur la variable passée en paramètre.

Exemple: Voici une fonction qui calcule les occurences de deux caractères différents dans un chaîne de caractères donnée en argument.

```
void  compter_occurences(char *str,int *a, char b, int *c, char d){
while (*str!=0){
        if (*str==b){
            *a=*a+1;
        }
        if (*str==d){
            *c=*c+1;
        }
}
}
```

Un autre exemple d'utilisation de ceci est visible dans la fonction **scanf** (voir Séminaires numeros trois et quatre).

6.3 Gestion de la Mémoire

Pour utiliser les fonctions que nous allons voir, il faut inclure la librairie **stdlib**.

Allocation de mémoire: malloc Lorsqu'on n'utilise pas une variable ayant une place mémoire allouée à sa création (c'est-à-dire une variable ayant une taille variable et non encore définie) on doit allouer de la mémoire pour pouvoir la définir et y stocker la valeur de la variable. Ceci se fait par l'utilisation de **malloc(int)**. Cette fonction renvoie un pointeur sur une zone mémoire réservée et qui compte autant d'octets (ou bytes) que la valeur de l'entier. À noter que si la valeur de retour est nulle, alors il y a eu une erreur d'allocation de la mémoire. Si on oublie de réserver de la mémoire, alors on risque un "segmentation fault" à l'execution.

Exemple d'utilisation: on veut récupérer une chaîne de caractères de moins de 100 caractères entrée au clavier.

```
char *str;
```

```
str=malloc(100);
scanf("%s",str);
```

Redimensionnement de zone allouée La fonction **realloc(void *, int)** va prendre en entrée une zone mémoire déjà allouée et la redimensionner à la taille entière en entrée.

Désallocation de mémoire : free La fonction **free(void *)** prend un pointeur en entrée et désalloue une zone mémoire déjà allouée.

Pour plus de précisions sur ces fonctions : **man -s3 malloc**.

6.4 Exercices

1. Faire un programme qui trie les éléments d'un tableau d'entiers.
2. Faire un programme qui regarde si deux chaînes de caractères sont symétriques l'une par rapport à l'autre.
3. Faire un programme qui copie une chaine de caractères dans une autre en tenant compte de la longueur des deux chaînes et en faisant attention à la mémoire allouée.
4. Expliquer ce que fait le bout de programme suivant, et quels sont les problèmes possibles :

   ```
   char *a;
   char *b;
   while (*(a++)=*(b++)){}
   ```

   (a++ représente a et a=a+1 et la valeur de retour d'une affectation est la valeur de la variable affectée)

Chapitre 7

Tableaux et Chaînes de Caractères

7.1 Types de données : les Tableaux

7.1.1 Simple Utilisation

Un tableau est un type de données qui permet de ranger plusieurs données du même type. Voici comment dfinir un tableau t d'entiers 10 entres :

```
int t[10];
```

Les éléments de t sont alors numérotés de 0 9 (t[0]...t[9])et l'élément i est accessible par t[i]. Exemple de programmation avec un tableau d'entiers:

```
#include <stdio.h>

// retourne le produit de deux entiers
int mult(int a, int b){

return a*b;
}

// calcule les factorielles des entiers entre 0 et 9
int main(){
int t[10];
int i;
// on calcule les valeurs pour les indices entre 0 et 9
for(i=0;i<10;i++){
   if (i != 0){
       t[i]=mult(i,t[i-1]);
   }else {
t[i]=1;
   }
}
// on affiche les valeurs calculees
for(i=0;i<10;i++){
       printf("valeur de t[%d] : %d\n",i,t[i]);
```

```
}
}
```

L'avantage d'avoir utilisé un tableau sur le fait de les avoir juste affichés est que l'on peut réutiliser ces données par la suite.

7.1.2 Utilisation Avance

Tableaux à plusieurs dimensions On peut avoir autant de dimensions que désirées dans un tableau. Exemple :

```
int t[10][20];
```

Ceci représente un tableau de 10 sur 20 entiers dont les éléments sont rpérés par t[i][j].

Utilisation en Arguments de Fonctions Un tableau peut être passé comme paramtre d'entrée d'une fonction. En général on laisse la taille du tableau libre et on entre celle-ci aussi en paramètre. Exemple :

```
float moyenne(int tableau[], int lg){
int i;
float somme=0;
for (i=0;i<lg;i++){
    somme = somme+tableau[i];
}
return (somme/lg);
}
```

Cette fonction pourrait être utilisée dans le programme précédent pour calculer la moyenne des valeurs du tableau précédemment calculé. Dans ce cas l'appel se ferait alors ainsi:

```
printf("%f\n",moyenne(t,10));
```

Mémoire et tableaux Un tableau qui a été déclaré avec ses bornes se verra affecter de la mémoire pour pouvoir ranger ses valeurs. On n'a donc pas besoin d'allouer de la mémoire pour les utiliser.

Les Chaînes de Caractère sont des Tableaux Le type pour les chaînes de caractère est **char *** mais on peut aussi les déclarer commes des tableaux de caractères. Ainsi on l'équivalence entre ces deux déclarations :

```
char *str="coucou\n";
char str[]="coucou\n";
```

Les Arguments du Programme Les arguments tapés sur la ligne de commande lors du lancement du programme peuvent être utilisés dans le programme lui-même, voici comment on définit le main pour réaliser cela :

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

argc représente alors le nombre de d'arguments et **argv** est un tableau de chaînes de caractères contenant ces arguments. Voici un programme qui ne fait qu'afficher ses arguments :

```
#include <stdio.h>

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

printf(" nombre d'arguments %d\n",argc);
for (i=0;i<argc;i++){
   printf(argv[i]);
   printf("\n");
}
}
```

Voici un exemple d'exécution :

```
[oriol@localhost TP5]: a.out 205 505 420
 nombre d'arguments 4
a.out
205
505
420
```

Nous allons maintenant voir ce que signifie le * dans la section suivante.

7.2 Manipulation de Tableaux

Voici quelques algorithmes expliqués et programmés en C destinés à vous aider à programmer des Manipulations de tableau. Ces algorithmes se présentent comme des fonctions ayant en argument d'entrée un tableau et sa taille.

7.2.1 Parcours Simple (ex : Affichage)

Dans ce cas de figure on parcourt tous les éléments du tableau et on leur applique une action s'ils vérifient une condition (ici on les affiche s'ils sont non-nuls).

```
void Afficher(int taille, int tab[]){
int i;

for(i=0;i<taille;i++){
  if (tab[i]!=0) {
     // action a executer
     printf("la valeur du tableau en %d est %d \n",i+1,tab[i]);
  }
}
}
```

7.2.2 Décalage d'une Portion de Tableau

Le but de cette fonction est de réaliser un décalage d'une portion de tableau (ici une case vers la fin du tableau).

```
void Decaler(int taille, int tab[], int indice){
int i;

for(i=taille-1;i>indice;i--){
  tab[i]=tab[i-1];
}
tab[indice]=0;
}
```

7.2.3 Parcours d'Extractions Multiples (ex : Tri simple)

Le but de cette fonction va être ici de trouver le plus grand élément d'un tableau puis de le mettre à la fin et de recommencer sur le reste du tableau (cet algorithme a déjà été donné en correction d'exercice).

```
void Trier(int taille, int tab[]){
int i,j;
int indice_du_max;
int temporaire;

for(i=taille;i>0;i--){// a chaque iteration on a trouve le maximum /// for 1
  indice_du_max=0;
  for(j=1;j<i;j++){// on cherche le maximum /// for 2
     if (tab[j]>tab[indice_max]){
        indice_max=j;
     }// fin if
  }// fin for 2
  // on echange le maximum avec le dernier element
  temporaire=tab[indice_max];
  tab[indice_max]=tab[i-1];
  tab[i-1]=temporaire;

}// fin for 1
}
```

7.2.4 Exercice : le tri à bulle

Le tri à bulle consiste à scruter toutes lesvaleurs d'un tableau en les comparant deux à deux avec leurs voisines et á les échanger si elles ne sont pas bien classées, puis à recommencer jusqu'à ce que tout le tableau soit trié.

7.3 Chaînes de Caractères

Voyons maintenant un exemple de pointeur que l'on connaît déjà : les chaînes de caractères. Le type que nous utilisons pour décrire une chaîne de caractères

est le type **char *** ce qui signifie qu'une chaîne de caractères est un pointeur sur un caractère qui est suivi des autres caractères de la chaîne. Une convention est que les chaînes de caractères doivent se terminer par le caractère nul : **'\0'**.

Pour mieux comprendre fabriquons une petite fonction qui compare deux chaines de caractères passés en entrée et qui retourne 1 si elles sont égales et 0 sinon :

```
int strcmp(char *str1,char *str2){

while (test){
         \\ on va faire le test tant qu'il reste des caracteres
  if (*str1==*str2) {
          \\ si les caracteres son egaux
\\ on continue
str1=str1+1;
str2=str2+1;
                if (*str1=='\0') {
                \\ s'ils sont nuls on est a la fin de la chaine
                       return 1;
                       \\ on sort de la fonction et on retourne 1
                }
          }
          else {
          \\ si les caracteres ne sont pas
          \\ egaux on sort en retournant 0
                return 0;
          }
        }
}
```

Une version un peu plus complete de cette fonction existe dans le package **string.h** (faîtes **man string** pour plus de détails).

On peut donc se demander comment char * peut être égal à un tableau de caractères. La réponse est qu'un tableau de caractères est un pointeur sur un zone réservée de la mémoire. Ainsi le compilateur interprète le déplacement à appliquer pour obtenir le bon élément. Voici un petit exemple:

```
#include <stdio.h>

int  main(){
char truc[10]="progsys";

*truc='g';\\ OUAH comme c'est sale...

printf(truc);
}
```

7.4 Manipulation Avancée de Chaînes de Caractères

Cette section a juste pour but de montrer quelques fonctions qui sont bien pratiques pour manipuler les chaînes de caractères. Ce ne sera qu'une énumération de fonctions suivies d'une brève description de leur utilité.

- strcat(char *dest,char *source) va placer source au bout de dest (sans réserver de mémoire).
- strcmp(char *str1,char *str2) compare str1 et str2 caractère aprés caractère et renvoie 0 s'ils sont tous égaux.
- strcpy(char *dest,char *source) copie source a la place de dest (sans réserver de mémoire.
- size strlen(char *s) renvoie la taille de la chaîne s.

Il en existe bien d'autres que vous pourrez découvrir en faisant **man string** puisqu'elles se trouvent toutes dans **string.h**.

7.5 Exercices : Lecture des arguments d'un Programme

Le but de cet exercice est de réaliser un programme qui prenne en compte des options passées en paramètres au lancement. Ce programme s'appellera **mywc** et comptera le nombre de mots, le nombre de lignes et le nombre de caractères d'un fichier au choix de l'utilisateur. Le source se nommera **mywc.c**

Syntaxe : **mywc nom_fichier** [**-l**] [**-w**] [**-c**] [**-h**]. L'option **-l** fera l'affichage du nombre de lignes. L'option **-w** fera l'affichage du nombre de mots. L'option **-c** fera l'affichage du nombre de caractères. L'option **-h** fera l'affichage d'un résumé des options et de l'utilisation. On pourra combiner les options (**mywc nom_fichier -l -w -c**) et on renverra un message d'erreur pour le cas où les options auront été mal passées. L'option **-h** prendra le pas sur les autres options et sur les erreurs éventuelles.

Chapitre 8

Fichiers

8.1 Manipulation de Fichiers

Plusieurs primitives existent pour manipuler les fichiers, en voici quelques-unes : **fopen, fclose, fputs, fscanf** et **fprintf**

1. **fopen** est une fonction qui prend deux chaine de caractères en entrée : le nom du fichier à ouvrir et le mode d'ouverture et renvoie un descripteur de fichier (un pointeur sur un fichier : FILE *).
2. **fscanf** fait la même chose que **scanf** sauf que cette fonction lit dans un fichier au lieu du clavier.
3. **fprintf** fait la même chose que **prinf** sauf que cette fonction écrit dans un fichier au lieu du clavier.
4. **fputs** va recopier des données dans un fichier passé en argument.

Pour toutes ces fonctions la meilleure définition est donnée par le **man**. Il est à noter que ces fonctions sont des fonctions d'entrée-sortie et donc sont présentes dans **stdio.h**. Exemple :

```
#include <stdio.h>

main(){
FILE *stream;
FILE *stream2;
char str[100];
int b;

stream = fopen("truc.txt","r");
stream2 = fopen("truc2.txt","w");
while(1){
 b=fscanf(stream,"%s",str);
  if (b!=EOF){
      fprintf(stream2,"%s",str);
  }
else break;
}
```

```
fclose(stream);
fclose(stream2);


}
```

Notez que **fscanf** retourne un entier qui est égal à EOF (défini dans stdio.h) si on est en fin de chaine.

8.2 Exercices

Faites un programme qui lit un fichier texte (au choix de l'utilisateur) contenant des mots séparés par des espaces et génère un fichier HTML (dont le nom sera aussi au choix de l'utilisateur) et qui rangera ces mots dans un tableau (dont la largeur sera demandée à l'utilisateur) dans la page HTML générée. Les mots n'excèderont pas 100 caractères. Vous proposerez de visualiser le résultat dans un lecteur HTML (par exemple Netscape Navigator).

Chapitre 9

Signaux et Handlers

9.1 Introduction

Lorsque l'on utilise la commande **kill** on peut envoyer un signal à un processus. Les différents signaux sont numérotés et un numéro correspond à un message envoyé au programme. Si on ne spécifie pas le contraire, le comportement d'un programme à la réception d'un tel signal est défini par défaut. Néanmoins, un comportement peut être défini dans le programme pour traiter tout ou partie des signaux reçus. Le code exécuté par le programme est alors appelé un **handler**. On utilise le même mécanisme que lorsqu'un périphérique veut communiquer avec le système.

9.2 Quelques Signaux Standards

Voici une partie du man (Linux) sur les signaux et leur Action et signification :

Signal	Value	Action	Comment
SIGHUP	1	A	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGABRT	6	C	Abort signal from abort(3)
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe: write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm(2)
SIGTERM	15	A	Termination signal
SIGUSR1	30,10,16	A	User-defined signal 1
SIGUSR2	31,12,17	A	User-defined signal 2
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped

```
SIGSTOP   17,19,23    DEF     Stop process
SIGTSTP   18,20,24     D      Stop typed at tty
SIGTTIN   21,21,26     D      tty input for background process
SIGTTOU   22,22,27     D      tty output for background process

Next various other signals.

Signal       Value     Action   Comment
------------------------------------------------------------------
SIGTRAP        5         CG     Trace/breakpoint trap
SIGIOT         6         CG     IOT trap. A synonym for SIGABRT
SIGEMT       7,-,7       G
SIGBUS      10,7,10      AG     Bus error
SIGSYS      12,-,12      G      Bad argument to routine (SVID)
SIGSTKFLT    -,16,-      AG     Stack fault on coprocessor
SIGURG      16,23,21     BG     Urgent condition on socket (4.2 BSD)
SIGIO       23,29,22     AG     I/O now possible (4.2 BSD)
SIGPOLL                  AG     A synonym for SIGIO (System V)
SIGCLD       -,-,18      G      A synonym for SIGCHLD
SIGXCPU     24,24,30     AG     CPU time limit exceeded (4.2 BSD)
SIGXFSZ     25,25,31     AG     File size limit exceeded (4.2 BSD)
SIGVTALRM   26,26,28     AG     Virtual alarm clock (4.2 BSD)
SIGPROF     27,27,29     AG     Profile alarm clock
SIGPWR      29,30,19     AG     Power failure (System V)
SIGINFO      29,-,-      G      A synonym for SIGPWR
SIGLOST      -,-,-       AG     File lock lost
SIGWINCH    28,28,20     BG     Window resize signal (4.3 BSD, Sun)
SIGUNUSED    -,31,-      AG     Unused signal
(Here - denotes that a signal is absent; there where three
values are given, the first one is usually valid for alpha
and  sparc,  the middle one for i386 and ppc, the last one
for mips. Signal 29 is SIGINFO / SIGPWR on  an  alpha  but
SIGLOST on a sparc.)

The  letters  in  the  "Action"  column have the following
meanings:

A      Default action is to terminate the process.

B      Default action is to ignore the signal.

C      Default action is to dump core.

D      Default action is to stop the process.

E      Signal cannot be caught.

F      Signal cannot be ignored.

G      Not a POSIX.1 conformant signal.
```

9.3 Comment Envoyer un Signal?

La fonction kill (en section 2 du manuel) permet d'envoyer un signal à un processus. Voici son utilisation :

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

int kill(pid_t pid, int sig);
```

9.4 Comment Programmer un Handler?

Pour programmer un handler, il faut définir une fonction qui va traiter le signal reçu par le processus. Cette fonction ne doit rien retourner (**void**) et prendre en entrée un entier (qui prendra la valeur du numero du signal lors de l'appel). Une fois ceci fait, dans le programme s'exécutant, il faut signaler au système que l'on va appeler notre handler pour les signaux désirés. Cette opération est réalisée a l'aide de la fonction **signal** qui a deux arguments : le numéro du signal et un pointeur sur le **handler** (voir exemples). Le **handler** est désormais installé et est fonctionnel.

9.5 Exemple 1 : Inhiber le CTRL+C

Si on veut inhiber le CTRL+C, il faut récupérer le signal émis par cette commande : SIGINT. Si nous voulons sortir proprement, il nous faut aussi utiliser un signal utilisateur ici le signal 10.

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

/** on definit un compteur pour les appels au handler **/
int compteur=0;

/** on definit le handler **/
void handler_manu(int i){
 printf("handler appele %d fois\n",compteur++);
 if (i==10) {
   printf("fin du processus\n");
   exit(0);}
 }
}

/** main **/
main(){
 // on affiche le pid du processus courant
 printf("PID %d\n",getpid());
```

```
 // on attribue aux signaux leur handler
 signal(10,&handler_manu);
 signal(SIGINT,&handler_manu);

 // on entre dans une boucle infinie
 while(1){}
}
```

9.6 Exemple 2 : Processus Zombie et SIGCHLD

Lorsqu'un processus fils se termine, il envoie un signal (**SIGCHLD**) à son père pour lui signifier son arrêt, puis il se met en veille et attend que le père ait traité ce signal ou qu'il meurre. Si ce dernier ne le fait pas, le fils se met en veille et reste présent en mémoire, on l'appelle alors **processus zombie**. Ceci peut devenir important et embêtant lorsque l'on programme un serveur à fort taux de demandes car on eut se retrouver sans numéro de processus disponible (no more process) ou sans mémoire disponible. Voici comment programmer cette notification :

```
/** on definit le handler **/
void handler_zombie(int i){
 printf("un fils vient de mourir");
}

...

signal(SIGCHLD,&handler_zombie);
```

9.7 Exercice

Faire un programme qui permette de tuer un processus dont on donne le numéro en argument

Chapitre 10

Préprocesseur et Librairies

10.1 Le Préprocesseur C

Le compilateur C (que ce soit gcc ou un autre) appelle un autre programme pour traiter des opérations définies dans le code source mais qui ne font pas partie du programme (par exemple l'inclusion d'une librairie). Ce précompilateur, appelé aussi macroprocesseur ou préprocesseur, va traiter les lignes commençant par un #. Il y a trois types d'ordre déstinés au préprocesseur: les macro-définitions, les inclusions de fichiers, la compilation conditionnelle.

10.1.1 Les Macros et Les Macro-définitions

On va ainsi définir des constantes ou des petites fonctions. Les macro-définitions utilisent deux types d'ordre: **define** et **undef**. Le précompilateur remplace alors dans le fichier source les occurences de la commande ou de la fonction. Exemple:

```
#define taille_max 100

char buffer[taille_max];
scanf("%s",buffer)
```

Sera remplacé par:

```
char buffer[100];
scanf("%s",buffer)
```

On peut aussi définir des fonctions comme par exemple:

```
#define determinant(a,b,c) b*b-4*a*c

int a,b,c;
...
if (determinant(a,b,c)<0){
   printf("pas de solution");
} else {
...

}
```

Qui sera remplacé pour la compilation en elle même par :

```
int a,b,c;
...
if ((b*b-4*a*c)<0){
   printf("pas de solution");
} else {
...

}
```

On peut aussi supprimer la définition d'une macro avec la commande undef, exemple :

```
#define taille_max 1000
char str[taille_max];

#undef taille_max
int taille_max=500;
int b;
...
b = taille_max;
...
```

Ceci est remplacé par :

```
char str[1000];

int taille_max=500;
int b;
...
b = taille_max;
...
```

10.1.2 L'Inclusions de Fichiers

Des fichiers peuvent être inclus dans les listings de programmes C de manière à n'écrire qu'une seule fois le même code. Par exemple, si on veut utiliser un ensemble de fonctions dans plusieurs programmes différents. Par exemple on écrit dans un fichier arithmetique.c plusieurs fonctions comme le calcul de la factorielle, du sinus... et que l'on veut utiliser ces fonctions dans deux programmes, par exemple calculette.c et plot.c, on écrira alors dans calculette.c et dans plot.c:

```
#include "arithmetique.c"
```

De même lorsque l'on veut utiliser des fonctions d'entrée-sortie avec l'utilisateur, on écrit:

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

On a donc deux sortes d'inclusions : l'inclusion de fichiers locaux, et l'inclusion de librairies standard fichiers (présents dans /usr/include). Les premiers ont leur nom délimité par des guillemets et les deuxièmes sont délimités par $<$ et $>$.

10.1.3 Les Directives Conditionnelle

Lorsque le précompilateur est lancé on peut aussi ne compiler que certaines lignes de code pour le cas ou certaines valeurs seraient ou pas définies. Pour cela on utilise les directives de compilations **if, ifdef, ifndef, else, elif** et **endif**. Voici la syntaxe générale de ces directives :

#directive de test (**if, ifdef** ou **ifndef**)

lignes de code compilées si le test est vérifié

#else

lignes de code compilées si le test n'est pas vérifié

#endif

La commande **if** attends une valeur entière du même style qu'un test en C, mais effectué sur une variable de macro. **ifdef** est un **if** qui teste la définition d'une macro et **ifndef** teste la non-définition d'une macro. À la place du **else** on peut mettre **elif** qui symbolise un elsif. Exemples :

Dans un fichier info.macro on a

```
#define PLATEFORME_STR "Linux"
#define PLATEFORME Linux
#define AUTEUR_STR "Manu"
#define AUTEUR Manu
#define VERSION_STR "1.0betaTest"
#define VERSION 1.0betaTest
```

et dans un fichier menu.c:

```
#include "info.macro"
#include <stdio.h>

main(int argc,char *argv[]){
#if PLATEFORME==Linux
char *browser="netscape";
#elif PLATEFORME==Windows
char *browser="IExplorer";
#else
char *browser="inconnu";
#endif

//traitement des arguments
if (argc != 1){
if ((argc == 2)&&(strcmp("-help",argv[1])==0)){
     printf("menu version %s pour %s par %s utilisant %s\n",
VERSION_STR,PLATEFORME_STR,AUTEUR_STR,browser);
     exit(0);
}}
//suite du programme...

}
```

Pour plus de détails, faites **man cpp**.

10.2 Headers et Compilation en Fichiers Séparés

10.2.1 Headers

Un header est un fichier dans lequel on place les squelettes des fonctions d'un autre fichier. Traditionnellement, les headers reçoivent l'extension **.h** alors que les fichiers sources reçoivent l'extension **.c**. Ces fichiers peuvent alors être inclus dans d'autres pour utiliser le code compilé correspondant. Si on veut obtenir une meilleure idée de l'opération voici un petit schéma résumant le principe de la compilation (voir figure 10.1).

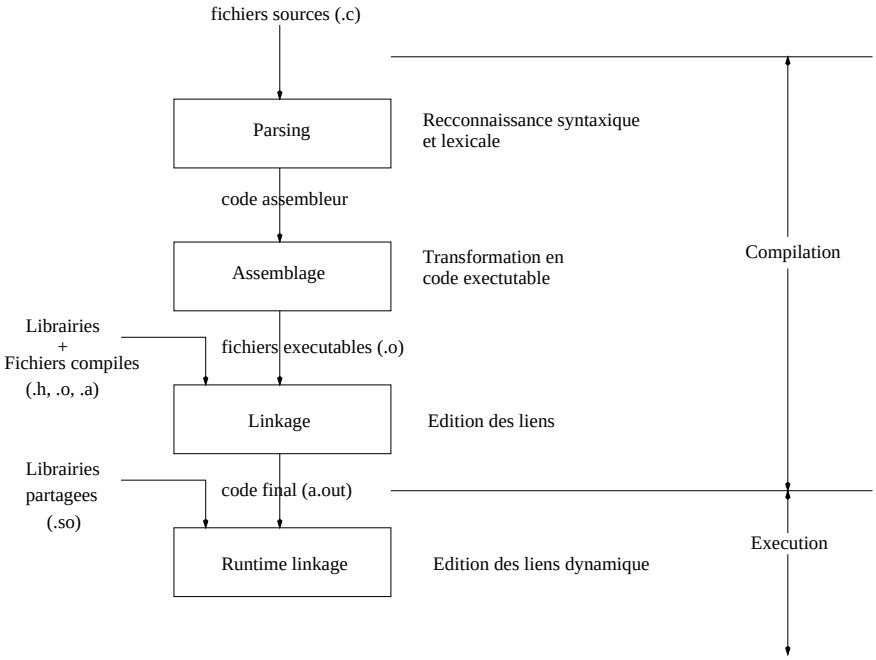

FIG. 10.1 *Différents Stades de la Compilation.*

Pour donner du code à quelqu'un et qu'il puisse l'utiliser sans problème, il suffit de lui donner le code compilé pour la plateforme spécifique (**.o**) et de lui fournir le header à inclure dand ses programmes utilisant le code compilé (**.h**).

10.2.2 Génération de Code et Utilisation

Un **header** est construit de la manière suivante : on identfie les fonctions que l'on veut mettre à la disposition de l'utilisateur et on ne garde que la première ligne de sa définition (en la terminant à '**)**' et en rajoutant **;**. Exemple :

```
[oriol@linoriol]% more Seminaire10.c
#include "Seminaire10.h"
int x=0;

/* fonction effectuant la trace */
```

```
void trace(){
 printf("Utilisation %d de Seminaire 10\n",x);
}

/* fonction additionnant 2 entiers*/
int plus(int a, int b){
 trace();
 return a+b;
}

/* fonction soustrayant 2 entiers*/
int moins(int a, int b){
 trace();
 return a-b;
}
```

```
[oriol@linoriol]% more Seminaire10.h
#include <stdio.h>

/* fonction ajoutant 2 entiers */
int plus(int a, int b);


/* fonction soustrayant 2 entiers */
int moins(int a, int b);
```

Pour générer le code exécutable il faut utiliser l'option -c. Exemple :

```
[oriol@linoriol]% gcc -c Seminaire10.c -o Seminaire10.o
```

L'avantage de ne fournir que le code exécutable est que l'on peut ainsi ne pas se soucier d'être copié par la personne à qui on l'a donné.

L'utilisation, elle est assez simple. Il suffit alors d'inclure le header et de compiler en utilisant le code compilé. Exemple :

```
[oriol@linoriol]% more truc.c
#include "Seminaire10.h"

main(){

plus(1,2);
moins(2,6);

}
[oriol@linoriol]% gcc truc.c Seminaire10.o -o truc
[oriol@linoriol]% truc
Utilisation 0 de Seminaire 10
Utilisation 1 de Seminaire 10
[oriol@linoriol]%
```

À chaque modification du header ou du code source associé, il faut tout recompiler. à chaque modification du programme client, il ne faut que recompiler celui-ci.

10.3 Construction de Librairies (ou bibliothèques)

10.3.1 Création

Une librairie est une archive qui réunit un ensemble de fichiers compilés (**.o**) en un seul fichier (généralement avec l'extension **.a**). Pour réaliser ceci on utilise la commande **ar**.

Commande	Fonction
ar *clé nom_arch nom_executables...*	manipule une archive (r pour remplacer la fonction et c pour créer l'archive)

Exemple d'utilisation : (utilisant l'exemple précédent)

```
[oriol@linoriol]% ar -rc libSeminaire10.a Seminaire10.o Seminaire9.o
```

À partir de ce moment on n'a plus besoin de Seminaire10.o et Seminaire9.o.

10.3.2 Utilisation de Librairies

Pour utiliser une librairie il faut alors compiler en indiquant son chemin d'accès en le mettant dans la variable d'environnement **LIBRARY_PATH** puis compiler en utilisant l'option **-l**. On peut aussi indiquer le chemin en utilisant **-L**. Exemple :

```
[oriol@linoriol]% gcc truc.c -o truc -L~oriol/lib/ -lSeminaire10
```

Ou encore :

```
[oriol@linoriol]% setenv LIBRARY_PATH ~oriol/lib/
[oriol@linoriol]% gcc truc.c -o truc -lSeminaire10
```

Notez que le nom donné est partiel (Seminaire10 pour libSeminaire10.a). Celui-ci est reconstruit par le compilateur.

10.3.3 Librairies Dynamiques

Les librairies (ou bibliothèques partagées) sont des librairies comme les autres sauf qu'elles ne sont pas incluses dans le fichier executable. Elles sont liées lorsque le programme est lancé. L'avantage est que les programmes sont ainsi plus petits et peuvent mieux répondre à la plateforme locale lorsqu'ils sont transportés. L'inconvénient est que le code n'est pas autosuffisant. Ces librairies adoptent généralement un suffixe **.so** et sont générées grâce à la commande **ld** avec les options **-G** et **-o**.

Commande	Fonction
ld -G -o *nom_arch nom_executables...*	créé une archive partagée

Exemple :

```
[oriol@linoriol]% ld -G -o libSeminaire10.so Seminaire10.o
```

Le fichier utilisant cette librairie est compilé **-L** pour lui indiquer l'emplacement de la librairie si cet emplacement n'est pas dans la variable **LD_LIBRARY_PATH**.

Exemple :

```
[oriol@linoriol]% gcc truc.c -o truc -L~oriol/lib/ -lSeminaire10
```

Ou encore :

```
[oriol@linoriol]% setenv LD_LIBRARY_PATH ~oriol/lib/
[oriol@linoriol]% gcc truc.c -o truc -lSeminaire10
```

Mentionnons la commande **ldd** qui permet de permet de vérifier les dépendances dynamiques (**man ldd** pour plus de détails).

10.4 Appels Externes

Pour appeler un programme extérieur à travers un interpréteur shell vous pouvez inclure la librairie **stdlib** en incluant stdlib.h et utiliser **system(const char *)**. Par exemple pour appeler xcalc dans votre calculette vous pouvez ajouter un choix et la ligne suivante :

```
system("xcalc");
```

10.5 Exercice : la librairie $Z/(nZ)$

L'Exercice de ce chapitre est de construire une petite librairie qui fera des calculs sur les nombres modulo n. La librairie deva comporter les fonctions suivantes :

int plus(int a,int b, int mod) valeur de la somme de 2 modulos.

int moins(int a, int b, int mod) valeur de la différence de 2 modulos.

int mult(int a, int b, int mod) valeur du produit de 2 modulos.

int divise(int a, int b, int mod) valeur de la division de 2 modulos.

int valeur_modulo(int a, int mod) valeur du modulo de l'entier.

int valeur_modulo_f(float a, int mod) valeur du modulo de la partie entière du float.

Ces fonctions seront intégrées dans une librairie statique (**libmodN.a**) et vous fournirez un **makefile** qui compilera votre fichier. L'usage de toute fonction vous calculant directement le modulo sera proscrit : vous devrez le faire vous-même.

Chapitre 11

Programmation Parallèle

11.1 Programmation Parallèle : fork()

11.1.1 fork()

La primitive **fork()** permet de créer un nouveau processus qui exécute le même code que l'ancien. Toutes les zones mémoires sont alors réallouées et les deux processus sont complètement indépendants. Notez que, lors de la création du nouveau processus, **fork()** retourne 0 dans le fils et retourne le numéro du processus fils dans le père, elle retourne -1 dans le cas ou cela s'est mal passe. Voici un squelette d'utilisation de **fork()** :

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

int main(){
int pid;


 if (pid=fork()){
// partie du pere

 }
 else {
        //partie du fils ou partie de mauvaise execution
 }
}
```

Un petit exemple d'utilisation peut être un processus qui attend une réponse pour effectuer quelque chose et l'autre qui fait quelque chose d'autre, exemple:

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

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

```
int main(){
int i;
int pid;
char c;

 if (pid=fork()){
   printf("voulez-vous tuer le fils?");
   scanf("%c",&c);
   if (c=='o') {
      kill(pid,SIGKILL);
   }
 }
 else {
   while(1) {
     printf("%d",i++);}
 }
}
```

Dans cet exemple, comme dans la ligne de commande Unix on a pu envoyer un signal pour tuer le processus fils au moyen de la primitive **kill(pid,signal)** (en section 2 du manuel). Quelques autres primitives peuvent être importantes : **getpid()** et **getppid()** retournent respectivement le numéro du processus courant et de son parent.

Le problème demeure comment faire communiquer 2 processus créés par un **fork()**.

11.1.2 pipe

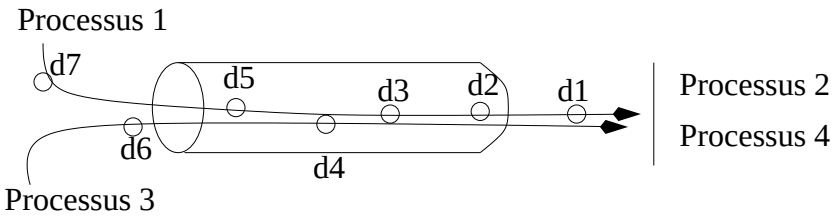

FIG. 11.1 *Exemple de pipe : Processus 1 et 3 nourissant le pipe lu par 2 et 4*

La primitive **pipe** va permettre de créer un canal de communication entre plusieurs processus (voir figure 11.1). Voici le squelette de son utilisation :

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


int main(){
int i=0;
int pid;
char c;
```

```
int p[2];

 pipe(p);
 if (pid=fork()){
   char *str ="message";
   write(p[1], str, 8);

 }
 else {
   char str[100];

   read(p[0], str, 8);
   printf(str);
 }
}
```

Les **descripteurs de fichiers** sont stockés dans un tableau d'entiers à deux entrées : p[0] correspond alors au descripteur en sortie du tube (lecture) alors que p[1] correspond au descripteur en entrée (écriture). On utilise alors les commandes **read** et **write** pour lire ou écrire sur un descripteur de fichier un **buffer** (pointeur sur une zone mémoire) du nombre d'octets (caractères) désiré. Nous verrons plus avant ces notions par la suite.

11.2 Exercices

1. Réaliser un programme qui réalise un fork et joue au ping-pong avec des messages envoyés sur un pipe.
2. Réaliser un programme qui lise des entrées au clavier dans un processus et qui fasse passer les données lues à son fils pour que celui-ci les affiche.

Chapitre 12

Typage

12.1 Typage et Structures

Le typage est un mécanisme qui a été introduit dans le but de garantir certaines propriétés aux programmeurs. Ceci est dû au fait que certains langages ont une syntaxe qui prête à confusion dès la première erreur de frappe. La communauté informatique s'est donc mis dans l'idée qu'il n'est pas très sage d'ajouter des entiers à des chaînes de caractères ou encore des entiers longs à des tableaux. Les langages fortements typés sont alors apparus et avec eux des propriétés de sécurité de programmation. C fait partie des langages qui ne sont pas fortement typés car il effectue lui-même de nombreuses conversions de type (des *cast*). Néanmoins le compilateur signalera souvent que certaines affectations lui semblent abusives : C est donc un langage typé.

Pour effectuer soi-même une conversion de type on peut indiquer entre parenthèses devant la donnée le type dans lequel on la convertit. Ceci n'est valable que si le compilateur a un moyen de convertir les données. Exemples :

```
int a=3;
double b;

b=(double)a;
```

ou encore

```
char *chaine;

chaine = (char *)malloc(100);
```

Une manière élégante de programmer est parfois de définir soi-même ses propres types. Le mot clé **typedef** va nous servir à définir un nouveau type alors que le mot clé **struct** peut-être utilisé pour composer des types. Si nous voulons définir un type compteur qui sera un entier voici comment le faire :

```
typedef int compteur;
```

On peut ensuite utiliser ce type pour définir des variables :

```
compteur b;
```

Imaginons que l'on veuille construire un type pour les rationnels, on pourra définir un type *rationnel* dans lequel on aura 2 champs *dividende* et *diviseur* entiers. Nous allons utiliser le mot clé **struct** pour définir ce nouveau type :

```
struct rationnel {
int dividende;
int diviseur;
};
```

Une fois cette structure définie on peut l'utiliser pour définir un type portant un nom quelconque (éventuellement le même).

```
typedef struct rationnel rationnel;
typedef struct rationnel quotient;
```

Pour définir des variables de ce type on utilise alors :

```
rationnel a;
```

L'accession aux champs de la structure se fait en écrivant le nom de la variable définie suivi d'un point et du nom du champ (comme on le ferait pour des classes en Java) :

```
a.dividende=3;
a.diviseur=2;
```

ou encore :

```
float approx_float(rationnel rat){
return (rat.dividende/rat.diviseur);
}
```

Cette manière de programmer est donc élégante lorsqu'on a des variables qui ont une certaine unité.

12.2 Exercice

Créez un type qui représente l'identité d'une personne et construisez quelques fonctions qui utiliseront ce type.

Chapitre 13

Conclusion

ceci est la conclusion

Annexe A

Algorithmique

A.1 Notion d'Algorithmique

Un programme, quel qu'il soit, a pour but d'automatiser un certain nombre de tâches difficiles à réaliser pour un être humain. Cette fonction oblige donc le programmeur à savoir construire ses programmes de telle sorte qu'ils répondent à ses attentes. Le programme réalisé est alors la traduction d'une suite de tâches appelée **algorithme** qui composent la tâche finale. La science qui étudie la manière dont on construit cette suite de tâches s'appelle **algorithmique** et c'est elle que nous allons étudier dans ce séminaire.

Réponddez à ces quelques questions :

1. Décrivez les différentes actions de manière détaillée que vous devez faire pour manger au self.
2. Décrivez comment marche votre ascenceur préféré.
3. Décrivez comment vous réalisez une sauce à salade.

Une fois ces questions traitées, écrivez-les en utilisant la syntaxe du langage C.

La programmation en général est ainsi faite : vous vous posez un problème et vous essayez de le résoudre. À votre disposition vous avez des méthodes et des structures de contrôle de votre algorithme et vous devez passer d'un état initial à un état final d'une certaine manière.

Je vous propose donc la méthode suivante pour construire vos programme :

1. Prenez une feuille de papier et écrivez la fonction générale de votre programme (Ex : ascenceur).
2. Identifiez les unités logiques de votre programme (Ex : une montée et une descente).
3. Identifier des unités logiques à l'intérieur de vos unités logiques et cela tant que vous n'aurez pas des unités indivisibles.
4. Lorsque vous avez chaque unité logique partez des plus grandes, allez vers les plus petites et indiquez dans chaque plus grande comment vous combinez les plus petites.
5. Décidez des structures de programmes à employer pour réaliser vos combinaisons. A aucun moment ne rentrez dans les détails.
6. En vous aidant du produit de l'étape 5 codez en C!

A.2 Récursivité

A.2.1 Principe

La récursivité est une technique de programmation qui consiste en l'appel d'une fonction dans celle-ci. Cette technique est très utilisée dans les languages fonctionnelle comme Lisp ou Skim car elle est la base de ces langages. Pour que cela marche, il faut toujours prévoir une condition d'arrêt (comme dans un **while**).

A.2.2 Exemple

Dans ce cadre, un problème intéressant est le calcul des factorielles (et des suites et séries en général). Une factorielle (!) est définie comme suit :

```
0! = 1
n! = n*((n-1)!)
```

Cette définition donne donc :

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
...
```

Une solution naturelle (et élégante) à ce problème, tel qu'il est posé peut donc être codée comme suit :

```
int factorielle(int n){
if (n==1) return (1);

return (n*factorielle(n-1));
}
```

A.3 Exercices :

1. Faîtes un programme qui fasse la moyenne d'un tableau d'entiers.
2. Faîtes un programme qui trie un tableau d'entiers.
3. Faîtes un programme qui construise un fichier texte contenant toutes les entrées d'un tableau d'entier.
4. Faîtes un programme qui puisse relire ce fichier et reconstruire le tableau.

Annexe B

Écriture de Code

Le but de cette annexe est de donner quelques conseils quand à l'écriture des listings.

B.1 Commentaires

Les commentaires sont indispensables : il ne doit pas y avoir plus de 5 lignes sans commentaire. Exemple :

```
void Trier(int taille, int tab[]){
int i,j;
int indice_du_max;
int temporaire;

for(i=taille;i>0;i--){
  indice_du_max=0;
  for(j=1;j<i;j++){
     if (tab[j]>tab[indice_du_max]){
        indice_du_max=j;
     }
  }
  temporaire=tab[indice_du_max];
  tab[indice_du_max]=tab[i-1];
  tab[i-1]=temporaire;

}
}
```

B.2 Noms de Variables

Les noms de variables doivent le plus possibles être explicite. Comme en math les indices seront i,j,k... et le reste aura un nom approprié. Exemple :

```
/** cette fonction trie un tableau d'entiers passe en argument **/
void FDurelk(int a, int gjh[]){
```

```
int rt,du;
int ih;
int opb;

for(rt=a;rt>0;rt--){// a chaque iteration on a trouve le maximum
  ih=0;
  for(du=1;du<rt;du++){// on cherche le maximum
     if (gjh[du]>gjh[ih]){
        ih=du;
     }
  }
  // on echange le maximum avec le dernier element
  opb=gjh[ih];
  gjh[ih]=gjh[rt-1];
  gjh[rt-1]=opb;

}
}
```

B.3 Indentation

La mise en page de votre listing doit refleter les structures de controle que vous utilisez : indentez donc le code de celles-ci. Exemple :

```
void Trier(int taille, int tab[]){
int i,j;
int indice_du_max;
int temporaire;
for(i=taille;i>0;i--){// a chaque iteration on a trouve le maximum
indice_du_max=0;
for(j=1;j<i;j++){// on cherche le maximum
if (tab[j]>tab[indice_du_max]){
indice_du_max=j;}}
// on echange le maximum avec le dernier element
temporaire=tab[indice_du_max];
tab[indice_du_max]=tab[i-1];
tab[i-1]=temporaire;}}
```

B.4 Indicateurs Lexicaux

Moins important que les autres ce point mérite tout de meme de s'y attarder : utilisez la même norme de représentation pour les mêmes structures de contrôle. Exemple :

```
/** fonction de tri **/
void Trier(int taille, int tab[]){
int i,j;
// definition de l'indice maximum
int indice_du_max;
int temporaire;
```

```
/** a chaque iteration on a trouve le maximum **/
for(i=taille;i>0;i--){
  indice_du_max=0;
  for(j=1;j<i;j++){// on cherche le maximum
     if (tab[j]>tab[indice_du_max]){
        indice_du_max=j;
     }
  }
  // on echange le maximum avec le dernier element
  temporaire=tab[indice_du_max];
  tab[indice_du_max]=tab[i-1];
  tab[i-1]=temporaire;

}
}
```

B.5 Combinaison de l'Ensemble

```
/** **/
void FDurelk(int a, int gjh[]){
int rt,du;
int ih; ////////////////////////////
int opb;
for(rt=a;rt>0;rt--){
ih=0;

/** **/
for(du=1;du<rt;du++){
if (gjh[du]>gjh[ih]){////////////////////////////////
ih=du;}}

/** **/
opb=gjh[ih];
gjh[ih]=gjh[rt-1];////////////////////////////
gjh[rt-1]=opb;}}
```

Annexe C

Make et Makefile

C.1 Makefiles et make

Comme on peut le voir, toutes ces dépendances entre librairies et programmes différents demandent d'écrire des lignes complexes juste pour compiler un programme un tant soit peu complexe. Des outils ont donc été développés pour permettre de réaliser ces tâches de manière autonomes. La commande **make** est, de loin, l'utilitaire le plus répandu pour réaliser ceci sous Unix. Du fait de ses nombreuses possibilités et de l'expressivité de ses fichiers de dépendances on pourrait écrire (et on l'a déjà fait!) de véritables petits livres à ce sujet. Dans cette section nous n'allons donc qu'effleurer le sujet tout en vous permettant déjà d'utiliser cet outil.

La commande **make** est un programme qui va, par défaut lire le fichier **makefile** et ensuite **Makefile** si celui-ci n'existe pas.

Une construction simple de ces fichiers est la suivante, chaque dépendance est codée ainsi :

```
nom_dependance : dependances_dont_elle_depend
```

Les lignes suivantes seront exécutées si elles commencent par le caractère **tab**. Sinon elles seront considérées comme de nouvelles dépendances. Les lignes commençant par # sont des **commentaires**. Exemple

```
[oriol@linoriol]% more makefile
# dependance par defaut
 : libSeminaire10.a

# construction de la librairie
libSeminaire10.a : Seminaire10.o
ar -rc libSeminaire10.a Seminaire10.o

# compilation du fichier
Seminaire10.o : Seminaire10.c
gcc -c Seminaire10.c -o Seminaire10.o
# compilation du fichier
Seminaire9.o : Seminaire9.c
gcc -c Seminaire9.c -o Seminaire9.o
```

Par défaut la première dépendance est exécutée (: libSeminaire10.a). Celle-ci appelle la deuxième (libSeminaire10.a : Seminaire10.o) si Seminaire10.o est plus ancien que libSeminaire10.a alors on exécute la ligne suivante dépendance. Si Seminaire10.o est plus ancien que Seminaire10.c alors on compile.

Pour faciliter l'écriture de ces règles certains racourcis sont bien utiles dans les commandes exécutées : $@ représente le nom de la dépendance et $? représente les dependances_dont_elle_depend. Exemple :

```
[oriol@linoriol]% more makefile
 : libSeminaire10.a
libSeminaire10.a : Seminaire10.o Seminaire9.o
ar -rc $@ $?
Seminaire10.o : Seminaire10.c
gcc -c $? -o $@
Seminaire9.o : Seminaire9.c
gcc -c $? -o $@
```

Pour réaliser une dépendance il faut exécuter **make** avec comme argument le nom de cette dépendance. Exemple :

```
[oriol@linoriol]% make Seminaire10.o
```

On peut utiliser des variables en effectuant (au début)

```
nom_var= valeur
```

et y faire référence en utilisant $(nom_var). Exemple :

```
[oriol@linoriol]% more makefile
# parametres
CONTENU_LIB= Seminaire10.o Seminaire9.o
COMPILER=gcc

# construction de la librairie
libSeminaire10.a : $(CONTENU_LIB)
ar -rc $@ $?

# compilation du fichier
Seminaire10.o : Seminaire10.c
$(COMPILER) -c $? -o $@

# compilation du fichier
Seminaire9.o : Seminaire9.c
  $(COMPILER) -c $? -o $@
```

Annexe D

Debuggage

D.1 Tra cage des Résultats

La technique la plus simple est encore de fournir une trace des résultats partiels de l'exécution d'un programme. Ceci signifie que l'on va inclure dans le code des **println** un peu partout et contrôler les résultats lors d'exécutions successives. Une bonne idée est d'inclure une option -v (v pour verbose) au programme et d'inclure ces **println** dans des **if** qui testent si l'option a été prise et affiche les traces dans ce cas-là.

D.2 Debugger : gdb

Des débuggers existent pour tous les langages et toutes (ou presque) les plateformes. certains sont plus conviviaux que d'autres. **gdb** est le débugger standard de GNU (Gnu is Not Unix). Il peut être appelé dans **xemacs** et vous permet d'aller de ligne en ligne ainsi que d'afficher l'état des variables de mettre des points d'arrêt. Dans **xemacs** un certain nombre de tâches sont automatisées. néanmoins, on peut l'utiliser en mode texte et voici quelques commandes utilies :

gcc -g truc.c pour rendre le code exécutable utilisable par gdb.

gdb a.out pour lancer le debugger.

et une fois dans gdb :

break (fichier.c) (ligne ou procédure) installe un point d'arrêt. (Exemple : break main)

run (arguments) lance l'exécution.

print (expression) permet de voir la valeur d'une expréssion (Exemple : print a).

up permet de retourner dans la fonction appelante.

down permet de descendre vers la fonction appelée.

Ce debugger n'est pas le plus beau ni le plus explicite mais il est le plus standard. D'autres debuggers sont construits sur le même modèle (par exemple **jdb** pour **Java**). Combiné à **xemacs** il constitue tout de même une bonne solution pour debugger vos programmes.

Annexe E

Correction des Exercices

E.1 Chapitre 2 : Introduction à Unix

1. Utiliser un éditeur de texte comme xemacs ou l'éditeur par défaut sous Solaris et sauvegarder (`exemple.txt`).
2. Solution (à taper dans un terminal):

   ```
   mkdir exempledir
   mv exemple.txt exempledir/.
   ```
3. Sauvegarder une image du web avec Netscape (click droit sur l'image, sauver). Puis taper dans un terminal `xv nomimage`
4. Solution (à taper dans un terminal):

   ```
   cd ~oriol
   ls
   cd public_html
   ls
   ```
5. Les droits d'accès au répertoire Sources_ecrits sont interdits en exécution pour tout autre que moi, on ne peut donc pas y pénétrer.

E.2 Chapitre 3 : Commandes

- find permet de trouver un fichier en spécifiant partiellement son nom.
- sed permet de filtrer un flux de données ligne par ligne et de procéder à des remplacements de séquences de caractères reconnues.
- grep permet de ne garder que les lignes contenant une séquence de caractères dans un flux de données.
- kill permet de manipuler un processus en lui envoyant des signaux (cf chapitre 8).
- rlogin permet de se loger sur une machine distante et de travailler sur celle-ci (dans un terminal) comme en local.
- xterm lance un termina spécifique.
- tar permet de créer des archives (regroupements de fichiers).
- awk permet de manipuler des flux de données.

- top permet de voir quels sont les différents processus qui utilisent le plus le tems machine sur la machine locale.
- telnet ressemble à rlogin mais est aussi implémenté sous Windows en standard.
- history permet de récupérer la liste des dernières commandes (celles dans l'historique)
- man permet de ire le manuel sur une commande précise.
- cd, pushd, popd, dirs, chdir permet de naviquer dans les répertoires.
- ls permet de lister le contenu d'un répertoire, par défaut le répertoire courant.
- last montre la liste des dernières personnes logées sur la machine locale.
- ps montre les processus (programmes) tournant actuellement sur la machine locale.
- bg, fg, jobs, stop, notify permet de mettre des processus en attente, de les arreter, de les mettre au premier plan dans le terminal courant.
- who (am i) donne les personnes logées sur la machine locale.
- csh interpréteur de C-shell.
- touch met la date de dernière modification à la date courante (par défaut).
- banner affiche en grandes lettres le texte passé en arguments.
- alias permet de créer u raccourcis vers une commande pour le shell courant.

E.3 Chapitre 4 : Programmation en C, Découverte

1. Le programme se plante car 99999999999 est plus grand que le maximum un entier, il boucle donc et retourne une valeur inadaptée.
2. Solution :

```
#include "stdio.h"
int main(){
        int i, j;

        printf("Entrez le premier nombre :");
        scanf("%d", i);

        printf("Entrez le deuxieme nombre :");
        scanf("%d", j);

        printf("Le r\'esultat est %d", i+j);

}
```

3. Solution :

```
#include "stdio.h"
int main(){
        int i, j;

        printf("Entrez le premier nombre :");
        scanf("%d", i);
```

```
        printf("Entrez le deuxieme nombre :");
        scanf("%d", j);

        printf("Le r\'esultat est %d", i*j);

}
```

4. Solution:

```
#include <stdio.h>

/** ce programme va permettre de calculer l'age de mon chien

**/



void main(){
        int date;// variable qui va servir a stocker la date du jour
        int date_chien=25101994; // mon chien est ne en 94 (le 25 octobre)
        int jour,mois;
        int jour_chien,mois_chien,annee_chien;
        int jour_temp,mois_temp,annee_temp;

        printf("entrez la date courante (jjmmaaaa)\n");
        scanf("%d ",&date);

        // on met a jour les variables a traiter...
        jour_chien = date_chien/1000000;
        jour_temp = date/1000000;
        jour = jour_temp - jour_chien; // on truande pour
        //ne pas s'embeter avec les mois a 30,
        //31 jours et autres annees bisextiles

        // on enleve les jours des deux dates
        date_chien = date_chien - jour_chien*1000000;
        date = date - jour_temp*1000000;

        // on calcule les mois
        mois_chien = date_chien/10000;
        mois_temp = date/10000;
        mois = mois_temp-mois_chien;

        // on corrige pour que le mois soit positif
        if (mois<0){
                mois = mois + 12;
                date =date - 1;
        }

        // on enleve les mois aux deux dates
        date = date - mois_temp*10000;
```

```
            date_chien = date_chien - mois_chien*10000;

            // on a toutes les informations donc on peut afficher
            printf(" Mon chien a %d annees, %d mois et%d jours\n"
                    ,date-date_chien, mois, jour);

    }
```

E.4 Chapitre 5 : Structures de Programmation

```
#include <stdio.h>

float resultat_precedent=0.0;
char *version="Calculette v0.000000001\n";


/* fonctions permettant de realiser les operations*/
float plus(float a,float b){
        return a+b;
}


float moins(float a,float b){
        return a-b;
}


float fois(float a,float b){
        return a*b;
}


float diviser(float a,float b){
        return a/b;
}


float factorielle(){

        int i,a;
        float b=1.0;

        printf("entrez le nombre entier:");
        scanf("%d",&a);

        for (i=1;i<(a+1);i++){
                b=b*i;

        }
```

```
        return b;
}


/*va proposer le menu et realiser le scanf si on ne reutilise pas le resultat*/

float operation_standard(){

        float a,b;

        char c;
        printf("entrez l'operation a calculer :");
        scanf("%f%c%f",&a,&c,&b);

        switch(c){
                case '+' : {
                        return plus(a,b);
                }

                case '-' : {
                        return moins(a,b);
                }

                case '*' : {
                        return fois(a,b);
                }

                case '/' : {
                        return diviser(a,b);
                }

                default :{
                        printf("Operation non reconnue, attendez la prochaine version...\n");
                }
        }

}


/* simple copier/coller de la fonction
 * precedente avec premier argument reutilise
 */
float operation_argument1(){
        float a,b;
        char c;

        a=resultat_precedent;
        printf("entrez l'operation a calculer : %e",a);
        scanf(" %c%f",&c,&b);
```

```
        switch(c){
                case '+' : {
                        return plus(a,b);
                }
                case '-' : {
                        return moins(a,b);
                }
                case '*' : {
                        return fois(a,b);
                }
                case '/' : {
                        return diviser(a,b);
                }
                default :{
                        printf("Operation non reconnue, attendez la prochaine version...
                }
        }
}



/* simple copier/coller de la fonction
 * precedente avec deuxieme argument reutilise
 */
float operation_argument2(){

        float a,b;
        char c;

        b=resultat_precedent;
        printf("Ans =%e\n",b);
        printf("entrez le premier argument, l'operation et Ans (ex: 135+Ans): ");
        scanf(" %f%cAns",&a,&c);
        switch(c){
                case '+' : {
                        return plus(a,b);
                }
                case '-' : {
                        return moins(a,b);
                }
                case '*' : {
                        return fois(a,b);
                }
                case '/' : {
                        return diviser(a,b);
                }
                default : {
                        printf("Operation non reconnue, attendez la prochaine version...
                }
        }
```

```
}


/* programme principal */

int main(){

        char c;
        int choix=1;

        printf(version);
        while (choix){

/*menu*/

        printf("Choisissez dans le menu suivant :
                1 Operation simple standard (+,*,-,/)
                2 Calculer une factorielle
                3 Reutiliser le resultat precedent en premier argument
                4 Reutiliser le resultat precedent en deuxieme argument
                0 Quitter\n");

        scanf("%d",&choix);

        switch (choix){
                case 1 : {// cas d'operations a 2 arguments
                        resultat_precedent=operation_standard();
                        printf("resultat : %e (c pour continuer)\n",
                                resultat_precedent);
                        scanf(" %c",&c);
                        break;
                }

                case 2 : {//cas de la factorielle
                        resultat_precedent=factorielle();
                        printf("resultat : %e (c pour continuer)\n",
                                resultat_precedent);
                        scanf(" %c",&c);
                        break;
                }

                case 3 : {
                        //cas de reutilisation du resultat en tant qu'argument 1
                        resultat_precedent=operation_argument1();
                        printf("resultat : %e (c pour continuer)\n",
                                resultat_precedent);
                        scanf(" %c",&c);
                        break;
                }
```

```
                case 4 : {
                        //cas de reutilisation du resultat en tant qu'argument 2
                        resultat_precedent=operation_argument2();
                        printf("resultat : %e (c pour continuer)\n",
                                resultat_precedent);
                        scanf(" %c",&c);
                        break;
                }

                case 0 : {//quitter
                        printf("Au revoir...\n");
                        break;
                }

                default : {//probleme de saisie
                        printf("operation impossible!!!!\n");
                }
        }
        }
}
```

E.5 Chapitre 6 : Pointeurs et Mémoire

1. Solution :

```
#include <stdio.h>

/** fonction permettant de trier un tableau passe en parametre avec sa taille

**/

void trier(int b[],int c){


// max va permettre de stocker des resultats temporaires...

int max,i,j,j_max;


// l'algorithme de tri est assez rapide et comprehensible :
// - on cherche le plus grand entier du tableau de dimension n
// - on l'echange avec le dernier element
// - on recommence mais en ne regardant que les n-1 premiers elements
// - et ainsi de suite
// on utilise i comme borne superieure pour les elements du tableau
// et on va le decrementer au fur et a mesure

for(i=c;i>0;i=i-1){
```

```
        // le premier element est d'abord mis dans max qui sera la valeur maximale
        max=b[0];
        // on garde l'indice du max
        j_max=0;


        // on parcourt les elements du sous tableau pour trouver le plus grand
        for(j=1;j<i;j=j+1){

                if (b[j] > max) {
                        // on sauvegarde son index et sa valeur
                        max=b[j];
                        j_max=j;

                }

        }

        // on a maintenant le plus grand du sous-tableau et on le met dans la
        // case adequate (la derniere) et on met l'element qui y etait a la place libre
        b[j_max]=b[i-1];
        b[i-1]=max;

}
}



/**
* Programme permettant de tester la procedure trier
*
**/

int main(){

        // on definit le tableau et on l'initialise
        int a[10]={10,2,4,9,6,7,8,3,1,5};
        int i;

        // on affiche le tableau de depart
        printf("Tableau de depart :\n");
        for(i=0;i<10;i++)
                printf("a[%d]=%d \n",i,a[i]);

        // on trie le tableau
        trier(a,10);

        // on affiche le tableau trie
        printf("Tableau trie :\n");
```

```
        for(i=0;i<10;i++)
                printf("a[%d]=%d \n",i,a[i]);


}
```

2. Solution :

```
#include <stdio.h>


/**

* Ce programme ecrit 1 si les deux arguments passes en parametres sont des

* versions inversees l'un de l'autre

**/

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

        // on definit les variables
        int taille=0,i,taille2=0;

        // la variable qui donnera le verdict final
        int palyn=1;

        // on teste le nombre d'arguments
        if (argc!=3) {
                printf("Usage : palyndrome word1 word2\n tests if word1==2drow\n");
                exit(0);
        }


        // on calcule la taille du premier argument
        while(argv[1][taille]!='\0'){
                taille=taille+1;
        }


        // on calcule la taille du deuxieme argument
        while(argv[2][taille2]!='\0'){
                taille2=taille2+1;
        }

        // on verifie qu'ils on la meme taille autrement c'est foutu
        if (taille!=taille2){
                printf("0\n");
                exit(0);
        }
```

```
        // on verifie que le premier caractere du premier argument est egale
        // au dernier du deuxieme argument et ainsi de suite

        for(i=0;i<taille;i++)
                if (argv[1][i]!=argv[2][taille-i-1]) palyn=0;


        // on affiche 1 si c'est un palyndrome et 0 sinon
        if (palyn) {
                printf("1\n");
        }

        else printf("0\n");
}
```

3. Solution :

```
#include <stdio.h>


/**
* fonction permettant de copier une chaine en allouant de la memoire
**/


char *copiestr(char *source){

        int taille=0,i;
        char *retour;

        // on calcule la taille de la source
        while(source[taille]!='\0'){
                taille=taille+1;
        }

        // on alloue la memoire necessaire
        retour = malloc(taille+1);

        //on fait la copie en copiant bien le dernier caractere ('\0')
        for(i=0;i<taille+1;i++){
                retour[i]=source[i];
        }

        // on retourne la chaine copie
        return retour;

}

/**
```

```
* main testant la fonction permettant de copier une chaine en allouant de la memoire

**/


int main(){

        // on definit la chaine a copier
        char *chaine = "Chaine pour l'exercice 3 \n";

        // chaine dans laquelle on va copier
        char *chaine2;


        // on affiche la chaine de depart
        printf(chaine);


        //on effectue la copie
        chaine2=copiestr(chaine);

        // on affiche la chaine finale
        printf(chaine2);

}
```

4. a et b sont deux pointeurs sur des chaines de caracteres donc *a et *b representent les zones memoire ou se trouvent les deux chaines de caracteres. *(a++) signifie que l'on accede a *a et que ensuite on incremente a de 1 (on passe au caractere suivant). De meme pour *(b++) donc ce que fait *(a++)=*(b++) est d'affecter la valeur pointee par b a la valeur pointee par a. La valeur de retour de cette affectation est la valeur du caractere affecte. Si celui-ci est egal a 0, alors on sort du while sinon on fait cela sur le caractere suivant et ainsi de suite. Moralite: cette portion de programme execute ce qui etait demande a l'exercice 3 mais sans reserver de memoire... d'ou les problemes possibles...

E.6 Chapitre 7 : Tableaux et chaînes de Caractères

```
/*
 * Programme ecrit par Manuel Oriol (Octobre 2000)
 * oriol@cui.unige.ch
 *
 * */
#include "stdio.h"

// 3 variables globales servant a stocker
// s'il faut calculer les mots, les lignes ou
// les caracteres
```

```
int w=0,l=0,c=0;

/*
 * fonction effectuant la reconnaissance
 * des options
 * Cette fonction retourne 1 si elle a reussi
 * 0 si c'est une option -h ou fausse
 * attention ceci n'est pas 'blinde'
 */
int reconnaissance(int nbarg, char *argv[]){
        // boucle sur les arguments
        while (--nbarg>1){
                // teste si c'est une option
                if (*argv[nbarg]!='-') return 0;
                // regarde le type d'option et
                // met a jour la variable correspondante
                switch (*(argv[nbarg]+1)){
                        case 'w' :
                                w=1;
                                break;
                        case 'l' :
                                l=1;
                                break;
                        case 'c' :
                                c=1;
                                break;
                        default : return 0;
                }
        }

        return 1;
}

/*
 * cette fonction sert a afficher l'usage
 * */
void help(char *nom){
        printf("Usage: %s fic_name\n",nom);
        printf("blablabla...\n");

}

/*
 * fonction principale du programme
 * */
int main(int argc, char *argv[]){

        // i stocke le nombre de lignes
        // j stocke le nombree de mots
        // c stocke le nombre de caracteres
```

```
        int i=0,j=0,k=0;
        // le descripteur de fichier
        FILE *file_descriptor_fic;
        // des caracteres
        char ch,c0=' ';

        // si le nombre d'arguments est trop petit
        // on arrete
        if (argc < 3) {
                help(argv[0]);
                exit(0);
        }
        // si la reconnaissance des options donne
        // un help, on le fait
        if (reconnaissance(argc, argv)==0) {
                help(argv[0]);
                exit(0);
        }
        // on ouvre le fichier
        file_descriptor_fic=fopen(argv[1],"r");
        // on compte les caracteres de certains types
        while(fscanf(file_descriptor_fic,"%c",&ch)!=EOF) {
                if (ch=='\n') i++;
                if (((ch=='\n')
                     ||
                     ((ch==' ')
                     ||
                     (ch=='\t'))) // '\t' est le
                                  // caractere de tabulation
                    &&((c0!='\n')
                       &&((c0!=' ')
                       &&(c0!='\t')))) j++;
                k++;
                c0=ch;
        }
        // on a finit, on referme le fichier
        fclose(file_descriptor_fic);

        // on affiche ce qu'il faut afficher
        if (l) printf("lignes : %d\n",i);
        if (w) printf("mots : %d\n",j);
        if (c) printf("caracteres : %d\n",k);
}
```

E.7 Chapitre 8 : Fichiers

```
// inclusion standard de package
#include "stdio.h"
#include "stdlib.h"
```

```
/*
 * fonction principale du programme
 * */
int main(){
        // On definit les deux descripteurs de fichier
        FILE *fdsource = NULL;
        FILE *fdcible = NULL;

        // Des chaines pour stocker les noms des
        // fichiers et les mots a passer de l'un a l'autre
        char *source;
        char *cible;
        char *tmp;
        char *systemstr;

        // des entiers
        int i,retour,max;

        // un caractere
        char test;

        // on alloue la memoire aux chaines de caracteres
        // pour les noms de fichier
        source = (char *)malloc(100);
        cible = (char *)malloc(100);


        // on lit le nom du fichier et on essaye de l'ouvrir
        while (fdsource==NULL){
                printf("\nEntrez le fichier a convertir :");
                scanf("%s",source);
                fdsource = fopen(source,"r");
                if (fdsource==NULL) printf("\nVerifiez que ce fichier existe ou est autorise en ac
        }

        // on lit le nom du fichier et on essaye de l'ouvrir
        while (fdcible==NULL){
                printf("\nEntrez le fichier cible :");
                scanf("%s",cible);
                fdcible = fopen(cible,"w");
                if (fdcible==NULL) printf("\nVerifiez que ce fichier est autorise en acces!");
        }

        // on lit le nombre de colones desire
        printf("\nEntrez le nombre de colonnes :");
        scanf("%d",&max);

        // on alloue la memoire pour la chaine de transfert
        // des mots
```

```
tmp = (char *)malloc(100);

// on ecrit l'en-tete du document
fprintf(fdcible,"<html><center>
                <h1>Resultat de txt2html<h1>");
fprintf(fdcible,"<table>");

// on commence a lire
retour = fscanf(fdsource,"%s",tmp);

// chaque iteration fait une ligne
while(retour!=EOF){

        fprintf(fdcible,"<tr>\n");

        // chaque iteration fait une case
        for (i=0;(i<max)&&(retour!=EOF);i++){
                retour = fscanf(fdsource,"%s",tmp);
                fprintf(fdcible,"<td> ");
                fprintf(fdcible,tmp);
                fprintf(fdcible,"</td> ");

        }

        fprintf(fdcible,"</tr>\n");
}

// on ferme les tags html...
fprintf(fdcible,"</table></center></html>\n");

// on ferme les deux fichiers
fclose(fdcible);
fclose(fdsource);


// on rassure l'utilisateur
printf("\n0peration reussie");

// on alloue de la memoire pour pouvoir
// fabriquer la chaine qui lancera netscape
systemstr=(char *)malloc(100);

//on ecrit qu'elle a 0 caracteres
*systemstr='\0';

// on concatene
strcat(systemstr,"netscape ");
strcat(systemstr,cible);

// on demande a l'utilisateur
```

```
        printf("Voulez-vous voir le resultat dans netscape[o/n]?");

        // on lit le 'Enter' de la lecture d'avant
        scanf("%c",&test);

        // on lit le caractere
        scanf("%c",&test);

        // si oui on lance le programme externe
        if (test=='o') system(systemstr);

        // on remercie l'utilisateur
        printf("\nMerci beaucoup!!!!!\n");
}
```

E.8 Chapitre 9 : Signaux et Handlers

```
// inclusion de librairies standard
#include <sys/types.h>
#include <signal.h>
#include "stdio.h"

/*
 * fonction renvoyant un help
 * */
void help(char *nom){
        printf("Usage : %s pid\n",nom);
        printf("kills the process with the pid\n");

}

/**
 * fonction principale
 * */
int main(int argc, char *argv[]){

        // on declare un entier pour stocker
        // le numero en argument
        int pid;

        // si on a le bon usage
        if (argc==2){
                // on traduit le numero de processus
                sscanf(argv[1],"%d",&pid);
                // on tue le processus
                kill(pid,9);
        }
        // sinon on met l'usage
        else help(argv[0]);
```

```
}
```

E.9 Chapitre 10 : Préprocesseur et Librairies

```
correction de Z/nZ
```

E.10 Chapitre 11 : Programmation Parallèle

```
correction de ping-pong
correction des entrees claviers
```

E.11 Chapitre 12 : Typage

```
correction de type pour representer identite de personne
```