Licence L.1.1 : Programmation 1

Maire-Pierre Béal
http ://igm.univ-mlv.fr/∼beal

L.1.1 : Programmation 1

0. Bibliographie
1. Introduction au système Unix. Linux Debian
2. Organisation du système de fichier
3. Éditeurs, Emacs
4. Le langage C
   - Compilation, exécution d'un programme : premiers exemples.
   - Notion de variable et de valeur
   - Types de base
   - Itérations
   - Fonctions, passage de paramètres.
   - Tableaux. Chaînes de caractères
   - Types structurés

Organisation

Organisation : douze semaines d'enseignement découpées en :

- Cours-TD : 12 séances de 3 heures,
  - 1 examen écrit en fin de semestre
- TP : 12 séances de 2 heures,
  - contrôles de TP plus un mini-projet.

– Plus les lundis du plan réussite en Licence.

Bibliographie

– *La programmation sous Unix*, 3e édition, Jean-Marie Rifflet, Science International, Paris, 1993

– *Passeport pour Unix et C*, Jean-Marc Champarnaud et Georges Hansel, Eyrolles, 2000.

– *Concepts Fondamentaux de l'Informatique*, A. Aho, V. Ullman, Dunod, 1993.

– *Méthodologie de la programmation en C*, Jean-Pierre Braquelaire, Masson, 1994.

– *Le langage C*, Brian Kernighan et Dennis Ritchie, Masson, 1988.

1

Système Unix

1. Système d'exploitation
2. Environnement de travail
3. Une session sous Unix
4. Le système de fichiers
5. Les éditeurs

Système d'exploitation

Les machines que vous utilisez comprennent un seul processeur.

Un système d'exploitation est une interface entre le niveau matériel et les programmes de l'utilisateur. Il permet de gérer la mémoire, les périphériques, les processus, l'allocation au processeur, les requêtes des utilisateurs, la communication, etc ...

Il est constitué d'une gamme d'outils à la disposition de l'utilisateur dont des *interpréteurs de langage de commande*. Il comprend aussi des *appels système* (fonction dont le code est chargé en mémoire), qui sont utilisés par les programmes.

Unix

– système multi-utilisateurs
– système multi-tâches.
– interpréteurs de langages de commande (shell) : il existe plusieurs langages de commande bash (Bourne-again shell, sur les systèmes GNU/Linux), sh, ksh, csh, tcsh, etc ...
– commandes de bases relatives à la gestion de fichiers, à la messagerie, l'archivage, etc ...
– autres outils : éditeurs de texte, compilateurs, éditeurs de liens, etc ...

Environnement de travail

Vous allez travailler sur des machines PC Dell avec Linux, distribution debian.

Utilitaires

– environnement de développement : compilateurs (C, C++, Java, Caml, Ocaml, Python), Eclipse
– éditeurs : vi, vim et variantes, emacs, xemacs, ...
– traitements de texte : TeX, LaTeX, Ghostview, soffice, Acrobat Reader et Writer,...
– mathématiques et calcul formel : scilab, MuPad, Maple

Une session Linux

Se loger

– En tant qu'utilisateur, vous allez avoir un *nom d'utilisateur* (ou encore *nom de login* et un *mot de passe* (ce sont deux chaînes de caractères avec certaines contraintes). Sur l'écran apparaît tout d'abord une fenêtre pour le nom de login, puis ensuite une autre pour le mot de passe

```
login : beal
password : xxxxxxxx
```

– Après avoir tapé chacune des chaînes, on appuie sur la touche `Enter` (ou retour chariot).

– Si le login et le mot de passe sont corrects, on se retrouve connecté, c'est-à-dire reconnu par le système comme un utilisateur ayant lancé une nouvelle session de travail.

– L'effet est l'ouverture d'une session dite sous X. L'affichage sur l'écran de plusieurs fenêtres est géré par un système de gestion des fenêtres ou window manager. Certaines fenêtres sont des fenêtres d'information et d'autres des fenêtres interactives dans lesquelles on va par exemple taper des commandes qui seront interprétées par le langage de commande shell.

– En général une fenêtre dite texte, ou encore un terminal alphanumérique xterm, est ouverte par défaut par le gestionnaire de fenêtre. Dans cette fenêtre apparaît le *caractère d'invite* ou *prompt*, par exemple `$`. Dans cette fenêtre tourne un shell.

Se déloger

- Attention, pour se déloger, il ne suffit pas de taper `exit` ou `logout` dans une fenêtre xterm, ce qui arrête la session attaché à cette fenêtre.
- On accède par le bouton droit de la souris à des menus déroulant. En sélectionnant `exit` ou `exit session` on est déconnecté et on retrouve l'écran d'acceuil initial.
- Il ne faut jamais partir avant d'avoir effectué la manœuvre précédente.
- L'environnement permet de créer plusieurs bureaux (ou écrans) par le fenêtre en haut à gauche.
- Des fenêtres en haut à droite permettre de créer une nouvelle fenêtre de type xterm et de configurer le gestionnaire de fenêtres.
- Testez les menus déroulant avec les différents boutons de la souris (en particulier le bouton de droite). Pour lancer le logiciel de navigation Netscape par exemple, sélectionner `Application`, `Net`, `Netscape`. La page d'accueil est le serveur Web des étudiants `http://etudiant.univ-mlv.fr/`

Quelques exemples de commandes
date, echo, who, finger, tty, man, wc

```
test@niznogood1:~$ date
lun sep 10 18:19:08 CEST 2001
test@niznogood1:~$ echo toto
toto
test@niznogood1:~$
```

Sur une autre machine, avec un autre prompt.

```
monge : ~ > who
bassino  pts/0    Sep  3 14:32
berstel  pts/2    Aug 27 10:49
herault  pts/5    Aug 21 14:33
ristov   pts/8    Sep 10 10:14
jyt      pts/9    Sep  5 11:12
ristov   pts/10   Sep 10 10:19
dr       pts/11   Sep  5 17:36
beal     pts/17   Sep 10 12:32
www      pts/19   Sep 10 15:49
beal     pts/20   Sep 10 16:09
sagot    pts/13   Sep 10 10:33
hivert   pts/15   Sep 10 12:19
marsan   pts/12   Sep 10 14:15
monge : ~ >
```

```
monge : ~ > finger berstel
Login: berstel                          Name: Jean Berstel
Directory: /home/institut/berstel       Shell: /bin/bash
On since Mon Aug 27 10:49 (CEST) on pts/2 from pcberstel
   3 days 5 hours idle
Mail last read Mon Sep 10 15:31 2001 (CEST)
No Plan.
monge : ~ > who
monge.univ-mlv.fr!beal     pts/21   Sep 10 16:14
monge : ~ > tty
/dev/pts/21
monge : ~ > man tty
--> affichage d'informations sur la commande tty
--> on en sort par q.
```

La commande `wc` permet de compter le nombre de lignes, de mots et de caractères d'un fichier.

```
monge : ~ > cat toto
Il fait
beau.
monge : ~ > wc toto
2  3 14 toto
```

Clavier, caractères de contrôle

– Certains caractères ont une action spéciale, par exemple l'appui sur la touche ← ou backspace provoque l'effacement du dernier caractère entré sur la ligne de commande.

– On peut modifier ces effets en reparamétrant la liaison entre le clavier et l'ordinateur par la commande **stty**.

– L'option **-a** de la commande **stty** permet de connaître l'état courant de la liaison.

```
beal@localhost ~/Enseignement/Prog1 $  stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?;
swtch = M-^?; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl
-iuclc ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
echoctl echoke
beal@localhost ~ $
```

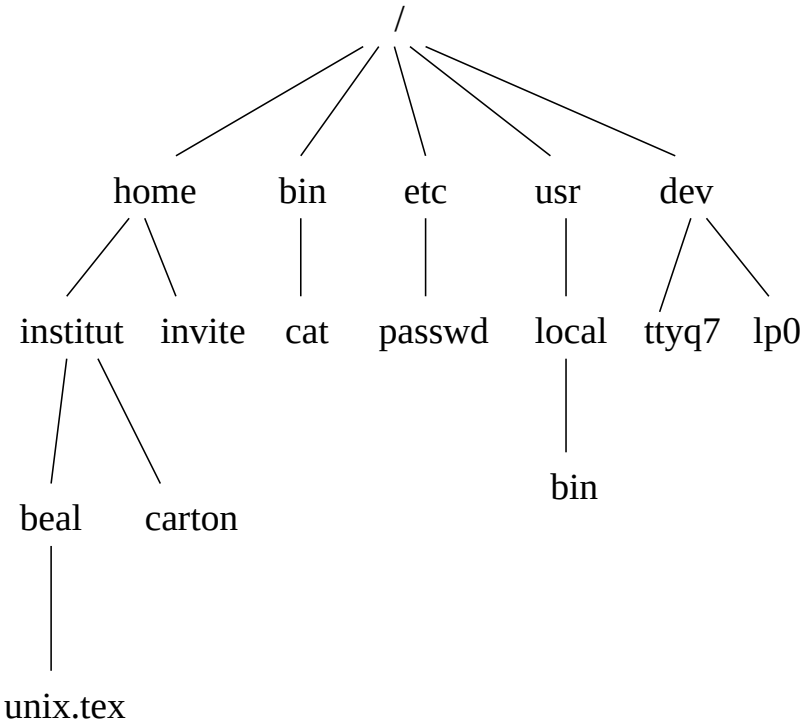
– Les principaux caractères de contrôle sont
  – **erase**, efface de dernier caractère, backspace ;
  – **kill**, efface la ligne, ctrl-u
  – **eof**, ferme un flux d'entrée en lecture, crtl-d
  – **newline**, valide une commande, crtl-j
  – **intr**, interruption, crtl-c
  – **quit**, interruption avec image mémoire, crtl-\

```
beal@localhost ~ $ sleep 1000
 --> frappe de crtl-c    ^C
beal@localhost ~ $
beal@localhost ~ $ stty intr w
beal@localhost ~ $ sleep 1000
 --> frappe de w          w
beal@localhost ~ $
```

Le système de fichiers

L'arbre des fichiers

– Un fichier sous Unix est non structuré.
– Une vision approchée du système de fichiers sous Unix est celle d'un arbre. On verra par la suite que ce n'est pas tout à fait vrai.
– Exemple d'arborescence.



– Tout nœud qui n'est pas une feuille de l'arbre est appelée *catalogue* (directory). Les feuilles sont soit des catalogues, soit des fichiers ordinaires, soit des fichiers spéciaux (désignant des périphériques).
– Cette structure permet de *référencer* un fichier ou un catalogue de plusieurs manières.

Références d'un fichier

– *Référence absolue* : Tout fichier ou catalogue peut être désigné par le chemin qui permet d'y accéder à partir de sa racine. Cette référence s'appelle référence absolue.

```
/home/institut/beal
/home/institut/beal/unix.tex
/dev/ttyq7
/
```

– *Référence relative* : A chaque instant, un utilisateur logé a un catalogue de travail. Il se situe virtuellement sur un nœud catalogue de l'arborescence. On connaît ce catalogue par la commande `pwd` (print working directory).

```
monge : ~/Recherche > pwd
monge : ~/Recherche > /home/institut/beal/Recherche
```

Tout fichier ou catalogue peut être référencé relativement à ce catalogue de travail. Une référence relative ne commence pas par le caractère `/`.

```
Article/Determinisation/det.tex
```

– *Les références* `.` *et* `..` : La référence `.` désigne le catalogue lui-même et la référence `..` désigne le catalogue père.

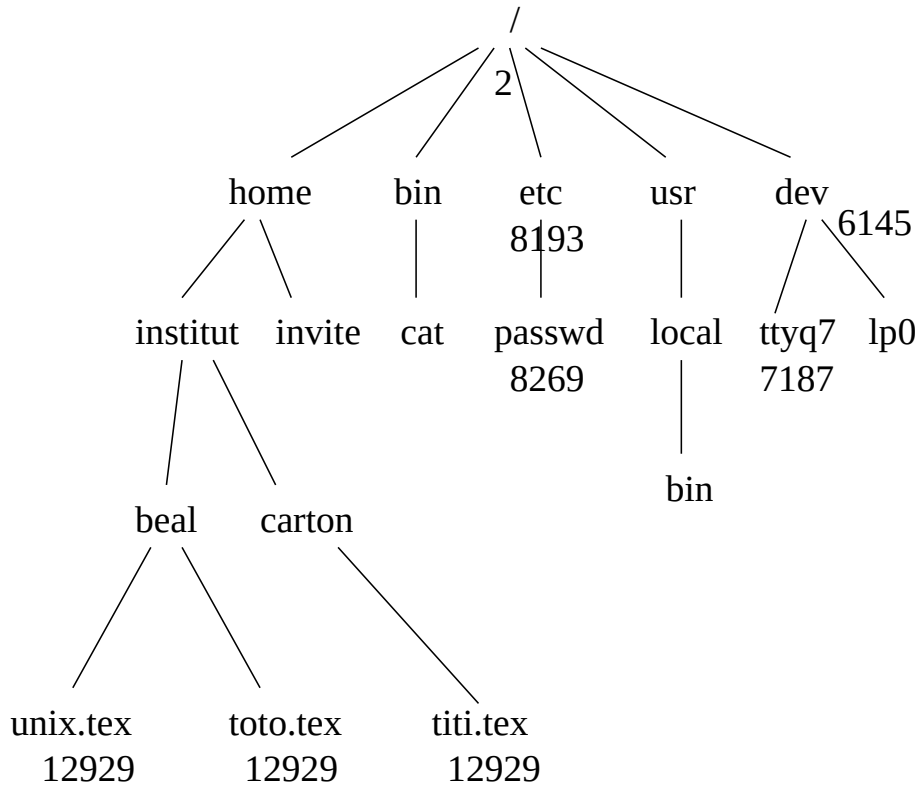
```
Article/Determinisation/det.tex
../Recherche/Article/Determinisation/det.tex
./Article/Determinisation/det.tex
../../beal/Recherche/Article/Determinisation/det.tex
```

– Chaque utilisateur se retrouve, après s'être logé, dans son catalogue privé.

```
/home/institut/beal
```

Le DAG[1] *des fichiers*

Nous allons donner une description plus précise de l'arborescence des fichiers.



A chaque fichier (ou catalogue) sur le disque est associé un *i-nœud*. C'est un bloc d'information écrit sur le disque qui contient les informations suivantes

– la taille du fichier ou catalogue,
– les adresses des blocs du disque ou est stocké le fichier ou le catalogue,
– l'identification du propriétaire et les droits d'accès,
– le type du fichier (catalogue, fichier ordinaire ou spécial),
– le nombre de liens du fichier, c'est-à-dire le nombre de fois ou il figure comme nom dans un catalogue.
– d'autres informations.

Le i-nœud ne contient pas de nom ou référence quelconque du fichier ou catalogue.

1. Directed Acyclic Graph

On peut alors préciser le contenu d'un catalogue. C'est une liste de couples `(numero de i-nœud,nom)`, où `nom` est un nom de fichier ou catalogue. Par exemple,

– le catalogue de référence absolue `/home/institut/beal` contient le couple

```
(12929,unix.tex)
```

– le catalogue de référence absolue `/home/institut/carton` contient le couple

```
(12929,titi.tex)
```

– A un fichier (ou catalogue) physique est associé un seul i-nœud et un seul numéro de i-nœud. Le numéro de i-nœud s'appelle l'index. Plusieurs couples peuvent avoir le même numéro et sont donc associés au même fichier (ou catalogue) physique. Chaque couple est appelé un *lien*.

– Le i-nœud d'index 12929 a un nombre de liens égal à 3. Les deux fichiers de référence absolue

```
/home/institut/beal/unix.tex
/home/institut/carton/titi.tex
```

sont donc les mêmes (un seul emplacement mémoire sur le disque).

– Combien y a t-il de liens sur le catalogue de référence absolue `/home/institut/beal` ?

Les fichiers spéciaux

A chaque ressource (terminal, imprimante, lecteur de disquette, disque logique, ...) est associé un fichier spécial. On le trouve dans le catalogue `/dev`. Sa taille est nulle. Les renseignements intéressants sont contenus dans le i-nœud. Il contient les informations suivantes

- propriétaire, groupe, droits d'accès,
- le *majeur* est un nombre entier qui indique la classe de ressource associée. Un terminal peut avoir pour majeur le nombre 3.
- le *mineur* est un nombre entier qui indique un exemplaire particulier de cette classe.
- le *mode* du fichier spécial, bloc ou caractère, qui indique le mode des lectures et écritures.

```
beal@localhost /dev $ tty
/dev/pts/1
beal@localhost /dev $ ls -s /dev/pts/1
0 /dev/pts/1
beal@localhost /dev $ ls -l /dev/pts/1
crw--w---- 1 beal tty 136, 1 Sep 15 14:28 /dev/pts/1
beal@localhost /dev $ mesg n
beal@localhost /dev $ ls -l /dev/pts/1
crw------- 1 beal tty 136, 1 Sep 15 14:29 /dev/pts/1
beal@localhost /dev $
```

Manipulations de base

On va voir les manipulations de base sur les fichiers qui comprennent les déplacements dans l'arborescence.

– La commande **cd** (change directory) permet de se déplacer dans l'arbre des fichiers.

```
monge : ~ > pwd
/home/institut/beal
monge : ~ > cd ..
monge : /home/institut > pwd
/home/institut/
monge : /home/institut >  cd iznogood
bash: iznogood: Aucun fichier ou répertoire de ce type.
monge :  /home/institut >  cd
monge : ~ >                --> retour à la maison
monge : ~ > cd ./../../institut/beal/Recherche
monge : ~/Recherche >
```

– Les commandes **cat** et **more** permettent l'affichage d'un fichier texte. On sort de more par **q**.

```
monge : ~ > echo coucou coucou > toto
monge : ~ > ls -l toto
-rw-rw-r--   1 beal     beal           14 sep 12 12:28 toto
monge : ~ > cat toto
coucou coucou
monge : ~ >
```

– La commande **ls** a de nombreuses options que l'on peut voir par la commande **man** (taper **man ls**). Sans option, elle affiche la liste des fichiers et catalogues qui ne commencent pas par un point. L'option **-a** permet d'avoir la liste complète de ces fichiers qui sont en général des fichiers d'initialisation et de paramétrage. L'option **-i** permet d'avoir les index des i-nœuds.

```
monge : ~ >  ls -al
drwxr-xr-x   4 root   root     1024 avr 15  1999 ..
drwxr-xr-x   2 beal   beal     1024 sep 22  1999 .Emacs
-rw-------   1 beal   beal      115 sep 12 10:04 .Xauthority
-rw-r--r--   1 beal   beal     1147 avr 27  1999 .Xdefaults
-rw-------   1 beal   beal    13244 sep 12 10:05 .bash_history
-rw-r--r--   1 beal   beal       24 avr 12  1999 .bash_logout
-rw-r--r--   1 beal   beal      307 jun 19  1999 .bash_profile
drwxr-xr-x   7 beal   beal     1024 fév  4  2001 Enseignement
drwxr-xr-x   7 beal   beal     1024 sep 12  2000 Recherche
-rw-r-x---   1 beal   beal       14 sep 12 12:28 toto
monge : ~ >
```

– La commande Unix **find** permet de rechercher un fichier dans une sous-arborescence d'un catalogue (faire **man find**).

```
[beal@localhost Unix]$ ls Cat
toto1.c  toto2.c
[beal@localhost Unix]$ find Cat -name toto1.c -print
Cat/toto1.c
[beal@localhost Unix]$
```

Manipulations de base suite

– La copie physique d'un fichier se fait avec la commande **cp**. Ceci crée deux fichiers distincts, avec des i-nœuds distincts, dont le contenu est identique.

– La création d'un lien supplémentaire sur un fichier déjà existant se fait par la commande **ln**. Il n'y a pas création d'un nouvel i-nœud mais seulement d'un nouveau lien. Le nombre de liens est incrémenté dans le i-nœud du fichier correspondant. Plusieurs références désignent alors un même fichier.

```
monge : ~ >  ls -ld
drwx------  36 beal     beal         5120 sep 12 12:28 .
monge : ~ >  ls toto
toto
monge : ~ >  ln toto tutu
monge : ~ >   cat tutu
coucou coucou
monge : ~ >  ls -il toto tutu
 211163 -rw-rw-r--   2 beal   beal   14 sep 12 12:28 toto
 211163 -rw-rw-r--   2 beal   beal   14 sep 12 12:28 tutu
monge : ~ >  cp toto titi
monge : ~ >  ls -il toto tutu titi
211191 -rw-rw-r--   1 beal   beal   14 sep 12 13:00 titi
211163 -rw-rw-r--   2 beal   beal   14 sep 12 12:28 toto
211163 -rw-rw-r--   2 beal   beal   14 sep 12 12:28 tutu
```

– La commande **rm** (remove) permet de détruire un lien sur un fichier (et non pas forcément le fichier lui même).

```
monge : ~ > rm toto
rm: détruire 'toto'? y                    --> alias sur rm -i
monge : ~ > cat toto
cat: toto: Aucun fichier ou répertoire de ce type.
monge : ~ > cat tutu
coucou coucou
monge : ~ >  ls -il toto tutu titi
ls: toto: Aucun fichier ou répertoire de ce type.
```

```
211191 -rw-rw-r--   1 beal   beal   14 sep 12 13:00 titi
211163 -rw-rw-r--   1 beal   beal   14 sep 12 12:28 tutu
```

– Un fichier est physiquement détruit lorsque son nombre de liens devient nul.

– La commande **mv** (move) permet de changer le nom d'un fichier dans un catalogue.

```
monge : ~ >  ls -il
211163 -rw-rw-r--   1 beal   beal   14 sep 12 12:28 tutu
monge : ~ >  mv tutu toutou
monge : ~ >  ls -il
211163 -rw-rw-r--   1 beal   beal   14 sep 12 12:28 toutou
```

– La création et suppression de catalogues se font par **mkdir** (make directory) et **rmdir** (remove directory).

```
monge : ~ >  mkdir Cat
monge : ~ >  mv toutou Cat/
monge : ~ >  cd Cat
monge : ~/Cat >  ls
toutou
monge : ~/Cat > cd
monge : ~ >  ls Cat
toutou                              ---> nouvelle action de ls
```

– Des caractères spéciaux du shell permettent de désigner un ensemble de fichier (voir plus loin).

```
monge : ~ > ls t*
titi toutou
 --> affiche la liste des fichiers (ou catalogues)
 --> commençant par t
```

Droits d'accès

- Les droits s'appliquent à trois catégories d'utilisateurs.
  - Le propriétaire du fichier (lettre `u` pour user)
  - Les membres du groupe du propriétaire (lettre `g` pour group)
  - Les autres utilisateurs (lettre `o` pour others)
  - Il existe un superutilisateur (root) qui a tous les droits.
- On distingue trois types de droits qui n'ont pas le même sens lorsqu'ils s'appliquent à un fichier ou à un catalogue.
  - Le droit en lecture (lettre `r` pour read) indique pour un fichier que l'on peut le lire, l'afficher, pour un catalogue que l'on peut lire la liste des fichiers ou catalogues qu'il contient.
  - Le droit en écriture (lettre `w` pour write) indique pour un fichier que l'on peut le modifier et pour un catalogue que l'on peut lui ajouter des fichiers ou supprimer les liens de fichiers dedans.
  - Le droit en exécution (lettre `x` pour exécute) indique pour un fichier qui est un programme exécutable que l'on est autorisé à l'exécuter. Pour un catalogue, le droit d'exécution indique que l'on peut se positionner sur lui et le traverser pour avoir accès à des fichiers qu'il contient.
- On peut changer les droits des fichiers ou catalogues dont on est propriétaire.

```
monge : ~/Cat > ls -l
total 1
-rw-rw-r--   1 beal  ens   14 sep 12 12:28 tutu
monge : ~/Cat > chmod u-r tutu
monge : ~/Cat > cat tutu
cat: tutu: Permission non accordée.
```

- On utilise souvent le code octal pour changer les droits (4 pour lecture, 2 pour écriture, 1 pour exécution).

```
monge : ~/Cat > chmod 750 tutu
monge : ~/Cat >  ls -l
total 1
```

```
-rwxr-x---   1 beal  ens   14 sep 12 12:28 tutu
```

Les éditeurs

L'éditeur vi

Pour lancer l'éditeur vi, on peut lancer
– `vi`,
– `vi toto`,
– `vi + toto`. Dans ce cas on se place à la fin du fichier `toto`.
L'éditeur vi a deux modes :
– le mode *commande*, dans lequel on est au moment de l'appel.
– le mode *insertion*, qui permet d'insérer directement dans le *tampon* ce qui est frappé au clavier.
Le passage du mode commande au mode insertion se fait par la frappe des caractères suivants
– `a`, `A` (insertion après le curseur, insertion en fin de ligne),
– `i`, `I` (insertion avant le curseur, insertion en début de ligne),
– `o` (insertion à la ligne suivante),
– `O` (insertion à la ligne précédente),
Le passage du mode commande au mode insertion se fait par la frappe du caractère `Esc`.

En mode insertion, on peut effacer ce qu'on vient d'écrire par le caractère backspace et on peut insérer un caractère spécial par `ctrl-v`. Par exemple, pour insérer le caractère `ctrl-a`, on frappe `ctrl-v ctrl-a` et on voit dans le tampon `^A`.

Dans le mode commande, on distingue deux moyens pour demander l'exécution d'une action. La plupart des commandes de recherche, remplacement, sauvegarde, se font en tapant d'abord `:`. Le curseur se positionne alors automatiquement au bas de l'écran et on peut taper une ligne de commande validée par un retour chariot. Il s'agit alors en fait de commandes de l'éditeur ligne `ed`.

– Quelques commandes de déplacement du curseur
  – `h,j,k,l`. On peut aussi utiliser les flèches.
  – `$` place le curseur en fin de ligne.
  – `G` place le curseur en fin de fichier.
  – `^F` : retour à la page précédente.
  – `^B` : avance à la page suivante.
– Quelques commandes d'effacement et de remplacement
  – `x` efface le caractère courant.
  – `dw` efface le mot courant.
  – `dd` efface la ligne courante.
  – `5dd` efface 5 lignes.
– Quelques commandes de l'édition en ligne
  – `:$` va à la fin du fichier.
  – `:r toto` ajoute le contenu du fichier `toto` après le curseur.
  – `:w` sauvegarde du tampon dans le fichier (`toto` si on a lancé `vi toto`).
  – `:w titi` sauvegarde du tampon dans le fichier `titi`.
  – `:wq` sauvegarde et sortie de vi.
  – `:q!` sortie de vi sans sauvergarde.

– Quelques commandes de recherche de motifs
  – `:/coucou` recherche la prochaine occurrence de `coucou`.
  – `:1,10s/coucou/doudou/g` remplace toutes les occurrences de `coucou` par `doudou` dans les lignes 1 à 10.
  – `:1,$s/coucou/doudou/g` remplace toutes les occurrences de `coucou` par `doudou` dans toutes les lignes.

L'éditeur Emacs

L'éditeur emacs est beaucoup plus puissant que vi. Il est aussi beaucoup plus gourmand en mémoire. La différence de modes (insertion, commande) disparaît. Les caractères frappées sont pour la plupart directement insérés dans le tampon. On peut aussi en particulier positionner le curseur à un endroit quelconque du texte en cliquant sur le bouton gauche de la souris.

Beaucoup de commandes sont accessibles par des menus et les menus sont configurables dans un fichier `.emacs` qui comprend des commandes en lisp.

Pour lancer l'éditeur emacs, on peut lancer

– `emacs`,
– `emacs toto`,
– `emacs + toto`. Dans ce cas on se place à la fin du fichier `toto`.

emacs@monge.univ-mlv.fr

Buffers Files Tools Edit Search Index LaTeX Command Help

```
\usepackage[dvips]{graphicx}
\oddsidemargin  0pt
\evensidemargin 0pt
\textwidth      16true cm
\textheight     24true cm
\topmargin      -30pt
\parindent = 0pt
\parskip=\smallskipamount
\setlength{\fboxsep}{4mm}
\newcounter{entree}
\newcommand{\entree}{\addtocounter{entree}{1}\thee\
ntree}

\sloppy
\def\titre#1{\newpage
\begin{center}
\shabox{{\color{red}#1}}
\end{center}
\bigskip\bigskip\bigskip}
%
\newif\ifdater\datertrue
```

-:** toutunix.tex (LaTeX Fill)--L19--C24--17%

```
  \item \texttt{emacs + toto}. Dans ce cas on se p\
lace à la fin du fichier \texttt{toto}.
\end{itemize}


\begin{itemize}
  \item le mode \emph{commande}, dans lequel on es\
t au moment de l'appel.
  \item le mode \emph{insertion}, qui permet d'ins\
érer directement dans le
\emph{buffer} ce qui est frappé au clavier.
\end{itemize}

\begin{center}
\scalebox{0.9}{\includegraphics{grab.ps}}
\end{center}

\begin{listing}\begin{verbatim}

\end{verbatim}\end{listing}
```

-:-- unix0.tex (LaTeX Fill Iso)--L829--C13

Auto-saving...done

Pour décrire les commandes de emacs, on note
– `C-p` au lieu de `ctrl-p` (rappel : la touche `ctrl` et `p` sont frappées en même temps).
– `M-p` au lieu de `alt-p`.
Certaines commandes s'obtiennent par une combinaison de touches qui provoquent l'ouverture d'un mini tampon au bas de la page. La commande est alors complétée dans le mini tampon. Ainsi

```
C-x i
```

Affiche un mini tampon avec

```
Insert file: /mnt/monge/sda1/institut/beal/Unix/
```

On indiquera ci-dessous de façon raccourcie

```
C-x i toto
```

La frappe de `C-g` permet d'annuler une commande en cours.

Quelques commandes de base
– `C-x C-s` sauvegarde du tampon.
– `C-x C-c` sortie de emacs.
– `C-x C-f toto` ouverture du fichier toto.
– `C-h` de l'aide
– `C-h t` tutorial.
– `C-x 2` sépare le tampon en deux.
– `C-x 1` remets une seul tampon.
– `C-s coucou` recherche la prochaine occurence du mot coucou. La recherche est incrémentale. Elle commence dès la frappe des premières lettres de `coucou`.
– `C-s C-s` répétition de la recherche.
– `M-% coucou doudou` recherche-remplacement des occurrences de `coucou` par `doudou`.
– `C-x g` annule la commande en cours
Nous renvoyons à la liste des commandes distribuée pour les autres commandes.

Il est possible de sélectionner une portion de texte à la souris et d'effacer la zone par un double clic sur le bouton de droite.

Compression et archivages

– Il existe plusieurs commandes de compression et décompression possibles. Elles correspondent à des algorithmes différents.

```
gzip toto
monge : ~/Unix > gzip toto
monge : ~/Unix > ls tot*
toto.gz              --> compression effectuee
monge : ~/Unix > gunzip toto
monge : ~/Unix >     --> pour decompresser
```

Les commandes **gzip** et **gunzip** sont les mêmes.

```
monge : ~/Unix > ls -il /bin/gunzip    /bin/gzip
30756 -rwxr-xr-x    3 root  48368  /bin/gunzip
30756 -rwxr-xr-x    3 root  48368  /bin/gzip
monge : ~/Unix >
```

– L'archivage peut se faire au moyen de la commande **tar**. L'archive est en général ensuite compressée. Ci-dessous le nom de l'archive du catalogue **Cat** est **titi.tar**.

```
monge : ~ > tar cvf titi.tar Cat
Cat/
Cat/toto
Cat/tutu
monge : ~ > tar xvf titi.tar
Cat/
Cat/toto
Cat/tutu
monge : ~ >
```

2

Le langage C

1. Compilation, exécution d'un programme
2. Types de base
3. Itérations
4. Fonctions, passage de paramètres
5. Tableaux. Chaînes de caractères
6. Types structurés

Un exemple étonnant

Que fait le programme suivant ?

```
long a = 10000, b, c = 8400, d, e, f[8401], g;

int main(void) {
for (; b-c;) f[b++] = a/5;
for (; d = 0,g = c * 2;c -= 14,printf("%.4ld", e+d/a),e =d % a)
for (b = c; d += f[b] * a, f[b] = d%--g, d /= g--, --b; d *= b);
return 0;
}
```

Réponse

```
31415926535897932384626433832795028841971693993751058209749445922148086513282306647093844609550582231725359408128481117450284102
81097566593344612847564823378678316527120190914564856692346034870066063155881748815209209628292540917153643678925903600113305300365759591953092186117381932611793105118548074462379962749567351244065664308602139494639522473719070217986094370277053921717629352635608277857713427577896091736371787214684409012249534301465499021960864034418159813629774771309960518707211349999998372978049830264252230825334468503526193118817101000313783875288658753320887311595628638823537875937519577818577805321712268066130019278763278865936153381827968230301952035301852968995773622599413891249082953311686172785588907509838175463746493931925506040092770167101819429555961989467678374494482553797747268471040475346462080462056966024058038150193511253382430035587640247496473263914199272992458631503028618297455570674983850549458858692695690927210797549911988183479775356636980742654252786255181841757467289097777223501414419735685481613611573525521334757418494684385233239073949222184272550254256887671790494601653466804988627232791786085784006422512520511739298489608412848862694560424196528502221066118663643719172874677646575739624138908658326459958133904780275900990522489407726719478268482601476990902640136394437455305068203496169646151570985838741059788595977297549893016175392846813826868325246808459872736446958486538367362226260991246080512438843904 51
```

Un exemple introductif

On veut calculer par un programme en C une table de conversion de Francs en Euro comme

```
10    1
20    3
30    4
40    6
50    7
60    9
70    10
80    12
90    13
100   15
```

On rappelle qu'un euro vaut 6,50 francs. La table ci-dessus est arrondie à l'euro inférieur.

Le programme

```
#include <stdio.h>

int main(void) {

/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/

  int euro, francs;
  int bas, haut, pas;

  bas = 10;
  haut = 100;
  pas = 10;
  francs = bas;
  while (francs <= haut) {
    euro = francs / 6.50;
    printf("%d\t%d\n",francs,euro);
    francs = francs + pas;
  }
  return 0;
}
```

Analyse

Les deux lignes

```
/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/
```

sont un *commentaire*. Ensuite

```
int euro, francs;
int bas, haut, pas;
```

sont des *déclarations*. Les variables sont déclarées de type entier (`int`). On a d'autres types en C comme `float` pour les nombres réels.

Les lignes suivantes

```
bas = 10;
haut = 100;
pas = 10;
francs = bas;
```

sont des *instructions d'affectation*. Une instruction se termine par ;

La boucle d'itération

Chaque ligne de la table se calcule de la même façon. On utilise donc une *itération* sous la forme :

```
while (francs <= haut) {
   ...
}
```

Signification : on teste d'abord la condition. Si elle est vraie, on execute le corps de la boucle (les instructions entre accolades). Ensuite on teste de nouveau la condition. Si elle est vraie on recommence, et ainsi de suite.

Si elle est fausse, on passe à la suite (ici à la fin).

La fonction printf

L'instruction

```
printf("%d\t%d\n",francs,euro);
```

est un appel de la fonction `printf`. Son premier argument donne le format d'affichage : Les `%d` sont des indications pour l'affichage des arguments suivants. Le `d` est là pour indiquer une valeur entière (en décimal). Si on avait écrit

```
printf("%3d %6d\n",francs, euro);
```

on aurait écrit sur un nombre fixe de caractères (3 et six) d'où le résultat justifié à droite :

```
 10      1
 20      3
 ...
 70     10
 80     12
 90     13
100     15
```

Et les centimes ?

```
#include <stdio.h>

int main(void) {

/* ecrit la table de conversion
 Francs-Euro de 10 a 100 francs*/

  float euro, francs, taux;
  int bas, haut, pas;

  bas = 10;
  haut = 100;
  pas = 10;
  francs = bas; taux = 1/6.5;
  while (francs <= haut) {
    euro = francs * taux;
    printf("%3.0f %6.2f\n",francs,euro);
    francs = francs + pas;
  }
 return 0;
}
```

Résultat

```
10      1.54
20      3.08
30      4.62
...
```

Si on avait utilisé `taux = 1/6.5` avec taux déclaré entier, on aurait obtenu 0 (facheux).

Le format `%6.2f` signifie : nombre réel écrit sur six caractères avec deux décimales. Pour `%3.0`, c'est sans décimale ni point.

L'instruction for

Autre forme du programme :

```
#include <stdio.h>

int main(void) {
  int fr;

  for (fr = 10; fr <= 100; fr = fr + 10)
    printf("%3d %6.2f\n", fr, fr/6.5);
  return 0;
}
```

L'initialisation

```
fr = 10
```

est faite d'abord. On teste ensuite la condition

```
fr <= 100
```

Si elle est vraie, on exécute le corps de la boucle. On exécute ensuite l'instruction d'incrémentation

```
fr = fr + 10;
```

On reteste la condition et ainsi de suite.

Constantes symboliques

On peut utiliser des définitions préliminaires pour définir des constantes.

```
#include <stdio.h>

#define BAS 10
#define HAUT 100
#define PAS 10

int main(void) {
  int fr;

  for (fr = BAS; fr <= HAUT; fr = fr + PAS)
    printf("%3d %6.2f\n", fr, fr/6.5);
  return 0;
}
```

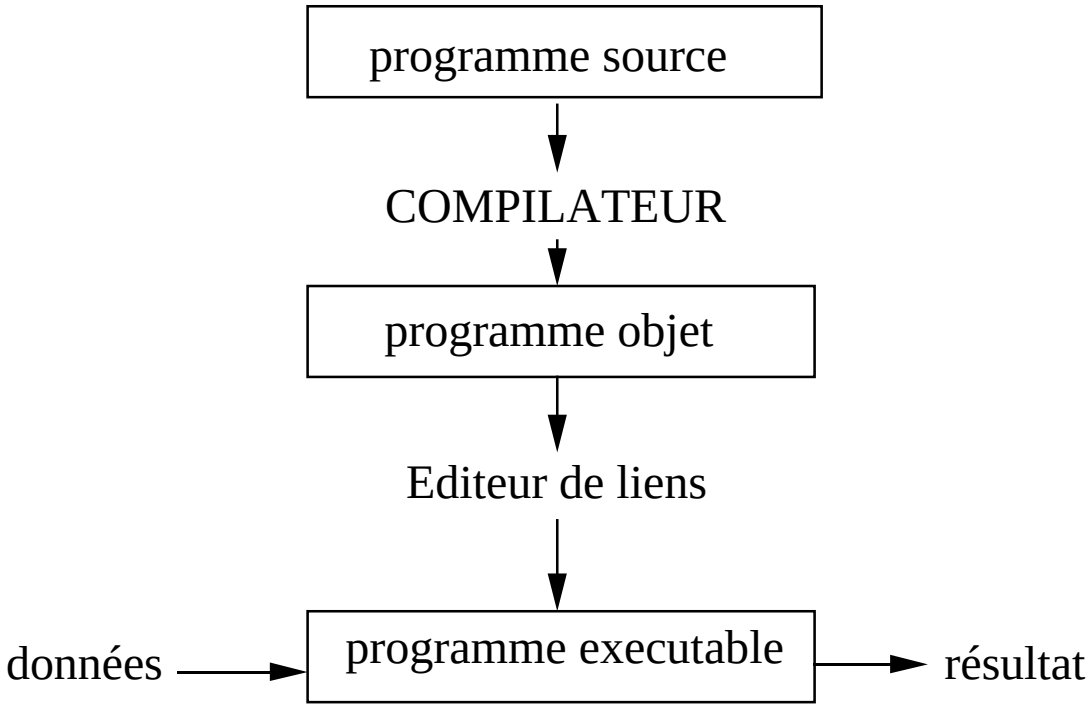
Cours 2 : Le langage C

Le langage C a été crée en 1970 aux Bell Laboratories par Brian Kernighan et Denis Ritchie.

Il existe beaucoup d'autres langages de programmation.
– Fortran (surtout pour le calcul numérique)
– Pascal (ancien)
– Caml, Ocaml (langages fonctionnels)
– Pearl
– Java, C++, Python
Ce sont des langages de *haut niveau* par opposition au langage machine et à l'assembleur, dits de *bas niveau*.

Il existe par ailleurs un très grand nombre de langages *spécialisés* allant des langages de commande des imprimantes aux systèmes de calcul formel.

Exécution d'un programme



Exemple de programme en C

Affiche au terminal la chaîne de caractères *bonjour* et passe à la ligne.

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

int main(void) {
  printf("bonjour\n");
  return 0;
}
------------------------------
```

ou encore

```
#include <stdio.h>

void afficher(void){
 printf("bonjour\n");
}
int main(void) {
 afficher();
 return 0;
}
```

Deuxième exemple

Programme qui lit un entier et écrit son carré.

```
int main(void){
 int n,m;
 printf("donnez un nombre :");
 scanf("%d",&n);
 m=n*n;
 printf("voici son carre :");
 printf("%d\n",m);
 return 0;
}
```

Execution :

```
donnez un nombre : 13
voici son carre : 169
```

Structure d'un programme C

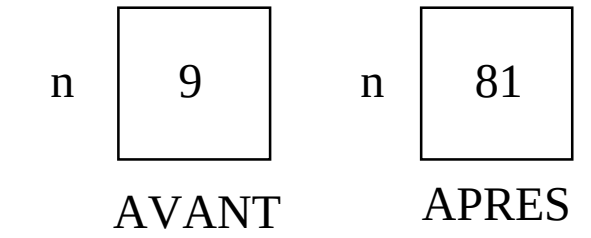
Un programme C est constitué de fonctions. Elles ont toutes la même structure.

Type retourné Nom(*Types et Noms des paramètres*)
{
 déclarations
 instructions
}

La fonction principale s'appelle `main`.

Les variables

Une *variable* en C est désignée par un nom qui est une chaîne de caractères (commençant par une lettre) appelée un *identificateur*. Une variable a une *adresse*, un *type* et une *valeur*.



La mémoire centrale

Les données sont rangées dans la mémoire sous forme de suites de *bits* égaux à 0 ou 1.

Les bits sont regroupés par groupe de 8 (un octet) puis, selon les machines, par *mots* de 16, 32 ou 64 bits.

Chaque mot porte un numéro : son *adresse* en mémoire.

On utilise les octets pour coder les caractères en utilisant le *code ASCII*. On aura par exemple pour la lettre A le code :

place	7	6	5	4	3	2	1	0
valeur	0	1	0	0	0	0	0	1

qui représente 65 en binaire.

Le premier bit est utilisé comme bit de parité en général (non standard). Sur les 7 autres, on peut coder $2^7 = 128$ caractères (pas beaucoup pour les accents !).

Les types

Chaque variable a un *type*. Elle est *déclarée* avec son type par :

```
int n;
```

Les types de base en C sont :

`char`	caractères
`int`	entiers
`float`	flottants
`double`	flottant double précision

Il existe aussi des types plus compliqués : les *tableaux* et les *structures*.

Les entiers

Le type entier (`int`) est représenté sur 16 ou 32 bits selon les machines (à tester dans `<limits.h>`).

Sur 16 bits, les valeurs vont de $-2^{15} = -32768$ à $2^{15} - 1 = 32767$.

Sur 32 bits, on a comme valeurs les entiers relatifs de -2^{31} à $2^{31} - 1$.

Les opérations sont $+, -, *, /$ (division entière par suppression de la partie fractionnaire) et % (reste de la division euclidienne).

Les réels

C'est le type `float` avec une représentation en *flottant* de la forme 1.25×10^{-4}. Ils sont représentés en général sur 32 bits avec six chiffres significatifs et une taille comprise entre 10^{-38} et 10^{+38}.

Les opérateurs sont $+, -, *, /$, ...

Il y a des *conversions* entre les divers types. Par exemple, si x est de type `float`, x+1 est de type float.

Il y aussi un type `double` pour les réels en double précision.

Les caractères

Le type `char` est représenté sur un seul octet. Ce sont les caractères représentés dans le code ASCII. C'est en fait un entier.

Les constantes de type `char` s'écrivent `'x'`.

Attention `'0'` vaut 79, pas 0. C'est `'\0'` qui vaut zéro.

Certains caractères s'écrivent de façon particulière, comme `'\n'` (newline).

On forme avec des caractères des chaînes comme

```
"bonjour\n"
```

Les expressions

On construit une expression en utilisant des variables et des constantes puis des opérateurs choisis parmi
– les opérateurs arithmétiques : $+, -, *, /, ...$
– Les comparateurs : $>, >=, ==, !=, \ldots$
– Les opérateurs logiques : && (et), || (ou)
Elle peut être de type char, int ou float.

Exemple : si i, j sont entiers et x réel alors :

$i - j$ est entier

$2 * x$ est réel

'2'<'A' est entier (la valeur 'vrai' est représentée par l'entier 1)

Expressions logiques

Les valeurs logiques 'vrai' et 'faux' sont représentées en C par les entiers 1 et 0 (en fait toute valeur non nulle est interprétée comme vrai).

Les opérateurs logiques sont `&&` (et), `||` (ou) et `!` (non) avec les *tables de vérité* :

ou	0	1
0	0	1
1	1	1

et	0	1
0	0	0
1	0	1

Le C pratique l'évaluation *paresseuse* : si C est vraie, alors `C || D` est vraie, même si D n'est pas définie.

De même, si C est fausse, `C && D` est fausse.

Exemple

Conversion des températures des Fahrenheit aux Celsius. La formule est

$$C = \frac{5}{9}(F - 32)$$

```
int main(void)
{
 float F, C;

 printf("Donnez une temperature
 en  Fahrenheit : ");
 scanf("%f",&F);
 C = (F - 32) * 5 / 9;
 printf("Valeur en  Celsius : %3.1f\n",C);
 return 0;
}
```

Exécution :

```
donnez une temperature en Fahrenheit :  100
valeur en Celsius : 37.8
```

Les déclarations

On peut regrouper des déclarations et des instructions en un bloc de la forme

{
liste de déclarations
liste d'instructions
}

Les déclarations sont de la forme *type nom ;* comme dans

```
int x;
```

ou *type nom=valeur ;* comme dans

```
int x=0;
```

qui initialise x à la valeur 0.

L'affectation

C'est le mécanisme de base de la programmation. Elle permet de changer la *valeur* des variables.

```
x = e;
```

affecte à la variable `x` la valeur de l'expression `e`. Attention : A gauche de =, on doit pouvoir calculer une adresse (on dit que l'expression est une *l-valeur*).

Exemples

```
i = i+1;
```

augmente de 1 la valeur de i. Aussi réalisé par l'instruction `i++;`

```
temp = i; i = j; j = temp;
```

échange les valeurs de i et j.

Instructions conditionnelles

Elles ont la forme :

if (*test*)
  instruction

(forme incomplète)

ou aussi :

if (*test*)
  instruction
else
  instruction

(forme complète)

Le test est une expression de type entier comme `a<b` construite en général avec les comparateurs `<`, `==`, `!=`, ... et les opérateurs logiques `&&`, `||`, ...

Exemple

Calcul du minimum de deux valeurs a et b :

```
if (a < b){
  min = a;
} else {
  min = b;
}
```

Forme abrégée :

```
min = (a<b)? a : b;
```

Minimum de trois valeurs :

```
if (a < b && a < c)
    min =  a;
  else if (b < c)
    min = b;
  else
    min = c;
```

Branchements plus riches

Si on a un choix avec plusieurs cas, on peut utiliser une instruction d'aiguillage.

Par exemple, pour écrire un chiffre en lettres :

```
switch (x){
  case 0: printf("zero"); break;
  case 1: printf("un"); break;
  case 2: printf("deux"); break;
  ...
  case 9: printf("neuf"); break;
  default: printf("erreur");
}
```

Cours 3 : Les itérations

1. Les trois formes possibles
2. Traduction
3. Itération et récurrence
4. Ecriture en binaire

Les trois formes possibles

Elles ont l'une des trois formes suivantes :

1. boucle **pour**.
2. boucle **tant que** faire.
3. boucle faire **tant que**.

Boucle pour

`for` (*initialisation*; *test*; *incrémentation*) {
liste d'instructions
}

comme dans :

```
for (i = 1;  i<=n; i++) {
  x = x+1;
}
```

L'exécution consiste à itérer
initialisation
(test, liste d'instructions, incrémentation)
(test, liste d'instructions, incrémentation)
....
On peut aussi mettre une liste d'instructions à la place de incrémentation.

Boucle tant que

`while` (*test*) {
liste d'instructions
}
comme dans :

```
while (i <= n) {
 i = i+1;
}
```

Si $i \leq n$ avant l'exécution, on a après $i = n + 1$. Sinon, i garde sa valeur.

L'autre boucle tant que

C'est la boucle 'faire tant que'.

`do {`
liste d'instructions
`} while` (*test*) ;
comme dans :

```
do {
 i = i+1;
} while (i <= n);
```

Si $i \leq n$ avant, on a après $i = n + 1$. Sinon, i augmente de 1.

Exemples de boucle pour

Calcul de la somme des n premiers entiers :

```
s = 0;
for (i = 1; i<= n; i++) s = s+i;
```

Après l'exécution de cette instruction, `s` vaut :

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Calcul de la somme des n premiers carrés :

```
s = 0;
for (i = 1; i<= n; i++) s = s+i*i;
```

Après l'execution de cette instruction, `s` vaut :

$$1 + 4 + 9 + \ldots + n^2$$

(au fait, ça fait combien ?)

Exemple de boucle tant que

Calcul de la première puissance de 2 excédant un nombre N :

```
p = 1;
while (p < N) {
  p = 2*p;
}
```

Résultats :

Pour N=100 on a p=128

Pour N=200 on a p=256

...

Test de primalité

Voici un exemple de boucle 'do while' : le test de primalité d'un nombre entier.

```
int main(void){
 int d, n, r;
  printf("donnez un entier: ");
  scanf("%d", &n);
  d = 1;
  do {
   d = d+1;
   r = n % d;
  } while (r >= 1 &&  d*d <= n);
  if ((r==0) && (n > 2))
        printf("nombre divisible par %d\n",d);
   else
        printf("nombre premier\n");
  return 0;
}
```

Traduction des itérations

L'instruction

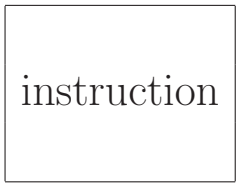
while (i <= n) {

instruction

}

se traduit en assembleur par :

```
1: IF i > n  GOTO 2
```




```
   GOTO 1
2:
```

De même l'itération

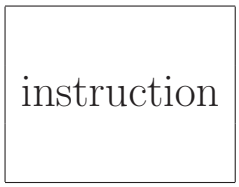
`for` (i = 1 ; i<= n ; i++) {

instruction

}

est traduite par :

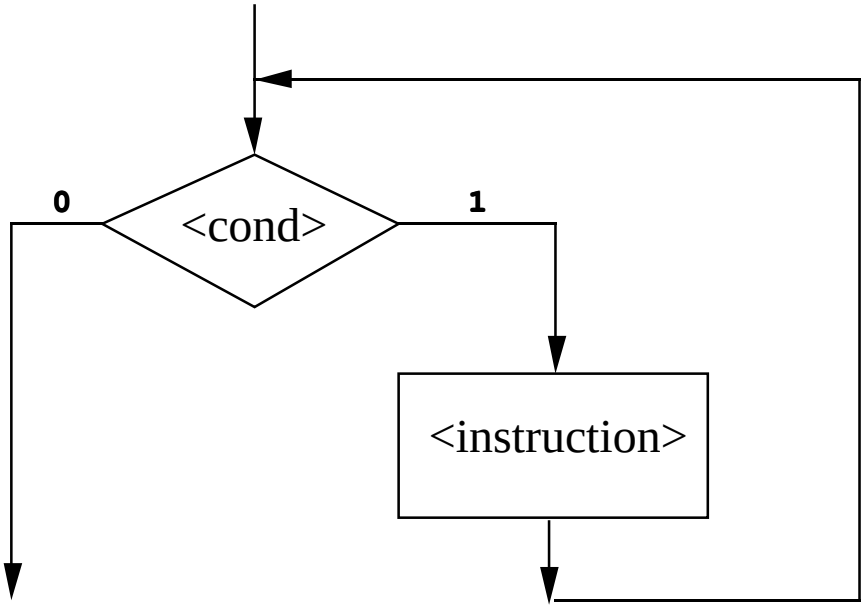
```
   i = 1
1: if i > n GOTO 2
```




```
  i = i+1
  GOTO 1
2:
```

Organigrammes

On peut aussi traduire en schémas appelés *organigrammes* ou *chartes*.



Temps de calcul

Les itérations permettent d'écrire des programmes qui tournent longtemps. Parfois trop !

Le programme

```
int main(){
 while (1);
 return 0;
}
```

ne s'arrête jamais.

Itération et récurrence

Les programmes itératifs sont liés à la notion mathématique de *récurrence*.

Par exemple la suite de nombre définie par récurrence par $f_0 = f_1 = 1$ et pour $n \geq 2$ par

$$f_n = f_{n-1} + f_{n-2}$$

s'appelle la suite de Fibonacci :

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Elle peut être calculée par le programme suivant (qui affiche les 20 premières valeurs) :

```
int main(void){
 int i,u,v,w;

  u = 1; v = 1;
  for (i= 1; i<= 20; i++) {
     w = u+v;
     printf("%d ",w);
     u = v; v = w;
  }
 return 0;
}
```

Résultat :

```
   2    3    5    8   13   21   34
  55   89  144  233  377  610  987
1597 2584 4181 6765 10946 17711
```

Invariants de boucles

Pour *raisonner* sur un programme comportant des itérations, on raisonne par récurrence.

On utilise une propriété vraie à chaque passage dans la boucle, appelée *invariant de boucle*. Par exemple, dans le programme précédent, la propriété :

$$P(i) : \ u = f_{i-1}, v = f_i$$

est un invariant de boucle : elle est vraie à chaque fois qu'on entre dans la boucle.

Pour le démontrer, on doit

1. vérifier qu'elle est vraie à la première entrée (et donc pour $i = 1$).
2. Que si elle est vraie à un passage (pour i), elle est vraie au suivant (pour $i + 1$).

Un autre exemple : La fonction factorielle

Le programme suivant

```
int main(void) {

 int i, n, fact;

  scanf("%d",&n);
  fact=1;
  for (i=2; i<= n; i++)
   fact = fact * i;
  printf("n!=%d\n",fact);
  return 0;
}
```

calcule

$$n! = 1 \times 2 \times \cdots \times n$$

L'invariant de boucle est $fact = (i-1)!$.

Un exemple plus difficile : l'écriture en binaire

La représentation binaire d'un nombre entier x est la suite $(b_k, \ldots, b_1, b_0)$ définie par la formule :

$$x \;=\; b_k 2^k + \cdots + b_1 2 + b_0$$

Principe de calcul : en deux étapes

1. On calcule $y = 2^k$ comme la plus grande puissance de 2 t.q. $y \leq x$.
2. Itérativement, on remplace y par $y/2$ en soustrayant y de x à chaque fois que $y \leq x$ (et en écrivant 1 ou 0 suivant le cas).

```
int main(void) {

int x,y,n;

  scanf("%d", &n); x = n; y = 1;
  while (y+y <= x)  y = y+y;
  while (y != 0) {
    if (x < y) printf("0");
    else {
      printf("1"); x = x - y;
    }
    y = y / 2;
  }
  printf("\n");
  return 0;
}
```

Cours 4 Fonctions

Idée : écrire un programme comme un jeu de fonctions qui s'appellent mutuellement (informatique en kit).

Chaque fonction aura :

1. Une *définition* : qui comprend son nom et
   (a) Le type et le nom de ses paramètres.
   (b) Le type de la valeur rendue.
   (c) Son code : comment on la calcule.
2. Un ou plusieurs *appels* : c'est l'utilisation de la fonction.
3. Un ou plusieurs *paramètres* : ce sont les arguments de la fonction.
4. Un *type* et une *valeur*.

Exemples sans paramètres

Nous avons déja vu la fonction

```
void afficher(void){
 printf("bonjour\n");
}
```

Le type `void` est utilisé par convention en l'absence de valeur ou de paramètres. On peut aussi simplement écrire

```
double pi(void){
 return 3.1415926535897931;
}
```

dont la valeur est de type `double` (nombres réels en double précision). On pourra ensuite, dans une autre fonction, écrire l'expression `0.5*pi()`.

Exemple avec paramètres

La fonction suivante rend le maximum de deux valeurs de type flottant.

```
float fmax(float a, float b){
 if (a > b)
   return a;
 else
   return b;
}
```

de sorte que l'expression `fmax(pi()*pi(),10.0)` vaut 10.

Version plus courte

```
float fmax(float a, float b){
 return (a > b) ? a : b;
}
```

Définition de fonctions

La définition d'une fonction comprend deux partie : l'en-tête (ou prototype) qui est de la forme

Type de valeur Nom (*Types des paramètres*) suivi de son code : un bloc de la forme

{
liste-de déclarations
liste-d'instructions
}

Les déclarations sont valables à l'intérieur du bloc seulement (variables locales).

On peut aussi avoir des variables *globales* déclarées au même niveau que les fonctions.

L'instruction `return` *expression*; permet de rendre une valeur et arrête l'exécution.

Une fonction doit être définie avant d'être utilisée.

Appels de fonctions

Une fonction est déclarée avec des paramètres formels. Elle est appelée avec des paramètres réels (en nombre égal).

déclaration de fonction	appel de fonction
paramètre formel	paramètre d'appel

```
float fmax(float a, float b){
 return (a> b) ? a : b;
}

int main(void){
 float x;
 scanf("%f",&x);
 printf("%f",fmax(x,x*x));
 return 0;
}
```

Autres exemples de fonctions

Pour tester si un nombre est premier :

```
void premier(int n){
 int d;
  d = 1;
  do
   d = d+1;
  while (n % d >= 1 &&  d*d <= n);
  if ((n % d == 0) && (n > 2))
        printf("nombre divisible par %d\n",d);
   else
        printf("nombre premier\n");
}
```

La fonction factorielle :

```
int fact(int n){
 int i, f;

  f = 1;
  for (i=2; i<= n; i++)
   f = f * i;
  return f;
}
```

Modes de transmission des paramètres

L'appel d'une fonction

```
truc(x)
```

transmet à la fonction `truc` la *valeur* de la variable x (ou de l'expression x).
Pour changer la valeur de la variable x, il faut transmettre son *adresse*. On utilise l'opérateur & comme dans

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

qui permet de lire au clavier une valeur qui sera affectée à la variable n.

Exemple

```
void echange (int x, int y){
int t;

t = x; x = y; y = t;
}
```

L'appel de `echange(a,b)` ne change pas les valeurs de a et b : la fonction `echange` travaille sur des copies auxquelles on passe la *valeur* de a et b.

Par contre :

```
void echange (int *p, int *q){
int temp;

temp = *p; *p = *q; *q = temp;
}
```

Réalise l'échange en appelant `echange(&a,&b);`

Explication

Les opérateurs `&` (*référence*) et `*` (*déréférence*) sont inverses l'un de l'autre.

– Si x est une variable de type t, alors `&x` est l'adresse de x. C'est une expression de type $t*$

– Si p est déclaré de type $t*$, alors `*p` est une expression de type t.

On dit que p est un *pointeur*. C'est une variable dont la valeur est une adresse.

Exemple

Supposons que l'adresse de la variable x soit 342 et sa valeur 7.

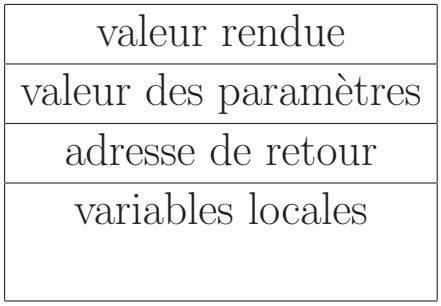
L'instruction

```
p=&x;
```

donne à la variable p la valeur 342. La valeur de l'expression `*p` est 7.

Implémentation des appels de fonctions

Chaque appel de de fonction est effectué dans une zone mémoire nouvelle et réservée à cet appel.



On dit que cette zone est un *enregistrement d'activation* (‘*activation record*’). Quand l'appel de fonction est terminé, la zone est libérée (on dit que c'est une gestion *dynamique* de la mémoire).

Cours 5 Tableaux

Type de données *construit* le plus simple. Permet de représenter des suites d'éléments d'un ensemble.

0	1	2	3	4
0.3	1.4	15.0	1.6	4.8

Avantage : pouvoir accéder aux éléments avec une adresse calculée.
Contrainte : tous les éléments doivent être du même type.
Déclaration :

```
float a[5];
```

Tous les indices commencent à 0. Le *nom* de l'élément d'indice i est :

```
a[i]
```

Définition des tableaux

On utilise souvent une *constante* pour désigner la dimension d'un tableau.

```
 #define N 50 /* dimension des tableaux */
..
 float Resultats[N];
```

déclare un tableau de 50 nombres réels de R[0] à R[49].
On peut définir et initialiser un tableau de la façon suivante.

```
int a[5]={1,1,1,1,1};
```

initialise tous les éléments de a à 1.

Lecture et écriture d'un tableau

On peut lire les valeurs d'un tableau de N nombres entiers

```
void Lire(int a[]){
 int i;
 for (i = 0; i < N; i++)
   scanf("%d",&a[i]);
}
```

ou les écrire

```
void Ecrire(int a[]){
 int i;
 for (i = 0; i < N; i++)
   printf("%d ",a[i]);
 printf(``\n'');
}
```

Le passage est par adresse car le nom d'un tableau est une adresse (celle du premier élément). De ce fait on peut écrire

```
void Lire(int * a)
```

au lieu de `void Lire(int a[])`.

Si on veut passer la dimension du tableau en paramètre (au lieu d'utiliser une constante symbolique), il faut écrire :

```
void Lire(int a[], int n){
 int i;
 for (i = 0; i < n; i++)
   scanf("%d",&a[i]);
}
```

et

```
void Ecrire(int a[], int n){
 int i;
 for (i = 0; i < n; i++)
   printf("%d ",a[i]);
 printf(``\n'');
}
```

On pourra alors écrire par exemple

```
int main(void){
  int a[N];
  scanf("%d",&n);
  if(n>N) return 1;
  Lire(a,n);
  Ecrire(a,n);
  return 0;
}
```

Recherche du minimum

Donnée : Tableau $a[0], a[1], \ldots, a[N-1]$ de nombres entiers.

Principe : On garde dans la variable m la valeur provisoire du minimum et on balaye le tableau de 0 à $N-1$.

```
int Min(int a[]){
 int i, m;

  m = a[0];
  for  (i = 1; i < N; i++)
   if (a[i] < m)  m = a[i];
  return m;
}
```

Somme des éléments

Calcul de la somme $a[0] + a[1] + \ldots + a[N-1]$.

```
int Somme(int a[]){
  int i,s;
  s=0;
  for(i=0;i<N;i++)
    s+=a[i];
  return s;
}
```

Tri par sélection

Donnée : N nombres entiers $a[0], a[1], \ldots, a[N-1]$.

Résultat : Réarrangement croissant $b[0], b[1], \ldots, b[N-1]$ des nombres $a[i]$.

Exemple :

$a =$

24	5	81	2	45

$b =$

2	5	24	45	81

Principe : A la i-ème étape, on cherche le minimum de $a[i], a[i+1], \ldots, a[N-1]$ et on l'échange avec $a[i]$.

Tri par sélection

```
#define  N 10
/* prototypes*/
void Lire(int a[]);
void Ecrire(int a[]);

int IndiceMin(int a[], int i){
/* calcule l'indice du minimum*/
/* a partir de i */
  int j,k;

  j = i;
  for  (k = i+1; k < N; k++)
    if (a[k] < a[j])  j = k;
  return j;
}
```

```
void Trier(int a[]){
  int i;

  for (i = 0; i < N; i++){
     int j,t;
     j=IndiceMin(a,i);
     t=a[j]; a[j]=a[i]; a[i]=t;
  }
}

int main(void){
  int a[N];
  Lire(a); Trier(a); Ecrire(a);
  return 0;
}
```

Le nombre d'opérations est de l'ordre de N^2. Pour $N = 10^5$ avec 10^6 opérations/s, le temps devient de l'ordre de 10^4 s (près de 3 h).

Interclassement

On veut réaliser la fusion de deux suites croissantes $a[0] < a[1] < \cdots < a[n-1]$ et $b[0] < b[1] < \cdots < b[m-1]$, c'est à dire une suite $c[0] < c[1] < \cdots < c[m+n-1]$ qui est un interclassement des suites a et b.

Le principe est de parcourir linéairement les suites a et b de gauche à droite en insérant à chaque fois dans c le plus petit des deux éléments courants.

Par exemple : Si

a =	2	10	12	25	41	98
b =	5	14	31	49	81	

Alors :

c =	2	5	10	12	14	25	31	41	49	81	98

On utilise deux éléments supplémentaires à la fin des tableaux a et b plus grands que tous les autres (sentinelles).

```
void Interclassement(int a[], int b[], int c[],int p){
  /*i et j servent a parcourir a et b*/
  int i = 0, j = 0;
  while (i+j < p)
    if ( a[i] <= b[j] ){
      c[i+j] = a[i];
      i++;
    }
    else if (b[j]< a[i]){
      c[i+j] = b[j];
      j++;
    }
}
```

La fonction principale s'écrit alors

```
int main(void){
  int a[N]; int b[M];
  int k=N+M-2; int c[k];
  Lire(a,N); Lire(b,M);
  Interclassement(a,b,c,k);
  Ecrire(c,k);
  return 0;
}
```

Inversion d'une table

Problème : On dispose d'une table qui donne pour chaque étudiant(e) (classé dans l'ordre alphabétique) son rang au concours de (beauté, chant, informatique,..) :

numéro	nom	rang
0	arthur	245
1	béatrice	5
2	claude	458
	...	

On veut construire le tableau qui donne pour chaque rang le numéro de l'étudiant :

rang	nom	numéro
0	justine	125
1	nathan	259
	...	

On part du tableau `int rang[N]` et on calcule le résultat dans un tableau `numero` du même type.

```
void Inversion(int rang[], int numero[])
{
 int i;
 for (i = 0; i < N; i++)
     numero[rang[i]] = i;
}
```

On a en fait réalisé un *tri* suivant une clé (le rang au concours) par la méthode de *sélection de place*.

Moralité : pour parcourir un tableau le plus efficace n'est pas toujours de le faire dans l'ordre. Ici on utilise un *adressage indirect*.

Tri par séléction de place

On fait l'hypothèse que toutes les valeurs sont $< M$. On utilise un tableau de M booléens.

```
void Tri(int a[]){
  int i,j;
  int aux[M];
  for(i=0;i<M; i++) aux[i]=0;
  for(i=0;i<N; i++) aux[a[i]]=1;
  j=0;
  for(i=0;i<M; i++)
    if(aux[i]==1){
      a[j]=i;
      j++;
    }
}
```

Résultat : le nombre d'opérations est de l'ordre de M.

Cours 6 Tableaux (suite)

– tableaux à deux dimensions
– débordements
– chaînes de caractères

Tableaux à deux dimensions

Un tableau à deux dimensions permet d'accéder à des informations comme une image (en donnant pour chaque point (i, j) une valeur comme la couleur).

L'élément en ligne i et colonne j est noté

```
a[i][j]
```

Du point de vue mathématique, c'est une *matrice*. Voici par exemple comment on initialise un tableau $N \times N$ à 0.

```
int C[N][N];
int i, j;

for (i = 0; i < N, i++)
 for (j = 0; j < N; j++)
     C[i][j] = 0;
```

Les tableaux à plusieurs dimensions sont rangés en mémoire comme un tableau à une seule dimension. Si on range (comme en C) les éléments par ligne alors $a[i][j]$ est rangé en place $i \times m + j$ (pour un tableau $[0..n-1] \times [0..m-1]$).
Pour passer un tableau à deux dimensions en paramètre, on doit donner la deuxième dimension. Exemple : lecture d'un tableau à deux dimensions.

```
void Lire2(int a[][N]){
  int i,j;
  for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    scanf("%d",&a[i][j]);
}
```

L'expression `a[i]` désigne la ligne d'indice i du tableau. On peut donc aussi écrire :

```
void Lire2(int a[][N]){
  int i;
  for(i=0;i<N;i++)
    Lire(a[i]);
}
```

Exemple

Calcul des sommes par lignes d'un tableau.

```
void Somme2(int a[][N],int s[]){
  int i;
  for(i=0;i<N;i++)
    s[i]=Somme(a[i]);
}

int main(void){
  int a[N][N];
  int s[N];
  Lire2(a);
  Somme2(a,s);
  Ecrire(s);
  return 0;
}
```

Débordements

Le langage C ne réalise pas de contrôle sur les valeurs des indices. On doit faire attention à ne jamais appeler un indice hors des valeurs autorisées.

Sinon, il peut se produire des choses désagréables :

1. modifications de valeurs d'autres variables.
2. erreur d'execution

Exemples

L'affectation change la valeur d'une autre variable :

```
int main(void){
  int x=0;
  int a[5];
  a[5]=1;
  printf("%d\n",x);
  return 0;
}
```

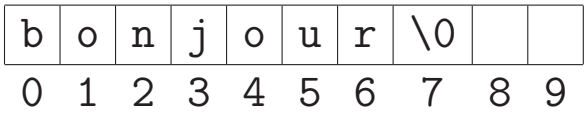
Le résultat est 1. Par contre, l'exécution de

```
int main(void){
  int a[100];
  a[1000]=1;
  return 0;
}
```

provoque une erreur à l'execution ('erreur de segmentation').

Chaînes de caractères

Une chaîne de caractères est un tableau de caractères terminé par le caractère `NUL` ('\0', entier zéro).



L'initialisation peut se faire par

```
char salut[10]="bonjour";
```

ou aussi

```
char *s="bonjour";
```

Attention : une chaîne initialisée comme `s` ci-dessus n'est plus modifiable.

La bibliothèque string.h

La bibliothèque **<string.h>** contient des fonctions sur les chaînes de caractères comme

```
char *strcpy(char *destination,
             const char *source)

int strcmp(char *s,char *t)

char *strcat(char * destination,
          const char *source);
int strlen(const char *s);
```

Pour les utiliser, on doit ajouter en tête de fichier

```
#include <string.h>
```

Recopie

Voici une fonction qui copie une chaîne de caractères. Ci-dessous le tableau `t` est recopié dans le tableau `s`.

```
void copie(char s[], char t[]) {
  int i=0;
  do  {
    s[i] = t[i];
    i++;
  } while (t[i-1] != '\0')
}
```

Ou encore,

```
void copie(char s[], char t[]) {

 for (i=0; t[i]!='\0';i++){
    s[i] = t[i];
 }
 s[i]='\0';
}
```

Ou encore, de façon plus compacte

```
void copie(char s[], char t[]) {
 int i=0;
 while (s[i] = t[i])
   i++;
}
```

On peut aussi utiliser

```
char *strcpy(char *destination,
             const char *source);
```

Concaténation

Pour concaténer deux chaînes,

```
void concat(char s[], char t[]){
     int i=0, j=0;
     while(s[i]!='\0')
        i++;
     while(t[j] !='\0'){
        s[i+j]=t[j];
        j++;
     }
     s[i+j]='\0';
}
```

on peut aussi utiliser

```
char *strcat(char * destination,
          const char *source);
```

Ainsi, avec

```
char d[50]="Bonjour";
char *s="a tous";
char *p;
```

après l'instruction

```
p=strcat(d,s);
```

le tableau `d` contient "bonjour a tous" et le pointeur `p` pointe sur `d`.

Voici un exemple de tableau à deux dimensions de caractères : les noms des jours de la semaine.

```
char Nom[7][10]={
 "lundi",
 "mardi",
 "mercredi",
 "jeudi",
 "vendredi",
 "samedi",
 "dimanche",
};
```

La fonction ci-dessous donne le nom du n-ième jour :

```
void lettres(int n) {
  printf("%s\n",Nom[n]);
}
```

Recherche d'un mot dans un texte

L'algorithme suivant recherche le mot x dans le texte y. Il rend le premier indice j de y tel que $y_j \cdots = x_0 \cdots x_n$ et -1 si on ne l'a pas trouvé..

```
int naif(char *x, char *y){
  int i=0,j=0;
  do {
    if(x[i]==y[j]){
      i++;j++;
    }
    else{
      j=j-i+1;
      i=0;
    }
  } while(x[i] && y[j]);
  if(x[i]==0) return j-i else return -1;
}
```

Cours 7 Structures

Une structure est, comme un tableau, un type construit. Avec une structure, on peut grouper dans une seule variable plusieurs variables de types différents. Ceci permet de créer des objets complexes.

Chaque objet possède ainsi des *champs* qui ont eux-même un type. Si p est une structure ayant des champs `x,y,z` on note `p.x, p.y, p.z` ces differentes valeurs.

L'affectation d'une structure recopie les champs.

Une structure peut être passée en paramètre d'une fonction et rendue comme valeur. On pourra par exemple grouper une chaîne

de caractères et un entier pour former une fiche concernant une personne (nom et âge).

```
struct fiche{
  char nom[10];
  int age;
};
struct fiche x;
```

Les fonctions

```
void lire(struct fiche *x){
  scanf(``%s'',x->nom);
  scanf(``%d'',&x->age);
}
void ecrire(struct fiche x){
  printf(``nom : %s\n
           age : %d\n'',x.nom,x.age);
}
```

permettent de lire et d'ecrire une fiche.

Nombre complexes

On peut ainsi déclarer un type pour les nombres complexes

```
struct complexe{
  float re;
  float im;
};

struct complexe z;

z.re = 1;
z.im = 2;
```

initialise $z = 1 + 2i$.

z | 1 | 2 |

On peut ensuite ecrire des fonctions comme

```
struct complexe somme(struct complexe u,
                      struct complexe v) {
  struct complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}

void afficher(struct complexe z){
  printf(``%f+ %fi\n'',z.re,z.im);
}
```

Exemple

On peut calculer le module d'un nombre complexe par la fonction :

```
float module(struct complexe z){
  float r;
  r= z.re*z.re + z.im*z.im;
  return sqrt(r);
}
```

Puis l'inverse :

```
struct complexe inverse(struct complexe z){
  struct complexe w;
  float c;
  float module = module(z);
  if (module != 0){
    c = module * module;
    w.re= z.re/c;
    w.im= -z.im/c;
    return w;
  }
  else exit(1);
}
```

Usage de `typedef`

On peut créer un alias pour un nom de type comme :

```
typedef struct {
  float re;
  float im;
} Complexe;
```

On obtient ainsi un alias pour le nom du type.

```
Complexe somme(Complexe u,Complexe v) {
  Complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}
```

La règle est que le nom de type défini vient à la place du nom de la variable.

On peut ainsi définir

```
typedef int Tableau[N][N];
```

et ensuite :

```
Tableau a;
```

Adresses

Pour représenter une adresse postale :

```
typedef struct {
  int num;
  char  rue[10];
  int code;
  char  ville[10];
} Adresse;
```

```
Adresse saisir(void) {
  Adresse a;
  printf("numero:"); scanf("%d",&a.num);
  printf("rue:"); scanf("%s",a.rue);
  printf("code:");scanf("%d",&a.code);
  printf("ville:"); scanf("%s",a.ville);
  return a;
}
```

```
void imprimer(Adresse a) {
  printf("%d rue %s\n%d %s\n",
           a.num,a.rue,a.code,a.ville);
}

int main(void) {
  Adresse a = saisir();
  imprimer(a);
  return 0;
}
```

Execution :

```
numero: 5
rue: Monge
code: 75005
ville: Paris

5 rue Monge
75005 Paris
```

Utilisation de plusieurs fichiers

En général, on utilise plusieurs fichiers séparés pour écrire un programme. Cela permet en particulier de créer des bibliothèques de fonctions réutilisables.

Pour l'exemple des nombres complexes, on pourra par exemple avoir un fichier `complexe.c` contenant les fonctions de base et les utiliser dans un autre fichier à condition d'inclure un en-tête qui déclare les types des fonctions utilisées.

On pourra créer un fichier `entete.h` et l'inclure par `#include "entete.h"`.

Le fichier complexe.c

```
#include "Entete.h"
void ecrire(Complexe z){
  printf("%f+i%f\n",z.re,z.im);
}
Complexe somme(Complexe u,Complexe v) {
  Complexe w;
  w.re = u.re + v.re;
  w.im = u.im + v.im;
  return w;
}
Complexe produit(Complexe u, Complexe v){
  Complexe w;
  w.re=u.re*v.re-u.im*v.im;
  w.im=u.re*v.im+u.im*v.re;
  return w;
}
float module(Complexe z){
  float r;
  r= z.re*z.re + z.im*z.im;
  return sqrt(r);
}
```

Le fichier Entete.h

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


typedef struct {
  float re;
  float im;
} Complexe;

void ecrire(Complexe z);

Complexe somme(Complexe u, Complexe v);

Complexe produit(Complexe u, Complexe v);

float module(Complexe z);

Complexe inverse(Complexe z);
```

Le fichier principal

```
#include "Entete.h"


Complexe exp(float teta){
  //calcule exp(i*teta)
  Complexe w;
  w.re=cos(teta);
  w.im=sin(teta);
  return w;
}
int main(void){
  ecrire(exp(3.14159/3));
  return 0;
}
```

On compile par

```
gcc -lm complexe.c exp.c
```

Résultat

```
0.500001+i0.866025
```

Cours 8 Appels récursifs

Une fonction peut en appeler une autre. Pourquoi pas elle-même ?

```
int puissance(int x,int n) {
 int y;

   if (n == 0)
    y = 1;
   else
    y = x * puissance (x,n-1);
   return y;
}

int main(void) {
  printf("%d\n",puissance(2,10));
  return 0;
}
```

Résultat : 1024

Récursivité

Un appel récursif est la traduction d'une *définition par récurrence*.

Par exemple, pour la fonction puissance, $y_n = x^n$ est défini par $y_0 = 1$, puis pour $n \geq 1$ par :

$$y_n = \begin{cases} 1 & \text{si } n = 0 \\ x \times y_{n-1} & \text{sinon} \end{cases}$$

Un autre exemple

La fonction factorielle

$$n! = n(n-1)\dots 1$$

est définie par récurrence par $0! = 1$ puis

$$n! = n \times (n-1)!$$

Programmation

```
int fact(int n) {
  return (n == 0) ?  1 : n * fact(n-1);
}
```

Comment ça marche ?

La mémoire est gérée de façon *dynamique* : une nouvelle zone est utilisée pour chaque appel de la fonction dans la *pile d'exécution*. De cette façon, les anciennes valeurs ne sont pas écrasées.

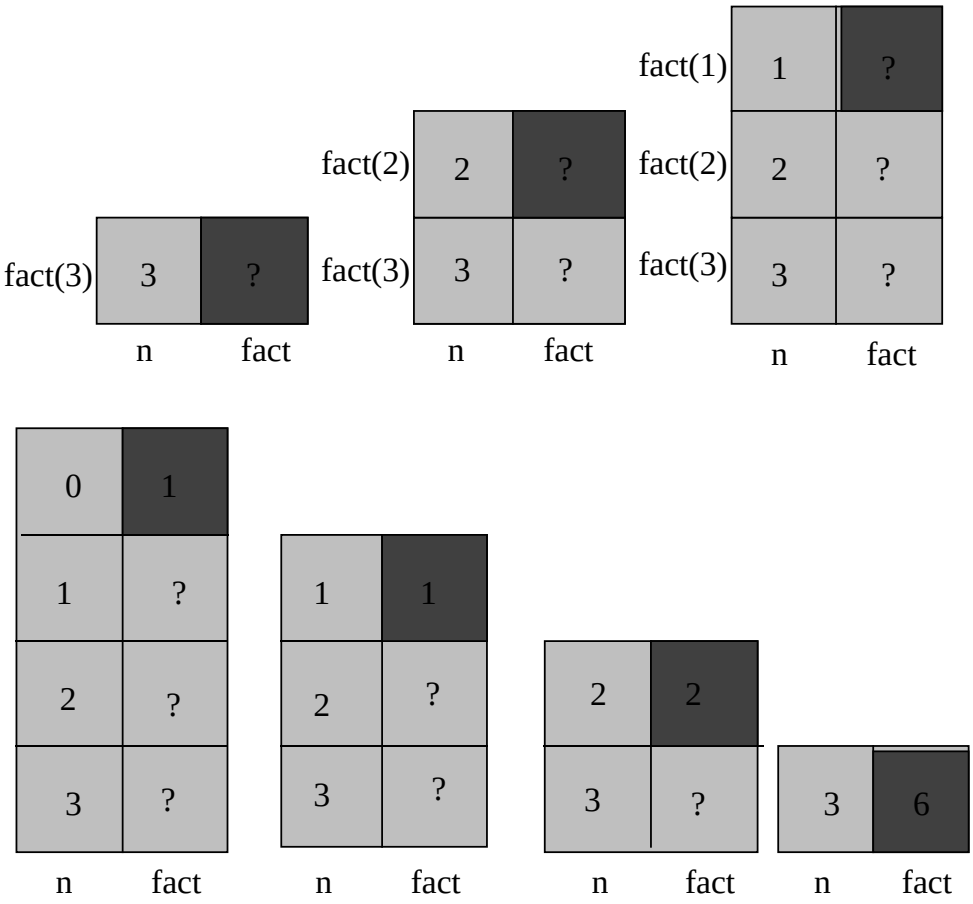


FIGURE 1 – La pile d'execution

Encore un exemple

La suite de Fibonacci est définie par $f_0 = f_1 = 1$ et pour $n \geq 1$

$$f_{n+1} = f_n + f_{n-1}$$

Le programme recursif ci-dessous traduit directement la définition :

```
int fibo(int n)
{
  return (n==0 || n==1)?
     1 : fibo(n-1)+fibo(n-2);
}
```

Mais cette fois le résultat est catastrophique : il faut plus de 10mn sur mon Mac pour calculer (en long) fibo(30) alors que le calcul est immédiat en itératif.

En fait, le temps de calcul est passé d'une fonction linéaire en n (en itératif) à une exponentielle en n (en récursif).

La représentation binaire

La représentation binaire d'un nombre entier x est la suite $\text{bin}(x) = (b_k, \ldots, b_1, b_0)$ définie par la formule :

$$\begin{aligned} x &= b_k 2^k + \cdots + b_1 2 + b_0 \\ &= 2(b_k 2^{k-1} + \cdots + b_1) + b_0 \end{aligned}$$

ce qui s'écrit aussi

$$\text{bin}(x) = (\text{bin}(q), r)$$

si q, r sont le quotient et le reste de la division entière de x par 2 :

$$x = 2q + r$$

On va voir qu'il est *plus facile* d'écrire le programme sous forme récursive : c'est une simple traduction des relations de récurrence.

Programme récursif

La fonction suivante écrit un nombre entier en binaire.

```
void binaire (int x) {
   if (x<=1)  printf("%d",x);
   else {
     binaire(x/2);
     printf("%d",x%2);
   }
}
```

ou encore plus court :

```
void binaire(int x) {
  if (x >= 2) binaire(x/2);
  printf("%d", x%2);
}
```

La multiplication du paysan russe

Par exemple, pour calculer 135×26, on écrit

135	26
270	13
540	6
1080	3
2160	1
3510	

A gauche on multiplie par 2 et à droite, on divise par 2. On somme la colonne de gauche sans tenir compte des lignes où le nombre de droite est pair. L'écriture récursive est très simple.

```
int mult(int x, int y){
 if(y==0) return 0;
 return x*(y%2) + mult(x*2,y/2);
}
```

C'est une variante de la décomposition en base 2.

Le jeu des chiffres

On cherche à réaliser une somme s donnée comme somme d'une partie des nombres $w_0, w_1, \ldots, w_{n-1}$.

On remarque qu'on peut se ramener à étudier deux cas :

1. il y une solution utilisant w_0, et donc une solution pour obtenir $s - w_0$ avec $w_1, \ldots, w_{n-1}$.
2. il y a une solution pour obtenir s avec $w_1, \ldots, w_{n-1}$.

De façon générale, le problème $(s, 0)$ se ramène à $(s - w_0, 1)$ ou $(s, 1)$.

On considère donc le problème (s, k) de savoir si on peut atteindre s avec $w_k, \ldots, w_{n-1}$. La fonction suivante rend 1 si c'est possible et 0 sinon.

```
int chiffres(int s, int k) {
  if (s == 0) return 1;
  if (s < 0 || k>=n) return 0;
  if (chiffres(s-w[k],k+1) return 1;
  else return chiffres(s,k+1);
}
```

Moralité

Pour utiliser la récursivité, on doit, en général se poser plusieurs questions :

1. quelles sont les variables du problème ?
2. quelle est la relation de récurrence ?
3. quels sont les cas de démarrage ?

L'utilisation de la récursivité permet d'écrire en général plus facilement les programmes : c'est une traduction directe des définitions par récurrence.

Cependant, les performances (en temps ou en place) des programmes ainsi écrits sont parfois beaucoup moins bonnes.

On peut dans certains cas commencer par écrire en récursif (prototypage) pour optimiser ensuite.

Cours 9 Algorithmes numériques

1. Types numériques en C
2. Arithmétique flottante
3. Calcul de polynômes
4. Le schéma de Horner
5. Calcul de x^n

Représentation des nombres réels

Représentation en *virgule flottante* (on dit aussi notation scientifique) :

$$6.02 \times 10^{23}$$

En général la représentation interne d'un nombre réel x se compose de :

1. La *mantisse* m comprise dans un intervalle fixe (comme $[0, 1]$)
2. L'*exposant* e qui définit une puissance de la base b.

qui sont tels que

$$n = m \times b^e$$

Le *nombre de chiffres significatifs* est le nombre de chiffres de la mantisse (prise en base 10). Ainsi, si $b = 2$ et que m a k bits, on a $\lfloor k \log_{10}(2) \rfloor$ chiffres significatifs.
Intuitivement, c'est le nombre de décimales exacts d'un nombre de la forme $0, \cdots$.

Le standard IEEE

1. simple précsion : 32 bits
2. double précision : 64 bits
3. quadruple précision : 128 bits

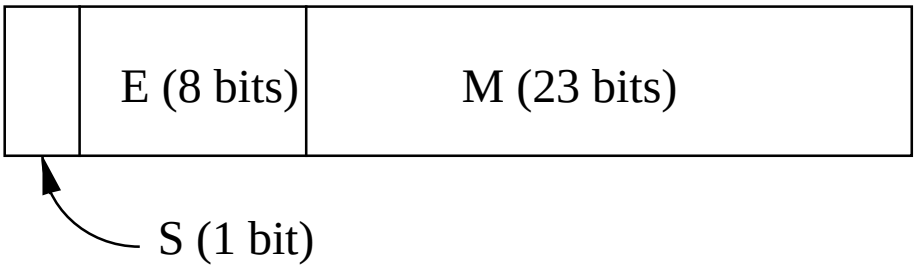


FIGURE 2 – Simple précision

1. Le premier bit est le **bit de signe**
2. Les 8 bits suivants sont l'**exposant** interprété en 127-excès
3. Les 23 bits restants forment la **mantisse** M. Elle est interprétée comme $1.M$

Nombre de chiffres significatifs : 23 bits, soit 6 chiffres significatifs puisque $10^6 < 2^{23} < 10^7$.

Exemple : Le mot de 32 bits

$$1\ 10000111\ 10100000000000000000000$$

représente le nombre

$$\begin{aligned} -2^{135-127} \times 1,625 &= -256 \times 1,625 \\ &= -416 \end{aligned}$$

qui se trouve être entier. Si l'exposant est 01111000 qui représente $(120)_{10} - (127)_{10} = (-7)_{10}$ en code $(127)_{10}$-excès, la valeur est :

$$-2^{-7} \times 1.625 \approx (.0127)_{10}$$

Le drôle de monde de l'arithmétique flottante

Les *erreurs d'arrondi* introduisent des comportements surprenants.

1. $n \oplus 1 = n$ dès que n est assez grand devant 1 : si $n = m \times b^e$ avec $0 < m < 1$, on a

   $$n \oplus 1 = b^e(m \oplus b^{-e}) = n$$

   dès que e est plus grand que le nombre de décimales de m. Par exemple, en C,

   ```
   float x=1.0e+8;
   x=x+1;
   printf(``%f\n'',x);
   ```

   affiche 100000000.000000.

2. La série $1 + 1 + 1 + \ldots$ converge.

3. L'addition n'est plus associative. Ainsi

   $$\begin{aligned}(1113. \oplus -1111.) \oplus 7.511 &= 2.000 \oplus 7.511 \\ &= 9.511\end{aligned}$$

   alors que

   $$\begin{aligned}1113. \oplus (-1111. \oplus 7.511) &= 1113. \oplus -1103. \\ &= 10.00\end{aligned}$$

Types numériques en C

On a en C trois types pour représenter des nombres :

1. Le type `int` pour les entiers.
2. Le type `float` pour les réels en simple précision.
3. Le type `double` pour les réels en double précision.

Les constantes de type réel s'écrivent sous la forme

```
3.1415  ou  87.45e+8
```

Le compilateur effectue les *conversions de type* nécessaires. Par exemple, la valeur de l'expression

```
2.0+1
```

est de type réel.

Formats

Pour lire ou écrire un nombre, on a les fonctions usuelles `scanf` et `printf` avec la syntaxe `printf("..",`*exp*) et `scanf("..",`*variable*) où le premier argument est un *format*. Il contient des caractères ordinaires et des spécifications de *conversion*. Chacune est de la forme % suivi d'un caractère.

– %d pour les entiers.

– %f pour les réels.

Des options permettent de préciser le format. Par exemple

`printf("%3d",n)` pour imprimer sur au moins trois caractères.

`printf("%.5f,x)` pour imprimer 5 décimales.

Exemples

Programme :

```
...
x= 10.999;
printf("%f\n",x);
printf("%25f\n",x);
printf("%25.5\n",x);
printf("%25.1f\n",x);
```

Résultat :

```
 10.999000
                 10.999000
                  10.99900
                       11.0
```

Calcul de polynômes

On utilise des polynômes

$$p(x) = a_n x^n + \ldots + a_1 x + a_0$$

pour :

1. représenter des entiers :

$$253 = 2 \times 10^2 + 5 \times 10^1 + 3$$

2. approximer des fonctions

$$sin(x) \approx \frac{x^5}{120} - \frac{x^3}{6} + x$$

3. tracer des courbes
4. ...

Programme en C

On suppose que le polynome $p(x)$ est représenté par le tableau

```
float a[N];
```

qui donne les coefficients

$$a[0], a[1], \ldots, a[N-1]$$

On pourra par exemple écrire

```
void ecrirePoly(float p[]){
  int i;
  printf("%.1f",p[0]);
  for(i=1;i<N;i++)
    printf("+%.1fx^%d",p[i],i);
  printf("\n");
}
```

L'exécution de

```
int main(void){
  float a[N]={1,0,1,0};
  ecrirePoly(a);
  return 0;
}
```

produit

```
1.0+0.0x^1+1.0x^2+0.0x^3
```

On peut aussi utiliser une *structure* qui conserve le degré du polynôme comme une information supplémentaire.

```
typedef struct  {
  float coeff[N];
  int degre;
    } Poly;
```

Pour entrer le polynome $x^3 + 2x - 1$, on écrit

```
#define N 20


Poly p;
p.degre = 3;
p.coeff[3] = 1;
p.coeff[2] = 0;
p.coeff[1] = 2;
p.coeff[0] = -1;
```

et cette fois

```
void ecrire(Poly p){
  int i;
  printf("%.1f",p.coeff[0]);
  for( i=1 ; i <= p.degre; i++)
    printf("+%.1f x^%d",p.coeff[i],i);
  printf("\n");
}
```

Opérations sur les polynomes

On a sur les polynomes les opérations de somme et de produit. Commençons par la somme. Si

$$\begin{aligned} p(x) &= a_n x^n + \cdots + a_1 x + a_0 \\ q(x) &= b_n x^n + \cdots + b_1 x + b_0 \end{aligned}$$

on a

$$p(x) + q(x) = (a_n + b_n)x^n + \cdots + (a_1 + b_1)x + (a_0 + b_0)$$

La somme se calcule de la façon suivante :

```
void  somme(float p[],float q[], float r[]){
 int i;
 for (i= 0; i<N; i++)
   r[i]= p[i]+q[i];
}
```

L'exécution de

```
int main(void){
  float a[N]={1,0,1,0}; float b[N]={0,1,-1,0};
  float c[N];
  somme(a,b,c);
  ecrire(c);
  return 0;
}
```

produit

```
1.0+1.0x^1+0.0x^2+0.0x^3
```

ou encore, avec l'autre structure de données

```
Poly somme(Poly p, Poly q) {
  int i;
  Poly r;
  for (i= 0; (i<= p.degre)&&(i<=q.degre); i++)
    r.coeff[i]= p.coeff[i]+q.coeff[i];
  if (p.degre > q.degre) {
    for (i= q.degre + 1; (i<= p.degre); i++)
      r.coeff[i]= q.coeff[i];
  }
  if (p.degre < q.degre) {
    for (i= p.degre + 1; (i<= q.degre); i++)
      r.coeff[i]= p.coeff[i];
  }
  r.degre= (p.degre>q.degre)? p.degre:q.degre;
  for (i= r.degre; (i>=0) && (r.coef[i] == 0); i--)
    r.degree--;
  if (r==-1) r++; /* polynome nul */
  return r;
}
```

Pour le produit, on a

$$p(x)q(x) = c_{2n}x^{2n} + \cdots + c_1 x + c_0$$

avec

$$c_k = \sum_{i+j=k} a_i b_j$$

Pour le calcul, on suppose que le degré de pq est au plus égal à $N-1$.

```
void produit(float p[], float q[], float r[]) {
```

```
 int i,k,s;
 for( k= 0; k<N; k++) {
   s =0;
   for (i=0; i<=k ; i++)
     s =s+p[i]*q[k-i];
   r[k]=s;
 }
}
```

Avec l'autre structure de données, on obtient une fonction plus compliquée.

```
int produit(Poly p, Poly q, Poly* r) {
  int i,j,k;
  r->degre = p.degre+q.degre;
  if (r->degre >= N) return 1;
  for (k = 0; k<= r->degre; k++) {
    r->coef[k] = 0;
  }
  for (i = 0; i<= p.degre; i++) {
    for (j = 0; j<= q.degre; j++) {
       r->coef[i+j] += p.coef[i] * q.coef[j];
    }
  }
  return 0;
}
```

La fonction renvoie 0 si le produit a été effectué et 1 sinon.

Evaluation en un point

On veut évaluer le polynôme

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

au point $x = k$.

On veut que le calcul soit :

1. rapide
2. direct : sans consulter une table de valeurs.

Première méthode

On évalue la somme en calculant pour $i = 1, \ldots, n$:

1. La puissance k^i
2. Le produit $a_i k^i$

Pour que le calcul de k^i soit rapide, on aura avantage à calculer **de droite à gauche** :

$$1, k, k^2, \ldots, k^n$$

La méthode naïve

Le calcul de la valeur de $p(x)$ pour $x = k$ se fait par la fonction suivante :

```
float eval(float a[], float k) {
 float val,y ;
 int i;
  val= 0;
  y= 1;
  for (i= 0; i<N; i++)
  {
    val= val + a[i]*y;
    y = y*k;
  }
 return val;
}
```

Deuxième méthode : le schéma de Horner

On fait le calcul en utilisant la factorisation :

$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$$

Le calcul se fait **de gauche à droite**.
Pour $i = n-1, \ldots, 1, 0$ on fait :

1. multiplier par k
2. ajouter a_i

Programme en C

```
float horner(float a[], float k) {
  float val;
  int i;
  val = a[N-1];
  for (i = N-2; i>=0; i--)
     val = val*k+a[i];
 return val;
}
```

Comparaison des méthodes

1. Nombre d'opérations :

	Naïve	Horner
additions	n	n
multiplications	$2n$	n

2. La méthode de Horner ne nécessite pas de connaître les indices i des coefficients a_i : application au calcul de la valeur d'un nombre écrit en base k.
3. La méthode de Horner a une définition *récursive* :

$$p(x) = q(x)x + a$$

Grands nombres

Pour utiliser des nombres de taille arbitraire, il est facile d'écrire des fonctions qui réalisent les opérations de base. On pourra par exemple déclarer

```
typedef struct nombre{
  char chiffres[N];
  int taille;
} Nombre;
```

puis, pour lire caractère par caractère.

```
void lire(Nombre *x){
  int i=0; char c; char tab[N];
  do{
    scanf("%c",&c);
    tab[i] = c;
    i = i+1;
  } while('0'<= c && c <= '9');
  x->taille = i;
  for(i=0;i<x->taille;i++)
    x->chiffres[i] = tab[x->taille-i-1];
}
```

```
void ecrire(Nombre x){
  int i;
  for(i=x.taille; i>0;i--)
    printf("%c",x.chiffres[i-1]);
  printf("\n");
}
// chiffre to int
int c2i (char c){
  if ('0'<= c  && c <= '9') return c-'0';
  else return -1;
}

int somme(Nombre x, Nombre y, Nombre *az){
  int i,t,r=0;

  for(i=0;
      (i<x.taille || i<y.taille || r!=0);
      i++)
  {
   if (i<x.taille && i<y.taille)
      t= c2i(x.chiffres[i])+
         c2i(y.chiffres[i])+ r;
   else if (i<x.taille)
      t= c2i(x.chiffres[i])+ r;
   else if (i<y.taille)
      t= c2i(y.chiffres[i])+ r;
   else
      t = r;
   r=t/10;
   if (i == N) return 1; //addition impossible
   else az->chiffres[i]=t%10+'0';
```

```
  }
  az->taille = i;
  return 0;
}
```

Evaluation de x^n

Pour évaluer des polynômes particuliers, on peut parfois aller plus vite que dans le cas général.

Ainsi le monôme x^n ne nécessite pas n multiplications.

Exemple :

x^{13} peut se calculer en 5 multiplications :

1. on calcule $x^2 = x \times x$
2. on calcule $x^4 = x^2 \times x^2$
3. on calcule $x^8 = x^4 \times x^4$
4. on calcule $x^{12} = x^8 \times x^4$
5. on obtient $x^{13} = x^{12} \times x$

Cette méthode revient à écrire l'exposant en base 2.

Progamme en C

Avec utilisation de la récursivité : on calcule la décomposition en base 2 sous forme récursive (utilisant en fait un schéma de Horner).

```
float puissance (float x, int n) {
  float y;
  if (n== 0) return 1;
  else if (n== 1) return x;
  else if (n%2==1) return x*puissance(x,n-1);
  else {
    y = puissance(x,n / 2);
    return y*y;
  }
}
```

Calcul de complexité

Soit $T(n)$ le nombre de multiplications effectués. On a $T(0) = T(1) = 1$.

On a pour tout entier n pair ou impair

$$T(n) \leq 2 + T(n/2)$$

On va montrer par récurrence que ceci entraine

$$T(n) \leq 2 \log_2 n$$

Ceci est vrai pour $n = 1$. Ensuite, on a

$$\begin{aligned} T(n) &\leq 2 + T(n/2) \\ &\leq 2 + 2 \log_2(n/2) = 2 + 2 \log_2 n - 2 \\ &\leq 2 \log_2 n \end{aligned}$$

Cours 10 Résolution d'équations

1. Généralités
2. Dichotomies
3. La méthode de Newton
4. Equations différentielles

Généralités

On va étudier quelques méthodes de résolution d'équations de la forme :

$$f(x) = 0$$

où f est une fonction (polynôme ou autre) et x une inconnue réelle. On se place dans le cas où on ne peut pas obtenir de formule explicite pour x. On en cherche une approximation obtenue en appliquant une *itération* :

$$x_{n+1} = g(x_n)$$

ou

$$x_{n+1} = g(x_n, x_{n-1})$$

Pour programmer, on utilise des fonctions de la bibliothèque `math` comme `fabs(),sin(),...` Il faut mettre dans l'en tête `#include math.h` et compiler avec l'option `>gcc -lm fichier.c`.

Dichotomies

C'est la méthode la plus simple. Elle s'applique dès que f est *continue* (i. e. que ses valeurs n'ont pas de saut).

On part de deux valeurs x_1, x_2 telles que

$$f(x_1)f(x_2) < 0$$

et on pose $x_3 = (x_1 + x_2)/2$. Si $f(x_1)f(x_3) < 0$, on pose $x_4 = (x_1 + x_3)/2$, sinon on pose $x_4 = (x_3 + x_2)/2$.

L'erreur $e_n = \mid x_n - x_{n-1} \mid$ vérifie :

$$e_n < e_1/2^n$$

On dit que la méthode est *linéaire* : cela signifie en pratique que le nombre de décimales exactes augmente proportionnellement à n.

Programme

On choisit une précision ϵ et on teste la longueur de l'intervalle courant $[a, b]$.

```
float dicho(float a, float b, float eps){
 float x,fa,fx;

  fa = f(a);
  do {
   x = (a+b)/2;
   fx = f(x);
   if (fa*fx < 0)  b = x;
   else {
      a = x; fa = fx;
    }
  } while ((b-a) > eps);
 return x;
}
```

Variante

Si la fonction f est croissante, on peut écrire de façon plus simple :

```
float dicho(float a, float b, float eps) {
 float x,fa,fx;

  do {
    x = (a+b)/2;
    fx = f(x);
   if (fx > 0)
     b = x;
    else
     a = x;
  } while ((b-a) > eps);
  return x;
}
```