Les bases de l'utilisateur du système Unix/Linux

Module d'informatique - Licence DSM ENS Lyon

TD n° 1 — 2005-06

28 septembre 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Premiers pas sur un système d'exploitation type Unix, notion de répertoire/fichier, éditeur de fichier.

1 Historique

UNIX est né en 1969 dans les laboratoires de BELL (K. Thompson). La réécriture complète d'UNIX en C date de 1973 dans le but de rendre le système le plus largement portable sur différentes plate-formes. Des licences d'UNIX ont été distribuées aux universités qui ont ainsi pu contribué au développement du système et à sa diffusion. Au début des années 80 ATT, propriétaire du système, se lance dans sa commercialisation. Ceci explique que chaque UNIX livré par les constructeurs de matériel porte un nom spécifique : HP-UX (Hewlett-Packard), AIX (IBM), SUN-OS (SUN), etc.

En 1994-1995 le système UNIX est cédé au groupe d'utilisateurs X-OPEN GROUP dont le but est de définir et promouvoir les systèmes ouverts. Ceci rend UNIX accessible à tout utilisateur respectant les standards des systèmes ouverts. L'historique d'UNIX permet d'expliquer à la fois la non-« unicité » du système mais également sa très grande richesse.

Linux, développé par L. Torvalds dans les années 90, constitue une implémentation libre des concepts unixiens. Dans ce TD, les différents « unices » (Linux, HP-UX, Solaris, etc.) seront regroupés sous le terme générique d'UNIX.

2 Concepts

UNIX est un système *multi-tâches*, *multi-utilisateurs*, avec gestion de mémoire virtuelle et travaillant sur une architecture de fichiers hiérarchisés. Les deux principales caractéristiques sont :

– la prépondérance de la *notion de fichier*. Par exemple, chaque commande du système, chaque périphérique correspondent à des fichiers. Ces fichiers se présentent sous la forme d'une *arborescence* (Fig. 1).

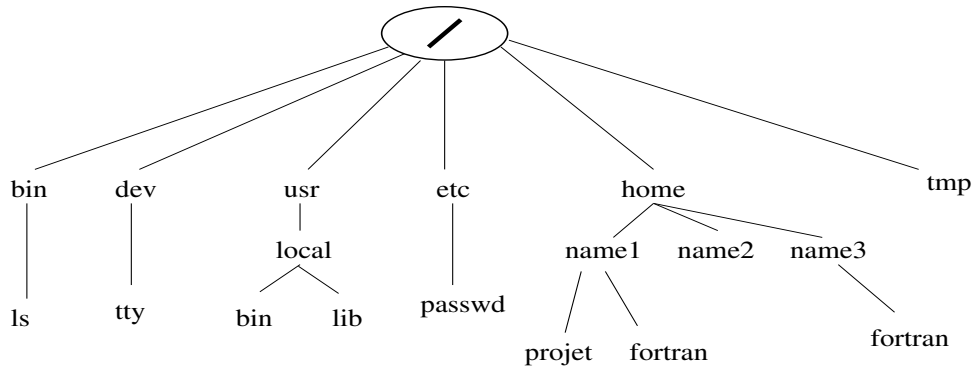

FIG. 1 – Exemple d'arborescence.

On distingue deux entités : les *fichiers* et les *répertoires* (*directory*). Ces derniers permettent le « rangement » des fichiers. Le répertoire au sommet de l'arborescence est repéré par le symbole '/'. Chaque utilisateur du système possède un répertoire personnel baptisé le *home*.

– l'importance de la *notion de tâche* ou *processus* (*process*). Toute action génère en machine un processus identifiable par un numéro. UNIX permet la communication entre *process* et également l'héritage d'un *process* père vers un *process* fils.
L'accès au numéro et à l'état d'un process se fait par la commande `ps` (voir § 5.3).

3 Session UNIX

3.1 Débuter une session UNIX : se logger...

Pour entrer sur le système UNIX on vous invite à saisir votre nom de *Login* et votre mot de passe :

`Login` : Entrer votre nom de Login

`Password` : Entrer votre mot de passe

Important : à la première utilisation, un mot de passe systématique est attribué à chaque utilisateur par l'administrateur. Pour changer de mot de passe il faut utiliser la commande `passwd`. Cette commande demande l'ancien mot de passe et demande deux fois le nouveau mot de passe.

Très important : UNIX est *case-sensitive* (sensible aux différences minuscules/majuscules). Par exemple les fichiers `toto.c` et `ToTo.c` sont bien *deux* fichiers différents.

3.2 L'utilisateur

La procédure de *login* permet d'identifier chaque utilisateur du système, et de lui accorder les droits appropriés (p.ex. le droit de lire *son* courrier, mais pas celui de son voisin). Les mots de passe doivent *absolument* rester personnels : un responsable n'a *jamais* besoin de vous le demander pour régler un problème, et un collègue n'a pas à le connaître.

3.3 L'interpréteur de commandes : le *SHELL*

Le *shell* gère notamment :
– les entrées-sorties
– la définition des variables
– le rappel des commandes précédentes
– la création de synonymes (*alias*)
– les évaluations arithmétiques
– les structures conditionnelles et de répétition.

Il est également possible de créer un fichier contenant une liste d'instructions (*script UNIX*). Pour information, vous pourrez être confrontés à différents types de *shells* : le Bourne *shell* (sh, du nom de son créateur), le C-*shell* (csh) et sa version améliorée, le tcsh, le Korn-*shell* (ksh, du nom de son créateur)... Ces *shells* diffèrent par leurs fonctions internes ou les fonctions de programmation élaborées. Un *shell* est lancé automatiquement au moment de la connection (*login*) mais un nouveau *shell* peut-être appelé à tout moment au cours d'une session.

Exercice : Ouvrez une session sur votre machine.

4 L'environnement

Lorsque vous débutez une session UNIX, celle-ci se déroule dans un environnement défini. Par défaut, vous utilisez un système Linux (version PC d'UNIX) de type DEBIAN et vous travaillez dans un environnement appelé KDE (voir Fig. 2).

L'idée est de travailler avec un système de fenètres, dans un environnement proche de ce que l'on peut trouver avec Windows. Vous disposez donc d'un « bureau » matérialisé par un *fond d'écran* (sur lequel peuvent se trouver plusieurs icônes) et une *barre des tâches* ou tableau de bord en bas.

Les fonctions et utilitaires les plus importants sont accessibles sur le tableau de bord (de gauche à droite) :
– icône KDE (équivalent du *Démarrer* de Windows) : menu déroulant des différentes fonctions
– icône *show desktop* : affiche le fond d'écran en msquant toutes les fenêtres ouvertes

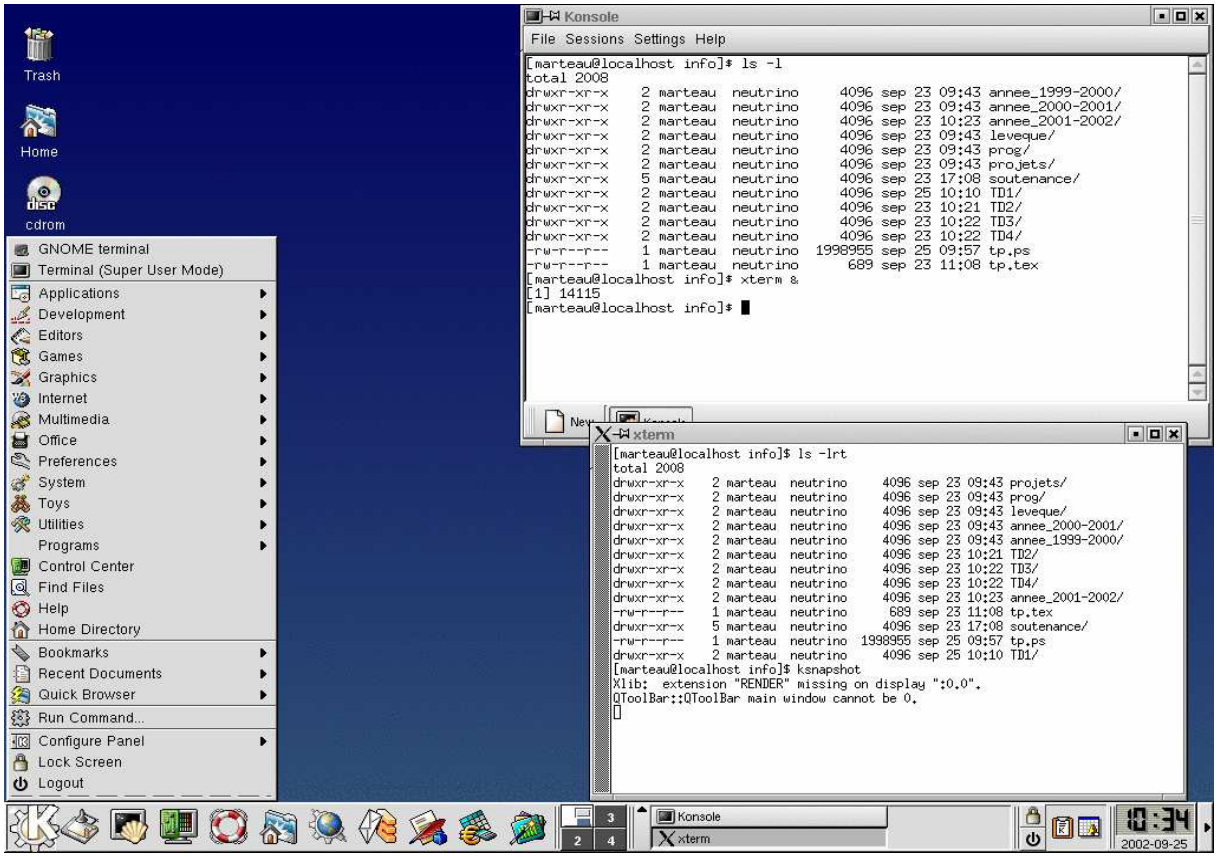

FIG. 2 – Bureau KDE.

- icône *terminal* : lance un terminal
- icône *control center* : permet de configurer l'environnement
- icône *help* : aide en ligne
- icône *home directory* : se place sur le répertoire *home* et en liste le contenu
- icône *konqueror* : navigateur WEB
- icône *kmail* : courrier électronique
- icône *kword* : éditeur de texte (type *Word*)
- icône *kspread* : tableur (type *Excel*)
- icône *kpresenter* : éditeur de présentation (type *Powerpoint*)

La signification de chaque icône s'affiche lorsque l'on place le curseur dessus.

Viennent ensuite quatre icônes représentant chacune un *espace de travail* (*workspace*). Ces différents espaces de travail possèdent le même agencement que celui décrit ci-dessus et permettent entre autre de ne pas se laisser envahir par la multiplication des fenètres. Une fenètre ouverte dans un espace de travail peut se déplacer ou se dupliquer dans un autre espace de travail.

À partir du menu déroulant (accessible par l'icône KDE en bas à gauche) il est possible de lancer un certain nombre d'utilitaires et d'applications. Vous devez trouver les menus suivants (la liste n'est pas exhaustive et dépend de la configuration) : *editors*, *games*, *graphics*, *internet*, *office*, *system*, *utilities*, *control center*, *find files*, *help*, *home*, *bookmarks*, *recent documents*, *quick browser*, *run command*, *configure panel*, *lock screen*, *logout*...

Noter que les applications que l'on retrouve dans la barre des tâches se retrouvent dans ce menu. Tout est affaire de configuration. Tout cet environnement a pour but de rendre plus accessibles les outils d'UNIX dans un cadre plus « convivial ». Ainsi il est possible de se déplacer dans l'arborescence des fichiers au moyen du *file manager* qu'on lance à partir de l'icône *home directory*.

Pour bien comprendre le fonctionnement d'UNIX, nous allons détailler le passage des commandes de base à partir d'un *terminal*.

Exercice : Ouvrir un terminal à partir de la barre des tâches (ou du menu déroulant).

5 Passer une commande

Pour plus de détails sur la jungle des commandes UNIX, vous pouvez aller par exemple sur http://snovae.in2p3.fr/ycopin/enseignements/info_DEA.html.

5.1 Format

Une commande UNIX revient à taper le nom d'une éxécutable qui va être recherché dans un des répertoires du système de fichiers (p.ex. `/bin/` ou `/usr/bin/` ou `/usr/local/bin/`). Les commandes UNIX admettent des arguments séparés entre eux par des blancs et qui sont traités comme des options s'ils sont précédés par un signe '`-`' ou comme des paramètres sinon. Le format général d'une commande est donc le suivant :

```
commande [-options [paramètres]] [paramètres]
```

Une option est en général une lettre. Plusieurs options peuvent être collées derrière le '`-`'. Il est possible de passer plusieurs commandes sur une seule ligne en les séparant d'un '`;`'.

5.2 Où passer les commandes ?

Le moyen le plus simple, le plus rapide et le plus direct d'interagir avec le système Unix est l'utilisation d'un terminal : console ou terminal-X. Une fois le terminal lancé, on tape la commande après le symbole d'invite.

La commande `ls` permet de lister le contenu d'un répertoire (i.e. les fichiers et les autres répertoires qu'il contient). Tester cette commande et les différentes options :

- `-l` en plus du nom, affiche le type du fichier, les permissions d'accès, le nombre de liens physiques, le nom du propriétaire et du groupe, la taille en octets, et l'horodatage (de la dernière modification, sauf si une autre date est réclamée par les options `-c` ou `-u`).
- `-a` affiche tous les fichiers des répertoires, y compris les fichiers commençant par un '`.`' (fichiers cachés).
- `-t` trier le contenu des répertoires en fonction de la date et non pas en ordre alphabétique. Les fichiers les plus récents sont présentés en premier.
- `-r` inverser le tri du contenu des répertoires.
- `-R` afficher récursivement le contenu des sous-répertoires.

Noter la différence entre les deux dernières options (minuscule/majuscule).

Exercice : Essayer d'afficher le contenu du répertoire courant, en format long et par ordre chronologique.

Les métacaractères gérés par le *shell* permettent de substituer tout ou partie d'une chaîne de caractères. Par exemple '`*`' remplace un groupe de caractères, éventuellement la chaîne vide.

5.3 Les *process*

L'accès au numéro d'un process généré en machine se fait par la commande `ps` à laquelle on peut associer les options `-e`, `-f`, `-l`.

L'éxécution d'une commande peut se faire « en direct » ou en « tâche de fond ». Dans le deuxième cas, on peut continuer à travailler à partir du terminal qui a servi à lancer la commande. Pour réaliser ceci, on doit faire suivre la commande du caractère '`&`'.

Exercice : Pour écrire un programme, on utilise un éditeur de texte. Il en existe plusieurs : `xedit`, `xemacs`... Pour lancer cet éditeur à partir d'un terminal, il suffit de taper par exemple `xedit` et de valider. Sans '`&`' final, le terminal reste bloqué, jusqu'a ce que vous ayez quitté l'application qui vous rend alors « la main » (i.e. le terminal).

Une autre façon de procéder est de taper dans le terminal `ctrl-Z` puis `bg`. Ceci garde l'application « vivante » en tâche de fond. Le numéro qui apparait est le numéro de process.

1. Relancer `xedit` directement en tâche de fond. Le terminal doit afficher deux informations : le numéro de la commande dans l'historique et le numéro de process.
2. Lancer un autre `xedit` puis quitter le premier. Quelles sont les infos affichées par le terminal ?
3. Taper `ps -ef`. Quels sont les process qui tournent ?

5.4 Historique des commandes

Le *shell* permet de rappeler et modifier une commande antérieure. La commande `history` permet de lister les commandes précédentes en leur attribuant un numéro d'ordre. Le rappel de la commande numéro N se fait en tapant `!N`. On peut également rappeler la dernière commande en tapant `! !`. Enfin si l'on tape `!` suivi d'une chaîne de caractères, alors le système exécute la dernière commande commençant par cette chaîne de caractères.

Exercice :

1. Ré-exécuter la commande `ls -r` à partir de son numéro dans l'historique.
2. Exécuter la dernière commande commençant par un 'g'.

5.5 Le manuel

Certaines commandes UNIX ont un nom qui évoque leur action. Ce n'est malheureusement pas général. Il existe cependant un manuel d'aide en ligne très performant et très développé (mais souvent assez cryptique). On y accède en tapant la commande `man commande-dont-on-veut-le-manuel`.

Exercice : À quoi sert la commande `cat` ?

6 Le système de fichiers

6.1 L'arborescence

Sous UNIX les fichiers sont rangés dans un système arborescent inversé (la racine, repérée par le symbole '`/`' se situe au plus haut niveau). L'arborescence résulte de la succession de répertoires d'autant plus bas qu'ils sont éloignés de la racine. Un nom de répertoire commençant par '`/`' décrit l'accès à ce répertoire depuis la racine (on parle alors de chemin absolu). Chaque utilisateur est positionné sur un répertoire dit répertoire courant. Le métacaractère désignant le répertoire courant est le point '`.`' Les autres métacaractères utiles sont :

`..` désigne le répertoire « père » du répertoire courant

`~` désigne le répertoire `home` de l'utilisateur

`/` désigne la racine du système

Ces métacaractères sont repris par les utilitaires de l'environnement permettant de se déplacer dans l'arborescence.

6.2 Commandes concernant l'arborescence

`pwd` permet de connaître le répertoire courant

`mkdir [répertoire]` crée un répertoire

`rmdir [répertoire]` détruit un répertoire (dans la mesure ou celui-ci est vide de tout fichier)

`cd [répertoire]` permet de se positionner dans un répertoire

`du [-as] [répertoire]` donne le nombre de kilo-octets occupés par les fichiers du répertoire et des sous-répertoires éventuels.

Exercice : Créer le répertoire `prog-C` dans votre `home`. Créer les répertoires `TD1/` et `TD2/` dans `prog-C`. Placez-vous dans le répertoire `TD1/`. Taper `pwd`. Que se passe-t-il ?

6.3 Propriétés des fichiers

Pour créer un fichier vide on peut utiliser la commande : `touch nomdu fichier`. Le nom d'un fichier peut contenir jusqu'à 256 caractères (il faut en exclure les caractères ayant un sens pour le *shell* tels que '`$`', '`/`', '`?`'). On peut faire suivre le nom de chaque fichier par une extension précédée d'un point. Par exemple, certains compilateurs n'acceptent pas les fichiers C n'ayant pas l'extension `.c` (p.ex. `toto.c`).

Exercice : Créer un fichier `toto.c` dans le répertoire `TD1/`. Lister le contenu de ce répertoire en format long.

À chaque fichier ou répertoire sont associés :
- un mode (chaîne de 1+9 caractères)
- un entier désignant le nombre de sous-répertoires d'un répertoire (1 désigne un fichier, 2 un répertoire sans sous-répertoire et 2+n un répertoire avec n sous-répertoires)
- le nom de l'utilisateur ayant créé le fichier ou le répertoire
- le nom du groupe auquel cet utilisateur appartient
- la taille en octets du fichier
- la date et l'heure de dernière modification
- le nom suivi du signe '`/`' dans le cas d'un répertoire

6.4 Protection des fichiers

Le premier caractère du *mode* d'un fichier (ou répertoire) peut prendre les valeurs suivantes : '–' pour un fichier ordinaire, '`d`' pour un répertoire.

Champ de protection d'un fichier. Ce sont les neuf caractères suivants du mode d'un fichier (ou répertoire). La protection d'un fichier s'effectue à trois niveaux : vis-à-vis du propriétaire du fichier (noté `u`), vis-à-vis du groupe auquel appartient le propriétaire du fichier (noté `g`) et vis-à-vis des autres utilisateurs (notés `o`). Pour chacun il existe trois types de droits sur le fichier :
- `r` droit en lecture (possibilité de lister le contenu)
- `w` droit en écriture (possibilité de modifier le contenu ou de créer des fichiers dans le cas d'un répertoire)
- `x` droit en exécution (possibilité d'exécuter un programme par exemple)

Le champ de protection s'organise comme une suite de 3×3 caractères : `rwx` pour `u`, `rwx` pour `g`, `rwx` pour `o`. Si un – remplace une des lettres de ce champ, cela signifie que le droit est absent pour le niveau concerné.

Exemple : `rwxr-xr- tata.c` signifie que l'utilisateur a tous les droits sur le fichier, que les membres du groupe peuvent lire et exécuter ce fichier et que les autres utilisateurs ne peuvent que le lire.

Le changement des droits d'un fichier ou d'un répertoire s'effectue à l'aide de la commande :
`chmod [[destinataire] [opérateur] [mode]] nomdufichier`
où l'opérateur + autorise un accès, l'opérateur – interdit un accès, et l'opérateur = autorise exclusivement l'accès indiqué. `destinataire` prend les valeurs `u`, `g` ou `o` et `mode` les valeurs `r`, `w` ou `x`.

Exercice : Rendre le fichier `toto.c` accessible en lecture/écriture aux membres du groupe mais pas aux autres utilisateurs.

6.5 Opération sur les fichiers

`rm [-ir] nomdufichier` Effacer un fichier.
- **`-i`** demande confirmation avant l'effacement,
- **`-r`** permet d'effacer récursivement tout un répertoire, **DANGER !** Faites également attention aux commandes du type `rm *` qui vous effacent tous les fichiers du répertoire courant.

`mv [-i] fichier-source fichier-cible` Déplacer un fichier.
- **`-i`** demande confirmation avant déplacement. **ATTENTION**, `fichier-source` n'existera plus.

`cp [-i] fichier-source fichier-cible` Copier un fichier.
- **`-i`** demande confirmation avant l'écrasement d'un fichier s'il existe déjà,
- **`-r`** permet de copier récursivement tout un répertoire.

`file fichier` Déterminer la nature d'un fichier.

`diff fichier1 fichier2` Lister la différence existant entre deux fichiers (très utile lorsque l'on a effectué de petites modifications sur un programme).

`rmdir nomdurepertoire` Effacer un répertoire vide (il faut préalablement en effacer le contenu avec la commande `rm`).

`find répertoire [critères de recherche]` Rechercher un fichier. La recherche s'effectue de manière récursive à partir du répertoire spécifié et avec les critères indiqués, par exemple `-name nomdufichier` recherche en fonction du nom du fichier.

`grep` Rechercher une chaîne de caractères dans des fichiers (très utile pour retrouver par exemple le fichier contenant une routine que l'on souhaite insérer dans un programme).

`lpr [-#n -Pnomimprimante] nomdufichier` Imprimer des fichiers.

`-#n` indique que l'on désire imprimer n copies. Il faut spécifier le nom de l'imprimante sur laquelle on désire effectuer l'impression (exemple : lw109 pour l'imprimante laser de la salle 109, lw106 pour la salle 106)

Deux autres commandes relatives à l'impression sont très utiles :

`lpq [-Pnomimprimante] [user ou jobnumber]` renvoie l'état de la queue d'impression pour l'imprimante spécifiée (permet de savoir si le travail d'impression que l'on a lancé est en attente ou en cours d'exécution)

`lprm [-Pnomimprimante] [user ou jobnumber]` tue le job spécifié sur l'imprimante mentionnée. Cette commande teste si le job appartient bien à l'utilisateur.

Exercice : Copier dans le répertoire `TD1/` le fichier toto.c en tata.c. Déplacer tata.c dans le répertoire `TD2/`. Rechercher à partir de votre *home* tous les fichiers ayant une extension `.c`.

7 Édition de fichiers

Les fichiers « texte » sont des fichiers simples où du texte non-formatté est stocké à l'aide d'un codage de bas-niveau sous la forme d'une suite de caractère (p.ex. code ASCII ou Unicode). *A contratio*, un fichier « non texte » est appelé « fichier binaire », dans le sens où les bits contenus dans le fichier ne peuvent pas être représentés pas une simple suite de caractères.

Les fichiers texte, tels que les fichiers *source* des langages de programmation, peuvent être édités à l'aide d'un *éditeur de texte* (tandis que les fichiers binaires ne se modifie qu'à l'aide d'un logiciel approprié, p.ex. un fichier `doc` avec Microsoft Word[1]).

Il existe de nombreux éditeurs de texte, plus ou moins performants, sur tous les systèmes :
- p.ex. `notepad` sous Windows,
- p.ex. `textedit` sous Mac OS X,
- p.ex. `xedit`, `nedit`, `emacs` & Co., `vi` & Co. sous Unix.

Ces éditeurs diffèrent par leurs possibilités évoluées (au-delà de la simple édition de texte), entre autres :
- ouverture simultanée de plusieurs fichiers,
- définition de macro-commandes,
- fonctionnalités avancées de recherche et remplacement de texte ou motifs (notamment par l'usage des expressions régulières),
- interaction avec des programmes externes sur les fichiers (compilateurs notamment),
- indentation automatique pour certaines extensions de fichiers, comme le code source de divers langages de programmation,
- coloration syntaxique.

Même si des éditeurs évolués (p.ex. `emacs` ou `vi`) peuvent s'avérer difficiles à prendre en main, il est judicieux de les utiliser le plus rapidement possible.

Exercice : Éditer le fichier `TD1/toto.c` avec l'éditeur de texte de votre choix. Le sauvegarder et fermer l'éditeur de texte. Recommencer.

[1] Qui n'est pas un éditeur de texte, mais un *traitement* de texte.

Premiers pas en C

Module d'informatique - Licence DSM ENS Lyon

TD n° 2 — 2004-05

25 janvier 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : source, compilation & exécution, structure, déclarations

1 Méthodologie de programmation

L'utilisation d'un langage de programmation tel que le Fortran, le C, le C++, etc., permet de générer un programme exécutable par la machine (un fichier binaire appelé l'*exécutable*) à partir d'un simple fichier texte (un fichier « ASCII » dit fichier *source*) indiquant précisément et dans une syntaxe particulière la marche à suivre.

Puisque l'on n'écrit pas du code uniquemenent pour le plaisir, mais pour faire exécuter par la machine des tâches précises, la programmation peut se décomposer en quelques grandes étapes :

Analyse : il s'agit de bien préciser, dans un langage libre, le *cahier des charges*, c-à-d. ce que l'on veut que le programme fasse, et les méthodes générales à utiliser. P.ex., « le programme devra afficher la chaîne de caractères "*Hello world !*" à l'écran ».

Pseudo-code : il s'agit maintenant de décrire en détail les différentes étapes du programme – l'*algorithme* – dans un langage plus formel. P.ex.,

Exemple 1 – Pseudo-code

```
programme (pas d'argument d'entrée)
afficher  "Hello world!"
fin  (exécution réussie)
```

Codage : il faut alors traduire l'algorithme décrit précédemment dans le langage de programmation proprement dit, dans notre cas en C. P.ex.,[1] :

Exemple 2 – Premier programme

```
int␣main()
{
␣␣␣␣printf("Hello␣world!\n");
␣␣␣␣return(0);
}
```

Debug : après avoir généré l'exécutable et l'avoir lancé sur la machine, il faut s'assurer que le programme satisfait bien aux objectifs définis dans la phase d'analyse, et éventuellement le corriger si ce n'est pas le cas.

Attention : Ne pas négliger les deux premières étapes – l'analyse et le pseudo-code –, au risque de voir la phase de debug – de loin la plus fastidieuse – s'éterniser...

[1] Le symbole « ␣ » représente simplement une espace, mais tous les autres caractères sont significatifs. Par ailleurs, les différences de polices (gras, italiques, etc.) ne sont que décoratives, puisque le source n'est qu'un simple fichier texte ne contenant aucune instruction spécifique de mise en page.

2 De la source à l'exécution

Même si les autres étapes ne sont pas à négliger[2], nous nous attardons maintenant sur la phase la plus technique de la programmation : le *codage*. Techniquement, la création d'un programme s'articule en 3 étapes :

Codage : À l'aide de l'éditeur de texte de votre choix (p.ex. `emacs`), il faut écrire un fichier texte contenant le programme en langage C. Par convention, le nom du fichier portera l'extension `.c`.

Compilation : Il faut alors traduire le source en un exécutable : c'est la *compilation*, réalisée par un programme spécifique, le *compilateur* (dans notre cas, `gcc`). Pour un fichier source `prog.c`, on utilise la commande suivante dans un terminal :

```
gcc prog.c
```

ce qui produit – s'il n'y a pas d'erreur dans le code – un exécutable nommé par défaut `a.out`. Pour imposer un nom plus explicite, p.ex., `prog.exe`, on peut utiliser l'option `-o` :

```
gcc -o prog.exe prog.c
```

Si la compilation échoue, le compilateur vous indique l'endroit dans votre fichier source où il ne comprend pas la syntaxe de votre code[3]. Il vous faut alors corriger les erreurs avant de relancer la compilation.

Exécution : Il s'agit maintenant d'exécuter votre programme dans le terminal, p.ex., à l'aide de la commande `./prog.exe`.

Si vous n'obtenez pas le résultat escompté ou si le programme s'interrompt à l'exécution avec un affichage du style `Segmentation fault`, alors les ennuis commencent : il faut debugger votre programme, c.-à-d. identifier et corriger les erreurs *conceptuelles* (et non pas simplement de syntaxe) présent dans votre programme.

Exercice : Recopier *très précisément* – y compris les espaces – le code C de la section précédente dans un fichier `hello.c`. Compiler et exécuter votre programme `hello.exe`. En êtes-vous satisfait ? Sinon, corriger (« debugguer ») votre programme.

3 Structure d'un programme C

3.1 Syntaxe des lignes de code

La syntaxe des lignes de code C est relativement simple :

- Chaque instruction se termine par le caractère « `;` ».
- Une instruction peut s'étendre sur plusieurs lignes, et une même ligne peut contenir plusieurs instructions.
- Hormis dans les chaînes de caractères, les espaces (et donc les retours à la ligne) n'ont pas de signification particulière : on peut donc en abuser pour réaliser une mise en page lisible.
- Le C fait la distinction entre les majuscules et les minuscules. Ainsi, `TOTO`, `ToTO` et `toto` ne représentent pas la même chose. En particulier, tous les mots-clé du C doivent être écrit en *minuscules*.
- Un ensemble d'instructions encadré par des accolades est syntaxiquement équivalent à une seule instruction.
- Toute chaîne de caractères comprise entre les symboles « `/*` » et « `*/` » est assimilée à un commentaire.

  Attention : on ne peut pas imbriquer plusieurs commentaires les uns dans les autres.

- Il existe un type spécial d'instruction, commençant par le symbole « # » : il s'agit en fait non pas d'instructions C, mais de directives du préprocesseur, qui agit juste *avant* la compilation. Comme ces directives ne sont pas des instructions C, elles peuvent être situées *n'importe où* dans le code, en général avant toute autre instruction. Ainsi, `#define MONPI 3.14` permet de définir dans la suite du fichier source une équivalence entre la chaîne MONPI[4] et la chaîne 3.14.

[2] L'analyse et le pseudo-code relèvent plutôt de l'analyse numérique et de l'algorithmique.

[3] Pour forcer le compilateur à vous avertir du moindre problème potentiel, ajouter l'option `-Wall` à la ligne de compilation : `gcc -Wall -o prog.exe prog.c`

[4] Il est de tradition d'écrire les symboles soumis à `#define` en majuscules.

Exemple 3 – Syntaxe des lignes de code

```
#define MONPI 3.14
x = MONPI;          /* MONPI sera remplacé par 3.14 */
y = 4; z = 5;       /* 2 instructions sur une ligne */
Sum = sin(x) +      /* Une instruction sur 3 lignes */
      cos(y) +
      tan(z);
if (Sum < 0) {
  Sum = 0;          /* Remise à zéro */
  printf("La somme était <0, "  /* Une chaîne sur 2 lignes */
         "maintenant =0\n");
}
```

3.2 Structure générale

Un programme C est considéré comme une fonction – systématiquement intitulée `main` – prenant en entrée les arguments du programme, et fournissant en sortie le « *status* » du programme (par convention la valeur 0 si tout c'est bien déroulé). De fait, notre premier programme (Ex. 2) commence par `int main()` (pas d'argument en entrée) et se conclut par `return(0);` (l'exécution a été réussie), et l'ensemble des instructions sont regroupées entre accolades.

Chaque fonction – y compris le `main` – se compose de deux parties : une partie *déclarative* (la « liste des ingrédients ») et une partie *exécutive* (la « recette »).

La partie déclarative. La déclaration des variables se fait selon les règles suivantes :

- Toute variable possède un type. Les types usuels sont les suivants :

  `int` nombre entier,

  `float` nombre réel, simple précision,

  `double` nombre réel, double précision,

  `char` caractère (lettres, chiffres, symboles, etc.).
- La déclaration d'une variable se fait en donnant son type suivi de son nom. Plusieurs variables de même type peuvent être déclarée en même temps.
- Les variables ne sont pas initialisées automatiquement : leur valeur immédiatement après leur declaration est *quelconque*.

> **Attention :** Ne pas oublier d'initialiser les variables si nécessaire, éventuellement en même temps que la déclaration.

Exemple 4 – Déclaration des variables

```
int i;              /* i de type entier */
float x, y;         /* x et y de type réel (simple) */
double sum = 0;     /* sum de type réel (double) initialisée à 0 */
```

C'est en général dans la partie déclarative (ou juste avant) que l'on peut/doit ajouter les commentaires précisant l'auteur, la date de création et la version de la fonction (ou du programme), ses objectifs et son mode d'emploi, ses limites, sa ligne de compilation, etc. (voir exemple ci-dessous).

La partie exécutive. Après la partie déclarative vient le *corps* du programme principal, constitué d'instructions. Encore une fois, il est de *très* bon ton de commenter un maximum son programme, afin d'expliciter son objectif, sa démarche, ses variables, etc.

La structure type d'un programme C (sans argument) est donc la suivante :

Exemple 5 – Structure type d'un programme C

```
/* Directives du préprocesseur */
#directive_1
#directive_2
...

int main()
{
  /* Commentaires sur le programme:
      Date de creation, auteur, version
      Objectif, mode d'emploi, ligne de compilation, etc.
   */

  /* Declaration des variables */
  declaration_1;
  declaration_2;
  ...

  /* Instructions */
  instruction_1;
  instruction_2;
  ...

  return(0); /* Exécution réussie */
}
```

3.3 Liens utiles

Le C est un langage très populaire[5], et les cours en ligne nombreux (cf. votre moteur de recherches favori). Outre le cours de C du DEA d'Astrophysique disponible en ligne (`http://snovae.in2p3.fr/ycopin/enseignements/docs/cours_C.pdf`), une liste de cours – majoritairement en français – est tenue à jour sur `http://c.developpez.com/cours/`.

4 Premiers programmes

Nous présentons ici deux petits programmes permettant de mettre en œuvre les points précédents. Les notions utilisées (p.ex. affichage, librairie, pointeurs, etc.) seront plus amplement étudiées dans les TD à venir.

Exercice : Écrire, compiler et exécuter les programmes suivants. Les assaisonner à votre goût.

4.1 Hypothénuse

Comme nous l'avons déjà vu, la commande permettant d'afficher un message à l'écran est `printf`, tandis que la commande permettant de lire des données du clavier est `scanf`. En outre, les fonctions mathématiques usuelles (telles que `hypot`) ne sont pas connues d'emblée du C : elles sont fournies par la librairie mathématique standard. Pour y accéder, il faut ajouter la directive `#include <math.h>` en tête du fichier (*avant* le `main`), et l'option « `-lm` » à la commande de compilation.

Exemple 6 – Hypothénuse

```
#include <math.h> /* Accès à la librairie mathématique */

int main()
{
  /* Affiche z = sqrt(x^2 + y^2), avec x et y entres au clavier */
  /* Compilation: gcc -o hypothenuse.exe hypothenuse.c -lm */
```

[5] Les systèmes UNIX sont notamment codés en C.

```
  float x,y,z;                    /* Déclaration de 3 réels */

  printf("Entrer 2 réels: ");     /* Affichage à l'écran */
  scanf("%f %f",&x,&y);           /* Entrée au clavier de 2 réels */
  z = hypot(x,y);                 /* hypot(x,y) = sqrt(x^2 + y^2) */
  printf("Hypothénuse = %f\n",z); /* Affichage formaté */

  return(0);
}
```

4.2 Sinus

Les instructions de gestion des fichiers (`fopen`, `fclose`, etc.) sont fournies par la librairie d'entrée/-sortie standard `stdio.h`. Outre la lecture de données au clavier, `scanf` retourne le nombre de variables *effectivement* lues.

Exemple 7 – Sinus

```
#include <math.h>  /* Librairie mathématique standard */
#include <stdio.h> /* Librairie entrée/sortie standard */
#define PI 3.14159265358979323844 /* Définition */
#define ENTREE "input.dat" /* Nom du fichier d'entrée */
#define SORTIE "sinus.dat" /* Nom du fichier de sortie */

int main()
{
  /* Stocke une période de sinusoïde dans le fichier SORTIE,
     la période et le nb de points étant donnés dans les 2 premières
     lignes du fichier ENTREE.

     Compilation: gcc -o sinus.exe sinus.c -lm
  */

  FILE *entree, *sortie; /* Pointeurs sur fichiers */
  int nvar,npts, i;
  float periode, x,y;

  /* Lecture du fichier de paramètres */
  printf("Lecture du fichier %s: ",ENTREE);
  entree = fopen(ENTREE,"r"); /* Ouverture du f. en lecture */

  /* Lecture de 2 variables */
  nvar = fscanf(entree, "%f\n%d", &periode,&npts);
  if (nvar != 2) { /* ERREUR !!! */
    printf("Erreur dans la lecture de %s\n",ENTREE);
    fclose(entree);
    return(1); /* Échec durant l'exécution */
  }
  else printf("P = %f, n = %d\n", periode,npts);

  fclose(entree); /* Fermeture du f. d'entrée (NE PAS OUBLIER!) */

  /* Fichier de sortie */
  printf("Création du fichier %s\n",SORTIE);
  sortie = fopen(SORTIE,"w"); /* Ouverture du f. en écriture */

  /* Boucle sur une période */
  for (i=0; i<npts; i++) { /* i = 0 ... npts-1 */
```

```
    x = i*periode/(npts-1);
    y = sin(x/periode * 2*PI);

    /* Écriture dans le fichier de sortie */
    fprintf(sortie, "%f \t%f \n", x,y);
  }

  fclose(sortie); /* Fermeture du f. de sortie (NE PAS OUBLIER!) */

  return(0); /* Exécution réussie */
}
```

Exercice : Que se passe-t'il si le fichier d'entrée n'existe pas ? Ajouter un test permettant de prendre en compte cette éventualité.

Les instructions de base en C

Module d'informatique - Licence DSM ENS Lyon

TD n° 3 — 2004-05

14 février 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Les instructions du C, déclarations, boucles, conditions, entrées-sorties.

1 Déclaration/affectation de variables

Types de variables. Les types de variables (entiers, réels et caractères) ont la syntaxe suivante : `int`, `float` ou `double`, `char`. Les variables d'un programme doivent être déclarées, qu'elles soient globales – c'est-à-dire communes à l'ensemble du programme – ou locales – propres à un bloc du programme. Lorsque l'on déclare plusieurs variables d'un même type, on sépare le nom de chaque variable par une virgule.

Le type `char` ne correspond qu'à *un seul* caractère, p.ex. `char lettre='A';`. Nous verrons au TD n° 4 que si l'on veut stocker une *chaîne* de charactères, il faut utiliser un *tableau* de `char`, p.ex. `char nom[]="Zorro";` ou `char nom[50];` pour déclarer une variable permettant de recueillir un chaîne de 50 caractères.

Conversion de type. Il est possible de forcer une conversion de type, p.ex. convertir un `int` en `float`. Cette conversion, ou « cast », s'opère avec la syntaxe `(type)variable`, p.ex. : `x = (float)i`. On verra dans la section suivante les conséquences d'une telle opération.

Affectation simple. Elle est réalisée par le signe '=', et peut être multiple.

```
x = 3;
y = x + 2;
a = b = 5;          /* Affectation multiple */
c = 3 + (i = 2); /* Affecte c=5 *et* i=2 */
```

Affectation composée. Elle a la forme générale `x <opérateur>= y`, ce qui équivaut à `x = x <opérateur> y`. P.ex. : `x -= a` est un raccourci de `x = x - a`. Pour les opérateurs possibles, cf. la section suivante.

Incrémentation/décrémentation : il existe une écriture compacte pour ces opérations très courantes :

`++x` Pré-incrémentation, `x` devient `x+1` *avant* d'être utilisé,

`x++` Post-incrémentation, `x` devient `x+1` *après* avoir été utilisé.

P.ex., si `j=2` initialement :

`i = ++j;` équivaut à `j=j+1; i=j;`. Résultat final : `i=3` et `j=3`,

`i = j++;` équivaut à `i=j; j=j+1;`. Résultat final : `i=2` et `j=3`.

Le décrémentation est identique après remplacement de `+` par `-`.

Attention : Si certaines notations du C ont l'avantage d'être très compactes, elles ont l'inconvénient d'être parfois obscures, et à l'origine de nombreuses erreurs. Ne pas hésitez alors à décomposer une jolie instruction composée (mais cryptique) en plusieurs instructions simples.

Variables de blocs. Si nécessaire, il est possible de définir des variables dans des blocs d'instructions délimités par une paire d'accolades. Elles n'existent alors *que* dans ce bloc, p.ex.

```
void main() {
  int i = 1;

  {                                /* début du bloc */
    int i = 3, j = 2;              /* Variables internes au bloc */
    printf("i du bloc: %d\n", i);
  }                                /* fin du bloc */

  printf("i de main(): %d\n", i);
}
```

2 Les opérateurs

Les opérateurs arithmétiques : '+', '-', '*', '/', '%' (opération « modulo »). Ces opérateurs agissent sur tous les types de variables, sauf `char`. L'opération modulo s'effectue sur les entiers uniquement.

Attention : La division entre entiers correspond à la division *euclidienne*.

Comparer à titre d'exemple le résultat des trois opérations suivantes, avec `int i = 10, j = 4, k = 6` et `float x` :

`x = i/j;` donne `x = 2.0`, car la division de deux entiers correspond à la division *euclidienne*, et le résultat entier est alors automatiquement converti en `float`.

`x = (float) i/j;` On a forcé la conversion de type `int` ⟶ `float`. Dans ce cas c'est la première variable rencontrée qui a été convertie, soit ici `i`. Un des membres de la division étant maintenant de type réel, la division s'effectue sur des réels, et donne `x = 2.5`.

`x = (float) (j/k);` Les parenthèses forcent la précédence (priorité) de la division euclidienne par rapport à la conversion de type. Résultat : **0**.

Il faut donc faire attention de forcer la division réelle entre entiers si nécessaire à l'aide d'une conversion de type, explicite (exemple précédent) ou implicite (p.ex. `x = 1.*i/j`).

Le C ne fournit pas d'opérateur puissance mais la librairie mathématique[1] dispose d'une fonction `pow(x,y)` qui retourne x^y.

Les opérateurs relationnels et logiques. Ils comparent la valeur de deux expressions arithmétiques :

Opérateurs	Effet
`< >`	test de supériorité (inf.) stricte
`=< >=`	test de supériorité (inf.) large
`== !=`	test d'égalité (inégalité)
`&&`	ET logique
`||`	OU logique
`!`	NON logique

Opérateur conditionnel. C'est le seul opérateur ternaire du C :
`(condition) ? (instruction1) : (instruction2)`
Il s'interprète de la manière suivante : l'`instruction1` est réalisée si la condition est vraie, sinon c'est l'`instruction2`. P.ex., pour retourner le max de deux nombres, `max = (a > b) ? a : b`. Il est alors courant de définir la fonction de préprocesseur suivante :

```
#define MIN(a,b) ((a < b)? a : b)
```

[1] La librairie mathématique est disponible en incluant `#include <math.h>` en tête du fichier source, et en ajoutant l'option `-lm` à la ligne de compilation.

Les opérateurs sur les adresses. Ces opérateurs sont utilisés pour manipuler des pointeurs (cf. TD n° 4) :
- opérateur *adresse* `&x` : retourne l'adresse où est stockée la variable,
- opérateur *indirection*, appliqué à un pointeur `*p` : retourne la valeur de l'objet pointé.

L'utilisation de ces objets sera explicitée ultérieurement.

3 Les instructions exécutables

Ces instructions permettent d'enchaîner les séquences de calcul. Parmi ces structures de commande, on distingue les *structures conditionnelles* (`if/else`, `switch/case`) qui permettent une sélection et les *structures répétitives* (boucles d'exécution `while`, `do/while`, `for`).

3.1 Structures conditionelles

Ces structures permettent d'effectuer des choix et peuvent s'appliquer à une instruction ou à un bloc.
La sélection `if/else` s'écrit de manière générale :

```
if (condition_a) {
  instruction_1;
  ...
  instruction_n;
}
else if (condition_b) {
  bloc_b;
}
else {
  bloc_c;
}
```

Dans l'exemple précédent les différents `bloc_...` peuvent contenir plusieurs instructions (il faut alors les délimiter par une paire d'accolades) ou se résumer à une seule instruction (accolades préférables pour des questions de sécurité et de clarté mais non obligatoires).
P.ex. :

```
if (i > 0) {
  printf("i positif");
  NbValPos ++;    /* on incrémente un compteur */
}
else if (i >= 2 && i <= 10)
  printf("Ce message ne sera JAMAIS affiché!");
```

La structure de commande `switch/case` sera vue ultérieurement.

3.2 Structures répétitives

On dispose de la boucle classique `for` et de boucles avec tests en début et fin de structure telles que `while` et `do/while`. On peut sortir arbitrairement et instantanément de ces boucles à l'aide de l'instruction `break`.

La boucle `for`. Elle est de la forme suivante :

```
for ([initialisation(s)]; [condition(s) de poursuite]; [incrémentation(s)]) {
      bloc_instructions;
}
```

P.ex. `for (i=0; i<10; i++) printf("i = %d",i);`. Exemple plus compliqué :

```
for (i=2, j=4; i<10, j>0; i++, j--) {
  printf("i = %d et j = %d", i, j);
}
```

L'instruction `continue` permet de passer arbitrairement et directement à l'itération suivante.

La boucle `while`. Elle permet d'exécuter une boucle tant qu'une condition d'arrêt n'est pas réalisée :

```
while (condition) {
  bloc_instructions;
}
```

La boucle do/while. Elle permet de reporter le test de la condition à la fin de la première itération (p.ex. lorsque c'est cette itération qui permet le calcul de la condition) :

```
do {
  bloc_instructions;
} while (condition);
```

Cette boucle est donc toujours exécutée au moins une fois.
Exercice : On réalise un test sur les erreurs cumulées lors d'un calcul numérique (p.ex. l'évaluation de $\tan(\arctan(\exp(\ln(\sqrt{a^2}))))/a - 1$).

```
#include <math.h>  /* librairie mathématique standard */
#include <stdio.h> /* librairie entrée-sortie standard */

int main() {
  int i=0;
  double a=1.,b,c=0.,cmax=0.0000001;

  do {
    b = tan(atan(exp(log(sqrt(pow(a,(double)2.))))))  / a - 1.;
    a = a + 1.;
    c = c + b;
    i ++;
  } while ( c < cmax);
  printf("Itération %d:   Erreur = %12.10lf\n",i,c);
  return (0);  /* Exécution réussie */
}
```

Combien d'itérations ont été effectuées pour un seuil limite de 10^{-10} ?

4 Les entrées-sorties

Les fonctions d'entrées-sorties sont définies dans le fichier `stdio.h` (à inclure au début du source). Les principales fonctions sont :

`printf/scanf` affichage à l'écran et lecture du clavier (entrée/sortie standard),

`fprintf/fscanf` écriture/lecture d'un fichier,

`sprintf/sscanf` écriture/lecture d'une chaîne de caractères,

4.1 Entrée/sortie standard

`printf(format, [arg1, arg2, ...])` P.ex. :

```
printf("%f + %f est égal à %f\n", a, b, a+b);
```

`scanf(format, [&arg1, &arg2, ...])` Cette fonction effectue une conversion entre la chaîne saisie au clavier (en ASCII) et la représentation interne de la variable. P.ex. :

```
#include <stdio.h>
...
int age; float taille; char nom[50];

scanf("%s, %d, %f", nom, &age, &taille); /* Lecture des 3 variables */
printf("%s est âgé de %d ans et mesure %.2f m\n", nom, age, taille);
```

La chaîne de contrôle permet d'inclure du texte et de définir le format des arguments à écrire ou à lire. Le nombre d'arguments est connu au moment de la compilation après décodage de la chaîne de contrôle en comptant les descripteurs de format commençant par '%' (cf. paragraphe suivant). Le caractère `\n` (retour à la ligne en C) permet de passer à la ligne après l'affichage de la chaîne.

Noter que les variables utilisées dans `scanf` doivent être précédées de '`&`', *sauf* pour les chaînes de caractères (tout cela deviendra limpide au prochain TD).

Attention : L'oubli du caractère '&' est une erreur fréquente (et mortelle) dans l'usage de `scanf()`.

Descripteurs de format. Les descripteurs de format les plus utiles sont :

Format	Signification
`%d`	conversion en décimal
`%c`	caractère simple
`%s`	chaîne de caractères
`%e`	réel sous la forme $\pm a.bbb$ `e` $\pm\ cc$
`%f`	réel en virgule flottante $\pm aaa.bbbb$
`%%`	le caractère '%'

Entre le % et le caractère de conversion peuvent en outre être introduits un formatage supplémentaire :

- un nombre indiquant la largeur du champ d'affichage (p.ex. `%8d` pour un entier sur 8 caractères),
- un chiffre indiquant le nombre de décimales d'un réel ; il faut séparer les deux nombres par un point (p.ex. `%12.5f` pour un réel sur 12 caractères, dont 5 après la virgule),
- la lettre `l` pour indiquer l'affichage d'un `double`.

4.2 Les fichiers

Ouverture et fermeture. Un fichier est traité en C comme un « flux » d'octets (type `FILE *`). L'ouverture d'un flux se fait par `fopen()` prenant en argument un nom de fichier `filename`, et un mode d'ouverture (lecture, écriture, etc.), qui peut être :

`"r"` ouverture en lecture seule,

`"r+"` mise à jour d'un fichier existant,

`"w"` ouverture en écriture seule,

`"w+"` création en lecture et écriture,

`"a"` ajout en fin de fichier (s'il existe) ou création de fichier (s'il n'existe pas),

`"b"` si le fichier est binaire.

Exemple :

```
#include <stdio.h>
...
FILE *fichier;

fichier = fopen("fichier.dat", "w+"); /* ouverture du fichier */
/* Utilisation du fichier */
fclose(fichier)                       /* fermeture du fichier */
```

Ne pas oublier de fermer le fichier par la commande `fclose()`. On se reportera à la solution du TD n° 2 pour un exemple complet de gestion de fichiers d'entrées-sorties (en particulier pour le test d'existence d'un fichier que l'on tente d'ouvrir).

Lecture et écriture. On peut lire et écrire sur fichier précédemment ouvert à l'aide des fonctions `fscanf()` et `fprintf()`. La syntaxe est proche de celle des fonctions `scanf()` et `printf`, il suffit simplement de spécifier en premier argument le nom du fichier sur lequel s'effectuent les opérations d'entrées-sorties :

`fprintf(fichier, format, [arg1, arg2, ...])` P.ex. :

```
fprintf(fichier, "Le résultat de %f + %f est %f\n", a, b, a+b);
```

`fscanf(fichier, format, [&arg1, &arg2, ...])` P.ex. :

```
fscanf(fichier, "%s, %d, %f", nom, &age, &taille);
```

5 Application

Exercice : On se propose d'illustrer les problèmes de précision numérique en calculant les premiers termes de la suite de récurrence suivante :

$$\begin{aligned} a_0 &= 11/2 \\ a_1 &= 61/11 \\ a_{n+1} &= 111 - 1130/a_n + 3000/a_n\, a_{n-1} \end{aligned}$$

Cette suite converge vers la valeur exacte 6. Écrire un programme qui calcule au moins les 30 premiers termes de la suite en utilisant des `(double)` et des `(float)`. On souhaite que le résultat des différentes itérations soit écrit dans un fichier `output.dat`.

Deux problèmes se posent dans ce calcul. Le premier résulte des erreurs d'arrondi successives, qui est aggravé avec des variables de type `float`. Le second résulte de « l'instabilité » de la récurrence elle-même. On constate que même en `double` la suite converge vers 100 ! Augmenter la précision n'apporte que peu d'améliorations, il faudrait réécrire la relation de récurrence pour la rendre numériquement plus stable.

C : Tableaux & pointeurs

Module d'informatique - Licence DSM ENS Lyon

TD n° 4 — 2004-05

25 février 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Tableaux et pointeurs en C. *Application* : le carré magique

Rappels : Types de variable

- Toute variable possède un type. Les types les plus courants sont les suivants :

  `int` nombre entier,

  `float` nombre réel, simple précision,

  `double` nombre réel, double précision,

  `char` caractère (lettres, chiffres, symboles, etc.).
- Les variables ne sont pas initialisées automatiquement : leur valeur immédiatement après leur declaration est *quelconque*. Il faut donc bien penser à les initialiser si nécessaire.
- Il n'existe pas de type booléen à proprement parler, mais on peut utiliser le fait que toute variable nulle est logiquement équivalente à « *faux* », tandis que toute variable non nulle est « *vraie* ».

1 Les tableaux

On a souvent besoin de regrouper dans une seule variable — et donc un seul nom — plusieurs éléments d'un même type, p.ex., les coordonnées d'un point, les éléments d'une matrice, etc. On utilise pour cela les *tableaux*[1].

1.1 Déclaration

La déclaration d'un tableau est très proche de celle d'une variable simple, puisqu'il suffit d'ajouter sa `[dimension]` à la suite de la déclaration. P.ex. :

Exemple 1 – Déclaration de tableaux numériques

```
int index[10];
float matrice[10][10];
```

Dans ces exemples,

- La variable `index` est un tableau mono-dimensionnel comprenant 10 entiers d'indices variant *de 0 à 9* ;

  Attention : Les tableaux du C sont indexés à partir de 0 (et donc jusqu'à n-1).

- `matrice` est un tableau bi-dimensionnel de 10×10 réels, les indices variant de $[0][0]$ à $[9][9]$.

Noter que les indices des tableaux sont *naturellement* des entiers.

[1] Les *structures* permettent quant à elles de regrouper dans une même variable plusieurs éléments éventuellement de types différents.

1.2 Allocation

Dans les déclarations précédentes, la dimension du tableau doit *impérativement* être connue au moment de la compilation : on ne peut donc pas écrire `int n,index[n];` pour ajuster la taille du tableau selon des besoins déterminés au cours de l'exécution[2]. La taille doit par conséquent être prévue dès la conception suffisamment grande pour les domaines d'application du programme (qui sont donc toujours à expliciter et à vérifier).

Pour plus de flexibilité dans la maintenance du programme, on utilise généralement une instruction `#define` pour définir globalement la taille des tableaux :

Exemple 2 – Taille fixe d'un tableau

```
#define NMAX 5          /* Facile à changer si nécessaire */

  [...]

  int dtab;
  int itab[NMAX];       /* indice de 0 à NMAX-1 */

  printf("Dimension utile du tableau? (>0,<=%d): ",NMAX);
  scanf("%d",&dtab); /* Entrée au clavier d'un entier */
  if (dtab > NMAX) {
    printf("ERREUR");
    return(1);          /* Échec du programme */
  }
```

1.3 Manipulation

On accède à un élément d'un tableau à l'aide de son indice :

Exemple 3 – Accès aux tableaux

```
  x = itab[3]; /* x reçoit la valeur du *4ème* élément de itab */

  for (i=0; i<dtab; i++)
    itab[i] = i; /* itab = 0,1,...,dtab-1 */
```

L'écriture d'un tableau se fait en général à l'aide d'une ou plusieurs boucles – autant que la dimensionalité du tableau en question. Pour les petits tableaux, l'initialisation peut éventuellement se faire au moment de la déclaration, p.ex.

Exemple 4 – Initialisation de tableaux

```
  int mois[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  int identite[3][3] = {{1, 0, 0},
                        {0, 1, 0},
                        {0, 0, 1}}
```

Attention : Les opérations *symboliques* sur les tableaux – du type `vecteur1 + vecteur2` – sont impossibles en C : toutes les opérations doivent se faire au niveau des *éléments*.

1.4 Les chaînes de caractères

Il faut faire la distinction entre un *tableau* de caractères et une *chaîne* de caractères. Le tableau n'est effectivement qu'une collection de caractères, tandis que la chaîne contient en outre un caractère final obligatoire – le caractère *null* '\0' – indiquant en mémoire la fin de la chaîne.

[2] A contrario, l'*allocation dynamique de mémoire* permet d'ajuster à l'exécution la taille des tableaux en fonction des besoins du programme. Elle sera vue dans un TD ultérieur.

Exemple 5 – Tableaux et chaînes de caractères

```
char voyelles[6] = {'a','e','i','o','u','y'}; /* 6 car. */
char nom[6] = "Zorro";  /* 'Z'+'o'+'r'+'r'+'o'+'\0' = 6 car. */
```

On aurait également pu déclarer `char nom[] = "Zorro";`, le compilateur ayant le bon goût de compter les caractères pour vous.

Attention : Le caractère de fin de chaîne doit être compté dans la taille d'une chaîne de caractère.

Outre l'initialisation à la déclaration, on définit en général une chaîne de caractères à l'aide de la fonction `sprintf` (écriture formattée dans une chaîne).

2 Les pointeurs

2.1 Notions de base

Le pointeur est à la valeur ce que le contenant est au contenu.

Une variable « de base », de type **`int`**, **`double`**, etc., est destinée à être affectée d'une valeur, p.ex. 42 ou 'z'. En pratique, cette variable est stockée dans une case mémoire, que l'on peut repérer par son adresse. Ces addresses sont appelée des *pointeurs*. Puisque les cases mémoire doivent être de tailles différentes pour contenir des variables de types différents (p.ex. un **`char`** est stocké sur un octet tandis qu'un **`float`** en occupe 4), il existe plusieurs types de pointeur : les pointeurs sur les **`char`**, sur les **`int`**, etc.

Les pointeurs ont une importance capitale dans la programmation C, puisqu'ils permettent entre autre de passer le paramètre d'une fonction par *adresse* (ce qui permet en retour de le modifier) et non pas uniquement par valeur.

2.2 Notations

Les pointeurs sur un type de base **`xxx`** sont déclarés par le type pointeur **`xxx *`**. P.ex. **`float *`** représente les pointeurs pointant sur des **`float`**.

Soit une variable de base `x`, p.ex. de type **`double`**. L'adresse de cette variable s'obtient en écrivant `&x`, et est donc de type **`double *`**. À l'inverse, soit maintenant un pointeur `p`, p.ex. déclaré de type **`int *`** : la valeur pointée, c-à-d le contenu de la case mémoire dont l'adresse est `p`, s'écrit `*p` et est de type **`int`**[3].

Ainsi, `x` et `*(&x)` représentent la même chose, à savoir la valeur de `x`. De même, `p` et `&(*p)` sont équivalents. En revanche, `*x` *ne veut rien dire* puisque `x` n'est pas de type pointeur[4].

2.3 Relation avec les tableaux

Un tableau (mono-dimensionnel) est en fait un pointeur sur la première case du tableau. P.ex., en déclarant **`float`** `tab[10]`, le compilateur prévoit 10 cases mémoires permettant de contenir des réels simples, et `tab` est un pointeur **`float *`** pointant sur la première valeur du tableau. Par conséquent, `tab[0]` et `*tab` représentent *exactement* la même chose : la valeur du premier élément du tableau. En outre, `tab + 1` (la somme d'un pointeur et d'un entier) désigne l'adresse de la *deuxième* case du tableau, c-à-d `&(tab[1])`. On comprend ainsi pourquoi les tableaux du C sont indicés à partir de 0.

Selon la même logique, un tableau bi-dimensionel (p.ex. **`float`** `mat[3][3]`) est en fait un pointeur de pointeur (ici **`float **`**), pointant sur un tableau 1D de pointeurs, chacun d'entre eux pointant sur la première case des lignes du tableau 2D.

3 Le carré magique

Un carré magique d'ordre n est un tableau carré $n \times n$ dans lequel on écrit une et une seule fois les nombres entiers de 1 à n^2, de sorte que la somme des n nombres de chaque ligne, colonne ou diagonale

[3] Notez la cohérence de la notation : en déclarant `p` comme un pointeur sur un entier par **`int`** `*p;`, la valeur `*p` est bien effectivement de type **`int`**.

[4] Mais `&p` est un pointeur sur un pointeur sur un **`int`**, noté **`int **`**.

soit constante. P.ex. le carré magique d'ordre 5, où toutes les sommes sont égales à 65 :

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Pour les carrés magiques d'ordre impair, on dispose de l'algorithme suivant ((i, j) désignant naturellement la case de la ligne i, colonne j ; on se place en outre dans une indexation « naturelle » commençant à 1) :

1. la case $(n, (n + 1)/2)$ contient 1 ;
2. si la case (i, j) contient la valeur k, alors on place la valeur $k + 1$ dans la case $(i + 1, j + 1)$ si cette case est vide, ou dans la case $(i - 1, j)$ sinon. On respecte la règle selon laquelle un indice supérieur à n est ramené a 1.

Exercice : Programmer cet algorithme (après avoir détaillé les étapes d'analyse et de pseudo-code) pour pouvoir construire et afficher un carré magique d'ordre impair quelconque (<`NMAX`).

Utilisation des fonctions en C

Module d'informatique - Licence DSM ENS Lyon

TD n° 5 — 2004-05

9 mars 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Définition et utilisation des fonctions en C. Application à la méthode de Newton et aux générateurs aléatoires.

1 Généralités

Les fonctions sont les briques de base d'un programme. Elles permettent de construire des programmes structurés et logiques dans leur découpage. Le C ne distingue pas entre « procédures » (qui produisent un *effet de bord*) et « fonctions » (qui retourne en outre un résultat) comme d'autres langages de programmation (p.ex. Pascal et Fortran) : tout est de type *fonction*. Ces fonctions retournent un objet de type simple (un des type de base, p.ex. `int` ou `double`, ou un pointeur, p.ex. `float *`), ou éventuellement rien. Les fonctions admettent généralement des paramètres (mais éventuellement aucun), eux aussi de type simple.

Un programme C contient au moins une fonction dont le nom est `main()`. Les autres fonctions peuvent être appelées par `main()` ou par toute autre fonction. Un TD ultérieur indiquera comment réaliser des « collections » autonomes de fonctions, appellées *librairies*.

2 Éléments de syntaxe

Définition. La syntaxe de définition est la suivante :

```
type nom_fonction([type parametre_1, type parametre_2, ...]) {
  corps de la fonction;
  ...
  [return(valeur)];
}
```

où `type` est un type de base (p.ex. `int`, `float *`, `char *`, etc.) décrivant le type de la valeur de retour de la fonction, ou le type particulier `void` (« vide »), qui permet de ne pas spécifier de type de retour et de fait de ne retourner aucune valeur (à la façon d'une procédure).

Les paramètres de la fonction sont eux-aussi strictement typés, et leur liste (éventuellement vide) doit être *exhaustive* et invariable[1].

La valeur de retour d'une fonction est celle de l'instruction `return` permettant de sortir de la fonction.

Prototypage. Le *prototypage* d'une fonction permet au compilateur d'anticiper l'existence d'une fonction (qui pourrait être définie dans un fichier séparé) et de contrôler la cohérence entre sa déclaration et son utilisation durant la phase de compilation et d'édition de liens.

Pour ce faire, il est préférable de préciser, avant son utilisation, le type de la fonction et de ses paramètres éventuels. Les paramètres étant muets, il n'est pas nécessaire d'indiquer leurs noms (mais il est préconisé de les laisser), p.ex.

```
float get_ratio(float, float);          /* fn à 2 arguments réels */
int get_params(float *p1, float *p2);   /* fn à 2 pointeurs sur des réels */
```

En particulier les fichiers d'en-tête (extension `.h`) inclus (à l'aide de l'instruction `#include`) au début du fichier contiennent entre autre les prototypes de fonctions utilisables par le programme (p.ex. `printf()` dans `stdio.h`, `exp()` dans `math.h`, etc.). Ces déclarations permettent d'utiliser en toute sécurité des fonctions qui ne sont pas explicitement définies dans le corps du programme (p.ex. dans un autre fichier, ou dans une librairie externe).

[1] Il existe une façon en C de définir des fonctions avec un nombre *variable* d'arguments (p.ex. `printf`), mais nous ne l'aborderons pas.

Utilisation. L'appel à une fonction se fait par l'opérateur `()`, contenant une liste d'expressions correspondant aux valeurs d'appel des différents paramètres formels de la fonction :

```
nom_fonction(paramètres);              /* façon procédure */
variable = nom_fonction(paramètres);   /* façon fonction */
```

Dans ce dernier cas, `variable` prend la valeur retournée par la fonction.

Exemple 1 – Exemple d'utilisation d'une fonction.

```
#include <stdio.h>

float func_half(float x) {                /* la fonction */
  float y;

  y = x/2.;
  return(y);
}

int main()                                /* le programme principal */
{
  float val=3.,z;
  float func_half(float);                 /* prototypage */

  z = func_half(val);                     /* appel de la fonction */
  printf("La moitié de %f est %f\n",val,z);
  return(0);
}
```

3 Passage des paramètres

3.1 Paramètres scalaires

En C, le passage des paramètres s'effectue soit *par valeur* – on passe le contenu de la variable – soit *par adresse* – on passe l'adresse où est stockée la variable. Dans quel cas utiliser l'une ou l'autre méthode ?

Passage par valeur : Si la valeur de la variable n'a pas à être modifiée dans la fonction, la variable *peut* être passée par valeur.

Exemple 2 – Exemple de passage par valeur.

```
double carre(double x) {
  return(x*x);            /* Retourne le carré de la valeur */
}

int main()                /* Programme principal */
{
  double z=3, z2;
  doudle carre(double); /* Prototypage */

  z2 = carre(z);         /* La valeur de z est inchangée */
  return(0);
}
```

Passage par adresse : Si la valeur de la variable doit être modifiée dans la fonction, la variable *doit* être passée par adresse.

Exemple 3 – Exemple de passage par adresse.

```
void incremente(int *pi) {
  (*pi)++;                        /* Ajoute 1 à la valeur pointée par pi */
}
```

```c
int main()                          /* Programme principal */
{
  int i=3;
  void incremente(int *);           /* Prototypage */

  incremente(&i);                   /* La valeur de i est modifiée */
  return(0);
}
```

Remarque : le type `void` est un type particulier aux fonctions et signifie que la fonction ne retourne aucune valeur.

Exemple 4 – Illustration des deux cas précédents.

```c
#include <stdio.h>
int f(int x) {      /* fonction f: passage par valeur */
  x *= 2;           /* l'affectation a une portée locale */
  return(x);
}

int g(int *px) {    /* fonction g: passage par adresse */
  *px *= 2;         /* modification de la valeur pointée par px */
  return(*px);
}

int main()
{
  int x = 10, z;

  z = f(x);
  printf("x=%f, z=%f\n",x,z);   /* x=10, z=20 */
  z = g(&x);
  printf("x=%f, z=%f\n",x,z);   /* x=20, z=20: x est modifié! */

  return(0);
}
```

3.2 Paramètres tableaux

Dans le cas d'un tableau à une dimension, on peut utiliser le fait qu'un tableau est en fait un pointeur sur la première case du tableau. On peut donc passer comme argument à une fonction soit le tableau, p.ex. `int f(int tab[])`, soit le pointeur, p.ex. `int f(int *tab)`. La première notation a l'avantage de bien montrer la nature « tableau » du pointeur.

Dans le cas d'un tableau à plusieurs dimensions, seule la première dimension peut être omise ou remplacée par un pointeur, p.ex. `int f(int tab[3][4][5])` ou `int f(int *tab[4][5])`.

3.3 Pointeurs sur des fonctions

Les pointeurs de fonctions permettent d'appeler une fonction en spécifiant une valeur plutôt que le nom de la fonction. La déclaration est : `<type> (*nom)()`.

Dans ce cas, `<type>` représente l'objet retourné par la fonction dont `nom` est une variable pointeur ayant pour valeur l'adresse d'une fonction. Remarque :

- `int (*fonc)()` est un pointeur sur une fonction qui retourne un entier.
- `int *fonc()` est une fonction retournant un pointeur sur un entier.

Il est à noter que l'appel d'une fonction pointée nécessite les parenthèses :

Exemple 5 – Utilisation d'une fonction pointée.

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

float sinus(float x) {
  return (sin(x));
}
```

```
float derive(float (*f)(float), float x, float eps) {
  return( ((*f)(x+eps)-(*f)(x-eps))/(2*eps) );
}

int main()
{
  float yprime;
  float sinus(float x); /* fonction */

  yprime = derive(sinus, 3, 1.e-6);

  return(0);
}
```

4 Exercices

4.1 Recherche d'un zéro d'une fonction par la méthode de Newton

La méthode de Newton permet de trouver un zéro d'une fonction. Elle est basée sur le développement d'une fonction autour d'une de ses racines. Formellement si l'on développe $f(x_0) = 0$ autour de x_0 on trouve :

$$f(x_0) \simeq f(x) + (x_0 - x)f'(x) + \ldots = 0,$$

où x est une valeur d'essai pour x_0. Au pas $n+1$ la valeur approximative de x s'extrait de l'équation :

$$f(x_{n+1}) \simeq f(x_n) + (x_{n+1} - x_n)f'(x_n) \simeq 0.$$

On recherche donc la convergence de la suite :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

On se propose d'étudier les zéros de la fonction $f(x) = \mathrm{e}^x \ln x - x^2$.

1. Écrire un programme comportant la fonction f et qui écrive dans un fichier $donnees.dat$ les valeurs de la fonction sur l'intervalle $]0, 4]$.
2. Visualiser le contenu du fichier $donnees.dat$ à l'aide du logiciel `xmgr`.[2]
3. Écrire un programme pour trouver un zéro de la fonction f à l'aide de la méthode de Newton. Il faudra utiliser une `function` retournant la position du zéro.
4. Visualiser la convergence du programme tous les deux pas de calcul avec `xmgr`.

4.2 Générateur de nombres aléatoires

Un générateur de nombre aléatoire est une fonction qui renvoie à chaque appel un nombre quelconque pris dans un intervalle donné $[0, m[$. Réaliser un générateur réellement "aléatoire" n'est en fait pas simple du tout. Dans ce TD, nous utiliserons le générateur suivant :

$$I_{n+1} = (aI_n) \bmod m$$

avec $a = 7^5 = 16807$ et $m = 2^{31} - 1 = 2147483647$. I_0 s'appelle la graine (« seed ») de la série de nombres aléatoires générée par cette récurrence.

1. Écrire un programme, contenant une fonction `RANDGEN`, qui génère une centaine de nombres aléatoires sur l'intervalle $[0, 1[$ (on prendra $I_0 = 13762$ et on utilisera les valeurs de a et m données ci-dessus). Sauvegarder la liste de nombres aléatoires dans un fichier.
2. Visualiser ces nombres à l'aide du logiciel `xmgr`.
3. Faire des essais avec des valeurs de m de 10, 100, 1000, 10000 et 100000. Qu'en pensez-vous ?

[2] Lancer la commande `xmgr` dans un terminal. Un fichier texte contenant deux colonnes (p.ex. x et y) peut être lu (menu `File`→`Read`→`Sets`) sous format XY et tracé sur une fenêtre graphique. Ne pas oublier de sélectionner l'option `Autoscale On Read` qui vous permet de gérer automatiquement les échelles en x et y.

Méthodes d'intégration numérique

Module d'informatique - Licence DSM ENS Lyon

TD n° 6 — 2004-05

13 mars 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : pointeur sur des fonctions, algorithmes d'intégration numérique

Comme dit le proverbe, « *Dérive qui veut, intègre qui peut* ». Le calcul *numérique* d'intégrales joue donc un rôle majeur dans de nombreux problèmes physiques.

Il s'agit ici d'écrire une série de procédures permettant de calculer numériquement l'intégrale $\int_a^b f(x)\,\mathrm{d}x$, et de comparer leurs performances (vitesse de convergence pour une précision requise). Durant ce TD, on pourra plus particulièrement tester le cas analytique $\int_0^\pi \sin x\,\mathrm{d}x = 2$, mais les procédures doivent pouvoir s'appliquer à toute fonction f définie (et intégrable) sur le domaine choisi. Les techniques d'intégration sont généralement mieux adaptées à un certain type de fonction, il est donc toujours conseillé de vérifier la pertinence d'une procédure sur une fonction aussi proche que possible de celle que vous utiliserez réellement dans votre application physique.

Pour une discussion plus détaillée, voir p.ex. *Numerical Recipes* (http://www.nr.com/).

1 Pointeur sur les fonctions

Afin de passer des fonctions en paramètre d'une procédure (p.ex. une procédure d'intégration), on dispose des *pointeurs sur les fonctions*. Ainsi, pour annoncer que la fonction `f` est une fonction prenant un ***float*** en entrée et délivrant un ***double***, on déclarera le pointeur suivant : ***double*** `(*f)(`***float***`)`. Il est impératif de déclarer le type de retour de la fonction, mais il est en revanche facultatif de déclarer le type de ses arguments : on aurait pu écrire ***double*** `(*f)()`. Mais de manière générale, si on connait le type des paramètres de façon certaine, il est préférable de les déclarer afin que le compilateur puissent faire des vérifications de conformité de type.

P.ex., soit la fonction `deriv` permettant de dériver numériquement une fonction retournant un ***double*** :

Exemple 1 – Utilisation des pointeurs sur les fonctions

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

double deriv(double (*f)(), float x, double eps)
{
  return( (f(x+eps)-f(x-eps))/2/eps );
}

int main()
{
  double (*fn)();

  fn = sin;                                   /* fn = sin */
  printf("sin'(0) = %f\n",deriv(fn,0,1e-6));  /* fn'(0)   */
  printf("cos'(0) = %f\n",deriv(cos,0,1e-6)); /* cos'(0)  */

  return(0);
}
```

On note que l'on n'a pas déclaré le type de la variable de la fonction à dériver, par conséquent, le compilateur ne peut pas vérifier la conformité de type (en particulier, il ne peut pas s'assurer que la fonction doit être à *un seul* paramètre).

2 Méthode de Simpson

Pour le calcul d'une quadrature (aire sous une courbe), vous connaissez sans doute la *méthode des rectangles* et la *méthode des trapèzes*. Dans la première méthode, chaque arc de courbe est approximé localement par une constante, tandis que la seconde méthode utilise une approximation linéaire. Ces méthodes calculent la fonction à intégrer en des abscisses régulièrement espacées, et font partie des méthodes de *Newton-Cotes*.

La *méthode de Simpson* quant à elle approche ce même arc de courbe par une parabole (approximation quadratique)[1]. On peut alors montrer qu'une valeur approchée de $\int_a^b f(x)dx$ peut se calculer à partir de $2n+1$ estimations de f :

$$\int_{x_0}^{x_{2n}} f(x)\,\mathrm{d}x \simeq \frac{h}{3}[f_0 + 4\times(f_1+f_3+\cdots+f_{2n-1}) + 2\times(f_2+f_4+\cdots+f_{2n-2}) + f_{2n}],$$

avec $h=(b-a)/2n$, $x_0=a$, $x_{2n}=b$ et $f_i=f(x_i)$.

Exercice : Coder la procédure `int_simpson` mettant en œuvre cette méthode d'intégration. Analyser la vitesse de convergence (c-à-d. l'évolution de la précision ϵ en fonction du nombre d'appels de la fonction f).

3 Méthode de Monte-Carlo

On a

$$\int_a^b f(x)dx \simeq \frac{(b-a)}{n}\sum_{i=1}^{n} f(x_i),$$

où les n points x_i sont tirés au hasard dans l'intervalle d'intégration $[a,b]$.

Exercice : Coder la procédure `int_monte_carlo` mettant en œuvre cette méthode d'intégration, en utilisant le générateur de nombres aléatoires vu au TD précédent. Comparer graphiquement la vitesse de convergence avec celle de la méthode précédente. Conclusion ?

4 Méthode de Romberg

Il existe une abondante littérature présentant des algorithmes d'intégration plus performants que les méthodes précédentes. Tout en restant dans les formules dites « de Newton-Cotes », l'algorithme *de Romberg* permet d'obtenir de meilleurs résultats (voir p.ex. `http://www.mat.ulaval.ca/anum/ch5/html/node12.html`) :

Algorithme de Romberg :

- Initialisation : $i=j=0$, $h_0=b-a$, $T_{0,0}=\frac{2}{h_0}\left[f(b)+f(a)\right]$,
- Boucle : tant que $|T_{i,i}-T_{i-1,i-1}|>\epsilon$:
  - $i=i+1$
  - $h_i=\frac{h_{i-1}}{2}$
  - $T_{i,0}=\frac{1}{2}\left[T_{i-1,0}+h_{i-1}\sum_{j=1}^{2^{i-1}} f\left(a+(2j-1)\frac{h_{i-1}}{2}\right)\right]$
  - Pour j variant de 0 à i : $T_{i,j}=\frac{4^j T_{i,j-1}-T_{i-1,j-1}}{4^j-1}$
- Conclusion : $\int_a^b f(x)\,\mathrm{d}x \simeq T_{i,i}$

Exercice : Coder la procédure `int_romberg` mettant en œuvre cette méthode d'intégration. Comparer la vitesse de convergence avec celle des méthodes précédentes.

[1] On peut évidemment utiliser des approximations de degré toujours plus élevé.

Équations différentielles, méthodes de Runge-Kutta

Module d'informatique - Licence DSM ENS Lyon

TD n° 7 — 2004-05

20 mars 2005, y.copin@ipnl.in2p3.fr

1 Équations différentielles ordinaires du 1er ordre

On se propose de résoudre l'équation différentielle du 1er ordre :

$$y' = f(x,y) \quad \text{pour} \quad x \in [a,b] \quad \text{avec} \quad y(a) = y_0 \tag{1}$$

Les schémas de Runge-Kutta assurent, en général, une meilleure stabilité de la solution que la méthode d'Euler. Il existe des schémas à 2, 3 ou 4 points, avec des termes d'erreurs en $O(h^3)$, $O(h^4)$ et $O(h^5)$ respectivement.

L'un des deux schémas de Runge-Kutta[1], d'ordre 2, s'écrit :

$$\begin{aligned} y_{n+1} &= y_n + (k_1 + k_2)/2 + O(h^3); \\ k_1 &= h f(x_n, y_n) \\ k_2 &= h f(x_n + h, y_n + k_1) \end{aligned} \tag{2}$$

Exercice : Écrire un programme contenant les modules suivants :

- Le programme lui-même, qui doit se résumer à un appel de fonctions ou de procédures et éventuellement l'affichage des résultats.
- La fonction $f(x,y)$. On prendra à titre d'exemple la fonction suivante :

$$y' = f(x,y) = -2xy^2, \quad x \in [0,5] \quad \text{avec} \quad y(0) = 1 \tag{3}$$

- La procédure de lecture au clavier des bornes d'intégration a et b, du nombre de points d'intégration N et de la valeur y_0 : $y(a) = y_0$.
- Le module qui implémente le schéma de Runge-Kutta proposé. Vous pouvez alors étudier la précision de Runge-Kutta en comparant avec la solution analytique : $y(x) = 1/(1 + x * x)$.

2 Équations différentielles ordinaires du 2ème ordre

On s'intéresse maintenant au mouvement d'un pendule de longueur l auquel est attachée une masse m et soumis à une force d'excitation périodique $f(t) = f_0 \cos(\omega t)$. L'angle que fait le pendule avec la verticale est solution de l'EDO :

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \sin\theta = b\cos\omega_0 t$$

où $q = k/m$ et $b = f_0/ml$. L'équation est du type : $y'' = g(y, y', t)$. On se propose d'utiliser deux méthodes différentes pour la résoudre.

2.1 Application du schéma de Runge-Kutta du 1er ordre

Dans cette méthode, on réécrit l'EDO du deuxième ordre sous la forme :

$$\begin{aligned} \frac{dy_1}{dt} &= y_2 = g_1(y_1, y_2, t) \\ \frac{dy_2}{dt} &= -q\,y_2 \;-\; \sin y_1 \;+\; b\,\cos\omega t = g_2(y_1, y_2, t) \end{aligned}$$

avec $y_1 = \theta$ et $y_2 = d\theta/dt$.

[1] Vous pouvez trouver d'autres formules dans : *Handbook of Mathematical Functions*, M. Abramowitz and I. Stegun, Dover. Ce livre est *une mine* pour les fonctions spéciales.

On applique ensuite un schéma de Runge-Kutta à chacune de ces équations du 1$^{\text{er}}$ ordre couplées. Pour une équation du type $y' = g(y,t)$, ces schémas peuvent être à deux ou quatre points :

– schéma à deux points :

$$\begin{aligned} y_{n+1} &= y_n + (k_1 + k_2)/2 \\ k_1 &= h * g(y_n, t) \\ k_2 &= h * g(y_n + k_1, t + h) \end{aligned}$$

– schéma à quatre points :

$$\begin{aligned} y_{n+1} &= y_n + (k_1 + 2 * (k_2 + k_3) + k_4)/6 \\ k_1 &= h * g(y_n, t) \\ k_2 &= h * g(y_n + \frac{k_1}{2}, t + \frac{h}{2}) \\ k_3 &= h * g(y_n + \frac{k_2}{2}, t + \frac{h}{2}) \\ k_4 &= h * g(y_n + k_3, t + h) \end{aligned}$$

Exercice : Écrire un programme qui résout l'EDO du second ordre pour $\theta \in [-\pi, \pi]$ avec les valeurs des paramètres suivantes :

$$\begin{aligned} (q, b, \omega_0) &= (0.5, 0.9, 2/3) \\ (q, b, \omega_0) &= (0.5, 1.15, 2/3) \end{aligned}$$

On prendra comme conditions initiales : $\theta_0 = 0$ et $\theta'_0 = 2$. Comparer la précision de chacun des schémas de Runge-Kutta à deux et quatre points.

2.2 Application d'un schéma de Runge-Kutta du deuxième ordre

Comme pour le 1er ordre il existe une série de méthodes de Runge-Kutta. Nous appliquerons la méthode d'ordre 4 à la résolution d'un oscillateur, par exemple :

$$y_{n+1} = y_n + h\left(y'_n + \frac{1}{6}(k_1 + k_2 + k_3)\right) + O(h^5) \tag{4}$$

$$\begin{aligned} y'_{n+1} &= y'_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad (5) \\ k_1 &= hf(y_n, y'_n, t) \\ k_2 &= hf(y_n + \frac{h}{2}y'_n + \frac{h}{8}k_1, y'_n + \frac{k_1}{2}, t + \frac{h}{2}) \\ k_3 &= hf(y_n + \frac{h}{2}y'_n + \frac{h}{8}k_1, y'_n + \frac{k_2}{2}, t + \frac{h}{2}) \\ k_4 &= hf(y_n + hy'_n + \frac{h}{2}k_3, y'_n + k_3, t + h) \end{aligned}$$

Exercice : Écrire un programme qui résolve l'EDO du pendule forcé avec les mêmes jeux de paramètres que précédemment.

Les librairies en C

Module d'informatique - Licence DSM ENS Lyon

TD n° 8 — 2004-05

25 mars 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Compilation et édition de liens, réalisation et utilisation de librairies

1 Introduction

L'objectif de *modularité* est primordial pour une programmation efficace. En effet, comme dans de nombreux domaines, un certain nombre d'opérations de base sont communes à de très nombreux calculs, et il serait particulièrement vain de les redéfinir à chaque nouvelle utilisation. Il est donc important :

- de bien identifier ces fonctions élémentaires,
- de les isoler au mieux de leur environnement,
- de les optimiser et éventuellement d'accroître leur potentiel en généralisant leurs domaines d'application.

Ainsi, comme dans un jeu de construction, il suffira alors de les assembler, ou même simplement de les enrober, pour créer rapidement des programmes plus complexes.

En ce sens, le C fournit la notion de fonction, pour lesquelles il est important :

- de circonscrire l'action : chaque routine doit avoir un unique objectif bien défini, et déporter tout calcul complexe à une sous-routine (et ainsi de suite) ;
- de définir précisément son mode d'emploi, en particulier en précisant clairement l'interface (c-à-d la nature et le rôle des différents paramètres ou variables globales), les conditions d'utilisation (p.ex. tailles maximale des tableaux), les codes d'erreur, etc.

Cette démarche doit se faire non seulement au niveau du code lui-même (p.ex. par l'utilisation de noms de variable explicites), mais également au niveau de la documentation détaillée des procédures.

Une fois ce travail de normalisation et documentation réalisé, il est intéressant de regrouper l'ensemble de ces fonctions de base au sein d'une *bibliothèque* sur laquelle se basera les développements futurs. Ainsi, il existe de par le WEB de nombreuses bibliothèques C couvrant de multiples domaines (p.ex. calculs numériques, entrées-sorties, graphiques, etc.) dont l'utilisation permet non seulement d'accélérer le développement de son projet, mais aussi de contribuer à la structuration de son code.

2 Compilation et édition de liens

Le terme générique de « *compilation* », utilisé jusqu'à présent pour évoquer la production d'un programme exécutable à partir d'un fichier source, recouvre en fait deux actions distinctes :

La compilation à proprement parler traduit un fichier *source* (p.ex. d'extension `.c`), crée à l'aide d'un éditeur de texte, en un fichier *objet* (extension `.o`) : il s'agit simplement de la traduction en langage machine de la suite d'instructions du fichier source. Chaque donnée ou fonction définie dans un fichier objet possède un *nom symbolique*, et chaque référence à un symbole extérieur au fichier objet considéré est appelée *référence externe*.

> **Attention :** Les fichiers objets d'extension `.o` sont des fichiers binaires mais ne sont pas exécutables.

L'édition de liens (*to link*) tente de faire le lien entre les symboles définis ou requis dans les différents fichiers objets `.o` (ou archives `.a`) pour enfin produire le fichier exécutable. Si l'éditeur de liens ne peut pas lier une référence externe, il affiche un message d'erreur et ne peut pas produire l'exécutable.

Exercice : Les différents tests de compilation se feront sur les fichiers `integration.c` (contenant le programme principal), `proc_integration.c` (contenant les procédures d'intégration), `fonction.c` (décrivant les fonctions à intégrer) et `integration.h` (prototypant les fonctions et procédures précédentes) dérivés de la correction du TD n° 6 (cf. http://snovae.in2p3.fr/ycopin/enseignements/info_ENS.html).

Dans les cas simples – p.ex. production d'un programme à partir d'un ou plusieurs fichiers sources –, ces deux actions sont regroupées de façon transparente dans une seule commande. Dans notre cas :

```
gcc -Wall -o integration.exe \
    integration.c proc_integration.c fonction.c -lm
```

produit l'exécutable `integration.exe` à partir des 3 fichiers `integration.c`, `proc_integration.c` et `fonction.c`, comme si ces 3 fichiers avaient été concaténés en un seul (avec l'avantage cependant de la modularité).

Cependant, dans les cas plus complexes[1], les deux actions doivent être séparées.

2.1 Compilation

La production d'un fichier objet `.o` à partir d'un fichier source `.c` se fait dans notre cas à l'aide de l'option `-c` du compilateur `gcc`. Ainsi, les commandes

```
gcc -Wall -c integration.c
gcc -Wall -c proc_integration.c
gcc -Wall -c fonction.c
```

créent les fichiers `integration.o`, `proc_integration.o` et `fonction.o`. Si nécessaire, on peut consulter les symboles définis dans un fichier objet à l'aide de la commande `nm`, p.ex. `nm proc_integration.o`.

2.2 Édition de liens

L'éditeur de liens utilise la table des symboles des différents fichiers objets pour assortir les références externes (c-à-d les symboles utilisés mais non *définis* dans un fichier objet) aux définitions globales (les symboles définis dans un fichier objet). Sous UNIX, l'édition de liens se fait à l'aide de la commande `ld`, et ce *quelque soit le langage de programmation*. Cependant, on n'utilise rarement cette commande directement, puisque la plupart des compilateurs offre une interface plus simple d'emploi. Dans notre cas, il s'agit simplement de la commande `gcc`. P.ex.

```
gcc -o integration.exe \
       integration.o proc_integration.o fonction.o -lm
```

génère le programme exécutable `integration.exe` à partir des fichiers objets `integration.o`, `proc_integration` et `fonction.o` (et de la librairie mathématique).

3 Les librairies

Les *librairies* – ou *bibliothèques* – ne sont qu'un type particulier de fichier objet, regroupant dans un même fichier des fonctions de base très souvent utilisées. Tout langage de programmation dispose de ses propres librairies, et il est souvent intéressant de se constituer des librairies personnelles regroupant des routines autour d'une thématique précise. C'est également ce que l'on peut trouver sur le réseau Internet, offrant de multiples fonctionnalités dans de nombreux domaines (p.ex. calculs numériques, gestion des entrées-sorties, etc.).

> **Attention :** Une librairie doit être parfaitement *structurée* et *documentée* pour pouvoir être utile à terme (c-à-d au delà de l'utilisation immédiate).

Pour ce faire, il est préférable de séparer les différentes routines d'une librairie en autant de fichiers – contenant chacun une *unique* fonctionnalité (p.ex. « recherche de zéro », ou « quadrature ») et ses dépendances –, et de normaliser l'entête de ce fichier pour indiquer clairement les informations utiles.

[1] Même si ici, cela ne s'avère évidemment pas indispensable...

3.1 Création

Par convention, le nom d'une librairie (statique, par opposition aux librairies partagées que l'on n'abordera pas ici) est `lib`*`nomdelalibrairie`*`.a`. Une librairie se créé à partir d'une collection de fichiers `.o` à l'aide de la commande `ar -r`. Ainsi, dans notre cas simple :

```
ar -r libinteg.a proc_integration.o fonction.o
```

créé une librairie `libinteg.a` contenant les fichiers `proc_integration.o` et `fonction.o`.

> **Attention :** En règle générale, on inclut dans une archive des fichiers objets ne contenant *que* des fonctions ou sous-routines, mais pas de programmes principaux.

3.2 Prototypage

Il est de *très bon goût* d'associer à chaque librairie un fichier `.h` prototypant toutes les fonctions définies dans la librairie. Il suffit alors d'inclure ce fichier dans tous les programmes principaux faisant référence à cette librairie pour que toutes les fonctions soient automatiquement définies (c'est p.ex. le rôle de `math.h` vis-à-vis de la librarie mathématique).

Si le fichier de prototypage n'est pas stocké dans le répertoire courant, il faut indiquer le répertoire dans lequel le trouver au moment de la compilation (et non pas uniquement au moment de l'édition de liens) à l'aide de l'option `-I`*`répertoire`*.

3.3 Utilisation

Une librairie n'intervient bien entendu qu'au niveau de l'édition de lien. Elle peut alors être utilisée comme n'importe quel fichier objet, p.ex.

```
gcc -o integration.exe integration.o libinteg.a -lm
```

ou même plus directement

```
gcc -o integration.exe integration.c libinteg.a -lm
```

cette dernière commande incluant la compilation « à la volée » de `integration.c`.

Lorsque la librairie est stockée dans un répertoire différent du répertoire courant (p.ex. dans le répertoire de compilation de la bibliothèque, ou dans un répertoire dédié à toutes vos bibliothèques), on peut utiliser les options de compilation `-L`*`répertoire`* `-l`*`nomlib`*, pour une librairie `lib`*`nomlib`*`.a` stockée dans le répertoire *`répertoire`*.

Exercice : Créer une librairie personnelle (p.ex. `libperso.a` associée à `perso.h`) contenant les différentes routines développées dans les TD précédents, et reformuler les programmes « principaux » pour utiliser cette librairie.

4 Librairies externes

Nous vous fournissons deux librairies externes en C que vous pouvez maintenant utiliser pour le développement de vos programmes :

- la librairie pédagogique de *Numerical Recipes*,
- la *GNU Scientific Library* (GSL).

4.1 Numerical Recipes

La NUMREC (`www.nr.com`) est une librairie « pédagogique » regroupant les différentes routines introduites et discutées dans le célèbre livre *Numerical Recipes*, disponible en ligne à `http://www.library.cornell.edu/nr/bookcpdf.html`. Il ne s'agit donc pas réellement d'une librairie à usage professionnel (elle est même réputée peu fiable), mais plutôt d'un bon point de départ.

La librairie NUMREC est disponible sous `/home/ycopin/local/lib`, et se nomme `libnrc.a`. Le fichier de prototypage associé est `nr.h` sous `/home/ycopin/local/include/`. Par conséquent, il suffit pour pouvoir utiliser la NUMREC :

- d'inclure dans vos fichiers sources :
  ```
  #include <nr.h>
  ```
- d'ajouter les options suivantes à la commande de compilation :

```
-L/home/ycopin/local/lib/ -lnrc \
-I/home/ycopin/local/include/
```

> **Attention :** Les sources de la NUMREC ne sont pas gratuites ! Vous ne devez donc pas distribuer la librairie que l'on met à votre disposition.

Exercice : Utiliser la routine `mppi` de la NUMREC (§ 20.6) pour afficher un nombre arbitraire de décimales de π.

4.2 GSL

La *GNU Scientific Library* (`http://www.gnu.org/software/gsl/`) est une librairie numérique d'utilisation aisée contenant plus de 1000 routines mathématiques. Elle est amplement documentée sur `http://sources.redhat.com/gsl/ref/gsl-ref_toc.html`.

Cette librairie est mise à disposition par `/soft/eleves/`. Pour l'utiliser, il faut ajouter à la ligne de compilation :

```
-L/soft/eleves/arch/linux-i386/lib -lgsl -lgslcblas \
-I/soft/eleves/include
```

en ayant pris soin d'inclure le ou les fichiers de prototypage lié à la ou les fonctions vous concernant.

Exercice : Trouver dans la librairie GSL la routine permettant de calculer les racines d'un polynôme à coefficients réels, et l'intégrer dans un programme.

Allocation dynamique de mémoire

Module d'informatique - Licence DSM ENS Lyon

TD n° 9 — 2004-05

25 mars 2005, y.copin@ipnl.in2p3.fr

Objectifs du TD : Allocation dynamique de mémoire : `malloc`, `calloc`, `free`, débuggueur

1 Manipulation de tableaux

Les tableaux permettent de regouper plusieurs éléments d'un même type (ex. `int`, `float`...) sous un même nom. Il se déclare classiquement en indiquant le nombre d'éléments (i.e. la dimension du tableau) – qui doit être connu *au moment de la compilation* – associés à un nom de tableau. Ainsi `float tab[100]` va créer un tableau de 100 réels de type `float`.

Comme nous l'avons déjà vu, dans la déclaration précédente, `tab` est en fait un *pointeur* sur un `float` (`float *`), pointant sur la première case du tableau. Ainsi, `*tab` (valeur pointée par `tab`) et `tab[0]` (premier élément du tableau `tab`) représente *exactement* la même chose, ainsi que `*(tab+1)` (valeur pointée par le pointeur suivant `tab`) et `tab[1]`, etc.

La déclaration classique du tableau `float tab[100]` a donc un double rôle :

- le compilateur déclare une variable `tab` de type `float *`,
- connaissant la place mémoire nécessaire au stockage d'un `float` (en l'occurence `sizeof(float)` octets), il réserve une zone mémoire contiguë permettant de stocker 100 réels de type `float`, et stocke l'adresse de la première case dans `tab`.

Cette manière de procéder constitue une **allocation statique** de la mémoire pour les tableaux. Puisque l'opération d'allocation de mémoire se fait au même moment que la déclaration, la taille de la mémoire à réserver doit *impérativement* être connue *a priori*, c.-à-d. au moment de la compilation. Cela conduit à sur-dimensionner les tableaux lorsque l'on ne peut pas savoir d'avance la dimension de ces derniers (p.ex. lorsque la taille d'un tableau dépend d'un calcul effectué dans une étape précédente).

Pour contourner cette difficulté, le C offre la possibilité de recourir à une **allocation dynamique** de la mémoire où *la réservation s'effectue pendant l'éxécution, au fur et à mesure des besoins*. Cette fonctionnalité est extrêmement puissante, mais constitue la source majeure de bugs particulièrement difficiles à corriger.

2 Allocation dynamique

2.1 Fonctions d'allocation

L'allocation dynamique va scinder la déclaration d'un tableau en deux parties distinctes :

1. déclaration initiale d'un pointeur « nu » `float *tab;` non initialisé,
2. allocation, dans le corps du programme, d'une zone mémoire d'une taille suffisante pour les besoins courants, et stockage de l'adresse de la première case de la zone allouée dans `tab`.

Dorénavant, `tab` pointe sur la zone mémoire allouée, et permet de la gérer comme un tableau, p.ex. `tab[0] = 1`.

Le C dispose de deux fonctions d'allocation dynamique, prototypées dans le fichier `malloc.h` (`#include <malloc.h>`) :

- `malloc()` (pour `m`emory `alloc`ation) permet simplement d'allouer une zone mémoire *sans l'initialiser*,
- `calloc()` fait de même, mais initialise toutes les cases allouées à 0.

Attention :
– Ne surtout pas tenter d'accéder à une zone mémoire non allouée.
– Ne pas oublier d'initialiser un tableau alloué par `malloc`.
– `malloc` et `calloc` ont des syntaxes *subtilement différentes.*

2.2 Taille des différents types de données

Le(s) paramètre(s) de ces fonctions d'allocation de mémoire sont la taille de la zone à allouer *en octets*. Par exemple,

```
#include <malloc.h>

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

permet de réserver 100 octets en mémoire et renvoit l'adresse du premier. L'unité de stockage pour un caractère (`char`) est toujours l'octet.

En revanche, la taille nécessaire pour stocker un `int` ou un `float` peut dépendre de la machine. Pour déterminer la taille d'une variable de type quelconque, le C dispose donc de la fonction `sizeof()`. Par exemple, `sizeof(int)` retourne la taille nécessaire (en octets) pour stocker *un* `int` (en général 4, mais cela peut dépendre du système). On écrit donc de manière générale :

```
#include <malloc.h>

/* Declaration d'un tableau de float */
float *tab;

/* Allocation d'un tableau permettant de stocker 100 floats */
tab = (float *)malloc(100*sizeof(float));
```

Remarque sur les opérations « pointeur + entier » : un incrément sur un pointeur admet pour unité la taille (en octets) de son élément de base (par exemple 4 pour un entier dans la plupart des systèmes). Dès lors, l'opération :

```
float *tab1, *tab2;
tab2 = tab1 + 1;
```

permet de pointer `tab2` vers l'élément d'un tableau suivant celui pointé par `tab1`, quelque soit la taille mémoire en octet occupée par un `float` : on n'a pas à se préoccuper du calcul du décalage en octets, mais uniquement en case logique !

2.3 Allocation et initialisation

La fonction `calloc()` permet d'initialiser automatiquement à zéro tous les octets alloués. On a donc :

```
tab = (float *)calloc(100,sizeof(float));       /* noter la virgule !!!*/
```

qui équivaut à :

```
tab = (float *)malloc(100*sizeof(float));
for (i=0; i<100; i++) tab[i]=0.;
```

2.4 Tableaux à plusieurs entrées

Un tableau à plusieurs entrées peut être vu comme un tableau de pointeurs. Par exemple pour déclarer un tableau d'entiers `n` $\times$ `m`, on peut procéder de *trois* façons distinctes :

1. Allocation statique : les tailles `n` et `m` du tableau sont connues à l'avance.
2. Allocation semi-statique : la taille `n` du tableau est connue à l'avance. La longueur `m` des lignes n'étant pas connue à l'avance, il faudra les allouer dynamiquement.
3. Allocation dynamique : les tailles `n` et `m` du tableau ne sont pas connues à l'avance. Il faudra donc allouer dynamiquement une colonne de pointeurs de taille `n`, puis faire pointer chacun vers une zone de taille `m` allouée dynamiquement.

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

int main() {
  int n=2, m=3;
  int i,j;

  int tab1[n][m];          /* Tableau permettant le stockage de n x m entiers. */
                           /* INCONVENIENT: n et m doivent etre connu a priori. */

  int *tab2[n];            /* Tableau permettant le stockage de n pointeurs sur
                              des entiers : l'allocation de chacune des n lignes
                              du tableau final est a la charge de l'utilisateur.
                           */
                           /* INCONVENIENT: n doit etre connu a priori. */

  int **tab3;              /* Pointeur sur un pointeur sur un entier : permet de
                              pointer un tableau du type precedent, sans que la
                              taille n soit connue. */

  /* Allocation dynamique d'un tableau permettant le stockage de pointeurs sur
     des entiers, et donc de type int **. La taille (ici n) peut etre le
     resultat d'un calcul. */
  tab3 = (int **)malloc(n*sizeof(int *));

  /* Allocations dynamiques de tableaux permettant le stockage d'entiers, et
     donc de type int *. La taille (ici m) peut etre le resultat d'un calcul,
     et n'est pas necessairement constante (p.ex. pour une matrice
     triangulaire). */
  for (i=0; i<n; i++) {
    tab2[i] = (int *)malloc(m*sizeof(int));
    tab3[i] = (int *)malloc(m*sizeof(int));
  }

  /* Initialisation des tableaux: les trois tableaux s'accedent de la meme
     maniere, seule leurs allocations different (les plus souples etant les
     plus complexes). */
  for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
      tab1[i][j] = i+j;
      tab2[i][j] = i+j;
      tab3[i][j] = i+j;
    }

  for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
      printf("%d,%d = %d %d %d\n",
             i,j,tab1[i][j],tab2[i][j],tab3[i][j]);
    }

  /* Pour la liberation de la memoire, il faut effectuer les operations a
     l'envers. Seuls les tableaux alloues dynamiquement doivent etre
     explicitement liberes. */

  /* Liberation des lignes allouees dynamiquement. */
  for (i=0; i<n; i++) {
    free(tab2[i]);
    free(tab3[i]);
```

```
  }
60  /* Liberation de la colonne allouee dynamiquement. */
  free(tab3);

  return 0;
}
```

2.5 Libération de la mémoire

Afin d'optimiser l'utilisation de la mémoire pendant l'éxécution du programme, il est indispensable de pouvoir libérer la mémoire allouée dynamiquement lorsqu'elle n'est plus utilisée. Pour cela on dispose de la commande `free()` qui admet comme paramètre n'importe quel pointeur pointant vers une zone allouée dynamiquement.

> **Attention :**
> – Seules les zones mémoire allouées dynamiquement doivent être libérées par `free`,
> – Ne surtout pas tenter d'accéder à une zone mémoire précédemment libérée.

3 Et maintenant, les problèmes commencent...

3.1 Erreurs classiques

Les erreurs liées à la gestion de la mémoire sont à l'origine du célèbrissime `Segmentation fault`, qui indique que l'on essait d'accéder à une zone mémoire que ne nous appartient pas[1].

Les grands classiques :

– un indice est en train de se ballader dans un domaine de variation plus grand que celui qui a été initiallement prévu, p.ex. :
```
int tab[5];
for(i=0; i<10; i++) tab[i]=...;
```
– utiliser un tableau déclaré comme pointeur mais non encore alloué (oubli de `malloc()` ou `calloc()`) :
```
int *tab;
for(i=0; i<10; i++) tab[i]=...;
```
– utiliser un tableau alloué dynamiquement, mais libéré entre temps :
```
int *tab;

tab = (int *)malloc(10*sizeof(int));
...
free(tab);

for(i=0; i<10; i++) tab[i]=...;
```

3.2 Le débuggueur

Le **débuggeur** est un outil précieux permettant de suivre le déroulement du programme *pas à pas* et d'identifier par exemple les points d'arrêt (endroit où ça plante), les violations de mémoire, les calculs intermédiaires, etc.

L'utilisation de cet outil sort du cadre de ce module d'info, mais une utilisation typique est la suivante :

– il faut compiler votre programme avec l'option `-g`
– `gdb mon_programme` pour lancer le débuggueur,
– `run mes arguments si nécessaire` pour lancer le programme,
– plantage ! (sinon, vous ne débuggueriez pas...)
– `backtrace` pour essayer d'identifier la cause du plantage.

Vous pouvez trouver de la documentation en ligne (p.ex. « `gdb tutorial` » dans Google).

[1] Tout le problème vient du fait qu'une erreur de gestion de mémoire *peut ou peut ne pas* se traduire par un `SegFault`...