Méthodes Numériques

Université Paul Sabatier, Toulouse III. M1.

Année 2016-2017

Organisation

- 6 séances par groupe de niveau
  - 2 séances de prise en main
  - 4 séances de TD-cours sur les méthodes numériques
  - Contrôle continu 1h (50%)
- 6 séances de projets
  - 1-2 séances de cours spécialisé
  - 4-5 séances de projet
  - rapport de projet (50%)
- Présence obligatoire !

I. UNIX et son shell

UNIX

- UNIX = famille de systèmes d'exploitation
  - multi-tâches
  - multi-utilisateurs
- Organisation en couches distinctes
  - le noyau
  - le shell
  - le système graphique
  - les applications et programmes

Le shell

- shell = interpréteur de commandes
- accès via un terminal ou console
- plusieurs langages (bash, tcsh, csh, ksh…)
- l'utilisateur tape le nom des commandes

```
>whoami
```

```
>command
>command arg1 arg2 arg3
>command -option1 -option2
>command -option arg
>command arg -option
```

- Page de manuel : `>man command`

Arborescence de fichiers

- Noms de fichiers avec extension (.txt, .c, .f, .exe…)
- Position dans l'arborescence (path) = chaîne de caractères avec des /

```
rep1/rep2/rep3/fichier.txt
```

- Répertoires particuliers :
  - répertoire courant : `./`
  - répertoire en amont : `../`
  - répertoire racine (root) : `/`
  - répertoire de l'utilisateur (home) : `~/`

Opérations sur l'arborescence

```
>pwd
```

- Affiche le répertoire courant *(print working directory)*

```
>ls
>ls rep/
>ls -a -l -t
```

- Liste les fichiers et répertoires *(list)*

```
>cd rep/
```

- Déplacement dans l'arborescence *(change directory)*

```
>cd
```

```
>cd ./
```

```
>cd ~/
```

```
>cd ../
```

- ➠ *Applications:*
  - *Remonter d'un répertoire et revenir*
  - *Aller à la racine et revenir dans votre home*

Manipulation de fichiers

`>mkdir rep`	Crée un répertoire *(make directory)*
`>rmdir rep`	Efface un repertoire vide *(remove directory)*
`>touch fich`	Crée un fichier vide *(touch)*
`>rm fich`	Efface un fichier *(remove)* • **Tout fichier effacé est perdu !!!!!!!**
`>cp fich nv_fich`	Crée une copie d'un fichier *(copy)*
`>mv fich rep/`	Déplace un fichier *(move)*

Manipulation de fichiers

- ➠ *Applications :*
  - *Créez un répertoire “M1/“ dans le home “etudiant”*
  - *Créez un repertoire avec votre nom de binôme et allez-y*
  - *Créez un fichier vide toto.txt*
  - *Créez une copie appelée titi.txt*
  - *Effacez ces deux fichiers*

Astuces indispensables

```
>clear
```

- Efface le contenu du terminal

```
>↑
```

- Rappelle des commandes

```
>debut_de_command➡
>command debut_de_fichier➡
```

- Auto-complétion (tabulation)

```
>ls *
>ls *fin
>ls debut*
>ls debut*fin
```

- Le symbol * remplace toute chaîne de caractères et signifie *"n'importe quel ensemble de caractères"*

```
>programme
^C
```

- Tue un programme en cours

```
>programme &
```

- Exécute un programme en tache de fond

Astuces indispensables

- ➠ *Applications*
  - *créer 2 fichiers toto1.txt et toto2.txt*
  - *créer 2 fichiers titi1.txt et titi2.txt*
  - *créer 2 fichiers tata1.txt et tata2.txt*
  - *afficher tous les fichiers qui commencent par toto*
  - *effacer tous les fichiers qui contiennent le caractère “2”*
  - *effacer **UN PAR UN** tous les fichiers du repertoire de binôme*

II. Langage C

Le C : un langage non-interprété

- Langages interprétés
  - 2 étapes distinctes:
    - écriture
    - transcription en langage machine et execution simultanément
  - ex: Matlab, IDL…
- Langage non-interprétés:
  - 3 étapes distinctes:
    - écriture
    - conversion en langage machine
    - exécution
  - ex: C, fortran…

Écriture

- Dans un fichier texte
- Extension du fichier : .c
- Avec un éditeur quelconque (kwrite, gedit…)
- Exemple :

```
#include<stdio.h>

main(){printf("Bonjour\n");}
```

- ➠*Application : créer en enregistrer un fichier prog.c avec ces lignes*

Compilation

- La conversion en langage machine est exécutée par un **compilateur**.
- Elle produit un fichier exécutable
- Notre compilateur :
  - **gcc** (GNU C Compiler)
  - aide depuis un terminal :

```
>man gcc
```

```
>gcc --help
```

```
>gcc --help -v
```

Compilation

- La compilation:
  - `>gcc prog.c` génère un fichier executable “a.out”
  - `>gcc prog.c -o prog.x` génère un fichier executable “prog.x”

- Détecte de potentielles erreurs
  - warnings : erreur suspectées, mais la compilation génère un executable
  - errors : erreurs avérées, aucun executable généré

- ➠ *Application :*
  - *compiler le programme prog.c*
  - *l’éditer et y ajouter une erreur. Compiler*
  - *corriger l’erreur. Compiler.*

Exécution

- Le fichier généré par le compilation est un programme exécutable

- Il s'execute par : `>./prog.x`

- Tout modification du fichier texte doit être suivie d'une compilation !!

- ➠ Application :
  - exécuter le programme créé.

Structure d'un programme

```
#include<stdio.h>
main(){printf("Bonjour\n");}
```

```
#include<lib1.h>
#include<lib2.h>

main(){

   instruction1;
   instruction2; instruction3;

   // commentaire
   instruction4;

   /* début du commentaire…
   … suite du commentaire */
   instruction5;

}
```

Appel à des bibliothèques de fonctions

Nom du programme = main

Ensemble d'instructions entre { }

Les instructions finissent par ;

Les espaces multiples, retours à la ligne et tabulations ne sont pas interprétés

Commentaires après //
ou entre /* et */

Minuscules et majuscules sont traitées différemment (x≠X)

Affectations de variables

- Avec le signe =
- Se finissent par ;
- les éléments des tableaux sont repérés par leur indice entre []

```
i=2;

x=6.4; y=i;

tableau[3] = 3.2 ;

matrice[2][4] = 12 ;
```

- ➠ *Application :*
  - *Essayer d'écrire un programme qui attribue la valeur 2 à une variable x. Que dit le compilateur ?*

Déclarations

- Avant utilisation, toute variable doit être déclarée
  - À la déclaration, le compilateur réserve de l'espace mémoire pour la variable
  - Le type (entier, réel…) doit être précisé
  - Se finit par un ;
  - Syntaxe : `type variable;`
- Exemples :

```
long int n;
int i, j, k=5;
double x=5.4, y, z;
```

```
double tab[34];
int mat[2][5];
char mot[10], mot2="hello";
```

- Les types :
  - entiers : `short int` `long int` `long long int` `int`
  - réels : `float` `double` `long double`
  - caractères : `char`
  - vide : `void`
- ➠ Modifier le programme précédent pour qu'il marche

Opérations numériques

- Se finissent par ;
- Opérations classiques : `z=x+y; z=x-y; z=x*y; z=x/y;`
- Le résultat est du même type que les variables `?=i/j;`
- Reste d'une division entre entiers : `k=i%j;`
- bibliothèque mathématique :
  - `#include<math.h>`
  - `exp(x); sin(x); cos(x); fabs(x); sqrt(x); pow(x,n);`
  - `M_PI`
- Abréviations : `i++; i--; i+=3; i*=2;`

Entrées-sorties à l'écran

- Appel à des fonctions de la bibliothèque : `#include<stdio.h>`

- Affichage dans le terminal : `printf(format,var1, var2, ...);`
- Lecture depuis le terminal : `scanf(format,&var1, &var2, ...);`

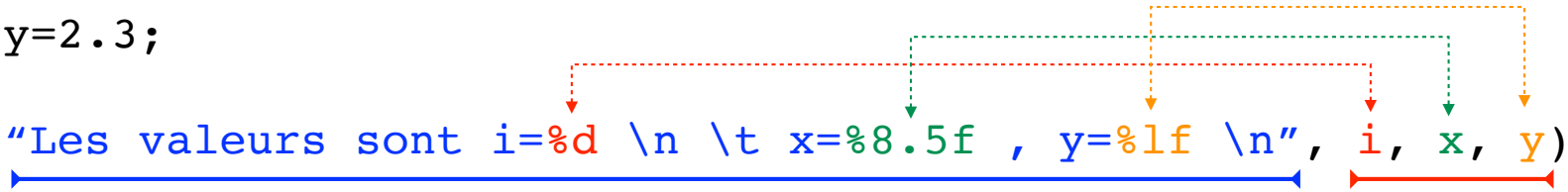
- Les variables :
  - L'ensemble des variables à lire ou écrire
  - lors d'une lecture, on passe en argument l'adresse des variables avec le symbol & !!

Entrées-sorties à l'écran

- Le format est une chaîne de caractère.
- Par exemple :
  - Le programme

```
int i=2;
float x=1.2;
double y=2.3;

printf("Les valeurs sont i=%d \n \t x=%8.5f , y=%lf \n", i, x, y);
```



  - donne le résultat suivant :

```
les valeurs sont i=2
   x=1.200000 , y=2.3
```

- Le format peut contenir du texte et des caractères spéciaux
  - retour à la ligne `\n` , tabulation `\t` , etc…)
- Il doit contenir autant de %fmt que de variables

Entrées-sorties à l'écran

- Chaque %fmt **doit** correspondre au type de la variable correspondante :
  - entiers :
    - simples `%d, %5d, %05d`
    - longs `%ld, %5ld, %05ld`

pour i=7:

```
7,     7, 00007
```

  - réels :
    - simples `%f %e, %6.2f`
    - longs `%lf, %le`

pour x=-π:

```
-3.141593, -3.141593e+00, _-3.14
```

  - chaîne de caractères : `%s, %9s`

Entrées-sorties à l'écran

- ➠ *Application :*
  - *Lire au clavier les valeurs de 2 entiers i et j.*
  - *Afficher leur somme, différence, produit, rapport*

- ➠ *Pour aller plus loin :*
  - *stocker la valeur 1/3 dans 2 variables x (réel simple) et y (réel double). Les afficher avec 20 chiffres significatifs*
  - *Définir une variable entière i=1 et une variable réelle x=1./3. Essayer d'afficher i avec un format réel et x avec un format entier… Essayer l'inverse…*
  - *Déclarer une variable réelle et la lire au clavier avec un format inadapté (entier par ex.). L'afficher…*

Entrées-sorties dans des fichiers

- Déclaration d'une variable pointeur de type fichier :

```
FILE *fich;
```

- Ouverture du fichier :

```
fich = fopen("mon_fichier.txt",action);
```

  - action = "r" (read)
  - action = "w" (write)
  - s'il existe, le fichier est effacé

- Écriture et lecture :

```
fprintf(fich, format, var1, var2, …);
```

  - reprend au point de dernière lecture/écriture

```
fscanf(fich, format, &var1, &var2, …);
```

- Fermeture du fichier :

```
fclose(fich);
```

- ➠ *Application:*
  - *Modifier le programme précédent pour écrire les 4 résultats dans un fichier "result.txt"*

Conditions

- Syntaxe :

```
if(condition){instruction si vrai;}
```

```
if(condition)
    {instructions si vrai;}
else
    {instructions si faux;}
```

- Tests :

==	!=	>	>=	<	<=	&&	||
egal	diff.	stric. sup	sup ou eg.	stric. inf	inf. ou eg.	et	ou

- Exemple :

```
if(n%2==0)
    {printf("n est pair \n");}
else
    {printf("n est impair \n");}
```

- ➠ *Application:*
  - *Écrire un programme qui calcule et affiche les racines réelles de ax^2+bx+c lorsqu'elles existent.*

Boucles

- But : effectuer un grand nombre de fois un ensemble d'opérations
- Syntaxe :
  - Boucle for :

```
for(i=0; i<n; i=i+1){instructions;}
```

    - i et n doivent être déclarés
    - 3 infos : min, max, et pas
  - Boucle while :

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

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

- Exemple :

```
int i,n=10;
for(i=0; i<n; i=i+1)
    {printf("l'indice de boucle est i=%d\n",i);}
```

- ➠*Applications*
  - *Écrire un programme qui remplit un tableau de taille n avec des valeurs lues au clavier et qui affiche ensuite ces valeurs en ordre inverse*
  - *Écrire un programme qui calcule la factorielle d'un nombre n*

Pointeurs

- Pointeur = variable qui contient une adresse mémoire
- Déclaration avec une * :

```
type *ptr;
```

- Affectation :
  - depuis un pointeur :

```
ptr2 = ptr1;
```

  - depuis une variable :

```
ptr = &var;
```

- Opérations :
  - pas d'opérations sur les adresses
  - opérations sur les valeurs adressées :

```
*ptr = 2.3;
*ptr = x;
x = *ptr*2
```

Fonctions

- Utilité : effectuer un ensemble d'opérations et retourner un résultat

- But : faciliter écriture, débuggage et augmenter la lisibilité d'un code

- 2 types de fonctions:
  - retournent une valeur
  - ne retournent pas de valeur

Fonctions : syntaxe

- Fonctions qui retournent une valeur :
  - appel

```
x = nom_fonction(arg1, arg2, …);
```

  - définition

```
type nom_fonction(type1 arg1, type2 arg2, …){
    instructions;
    return resultat;
}
```

- Fonctions qui ne retournent pas de valeur :
  - appel

```
nom_fonction(arg1, arg2, …);
```

  - définition

```
void nom_fonction(type1 arg1, type2 arg2, …){
    instructions;
}
```

Fonctions : localisation

- La définition des fonctions se fait en dehors du programme “main”
- Soit avant
  - simple mais souvent moins clair
- Soit après
  - souvent plus clair
  - la fonction doit en plus être déclarée avant le programme “main”

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

float sinpluscos(float x);
void affiche(float x);

// ____________________
main(){
    float x=3.4,y;
    y=sinpluscos(x);
    affiche(y);
}
// ____________________

float sinpluscos(float x){
    return sin(x)+cos(x);
}
void affiche(float x){
    printf(“le résultat est %f\n”,x);
}
```

```c
type nom_fonction(type1 arg1, type2 arg2, …)
```

Fonctions : arguments

- Si j'appelle une fonction f(x) qui modifie la valeur de x, que se passe-t-il ? => Ca dépend
- Les tableaux :
  - sont passés par *adresse*
  - leur valeur est changée dans le programme principal également
- Les scalaires
  - sont passés par *valeur*
  - une copie de la variable x est passée à la fonction et la variable du programme principal n'est pas modifiée
  - Si on veut vraiment qu'elle soit modifiée, il faut passer en argument l'**adresse** de la variable :

```
y=f(x);
```

```
double f(double x){
    x = -x;
    return sin(x);
}
```

```
y=f(&x);
```

```
double f(double *x){
    *x = -*x;
    return sin(*x);
}
```

Fonction : applications

- *Modifier le programme précédent et créer une fonction qui calcule la factorielle d'un nombre n. Appeler cette fonction depuis le programme principal*

- *Écrire un programme qui calcule le développement en série de l'exponentielle :* $e^{-x} \approx \sum_{k=0}^{n} \frac{(-x)^k}{k!}$ *et qui la compare au résultat de la fonction exp()…*