Université de PARIS SUD
Master de Physique M1
Parcours Physique Appliquée

Année Universitaire 2008-2009

Cours/TP : S. Friot & E. Trizac

Outils informatiques pour la physique
Programmation en C (P-APP-425A)

Le but de ce module de cours et TP sur machine est l'apprentissage du langage C. Cet enseignement est complété en aval par un cours/TD optionnel intitulé "Physique Numérique et Analyse Statistique", qui débute en janvier 2007.

Le C a été créé par D. Ritchie et B.W. Kernighan au début des années 70. Le C reste aujourd'hui l'un des langages les plus répandus. Il est portable (à condition de respecter la norme dite ANSI promulguée par l'*American National Standards Institute*), et génère un code rapide grâce à un compilateur performant. En contrepartie, le compilateur présuppose que le programmeur sait ce qu'il fait ! Nous allons tenter de lui donner raison ! Les principaux défauts du C sont par ailleurs les suivants : l'accès à la zone mémoire est assez permissif, la manipulation des pointeurs conduit facilement à des bogues, et le débutant peut produire des codes non portables, voire inextricables.

1 Principales commandes UNIX

Le quasi-monopole dont jouit Microsoft rend nécessaire de résumer les principales commandes UNIX (qui est un système d'exploitation permettant une utilisation efficace et commode d'un ordinateur ; UNIX est aujourd'hui diffusé gratuitement via Linux).

Une commande UNIX est un mot clé en *minuscule*, et il faut faire bien attention aux espaces. La syntaxe est toujours de la forme : `commande [options] [paramètres]`
La commande peut être limitée à un mot clé, ou être suivie d'options et/ou de paramètres facultatifs. Les options sont en général précédées d'un tiret et suivies d'au moins un caractère. Il peut y en avoir plusieurs. Principales commandes

`cd` : *change directory*, pour changer de répertoire courant. Si on ne donne pas de paramètre, renvoie dans le répertoire principal de l'utilisateur.

`cp` : *copy*, copie un fichier source sur un fichier destination. Utiliser l'option -i pour demander confirmation avant d'écraser le fichier destination, s'il existe dejà :
`cp -i fich_source fich_dest`

`ls` : liste le contenu du répertoire courant. Nombreuses options : `-a` pour obtenir également les fichiers cachés, `-l` pour listage long etc

`more` : affichage du contenu du(des) fichier(s) texte spécifié(s). Avancement dans le fichier par `<SPACE>` pour lire la page suivante, `b` pour la page précédente, `q` pour quitter...

`mkdir` : *make directory*, crée un répertoire de nom spécifié.

`mv` : *move*, pour renommer un fichier. L'option -i est une sage précaution
`cp -i fich_source fich_dest`

`ps` : donne la liste des processus et leur numéro. La commande `kill -9 numéro` tue le "job" référencé par "numéro". Utile : `top` donne le hit parade des "jobs" les plus gourmands en CPU (Central Processing Unit).

`pwd` : *print working directory*

`rm` : *remove*, pour détruire un fichier. Là aussi, utiliser l'option -i
`rm -i fichier`

`rmdir` : *remove directory* pour supprimer un répertoire.

`vi` : un éditeur de texte, pour les spécialistes

... et bien d'autres encore. N'oubliez pas : "*if you need help, ask the man*". Par exemple, `man top` vous expliquera les subtilités de la commande `top`, et avec `man man`, vous saurez tout sur la commande `man` !

Le système de fichiers est hiérarchisé : il peut y avoir plusieurs niveaux qui permettent de se déplacer dans un "arbre" de répertoires, pour arriver aux "feuilles" qui correspondent aux fichiers (données, programmes, exécutables, textes...). Tout élément peut être protégé (en lecture, écriture ou exécution), voire même inaccessible (protégé en lecture/écriture). Utile : à un endroit donné de la hiérarchie de répertoires, le répertoire "père" est désigné par .. On y accède via `cd ..`

2 Tour initiatique

1) Ecrire un programme qui affiche un message quelconque mais décent à l'écran.

2) Ecrire un programme qui calcule, avec l'instruction `sizeof`, la taille en octets des différents types de données élémentaires : `char, short, int, long` pour le type entier, et `float, double, long double` pour le type réel.

3) Convertir une lettre minuscule donnée par l'utilisateur en majuscule.

4) L'utilisateur rentre une heure au format 12H20. Afficher, avec le même format, l'heure en question plus une demi-heure.

5) Ecrire un programme qui alloue un tableau de 10 entiers, les initialise avec des valeurs de 9 à 0, puis affiche ces valeurs.

6) Ecrire un programme C qui calcule la somme de deux entiers rentrés par l'utilisateur.

7) Ecrire un programme C qui calcule au choix la somme, le produit, ou le quotient de deux entiers rentrés par l'utilisateur. Réécrire ce même programme, dans un autre fichier, en employant un `switch`

8) Ecrire un programme qui affiche, à l'aide du caractère `^`, une pyramide à n étages, où n est donné par l'utilisateur. Par exemple, pour $n = 2$:

```
    ^
  ^   ^
```

9) Qu'obient-on avec les instructions suivantes ?

```
i=1 ;
j=++i ;
printf(" (i,j) %d\n",(i,j)) ;
```

10) La valeur moyenne $\overline{x}$ et l'écart-type σ d'une suite de nombres $\{x_i\}_{1 \leqslant i \leqslant N}$ sont définis par :

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \text{et} \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2} = \sqrt{\overline{x^2} - \overline{x}^2}$$

Le carré de l'écart-type, σ^2, est aussi appelé variance de la distribution. Ecrire un programme permettant la lecture de N nombres, leur rangement dans un tableau et qui calcule la moyenne et l'écart-type de cette distribution.

11) Illustrer par un exemple la différence entre incrémentation préfixée (`++i`) et post-fixée (`i++`).

12) Illustrer la différence entre négation binaire et négation logique, par exemple sur `int i=2`. Afficher la négation logique de 0 au format signé `%d`, au format non signé `%u` et au format non signé short `%hu`. Méditer.

13) La précision d'un format de réel est la plus petite valeur positive p telle que $1 + p \neq 1$. Ecrire un programme qui calcule la précision associée aux formats `float`, `double` et `long double`. On affichera les résulats avec les formats respectifs `%g, %le` et `%Lg`.
Données utiles auxquelles comparer les résultats précédents : la plage des valeurs positives possibles en `float` va de 3.10^{-38} à 3.10^{38} ; en `double`, de 3.10^{-308} à 3.10^{308}. Pour vérifier les résultats, on pourra afficher les valeurs des variables `FLT_EPSILON` et `DBL_EPSILON`, définies dans le fichier d'en-tête `float.h`, et qui donnent respectivement la précision relative des formats `float` et `double`.

14) **Attention aux effets de bord** (qui consistent à modifier une variable pendant son évaluation ou par l'évaluation d'une expression). On déclare et initialise la variables entières `i` par `int i=1`. Qu'obtient-on (résultat du test, et valeur de `i`) suivant que la déclaration est suivie de l'une des instructions suivantes ?

```
i==0 && ++i
++i  && i==0
i==1 || ++i
++i  || i==1
```

15) **Les nombres pseudo-aléatoires**
L'ordinateur est une machine déterministe qui ne peut *stricto sensu* pas produire de nombres aléatoires. Toutefois, il est possible de générer des suites de nombres qui, à "toutes" fins pratiques, se comportent comme des séries aléatoires. On parle alors de nombre pseudo-aléatoires. Plusieurs méthodes existent (cf le cours optionnel de physique numérique), et nous nous contenterons ici d'utiliser la fonction `rand` de la bibliothèque standard qui retourne un entier pseudo-aléatoire dans l'intervalle [0,`RAND_MAX`]

où RAND_MAX=$2^{31}-1 = 2147483647$. Avant le premier appel de `rand()`, on peut exécuter la commande `srand(time(0))`, qui initialise la graine (*seed*) du générateur, à l'aide du résultat fourni par `time(0)` qui correspond au temps en secondes, écoulé depuis une date de référence (souvent 1970). On s'assure ainsi que la suite pseudo-aléatoire sera différente à chaque exécution du programme.
Programmer un jeu demandant de deviner un entier choisi pseudo-aléatoirement par la machine entre 0 et 10. On comptera le nombre de coups mis par l'utilisateur pour trouver le nombre, et l'on pourra préciser à chaque coup si l'essai est inférieur ou supérieur au nombre à deviner.

16) Après la déclaration

```
struct vitesses
  {
    int x ;
    int y ;
    int z ;
  } c ;
```

l'affectation suivante serait-elle correcte ? `vitesses.x=10 ;`

17) Quel est le problème dans le programme suivant ?

```
# include <stdio.h>                          /* pour printf, scanf */
main()
{
  long n ;
  int cnt ;                        /* compte les nombres pairs saisis */
  printf("Le programme avale les entiers pairs,\n") ;
  printf("jusqu'à ce que vous tapiez un nombre impair.\n") ;
  while ( !(n % 2))           /* tant qu'un nombre pair a ete entré */
  {
    printf("\n\n Un nombre pair SVP : ") ;
    scanf("%ld", &n) ;
    cnt++ ;
  }
  printf("\n\nVous avez entré %d nombres pairs :", cnt-1) ;
}
```

3 Classes de mémorisation

1) Qu'effectue le programme suivant, suivant que l'on déclare `x` comme `static`, comme c'est le cas ci-dessous ou `auto` dans la fonction `fonc` (c'est-à-dire via `auto int x=1` ou de manière équivalente `int x=1`) ?

```
#include <stdio.h>
main()
{
   fonc() ;
   fonc() ;
}
```

```
fonc()
{
   static int x = 1 ;
   printf("Appel no %d de fonc : la valeur de x est %d\n", x, x) ;
   x++ ;
}
```

2) Les quelques lignes qui suivent fournissent un exemple de définition de variables de même nom avec des domaines de validité se recoupant. Quel est le résultat de l'éxécution du programme ?

```
#include <stdio.h>
int x = 1000 ;
main()
{
   int x = 2000 ;
   printf("Valeur de x avant bloc intrieur %d", x) ;
   {
      int x = 3000 ;
      printf("\n Bloc interne de main : valeur de x %d\n", x) ;
   }
   printf("Valeur de x apres bloc intrieur %d", x) ;
   fonc() ;
}
fonc()
{
   int x = 4000 ;
   printf("\n fonc : la valeur de x est %d\n", x) ;
}
```

3) Le programme qui suit présente plusieurs utilisations erronées de variables locales. Lesquelles ?

```
#include <stdio.h>
int global = 1000 ;
main()
{
   int local1 = 2000 ;
   printf("Bloc externe main : valeur de global %d\n", global) ;
   printf("Bloc externe main : valeur de local1 %d\n", local1) ;
   printf("Bloc externe main : valeur de local2 %d\n", local2) ;
   printf("Bloc externe main : valeur de local3 %d\n", local3) ;
   {
      int local2 = 3000 ;
      printf("Bloc interne main : valeur de global %d\n", global) ;
      printf("Bloc interne main : valeur de local1 %d\n", local1) ;
      printf("Bloc interne main : valeur de local2 %d\n", local2) ;
      printf("Bloc interne main : valeur de local3 %d\n", local3) ;
   }
   fonc() ;
```

```
}
fonc()
{
   int local3 = 4000 ;
   printf("\n fonc : valeur de global est %d", global) ;
   printf("\n fonc : valeur de local1 %d", local1) ;
   printf("\n fonc : valeur de local2 %d", local2) ;
   printf("\n fonc : valeur de local3 %d", local3) ;
}
```

4 Algorithmes élémentaires

1) Ecrire un programme C qui affiche le triangle de Pascal des coefficients binomiaux C_n^p [noté aussi $\binom{n}{p}$ dans les pays anglo-saxons] :

$$\begin{array}{cccc} 1 & & & \\ 1 & 1 & & \\ 1 & 2 & 1 & \\ 1 & 3 & 3 & 1 \end{array}$$

L'utilisateur rentre le nombre de lignes souhaitées (limité à 10). On ne calculera pas de factorielle mais on mettra à profit la relation de récurrence (à justifier)

$$C_n^p \;=\; C_{n-1}^p + C_{n-1}^{p-1}.$$

On pourra stocker les valeurs des C_n^p dans un tableau, avec allocation statique de mémoire. Dans quels domaines des mathématiques et de la physique Pascal (1623-1662) s'est-il illustré ? De son vivant, qui était sur le trône ?

2) Ecrire un programme qui détermine les nombres premiers inférieurs à un entier donné, en appliquant un crible d'Eratosthène.
A l'âge de 15 ans, Carl Gauss (1777-1855) avait remarqué que la densité de nombres premiers au voisinage de n était $1/\log n$. Pouvez-vous retrouver ce résultat ? Une de ses conséquences est qu'un entier pris au hasard a *grosso modo* une chance sur $\log n$ d'être premier (une chance sur 230 pour un nombre de 100 chiffres).

3) Recherche dans un tableau
On a stocké N entiers dans un tableau. Le problème est ensuite de savoir si un entier i se trouve dejà dans le tableau. Ecrire un programme permettant de répondre à cette question, et, le cas échéant, de donner la position de i dans le tableau. On demande deux versions :

a) une recherche séquentielle (comparaison successive de i aux valeurs du tableau).

b) une recherche dichotomique, dont on précisera le principe. On suppose que les N entiers sont rentrés par ordre croissant.

Lorsque N est grand, quel est le nombre de comparaisons effectuées (au pire, ou plutôt, en général) dans chaque version du programme ? Quelle est donc la méthode la plus efficace ?

4) Il arrive qu'il faille ordonner les valeurs rangées dans un tableau. Imaginez un procédé de tri pour un programme lisant une série de valeurs et devant les classer par ordre croissant. Quelle en est la *complexité* (c'est-à-dire le nombre d'opérations nécessaires, en supposant grand le nombre de données concernées, cf ci-dessous question 5) ? Nous reviendrons sur le problème du tri avec la notion de récursivité (Quicksort, cf partie 7).

5) Intermezzo
La théorie de la complexité algorithmique s'intéresse à l'estimation de l'efficacité des algorithmes, avec la question : entre différents algorithmes réalisant une même tâche, quel est le plus rapide ? Pour que l'analyse ne dépende pas de la vitesse d'exécution de la machine sur laquelle le programme sera implémenté, il faut utiliser comme unité de comparaison des "opérations élémentaires" (comparaison de valeurs, opérations arithmétiques ou sur des pointeurs...) en fonction du nombre ou de la taille des données en entrée. On étudie systématiquemnt la complexité *asymptotique* (données de grande taille), en utilisant les notations de Landau [comme par exemple $\mathcal{O}(n^3)$ etc]. Quelles sont les limites de cette approche ? Son intérêt ?

6) On définit la suite

$$u_n = \frac{1}{2}\left(u_{n-1} + \frac{x}{u_{n-1}}\right).$$

Pour toute valeur initiale $u_0 > 0$, cette suite converge vers $\sqrt{x}$. Pourquoi ? En déduire une approximation de $\sqrt{10}$ avec une précision meilleure que 10^{-4}. Comparer ce résultat à celui que donne la fonction `sqrt` de la bibliothèque `math.h` (compiler avec l'option `-lm`).

7) Zéros d'une fonction : recherche par dichotomie
On considère une fonction continue possédant exactement un zéro sur l'intervalle $[a, b]$; $f(a)$ et $f(b)$ sont alors de signe opposé. On considère ensuite le point milieu $c = (a+b)/2$. Si $f(c)$ est du même signe que $f(a)$ (resp. $f(b)$), le zéro est dans $[c, b]$ (resp. $[a, c]$). Par itération, il est ainsi possible de localiser le zéro avec une précision *a priori* arbitraire. Appliquer cette méthode à un calcul de racine carrée, en considérant la fonction $f(x) = x^2 - A$. On renverra un message d'erreur si $f(a)$ et $f(b)$ sont de même signe. Si l'on souhaite connaître le zéro avec une précision p, combien d'itérations nécessite la recherche par dichotomie ? On donnera le résultat en fonction de a, b et p.

8) Zéros d'une fonction : méthodes de NEWTON et de la sécante
On cherche un zéro de la fonction $f(x)$, supposée continue et dérivable. Etant donnée une estimation x_1 pour le zéro, on détermine x_2, le zéro de la droite tangente à la courbe $y = f(x)$ en x_1 (faire un dessin). On remplace alors x_1 par x_2, et on poursuit par itération. Exprimer x_{i+1} en fonction de x_i, $f(x_i)$, et $f'(x_i)$. Réfléchir aux limitations de cette méthode. Ecrire ensuite un programme qui calcule $\sqrt{A}$ en recherchant le zéro de $f(x) = x^2 - A$, avec $A > 0$.
La méthode précédente, due à NEWTON, n'est pas très utile en ce qu'elle nécessite la connaissance de la fonction dérivée $f'(x)$. Une solution consiste à considérer le schéma itératif suivant :

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

Justifier ce choix, avant de le mettre en œuvre numériquement, de nouveau pour calculer $\sqrt{A}$.

5 Les pointeurs

1) Quelle est la taille en octets d'un pointeur vers `int` ? d'un pointeur vers un `long double` ?

2) Déclarer deux variables `char`, deux `short`, deux `int`, deux `long`, deux `float` et deux `double`. Afficher les adresses mémoire correspondantes. Que constate t-on ?

3) Reprendre l'exercice 1 de la partie 4 avec allocation dynamique de la mémoire. On définira ainsi un `int **` pour manipuler les valeurs des coefficients du binome.

4) (facultatif) On définit `i` comme un `unsigned long` (i.e. un entier stocké sans bit de signe sur 4 octets), et `pi` est un pointeur de type `unsigned short *` (où le format entier `short` est stocké sur 2 octets, sans bit de signe dans le cas `unsigned`). On initialise ces deux variables par
`i=4294967295 ; pi= &i ;`.
Quelle est la valeur de `pi[0]` ? Même question lorsque `pi` est un pointeur vers `unsigned long *` (on pourra comparer les résultats d'affichage avec les formats `%d` et `%u`). On remarquera que $2^{32} - 1 = 4294967295$, dont l'écriture en base deux ne comporte que des 1. On rappelle aussi que $2^{16} - 1 = 65535$.

5) (facultatif) Refaire l'exercice 10 de la partie 2 en utilisant un pointeur au lieu d'un tableau. L'utilisateur rentre d'abord le nombre de mesures au clavier. La mémoire nécessaire est ensuite allouée par `malloc`.

6) (facultatif) Pour une puissance de deux 2^n entre 2^0 et 2^8, afficher le nombre écrit en toutes lettres (l'utilisateur spécifie n). Les mots représentant les nombres seront rangés dans un tableau de pointeurs, afin de ne pas perdre d'espace mémoire. On fera ainsi appel à un pointeur de pointeur (vers `char`).

7) Que signifient les déclarations suivantes ?

```
const char *p
char * const p
const char * const *p
```

8) Que signifient les déclarations suivantes ?

```
int (*i)[4]
int *i[4]
```

6 Les fonctions

1) Faire l'exercice sur Bill Gates sur la page web

2) Ecrire une fonction qui prenne deux entiers comme arguments, et en échange les valeurs numériques.

3) Ecrire une fonction qui prenne comme argument une chaîne de caractère, et l'affiche à l'écran.

4) Ecrire et tester une fonction qui affiche un tableau 2D sous forme de matrice.

5) Reprendre l'exercice 3, en passant `printf` comme argument de la fonction d'affichage, en plus de la chaîne de caractère. On utilisera un pointeur de fonction.

6) A l'aide de la fonction prédéfinie `rand`, écrivez une fonction qui produise des nombres (pseudo-)aléatoires compris entre deux entiers p et q.

7) Que définit-on par `double * (*alpha[5])(char *,int)` ?
Quel est l'effet de l'instruction `(*alpha[0])(&x,10))` ?
Quel doit-être le type de `x` ?

8) Que signifient les déclarations suivantes ? On commence par réviser la question 8 de la partie 5

```
int (*i)[4]
int *i[4]
int (*i)()
int *i()
int (*i[5])()
```

9) Une fonction doit prendre comme paramètre un tableau d'entiers déclaré par `int x[2][3][4]`. Comment définir le paramètre formel correspondant (argument de la fonction lors de sa déclaration) ? Comment faudrait-il définir un pointeur vers `x` ?

7 La récursivité

1) Faire les exercices sur les éléphants sur la page web

2) Ecrire un programme qui calcule $n!$ de manière récursive, puis de manière itérative. Comparer les résultats à l'approximation de Stirling :

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

3) Faire l'exercice sur la récursivité et le dépilage sur la page web.

4) L'algorithme Quicksort fournit une des techniques de tri les plus efficaces.

a) On choisit un élément (pivot) dans la liste, généralement proche du milieu.

b) On divise les valeurs du tableau à trier en deux sous-ensembles, de telle sorte que les éléments situés avant (resp. après) le pivot soient tous inférieurs (resp. supérieurs) au pivot.

c) On applique récursivement le procédé aux deux sous-listes ainsi obtenues.

Le temps mis pour classer n nombres croît ainsi "typiquement" comme $n \log n$. Comparer ce résultat à la performance d'un algorithme de tri naïf.

5) "Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois

de son existence ?" se demandait Leonardo PISANO, sous le pseudo FIBONACCI, dans les années 1200. En notant F_n le nombre de couples de lapins au mois n, le problème se formalise en

$$F_n = F_{n-1} + F_{n-2} \qquad \text{pour } n \geqslant 2, \quad \text{avec } F_0 = F_1 = 1.$$

On retrouve la suite de Fibonacci intervient dans de nombreux domaines des mathématiques, en musique (cf les compositions de Béla BARTÓK), et semble t-il dans plusieurs structures végétales ou animales, voir par exemple le chapitre consacré à la phyllotaxie sur le site `http ://www.lps.ens.fr/~douady/`

Ecrire une fonction qui calcule F_n de manière récursive et une autre qui fasse le calcul itérativement. Dans les deux cas, estimer la complexité algorithmique (par exemple en comptant le nombre d'appels récursifs). En quoi l'exemple de la suite de Fibonacci est-il pédagogiquement catastrophique si l'on cherche à illustrer l'intérêt de la récursivité ?

6) Les tours de Hanoï

Dans le jeu des tours de Hanoï, inventé par le mathématicien français E. LUCAS en 1883, on dispose de n disques de tailles différentes, et de trois piquets (P1, P2, P3). Les disques, percés en leur centre, sont initialement empilés sur P1. Le but du jeu est de déplacer tous les disques sur P3 en respectant les règles suivantes : 1) on ne déplace qu'un seul disque à la fois et 2) un disque ne peut être posé sur un disque plus petit que lui.

Résoudre d'abord manuellement le problème avec $n = 3$ disques (en 7 coups). Ecrire ensuite un programme C qui affiche la séquence des déplacements permettant de résoudre le problème général de n disques. On pourra écrire une fonction récursive à 4 arguments : le nombre de disques à déplacer, le piquet de départ, le piquet intermédiaire, et le piquet d'arrivée. Quelle est la complexité de votre algorithme (nombre de déplacements nécessaires D_n en fonction de n) ? On pourra établir une relation de récurrence entre D_n et D_{n-1}. Combien de temps faudrait-il *grosso modo* pour résoudre le problème des 64 disques (sur l'ordinateur le plus puissant disponible aujourd'hui) ?