TP C avancé n^o 1 Licence Informatique L3 - 2021-2022
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux UFR SFA - Université de Poitiers

Reprise en main

Table des matières

1 Résumé et cadre de travail

Dans ce TP il s'agit de reprendre en main le langage C avec des exercices de niveaux de difficultés très variables. Des exercices sur le shell (*bash* en l'occurrence) sont également présents.

Nous travaillons sous Linux en mode console (ou ligne de commande), notamment pour la compilation des codes sources.

Du code vous est fourni et parfois la/une solution également. Lorsque la solution est fournie, elle ne doit servir qu'à se comparer avec votre solution ou à vous aider si vous êtes vraiment bloqué.

Directives quant à la compilation :

- Donc pas d'IDE [1].
- Il faut compiler très régulièrement pour éviter d'avoir plusieurs pages d'erreurs qui sont insupportables à traiter.
- De même, lorsque cela est possible, testez votre code au fur et à mesure.
- Les options de compilation suivantes sont imposées, et ce pour l'ensemble du module :
  -Wall -Wextra -pedantic -std=c99
  et il est conseillé de rajouter l'option *-g* pour le débugage.
- **Les warnings sont à éliminer à 99.9 pourcent**.
- Pour être parfaitement clair :
  - la phrase suivante est à bannir de votre vocabulaire : "ce n'est qu'un warning, ce n'est pas grave",
  - on ne bidouille pas le code sans comprendre pour enlever les warnings, on corrige le code !
  - si on est motivé, on peut ajouter l'option *-Werror*
- À votre niveau, il ne sera pas accepté un code qui ne compile pas ou qui compile avec des warnings.

Exemple d'une commande [2] [3] de compilation :

```
$ gcc -Wall -Wextra -pedantic -std=c99 -o toto toto.c
```

Et au cas où cela n'aurait pas encore été dit : ces options sont imposées lors des compilations et une compilation ne doit générer aucune erreur et aucun warning.

2 Bash

Point de cours

Vous devez maîtriser ;

- Les commandes de bases suivantes : *cd, ls, cp, mv, ln, mkdir, rm, rmdir, cat, more/less* et beaucoup d'autres
- Les chemins absolus et relatifs
- La différence entre code source et code exécutable
- Les droits des fichiers et répertoires (avec *chmod*)
- Les redirections de base : `>, >>, <, 2>, 2>>, |`

Vous devez savoir :

- lister les processus : *ps, top, jobs*
- envoyer un signal à un processus : *kill*
- lancer les processus en arrière et avant-plan : `&`, *ctrl-z, bg, fg*

1. Vous en utiliserez en POO et en Web.
2. Le signe $ est le prompt/invite du shell
3. Faire un alias dans le *.bashrc* est une bonne idée (cf. code fourni).

3 Sempiternel Hello World

Sans regarder le fichier fourni, écrivez le code, dans le fichier *hello.c*, qui affiche “Hello World !” suivi d’un retour à la ligne.

Compilez et testez le code.

Comparez votre code avec celui qui est fourni. Ils doivent être quasi identiques (à l’indentation près) ; si ce n’est pas le cas, vous devez le justifier ou corriger vos erreurs [4].

4 Archive, compression, somme de contrôle

Point de cours

Voici quelques rappels histoire de fixer le vocabulaire.

Une *archive* permet de regrouper toute une arborescence dans un seul fichier, généralement pour faciliter sa manipulation (déplacement, transmission, sauvegarde, ...).

Une archive peut être compressée. C’est généralement le cas mais pas une obligation. De fait les archiveurs proposent [a] systématiquement une option de compression.

Il s’agit juste d’insister sur le fait qu’archivage et compression sont deux processus complètement séparés.

Enfin, lorsqu’on transmet une archive, il faut s’assurer qu’elle n’a pas été corrompue (intentionnellement ou non) lors du transfert. Il est donc possible de l’accompagner d’une somme de contrôle.

Commande *tar*, voici une liste d’options :

c	créer une archive
x	extraire une archive
t	lister le contenu d’une archive
f	nom du fichier archive à créer ou à lire
z	manipulation d’une archive compressée avec *gzip*
j	manipulation d’une archive compressée avec *bzip2*

Les options *c/x/t* sont exclusives, ainsi que les options *z/j*.

La commande *md5sum* sans option permet de calculer une somme de contrôle, et avec l’option *-c* de vérifier une somme de contrôle.

a. et parfois imposent, ce qui est discutable

Récupérez le répertoire *PROJET_TEST* et recopiez-le dans un répertoire de travail (commande *cp -R*).

Créez une archive non compressée (nommée *projet.tar*) de ce répertoire avec la commande *tar*.

Déplacez l’archive dans un répertoire vide et désarchivez-la ; vérifiez que le répertoire *PROJET_TEST* est bien créé à l’endroit où il y a l’archive, puis effacez le répertoire.

Regardez la taille de l’archive. Compressez-la avec bzip2 et l’option qui permet le meilleur taux de compression. Regardez à nouveau la taille de l’archive. Enfin, en une seule commande désarchivez l’archive.

Dans le répertoire *SOURCE*, créez le fichier “verif” qui contient les sommes de contrôle des trois fichiers *.c*.
Lancez la commande de vérification des sommes de contrôle.
Modifiez un des trois fichiers et relancez la vérification.

4. Le mot “erreurs” est employé sciemment ; rappelez-vous qu’un programme qui marche n’est pas forcément un bon programme, et inversement d’ailleurs.

5 Redirections

Faites un programme qui affiche 3 messages différents :
- un avec *printf*,
- un avec *fprintf* sur la sortie standard,
- un avec *fprintf* sur la sortie erreur.

Puis il lit un entier sur l'entrée standard, entier qui est aussitôt affiché sur la sortie standard.

Chaque affichage est suivi d'un passage à la ligne.

Lancez le programme avec les redirections suivantes en essayant de deviner le résultat :

`>`	sur un fichier
`>`	sur */dev/null*
`>`	sur la commande *less* (???pourquoi cela n'a-t-il aucun sens ?)
`2>`	sur un fichier
`2>`	sur */dev/null*
`>` et `2>`	sur deux fichiers distincts
`|`	sur un fichier (???pourquoi cela n'a-t-il aucun sens ?)
`|`	sur la commande *less*

Pourquoi la commande suivante n'a que peu d'intérêt :

```
$ ls > result | less
```

De même pour la commande :

```
$ ls | less > result
```

Pourquoi la commande suivante est beaucoup plus intéressante :

```
$ ls 2> erreurs | less
```

Quelles sont les différences entre les deux commandes suivantes ?

```
$ ls > result 2> erreurs
$ ls 2> erreurs > result
```

Comment fait-on pour rediriger la sortie erreur dans un tube (i.e. avec un `|`) ?
Si quelqu'un a une solution je suis intéressé.
Si quelqu'un a une solution simple je suis très intéressé.

6 Variables shell et programmes C

Point de cours

Il y a trois prototypes valides pour la fonction *main* :

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

Le premier prototype est utilisé lorsque le programme ne prend aucun paramètre en ligne de commande.

Le deuxième prototype sert à récupérer les paramètres en ligne de commande.
- *argc* contient le nombre de paramètres (nom de l'exécutable compris)
- *argv* est un tableau de chaînes de caractères

Donc si *argc* vaut 2 alors il n'y a qu'un paramètre passé lors de la ligne de commandes :
- *argv[0]* contient le nom de l'exécutable
- *argv[1]* contient le paramètre (sous forme d'une chaîne donc)

Le troisième prototype sert à récupérer en plus les variables d'environnement du shell, enfin

une partie des variables (cf. exercice suivant) :
- *env* est un tableau de chaînes, avec une chaîne par variable d'environnement récupérée
On ne connaît pas a priori la taille du tableau *env* (alors que c'est le cas pour le tableau *argv*). Pour pallier ce problème, le tableau *env* a une case de plus initialisée à *NULL*. C'est donc un marqueur de fin (comme '\0' pour les chaînes de caractères).

Écrivez un programme, sans regarder la solution fournie, qui affiche :
- le nombre et la liste des arguments passés en ligne de commande
- le contenu du tableau *env*
Écrivez un code le plus lisible possible.

Comparez votre code avec les trois versions fournies.

7 Variables shell et programmes C (bis repetita)

Écrivez un programme qui prend un paramètre :
- le nom d'une variable d'environnement du shell
Et qui affiche sa valeur ou un message d'erreur si elle n'existe pas.
La commande suivante peut être utile :

```
$ man getenv
```

Pensez à traiter le cas où l'utilisateur oublie de passer un paramètre.

On suppose, dans la suite de l'exercice, que votre exécutable se nomme *affiche_env*. Essayez les trois commandes suivantes et vérifiez que tout est cohérent :

```
$ ./affiche_env
$ ./affiche_env TOTO
$ ./affiche_env USER
```

Essayez les commandes suivantes :

```
$ FOO=13
$ ./affiche_env FOO
```

Pourquoi le programme dit-il que la variable *FOO* n'existe pas ?

Créez une variable *BAR* initialisée à 15 pour que votre programme la détecte.

Lancez une autre console, et exécutez votre programme toujours sur la variable *BAR*. Pourquoi n'est-elle plus visible ? Que faudrait-il faire pour que cette variable soit définie automatiquement dans toutes les consoles ?

Profitez-en pour définir de façon permanente un alias pour compiler vos programmes, par exemple :

```
alias mygcc='gcc -Wall -Wextra -pedantic -std=c99'
```

Qui s'utilise alors ainsi :

```
$ mygcc -o affiche_env affiche_env.c
```

8 Relais

Écrivez un programme qui lit, avec *fgetc*, les caractères un par un sur l'entrée standard, et qui immédiatement les recopie, avec *fputc* sur la sortie standard.

Note : pour arrêter proprement votre programme (i.e. pour que *fgetc* réagisse normalement), il faut taper la séquence "Ctrl-D" en début de ligne ; *fgetc* renvoie alors la valeur *EOF*.

Dans le shell, utilisez votre programme (sans modifier le code) pour faire une copie de fichier.

9 B. a.-ba

Écrivez une fonction qui prend deux réels en paramètres et retourne la somme des dits paramètres.

Testez cette fonction, dans le *main*, en affichant la somme de 1.12032002 et 2.02127003.

10 Utilisation des anti-quotes

Essayez les deux commandes suivantes en essayant de deviner ce qu'elles font (attention le mot *tty* est entouré par des anti-quotes).

```
$ echo "Bonjour"
$ echo "Bonjour" > ‘tty‘
```

Expliquez.

De même essayez les commandes :

```
$ date
$ echo ‘date‘
$ echo ’date’
```

Pour différenciez l'utilisation des guillemets et des quotes, essayez les commandes suivantes :

```
$ echo ’Bonjour $toto’
$ echo "Bonjour $toto"
$ echo ’Bonjour $USER’
$ echo "Bonjour $USER"
```

Pourquoi la commande suivante n'a (vraisemblablement) pas de sens (attention ce sont encore des anti-quotes) ?

```
$ echo ‘Bonjour $USER‘
```

Alors que les commandes suivantes fonctionnent (ligne 1 : quotes ; ligne 2 : anti-quotes) ?

```
$ alias Bonjour=’echo Bonjour’
$ echo ‘Bonjour $USER‘
```

11 Pointeurs et allocation dynamique

Il est absolument indispensable que vous maîtrisiez les pointeurs et l'allocation dynamique.
Voici une série d'exercices qui à terme ne doit pas/plus vous poser de problème pour la bonne compréhension du module.

Point de cours

Une adresse est une valeur qui désigne le numéro d'une case mémoire.

En outre une adresse est typée, i.e. non seulement elle désigne le numéro d'une case mémoire, mais elle indique également ce qu'il y a à cette adresse (*int*, *char*, *float*, structure, ...).

C'est important car on connait ainsi le nombre d'octets concernés à partir de cette case mémoire (par exemple 4 pour un *int* [a]).

Supposons que l'adresse 0x9E58 contienne un *int* (que l'on suppose stocké sur 4 octets). Alors les cases 0x9E58, 0x9E59, 0x9E5A et 0x9E5B sont utilisées pour cet entier.

Notez que connaître le numéro d'une case mémoire n'implique pas qu'on a le droit de manipuler le contenu de cette case (en lecture et/ou écriture).

Un pointeur est une variable qui contient une adresse :
- elle est donc typée (pointeur sur *int* par exemple).
- comme c'est une variable, son contenu est stocké en mémoire et donc cette variable a elle-même une adresse. On parle alors de l'adresse d'un pointeur (ou plus improprement d'un pointeur sur un pointeur).

Pour obtenir l'adresse d'une variable (donc le numéro de la case mémoire dans laquelle est stockée la valeur de la variable), on utilise le signe '`&`' :

```
int i;
...
printf("adresse de i : %p", (void *) &i);
```

Pour déclarer un pointeur, on utilise le signe `*` après le type. On utilise le même signe, pour accéder au contenu de la case mémoire (des cases mémoires plus exactement) désignée par le pointeur.

```
// déclaration classique d'un entier
int i = 4;
// pointeur, sur un int, non initialisé (i.e. adresse aléatoire) et donc non utilisable
int *p;
// p contient l'adresse de i
p = &i;
// deux instructions identiques (on modifie la variable i)
i = 5;
*p = 5;
```

Il est rare que l'on écrive de tels codes : lorsqu'on connaît le nom d'une variable, il y a peu d'intérêt à avoir un pointeur dessus.

L'utilité est lorsqu'on passe une variable en paramètre d'une fonction :
- la fonction doit modifier la variable, le passage par pointeur est indispensable
- on veut éviter de transmettre une copie de la variable pour économiser la mémoire (utile lorsqu'on passe une structure volumineuse) ; le passage par pointeur est alors la solution.

a. ou 2 ou 8 selon l'architecture

Point de cours

Voici un moyen de retenir les significations des symboles `&` et `*`.

`&` : signifie "adresse de"
`*` : signifie "contenu de la zone pointée"

Lors d'instructions ces significations ne posent pas de problème ; lors d'une déclaration de variable le sens proposé est moins clair.

```
int *p;
```

En réalité il faut interpréter **p*. **p* est bien le contenu de la zone mémoire pointée par *p*. Ou encore **p* est bien un *int*.

Point de cours

Jusqu'à présent la réservation/libération des zones mémoire est complètement gérée par le programme. Cela n'est possible que lorsqu'on connaît a priori (au moment où on écrit le code) la taille nécessaire (il y a des exceptions).

Si on ne connaît pas la taille (généralement le nombre de cases d'un tableau) à l'avance d'une

variable, il faut utiliser l'allocation dynamique (*malloc/calloc*). L'inconvénient est qu'il faut gérer explicitement la libération (*free*) de la mémoire ; l'avantage est qu'on maîtrise la durée de vie de cette zone mémoire.

La zone mémoire utilisée par les allocations dynamiques (le tas) n'est pas la même que celle utilisée par les allocations automatiques (la pile). En général le tas est bien plus grand que la pile.

```c
// pointeur, sur un int, non initialisé (i.e. adresse aléatoire) et donc non utilisable
int *p;
// on demande au système d'allouer, dans le tas, une zone mémoire pour un int
p = malloc(sizeof(int));
// maintenant on peut utiliser cette zone mémoire
*p = 5;
printf("A l'adresse %p il y a la valeur %d", (void *) p, *p);
// et ne pas oublier de désallouer
free(p);
// *p ne doit maintenant plus être utilisé
```

```c
// pointeur, sur un int, non initialisé (i.e. adresse aléatoire) et donc non utilisable
int *p;
// on demande au système d'allouer, dans le tas, une zone mémoire pour un tableau de 10 int
p = malloc(10*sizeof(int));
// maintenant on peut utiliser cette zone mémoire
p[7] = 5;
printf("A l'adresse %p le tableau commence", (void *) p);
printf("Dans la 8me case il y a la valeur %d", p[7]);
// et ne pas oublier de désallouer
free(p);
// *p ou p[] ne doit maintenant plus être utilisé
```

11.1 Pointeurs sans allocation

Écrire un programme (tout le code est dans le *main*) qui :

- déclare une variable i entière
- l'initialise à 99
- l'affiche
- lui ajoute 2
- l'affiche

Continuez le programme qui :

- déclare une variable *pi* qui est un pointeur sur un entier
- l'initialise pour qu'elle pointe sur la zone mémoire de la variable i
- affiche la valeur du pointeur et de la zone pointée
- ajoute 2 à la zone pointée
- affiche la valeur du pointeur et de la zone pointée
- affiche le contenu de la variable i

11.2 Oubli d'allocation

On demande dans cet exercice un code incorrect avec un warning à la compilation qui le signale.

Dans la fonction *main* déclarez un pointeur sur un entier sans l'initialiser.
Affichez la valeur du pointeur, puis la valeur de la zone pointée.

Faites un autre programme avec exactement le même code mais dans une autre fonction autre que le *main*.

Lancez les deux programmes et comparez.

Relancez les deux programme avec *valgrind*. Si votre exécutable s'appelle *a.out*, lancez le ainsi :

```
$ valgrind ./a.out
```

11.3 Non-sens

Pourquoi ce code n'a pas de sens (et d'ailleurs ne compile pas) ?

```
int i = 5;
printf("l'adresse de l'adresse de i vaut : %pn", (void *) &(&i));
```

11.4 Allocation de structures

Définissez une structure qui contient :
- un *float*
- un tableau static de *float* de taille 10
- un entier qui est la taille du tableau ci-dessous
- un tableau dynamique de *float*

Écrivez une fonction qui initialise une telle structure. La fonction prend trois paramètres :
- la structure à initialiser
- un *float*
- un entier qui est la taille du second tableau

Initialisez tous les flottants à votre convenance.

Écrivez une fonction qui crée un tableau de structures. La taille du tableau est passée en paramètre (et c'est le seul paramètre) et la fonction renvoie le tableau créé. Les cases du tableau doivent être initialisées avec la fonction précédente (les paramètres sont laissés à votre choix).

Écrivez une fonction qui libère les ressources mémoire d'un tableau créé par la fonction précédente. La fonction prend comme seuls paramètres le tableau à libérer et sa taille.

Écrivez un *main* qui crée un tableau de 15 cases et le libère.

Testez votre programme avec *valgrind* pour assurer notamment qu'il n'y a pas de fuite mémoire.

11.5 Tableau à deux dimensions

Le but est de manipuler des tableaux dynamique de booléens à deux dimensions (i.e. on accède à une case avec la notation `tab[3][7]`).
On suppose que le premier indice correspond à la coordonnée en x et le second indice à la coordonnée en y. Si la case est à *true* alors le point est allumé.

Écrivez une fonction qui crée un tel tableau. Les deux paramètres sont la taille en x et la taille en y.
Vous initialisez les cases comme vous le souhaitez (une fonction vous est fournie).

Écrivez une fonction qui affiche le tableau en mode pseudo-graphique, par exemple une '*' si la case est à vrai, et '-' sinon.

Écrivez une fonction qui libère un tel tableau.

Écrivez un *main* qui teste vos fonctions.
Par exemple la taille en x à 70 et la taille en y à 20.

De nouveau testez votre programme avec *valgrind*.

11.6 Affectation de tableaux ?

On ne peut pas faire une affection de tableau (i.e. une recopie d'un tableau dans un autre avec un '=') en C. Et pourtant on peut écrire le code suivant (qui comporte par ailleurs une erreur) :

code avec une erreur

```
int tab[5] = {1, 2, 3, 4, 5};
int * autreTab = malloc(5 * sizeof(int));
autreTab = tab;
```

Expliquez cette apparente contradiction, et l'erreur induite par ce code.

12 Tableaux

Il n'y a pas d'allocation dynamique dans cet exercice, et on manipule des tableaux d'entiers.

Écrivez les fonctions suivantes :

- initialisation d'un tableau avec des valeurs aléatoires comprises entre deux bornes (avec *rand* et *srand*[5])
- affichage joli d'un tableau
- moyenne des cases d'un tableau
- somme de deux tableaux au sens géométrique du terme : le résultat est un tableau dont chaque case est la somme des deux cases correspondantes des deux tableaux à sommer
- trouver le numéro de la case qui contient le second maximum du tableau (on écrit l'algorithme en supposant que toutes les cases sont distinctes)
- tri à bulle

5. Pensez à lire le manuel des deux commandes.