

TP 1 : prise en main

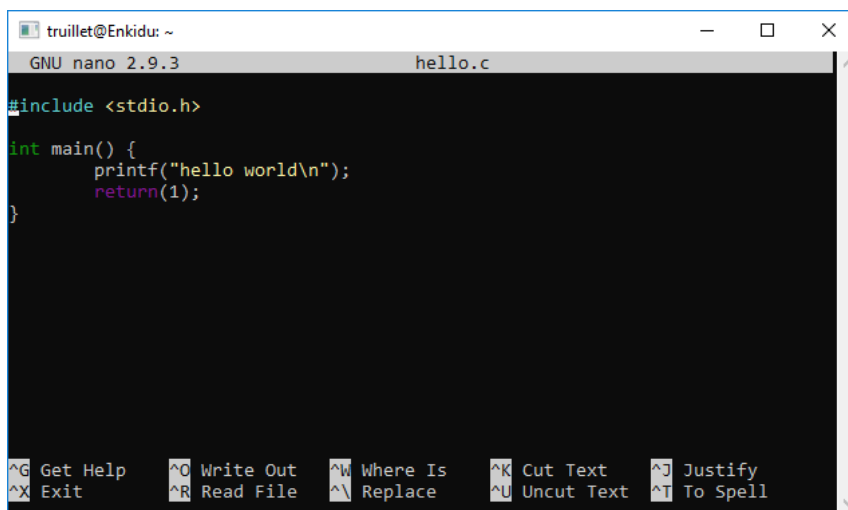
Tous les TP vont s'appuyer sur trois outils principaux :

- Une interface de commande en ligne (**CLI** - Command Line Interface) ou **shell**
- Un éditeur de texte (comme **nano**, **emacs**, **vi**, ou autre de votre choix ...)
- Un outil de compilation (**gcc**)

L'interface nous permettra de taper les différentes commandes nécessaires à l'écriture, la compilation et l'exécution de notre code ; l'éditeur à créer ou modifier notre code et l'outil **gcc** à traduire notre code en langage machine.

1. Exercice de démarrage

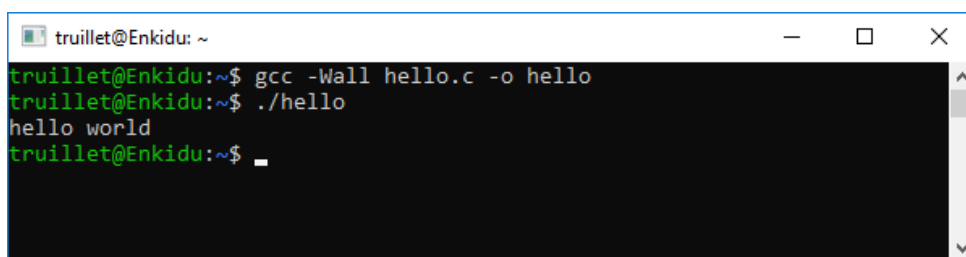
Tapez dans un fichier **hello.c** le code (**mythique**) suivant (cf. Figure 1) :



```
truillet@Enkidu: ~  
GNU nano 2.9.3 hello.c  
  
#include <stdio.h>  
  
int main() {  
    printf("hello world\n");  
    return(1);  
}
```

Figure 1 – « Hello world »

Compilez et exécutez ensuite le code produit (cf. Figure 2).



```
truillet@Enkidu: ~  
truillet@Enkidu:~$ gcc -Wall hello.c -o hello  
truillet@Enkidu:~$ ./hello  
hello world  
truillet@Enkidu:~$
```

Figure 2 – chaîne de compilation et d'exécution

Nota1 : **gcc** est l'outil de compilation que nous utiliserons en TP. Son fonctionnement précis sera précisé ultérieurement. L'option **-o hello** indique à **gcc** que le fichier produit devra s'appeler **hello** (et non **a.out** par défaut).

Nota2 : pourquoi faut-il appeler l'outil **gcc** avec la commande « **gcc** », alors qu'il faut appeler notre programme avec la commande **./hello** ?

A quoi sert ce « **./** » ? La raison est assez simple. Afin de pouvoir exécuter un programme en ligne de commande, il faut donner le chemin de ce programme sur le disque. C'est ce que nous faisons avec **./hello** en référant un chemin relatif grâce à « **./** ».

Comme **gcc** est un outil couramment utilisé, et il a été placé dans un répertoire que l'environnement d'exécution des commandes connaît. L'ensemble des chemins dans lesquels l'environnement de commande cherche les binaires que vous lancez est stocké dans la variable d'environnement **PATH**. Pour connaître la liste des répertoires dans lesquels l'environnement en ligne de commandes cherche les binaires, tapez **echo \$PATH**.

Vous voyez s'afficher une liste de chemins, séparés par « : ». Lorsque vous lancez **gcc**, l'environnement d'exécution des commandes cherche alors dans chaque chemin précisé, l'un après l'autre, s'il contient un fichier appelé **gcc**. Dès qu'il en trouve un (le premier donc !), il l'exécute.

2. Quelques commandes systèmes utiles

Répertoires

Lorsqu'un utilisateur ouvre un terminal ou se connecte à une machine via un terminal, il est positionné par défaut dans un répertoire qualifié de **home directory** (souvent noté **~**).

Quelques commandes utiles :

- **pwd** (**P**rint **W**orking **D**irectory) pour donner votre position dans l'arborescence
- **ls** (**L**i**S**t) pour lister les noms de fichiers du répertoire courant
- **mkdir dir** (**M**a**K**e **D**IRectory) pour créer le répertoire *dir*
- **cd dir** (**C**hange **D**irectory) pour aller dans le répertoire *dir*
- **cp fich1 fich2** (**C**opy) pour copier le fichier *fich1* vers le nouveau fichier *fich2*
- **mv fich1 fich2** (**M**ove) pour changer le nom du fichier *fich1* vers *fich2*

Pour chacune des commandes, il existe un descriptif détaillé, accessible via la commande **man**, par exemple **man ls**.

Exercice

Connectez-vous sur une session Unix. Depuis le terminal :

- trouvez votre position dans l'arbre depuis la racine /
- créez le répertoire **TP1** puis allez dans ce répertoire
- listez les fichiers du répertoire (y compris les fichiers cachés) ?

Redirection des entrées-sorties et « pipes »

Comme vous avez pu le constater, toutes les commandes émises depuis une fenêtre terminal, effectuent leurs sorties sur le terminal (**stdout**). Pour les rediriger sur d'autres supports on peut les faire suivre, entre autres, des symboles **>** ou **>>** :

> indique que la sortie n'est plus le terminal mais un autre support (fichier, etc.)

>> est utilisé pour concaténer la sortie d'une commande à un support déjà existant (rallongement)

De façon analogue, on peut rediriger le résultat d'une commande en entrée d'une autre commande. Le symbole utilisé est **|** et il s'agit d'un « pipe ».

Exercice

Les commandes qui vont être utilisées ici :

- **date** pour afficher la date
- **cat** pour afficher le contenu d'un fichier

Placez-vous dans votre sous-répertoire **TP1**. Envoyez le résultat de la commande **date** dans le fichier **output.txt**. Vérifiez le contenu du fichier créé avec la commande **cat**. Quels sont les droits du fichier **output.txt** ?

Vous allez maintenant utiliser un « *pipe* », outil très utile pour rediriger ou filtrer les sorties vers une autre commande. Pour cela, revenez dans votre répertoire « *home* » (**cd** ou **cd ~** par exemple). Localisez votre fichier **output.txt** en utilisant la commande **ls -lR | grep output.txt**

3. Programmation en C et usage de pointeurs

Comme nous l'avons vu en cours, un programme C commence à s'exécuter à partir d'une fonction **main**. La signature de cette fonction est **int main (int argc, char* argv[]);**

Rappel

Au lancement du programme, le paramètre **argc** est initialisé avec le nombre d'arguments passés sur la ligne de commande (tous séparés par une espace). Le paramètre **argv** est initialisé avec ces chaînes de caractères.

Prenons l'exemple de **gcc**, commande elle aussi écrite en C. Lorsque vous lancez **gcc** avec la ligne de commande suivante : **gcc mon_prog.c -o mon_prog** alors la fonction **main** de **gcc** (aussi appelée point d'entrée du programme), est appelée avec les valeurs suivantes pour **argc** et **argv** : **argc** vaut **4** et **argv** vaut {« **gcc** », « **mon_prog.c** », « **-o** », « **mon_prog** »}

Nous allons écrire un programme appelé « sum » qui prend en paramètre deux entiers.

Dans cet exercice, l'objectif est simplement de vérifier que lorsqu'il le lance, l'utilisateur a donné le bon nombre de paramètres et que ces paramètres sont des entiers.

Vous définirez en outre la fonction suivante dans votre code **void display_message(char* msg);**

Comme son nom l'indique, cette fonction affiche simplement à l'écran le message qui lui est passé en paramètre.

3.1 Exercice 1

Ecrire le code de la fonction **main** de sorte à afficher le message suivant : **Wrong usage, 2 parameters expected: ./sum param1 param2** lorsque l'utilisateur n'a pas passé le bon nombre de paramètres.

Testez votre implémentation en compilant et en exécutant le programme avec différentes configurations comme par exemple : **./sum 2 3 4**

3.2 Exercice 2

Ecrire le code de la fonction **int isCharInteger(char c);** Cette fonction renvoie 1 si le paramètre est un entier, et 0 sinon. Vous utiliserez la construction **switch/case** pour implémenter cette fonction.

3.3 Exercice 3

Compléter le code de la fonction **main** de sorte à afficher le message suivant **Wrong usage, parameters param1 and param2 should be integers when executing ./sum param1 param2** lorsque l'utilisateur n'a pas passé des paramètres sous la forme d'entiers.

Pour vérifier que les deux paramètres passés par l'utilisateur sont bien des entiers, il faudra vérifier que chaque caractère de chaque chaîne est un entier en utilisant la fonction **isCharInteger**.

Testez votre implémentation en compilant et en exécutant le programme avec différentes configurations.

3.4 Exercice 4

Nous allons maintenant continuer l'implémentation du programme **sum** que nous avons commencé précédemment. Pour cela, nous allons implémenter la fonctionnalité qui somme la valeur des entiers passés en paramètres sur la ligne de commande, puis qui affiche la valeur obtenue.

Afin de convertir une chaîne de caractère contenant des nombres en une valeur entière, vous allez coder la fonction **`int charToInteger(char c)`**, qui renvoie la valeur de `c` sous la forme d'un entier.

Codez ensuite la fonction **`int stringToInteger(char * c)`**, qui renvoie la valeur de `c` (un tableau de caractères représentant des chiffres) sous la forme d'un entier.

3.5 Exercice 5

Afin d'afficher le résultat du calcul, nous allons avoir besoin d'utiliser une fonction d'entrée sortie. Dans cet exercice, nous allons utiliser la fonction de sortie la plus simple : **`printf`** (utilisez **`#include <stdio.h>`**). Pour accéder à sa documentation depuis le terminal, tapez **`man printf`**

Modifiez votre programme dans **`sum.c`** pour afficher le résultat donné par la somme des entiers passés en ligne de commande.

Testez votre programme comme suit : **`sum 21 21`** puis **`sum 21 -21`**. Une fois que vous aurez obtenu le chiffre **`42`** en résultat de ce test, passez au test suivant : **`sum 1024 1024`**. Le résultat est-il celui attendu ?

Nota 1 : il existe déjà, dans la librairie standard du langage C (`stdlib`), une fonction qui convertit une chaîne de caractère en une valeur numérique entière. Cette fonction s'appelle **`atoi`** (**A**SCII **t**o **I**nteger), et vous pouvez l'utiliser en remplacement de **`stringToInteger`**

3.6 Exercice 6

Modifiez le programme précédent de telle sorte que le résultat soit renvoyé à la ligne de commandes. Testez votre solution.

3.7 Exercice 7

Plutôt que d'utiliser des arguments passés en ligne de commande, vous allez maintenant utiliser des fonctions d'entrées/sorties.

Faites évoluer votre programme en utilisant des interactions avec l'utilisateur, via la fonction **`scanf`** (`int scanf(const char *format [, arg1 [, arg2]...])`)

Nota 2 : Pour pouvoir initialiser `arg1` et `arg2` dans l'appel de fonction, `arg1` et `arg2` doivent être de type *pointeur*. Nous passerons donc des adresses ou des *pointeurs* en paramètre de cette fonction.