Atelier Polytech Info3

La Programmation en C

© Pr. Olivier Gruber

Université de Grenoble, France

Introduction

Si vous lisez ces lignes, vous avez récupéré ce fichier "readme.pdf". Vous avez vu qu'il y a des archives à télécharger, une par tâche. Ces tâches seront à faire pendant cet atelier, un atelier dont le but est de vous apprendre les bases de la programmation en C. A la fin de l'atelier, vous saurez concevoir et code un petit jeu : le jeu du pendu. Pour cela, nous allons progresser par tâches, chaque tâche représentant un pas vers notre but. Chaque pas vous permettra de voir ou de revoir certains fondamentaux de la programmation et des outils utilisés.

Au début, les tâches sont des tâches d'apprentissage, ne faite pas que lire et passer à la suite. Vous ne mémoriserez pas. Lisez, comprenez, puis expérimenter par vous même. Essayer d'utiliser les concepts que vous apprenez. N'hésitez pas à demander de l'aide à ceux qui savent, discuter avec d'autres de ce que vous apprenez et comprenez.

Le langage de programmation C est relativement de bas niveau, proche des détails du matériel de votre machine. C'est ce qui le rends puissant mais aussi assez délicat à maîtriser. Nous vous conseillons de faire vos propres « anti-sèches » pour le C, qui accumuleront vos connaissances acquises. Les concepts bien sûr, mais aussi les astuces sous forme de petits bouts de code commenté. Oui, Google est là pour cela, mais cela sera toujours plus lent de toujours refaire les mêmes recherches dans Google que de retrouver votre code que vous avez déjà compris, validé et commenté.

Le fait de faire ces anti-sèches va aussi vous aider à mieux comprendre et mémoriser. Prenez des notes au fur et à mesure, lorsque vous faites les tâches, peut-être même sur papier. Puis, reprenez ces notes et faites des anti-sèches propres et claires. Nous conseillons que ces anti-sèches soient informatique, avec des fichiers dans lesquels vous pouvez faire des recherches et avoir des bouts de code qui compilent et qui tournent. Mais le papier est encore imbattable pour faire des schémas rapidement et prendre des notes désorganisées que l'on organisera plus tard.

N'oubliez pas de nous faire des retours sur cet atelier. Vos commentaires et suggestions sont les bienvenus, nous cherchons toujours à améliorer nos ateliers. Aidez nous en particulier à mieux comprendre ce qui est difficile pour vous et comment mieux vous aider sur les points difficiles.

Merci d'avance et j'espère que cet atelier vous intéressera et sera soit une bonne révision soit un bon premier apprentissage. N'oubliez pas de collaborer et d'échanger entre vous, c'est primordial.

Cordialement,

Pr. Olivier Gruber.

Task 1 – Hello World

Tout programmeur a débuté avec le programme qui affiche le texte : Hello World ! Le programme n'est pas passionnant mais c'est l'occasion de voir ou de revoir les outils essentiels pour le développeur. En effet, bien développer commence par bien connaître les bons outils. Nous n'allons donc pas déroger à cette tradition, cela sera notre première tâche.

Si vous n'avez jamais utilisé un shell de commande, reportez vous à l'appendix sur le Shell à la fin de ce document.

Avec votre éditeur préféré, par exemple nano ou gedit, ou autre, vous allez ouvrir le fichier source "task1/hw.c". Vous noterez que les deux éditeurs colorent ce que vous tapez, pour vous aidez lors de la lecture du code. Cela s'appelle la **coloration syntaxique** de votre code.

Le fichier "task1/hw.c" est un source, c'est du texte qui représente du code qu'un humain peut lire. La machine ne sait pas l'utiliser tel quel. Nous allons maintenant faire la compilation de votre source vers un fichier exécutable avec la commande suivante :

```
$ cd task1
$ gcc -g -o hw hw.c
```

Le compilateur ***gcc*** est le programme qui sait traduire du source écrit dans le langage C vers le code binaire que comprends l'ordinateur. La commande ci-dessus a demandé à *gcc* que le code binaire soit généré dans le fichier « hw ». Voyons ce que ***gcc*** a fait :

```
$ ls -al
total 28
drwxrwxr-x 2 ogruber ogruber 4096 sept.  4 11:02 .
drwxrwxr-x 4 ogruber ogruber 4096 sept.  4 10:24 ..
-rwxrwxr-x 1 ogruber ogruber 8600 sept.  4 11:02 hw
-rw-rw-r-- 1 ogruber ogruber  126 sept.  4 11:02 hw.c
```

La command ***ls*** avec les arguments « -al » liste plein d'informations sur le contenu du dossier courant. La ligne avec « . » décrit le dossier courant. La ligne avec « .. » décrit le dossier parent. Rappelez vous, vous pouvez toujours remonter au dossier parent en tapant :

```
$ cd ..
```

Mais ici, les deux lignes qui nous intéressent sont les lignes qui correspondent aux fichiers « hw » et « hw.c ».

```
-rwxrwxr-x 1 ogruber ogruber 8600 sept.  4 11:02 hw
-rw-rw-r-- 1 ogruber ogruber  126 sept.  4 11:02 hw.c
```

Le fichier « hw.c » fait 126 octets, le fichier « hw » a bien été généré est fait 8600 octets. Si nous regardons les droits (***rwx)***, nous voyons que le fichier « hw » est un fichier que vous pouvez lire et écrire (avec 'r' pour read et 'w' pour write), mais aussi exécuter (avec 'x' pour exécute), c'est ce que l'on appelle un **exécutable** :

```
-rwxrwxr-x 1 ogruber ogruber 8600 sept.  4 11:02 hw
```

En effet, les droits sont rwx (read-write-execute) pour le propriétaire, le groupe, et tous les autres utilisateurs. Lançons notre programme :

```
$ ./hw
Hello World !
```

Rappel : n'oubliez pas le point-slash devant le nom de l'exécutable, sauf si le PATH de votre shell contient « . » dans la liste des dossiers où chercher un exécutable.

1.1 La Magie de la Compilation

La magie de la compilation est formidable. A partir d'un source, pratiquement écrit en anglais, comme si on discutait avec un être humain, le compilateur va générer un binaire que l'ordinateur peut exécuter. Rendons nous compte de la complexité et de la chance que nous avons de pouvoir programmer nos ordinateurs aussi facilement.

Comment savoir ce que contient un fichier ?

On peut bien sûr regarder l'extension du ficher :

toto.txt → généralement du texte

hw.c → un source c

hw.java → un source Java

Mais cela est juste une convention de nommage. Vous pouvez essayer la commande ***file***

```
$ file hw.c
hw.c: C source, ASCII text
```

Pas super intéressant, mais pour le fichier « hw », c'est bien plus complet :

```
$ file hw
hw: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=9ab99dcd2750ead56517568251eaa5851a950bed, not stripped
```

Ouch… c'est même un peu trop complet, non ?

Mais il y a des infos pertinentes : « executable, x86-64, version » . Cela nous dit que le fichier est un exécutable, pour une architecture matérielle x86-64, soit les processeurs Intel ou AMD 64bits. Cela peut être différent sur votre machine.

Pour faire le lien avec le premier cours, qui discutait les bases de l'architecture matérielle et de la programmation, on peut faire un « dump » du contenu du fichier exécutable « hw ». Pour le faire, nous allons utiliser le programme « objdump », comme suit :

```
$ objdump -D hw
```

Oops, encore une fois, bien trop d'informations, n'est-ce pas ? On n'arrive même pas à lire la sortie. Cela va vous arriver souvent, que la sortie d'un programme dans le terminal soit longue, avec bien trop de lignes. On peut utiliser un pipe dans "more", comme ceci:

```
$ objdump -D hw | more
```

Encore mieux, on peut garder cette sortie dans un fichier texte, que l'on pourrait ensuite voir tranquillement dans un éditeur de texte. Pour faire cela, nous allons utiliser la commande suivante :

```
$ objdump -D hw >> dump.txt
```

Plus rien n'apparait dans le terminal. Toute la sortie a été redirigée vers le fichier « dump.txt ». Remarquez que nous avons utilisé une extension pour indiquer la nature du contenu du fichier. Pour voir son contenu, utiliser votre éditeur préféré. Vous pouvez utiliser nano ou bien gedit.

```
$  gedit dump.txt
```

Il y a toujours beaucoup d'informations, mais au moins, il est possible de faire une recherche, pour trouver la fonction « main », que nous avions écrit dans notre source « hw.c ». Cela donne quelque chose comme cela :

```
0000000000400526 <main>:
  400526:    55                          push   %rbp
  400527:    48 89 e5                mov    %rsp,%rbp
  40052a:    48 83 ec 10             sub    $0x10,%rsp
  40052e:    89 7d fc                mov    %edi,-0x4(%rbp)
  400531:    48 89 75 f0             mov    %rsi,-0x10(%rbp)
  400535:    bf d4 05 40 00          mov    $0x4005d4,%edi
  40053a:    e8 c1 fe ff ff          callq  400400 <puts@plt>
```

```
40053f:    b8 00 00 00 00           mov    $0x0,%eax
400544:    c9                       leaveq
400545:    c3                       retq
400546:    66 2e 0f 1f 84 00 00     nopw   %cs:0x0(%rax,%rax,1)
40054d:    00 00 00
```

Il n'est pas demandé de lire et comprendre l'assembleur et encore moins le binaire, mais vous voyez, le compilateur ***gcc*** a fait la travail de transformer votre code source, un texte écrit en anglais, d'abord dans le langage assembleur, puis en binaire (des 0s et des 1s) que le processeur de votre machine peut exécuter. Il est temps de passer à l'exécution du programme et à la magie du debug.

1.2 La Magie du Debug

Le « debug » sera votre activité salvatrice pour toute votre carrière. Le « debug » permet de voir et de contrôler l'exécution d'un programme. Les développeurs avertis l'utilisent pour mettre au point leurs programmes, en corrigeant les « bugs », d'où le nom de l'activité. Historiquement, le nom de "bug" vient de l'anglais, un *bug* est un insecte, ceux qui se glissaient dans les premiers ordinateurs à lampe et qui faisaient griller quelques lampes et provoquaient des pannes.

Dans notre cas, nous allons utiliser le *debugger* comme un formidable outil d'apprentissage de la programmation. Mais commençons tout de suite. Tout d'abord, agrandissez la fenêtre de votre terminal, puis taper la commande :

```
$ gdb hw
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
…
Reading symbols from hw...done.
(gdb)
```

Le programme ***gdb*** est le debugger associé au compilateur ***gcc***. Lisez le message de gdb pour voir si il ne contient pas un message d'erreur. Si c'est le cas, lisez les détails car gdb vous donne la solution. Nous savons que vous ne lisez pas les messages d'erreur, il va falloir apprendre désormais. Comme conseillez, mettez en place le fichier .gdbinit sous votre dossier de login (votre « home »).

Maintenant que gdb fonctionne pleinement, utilisons le pour suivre l'exécution de notre programme, pas à pas. Tout d'abord, demandons à ***gdb*** de s'arrêter dans la fonction « main ».

```
(gdb) break main
Breakpoint 1 at 0x400535: file hw.c, line 5.
(gdb)
```

Super, ***gdb*** a ajouté un point d'arrêt (breakpoint) à la ligne 5 dans le fichier *hw.c*. Et si vous regardez votre source, c'est la première ligne de la fonction *main*, celle de l'appel à *printf*. Vous auriez pu ajouter le même point d'arrêt en donnant le fichier et la ligne :

```
(gdb) break hw.c:5
Breakpoint 1 at 0x400535: file hw.c, line 5.
(gdb)
```

Avec la commande "*break*", vous pouvez mettre autant de points d'arrêt que vous souhaitez, sur les lignes où vous désirez que l'exécution s'arrête. Mais comment se rappeler tous ses points d'arrêts. Pas besoin d'une mémoire infaillible. Si vous voulez voir la liste de vos *breakpoints*, vous pouvez faire ;

```
(gdb) info breakpoint
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400535 in main at hw.c:5
```

Notez que gdb donne un numéro à votre point d'arrêt (breakpoint), ici c'est le numéro 1, puisque c'est votre premier point d'arrêt. Donc si vous voulez enlever un point d'arrêt, vous pouvez simplement utiliser la commande suivante :

```
(gdb) delete 1
```

En donnant le numéro du point d'arrêt que vous souhaitez enlever. Bien sûr, nous n'allons pas enlever notre seul point d'arrêt. Lançons l'exécution maintenant :

```
(gdb) run
Starting program:  /home/.../HelloWorld/hw
Breakpoint 1, main (argc=1, argv=0x7fffffffdcc8) at hw.c:5
5       printf("Hello World! \n");
(gdb)
```

C'est trop cool, nous avons lancer une exécution et elle s'est arrêtée exactement où nous voulions. On peux le confirmer avec la commande « where »

```
(gdb) where
#0  main () at hw.c:5
```

Cela nous dit que nous sommes dans la fonction *main*. Peut-être que de voir une seule ligne du source n'est pas suffisant pour vous. Vous pouvez utiliser la commande « list » dont voici quelques formes utiles :

```
(gdb) list
(gdb) list main
(gdb) list hw.c:5
```

Super, l'exécution est suspendue et nous savons où. Nous allons maintenant voir comment contrôler cette exécution, vous pouvez utiliser quelques commandes :

```
(gdb) continue
Continuing.
Hello World!
[Inferior 1 (process 27114) exited normally]
(gdb)
```

Hum, que s'est-il passé ? Vous avez simplement demander à l'exécution de continuer et comme il n'y a plus de point d'arrêt (breakpoint) avant la fin, l'exécution s'est déroulée jusqu'à la fin. Mais ce n'est pas grave, on peut relancer l'exécution :

```
(gdb) run
Starting program:  /home/.../HelloWorld/hw
Breakpoint 1, main () at hw.c:5
5       printf("Hello World! \n");
(gdb)
```

L'exécution est de nouveau suspendue sur notre point d'arrêt. La commande *next* est celle que nous voulions.

```
(gdb) next
Hello World!
```

```
6       return 0;
(gdb)
```

Que s'est-il passé ? L'exécution a reprise, mais que pour la ligne courante. Donc le programme a exécuté l'appel de la fonction printf et s'est arrêté sur la ligne suivante, la ligne 6, celle du *return* de la fonction *main*. L'appel de cette fonction à imprimer à l'écran la chaîne de caractère passée en paramètre: "Hello World!". Appeler une fonction est comme demander un service, ici, le service était d'afficher une chaîne de caractères.

L'exécution de la fonction "*main*" touche à sa fin, c'est le sens de "*return*" qui indique la dernière ligne de la fonction "*main*". On peut donc demander à *gdb* de continuer et de finir l'exécution, puis nous quitterons le debugger :

```
(gdb) continue
Continuing.
[Inferior 1 (process 27770) exited normally]
(gdb) quit
$
```

Note Importante :

Le debugger « gdb » ne marchera bien que si on a compilé le source avec les informations de debug… comment faire cela ? Nous l'avions fait :

```
$ gcc -g -o hw hw.c
```

C'est l'option « **-g** » qui permet au compilateur de produire toutes les informations nécessaires au debugger dans le fichier « hw ». Essayons sans les informations de debug :

```
$ gcc -o hw hw.c
$ gdb hw
…
Reading symbols from hw...(no debugging symbols found)...done.
(gdb)
```

Ce n'est pas bon signe du tout… vous ne pourrez pas faire grand chose dans ce cas là. Il suffit de recompiler votre source pour avoir les informations de debug incluses dans l'exécutable :

```
$ gcc -g -o hw hw.c
```

Donc n'oubliez pas de compiler avec les informations pour le debugger.

Task 2 – Tout savoir sur les variables

Le debug est une façon très efficace pour apprendre les bases de la programmation. Le debugger permet de voir l'exécution d'un programme et donc de comprendre les concepts d'un langage de programmation. C'est ce que nous allons faire maintenant au travers de petits programmes. Ces programmes sont à regarder et à comprendre dans le détail.

2.1 Les variables sont essentielles à la programmation.

Une variable est un nom pour un bout de mémoire qui contient une donnée. Un nom (dit symbolique) est plus facile à utiliser qu'une adresse en mémoire pour le développeur. Mais une variable désigne toujours une zone mémoire, d'une taille qui dépend de la taille de la valeur et donc de son type :

int (valeur entière), exemples 24 or 14568

float (valeur réel), exemples 2.4f or 14.568f

char (un caractère d'un alphabet), exemples 'a', 'b', '&'

Le type d'une variable sert à contraindre les valeurs qu'elle peut contenir. Une variable de type « int » pourra contenir des valeurs entières alors qu'une variable de type « float » pourra contenir des valeurs réelles. Les valeurs entières peuvent avoir différentes tailles :

int8_t (8 bits ~one byte)

int16_t (16 bits ~ two bytes)

int32_t (32 bits ~ 4 bytes)

int64_t (64 bits ~8 bytes)

Rappelez vous que les chiffres entiers sont en fait encodé en interne en puissance de deux. Avec le chiffre donnant le nombre de bit (0 ou 1) utilisé pour encoder la valeur entière en puissance de 2. Usuellement, 8 bits sont appelés un octet (byte).

Les valeurs réelles (float) sont encodées sur 32 bits (4 octets, soit **4 bytes**).

Un caractère (char) est sur un octet (byte, 8bits).

Nota Bene : tout cela n'est pas super marrant, mais il faut l'apprendre et le mémoriser, ces notions apparaissent tout le temps dans l'activité de programmation. Désolé.

Regarder le programme suivant :

```
int8_t v1;
v1 = 12;
```

Nous avons la **définition** d'une variable ***v1***. Elle nomme un octet en mémoire (int8_t, donc sur 8 bits, soit un byte).

Puis nous avons l'**assignation** de la valeur 12 dans la variable v1. Cela veut dire que la valeur 12 est stockée dans la mémoire, à l'adresse nommée par la variable v1.

Une variable peut être *globale* ou *locale*. Lorsqu'elle est globale, elle est utilisable depuis n'importe quelle ligne de votre programme. Elle est globale lorsqu'elle définie en dehors de tout bloc lexical.

Lorsqu'elle est locale, elle est définie dans un bloc lexical délimité par des accolades :

```
{
   int8_t v2;
    v2 = 12 ;
 }
```

Une variable locale, comme sont nom l'indique, n'est utilisable que localement au sein du bloc lexical où elle a été définie.

Bien sûr, comme les pointeurs sont des variables, les mêmes règles s'appliquent. Nous allons choisir le nommage suivant pour les pointeurs sur des variables :

```
int8_t v1;
v1 = 12;
int8_t* v1_ptr = &v1;
*v1_ptr = *v1_ptr + 2;
```

On définie une variable ***v1*** nomme une zone mémoire, dans laquelle nous assignons la valeur 12. Puis nous définissons une autre variable ***v1_ptr***, qui contient une adresse, celle de la zone mémoire nommée par le nom ***v1***. Nous savons que cette zone mémoire contient une valeur entière, encodée sur 8 bits, soit un octet. C'est la signification de ***int8_t****. Le type int8_t nous dit que la valeur est encodée sur 8 bits, l'étoile nous dit que le variable est un pointeur, elle contient donc une adresse comme valeur.

A la dernière ligne, vous voyez comment utiliser un pointeur. A droite du signe égal, nous lisons la valeur qui se trouve dans la zone mémoire *pointée* par le pointeur ***v1_ptr***. C'est ici la zone mémoire nommée par la variable v1, donc c'est la valeur 12. A droite du signe égal, nous écrivons une valeur dans la zone mémoire *pointée* par le pointeur ***v1_ptr***. Nous écrivons donc la valeur 14 dans la variable v1. Les deux lignes suivantes sont équivalentes, elles font la même chose à l'exécution:

```
v1 = v1 + 2;
*v1_ptr = *v1_ptr + 2;
```

Rappel : Un pointeur est une variable qui nomme une zone mémoire qui contient l'adresse d'une autre zone. Une variable nomme un emplacement mémoire, en quelque sorte, une variable nomme un tiroir dans une commode. Un pointeur, c'est un tiroir dans lequel il y a un post-it qui indique le numéro d'un autre tiroir. Disons que le premier tiroir est un pointeur, vers vos clés de voiture. Et bien le premier tiroir ne contient pas vos clés, il contient le numéro d'un autre tiroir où se trouvent vos clés de voiture. Cela permet de ranger vos clés à différent endroit et de toujours les retrouver, via le pointeur. Sans pointeur, vous seriez obligé de ranger vos clés toujours au même endroit. Cela marche aussi...

Attention, le langage C n'impose aucune convention pour nommer les pointeurs, nous aurions pu écrire comme ceci :

```
int8_t v2;
v2 = 12;
int8_t* p = &v2;
*p = 12;
```

2.2 Il est temps de passer à la pratique, non ?

Tout d'abord, il faut aller dans le dossier "task2" de notre seconde tâche. Vous allez y trouver plusieurs fichiers qui vous montrent différents aspects des variables en C. Nous allons commencer par le fichier « var1.c » qu'il vous faut compiler. Pour cette fois, on va vous rappeler comment faire, mais vous devez mémoriser comment faire :

```
$ gcc -g -o var1 var1.c
```

Puis nous allons prendre le contrôle de l'exécution avec gdb :

```
$ gdb var1
(gdb) break main
(gdb) run
Breakpoint 1 at 0x40052e: file var1.c, line 41.
(gdb)
```

C'est tout bon, nous allons pouvoir apprendre plein de choses. Attention, il ne faut pas que lire et comprendre, il faut mémoriser ces concepts car vous allez devoir les maîtriser pour savoir programmer. N'hésitez pas à expérimenter avec les concepts que vous apprenez. Pour cela, l'informatique est extraordinaire, expérimenter est gratuit et sans danger. Pas besoin d'acheter quoi

que ce soit et on ne risque aucune catastrophe. C'est une bonne nouvelle, aucun risque de devenir Gaston Lagaffe (https://www.gastonlagaffe.com).

Nous allons commencer avec une variable et un pointeur sur la zone mémoire de cette variable :

```
int8_t v1;
int8_t* v1_ptr = &v1;
```

Vous pouvez demander à imprimer la valeur d'une variable :

```
(gdb) print v1
```

Ou bien encore comme cela (avec 'p' pour "print"):

```
(gdb) p v1
```

Vous pouvez demander la valeur en hexadécimale :

```
(gdb) p /x v1
```

Vous pouvez demander l'adresse que nomme une variable

```
(gdb) p &v1
```

Bien sûr,

```
(gdb) p *&v1
```

est équivalent à

```
(gdb) p v1
```

Vous pouvez demander la valeur d'une variable « pointeur » ou la valeur pointée

```
(gdb) p v1_ptr
(gdb) p *v1_ptr
```

Les trois lignes suivantes sont équivalents :

```
v1 = 12;
*&v1 = 12;
*v1_ptr = 12;
```

Ces trois lignes font la même chose, la variable v1 contiendra la valeur entière 12. Lorsque l'on dit la variable contient, c'est un abus de langage, ou plus exactement une simplification. La zone mémoire, nommée par le nom symbolique v1, contient la valeur 12. Ici, la zone mémoire ne fait que 1 octet, puisque le type de la variable est int8_t, ce qui indique que l'on ne veut utiliser que 8 bits pour encoder la valeur de la variable.

Nous pourrions vouloir manipuler des valeurs entières plus grande :

```
int32_t v2 ;
int32_t* v2_ptr;
v2 = 123456 ;
*v2_ ptr = 123456 ;
```

Ici la variable v2 nomme une zone mémoire qui fait 4 octets (4 bytes, 4*8 bits = 32 bits).

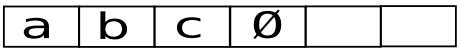
Pour information, la variable v2_ptr nomme une zone mémoire qui doit contenir une adresse. La taille d'une adresse dépend de votre ordinateur. Si c'est un processeur 32 bit, une adresse sera sur 32 bits (4 bytes). Si votre processeur est 64 bits, alors une adresse sera sur 64 bits (8 bytes).

Dans le source « var2.c » nous allons explorer l'usage des pointeurs pour la manipulation des chaînes de caractères.

```
char* s  = "abc";
```

Nous vous rappelons qu'une chaîne de caractères est une séquence en mémoire d'octets qui contiennent l'encodage ASCII des caractères. En effet, tout est chiffre en mémoire, les caractères, les couleurs, tout, absolument tout est encodé avec des 0s et des 1s en base 2.

En mémoire, la zone mémoire fait donc 4 octets, elle ressemble à cela :



Task-3 – Les fonctions en C

Les fonctions sont une construction très importante, qui va vous simplifier la vie et vous permettre de programmer plus vite et mieux. En fait, tout le code est organisé en fonctions. Le point d'entrée de l'exécution est d'ailleurs une fonction, la fonction « *main* ».

3.1 Le mieux est de découvrir les fonctions par la pratique

C'est ce que nous allons faire avec cette nouvelle tâche. Regardez le source task3/func1.c Compiler et exécuter ce code :

```
$ ./func1
string="Hello!"  length=6
```

Regardez le code. Il y a deux fonctions. La première est le point d'entrée, toujours appelé *main*. Il y aussi une fonction *string_length* dont le commentaire vous décrit ce que fait cette fonction : elle calcule la longueur d'une chaîne de caractère passée en paramètre.

Utilisez gdb pour comprendre ce qui se passe.

- N'oubliez pas de mettre un point d'arrêt dans la fonction ***main*** avant de lancer l'exécution.
- Suivre toute l'exécution de la fonction string_length (voir la remarque ci-dessous)
- Observez les valeurs des variables et comment elles évoluent.
- Observez le flot d'exécution, c'est à dire par où passe l'exécution.

Remarque Importante : Ici, vous allez faire de l'exécution pas-à-pas, avec la commande **next**, comme d'habitude. Mais vous allez voir que next ne vous montre pas l'exécution de la fonction *string_length*. Pour cela, lors de l'appel de la fonction *string_length*, il faut utiliser la commande ***step*** plutôt que la commande ***next***. Expérimentez avec ***next*** et ***step*** pour bien comprendre la différence. N'oubliez pas d'utiliser la commande ***where*** pour savoir où l'exécution est suspendue.

Regarder les variables, leur valeur, mais aussi leur adresse. Vous allez vite comprendre que vous pouvez avoir plusieurs variables, avec le même nom, mais qui ne nomment pas la même adresse, au sein de différents blocs lexicaux. En fait, les arguments passés lors de l'appel de la fonction sont des variables qui sont locales à l'appel de la fonction.

Notez que le corps d'une fonction définie un bloc lexical. Donc des variables peuvent être définies comme locales à une fonction, visible seulement dans son bloc lexical. Un bloc lexical peut aussi être associé à une boucle (while) ou un test (if-else). Mais que sont ces constructions « while » et « if-then-else » ? Elles permettent de contrôler le flot d'exécution, c'est à dire l'ordre dans lequel le processeur va exécuter les lignes de code dans votre fichier source. Une boucle ***while*** ressemble à ceci:

```
while (condition)  { ... }
```

La boucle va se répéter tant que la condition est vrai. C'est à dire que l'exécution va exécuter dans l'ordre la séquence de lignes de code défini dans le bloc lexical associé. On parle souvent de

statements, plutôt que de lignes de code.

La boucle est souvent associé avec le test conditionnel :

```
if (condition) { statements } else { statements }
```

Ici, la condition sert à choisir d'exécuter soit le premier bloc si elle est vrai soit le second si elle est fausse.

Pour exprimer une condition, vous pouvez utiliser les opérateurs suivant sur les valeurs entières et réelles :

- « x == y » pour tester que la valeur de x est égal à la valeur de y
- « x != y» pour tester que la valeur de x est différente de la valeur de y
- « x > y « pour tester que la valeur de x est supérieur à celle de y
- « x >= y « pour tester que la valeur de x est supérieur ou égale à celle de y
- « x < y « pour tester que la valeur de x est inférieur à celle de y
- « x <= y « pour tester que la valeur de x est inférieur ou égale à celle de y

Vous pouvez aussi utiliser des opérateurs de la logique booléenne :

- « (x == 10) || (x < 0) pour tester si la condition gauche **ou** droite sont vrais
- « (x != 0) && (x >= y) pour tester si la condition gauche **et** droite sont vrais

Vous pouvez rafraîchir vos souvenirs sur les tables de vérités avec Wikipedia.

3.2 Avec les fonctions, le concept de pile d'exécution apparaît.

La pile d'exécution ou encore la pile des appels en cours conserve à chaque instant l'emboîtement des appels de fonction. Ici l'exécution est simple, elle commence par la function *`main`*. Puis celle-ci appelle la fonction *`string_length`*, à la ligne 39.

```
37    void main(void) {
38         char *str = "Hello!";
39         int len = string_length(str);
40         printf("string=\"%s\" length=%d\n",str,len);
41         return;
42    }
```

C'est ce que l'on appelle un **site d'appel**. Du coup, lorsque l'exécution est dans la fonction ***`string_length`***, elle est là parce que l'exécution était dans la fonction ***`main`*** à la ligne 39 et à fait un appel à la fonction ***`string_length`***. Vous pouvez demander à gdb de vous montrez la pile d'exécution, ce qui vous montre comment l'exécution en est arrivée à exécuter la fonction courante. Pour cela, la commande de gdb est la suivante :

```
39          int len = string_length(str);
(gdb) s
string_length (s=0x400624 "Hello!") at func1.c:14
14          int len = 0;
```

```
(gdb) where
#0  string_length (s=0x400624 "Hello!") at func1.c:14
#1  0x000000000040057f in main () at func1.c:39
```

Vous voyez que gdb vous dit que vous êtes à la ligne 14 dans la fonction ***string_length***, à la ligne 14. Et pourquoi êtes vous là ? Parce que la fonction ***main*** a fait l'appel de la fonction ***string_length*** à la ligne 39.

Vous pouvez observer cette pile d'appel via gdb. Avec les commandes « up » et « down », vous pouvez demander à gdb de *voir* un appel de fonction ou un autre plus haut ou plus bas dans la pile. Cela introduit la notion d'appel de fonction courant pour les commandes gdb : l'exécution reste toujours suspendue au même endroit, par contre, les commandes s'appliqueront dans le contexte de l'appel de fonction courant. Ainsi, si l'exécution est suspendue dans la fonction « string_length », vous pouvez faire ceci :

```
 (gdb) where
#0  string_length (s=0x400624 "Hello!") at func1.c:14
#1  0x000000000040057f in main () at func1.c:39
(gdb) print s
$1 = 0x400624 "Hello!"
(gdb) up
#1  0x000000000040057f in main () at func1.c:37
(gdb) print str
$2 = 0x400624 "Hello!"
(gdb) print s
No symbol "s" in current context.
(gdb) down
#0  string_length (s=0x400624 "Hello!") at func1.c:14
14          int len = 0;
(gdb) print str
No symbol "str" in current context.
(gdb) print s
$3 = 0x400624 "Hello!"
(gdb)
```

Notez que gdb parle de symbole qui n'existe pas. En effet, il n'y a pas de variable nommée "s" dans la fonction ***main*** et il n'y a pas de variable "str" dans le fonction ***string_length***. Rappelez vous : **une variable est définie dans un bloc lexical et elle n'est visible que dans ce bloc lexical.**

Cela fait beaucoup de choses à comprendre et à retenir :

- le concept de fonction
- le concept de site d'appel d'une fonction
- le concept de pile d'appel
- les blocs lexicaux et les variables locales
- les arguments de fonction
- les expressions arithmétiques et booléennes

Tout cela est plus facile à comprendre via l'exécution pas-à-pas sous le contrôle du debugger, mais n'hésitez surtout pas à **demander de l'aide** à ceux qui savent déjà programmer, de **discuter entre vous de ces concepts**.

Conseil : pour mémoriser et mieux comprendre, ne faites pas que lire, faites quelques essais, écrivez du code, quelques lignes dans un source à vous avec votre fonction ***`main`***, pour valider votre compréhension...

3.2 Avec les fonctions, il faut discuter des arguments

Le passage d'arguments pour les fonctions est assez naturel pour toute personne ayant fait un peu de mathématique. Il est important de noter que les arguments sont passé par copie. Autrement dit, le pointeur "str" est copié lors de l'appel de la fonction ***`string_length`***, cela veut dire que sa valeur est copié dans le pointeur "s" définie comme argument de la fonction ***`string_length`***

Site d'appel :

```
int len = string_length(str) ;
```

Définition de la fonction :

```
int string_length(char *str) { … }
```

Observez sous debugger, vous verrez que la valeur de "s" et la valeur de "str" sont bien les mêmes. Pour cela, vous pourrez utiliser les commandes « up » et « down » pour vous ballader dans la pile d'appels. Pour vérifiez que vous avez bien deux variables et qu'il y a bien copie de la valeur. Pour cela, compilez et exécutez sous debugger le programme « task3/func2.c », arrêtez l'exécution dans la fonction « string_length » et balladez vous dans la pile d'appel avec les commandes « up » et « down », en affichant les variables « str ». Vous pouvez afficher les valeurs de ces deux variables avec la commande suivante :

```
(gdb) print str
```

Mais pour vous convaincre que vous avez bien deux variables, vous devez afficher l'adresse des zones mémoires nommée par ces deux variables :

```
(gdb) print &str
```

Vous allez voir que gdb vous indique que les deux adresses sont bien différentes, **donc les deux variables de même nom sont bien deux variables distinctes qui nomment deux zones mémoires différentes. Mais ces deux variables, qui sont des pointeurs, contiennent la même adresse ; c'est à dire qu'elles pointent la même chaîne de caractères.**

Maintenant, refaite l'exercice avec le programme « task3/func2.c » qui vous montre une autre façon d'écrire la fonction « string_length ». Notez que ce code modifie la valeur du pointeur « str » mais que cette modification reste totalement locale, la variable « arg » du la fonction « main » reste inchangée. Encore une fois, sous debugger, vous pouvez vérifier cela en regardant la variable « str » de la fonction « string_length » changer alors que celle de la fonction « main » ne change pas. C'est une autre façon de bien comprendre que nous avons donc bien deux pointeurs et qu'il y a bien un copie lors de l'appel de la fonction « string_length ».

Conseil : pour mieux mémoriser et mieux comprendre, ne faites pas que lire, faites quelques essais, écrivez du code, quelques lignes dans un « main », pour valider votre compréhension...

3.3 La vrai signature de la fonction « main »

Lorsque vous avez compris le source « task3/func1.c » et le source « task3/func2.c », il est temps de passer au code task3/func2.c qui vous montre la vrai signature de la fonction « main » et le passage d'arguments lors du lancement d'un programme :

```
$ ./func3
Printing the arguments:
  args[0]="./func3"       length=7
$ ./func3 Olivier Gruber
Printing the arguments:
  args[0]="./func2"       length=7
  args[1]="Olivier"       length=7
  args[2]="Gruber"  length=6
$ ./func3 "Olivier Gruber" ""
Printing the arguments:
  args[0]="./func2"       length=7
  args[1]="Olivier Gruber"      length=14
  args[2]=""        length=0
```

C'est parti, regardez le code dans le fichier source task3/func3.c pour comprendre comment le programme obtient les arguments passés lors du lancement du programme. N'oubliez pas d'expérimenter par vous-même...

3.4 Mise en pratique

C'est maintenant que la différence entre savoir et savoir-faire va vous apparaître pleinement. C'est à vous d'écrire du code maintenant, avec tous les concepts que nous avons vu jusqu'à présent. Vous allez mieux comprendre que savoir n'est pas savoir faire. La différence? La maîtrise par la pratique.

Nous allons coder quelques fonctions simples dans le source « task3/math.c ». Ces fonctions sont incluses dans le fichier « main.c », il suffit donc de compiler comme ceci :

```
$ gcc -g -o main main.c
```

Puis pour exécuter :

```
$ ./main
```

Notez que la fonction « main » est dans le source « task3/main.c ». C'est là que vous pouvez compléter nos quelques tests. Nous vous demandons donc de coder mais aussi de tester vos fonctions mathématique.

Task-4 – Les chaînes de caractères

Les chaînes de caractères (string en anglais) sont un outils particulièrement utilisé lors de la programmation d'applications. Il est donc très important de bien les comprendre. Dans cette tâche, on va vous demander d'implémenter quelques fonctions typiques sur les chaînes de caractères. Ces fonctions seront dans le fichier « task4/string.c ». Mais avant de vous lancer, il faut que nous discutions d'un point important.

Pour vous aider, nous avons écrit quelques tests qui vont vérifier que vos fonctions font ce qu'elles sont supposées faire. Pour écrire ces tests, nous avons eu recours à une nouvelle fonction de la librairie du langage de programmation C, **la fonction d'allocation mémoire « malloc »**. Comme vous le savez, toute variable se voit alloué une zone mémoire qu'elle nomme. Il est donc possible

d'avoir des pointeurs sur ces zones avec l'opérateur '&'.

```
int i;
int* p = &i;
```

Et comme vous le savez, la plupart de ces variables sont locales à un bloc lexical et ont donc une durée de vie limitée. Elles sont allouées lorsque l'exécution du bloc commence et désallouées lorsque l'exécution du bloc se termine.

Mais comment obtenir de la mémoire de manière plus durable? Comme par exemple pour contenir une chaîne de caractères comme "HelloWorld". Il nous faut une zone mémoire de 12 octets, 11 octets pour les caractères et un octet pour l'indicateur de fin de chaîne :

```
char* s = malloc(12);
```

Que veux dire cette ligne de code ? La fonction « malloc » alloue une zone mémoire et retourne l'adresse de cette zone. La taille de cette zone est passée en argument, soit 12 octet dans cet exemple. L'adresse devient la valeur du pointer « s ». Tout semble correct :

- nous avons une variable « s » qui nomme une zone mémoire
- la valeur dans cette zone mémoire est l'adresse d'une autre zone mémoire
- cette autre zone mémoire est valide puisque nous l'avons alloué via la fonction *malloc*

Comme la mémoire n'est pas infini, il est possible de libérer une zone mémoire alloué. Cela se fait par l'appel de la fonction « free », avec l'adresse de la zone à libérer. Cette adresse doit correspondre à une adresse retournée par un appel de la fonction « malloc ». Cela donne un paterne de programmation comme ceci :

```
char* s = malloc(12) ;
… // code qui utilise le pointeur s
free(s) ;
```

Avec ce nouveau concept d'allocation et de libération de mémoire, vous avez tout ce qu'il vous faut pour comprendre le code des fichiers sources « main.c » et «tests.c ». Ce code vous montre l'usage des chaînes de caractères au travers de tests simples. Faite bien attention de comprendre le code de ces tests et de l'usage des fonctions « malloc » et « free » :

- Comprenez pourquoi il faut allouer des zones mémoires.
- Pourquoi ces zones allouées ont-elles les tailles demandées ?
- Quand sont-elles libérées et pourquoi ?

Une fois que vous aurez compris le code dans « main.c », vous aurez compris l'usage des fonctions sur les chaînes de caractères dans le fichier « task4/string.c » et vous aurez donc la compréhension nécessaire pour écrire leur implémentation. Une fois que vous aurez fini ces implémentations, vous pourrez compiler et exécuter comme d'habitude :

```
$ gcc -g -o main main.c
$ ./main
```

Un fois que cela marche, vous pouvez vous assurer que nos tests passent :

```
$ gcc -g -o tests tests.c
$ gdb ./tests
```

L'idée avec ce fichier de tests (« tests.c » est de vous montrer comment écrire des tests pour que vous puissiez commencer à le faire pour vos propres programmes. Nous reparlerons souvent de tests cette année.

Conseil : nous vous conseillons :

- de lancer l'exécution sous le contrôle du debugger
- de placer un point d'arrêt dans la fonction « ensure » dans tests.c

Ainsi, si un test ne passe pas, l'exécution sera suspendu par gdb dans la fonction « ensure » et vous pourrez remonter la pile d'appel pour voir la raison pour laquelle le test n'est pas passé. Bien sûr, vous vous rappelez que les commandes pour cela dans gdb sont « up » et « down », n'est-ce pas ?

Have fun!

Task-5 – Les tableaux

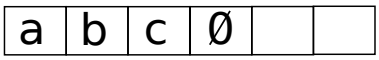
Les chaînes de caractères (string en anglais) que nous venons de discuter sont en fait un cas particulier des tableaux que nous allons discuter dans cette tâche.

5.1 Retour sur les chaînes de caractères

Une chaîne de caractères est en fait gérée dans un tableau de caractères, avec la convention que la fin de la chaîne de caractères soit indiqué par un octet dont la valeur zéro.

```
char *s1 = "abc";
int length = 6 ;
char *s2 = malloc(length) ;
string_copy(s2,s1) ;
```

Après la copie, le tableau alloué pour s2 s'étend donc sur 6 octets, mais la chaîne ne contient que 3 caractères et occupe 4 octets, comme ceci :



Notez que la zone mémoire peut être plus longue que la chaîne de caractères, c'est le cas dans notre exemple. Le tableau occupe 6 octets en mémoire, alors que la chaîne de caractères n'a que 3 caractères plus l'indicateur de fin de chaîne (0). La valeur des deux derniers octets nous est donc inconnu puisque nous n'avons rien écrit dedans.

5.2 Les tableaux

Pour les tableaux, **il faut toujours garder l'adresse du tableau et sa longueur**, car il n'y a aucun moyen de connaître la longueur d'un tableau. Il faut bien comprendre qu'un tableau en C n'est qu'un pointeur quelque part en mémoire sur une zone mémoire valide. Le type du pointeur va dicter comment la mémoire va être lu ou écrite. Un tableau n'a pas d'existence en tant que tel, toute zone mémoire valide, ou même invalide, peut être vu comme un tableau. Par exemple, le code ci-dessous est correct en C :

```
char *s = "abcd";
int8_t *pt1 = s ;
int32_t *pt2 = s ;
```

Le pointeur *pt1* permet de voir les caractères comme des entiers sur 8 bits, c'est à dire le code ASCII des caractères.

a' 0x61	*'b' 0x62*	*'c' 0x63*	*'d' 0x64*	*0x00*
**(pt1+0)*	**(pt1+1)*	**(pt1+2)*	**(pt1+3)*	**(pt1+4)*
pt1[0]	*pt1[1]*	*pt1[2]*	*pt1[3]*	*pt1[4]*

Pour le pointeur *pt2*, la justification est plus discutable. Le langage C le permets, mais attention, le langage C est très permissif et il autorise souvent de compiler du code qui n'a pas de sens. Ici, le tableau pointé par *pt2* aurait un seul élément valide et sa valeur serait celle de l'interprétation sur 32bit des valeurs des codes ASCII pour les lettres "abcd".

Pour les passionnés :

si vous prétendez connaître ou aimer le C, vous vous devez de comprendre cette explication ! La valeur sur 32bit va dépendre de votre machine et plus particulièrement du boutisme de votre processeur. Sur ma machine, la valeur serait :

1684234849 ou encore 0x64636261

Où nous reconnaissons les codes ASCII :

'a' 0x61 'b' 0x62 'c' 0x63 'd' 0x64

0x64636261 → 0x64 0x63 0x62 0x61 → 'd' 'c' 'b' 'a'

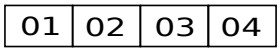
Cela nous dit que ma machine est petit-boutiste (little-endian), voir l'explication sur le boutisme, la relation au Voyage de Gulliver, et autres détails sur Wikipedia:

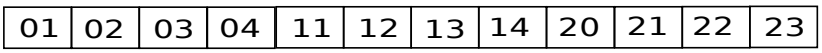
https://fr.wikipedia.org/wiki/Boutisme

Reprenons maintenant le cours normal de cet atelier...;-)

Voici deux exemples de tableaux dont les valeurs sont primitives. Le premier est un tableau d'entiers sur 8 bits (octets) d'une longueur de 4, il faut donc allouer 4 octets en mémoire :

```
int length = 4;
int8_t *array = malloc(length) ;
*(array+0) = 0x01;
*(array+1) = 0x02;
*(array+2) = 0x03;
*(array+3) = 0x04;
```





Le second est un tableau d'entier de 32bits (4 octets) et d'une longueur de 3, il faut donc allouer en mémoire 12 octets :

```
int length = 3;
int32_t *array = malloc(length*sizeof(int32_t)) ;
*(array+0) = 0x01020304;
*(array+1) = 0x11121314;
*(array+2) = 0x20212223;
```

La représentation graphique présuppose un processeur grand-boutiste (big-endian), mais ce n'est pas

l'important. **Ce qui est important est :**

- de comprendre c'est que chaque élément à une taille en mémoire
- que les éléments se suivent dans une zone mémoire continue
- que l'arithmétique sur les pointeurs prends en compte la taille des éléments

En effet, si la valeur du pointeur « array » est 0x10000 et son type est int32_t, alors sizeof(int8_t) vaut 4 et donc :

```
int32_t *array;
*(array+0)   0x10000 = 0x10000 + 0 * sizeof(int32_t)
*(array+1)   0x10004 = 0x10000 + 1 * sizeof(int32_t)
*(array+2)   0x10008 = 0x10000 + 2 * sizeof(int32_t)
```

Il est possible d'écrire l'accès à un élément d'un tableau avec une notation un peu plus naturelle :

```
int length = 3;
int32_t *array = malloc(length*sizeof(int32_t)) ;
array[0] = 0x01020304;
array[1] = 0x11121314;
array[2] = 0x20212223;
```

Mais il est important de bien comprendre que cela n'est qu'un sucre syntaxique, le compilateur va transformer votre notation [] en arithmétique de pointeurs. En C, il est impératif de comprendre que les tableaux sont en fait un pointeur sur une zone mémoire, rien de plus.

Mais comment savoir la taille d'un élément d'un tableau ? Et bien cela dépends de son type. La fonction « sizeof » vous donne la taille d'un type en octets. Cette fonction est importante pour pouvoir calculer la taille de la zone mémoire pour un tableau d'une longueur connue. D'où le code :

```
int length = 3;
int32_t *array = malloc(length*sizeof(int32_t)) ;
```

Mais attention, il vous faudra garder cette longueur. En effet, si vous n'avez qu'un pointeur sur un tableau, vous ne pouvez connaître sa longueur. Par exemple, si vous voulez écrire une fonction qui mets à zéro les éléments d'un tableau, il faut passer le pointeur et la longueur :

```
void zero(int32_t* array, int length) {
    for (int i=0; i<length; i++)
       array[i] = 0;
}
```

Notez aussi que si le pointeur ou bien la longueur est invalide, il est probable que l'exécution se fera sans provoquer d'erreur. Comme nous vous le disions, le C est très permissif à la compilation mais aussi à l'exécution. Pour ceux que cela intéresse, il y a l'outil *Valgrind* pour valider l'usage des pointeurs à l'exécution dans les programmes écrit en C. C'est un complément naturel à *gdb*.

5.3 Mise en pratique

Il est temps pour vous de mettre en pratique ce que vous venez d'apprendre sur les tableaux.
Nous vous conseillons d'expérimenter et de valider vos connaissances par vous même. Ecrivez de

petits bouts de code pour tester votre compréhension et votre capacité à manipuler des tableaux.

Puis, il sera temps d'implémenter les fonctions sur les tableaux dans le fichier "array.c". Regardez le code source de "main.c" pour comprendre l'usage de ces méthodes, cela va vous aider pour comprendre comment les implémenter. Puis, il sera temps de passer nos tests que vous trouverez dans le fichier « tests.c ».

HAPPY CODING!

Task-6 – Lire le clavier

Nous avons maintenant presque assez de compétence pour notre jeu du pendu. Il nous manque cependant encore quelque chose d'important : comment lire le clavier pour connaître les propositions du joueur à chaque tour de jeu?

On va donc faire un petit exercice pour voir comment faire un programme qui fait l'écho de ce que l'utilisateur tape au clavier. C'est simple, indépendant de la conception globale du jeu du pendu, et c'est facile de voir si notre programme marche.

Le programme que nous vous avons donné est complet, tel qu'un professionnel l'aurait écrit. Profiter de le lire, de réviser ou d'apprendre pleins de choses. Vous avez toutes les connaissances nécessaires pour comprendre ce programme. Seul le concept d'entrée et de sortie standard ne vous sont pas familiers.

Vous avez déjà utilisé la sortie standard sans le savoir puisque vous avez déjà imprimer dans la fenêtre du terminal dans lequel vous avec lancé l'exécution de vos programmes. Mais comme vous le savez, on peut lire et écrire dans un terminal. En effet, lancez des commandes dans vos terminaux sous Linux et ce sont des programmes qui prennent des arguments, affichent des messages, et demandent de saisir au clavier des informations. Chaque programme sous Linux se voit donc donné au lancement une sortie et une entrée standard.

Si la sortie standard sert à afficher du texte pour l'utilisateur, l'entrée standard permets à votre programme de lire ce que l'utilisateur saisi au clavier. Pour imprimer, vous utilisez la fonction « printf ». Pour lire, vous utilisez la fonction « read »:

```
ssize_t read(int fd, void *buffer, size_t count);
```

Elle permet de lire des octets depuis le clavier dans un buffer en mémoire. La zone en mémoire est indiqué par la valeur du pointeur « buffer ». Il vous faut donc une zone en mémoire valide avant d'appeler la fonction « read ». Le nombre maximum de caractère lu est indiqué par la valeur de la variable « count ». Si l'appel de la fonction réussi, il retourne le nombre de caractères lus dans la zone mémoire pointée par « buffer ».

Si l'appel échoue, il retourne la valeur -1. Pensez à regarder le code et ce qu'il fait si la fonction *read* retourne -1. **Il faut toujours gérer les cas d'erreurs dans vos programmes, toujours!**

Le code est dans le fichier source « task6/echo.c ». Comme d'habitude, voici comment compiler et lancer l'exécution sous le contrôle du debugger:

```
$ gcc -g -o echo echo.c
$ gdb ./echo
```

Soyez sûr de bien comprendre le code, pas juste la syntaxe mais surtout la logique du code. Les

points importants :

- Comprendre le concept d'entrée et de sortie standard.
- Comprendre la gestion des erreurs lors de la lecture au clavier (fonction read_char).
- Comprendre pourquoi une copie de la chaîne lue est nécessaire.
- Comprendre l'usage de malloc/free.
- Comprendre le comportement en cas d'erreur de lecture.

IMPORTANT : il est délicat de debugger un programme qui lit l'entrée standard.

Avant d'expliquer le problème et les solutions qui sont possibles, il vaut mieux que vous en fassiez l'expérience vous même : allez-y, prenez le programme « echo » sous debugger et commencer à faire son exécution pas-à-pas.

Il est plus que probable que, assez rapidement, vous allez penser que tout est bloqué et que le debugger ne marche plus. Cela va arriver dans la fonction « read_char ».

```
10    int read_char(char* ch) {
11        return read(STDIN, &ch, 1) ;
12    }
```

Vous étiez sur la ligne 11, sous gdb, et vous avez fait la commande « next », et là, plus rien ne se passe :

```
10    char read_char() {
(gdb) n
11          return read(STDIN, &ch, 1);
```

Effectivement, l'exécution est bloquée, mais cela est normale. En effet, le problème vient que le clavier est maintenant utilisé par l'application mais aussi par le debugger. En fait, la fonction *read* est bloquante, elle attends que l'utilisateur tape au clavier un caractère. Faite le, tapez 'H' par exemple. Cela ne débloque pas la situation, il vous faut aussi taper la touche ENTER. Pourquoi? Parce que le shell accumule les caractères saisis dans un buffer de line. C'est bien cela qui vous permet d'éditer la ligne que vous tapez tant que vous n'avez pas saisi la touche ENTER. Donc vous saisissez la ligne, par exemple "Hello", puis lorsque vous appuyez sur ENTER, le shell envoi les caractères saisis et la situation se débloque alors dans *gdb*.

Cette première approche peut vous permettre de comprendre le code et doit vous permettre de debugger dans le future vos codes qui lisent l'entrée standard. Mais si elle n'est pas suffisante, vous avec deux autres possibilités qui sont un peu plus complexes.

(a) Il est possible de demander à gdb de donner le contenu d'un fichier comme l'entrée standard du votre programme. Cela fait lors du lancement du programme :

```
$ gdb echo
(gdb) r < lines
```

A partir de là, vous debugger naturellement, puisque le clavier n'est plus partagé, il est uniquement utilisé pour interagir avec *gdb*. sachant que la lecture des caractères se fera depuis le fichier « lines » et non plus du clavier, le clavier ne sera utilisez que pour taper des commandes gdb.

Bien sûr, il faudra au préalable créer ce fichier avec ce que l'utilisateur aurait du taper, dans notre cas, c'est une suite de lignes de texte avec la dernière ligne vide. Le fichier doit être créé dans le dossier courant « task6 » ou se trouve l'exécutable « echo ». Un exemple de fichier pourrait être :

Bonjour !
Comment allez vous.
Super, merci.

(b) une autre possibilité existe mais elle est peut-être un peu difficile à comprendre pour ceux qui débutent avec Linux. Le debugger gdb peut lancer un programme pour le contrôler ou bien il peut prendre le contrôle d'un programme existant.

Il nous faut deux terminaux. L'un pour lancer notre programmer « echo ».

$ ./echo

Vous utiliserez ce terminal comme si le programme « echo » n'était pas sous le contrôle du debugger. En générale, on bloque le programme au début en demandant de frapper un touche lorsque le debugger est attaché et contrôle l'exécution.

Dans l'autre terminal, vous lancez gdb, en lui demandant de s'attacher au programme « echo » et d'en prendre le contrôle. Pour cela, il faut avoir les droits de root (super-utilisateur) sur la machine, autrement cela serait un énorme trou de sécurité si n'importe quel utilisateur pouvait prendre le contrôle de n'importe quel programme. Voici l'incantation magique :

```
$ pgrep echo
20219
$ sudo gdb --pid 20219
```

Ce qui peut s'écrire en une fois :

```
$ sudo gdb --pid $(pgrep echo)
```

Vous noterez qu'il faut donner le pid du processus auquel il doit s'attacher. Attention si vous avez plus d'un seul processus qui tourne sur votre machine et que plusieurs s'appellent « echo ». Pour plus d'infos, vous pouvez toujours demander la documentation de la commande « pgrep » comme ceci :

$ man pgrep

Task-7 – Le jeu du pendu

Vous avez les compétences pour une première version du jeu du pendu. Wikipedia: "*Le Pendu est un jeu consistant à trouver un mot en devinant quelles sont les lettres qui le composent. Le jeu se joue traditionnellement à deux, avec un papier et un crayon, selon un déroulement bien particulier"*.

Nous n'allons pas faire le coté graphique et nous allons faire une version que l'on peut jouer seul dans un terminal. Voici à quoi cela peut ressembler :

```
$ ./pendu
Vous avez 10 coups pour deviner le mot :
-----
? a
-----
? e
-----
? o
-oo--
? y
```

```
-oo-y
? W
Woo-y
? d
Woody
Vous avez gagné, voulez-vous rejouer ?
? non
Merci d'avoir joué, au revoir.
$
```

Comme vous le voyez, il n'y a pas de difficultés réelles dans l'interaction avec l'utilisateur, vous avez déjà vu les techniques dont vous avez besoin.

Maintenant, discutons de la conception de votre programme ; autrement dit, discutons de comment vous allez vous y prendre.

- Il vous faut un dictionnaire de mots.
- Il faut choisir aléatoirement le mot à deviner parmi ceux de notre dictionnaire.
- Le mot choisi sera affiché en mode caché, avec des '-' pour les lettres qui n'ont pas encore été devinées.
- Pour gagner, il faudra deviner le mot en un nombre maximal d'essai. Ce nombre doit au moins être égal à la longueur du mot à deviner, sinon la victoire est impossible.

Pour le dictionnaire, nous allons faire très simple dans cette première version, nous allons opter pour un tableau de chaîne de caractères. C'est une structure que vous connaissez. Le tableau sera initialisé au début du programme par des mots connus, en dure dans le source. Nous ferons la lecture du dictionnaire depuis un fichiers de mots dans la seconde version de notre jeu.

Pour choisir un mot au hasard, il faut pouvoir calculer le hasard... Heureusement, c'est déjà fait pour nous :

```
int rand(void);
```

Cette fonction retourne un nombre aléatoire à chaque appel. Ce nombre est compris entre 0 et RAND_MAX. Si vous souhaitez un nombre aléatoire entre 0 et N, il vous suffit de faire un modulo N, comme ceci :

```
int i = rand() % 10;
```

Qui vous donnera donc un nombre aléatoire dans l'intervalle [0,10[.

Par contre, il faut savoir qu'à chaque nouveau lancement de votre programme, la séquence des nombres aléatoires retournés sera la même. C'est très bien pour debugger, car votre programme se comporte à chaque fois de la même manière, mais pour le vrai jeu, ce n'est pas top. Donc la fonction *rand()* produit de l'aléatoire, mais pas totalement... Pour avoir des séquences différentes, il faut introduire une graine (*seed* en anglais), comme ceci :

```
void srand(unsigned int seed);
```

Mais comment avoir une graine aléatoire ? Car si nous donnons toujours la même graine, alors la séquence sera toujours la même. Une possibilité est d'utiliser le temps :

```
clock_t time = clock();
```

Comme ceci:

```
void main(void) {
  srand(clock());
}
```

Pour le reste... et bien c'est à vous de le concevoir, d'écrire le code, et de le faire marcher.

Task-8 – Lectures/Ecritures de fichier

Bravo, vous avez réussi à faire votre premier jeu. Si vous lisez ces lignes, c'est qu'il marche. Si vous le désirez, nous vous proposons de poursuivre votre apprentissage, mais **cette dernière tâche est optionnelle**, pour ceux qui ont avancé rapidement et qui souhaite approfondir le sujet.

Nous allons améliorer notre jeu du pendu. Dans notre première version, il fonctionne avec un dictionnaire de mots fixe. Pour changer cela, nous avons besoin lire le dictionnaire de mots depuis un fichier, comme cela, en changeant de fichier, on change le dictionnaire des mots.

Cela va nous permettre de voir comment lire et écrire dans un fichier, une connaissance qui vous sera utile. Pour découvrir comment créer un fichier et écrire dedans, nous allons écrire un programme pour créer le dictionnaire par programme. Comme nous allons écrire du texte dans le fichier, on pourra aussi facilement vérifier le contenu de ce fichier avec un éditeur classique.

Nous allons commencer par l'écriture du fichier dictionnaire, nous vous en avons donné un patron dans le fichier source « dico.c ». La fonction « main » est déjà implémenter pour vous, elle va d'abord faire la création d'un dictionnaire dans un fichier puis relire ce fichier pour afficher les mots du dictionnaire. Ce sont les fonctions :

```
void create_dictionary(char* filename);
void read_dictionary(char* filename);
```

8.1 Nous allons commencer par la création du dictionnary.

Au début de la fonction « create_dictionary », on vous a donné le code pour ouvrir un fichier en écriture et le créer si il n'existe pas. L'incantation est la suivante :

```
int fd = open(filename, O_RDWR | O_CREAT | O_TRUNC);
```

Si le fichier n'existe pas, il sera créé, c'est ce que veux dire *O_CREAT*. S'il existe, il sera ouvert en lecture et en écriture (*O_RDWR*). Dans tous les cas, le fichier sera tronqué à une longueur de zéro (fichier vide) (*O_TRUNC*).

La fonction « open » a la signature suivante :

```
int open(const char *pathname, int flags);
```

La documentation nous dit :

```
The return value of open() is a file descriptor, a small, non-negative
integer that is used in subsequent system calls (read(2), write(2),
 lseek(2), fcntl(2), etc.) to refer to the open file.

The argument flags must include one of the following access modes:
O_RDONLY, O_WRONLY, or O_RDWR.  These request opening the file read-only,
```

```
write-only, or read/write, respectively. O_CREAT  : if pathname does not
exist, create it as a regular file. O_TRUNC : if the file already exists
and access mode allows writing (i.e., is O_RDWR or O_WRONLY), it will be
truncated to length 0.
```

Puis nous avons une boucle pour lire les mots du dictionnaire et les écrire dans le fichier. Les fonctions correspondantes sont les suivantes :

```
char* read_line(int fd);
void write_line(int fd, char* line);
```

C'est à vous de les implémenter.

Pour la fonction `read_line`, il faudra s'inspirer de code de la tâche 6 qui faisait un écho de ce que saisissait l'utilisateur au clavier. Notez cependant que nous passons en argument le descripteur de fichier qui doit être lu.

Nota Bene : cette fonction ne devrait pas vous poser de problème particulier puisque vous avez déjà vu un code similaire. Si cela est encore difficile, il est probable que votre méthodologie de travail soit encore un peu trop scolaire. En effet, il ne s'agit pas seulement de comprendre, il s'agit d'apprendre un savoir-faire. Si vous n'arrivez pas à refaire, c'est que vous n'avez pas encore le savoir-faire. Pas de panique, il faut simplement revenir sur les concepts, votre compréhension, mais surtout sur votre pratique pour maîtriser les concepts.

Pour la fonction `write_line`, le problème est nouveau pour vous car vous ne connaissez pas la fonction qui permet d'écrire dans un fichier. Cette fonction est celle-ci :

```
ssize_t write(int fd, const void* buffer, size_t count);
```

Cette fonction est documentée sur le web :

http://man7.org/linux/man-pages/man2/write.2.html

Vous pouvez aussi l'avoir dans votre terminal avec la commande suivante :

$ man 2 write

En voici le résumé pertinent pour notre tâche d'écriture :

> Cette fonction écrit dans le fichier ***fd*** jusqu'à ***count*** octets depuis le buffer en mémoire pointé par l'argument ***buffer***. Si l'écriture réussi, l'appel de la fonction retourne le nombre d'octets effectivement écrit. Notez que le nombre d'octets écrit peut être inférieur à la valeur de l'argument *count*. Si une erreur se produit, l'appel de la fonction retourne la valeur -1.

Nota Bene : l'appel à write peux écrire moins d'octets que demandé. Ce point est très important, il faudra en conséquence écrire un code qui doit continuer d'essayer d'écrire tant que toute la ligne de texte ne sera pas écrite dans le fichier. Attention toute fois, si une erreur d'écriture survient, il ne sert à rien de continuer à essayer d'écrire ; il faut faire échouer votre programme en appelant la fonction `exit(-1)`.

Astuce Importante : vous avez remarqué qu'il faut passer des arguments au programme « dico ». Depuis le shell, ce n'est pas un problème, mais comment faire depuis gdb ? Pour pouvoir debugger, il faut savoir comment passer des paramêtres depuis gdb. Depuis un shell, il faut lancer le programme avec des arguments :

```
$ ./main -d dico.txt
```

Avec gdb, il faut le faire comme ceci :

```
$ gdb main
(gdb) run -d dico.txt
```

Une fois que votre programme tourne et fonctionne, il vous sera facile de vérifier le contenu du

fichier écrit. Voici une session vous montrant l'utilisation du programme (en gras ce que l'utilisateur saisi au clavier) :

```
$ ./main -d dico.txt
Please enter one word per line:
  (enter an empty line to quit)
? Droopy
? Tom
? Mignon
? Woody
?
Bye.
$ more dico.txt
Droopy
Tom
Mignon
Woody
$
```

8.2 Nous allons maintenant faire la lecture du dictionnaire depuis un fichier texte.

Cela se passe dans la fonction « **read_dictionary** ». Le format est donc d'un mot par ligne. Il se trouve que lire un fichier et lire l'entrée standard est la même chose. On peut utiliser la même fonction *read*, il faut seulement lui fournir un autre descripteur de fichier. Dans votre implémentation, vous avez codé une fonction « read_line » qui prends un descripteur de fichier en argument. Donc c'est tout bon, si on peut obtenir un descripteur de fichier. En fait, vous le savez déjà puisque vous avez déjà utilisé la fonction open pour un fichier en écriture. Vous vous en souveniez, n'est-ce pas ?

Voilà comment ouvrir un fichier en lecture seule :

```
char* filename = "dico.txt";
int flags = O_RDONLY;
int fd = open(filename, flags);
char ch;
read(fd,&ch,1);  // as seen before
```

Vous avez maintenant tout ce qu'il vous faut pour écrire la fonction « read_dictionary ». Vous savez ouvrir le fichier. Vous savez faire une boucle qui lit le fichier ligne à ligne. Il suffit d'imprimer ces lignes pour vérifier que ce que vous lisez correspond bien aux lignes du fichier.

8.3 Lire le dictionnaire dans une liste en mémoire.

Nous avons fini le programme dico qui permet de créer le dictionnaire et le relire. Cependant, nous savons le relire et l'afficher à l'écran, ce qui n'est pas suffisant pour notre jeu du pendu. Il nous faut construire le dictionnaire en mémoire.

Nous avions choisi pour cela un tableau de chaîne de caractères. Ce choix n'est plus idéal car vous ne savez pas combien de mots il y a dans le dictionnaire au moment de le lire. Vous ne pouvez donc pas allouer un tableau de la bonne taille. La liste est une structure de donnée plus adaptée.

Pour faire une liste, il faut utiliser les structures, un type un peu plus complexe que les types primitifs tels que « int » ou « float ». Imaginons que nous voulions un tableau de point, chaque point ayant ses trois coordonnées dans l'espace. Il est possible de définir le type point comme ceci :

```
struct point {
   int x;
   int y;
   int z;
};
struct point* array;
```

On voit qu'une structure est juste la composition de variable, chaque variable ayant son nom et son type. On peut alors utiliser la nouvelle structure comme le type d'une variable :

```
struct point p;
p.x = 2; p.y = 34; p.z = 0;
```

Comme p1 est une variable, on peut en demander l'adresse, comme pour toutes les variables :

```
struct point p;
struct point* pp = &p;

(*pp).x = 2; (*pp).y = 34; (*pp).z = 0;
```

ce qui s'écrit plus simplement comme ceci :

```
pp → x = 2; pp → y = 34; pp → z = 0;
```

Où la flèche → est la composition des caractères suivant : le tiret '- ' et le signe supérieur '>'.

Bien sûr, il est possible d'allouer en mémoire une structure :

```
struct point * p = malloc(sizeof(struct point) ;
p → x = 2; p->y = 34; p->z = 0;
```

Et enfin il est possible d'allouer un tableau de structures :

```
int length = 10;
struct point * p = malloc(length * sizeof(struct point) ;
p → x = 2; p->y= 34; p->z = 0;
p[0].x = 2; p[0].y = 34; p[0].z = 0;
*(p+0).x = 2; *(p+0).y = 34; *(p+0).z = 0;
```

Maintenant, revenons au concept de ce que nous voulions faire : une liste de mot. Cela suggère d'utiliser des structures comme celles-ci :

```
struct word {
  char* chars;        // la chaîne de caractères du mot
```

```
  struct word* next; // le lien vers l'élément suivant de la liste
};
struct dictionary {
  struct word* words;  // la tête de la liste des mots du dictionnaire
  int nwords;         // le nombre de mots dans le dictionnaire
};
```

Regardons un exemple simple. Bien sûr, l'idée est d'allouer une structure dictionnary pour représenter le dictionnaire (la liste de mots) et une structure word par mot dans le dictionnaire. Pour allouer une structure, c'est assez proche de ce que vous avez vu pour les tableaux :

```
struct dictionary *d = malloc(sizeof(struct dictionary)) ;
struct word *w1 = malloc(sizeof(struct word) ;
```

Pour créer un dictionnaire avec deux mots, voici un exemple :

```
struct dictionary *dico = malloc(sizeof(struct dictionary));
dico → nwords = 0;
dico → words = NULL;

struct word *w = malloc(sizeof(struct word);
w → chars = "Droopy";
w → next = NULL;
w → next = dico → words;
dico → words = w;

w = malloc(sizeof(struct word);
w → chars = "Woody";
w → next = NULL;
w → next = dico → words;
dico → words = w;
```

Vous avez maintenant une liste avec deux éléments. Pour bien comprendre, prenez une feuille de papier et faites l'exécution en représentant les zones mémoires alloués et les pointeurs entre ses zones. Vous allez vite comprendre ce qui se passe.

Il est important que vous compreniez bien la différence entre un tableau de structures et une liste de structures. Dans le cas du tableau, les structures se suivent dans la mémoire, il n'y a pas besoin d'une variable « next » pour trouver l'élément suivant, cela se fait par arithmétique de pointeur. Dans le cas d'une liste, les éléments ne sont pas consécutifs en mémoire, on ne peut les parcourir par calcul de pointeur, il faut la variable « next » pour passer d'un élément à son suivant.

Pour une explication plus complète ou bien si vous avez du mal à comprendre le code ci-dessus, vous pouvez lire cette page de Wikipedia :

https://fr.wikipedia.org/wiki/Liste_cha%C3%AEn%C3%A9e

A vous de jouer, il faut lire les lignes depuis le fichier, comme vous l'avez déjà fait, tout en les ajoutant à votre dictionnaire, c'est à dire à la liste des mots de ce dictionnaire. Une fois la lecture des mots terminés, vous devrez parcourir votre liste pour afficher tous les mots lus.

==========

GDB — Les Commandes Essentielles

Les commandes essentielles de ***gdb*** sont les suivantes, avec leur abréviation entre parenthèses :

run (r) pour lancer l'exécution du programme

il est possible de passer des arguments après la commande « run ».

vous pouvez relancer l'exécution à n'importe quel moment.

kill (k) pour tuer l'exécution du programme

quit (q) pour quitter ***gdb***.

break (br) pour positioner un point d'arrêt

br main (point d'arrêt dans une fonction)

br hw.c:5 (point d'arrêt à une ligne d'un fichier source)

delete (d) pour enlever un point d'arrêt (breakpoint)

info break (info br) pour lister les points d'arrêt

continue (c) pour continuer l'exécution

Ctrl-C pour demander à gdb de suspendre l'exécution et reprendre le contrôle dans la console

next (n) pour exécuter jusqu'a la ligne suivantes

step (s) pour exécuter jusqu'au prochain statement

finish (fin) pour finir l'exécution de la fonction courante

where pour avoir la call stack.

up pour voir la stack frame parente

down pour voir la stack frame fille

print (p) pour imprimer une variable

pour forcer un affichage en hexadécimale, utilise le format /x

print /x *variable*

display (disp) pour demander à gdb d'afficher la variable à chaque fois qu'il suspend l'exécution

display *variable-name*

==========

Le Shell — Les Commandes Essentielles

Un shell est un programme qui vous permet d'interagir avec votre ordinateur en ligne de commande.

Il y a les commandes relatives au système de fichiers et celles relatives aux processus, c'est à dire aux programmes qui tournent actuellement sur votre machine.

Comme vous le savez, le système de fichiers est organisé comme un arbre de dossiers et de fichiers. Un dossier ou un fichier est donc identifié par un chemin absolu depuis la racine de l'arbre ou bien un chemin relatif depuis un dossier de l'arbre. Voici quelques exemples de chemins absolus, notez qu'ils commencent tous par le caractère '/':

/home/john

/home/john/toto

En supposant que le dossier courant soit le dossier /home/john, voici quelques chemins relatifs avec leur équivalent en chemins absolus:

../vera	---	/home/vera
../vera/tata	---	/home/vera/tata
toto	---	/home/john/toto
./toto	---	/home/john/toto
toto/../../vera/tata/..	---	/home/vera

Quelques commandes relatives aux fichiers :

Les commandes relatives aux fichiers essentielles sont les suivantes, avec leur abréviation entre parenthèses :

- **cd** [*chemin*]
- **ls** [*chemin*]
- **mkdir -p** [*chemin*]
- **rmdir** *dossier*
- ***rm** [-r] chemin*
- **cat** *nom-de-fichier*
- **more** *nom-de-fichier*
- **file** *nom-de-fichier*
- **man** *nom-de-commande*

La commande '**cd**' est pour changer le dossier courant du shell.

La commande '**ls**' est pour lister le contenu du dossier dont le chemin est donné en argument à la commande, c'est à dire les dossiers et les fichiers qu'il contient. Le chemin peut-être absolu ou relatif. Sans argument, la commande 'ls' liste le contenu du dossier courant.

$ **ls** /home/john

$ **ls** ..

La commande "**mkdir**" permet de créer un dossier.

$ **mkdir** titi - crée un dossier "titi" dans le dossier courant.

$ **mkdir** -p titi/tata - crée un dossier "tata" dans le dossier "titi" dans le dossier courant.

La commande "**rmdir**" permet de détruire un dossier, mais attention, il faut que celui-ci soit vide.

La commande "rm" permet de détruire des fichiers ou des dossiers.

$ **rm** toto --- détruit le fichier toto dans le dossier courant.

$ **rm** /home/john/toto --- détruit le fichier /home/john/toto

$ **rm** -r /home/john --- détruit tout le sous-arbre /home/john

Attention: il n'y pas de "undo", vous ne pouvez pas défaire une destruction d'un fichier ou d'un dossier.

Les commandes "cat" et "more" permettent de voir le contenu d'un fichier sous forme de texte. Donc il vaut mieux afficher par ces commandes des fichiers qui contiennent du texte et non pas du code ou des données. Pour connaître le type d'un fichier, vous pouvez utiliser la commande "**file**".

La différence entre "cat" et "more" est toute simple, la commande "more" affiche le contenu du fichier page par page alors que la commande "cat" affiche tout le contenu du fichier d'un coup.

La dernière commande est la commande "man" qui est peut-être la plus importante, c'est la commande qui vous permets d'accéder au manuel des autres commandes, c'est à dire toute l'explication de ce que fait la commande et comment l'utiliser. Bien sûr, vous pouvez faire

$ **man** man

pour tout savoir comment utiliser la commande "man" afin de tout savoir sur les autres...

Quelques commandes relatives aux processus :

Un processus est le conteneur d'une exécution, une exécution est l'exécution par le processeur de votre machine du code contenu dans un fichier exécutable.

Tous les fichiers ne sont pas exécutable. Pour le savoir, vous pouvez utiliser la commande "file". Vous pouvez aussi regarder les droits sur le fichier. Tout fichier a des droits associés: lecture (read), écriture (write), et exécutable (execute). Pour voir les droits d'un fichier, vous pouvez utiliser la comme "ls" avec les options "-al"

```
ls -la /bin
total 13152
drwxr-xr-x  2 root root    4096 mars  31 06:53 .
drwxr-xr-x 28 root root    4096 sept.  8 06:23 ..
-rwxr-xr-x  1 root root 1037528 juil. 12  2019 bash
-rwxr-xr-x  1 root root  208480 févr. 15  2017 nano
...
```

Si la première lettre est un '**d**', il s'agit d'un dossier. Vous voyez les droits du dossier /bin sur la ligne qui se termine par "." et vous voyez les droits du dossier parent sur la ligne qui se termine par "..". Rappelez vous que dans un chemin, le dossier courant est nommé par "." et le dossier parent par "..".

Si la première lettre est un tiret '-', alors il s'agit d'un fichier, comme pour les lignes donnant les informations pour les fichiers /bin/nano et /bin/bash.

Après la première lettre ('d' ou '-'), les 9 lettres suivantes vous indiquent les droits du fichier ou dossier. Un fichier ou un dossier a un propriétaire et il appartient à un groupe. Pour connaître le propriétaire et le groupe, regarder les deux noms après les droits. Le premier est le propriétaire, le second est le groupe. Ici, tous les dossiers et tous les fichiers appartiennent au propriétaire "root" et

au groupe "root". Pour un autre dossier, le contenu serait différent:

```
$ ls -al /home
total 20
drwxr-xr-x  5 root    root    4096 mai   31  2018 .
drwxr-xr-x 28 root    root    4096 sept.  8 06:23 ..
drwxr-xr-x  2 grubero boye    4096 mai   31  2018 boye
drwxr-xr-x 23 boyer   users   4096 juil. 15  2019 boyer
drwxr-xr-x 59 ogruber users 4096 août  25 18:01 ogruber
```

Ici on voit que certains dossiers appartiennent au login "root" et au groupe "root". On voit d'autres dossiers appartenir au login "boyer" et "ogruber", avec le groupe "users". Le plus souvent, les fichiers que vous manipulerez vous appartiendront.

Revenons à notre exemple précédent et regardons la ligne pour le fichier "nano":

```
-rwxr-xr-x  1 root root  208480 févr. 15  2017 nano
```

On voit qu'il appartient à "root" et que les droits sont "-rwxr-xr-x". Le premier tiret nous dit que c'est un fichier, par un dossier. Les 3 lettres suivantes nous disent que le propriétaire "root" a tous les droits "rwx". Puis les 3 lettres suivantes nous donnent les droits du groupe puis les 3 suivantes nous donnent les droits des autres, ceux qui ne sont ni le propriétaire ni n'appartiennent au groupe.

Ici, vous n'êtes pas le propriétaire et vous n'appartenez pas au groupe, donc les trois dernières lettres vous concernent: "r-x". Vous pouvez donc lire ce fichier et vous pouvez l'exécuter.

Comment lancer une exécution? Regardez la liste ci-dessus, vous voyez le fichier "nano", il est bien exécutable puisqu'il y a le droit en exécution. La commande "file" vous dirait ceci:

```
$ file /bin/nano
```

/bin/nano: **ELF 64-bit** LSB executable, **x86-64**, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=e87c5ab33de1e9e225244efd8e2a03f55892c98f, stripped

Le format **ELF** est le format utilisé par Linux pour encoder du code dans un fichier exécutable. Le code est pour un processeur "**x86-64**", ce qui corresponds à un processeur Intel classique 64-bit des PCs et laptops.

Pour lancer l'exécution, il suffit de taper le chemin vers l'exécutable, un chemin relatif ou bien absolu:

```
$ nano
$ /bin/nano
```

Lorsque vous ne donnez que le nom du fichier exécutable et non le chemin (relatif ou absolu) pour y accéder, le shell utilise un ensemble de dossiers connu où il va chercher un fichier exécutable avec le nom que vous avez donné:

```
$ nano
```

Le shell va chercher un fichier "nano" et va le trouver dans le dossier /bin.

Pour connaître l'ensemble de ces dossiers, vous pouvez regarder la variable d'environnement PATH, comme ceci:

```
$ echo $PATH
```

/home/ogruber/bin:/usr/sbin:/usr/bin:/sbin:/**bin**:/snap/bin:/opt/git-lfs

Vous y voyez le dossier "/bin" et cela explique pourquoi le shell peut trouver "nano" et en lancer l'exécution.

Si vous voulez lancer un programme qui se trouve dans dossier qui ne se trouve pas sur le "PATH", et bien il vous suffit de donner le chemin relatif ou absolu. Souvent, le dossier courant n'est pas dans le "PATH" et pour lancer un programme dans le dossier courant, il faut donner le chemin relatif:

$ ls toto

-rwxrwxrwx 59 **john users** 4096 août 25 18:01 toto

$ ./toto

Pour arrêter l'exécution, vous pouvez taper "Ctrl-c", c'est à dire la touche "Ctrl" et la touche "c", en même temps.