Le prototype de la fonction main()

1. Introduction

Pour lancer une commande dans le Terminal, vous tapez le nom de la commande suivi de ses arguments – ou paramètres. Lorsqu'on lance un programme écrit en C depuis le Terminal, on peut de la même manière fournir des arguments au programme, par exemple en tapant :

```
$ ./monProgramme coucou 233 -c
```

Mais comment récupérer ces arguments dans le programme C lui même ?

2. Paramètres et type de retour de la fonction main()

La fonction `main()` d'un programme C est la première fonction exécutée lors du lancement du programme. Le langage C spécifie un prototype complet pour la fonction `main()` (paramètres et un type de retour) qui, justement, permet au programme de récupérer ses paramètres.

Le prototype complet de la fonction `main()` d'un programme C est :

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

où :

- `int argc` est le nombre de paramètres effectivement passé au programme. Ce nombre est toujours > 1 car le premier paramètre est toujours le nom de l'exécutable.
- `char * argv []` est un tableau de chaine de caractères contenant les paramètres effectivement passés au programme.

Le premier élément de ce tableau (`argv[0]`) contient toujours le nom du programme (ou plus précisément le chemin utilisé pour accéder au fichier exécutable). C'est bien utile car on ne sait pas, dans le code C, comment le fichier exécutable s'appelle : ce fichier exécutable a pu être renommé par exemple après avoir été compilé !
Le $i^{ème}$ élément du tableau, `argv[i]`, est une chaine qui contient le $i^{ème}$ paramètre passé au programme.

Par ailleurs, conformément au prototype de la fonction `main()`, un programme C doit retourner (`return`) lorsqu'il se termine « code de retour » entier. Ce code de retour permet au programme de faire savoir au système si une erreur a été détectée.

On retiendra que, par convention :

- le `main()` doit retourner 0 (ou la constante `EXIT_SUCCESS`, définie dans `stdlib.h`, qui vaut 0) si tout s'est bien passé
- le `main()` doit retourner un nombre **non nul** en cas d'erreur.

Le choix d'une valeur quelconque non nulle en cas d'erreur permet à votre programme de savoir quel est le type d'erreur, en associant une valeur entière à une erreur donnée. La constante `EXIT_FAILURE`, définie dans `stdlib.h`, non nulle, permet de retourner un code d'erreur par défaut si on a pas besoin de valeurs spéciales.

Toutefois, les codes de retour compris entre 126 et 255 sont réservés par le système pour des erreurs spéciales. Par exemple, un code de retour valant 139 signifie qu'une « *segmentation fault* » (erreur d'accès mémoire) a eu lieu dans le programme et que le programme a planté.

3. Exemple 1

Le programme simple suivant affiche les paramètres qui lui sont passés depuis le Terminal :

```
//prg.c
int main ( int argc , char * argv[] ) {
   int i;
   // affichage des arguments
   printf("Nombre d'arguments passes au programme : %d\n",
argc);
   for(i = 0 ; i< argc ; i ++) {
      printf("   argv[%d] : '%s'\n", i, argv[i]);
   }
   return 0 ;
}
```

Une fois ce programme compilé, si l'utilisateur le lance depuis le Terminal en tapant :

```
$ ./prg.exe test.txt 2
```

alors le programme affichera dans le Terminal :

```
Nombre d'arguments passes au programme : 3
   argv[0] : './prg.exe'
   argv[1] : 'test.txt'
   argv[2] : '2'
```

puis retournera 0 (tout s'est bien passé).

4. La fonction exit() du C

En C, la fonction exit(int status) permet de terminer le programme en retournant le code erreur status. Voir man exit –S3

Ainsi, ou que l'on soit dans la code, on peut utiliser exit() pour arrêter le programme. Par exemple :

```
long fact( long n) {
 if( n < 0 ) {
      exit( EXIT_FAILURE) ;
 }
 // …
}
```

5. Détecter le code de retour d'un programme depuis le Terminal

Pour information, puisqu'un programme C retourne un code erreur, le Terminal est bien sur armé pour traiter ce code erreur. Il suffit de savoir deux choses :

- La commande `echo` du Terminal permet d'afficher une chaine de caractère. Par exemple `echo toto` affiche toto.
- A tout instant, la variable `$?` contient toujours la valeur de retour du dernier programme exécuté

Ainsi, si on tente de supprimer un fichier toto qui n'existe pas avec rm, rm affiche un message d'erreur et retourne bien un code erreur non nul :

```
$ rm toto
rm: toto: No such file or directory
$ echo $?
1
```

6. *Parsing* des paramètres et exemple 2

Un programme commence par analyser (ou « *parser* ») les paramètres pour 1/ vérifier qu'ils respectent bien ce qui est attendu (qu'on a passé un entier si on attend un entier par exemple) et 2/ initialiser ses variables. L'exemple suivant propose une structure pour se faire, qui est très courante et que vous pourrez réutiliser.

Supposons qu'on veuille écrire un programme qui affiche plusieurs fois dans le Terminal le contenu d'un fichier, texte ou binaire dont le chemin est passé en paramètre.

Le format qu'on retient pour les paramètres à passer au programme est donc :

- Premier paramètre : option –txt ou –bin
- Second paramètre : nom du fichier, chaine de caractère
- Troisième paramètre : entier, optionnel

Le programme suivant analyse ces paramètres :

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

// Affiche l'aide du programme
void printHelp(char * prgName) {
    printf("%s\n", prgName);
    printf("  Displays the content of a txt or binary file\n");
    printf("  Usage:\n");
    printf("     %s <opt> <fileName> (<nb>)\n", prgName);
    printf("  where:\n");
    printf("   <opt> is either '-txt' or '-bin' (txt or bin file).\n");
    printf("     <fileName> is the name of the file to display.\n");
    printf("     <nb> is the number of display to achieve.\n
    printf("                    Optinal, default 1.\n");
}

// Affiche l'erreur errStr, puis l'aide du programme
void printError(char * prgName, char * errStr) {
    printf("Error : '%s'\n", errStr);
    printHelp(prgName);
}



int main(int argc, char *argv[])
{
    int mode ; // 0 => txt ; 1 => binary
    char *fileName = NULL;
    FILE * filePtr = NULL;
    int nbDisplay = 1;
    int i;

    ///////
    // parsing des arguments
    ///////
    if(argc < 3 || argc > 4) {
        printError( argv[0], "number of args invalid" );
        return EXIT_FAILURE;
    }

    // affichage aide
    if( strcmp(argv[1], "-h") == 0 || strcmp(argv[1], "-help") == 0) {
        printHelp(argv[0]);
        return EXIT_SUCCESS;
    }

    // type de fichier
    if( strcmp(argv[1], "-txt") == 0) {
        mode = 0;
    } else if( strcmp(argv[1], "-bin") == 0) {
        mode = 1;
    } else {
        printError( argv[0], "-txt or -bin required" );
        exit(EXIT_FAILURE); // equivalent a return EXIT_FAILURE;
    }


    // nom du fichier
    fileName = argv[2];
```

```
    // entier
    if(argc == 4) {
        // on attend un entier !
        if( sscanf(argv[3], "%d", &nbDisplay) != 1) {
              printError( argv[0], "Integer awaited" );
              exit(EXIT_FAILURE); // equivalent a return EXIT_FAILURE;
        }
    }

    ///////
    // c'est parti !
    ///////

    switch(mode) {
        case 0:
              filePtr = fopen (fileName, "rt");
              break;
        case 1:
              filePtr = fopen (fileName, "rt");
              break;
    }

    if(filePtr == NULL) {
        printError( argv[0], "Cannot open file" );
        exit(EXIT_FAILURE); // equivalent a return EXIT_FAILURE;
    }

    for(i = 0 ; i < nbDisplay ; i ++) {
        switch(mode) {
              case 0:
                    // displayTxtFile(filePtr);
                    // fct supposee existante
                    break;
              case 1:
                    // displayBinFile(filePtr);
                    // fct supposee existante
                    break;
        }
    }
    fclose(filePtr);

    return EXIT_SUCCESS;
}
```

Pour information, la fonction `getopt()` (voir `man getopt –S3`) définie dans `unistd.h` peut aider à l'analyse des paramètres d'un programme.