Chapitre 1

TP1 : Prise en main des outils et configuration

Avant de s'attaquer au coeur du problème, nous allons tester et mettre en place des fonctionnalités très pratiques en langage C (la gestion des paramètres de la ligne de commande) et en compilation (l'utilitaire **make**, la génération de documentation). Cela servira aussi de rappel pour les structures de données standards du C.

1.1 Les paramètres de la ligne de commande en C

1.1.1 Quelques rappels de C

Remarque 1.1
Lecture des paramètres de la ligne de commande. La fonction `main` reçoit 2 paramètres : 1 entier `int argc` et un tableau de pointeurs `char *argv[]`. L'entier représente le nombre de paramètres qui ont été écrits à la suite de l'appel au fichier exécutable (dans une fenêtre de terminal par exemple, ou bien en précisant *Set program's arguments...* dans le menu *Project* de CodeBlocks).

Exercice 1.1
Ecrire un programme C (`tp1_a.c`) qui affiche les paramètres qu'il reçoit sur la ligne de commande.

Par exemple, lorsque l'on tape dans un terminal :
`./tp1_a zutyr 788 127.0.0.1:9000`
où `tp1_a` correspond au nom de l'exécutable, on veut obtenir l'affichage :

```
<zutyr><788><127.0.0.1:9000>
```

(chaque paramètre est écrit entre <> qu'il soit chaîne de caractères, nombre ou adresse IP).

À faire ☐ Conserver cet exemple sous le nom `tp1_a.c`

1.1.2 Des arguments bien traités

Pour transmettre des informations à un programme, il suffit donc de les écrire à la suite de l'appel au programme et de les récupérer sous forme de chaînes de caractères comme dans l'exercice précédent. Bien sûr, on pourrait utiliser une interface plus interactive : poser des questions à l'utilisateur ou lui proposer une interface plus conviviale (des fenêtres, des boutons, des cases à cocher...). Cependant, la répétition d'une même

configuration (lancer le programme avec les mêmes paramètres) peut devenir dans ce cas très pénible et le passage des paramètres par la ligne de commande permet d'automatiser le lancement d'une configuration. L'idéal serait qu'un programme accepte les deux modes de configuration : l'interactif convivial et l'automatisable. Nous allons nous concentrer sur cette partie automatisation car cela sera utile lorsque le bot jouera de nombreuses parties en utilisant des paramètres identiques.

Une technique simple est de différencier les paramètres passés en argument sur la ligne de commande en fonction de l'ordre dans lequel il sont écrits. Tel que présenté dans l'exercice 1.1, seul l'ordre dans lequel on écrit les paramètres permet d'en connaître le rôle. Dans l'exemple, le premier paramètre peut signifier un nom d'utilisateur, le deuxième une valeur de *timeout* et le dernier une adresse IP. Si on ne respecte pas cet ordre, le programme ne récupèrera pas les bonnes valeurs. En effet, utiliser l'ordre d'écriture comporte des risques d'erreur et ce n'est pas facile à mémoriser. On préfère souvent utiliser un code sous la forme tiret+lettre permettant d'annoncer le paramètre qui va suivre.

L'exercice suivant va permettre de mettre en place la lecture, l'affichage des arguments de la ligne de commande afin de mettre à jour des paramètres du programme en autorisant 3 types : entier, réel ou chaîne de caractères. Par rapport au programme précédent, on tappera la commande :

```
./tp1_b -a zutyr -t 788 -s 127.0.0.1:9000
```

Dans ce cas, l'ordre n'a plus d'importance et on aurait pu tout aussi bien écrire :

```
./tp1_b -s 127.0.0.1:9000 -t 788 -a zutyr
```

Remarque 1.2

Déclaration de types en C. L'exercice suivant permet de réviser `typedef`, `union`, `enum` et `struct`...

Remarque 1.3

Dans l'exercice suivant, les commentaires sont écrits au format doxygen. Il faudra utiliser ce formalisme tout au long de cet enseignement.

❑ Exercice 1.2

L'objectif est d'écrire un programme C qui affiche les paramètres qu'il reçoit sur la ligne de commande.

Par exemple, lorsque l'on tappe :

```
./tp1_b -a zutyr -t 788 -s 127.0.0.1:9000
```

on veut obtenir l'affichage :

```
Valeurs par defaut :
-s serveur (chaine) [??]
-a appli (chaine) []
-t tours (entier) [200]
Valeurs des parametres :
-s serveur (chaine) [127.0.0.1:9000]
-a appli (chaine) [zutyr]
-t tours (entier) [788]
```

On fournit les types de données à utiliser, la fonction `main` et les prototypes des fonctions à écrire :

```c
/**
 * \file tp1_b.c
 * \brief 2ieme etape du TP IL
 * \author NM
 * \version 0.1
 * \date 17/11/2014
 *
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

/**
 * \enum TParamType
 * \brief Constantes pour le type des parametres de la ligne de commande
 */
typedef enum {
    PTentier, /**< un nombre entier */
    PTreel, /**< un nombre reel */
    PTchaine /**< une chaine de caracteres */
  } TParamType;

/** constante chaine de caracteres pour l'affichage des types */
char * ParamTypeChaine[] = {"entier","reel","chaine"};

/**
 * \union TParamValue
 * \brief union permettant de  manipuler un entier/reel/chaine
 */
typedef union {
    int entier;
    float reel;
    const char * chaine;
  } TParamValue;

/**
 * \struct TParamDef
 * \brief represente un parametre de la ligne de commande (nom, type,valeur...)
 */
typedef struct {
    char * nom; /**< nom du parametre */
    TParamType type; /**< type (entier,reel,chaine) */
    char lettre; /**< lettre a utiliser sur la ligne de commande */
    TParamValue valeur; /**< valeur a affecter au parametre */
```

```
47    } TParamDef;
48
49  /**
50   * \fn char * ValeurParamToString(TParamDef * _tabParam, const
        int _index)
51   * \brief fonction qui transforme la valeur d'un parametre en
        chaine de caractere
52   *
53   * \param[in] _tabParam tableau des parametres de la ligne de
        commande
54   * \param[in] _index indice du parametre a considerer dans le
        tableau
55   * \return une nouvelle chaine (qu'il faudra libÃ©rer par la
        suite)
56  */
57  char * ValeurParamToString(TParamDef * _tabParam, const int
      _index) {
58  /***** A ECRIRE *****/
59  }
60
61  /**
62   * \fn PrintParam(TParamDef * _tabParam, const int _nbParam)
63   * \brief fonction qui affiche a l'ecran les parametre, leur
        type et leur valeur
64   *
65   * \param[in] _tabParam tableau des parametres de la ligne de
        commande
66   * \param[in] _nbParam taille du tableau
67   * \return neant
68  */
69  void PrintParam(TParamDef * _tabParam, const int _nbParam) {
70  /***** A ECRIRE *****/
71  }
72
73  /**
74   * \fn int ReadParamFromCommandLine(TParamDef * _tabParam,
        const int _nbParam, const int _argc, const char * _argv[])
75   * \brief fonction qui analyse la ligne de commande pour en
        extraire des valeurs pour les parametres
76   *
77   * \param[out] _tabParam tableau des parametres de la ligne de
        commande
78   * \param[in] _nbParam taille du tableau
79   * \param[in] _argc nombre d'arguments passes sur la ligne de
        commande
80   * \param[in] _argv tableau qui contient les chaines de
        caracteres passees en arguments du programme
81   * \return >=0 : nombre de parametres mis a jour, <0 : erreur
82  */
83  int ReadParamFromCommandLine(TParamDef * _tabParam, const int
```

```
      _nbParam, const int _argc, const char * _argv[]) {
84  /***** A ECRIRE *****/
85  }
86
87  /**
88   * \fn int main (const int _argc, const char * _argv[])
89   * \brief fonction principale
90   *
91   * \param[in] _argc : nombre d'arguments passes sur la ligne de
         commande
92   * \param[in] _argv : tableau qui contient les chaines de
       caracteres passes en arguments du programme
93   * \return 0 si terminaison normale
94   */
95  int main (const int _argc, const char * _argv[]) {
96    /* declaration des paramtres avec leur type+valeurs par
         defaut */
97    TParamDef tab_param[] = {
98      {"serveur",PTchaine,'s',.valeur.chaine="??"},
99      {"appli",PTchaine,'a',.valeur.chaine=""},
100     {"tours",PTentier,'t',.valeur.entier=200}};
101   int nb_param = 3;
102   int result_arg;
103
104   /* affichage des formats de parametre */
105   printf("Valeurs par defaut :\n");
106   PrintParam(tab_param,nb_param);
107   /* analyse de la ligne de commande */
108   result_arg = ReadParamFromCommandLine(tab_param,nb_param,
         _argc,_argv);
109   /* affichage des nouveaux parametres */
110   printf("Valeurs des parametres :\n");
111   PrintParam(tab_param,nb_param);
112
113
114   return 0;
115 }
```

Remarque 1.4

Nous utiliserons une norme de nommage (valable pour toute la suite du projet) :

- les types de données sont écrits avec une majuscule à la première lettre de chaque mot et commence par la lettre T. Par exemple : `TParamDef`
- les champs des types `enum` sont précédés des majuscules du nom de type (car ce sont des constantes que l'on ne voudra pas confondre : un entier des paramètres n'est pas identique à un entier qu'on lira dans un fichier)
- les variables passées en paramètre de fonction commencent par le symbole souligné. Par exemple `_tabParam` et sont écrit avec une majuscule pour la première lettre de chaque mot, sauf la première (cela permet de différencier de noms de fonction).
- les variables locales sont écrites sans majuscules et avec des `_`. Par exemple : `int`

`result_arg;`

— les constantes déclarées par `#define` sont écrites toute en majuscules.

De plus, il faudra veiller à une mise en page claire et cohérente, cela peut-être atteint par un indentation précise et régulière...

Tout cela améliore la lisibilité de votre programme.

Remarque 1.5

Il faudra être très vigilant sur la réservation mémoire, en particulier pour les chaînes de caractères :

— penser à regarder la fonction `asprintf(...)`. On peut avoir de l'aide en tapant `man asprintf` dans une fenêtre de terminal...

☞**À faire** ☐ Compléter le code source du fichier `tp1_b_a_completer.c` (renommez le `tp1_b.c`).

Remarque 1.6

Doxygen est un outil qui analyse le code et extrait les commentaires d'un fichier source pour construire une documentation.

1. Préparation : dans un premier temps, il faut générer un fichier de configuration pour le projet en cours par la commande :
   `doxygen -g`
   Dans le fichier généré (`Doxyfile`), il faut indiquer :
   — le titre du projet : `PROJECT_NAME = "TP1_b"`
   — le dossier dans lequel installer la documentation (sinon c'est en vrac dans le dossier courant) : `OUTPUT_DIRECTORY = doc`
   — le langage utilisé (ici C) : `OPTIMIZE_OUTPUT_FOR_C = YES`
   — on peut aussi spécifier le fichier qui sera documenté (ou bien un dossier si besoin) : `INPUT = TP1_b.c`
   — ensuite, on peut configurer beaucoup de choses comme les formats de sortie de la documentation (html, latex, rtf...) mais ça ne sera pas nécessaire pour l'instant.
   Les commentaires du code du type `/** */` sont analysé par doxygen (en particulier les valeurs de retour des fonctions).
2. Génération : la documentation est générée en appelant la commande `doxygen` dans le dossier qui contient le fichier de configuration généré et modifié à l'étape précédente.
3. Par défaut, la documentation html et LaTeX sont générées. Pour fabriquer le pdf grâce à LaTeX, il faut aller dans le bon dossier `doc/latex` et lancer la commande `make` (on verra plus tard comment cela fonctionne). Si tout va bien le pdf `refman.pdf` est généré.

☞**À faire** ☐ Générer la documentation avec Doxygen

1.2 Préparons l'automatisation : l'utilitaire `make`

Il arrive souvent de lancer la compilation dans un terminal, sans l'aide d'un environnement de développement tel que CodeBlocks. Par exemple, le compilateur GCC peut

être lancé sur l'exemple précédent par la commande :

```
gcc tp1_b.c -o tp1_b
```

Ce qui permet de compiler `tp1_b.c` et de fabriquer l'exécutable `tp1_b`.

Quand les projets se compliquent, on peut mémoriser les commandes de compilation dans un fichier `makefile` que l'utilitaire `make` permet d'exploiter. La syntaxe (simplifiée) est la suivante :

```
cible:src1.c src2.c
       action
```

où `cible` représente ce que l'on veut fabriquer, `src1.c src2.c` est une liste de fichiers dont la fabrication de la cible dépend (si ces fichiers ont été modifiés après la derniere fabrication de la cible, celle-ci est reconstruite). La partie `action` (ne pas oublier de mettre une tabulation en début de ligne) contient la commande à lancer pour obtenir la `cible`.

Pour l'exemple précédent, le fichier `makefile` serait :

```
tp1_b:tp1_b.c
        gcc  tp1_b.c -o tp1_b
```

Quand on a un seul fichier source à compiler, ce n'est pas vraiment intéressant. Nous allons « éclater » l'exemple précédent sur plusieurs fichiers : cela permet de ne mettre dans un fichier que le code correspondant à une seule notion.

❑ Exercice 1.3

1. Reprendre l'exercice précédent en créant deux fichiers `paramcmdl.c` et `paramcmdl.h` qui contiendront toute la gestion des paramètres : les déclarations de type de donnée et les prototypes des fonctions dans `paramcmdl.h` et le code des fonctions dans `paramcmdl.c`. Le fichier principal (initialement `tp1_b.c`) ne contiendra plus que la fonction `main` (à cette occasion, il changera de nom : `tp1_c.c`). Placer tous les fichiers sources dans un même dossier `tp1_c` et écrire le fichier `makefile` qui compilera l'ensemble.

   Remarque 1.7
   Penser à protéger le fichier `paramcmdl.h` des inclusions multiples avec les commandes `#ifndef... #define ... #endif`

   À faire ☐ Réécrire le code pour séparer les fonctions de gestions des paramètres, les déclarations de types liés à la gestion des paramètres de la ligne de commande et le code d'exemple qui utilise ces types et fonctions.

   Dans ce dossier, vous aurez donc les fichiers suivants :

   ```
   tp1_c.c
   paramcmdl.c
   paramcmdl.h
   makefile
   ```

☞**À faire** □ Vérifier qu'en lançant 2 fois de suite la commande `make` dans ce dossier, la compilation n'aura pas lieu la deuxième fois[1].

2. On peut améliorer la compilation en séparant la compilation de la production de l'exécutable. La commande `gcc -c fichier.c` permet de ne faire que l'étape de compilation et, en cas de réussite, cela produit un fichier `fichier.o` qui pourra être utilisé pour fabriquer l'exécutable.
   Séparer la compilation (dans le fichier `makefile`) des fichiers `paramcmdl.c` et `tp1_c.c`.
3. Générer la documentation de `tp1_c` avec doxygen.

1.3 Livrables

Déposez sur l'ENT :
— dans un dossier compressé `tp1.zip` (ou `tp1.gz`) : le code de l'exercice 1.3 :
  — `tp1_c.c`
  — `paramcmdl.c`
  — `paramcmdl.h`
  — `makefile`
  — `refman.pdf`

1.4 Conclusion

Ce chapitre aura permis :
— de mettre en œuvre le passage d'arguments par la ligne de commande et plus précisément en permettant de la souplesse dans l'ordre des arguments et les types associés ;
— de compiler un code C en ligne de commande avec `gcc` ;
— de générer de la documentation du code avec un outil tel que `doxygen`
— de s'initier à l'utilisation de fichier `makefile` et à l'utilitaire `make` qui correspond ;
— de séparer le code en plusieurs fichiers (fichiers d'en-tête, fichiers C, variable `extern`).

Pour aller plus loin :
— Il n'y a pas de contrôle d'erreur sur les arguments de la ligne de commande (présence, type valide, etc.) ;
— on pourrait hiérarchiser les arguments de la ligne de commande : obligatoire/optionnel ;
— la syntaxe du fichier `makefile` peut etre bien plus complexe et permettre de générer des projets complexes (on peut utiliser des variables et utiliser des options de compilation dépendant du système utilisé).

1. Bien sûr, il faut que la première compilation ait abouti