TP 1– rappels de C et outils de développemenent

Concepts abordés

— Make et compilation séparée
— Débogage avec GDB et Valgrind
— Les arguments du main()

Au delà de cette séance, aucune séquence de compilation ne devra se faire sans `make`. Par ailleurs, nous vous inciterons à utiliser les outils de débogage présentés dans ce TP.

Pensez à utiliser la commande `man` pour obtenir de la documentation sur une fonction.

Contraintes et remise des TPs

— **Le code de chaque exercice doit être dans un repertoire séparé (exercice1, exercice2, etc.).**
— **Chaque répertoire d'exercice doit contenir un fichier `Makefile` permettant de compiler les réponses à chaque question.**
— Quand une question ne demande pas uniquement du code mais aussi du texte, des explications, des commentaires, etc. vous pouvez soit les rendre dans un fichier texte (.txt) dans le dossier de l'exercice, soit si ce n'est pas trop gros les mettre en commentaire dans votre code. **Si une question demande des explications ou une réponse texte et que vous vous contentez de rendre du code, vous n'aurez pas tous les points.**
— Une note de propreté et style du code est attribuée à votre travail et prise en compte lors de l'évaluation. Elle inclue : la lisibilité du code (taille des fonctions, noms de variables explicites, etc.), sa propreté (libération de mémoire, contrôle des erreurs, réemploi de variables, etc.), sa qualité algorithmique, la présence et la pertinence des commentaires, les gardes `#ifndef` dans les headers, l'indentation et mise en page. L'ensemble de ces consignes est détaillé dans le document "propreté et lisibilité du code" disponible sur la page du module.
— Le code source doit être **mis en ligne sur Moodle** sous forme d'une archive tar.gz [1] **au plus tard 15 minutes après la fin du TP.** Votre archive doit être propre : pas d'exécutables, pas de .o, pas de fichiers temporaires. En cas de problème de remise, contacter vos encadrants.
— Les supports de cours, les exemples et les fichiers utiles aux TPs sont **sur la page du cours sur Moodle**.

Barème de notation

Exercices 1 et 2 : coefficient 1. Exercice 3 : coefficient 2. Style, propreté et lisibilité du code : coefficient 1.

1. Pour faire une archive tar.gz : `tar cvzf mon_archive.tar.gz mon_repertoire`

Exercice 1

Cet exercice a pour but de vous familiariser, à partir d'un programme très simple, avec l'outil make et le concept de compilation séparée.

1. Dans un fichier source calcul.c, ecrivez la fonction eval_polynome qui prend pour arguments un nombre entier n positif ou nul, un tableau de réels tab de dimension n+1, un nombre réel x. et qui retourne la valeur du polynome $tab[n] \times x^n + tab[n-1]x^{n-1} + ... + tab[1] \times x + tab[0]$. Défininissez le prototype de cette fonction dans le fichier d'en-tête calcul.h. Compilez le fichier calcul.c avec la commande gcc -c calcul.c.
2. Dans un fichier source affiche.c, écrivez la fonction affiche_polynome qui prend pour arguments un nombre entier n positif ou nul, un tableau de réels tab de dimension n+1, un nombre réel x. et qui affiche la valeur du polynome $tab[n] \times x^n + tab[n-1] \times x^{n-1} + ... + tab[1] \times x + tab[0]$. Défininissez le prototype de cette fonction dans le fichier d'en-tête affiche.h. Compilez le fichier affiche.c avec la commande gcc -c affiche.c.
3. Dans un fichier source saisie.c :
   — Écrivez la fonction saisie_valeur_reelle qui prend pour arguments un pointeur sur un réel x ; demande une valeur réelle à l'utilisateur du programme et l'affecte à x.
   — Écrivez la fonction saisie_valeur_entiere qui prend pour arguments un pointeur sur un entier n ; demande une valeur entière à l'utilisateur du programme et l'affecte à n après s'être assuré que la valeur indiquée est bien entière, positive ou nulle.
   — À l'aide des deux fonctions précédentes, écrivez la fonction saisie_coeff qui prend pour arguments un pointeur sur un tableau tab de réels non alloué, un pointeur sur un entier n ; demande le degrés du polynome considéré à l'utilisateur du programme ; l'affecte à n après s'être assuré que la valeur indiquée est bien entière, positive ou nulle ; procède à l'allocation dynamique de la mémoire pour le tableau tab ; demande à l'utilisateur de saisir les coefficients du polynome et les affecte dans le tableau tab.

   Défininissez le prototype de ces fonctions dans le fichier d'en-tête saisie.h. Compilez le fichier saisie.c avec la commande gcc -c saisie.c.
4. Dans un fichier source main.c, écrivez la fonction principale main() qui appelle les fonctions de saisie d'un polynome et d'affichage de sa valeur. Compilez le fichier main.c avec la commande gcc -c main.c. Effectuez l'édition de liens afin d'obtenir l'exécutable polynome avec la commande :
   gcc -o polynome main.o calcul.o affiche.o saisie.o
5. Etablissez le graphe des dépendances entre les différents fichiers source, d'en-tête et .o nécessaires à l'obtention de l'exécutable polynome. A partir de ce graphe, créez un fichier Makefile contenant les règles d'obtention de votre exécutable (Cf. ci-après pour un rappel sur les Makefile). Effacez l'ensemble des fichiers .o ainsi que l'exécutable polynome. Lancez la commande make et assurez vous que votre exécutable est bien créé.

Rappel : Un fichier Makefile permet d'indiquer l'ensemble des opérations nécessaires à l'obtention d'un exécutable[2] (cadre général de la compilation en C). La commande

2. make est en fait beaucoup plus général que cela. Par exemple, ce document a été réalisé en utilisant make.

make exécute les opérations indiquées dans le fichier Makefile. Cet outil permet donc d'automatiser l'obtention d'exécutables pour lesquels le nombre d'opérations à effectuer préalablement à leur génération est suffisament grand pour que cela devienne fastidieux et compliqué d'effectuer ces opérations "à la main" (en ligne de commande, les unes à la suite des autres). Par ailleurs, en utilisant les dates d'accès aux fichiers, make est capable de déterminer quels fichiers doivent être recompilés ou non. Dans le cadre de projets informatiques conséquents, cela permet de gagner un temps considérable pendant les étapes de compilation.
La seule connaissance strictement nécessaire pour l'écriture de fichiers Makefile "simples" est celle de l'écriture des règles :

```
CIBLE:    DEPENDANCES
        COMMANDE
...
```

CIBLES peut représenter le nom d'un fichier à produire (un fichier .o, un exécutable...) ou de manière plus générale, un nom quelconque (question2 par exemple).
DEPENDANCES représente l'ensembles des règles qui doivent avoir été exécutées et/ou l'ensemble des fichiers qui doivent exister afin de pouvoir lancer la commande COMMANDE.
L'exemple qui suit produit un exécutable monprog à partir des fichiers main.o et fichier1.o. Ces deux fichiers .o dépendent de leurs fichiers source respectifs. De plus main.o dépend de fichier1.h. La règle clean permet d'effacer les fichiers .o. La règle vclean applique la règle clean puis efface l'exécutable monprog.

```
1  #### Exemple de Makefile
2  monprog: main.o fichier1.o
3          gcc -o monprog main.o fichier1.o
4
5  fichier1.o: fichier1.c
6          gcc -c fichier1.c
7
8  main.o: main.c fichier1.h
9          gcc -c main.c
10
11 clean:
12         rm -f *.o
13
14 vclean: clean
15         rm -f monprog
```

Un bon document de référence sur make peut être trouvé à cette adresse :
http://www.laas.fr/~matthieu/cours/make/.

Exercice 2

Cet exercice a pour but de vous familiariser, à partir d'exemples simples, avec les outils de débogage classiques.

1. Soit le programme suivant :
   #### exemple.c ####

```c
#include <stdio.h>

void somme_n(void){

  int n_max;

  printf("calcule la somme des n premiers entiers, entrez n : ");
  scanf("%d", n_max);

  int somme = 0;

  for(int i = 0 ; i <= n_max ; i++)
    somme += i;

  printf("la somme est %d\n", somme);

}

int main(void){

  somme_n();
  return 0;

}
```

Téléchargez-le depuis le site du cours et écrivez le Makefile nécessaire à sa compilation. Générez l'exécutable et lancez le. Que constatez-vous ?

2. Pour débugger ce programme, nous allons utiliser le debugger GDB. Ajoutez dans votre Makefile les options -g -O0 aux appels de la commande gcc. Ces options permettent le debugage de votre programme. Recompilez avec make (au préalable, un clean est nécessaire). Lancez votre programme comme suit : gdb ./mon_executable. La console de l'outil gdb se lance. Utilisez-le pour débugger votre programme. Quel est l'orgine du problème ? Comment le corriger ?

Fonctionnement de GDB :

— Vous pouvez lancer l'exécution de votre programme en tapant run dans la console
— Au moment où le programme plante, il va rendre la main à gdb qui va vous permettre d'explorer l'état du processus à cet instant. La commande backtrace (ou bt) vous permet de visualiser la pile d'appel des fonctions, c'est à dire l'ensemble des fonctions appelées dans votre programme à cet instant, depuis le main jusqu'à la fonction directement à l'origine de l'erreur ;
— Vous pouvez naviguer dans la pile par les commandes up et down, ou aller directement à un niveau de la pile par la commande frame n (où n est le numéro de la tranche de pile de la fonction où vous souhaitez vous rendre, telle qu'affichée par backtrace. Utilisez ceci pour vous rendre dans la tranche de pile de la fonction foo
— Une fois dans la tranche de pile de foo, vous pouvez inspecter le contenu des variables avec la commande print v (où v est une variable dans le scope courant). Utilisez cette commande pour inspecter le contenu des variables de foo et identifier le problème.

Note : gdb est beaucoup plus puissant que les quelques commandes présentées ici. Il est possible de définir des points d'arrêt (breakpoint), de dérouler l'exécution du programme pas à pas (step), etc. Vous pouvez consulter l'aide de GDB (commande help) ou les nombreux tutoriels comme https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/gdb.html ou http://perso.ens-lyon.fr/daniel.hirschkoff/C_Caml/docs/doc_gdb.pdf.

3. Soit le programme suivant : #### exemple2.c ####

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

double * gen_n_doubles(int n){

  // Allouer de la memoire
  double* x = (double *) malloc(100*sizeof(double));

  // Initialise le generateur de nombres pseudo-aleatoires
  srand(time(NULL));

  for(int i = 0; i <= n; i++)
  {
    // Genere un reel compris entre 0 et 1
    *(x+i) = (double) rand()/(RAND_MAX);
  }
  return x;
}


int main(void){

  while(1)
  {
    // Generer 100 doubles
    double* nombres = gen_n_doubles(100);
    // Les afficher :
    for(int i = 0; i <= 100; i++)
      printf("%d | %f\t",i,nombres[i]);
    printf("\n");
  }
  return 0;
}
```

Téléchargez-le depuis le site du cours et ajoutez au Makefile précédent une règle permettant sa compilation. Générez l'exécutable et lancez le. A priori, le programme fonctionne-t-il conformément à vos attentes ?

4. Lancez le programme avec l'outil de debugage mémoire valgrind : valgrind ./question2, laissez-le tourner un moment et arrêtez-le avec Ctrl-C : vous devriez obtenir un grand volume de mémoire définitivement perdu ("definitely lost"). On suspecte donc un problème avec l'allocation mémoire. Pour mieux comprendre ce qu'il se passe, on va utiliser valgrind avec l'outil massif : valgrind --tool=massif ./question2 ; puis ms_print massif.out.xxx

où xxx représente le numéro devotre processus. Le premier graphique sorti par ms_print correspond à l'utilisation mémoire en fonction du temps. Copiez ce graphique dans vos réponses. Que constatez vous ? Comment s'explique ce comportement et comment le corriger ? Corrigez-le.

5. Ce programme contient encore des erreurs. Retirez maintenant le **while**(1) du main pour n'exécuter la boucle qu'une seule fois et ajoutez dans votre Makefile les options -g -O0 à la commande gcc. Recompilez avec make (au préalable, un make clean est nécessaire). Lancez votre programme comme suit : valgrind ./question2. Lisez attentivement les informations fournies par valgrind, en particulier les erreurs de lecture et d'écriture ("read error" et "write error"). Quelle est l'origine de ces erreurs ? Modifiez votre programme afin de toutes les corriger (la dernière ligne de valgrind doit contenir : "ERROR SUMMARY : 0 errors from 0 contexts").

Exercice 3

Cet exercice a pour but de vous familiariser, à partir d'exemples simples, avec les arguments standards de la fonction main() les plus couramment utilisés.

La fonction main() peut en effet prendre deux arguments standards auquel cas son prototype s'écrit :

```
int main(int argc, char **argv);.
```

L'argument argc représente le nombre de paramètres + 1 passés lors de l'appel de votre exécutable. Par exemple, si vous lancez l'exécutable mon_prog avec deux paramètres comme suit : ./mon_prog param1 param2, argc sera égal à 3.

L'argument argv est un pointeur sur un tableau de chaine de caractères contenant (nombre de paramètres + 1) cases. Dans l'exemple précédent nous avons :

- argv[0] qui contient la chaine de caractère correspondant à la commande, soit "./mon_prog" ;
- argv[1] qui contient la chaine de caractère correspondant au premier paramètre, soit "param1" ;
- argv[2] qui contient la chaine de caractère correspondant au second paramètre, soit "param2".

Ce mécanisme permet un assez grande souplesse de passage de paramètres à un programme. Ce nombre de paramètres peut varier pour un même programme rendant très flexible son utilisation. Les commandes du shell sont des exemples typiques de cette souplesse. Par exemple, il est possible d'appeler la commande ls de plein de manières différentes parmi lesquelles :

- ls ;
- ls -al ;
- ls -lt *.o ;
- ...

Enfin il est important de remarquer que le tableau argv ne contient que des chaines de caractères et si un paramètre passé au programme est sensé représenter un nombre, il faut alors dans le code faire appel à des fonctions de conversions telles que atoi et atof.

1. Ecrivez un programme qui affiche l'ensemble des paramètres qui lui sont passés.
2. En utilisant la fonction atof ou strtof, écrivez un programme qui prend en arguments un nombre quelconque de nombre décimaux et en affiche la somme. Par exemple :

```
./question2 1.5 3 -0.2
La somme est 4.300000
```

3. Reprenez l'exercice sur le calcul de polynome en considérant que la valeur des coefficients et de x sont passés comme des paramètres du programme par l'utilisateur. Le formatage retenu pour le passage des paramètres est le suivant :
   ./polynome --val x --coeff coeff_x^0 coeff_x^1 .... coeff_x^n

   Si aucun paramètre n'est passé au programme ou si la liste des paramètres passés n'est pas cohérente, le programme doit afficher un manuel d'utilisation. De la même manière, si le programme est appelé avec le paramètre --h ou --help, le programme doit afficher un manuel d'utilisation.

   Attention : pour avoir le maximum des points, l'ordre des arguments doit être indifférent, par exemple :

   ```
   endy@demandred % ./polynome --val 2 --coeff 1 2 3
   valeur du polynome : 17.000000
   endy@demandred % ./polynome --coeff 1 2 3 --val 2
   valeur du polynome : 17.000000
   endy@demandred % ./polynome --help
   usage : ./polynome --coeff c1 c2 c3 ... --val x
   ```

4. **BONUS :**

   Ajoutez une fonction affiche_formel au fichier source affiche.c et une option --formel à la ligne de commande qui affiche le polynome de manière formelle, soit, par exemple :

   ```
   endy@demandred % ./question2 --formel --val 2 --coeff 1 2 3
   1.000000 + 2.000000 x + 3.000000 x^2
   valeur du polynome : 17.000000
   ```

   Cette affichage formel est réalisé en plus de l'affichage du résultat si le paramètre --formel est passé comme premier ou comme dernier paramètre. L'option --formel peut être placée n'importe où dans la ligne de commande (avant le --val x, aprés les coefficients, ...).