TP 6 – Tableaux et debugger

1. Le debuggeur en ligne.

Nous allons nous familiariser avec gdb, le debuggeur en ligne. Vous trouverez son interface sommaire, et vous vous demanderez pourquoi on utilise pas de versions graphiques. C'est parce qu'il est très léger et disponible sur tous les Linux. **La syntaxe utilisée par gdb est la même que celle du langage C pour l'accès aux variables. Voici un mini résumé des commandes de gdb (*disponible évidemment en tapant man gdb*) :**

commande	exemple	signification
list	l	liste le programme
l 10	l 10	liste à partir de la ligne 10
help	h list	aide de la commande list
breakpoint	b 12	met un point d'arrêt à la ligne 12
	b main	met un point d'arrêt à la fonction main()
run	r	exécute un programme
continue	c	continuer (après un point d'arrêt)
next	n	exécute la prochaine instruction ou si c'est une fonction, la fonction en entier
step	s	exécute la prochaine instruction ou juste l'appel si c'est une fonction
print	p i	affiche le contenu de la variable i
	p &i	affiche l'adresse de i
x (examine)	x /16db &i	affiche 16 octets (b) en décimal (d) à partir de l'adresse de i
quit	q	quitte le debugger

Editez le programme suivant.

```
#include <stdio.h>
#define TAB_LEN 5

/** saisi à partir du clavier des entiers
@param tableau : le tableau à remplir
@param taille : la taille du tableau
@return : rien */
void saisir (int tableau[], int taille) {
    int i ;
    for (i = 0 ; i<taille ; i++ ) {
        printf ("Entrer la valeur %d : ", i);
        scanf ("%d", &tableau[i] ) ;
    }
    return ;
}

/** Affiche un tableau 1D
@param tableau : le tableau
@param taille : la taille du tableau
@return : rien */
void afficher (int tableau[], int taille) {
    int i ;
    for (i = 0 ; i<taille ; i++ ) {
        printf ("valeur %d : %d\n", i, tableau[i] ) ;
    }
    tableau[10000] = 0 ;
    return ;
}

void main () {
    int tab[TAB_LEN] ;
    saisir(tab, TAB_LEN) ;
    afficher (tab, TAB_LEN) ;
}
```

Suivez ce tutoriel afin d'apprendre à vous servir de gdb.

1) **Compilez** le programme précédent en ajoutant les options de debuggage « -g » de cette manière :
   `gcc -g ex1.c -o ex1.bin`
2) Dans une console **lancez** `gdb ex1.bin`
3) Un message s'affiche. **Tapez** les commandes suivantes, sans taper `(gdb)` qui est le prompt. Affichons d'abord le programme :
   `(gdb) list`
4) ou encore pour afficher à partir de *main()* :
   `(gdb) list main`
5) Mettons un point d'arrêt à la fonction *main()*:
   `(gdb) b main`
6) Exécutons le programme ; l'exécution s'arrêtera aussitôt à cause du point d'arrêt et affichera la prochaine instruction à exécuter :
   `(gdb) r`
7) Affichons le contenu du tableau. Ici, il n'est pas initialisé donc les valeurs n'auront aucun sens :
   `(gdb) p tab`
8) Exécutons d'un seul coup la prochaine instruction (c'est l'appel de la fonction saisir) :
   `(gdb) n`
9) Entrez les valeurs demandées par *saisir( )* une par une. Prenons par exemple « 1 2 3 4 5 ».
10) Le debugger s'arrête donc sur l'appel de la fonction *afficher( )*. Affichons de nouveau le contenu du tableau :
    `(gdb) p tab`
11) Nous allons maintenant exécuter l'appel de la fonction *afficher( )* puis s'arrêter sur la première instruction de la fonction *afficher( )* :
    `(gdb) s`
12) Le debugger s'arrête donc sur la boucle *for.* Exécutons pas à pas ces instructions :
    `(gdb) n`
13) Chaque fois que l'on appuie sur la touche entrée, le debugger exécute la dernière commande (ici next) :
    `(gdb) <appuyez sur Entrée>`
14) Affichons le tableau. Ici « *tableau* » est l'adresse de la première case. Son affichage provoquera l'affichage de cette adresse :
    `(gdb) p tableau`
15) Si on veut afficher le contenu à cette adresse (on retrouve la première valeur du tableau) :
    `(gdb) p *tableau`
16) Si on veut afficher le contenu de la deuxième case :
    `(gdb) p tableau[1]`
17) Affichons en décimal (d) les 5 valeurs du tableau qui sont des entiers sur 4 octets (w) à partir de la première case :
    `(gdb) x/5wd tableau`
18) Inspectons la mémoire à partir de l'adresse *tableau* en hexadécimal (x) les 64 octets (b) :
    `(gdb) x/64bx tableau`
    Remarquez que le premier octet vaut 0x01, c'est à dire 1 en décimal. Les 3 suivants sont à zéro. C'est normal, puisque les entiers sont codés sur 4 octets. Remarquez également que c'est l'octet de poids faible qui arrive en premier. C'est parce que les processeurs Intel travaillent de cette façon. Ils sont appelés architecture Little Endian.
19) Continuons l'exécution du programme jusqu'au prochain point d'arrêt. Puisqu'il n'y en a pas, le debuggger ira jusqu'à la fin :
    `(gdb) c`
20) Le programme s'arrête sur un message d'erreur SIGSEGV. Vous en aurez beaucoup dans la suite des TP. Gdb pourra vous aider. Saurez-vous trouver cette première erreur?
21) Mettons fin à la session de debug :
    `(gdb) q`

Obscure ??
gdb est l'outil de base pour déboguer. Il reste souvent dans des applications embarquées sans écrans le seul moyen de débogage.
Pour les TP son utilisation peu intuitive peut dérouter. Si l'aspect rustique de gdb vous rebute, utilisez codeblocks

2. remplir un tableau de nombres aléatoires

Écrire le code de la fonction *remplirAlea* dont la signature est

```
void remplirAlea (int tableau[], int taille, int modulo)
```

Cette fonction remplit un tableau de nombres tirés aléatoirement et compris entre 0 et (*modulo - 1)*. Le paramètre *taille* est la taille du tableau.
Dans le programme précédent, **remplacer** l'appel de la fonction *saisir* par l'appel de la fonction *remplirAlea*.

3. Crible d'Eratosthène *–correction disponible–*

Écrire un programme qui affiche tous les nombres premiers inférieurs ou égaux à un nombre entier positif N donné, sans faire appel à l'opération de division. Pour cela, l'algorithme utilisé sera le crible d'Eratosthène, dont les principales étapes sont :

- On crée un tableau à N éléments, tous initialisés à la valeur 1 ;(ci-dessous exemple avec N=20)
- On affecte 0 aux 2 premiers éléments du tableau, pour signifier que les nombre 0 et 1 sont rayés (car ils ne sont pas premiers) ;

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
valeur	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- On commence au deuxième élément, non rayé (i.e. sa valeur reste à 1), car le nombre 2 est premier ; en sautant de 2 en 2 les éléments suivants du tableau, on atteint tous les multiples de 2, que l'on raye ;

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
valeur	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

- On aborde ensuite le troisième élément ; il est non rayé : 3 est premier ; en sautant de 3 en 3 les éléments suivants du tableau, on atteint tous les multiples de 3, que l'on raye ;

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
valeur	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1

- On aborde ensuite le quatrième élément ; il est rayé : 4 n'est pas premier ;
- Et ainsi de suite ; on peut s'arrêter lorsque l'on a abordé tous les entiers inférieurs ou égaux à √N.

Pour répondre à ce problème on créera 3 fonctions qui seront appelées dans la fonction main :

- `void initTab(int tab[], int taille)`
- `...eratostene(...)`
- `...afficherPremier(...)`

Il n'est pas demandé de dialogue avec l'utilisateur.

4. Parcours de tableaux

Soit un tableau d'entiers contenant n valeurs tirées au hasard. **Écrire** les algorithmes suivants sous forme de fonction :

– calculer la somme des éléments du tableau ;
– rechercher l'élément maximum ;
– trier le tableau par ordre croissant. Le tri sera fait par l'algorithme du tri bulle (cf. sur le web).

5. Tableau à 2 dimensions

Rappel : en langage C, les tableaux à deux dimensions se déclarent de manière semblable à ceux à une dimension ; par exemple : int tableau [10][20] ; . Par convention, le premier indice repère la ligne et le second la colonne, ce qui signifie que, dans le cadre d'une énumération, les éléments sont rangés par ligne (on ne trouve les éléments de la deuxième ligne qu'une fois parcourus ceux de la première ligne). Comme pour les tableaux à une dimension, il convient d'être attentif aux risques de débordements d'indice.

Écrire une fonction saisir(… à vous de définir le prototype...) qui saisit les 9 valeurs d'une matrice 3x3, à entrer ligne par ligne à l'utilisateur et une fonction afficherTransposee(… à vous de définir le prototype...) qui affiche la matrice colonne par colonne. Voici un exemple d'exécution :

```
Donnez les valeurs de la ligne 0 : 1 2 3
Donnez les valeurs de la ligne 1 : 4 5 6
Donnez les valeurs de la ligne 2 : 7 8 9
Voici la colonne 0 : 1 4 7
Voici la colonne 1 : 2 5 8
Voici la colonne 2 : 3 6 9
```

6. Chaînes de caractères

1. préambule

Ouvrir le fichier ex6.c de l'archive. Expliquez les résultats affichés et la différence entre *sizeof* et *strlen*. Pourquoi *fgets* a été utilisé à la place de scanf ?

```
int main(int argc, char** argv)
{
        char objet1[]="Bonjour le monde" ;
        char *objet2 =  "Bonjour le monde" ;
        char phrase[PHRASE_LEN] ;


        printf (" longueur de objet1 = %ld   (c'est la longueur du tableau de caractères)\n", sizeof (objet1) ) ;
        printf (" longueur de objet2 = %ld   (c'est la longueur du pointeur)\n", sizeof (objet2) ) ;
        printf (" longueur de la chaine référencée par objet1 = %ld   (c'est la longueur de la chaine)\n", strlen (objet1) ) ;
        printf (" longueur de la chaine référencée par objet2 = %ld   (c'est la longueur de la chaine)\n", strlen (objet2) ) ;

        printf("Entrez une phrase : ") ;
        fgets (phrase, PHRASE_LEN, stdin );
        mofifierFinChaine( 5, phrase ) ;
        printf (" longueur de phrase : %ld\n", sizeof (phrase) ) ;
        printf (" longueur de phrase : %ld\n", strlen (phrase) ) ;
}

/** insère un caractère de fin de chaine à un index
@param idx : index d'insertion du caractère de fin de chaine
@param chaine : la chaine à modifier
@return : rien
*/
void mofifierFinChaine( int idx, char *chaine )
{
        if (idx >= 0) {
                chaine[idx] = '\0' ;
        }
}
```

2. Exercice

Écrire une fonction qui retourne le nombre d'espaces dans une chaîne de caractères.

```
int nbEspaces (const char *chaine) ;
```

3. Exercice *–correction disponible–*

Écrire un programme qui vérifie si un texte saisi au clavier et se terminant par un point est correctement parenthésé ou non.
Exemples :

```
(Cette) ((chaîne est) ( ) (correctement) parenthésée)
Cette (chaîne)) ne ((l'est pas)
(Celle-ci non) ((plus)
```

4. Exercice

Écrire une fonction qui remplace tous les caractères *motif* par le caractère *nouveau* à partir d'une chaîne de caractères *src*. Le résultat est écrit dans une chaîne *dest.*
Le prototype de le fonction est le suivant :

```
void remplaceChar (char *dest, const char *src, char motif, char nouveau ) ;
```

Par exemple, l'appelle suivant remplace les « , » par des « . » et retourne dans la chaîne *destination* «12.3 7.4»

```
char destination[100] ;
remplacer (destination, "12,3 7,4", ',', '.' ) ;
```

5. Exercice *(pour les plus rapides)*

Écrire une fonction retournant le nombre de mots d'une chaîne de caractères. De plus chaque mot de la chaîne sera rangé dans un tableau passé en paramètre de la fonction. Pour simplifier cet exercice, on supposera que la dimension des mots sera au plus de 14 caractères, et la chaîne ne contiendra pas plus de 10 mots. Écrire un main() de validation de la fonction.

```
int parseMots( const char *chaine, char tabMots[][15] ) ;
```