Université de Corse - Pasquale PAOLI

Diplôme : Licence SPI 3ème année, parcours Informatique | 2023-2024

UE : Programmation impérative avancée

Enseignants : Marie-Laure Nivet & Diego Grante & Sakhi Shokhou

Exercices C – TD 1

Pour tous les exercices il vous est demandé de respecter la structure de code proposée en cours via le fichier structureProgrammeC.c.
Vous veillerez également à assurer la lisibilité de vos codes et pour cela à les commenter, les indenter, à choisir les noms de vos variables et de vos fonctions.
Vous réaliserez également pour chaque programme ou groupes d'exercices un MakeFile (cf. transparents du cours) associé, permettant de faire le build de l'exécutable ainsi que le clean à posteriori.
Vous soignerez également vos tests.

A. Exercices de mise en jambes ! Très/trop faciles !

1. Récupération et somme des arguments passés en ligne de commande

La vraie signature de la fonction main est la suivante :

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

- où `argc` représente le nombre d'arguments passés en ligne de commande (séparés par un espace)
- et `argv[]` est initialisé avec ces chaines de caractères passées en ligne de commande au lancement du programme. La première chaîne stockée est toujours le nom du programme.

On souhaite écrire un programme nommé `sum` qui récupère les nombres passés en ligne de commande (au moins deux) qui les convertit en entier, les additionne tous et affiche :

- la somme obtenue

```
$>./sum 4 16 20 5
45
```

- ou un message d'erreur si :
  - au moins deux paramètres entiers n'ont pas été passé en ligne de commande, dans ce cas on affichera

```
$>./sum 4
Wrong usage, at least 2 parameters expected:
./sum param1 param2
```

  - un moins l'un des paramètres passés n'a pu être converti en entier via la fonction atoi pour ASCII to integer qui renvoie l'entier représenté par la chaine de caractère passée en argument ou 0 si la chaine ne peut être convertie.

```
int atoi( const char * theString ); //Fonction définie dans stdlib.h

$>./sum 4 essai
There is a problem with args 2, essai. It could not be transformed
in int. Please retry !
```

2. Expressions booléennes (à ne faire que pour les personnes se sentant fragiles à ce niveau)

Dans un algorithme qui analyse des résultats d'un examen, quatre variables permettent de décrire l'environnement : les variables numériques Nlv, Nf, Nm, Np qui indiquent

respectivement, pour un candidat donné, des notes littéraires : langue vivante (Nlv), de français (Nf), et des notes scientifiques : mathématiques (Nm), et physique (Np). On suppose que les notes sont calculées sur 20 et qu'elles ont toutes le même coefficient.

Former les expressions logiques (et seulement elles) correspondant aux situations suivantes :

1) la moyenne des quatre notes est supérieure à 10
2) les notes de mathématiques et de français sont supérieures à la moyenne des quatre notes
3) il y a au moins une note supérieure à 10
4) toutes les notes sont supérieures à 10
5) la moyenne (10) est obtenue pour l'un des deux types (littéraire et scientifique)
6) la moyenne des quatre notes est supérieure ou égale à 10 et la moyenne (10) est obtenue pour au plus l'un des deux types

3. Entrée/sortie de base, lecture simple

Écrivez un programme (donc un main mais sans nécessairement un tableau d'argument !) qui demande l'âge de l'utilisateur, puis qui l'affiche. Pour lire l'âge, vous utiliserez la fonction `scanf` déclarée dans stdio.h. Par exemple utilisez l'invocation `scanf("%d",&ageLu);`

4. Entrée/sortie de base, lecture multiple de simples caractères

Écrivez un programme qui demande à l'utilisateur de saisir trois caractères et qui ensuite affiche les caractères saisis.

5. Définition de macro fonction

Dans un programme conduisant un dialogue au terminal, chaque introduction de données, par appel de la fonction `scanf()`, répond à un message de demande affiché par printf(). Écrire une macro définition `pscanf()` recevant trois paramètres : le texte du message de demandé, le format de la réponse, l'adresse de la donnée à lire.

- Testez ensuite cette macro depuis un main.
- Arrêtez la compilation après l'étape de pré-processing pour vérifier que votre macro est bien traduite correctement.
- Préparez un MakeFile et faîtes une *target* permettant d'arrêter la compilation après le pré-processing.

6. Entrée/sortie de base, Chiffrement

Écrivez un programme qui chiffre un mot saisi par l'utilisateur à l'aide d'une valeur de clé (un entier qui va servir de décalage de caractères) entrée également par l'utilisateur. Proposez les deux versions décrites ci-dessous (2) et (3).

1. Écrivez en premier lieu les spécifications de ces fonctions de chiffrement. Vous veillerez à les inclure en en-tête dans vos codes.
2. Le mot chiffré sera affiché en une seule fois et ce, une fois le mot origine et la clé saisis. Pour afficher un caractère seul vous pouvez utiliser `putchar(char c)`. Vous écrirez la fonction de chiffrage et la fonction de déchiffrage, à vous d'en définir les paramètres.
3. La clé sera en premier lieu demandée à l'utilisateur et ensuite, le mot chiffré sera affiché au fur et à mesure de la saisie du mot origine. Pour cela vous devrez coder en mode console afin de pouvoir déplacer le curseur d'écriture où vous souhaitez. L'une des façons de gérer cela est d'utiliser la librairie ncurses qui permet de gérer les entrées/sorties non blocantes en mode console. Pour compiler vous devrez utiliser l'option -lncurses.

7. Conversion en chiffres romains

Écrire un programme qui convertit un nombre entier en chiffres romain en utilisant l'ancienne notation : par exemple 4(IIII), 9(VIIII), 900(DCCCC).
Les éléments de base sont :
I : 1 ; V : 5 ; X : 10, L : 50, C : 100, D : 500, M : 1000.
Une fois que votre programme fonctionnera avec cette ancienne notation, vous intégrerez les modifications permettant d'effectuer les corrections pour 4(IV) et 9(IX).
Vous veillerez à organiser votre code en fonctions.

B. Pointeurs et gestion de la mémoire

1. Identifiez dans les extraits de code suivants[1] les éventuels problèmes/erreurs de gestion de mémoire créés (cf. pour une vue d'ensemble des pbs T114, 115).

```c
include <stdio.h>
#include <stdlib.h>
#define NB  100
int* fonctionX(){
   int tab[NB] = { 0 };
   //...
   return tab;
}

int main(){
  int* t = fonctionX();
  for (int i = 0; i < NB; i++){
    t[i] = rand() % 100;
    printf("t[%d]=%d\n", i, t[i]);
  }
  return 0;
}
```

```c
#include <stdio.h>

int *fun(){
   int x = 5;
   return &x;
}
int main(){
        int *p = fun();
   fflush(stdin);
   printf("%d\n", *p);
   return 0;
}
```

```c
void main()
{
  int *ptr;
  .....
  .....
  {
     int ch;
     ptr = &ch;
  }
  .....
  printf("%d\n", *ptr);
}
```

```c
#include<stdio.h>
int main(){
   int *piData;
   *piData = 10;
   return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
   int *piData = NULL;
   piData = malloc(sizeof(int) * 10);
   if(piData == NULL){
      return -1;
   }
   free(piData);
   free(piData);
   return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
int main (){
   int *piBuffer = NULL;
   int n = 10;
   //creating an integer array of size n.
   piBuffer = malloc(n * sizeof(int));
   //make sure piBuffer is valid or not
   if (piBuffer == NULL){
      fprintf(stderr, "Out of memory!\n");
      exit(1);
   }
   printf("Size of allocated array is
%d\n",sizeof(piBuffer));
   free(piBuffer);
   return 0;
```

[1] D'après des exemples tirés de https://aticleworld.com/mistakes-with-memory-allocation/

		} }

#include<stdio.h>
#include<stdlib.h>
int main(void){
int *piBuffer = NULL;
int n = 10, i = 0;
piBuffer = malloc(n * sizeof(int));
//Assigned value to allocated memory
for (i = 0; i < n; ++i){
piBuffer [i] = i * 3;
}
//Print the value
for (i = 0; i < n; ++i){
printf("%d\n", piBuffer[i]);
}
//free up allocated memory
free(piBuffer);
return 0;
}
	#include <stdio.h>
	#include <stdlib.h>
	int main(){
	int *piData1 = NULL;
	int *piData2 = NULL;
	//allocate memory
	piData1 = malloc(sizeof(int));
	if(piData1 == NULL){
	return -1;
	}
	*piData1 = 100;
	printf(" *piData1 = %d\n",*piData1);
	piData2 = piData1;
	printf(" *piData1 = %d\n",*piData2);
	//deallocate memory
	free(piData1);
	*piData2 = 50;
	printf(" *piData2 = %d\n",*piData2);
	return 0;
	}
		int main (){
		char * pBuffer = malloc(sizeof(char) * 20);
		/* Do some work but nothing to do with pBuffer*/
		return 0;
		}

#include<stdio.h>
#include<stdlib.h>
int main(){
int *piData = NULL;
piData = malloc(sizeof(int) * 10);
free(piData);
*piData = 10;
return 0;
}
	#include<stdio.h>
	#include<stdlib.h>
	int main(){
	int Data = 0;
	int *piData = &Data;
	free(piData);
	return 0;
	}
		int * Foo(int *x, int n){
		int *piBuffer = NULL;
		int i = 0;
		//creating an integer array of size n.
		piBuffer = malloc(n * sizeof(int));
		//make sure piBuffer is valid or not
		if (piBuffer == NULL){
		// allocation failed, exit from the program
		fprintf(stderr, "Out of memory!\n");
		exit(1);
		}
		//Add the value of the arrays
		for (i = 0; i < n; ++i){
		piBuffer[i] = piBuffer[i] + x[i];
		}
		//Return allocated memory
		return piBuffer;
		}

2. Fonction d'addition (exercice très basique !)

Écrivez une fonction/procédure `add` qui prend trois paramètres a et b et c et qui met dans c le résultat de l'addition entre a et b. Écrivez ensuite un main où vous invoquerez votre procédure et vérifierez qu'elle a bien le comportement souhaité.

3. Allocation dynamique

Dans un programme

- allouer dynamiquement de la mémoire pour un char, un entier, un float,
- initialisez avec des valeurs entrées par l'utilisateur,
- affichez les valeurs saisies
- quittez si l'utilisateur le demande, sinon recommencez.

Vous ferez bien attention à la gestion de la mémoire.

4. StackOverflow…

Écrivez un programme générant une erreur de type StackOverflow.

C. Arrêt sur frappe de touche ?

Est-il possible avec scanf() d'écrire un programme qui affiche des zéros en continu et si une touche est appuyée, un nombre aléatoire de fois la lettre ou le chiffre correspondant ? Programmez ce qui est possible.

D. Nombres aléatoires

1. Librairie

Écrire une série de fonctions, qui génèrent et retournent des nombres aléatoires selon les contraintes suivantes :

- un nombre entier aléatoire selon la plage maximum du générateur aléatoire.
- un nombre entier aléatoire compris entre 0 et une valeur "seuil haut" passée en paramètre.
- un nombre entier aléatoire entier compris entre deux bornes "seuil bas" et une valeur "seuil haut" passées en paramètres.
- un nombre réel aléatoire compris entre 0 et 1.
- un nombre réel aléatoire à deux décimales compris entre deux bornes "seuil bas" et une valeur "seuil haut".

Faire une librairie `util_rand.c` et `util_rand.h` avec ces fonctions.

2. Dés

Écrire un programme affichant un menu proposant de jouer avec un, deux, trois ou quatre dés. Selon le choix fait, le programme lance les dés. Les dés identiques sont relancés et il y a cumul des points. L'utilisateur gagne si le total est supérieur aux deux tiers du maximum (avec deux dés cela fait 8 : (12/3)*2. Le programme indique combien il manque pour gagner ou combien il y a en plus.

Exemples d'interaction :

```
Avec combien de dés voulez vous jouer ? Tapez 1, 2, 3 ou 4 ? 2
Vous avez choisi 2 dé(s)
Dé 1 : 5
Dé 2 : 5
La somme des 2 dés lancés est de 10, le seuil était de 8
score dépassant de 2 au seuil
Bravo, vous avez gagné !
```

```
Avec combien de dés voulez vous jouer ? Tapez 1, 2, 3 ou 4 ? 3
Vous avez choisi 3 dé(s)
Dé 1 : 1
Dé 2 : 4
Dé 3 : 5
La somme des 3 dés lancés est de 10, le seuil était de 12
score inférieur de 2 au seuil de 12
Désolé, vous avez perdu !
```

3. Test statistique de rand

Écrire une fonction qui tire aléatoirement 100000 fois une valeur comprise entre 0 et 5, compte les occurrences de chaque résultat et en affiche le pourcentage. Testez ensuite cette fonction depuis un main.

E. BufferOverflow…

Écrivez un programme générant une erreur de type BufferOverflow. Attention ce n'est jamais une bonne idée !! C'est juste pour vous faire réfléchir un peu à la manière de procéder !

F. Manipulation de tableaux dynamiques via pointeurs

Lorsque la taille d'un tableau doit varier au fil de l'exécution, on peut utiliser une démarche rustique mais simple consistant à définir une taille maximale, connue à la compilation, qu'on est certain de ne pas dépasser. Mais on peut améliorer les choses en recourant aux possibilités de gestion dynamique.

S'il s'agit d'un tableau qu'on est amené uniquement à agrandir ou à rétrécir par la fin, on pourra faire appel à realloc. En revanche, si, comme c'est le plus probable, on a besoin de pouvoir supprimer ou ajouter un élément en position quelconque, tout en conservant les autres, il peut être judicieux d'utiliser un tableau de pointeurs sur chacun des éléments. Ces

derniers ne seront plus nécessairement contigus en mémoire, mais grâce à l'existence du tableau de pointeurs (formé, lui, d'éléments consécutifs), on pourra accéder directement à n'importe quel élément du tableau.

Par exemple, supposons que l'on ait besoin d'un tableau d'éléments du type structure `point` défini ainsi :

```
struct point { int num ;
               float x ;
               float y ;
             } ;
```

Si l'on suppose que le nombre initial d'éléments est `n_elem` (il peut bien sûr s'agir d'une variable), on commencera par allouer un tableau de pointeurs par :

```
struct point **adp ;  /* notez les deux * : adp est du type pointeur sur un */
                      /* pointeur sur un objet de type struct point        */
   .....
adp = malloc (n_elem * sizeof (struct point *) ) ;
```

Puis, on allouera chacun des `n_elem` éléments (de type `point`) en plaçant son adresse dans l'élément correspondant du tableau `adp` :

```
for (i=0 ; i<n_elem ; i++)
   adp[i] = malloc (sizeof(struct point)) ;
```

À partir de là, `adp[i]` contient l'adresse du point de rang `i`, `*adp[i]` est un objet de type `struct point` correspondant au point de rang `i`. Les champs d'un tel point se notent simplement : `adp[i]-> num`, `adp[i]->x` ou `adp[i]->y`.

Voici un exemple simple de définition et d'utilisation d'une fonction qui se fonde sur ce mécanisme pour créer un tableau dont le nombre d'éléments lui est fourni en argument, tandis que leurs valeurs sont lues en données. On notera qu'il est nécessaire de lui fournir l'adresse de la variable `adp`, c'est-à-dire finalement une valeur de type `struct point ***`.

Gestion de tableaux dont la taille varie lors de l'exécution

```
#include <stdio.h>
#include <stdlib.h>
struct point { int num ;
               float x ;
               float y ;
             } ;
void init (struct point ***, int) ;   /* déclaration de init */
int main()
{ int n_elem ;                        /* nombre d'éléments */
  struct point **ad ;                 /* ad : adresse du tableau de pointeurs */
  printf ("combien d'elements : " ) ;
  scanf ("%d", &n_elem) ;
  init (&ad, n_elem) ;
}
void init (struct point ***adp, int n_elem)
{ int i ;
  *adp = malloc (n_elem * sizeof (struct point *) ) ;
  for (i=0 ; i<n_elem ; i++)
     (*adp)[i] = malloc (sizeof(struct point)) ;
  for (i=0 ; i<n_elem ; i++)
    { printf("numero, x, y : ") ;
      scanf("%d %f %f", &((*adp)[i]->num), &((*adp)[i]->x), &((*adp)[i]->y)) ;
    }
}
```

En pratique, il faudra bien entendu disposer d'autres fonctions telles que :

- suppression d'un élément de rang donné ;
- insertion d'un nouvel élément à un rang donné...

De telles opérations nécessiteront alors théoriquement une nouvelle allocation du tableau de pointeurs, ainsi que la recopie de certaines de ses valeurs. En revanche, elles ne nécessiteront aucune recopie des éléments eux-mêmes. Il suffira seulement soit d'une allocation d'un nouvel emplacement par `malloc`, soit d'une suppression d'un emplacement par `free`.

Dans ces conditions, on voit que le « prix à payer » pour pouvoir accéder directement à un élément quelconque se limite à la gestion du tableau de pointeurs lui-même et non plus du tableau d'éléments. La différence deviendra d'autant plus sensible que les éléments concernés seront de grande taille.

En pratique, on pourra améliorer les temps d'exécution en surdimensionnant le tableau de pointeurs, de manière à éviter sa réallocation systématique lors de chaque opération.

(Le guide complet du Langage C p. 627-628)

G. Faire sa propre librairie C, inverser une chaine de caractère en place

Ecrire une fonction qui prend en paramètre une chaîne de caractères (donc un pointeur sur le caractère de début !) qui renverse, en place (en modifiant la chaîne passée en paramètre), les caractères de la chaine de départ et qui en plus retourne la chaine renversée. Cette fonction sera nommée reverse et sera placée dans un fichier nommé libvotrenom.c.

H. Implémentation de la structure de données Pile d'entiers à l'aide d'une liste chainée

1. Proposez une structure permettant de représenter une pile d'entiers par une liste chainée.
2. Donnez les entêtes des fonctions permettant
    a. De savoir si la pile est vide
    b. D'empiler un élément
    c. De dépiler un élément
3. Donnez le code des fonctions précédentes