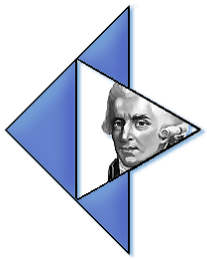

Université de Corse - Pasquale PAOLI

Diplôme : Licence SPI 3ème année, parcours Informatique | 2022-2023

UE : Programmation impérative avancée

Enseignants : Marie-Laure Nivet & Mickaël Lanfranchi

Exercices C

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

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'age, 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 deux versions.

1. 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.
2. 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;
}
```

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

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

```c
int main (){
```

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

```c
#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;
}
```

```c
#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;
}
```

```c
   char * pBuffer = malloc(sizeof(char) *
20);
   /* Do some work but nothing to do
with pBuffer*/
   return 0;
}
```

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

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

```c
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

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.