L2 S3 2010-2011

Programmation impérative en langage C

feuille de TDM n°5 : Diverses choses et la pile statique

---

1 Exercice 1 : Passages de paramètres

1.1 Programme 1

Voici un programme. Répondez aux questions posées en commentaires, puis tapez le programme et vérifier vos réponses et comprenez les warnings du compilateur et les erreurs d'exécution.

```
#include <stdio.h>
/* ca compile ou pas ?
faire attention aux warning,
c'est au programmeur de faire attention (en c)
*/
void f1( int t[10] ){int i;for(i=0;i<10;i++) printf("f1 %d ",t[i]);printf("\n");}

void f2( int t[20] ){int i;for(i=0;i<20;i++) printf("f2 %d ",t[i]);printf("\n");}

void f3( int t[] ){int i;for(i=0;i<20;i++) printf("f3 %d ",t[i]);printf("\n");}

void f4( int *t ){int i;for(i=0;i<20;i++) printf("f4 %d ",t[i]);printf("\n");}


int main(void){
    int tab1[10]={0};
    int tab2[5]={1,1,1,1,1};
    int i=0;

/* tout ça compile. Qu'est-ce cela donne à l'exécution ? */
printf("tab1\n");
    f1(tab1);    f2(tab1);    f3(tab1);    f4(tab1);
printf("ta21\n");
    f1(tab2);    f2(tab2);    f3(tab2);    f4(tab2);
printf("i\n");
    f1(&i);      f2(&i);      f3(&i);      f4(&i);


/* et meme ça compile, mais ca fait un warning, pourquoi ? On peut même l'exécuter */
printf("Scalaires\n");
```

```
    f1(2);       f2(2);       f3(2);        f4(2);

    return 0;
}
```

1.2 Programme 2

Même procédure qu'à l'exercice 1.

```
#include <stdio.h>

/* Ces déclarations de fontions sont-elles valides pour le compilateur ? */

void ftab(int t[], int i, int k){
    t[i] = 5;
}

void fvar1(int *i, int j){
    *i = j;
}


void fvar2(int i, int j){
    i = j;
}

/* Dans les trois cas ci-dessus, les modifications des valeurs t[i] et i sont-elles accessib

int main(void){
    int tab1[10]={0};
    int i=0, j=0, k=0;

    ftab(tab1, 2, 3);

    fvar1(&j, 2);
    fvar1(&j, tab1[2]);
/* warning */
    fvar1(4, 2);

    fvar2(j, k);

    /* warning ici */
    fvar2(&j, k);

    return 0;
}
```

2 Exercice 2 : Calcul de combinaisons

On se propose de calculer de façon optimale les combinaisons. On rappelle que : [1]

$$\binom{n}{p} = \frac{n!}{p! * (n-p)!} = \frac{n * (n-1) * (n-2) * \cdots * (n-p+1)}{p * (p-1) * (p-2) * \cdots * 2 * 1} \quad \text{pour } 0 \leq p \leq n$$

Pour éviter de faire des calculs qui dépassent la capacité de la machine, on peut imaginer de calculer à part le numérateur et le dénominateur et de simplifier la fraction correspondante en utilisant leur PGCD.

1. Ecrire une fonction renvoyant le PGCD de deux nombres entiers passés en paramètres. On utilisera l'algorithme itératif suivant pour calculer le pgcd de deux nombres a et b :
   $\{a, b \in N \wedge a > b\}$

   ```
   pgcd(a,b)
   {
     x <- a ;
     y <- b ;
     tant que (y>0) faire
      {
       aux <- y ;
       y <- x modulo y ;
       x <- aux ;
      }
   return x;
   }
   ```

   $\{(\exists c \exists d, a = x * c \wedge b = x * d) \wedge (\exists c' \exists d' \exists x', a = c' * x' \wedge b = d' * x' \rightarrow x' < x)\}$

2. Ecrire un programme de calcul de $\binom{n}{p}$ en utilisant la fonction précédente. Pour cela on utilisera deux variables Num et Den représentant le numérateur et le dénominateur de la fraction. On utilisera l'algorithme suivant :
   $\{n, p \in N \wedge n > p\}$

   ```
   {
   Num<-n;
   Den<-p;
   pour i=1 à p-1 faire
       {
       Num<-Num*(n-i);
       Den<-Den*(p-i);
       factCom<-pgcd(Num,Den);
       si factCom !=1
           alors {Num<-Num divEntière factCom;
                  Den<-Den divEntière factCom;}
   ```

1. La notation $\binom{n}{p}$ correspond à la notation française C_n^p.

```
    }
le résultat est Num;
}
```

$\{Num = \binom{n}{p}\}$

3. Comparer avec le calcul de $\binom{n}{p}$ utilisant la fonction factorielle en récursif et en itératif.

3 Exercice 3 - Implémentation d'une pile statique d'entiers

On se propose d'explorer le principe de la programmation modulaire. On veut créer un programme C implémentant une pile statique d'entiers. Ce programme peut par la suite être une boîte à outils permettant d'utiliser une pile dans un autre programme. Dans ce TDM, vous devez écrire de nombreux sous-programmes **en les testant impérativement au fur et à mesure de leur écriture, un par un.** C'est ce qu'on appelle des tests unitaires. De plus vous devez écrire les spécifications d'entrée et de sortie de chaque sous programme.

3.1 Implémentation d'une pile au moyen d'un tableau

3.1.1 Première étape

Cette étape rassemble les procédures dépendant du type des données stockées dans la pile.

1. Définir un type ELEMENT qui sera le type des éléments de la pile. Ce type sera pris équivalent à **int** pour simplifier, mais il pourrait être aussi un type enregistrement doté de nombreux champs,
2. Ecrire les procédures *Affiche_Elément* et *Saisit_Elément* permettant respectivement d'écrire un élément (type ELEMENT) à l'écran et de lire un élément au moyen du clavier. L'élément sera passé en paramètre,
3. Ecrire une procédure *Affecte_Elément* qui reçoit en paramètre deux variables de type ELEMENT et qui affecte la première variable avec la valeur de la deuxième. Si le type ELEMENT était un enregistrement la procédure devrait copier tous les champs.

3.1.2 Deuxième étape

Dans cette partie on implémente la structure de donnée PILE (LIFO en anglais : Last In First Out). On choisit une implémentation statique utilisant un tableau de taille fixée (TMAX déclarée au moyen d'un **#define**). Les éléments sont ajoutés dans le sens des indices croissants du tableau. L'élément en tête de pile est repéré par son indice mémorisé par une variable (Tête). Insérer un

élément c'est incrémenter la tête et mettre le nouvel élément dans la case d'indice tête. Retirer un élément consiste à décrémenter la tête (il n'est pas nécessaire de supprimer physiquement l'élément).

1. Définir le type PILE sous forme d'une structure ayant deux champs : le tableau et la tête. Les éléments du tableaux sont de type ELEMENT,
2. Définir la procédure *Initialise_Pile* qui initialise une pile,
3. Définir la procédure *Affiche_Pile* qui écrit à l'écran tous les éléments d'une pile. Cette procédure utilisera *Affiche_Elément* et est surtout destinée au débogage,
4. Définir la fonction *Pile_Pleine* qui teste si une pile est pleine. Pour clarifier la programmation on utilisera les constantes définies dans *<stdbool.h>* : *true* et *false* ainsi que le type *bool*,
5. Définir la procédure *Empile* qui reçoit un élément et une pile et ajoute l'élément dans la pile,
6. A ce stade on peut faire un premier test d'intégration, c'est à dire tester l'intégration de plusieurs sous-programmes dans un même *main*. Ecire donc un *main* qui définit une pile, l'initialise, appelle *Empile* plusieurs fois et contrôle le résultat en appelant *Affiche_Pile*,
7. Définir la fonction *Pile_Vide* qui teste si une pile est vide. On utilisera comme précédemment les constantes définies dans *<stdbool.h>*,
8. Définir la fonction ou procédure *Sommet_Pile* qui renvoie l'élément sommet de la pile sans le détruire.
9. Définir la procédure *Dépile* qui supprime la tête de la pile,
10. A ce stade ajouter, par commodité, la procédure *Saisit_Pile* qui demande à l'utilisateur des éléments et les insère dans la pile. Cette procédure utilisera *Saisir_Elément*,
11. Faire un *main* plus ambitieux qui teste l'ensemble des opérations sur une pile. On peut appeler *Affiche_Pile* après chaque opération pour contrôler l'intégrité de la pile.

3.2 Modularité

Nous allons maintenant aborder le principe de la programmation modulaire.

1. Sauver toutes les procédures de la première étape dans un fichier *élémentpile.c*.
2. Mettre le type ELEMENT dans un fichier *pile.h* ainsi que tous les **#define** et les types définis.
3. Ajouter à *élémentpile.c* la ligne **#include "pile.h"**.
4. Créer maintenant un fichier *pile.c* contenant les procédures de la deuxième étape et utilisant les procédures de *élémentpile.c*. Ce fichier devra aussi inclure *pile.h*.
5. Il nous faut un dernier fichier *main.c* pour le programme principal.

Pour compiler ce programme constitué de plusieurs parties on peut utiliser : **gcc main.c pile.c élémentpile.c -o pile**.

Ou encore une suite de commandes de compilation indépendantes (en apparence) les unes des autres :

```
gcc -c élémentpile.c
gcc -c pile.c
gcc -c main.c
gcc élémentpile.o pile.o main.o -o pile
```

L'intérêt de cette façon de programmer est de pouvoir organiser le programme en parties distinctes, souvent appelées modules, permettant un meilleur repérage du rôle de ces parties, des fichiers où chercher un bug ou bien impliqués dans une modification. Ainsi, un changement du type ELEMENT dans votre programmation ne nécessite la recompilation que du module *élémentpile.c* (il faut refaire tout de même l'édition de lien mais gcc s'en charge).

1. Changer le type ELEMENT pour :

   ```
   typedef struct {char Nom[10];
                   char Prenom[10];
                   int Age;} ELEMENT;
   ```

2. Modifier en conséquense les procédures *Saisit_Elément*, *Affiche_Elément*et *Affecte_Elément*.
3. Recompiler *élémentpile.c* et taper

   ```
   gcc élémentpile.o pile.o main.o -o pile
   ```

4. Exécuter le programme, ça doit marcher sans bavure.

Résumé : On a voulu changé le type ELEMENT.

1. on savait précisément quoi changer, et où se trouvait le code concerné (module et fichiers),
2. on n'a rien changé au reste,
3. on ne recompile que ce qui est nécessaire,
4. on appelle gcc pour qu'il fasse l'édition des .o (code objet)

On peut automatiser d'avantage certaines opérations (makefile). Cela sera vu en détail au S4.