Informatique de base-II

9 septembre 2014

Note Dans ce qui suit nous proposons un programme toujours structuré de la même manière (cependant certains éléments ne sont pas toujours présents) : #include, #define, types, prototypes de fonctions, fonction principale, et enfin définitions de fonctions. Cet ordre doit être considéré comme obligatoire. De plus le programme contient des commentaires (/* ... */ (ou simplement //)) et à la fin un exemple d'exécution sous la forme également d'un commentaire.

1 Enregistrements

Un enregistrement est un type de donnée structurée, comme un tableau, mais dans le cas d'un enregistrement, l'accès à une partie de l'enregistrement se fait par le nom du *champ* correspondant, et non par un indice entier. En langage C les enregistrements s'appellent des structures.

1.1 Les rationnels

La syntaxe est illustrée sur l'exemple de programme ci-dessous où un nombre rationnel est représenté par une structure dont les champs `num` et `den` représentent le numérateur et le dénominateur d'une écriture en fraction du rationnel [1].

```
/* ratEssai.c */
#include <stdio.h>
#include <stdlib.h>
/* type */
struct rat{
    int num;
    int den;
};
/*fonction principale*/
int main(void){
    /* rationnels    */
        struct rat r, rinv;
        printf("numérateur et dénominateur du rationnel r? ");
        scanf("%d",&r.num);
        scanf("%d",&r.den);
```

1. Le résultat de l'exécution est donné en commentaire à la fin du programme.

```
        rinv.num=r.den;
        rinv.den=r.num;
        printf("l'inverse de r est (%d/%d) \n", rinv.num, rinv.den);
        return EXIT_SUCCESS;
}
//dyn246:dufee ratEssai
//numérateur et dénominateur du rationnel r ? 1 2
//l'inverse de r est (2/1)
```

Ci-dessus `struct rat { ...};` déclare un type `struct rat` qui est ensuite utilisé, comme tout type, pour déclarer des variables, ici `r` et `rinv`. `r.num` indique l'emplacement où est rangée la valeur du champ `num` de la variable `r`. Ainsi, par exemple, on doit donner l'adresse du champ `num` pour lire la valeur de ce champ sur l'entrée standard par `scanf`. Remarquons qu'uns structure est différente d'un tableau sur un point très important : `r` représente le rationnel lui-même et non son adresse. En conséquence lorsqu'on transmet une variable de type structure comme argument, c'est la *valeur* qui est transmise et non l'*adresse* de la variable. De même une fonction peut renvoyer une variable de type structure, c'est-à-dire sa valeur. Voici le même programme que précédemment, dans lequel on utilise une fonction pour lire la variable `r`.

```
/* ratEssaiF.c */
#include <stdio.h>
#include <stdlib.h>

/* type */
struct rat{
    int num;
    int den;
};
/* prototype */
struct rat    lit_rat (void);

/*fonction principale*/
int main(void){
    /* rationnels   */
        struct rat r, rinv;;
        printf("numérateur et dénominateur ? ");
        scanf("%d",&r.num);
        scanf("%d",&r.den);
        rinv.num=r.den;
        rinv.den=r.num;
        printf("l'inverse de r est (%d/%d) \n", rinv.num, rinv.den);
        return EXIT_SUCCESS;
}
struct rat    lit_rat (void){
    /* lit la valeur d'un rationnel et renvoie cette valeur */
        struct rat r;
        printf("num et den ? ");
        scanf("%d",&r.num);
```

```
        scanf("%d",&r.den);
        return r;
}
//dyn246:dufee ratEssaiF
//numérateur et dénominateur ? 1 2
//l'inverse de r est (2/1)
```

1.2 Représentation d'une matrice dans une structure

Nous reprenons ici notre programme de calcul matriciel en changeant la représentation des matrices : une matrice n'est plus représentée par 3 objets, un tableau et deux dimensions, mais par un seul objet, une structure. Voici le nouveau programme obtenu, en faisant le choix d'avoir les résultats des opérations comme valeur de retour.

```
/*
 *  calcul matriciel avec des structures
 */
#include <stdio.h>
#include <stdlib.h>
#define NL 100
#define NC 100

/* types */

struct Matrice{
        int t[NL][NC];
        int nl;
        int nc;
} ;

typedef struct Matrice matrice_t;

/* prototypes */

// primitives de calcul matriciel avec structures

matrice_t lireMatrice(int nl, int nc);
void afficheMatrice(matrice_t m);
matrice_t produitMatrices(matrice_t m1, matrice_t m2);
matrice_t transpose(matrice_t m);

/* fonction principale */

int main () {
//    primitives de calcul matriciel avec structures
        int nl1, nc1;
        matrice_t m1,m2,mr;
```

```
        //définit les dimensions de m1, lit m1 et affiche m1
        scanf("%d",&nl1);
        scanf("%d",&nc1);
        m1=lireMatrice(nl1, nc1);
        afficheMatrice(m1);
        // transposition de m1 dans mr
        // et affichage
        mr=transpose(m1);
        afficheMatrice(mr);
        // produit de m1 et de sa transposée mr dans m2
        // et affichage
        m2=produitMatrices(m1,mr);
        afficheMatrice(m2);
        return 0;
}

/* fonctions */

matrice_t lireMatrice(int nl, int nc){
        /* lit et renvoie une matrice de dimensions nl nc */
        int i,j;
        matrice_t m;
        m.nl=nl;
        m.nc=nc;
        for (i=0;i<m.nl;i++){
                for (j=0;j<m.nc;j++){
                        scanf("%d",&m.t[i][j]);
                }
        }
        return m;
}

void afficheMatrice(matrice_t m){
        /* affiche la matrice m */
        int i,j;
        for (i=0;i<m.nl;i++){
                for (j=0;j<m.nc;j++){
                        printf("%d ",m.t[i][j]);
                }
                printf("\n");
        }
        return;

}

matrice_t produitMatrices(matrice_t m1, matrice_t m2){
        /* renvoie le produit de deux matrices m1 et m2 */
        int i,j,k;
        matrice_t mr;
        mr.nl=m1.nl;
```

```
        mr.nc=m2.nc;
        for (i=0;i<mr.nl;i++){
                        for (j=0;j<mr.nc;j++){
                        mr.t[i][j]=0;
                        for(k=0;k<m1.nc;k=k+1){
                                        mr.t[i][j]= mr.t[i][j]+ m1.t[i][k]*m2.t[k][j];
                        }
                        }
        }
        return mr;
}

matrice_t transpose(matrice_t m) {
        /*      renvoie la transposee d'une matrice m  */
        int i,j;
        matrice_t mr;
        mr.nl=m.nc;
        mr.nc=m.nl;
        for (i=0;i<m.nl;i++){
                for (j=0;j<m.nc;j++){
                        mr.t[j][i]=m.t[i][j];
                }
        }
        return mr;
}



/*
 dyn251:Alternance-Info de base soldano cat calc.in2 32 3 13 5 7dyn251 :Alternance-
Info de base soldano ./calculMatricielAIR1-tp3-2014 < calc.in
 2 3 1
 3 5 7
 2 3
 3 5
 1 7
 14 28
 28 83
*/
```

2 Allocation dynamique de mémoire

2.1 Allocation dynamique d'un tableau

Nous avons vu précédemment que dans un programme, un objet est associé à une adresse dans la mémoire : l'adresse du début de la représentation de l'objet. Pour une variable entière `n`, `&n` est la valeur de cette adresse (disons $a28$). Les valeurs de type `adresse` peuvent être rangées dans des variables particulières

appelées *pointeurs*, par exemple `int *an;` pour définir un *pointeur* `an` *vers un entier*.

Nous avons également vu que l'on pouvait accéder ou modifier les objets par leur adresse : ainsi `*an` est l'objet à l'adresse `an`, et en conséquence `&(*an)` a la même valeur (une adresse) que `an`. De même `*(&n)` a la même valeur (un entier) que `n`. Ainsi après `an=&n;(*an)=1;` l'objet `(*an)`, c'est-à-dire la variable `n`, a la valeur 1. Précisément `an=&n;` a eu pour effet de mettre l'adresse de `n`, c'est-à-dire $a28$, dans `an`.

Jusqu'à présent cependant les objets étaient définis de manière *statique*, c'est-à-dire par une définition comme `int n;` dans le texte du programme en C. Précisément cela signifie que la place occupée par cet objet est réservée lors de la compilation (par `gcc`). L'inconvénient est qu'il faut connaître à l'avance la place dont on a besoin, ce qui explique la nécessité de définir la taille d'un tableau par une constante (par exemple `int t[100];` ou `int t[N]`, `N` étant une constante définie pendant la précompilation par `#define N 100`).

Il est cependant utile de pouvoir réserver de la place pour des objets de manière *dynamique*, c'est-à-dire *pendant* l'exécution du programme. Pour cela on utilisera l'instruction `malloc`, dont voici un exemple d'usage :

```
/*
 *  allocationDynamique.c
 */
#include <stdio.h>
#include <stdlib.h>

/* Fonction principale */
int main(void){
        int i,n;
        float s;
        float *t, *t2; // la valeur de t doit être une adresse de type float
        /* allocation dynamique de la mémoire associée au tableau dont l'adresse
         de début (celle du premier élément t[0]) est t */
        printf(" taille du tableau ? (>1) ");
        scanf("%d",&n);
        t= (float *) malloc (n* sizeof(float)); //allocation 1
        for(i=0; i<n;i=i+1){
                t[i]=(1+(i*2.0))/n;
        }
        /* calcul de la somme de 0 à n-1 de (1+(i*2) / n ) (somme des éléments du tableau)*/
        s=0;
        for(i=0; i<n;i=i+1){
                s=s+t[i];
        }
        printf(" somme de 0 à %d de (1+(i*2)) / %d ) = %f\n", n-1,n, s );
        /* restitution de la mémoire allouée à t */
        free(t);
        /* allocation dynamique de la mémoire associée au tableau dont l'adresse
         de début (celle du premier élément t2[0]) est t2 */
        printf(" taille du tableau ? (>1) ");
```

```
        scanf("%d",&n);
        t2= (float *) malloc (n* sizeof(float));
        for(i=0; i<n;i=i+1){
                t2[i]=(1+(i*2.0))/n;
        }
        /* calcul de la somme de 0 à n-1 de (1+(i*3) / n ) (somme des éléments du tableau)*/
        s=0;
        for(i=0; i<n;i=i+1){
                s=s+t2[i];
        }
        printf(" somme de 0 à %d de (1+(i*3)) / %d ) = %f\n", n-1,n, s );
        free(t2);
        return EXIT_SUCCESS;
}

//dyn246:dufee soldano gcc allocationDynamique.c -o allocationDynamique
//dyn246:dufee soldano ./allocationDynamique
// taille du tableau ? (>1) 3
//somme de 0 à 2 de (1+(i*2)) / 3 ) = 3.000000
// taille du tableau ? (>1) 5
//somme de 0 à 4 de (1+(i*3)) / 5 ) = 5.000000
//dyn246:dufee soldano
```

Quelques remarques :

- Dans l'exécution ci-dessus, nous avons demandé l'allocation d'un tableau `t` de 3 éléments.
- Pour cela, nous avons demandé par l'appel `malloc (n* sizeof(float))` la réservation d'un bloc de mémoire de `n` fois la taille de la représentation d'un `float`, et avons rangé le résultat, c'est-à-dire l'adresse du début de ce bloc réservé (par exemple $a32$) dans la variable `t`. De plus nous avons précisé par `(float *)` que la valeur de retour de l'appel `malloc (n* sizeof(float))` est de type *adresse d'un float* (voir *// allocation 1*).
- A la fin de la première partie nous avons restitué la mémoire allouée à `t` par `free(t)`. `t` ne représente alors plus un tableau, seulement un pointeur sur un objet de type `float`, mais à cette adresse il n'y a pas de mémoire réservée, et des instructions comme `t[0]=1.0;` ou `(*t)=1.0;` provoqueraient une erreur.
- Nous avons ensuite demandé l'allocation d'un tableau `t2` de 5 éléments.
- A la fin de la seconde partie nous avons restitué la mémoire allouée à `t2`.

2.2 Allocation dynamique d'un tableau dans une structure : les matrices revisitées -II-

Ci-dessous, nous reprenons la représentation des matrices représentées dans des structures. Nous voulons allouer dynamiquement la mémoire au tableau représentant la matrice. Plus précisément, dans la structure `m`, le tableau `m.tab` aura exactement la taille nécessaire pour représenter les `m.nl` * `m.nc` éléments de

la matrice. Pour cela, sachant qu'une matrice a `nl` lignes et `nl` colonnes, on utilisera une fonction `consMatrice` qui effectuera l'allocation. Cette fonction sera utilisée à chaque fois que l'on devra représenter une nouvelle matrice. L'allocation se passe de la manière suivante : on alloue d'abord le tableau de pointeurs `m.tab` de dimension `m.nl`. Chacun de ces pointeurs (des adresses d'entiers) est destiné à pointer vers une ligne, c'est à dire un tableau de `m.nc` éléments.

```
/*
  calcul matriciel avec structures et allocation dynamique du tableau
 */
#include <stdio.h>
#include <stdlib.h>



/* types */

struct Matrice{
        int **t;
        int nl;
        int nc;
} ;

typedef struct Matrice matrice_t;

/* prototypes */

// primitives de calcul matriciel avec structures
matrice_t construitMatrice(int nl, int nc);
matrice_t lireMatrice(int nl, int nc);
void afficheMatrice(matrice_t m);
matrice_t produitMatrices(matrice_t m1, matrice_t m2);
matrice_t transpose(matrice_t m);

/* fonction principale */

int main () {
//    primitives de calcul matriciel avec structures
        int nl1, nc1;
        matrice_t m1,m2,mr;

        //définit les dimensions de m1, lit m1 et affiche m1
        scanf("%d",&nl1);
        scanf("%d",&nc1);
        m1=lireMatrice(nl1, nc1);
        afficheMatrice(m1);
        // transposition de m1 dans mr
        // et affichage
         mr=transpose(m1);
        afficheMatrice(mr);
```

```
        // produit de m1 et de sa transposée mr dans m2
        // et affichage
        m2=produitMatrices(m1,mr);
        afficheMatrice(m2);
        return 0;
}

/* fonctions */

 matrice_t construitMatrice(int nl, int nc)
{
        /* construit une matrice de dimensions nl nc */
        int i,j;
        matrice_t m;

        m.nl=nl;
        m.nc=nc;
        m.t=(int **) malloc(m.nl*sizeof(int*));//alloue un tableau de nl tableaux

        for (i=0;i<m.nl;i++){
                m.t[i]= (int *) malloc (m.nc* sizeof (int));
        }


        return m;
}
matrice_t lireMatrice(int nl, int nc){
        /* lit et renvoie une matrice de dimensions nl nc */
        int i,j;
        matrice_t m;

        m=construitMatrice(nl,nc);

        for (i=0;i<m.nl;i++){
                for (j=0;j<m.nc;j++){
                        scanf("%d",&m.t[i][j]);
                }
        }
        return m;
}

void afficheMatrice(matrice_t m){
    /* affiche la matrice m */
        int i,j;
        for (i=0;i<m.nl;i++){
                for (j=0;j<m.nc;j++){
                        printf("%d ",m.t[i][j]);
                }
                printf("\n");
        }
```

```
        return;

}


matrice_t produitMatrices(matrice_t m1, matrice_t m2){
    /* renvoie le produit de deux matrices m1 et m2 */
        int i,j,k;
        matrice_t mr;
        mr=construitMatrice(m1.nl,m2.nc);
        for (i=0;i<mr.nl;i++){
                for (j=0;j<mr.nc;j++){
                        mr.t[i][j]=0;
                        for(k=0;k<m1.nc;k=k+1){
                                mr.t[i][j]= mr.t[i][j]+ m1.t[i][k]*m2.t[k][j];
                        }
                }
        }
        return mr;
}

matrice_t transpose(matrice_t m) {
        /*      renvoie la transposee d'une matrice m   */
        int i,j;
        matrice_t mr;
        mr=construitMatrice(m.nc,m.nl);
        for (i=0;i<mr.nl;i++){
                for (j=0;j<mr.nc;j++){
                        mr.t[i][j]=m.t[j][i];
                }
        }
        return mr;
}



/*
 dyn251:Alternance-Info de base soldano cat calc.in2 32 3 13 5 7dyn251 :matrices
soldano ./calculMatricielAIR1-tp4-2014 <calc.in
 2 3 1
 3 5 7
 2 3
 3 5
 1 7
 14 28
 28 83

*/
```

Ce qu'il est important de comprendre ici est ce qu'il se passe lorsque on utilise `construitMatrice`. A l'appel de cette fonction pendant l'exécution de `lireMatrice`, on range dans la structure `m` la valeur de la structure renvoyée par `construitMatrice` et dont le premier champ est une adresse de début de tableau allouée au programme. Le résultat est que maintenant `m` est utilisable pour ranger une matrice. Remarquons que le programme est très proche de celui présenté au paragraphe 1.2 où le tableau était alloué statiquement.

3 Structuration des sources d'un programme et compilation séparée

3.1 Structuration des sources

Lorsqu'on utilise un ensemble de fonctions, se rapportant par exemple à un ou plusieurs types particuliers, on représente

- dans un fichier d'*en-têtes*, dont l'extension est `.h`, par exemple `fg.h`, l'ensemble des déclarations se rapportant à ces types et fonctions, c'est-à-dire les types, les prototypes des fonctions, ainsi que les constantes propres à ces types et fonctions. Remarquons que par déclaration on entend ici des informations pour le compilateur mais en aucun cas des instructions à exécuter, y compris des définitions de variables. Par exemple écrire `int x;` est une instruction allouant la mémoire nécessaire à la représentation d'un entier dont l'identificateur est `x`, : une telle instruction ne figurera pas dans un fichier d'*en-têtes*.
- Dans un fichier d'extension `.c`, par exemple `fg.c`, l'ensemble des définitions des fonctions. Ce fichier contiendra une instruction `#include` fg.h" permettant d'inclure le fichier d'en-têtes `fg.h`.

On pourra alors utiliser ces fichiers pour simplifier l'écriture de programmes utilisant cet ensemble de types et de fonctions. En voici un exemple simple, dans lequel aucun nouveau type n'est défini, mais ou on définit deux fonctions `f` et `g` dépendant d'une constante `M`. Ces définitions et déclarations sont utilisées ensuite dans un fichier contenant la fonction principale `main`.

`--------fg.h--------`

```
/* declarations */
/* constantes */
# define M 5
/* prototypes */
int f (int n); // ajoute M
int g (int n); // multiplie par M
```

`--------fg.c--------`

```
/* déclarations prises dans le fichier en-tête correspondant */
#include "fg.h"
/* definitions */
int f (int n){ //ajoute M
```

```
        return (n + M);
}
int g (int n){// multiplie par M
        return (n * M);
}
```

`--------essai2.c--------`

```
/* essai2.c */
/* déclarations prises dans les fichiers en-têtes */
# include <stdio.h>
# include <stdlib.h>
# include "fg.h"
/* fonction principale */
int main(void){
        printf("(%d*%d)+%d = %d \n", 100, M, M, f(g(100)));
        return EXIT_SUCCESS;
}
```

Quel est le rôle exact des instructions `# include` et `# define` dans l'écriture des fichiers sources `.h` et `.c` ?. Pour le comprendre, il faut aborder les différentes étapes de la compilation.

3.2 Compilation séparée

Pour compiler et executer le programme précédent on utilise les commandes suivantes :

```
dyn246:dufee gcc essai2.c fg.c -o essai2
dyn246:dufee ./essai2
(100*5)+5 = 505
```

La première commande `gcc` effectue les opérations suivantes :

1. Précompilation : les deux fichiers `essai2.c` et `fg.c` sont chacun précompilés c'est-à-dire chacun traduit en un fichier d'extension `.i`. `essai2.i` et `fg.i` contiennent le même texte que `essai2.c` et `fg.c`, mais dans lesquels les instructions pour le précompilateur, celles commençant par `#`, ont été exécutées :
   - l'instruction `# include <stdio.h>` est remplacée dans le texte par le contenu du fichier `stdio.h`. Les symboles `<>` signifient que `stdio.h` sera cherché dans un répertoire particulier connu par `gcc`.
   - l'instruction `# include fg.h` est remplacée dans le texte par le contenu du fichier `fg.h`. Les symboles `""` signifient que `fg.h` est l'adresse où se trouve le fichier, c'est-à-dire le répertoire courant.
   - l'instruction `# define M 5` remplace dans tout le texte `M` par `5`.

   Il faut remarquer que le même fichier `fg.h` est inclus dans `essai2.i`et `fg.i`, ce qui permet d'assure la cohérence de la programmation : si on change `fg.c`, par exemple en ajoutant une fonction, on changera aussi

`fg.h`, et ce changement sera transmis à `fg.i` et `essai2.i` par une nouvelle compilation.

2. Compilation : les deux fichiers `essai2.i` et `fg.i` sont traduits en deux fichiers *objets* `essai2.o` et `fg.o` contenant chacun des instructions exécutables par la machine.
3. Edition de liens : les références communes aux deux fichiers `essai2.o` et `fg.o` sont résolues : si deux fonctions d'identificateur `f` sont présentes dans ces fichiers objets, l'éditeur de lien remarque qu'il s'agit du même objet. Le résultat de l'édition de lien est le fichier exécutable `essai2`.