OUAHIOUNE Yidir
HOUDROUGE Léo **L1-D**

Rapport de projet

Sommaire

Introduction

Pour la fin de notre première année de prépa intégrée, nous nous sommes vu dans le devoir d'écrire en langage C un programme permettant de gérer un polynôme. La consigne du projet nous demandait d'écrire le polynôme via deux méthodes différentes, la première méthode voulait que l'on utilise un tableau de structures et la deuxième voulait que l'on utilise une liste chaînée.
La première partie du projet ,sur l'initialisation des polynômes avec une structure, nous a donc beaucoup aidé pour la deuxième partie sur les listes chaînées, pour nous plus dure. Puisque plusieurs méthodes étaient demandées, ce projet nous a permis de mettre en œuvre toutes nos connaissances acquises au cours de l'année.
Nous avons donc fait le projet dans l'ordre, dont la priorité a été de programmer la fonction « saisie », cœur du projet, elle sert notamment à répondre à beaucoup de points demandés dans le cahier des charges.
Le projet est à rendre avant le 23 juin, nous l'avons commencé le 10 juin, pour cela nous avons utilisé Clion afin de programmer à deux simultanément et pour le partage des différents fichiers. De plus, nous n'avons pas pu utiliser Valgrind, logiciel pour les fuites mémoires, car nous ne possédions pas Linux.
Dans notre projet, parmi toutes les fonctions qui nous étaient demandées, la fonction de saisie était le moteur de notre programme, et a influencé le modèle de toutes les autres fonctions (somme, produit, dérivé…etc).
De là, une question se pose : Comment avons-nous élaboré toutes les fonctions qui constituent ce programme et quelles ont été ses limites?

Quel est le but des différentes fonctions ?

Puisque le traitement des polynômes est en deux parties, nous avons mis nos fonctions dans deux fichiers différents en fonction des parties abordées. Toutes les fonctions pour traiter le polynôme via un tableau de structures se retrouve dans le fichier "*fonctions.c*" ainsi que celle pour le traiter via une liste chaînée sont dans le fichier "*fonction_LC.c*".

Cependant chacunes des fonctions ont été conçues sur le même principe. Peu importe la partie, les fonctions agissent plus ou moins pareil.

La fonction d'initialisation sert à transformer un pointeur polynôme en un polynôme dont la valeur est nulle, c'est à dire qu'il n'est constitué que d'un seul monôme, et que son coefficient est son degré sont tous deux égaux à zéro.
Si l'on affiche un polynôme après la fonction initialisation, le terminal montre (0.0,0)

La fonction saisir quant à elle est là pour ajouter un monôme dans un pointeur polynôme renseigné en paramètre de la fonction. Les valeurs du monômes à ajouter sont elles aussi en paramètre.
Cette fonction est optimisée de façon à fournir un monôme réduit. Pour ce faire, elle additionne les coefficients de deux monômes de même degré et range le polynôme dans l'ordre décroissant selon les degrés des monômes.

Elle détermine aussi la taille du polynôme et son plus haut degré. Cette fonction a été la plus longue à coder autant dans la première partie que dans la deuxième.

Comme son nom l'indique, la fonction somme fait l'addition de deux polynômes. Elle agit en ajoutant dans un polynôme nommé "somme", tous les monômes du premier polynôme mis en paramètre via la fonction saisir, puis elle fait de même avec le second polynôme lui aussi mis en paramètre.

La fonction produit suit un principe similaire à la fonction somme. Elle crée un nouveau polynôme produit puis avec la fonction Saisir, elle réduit le polynôme qui résulte du produit de p1 et p2 et le met dans le polynôme produit. Enfin elle retourne ce polynôme.

Pour l'affichage, la fonction parcours le tableau de monômes ou la liste chaînée et affiche chacun des monômes de cette façon : (coefficient, degré)
Par exemple, 3x^2 + 2 s'affiche (3, 2) (2, 0).

La fonction de calcul additionne dans une variable les valeurs de chaque monôme selon la valeur de x mis en paramètre puis elle retourne cette variable.

La fonction de dérivation ajoute la dérivée de chacun des monômes du polynôme mis en paramètre dans un autre polynôme puis elle retourne ce dernier polynôme.

Pour la dérivé de la somme de deux polynômes, la fonction fait la dérivée des deux polynômes mis en paramètre via la fonction de dérivation puis elle retourne la somme des deux via la fonction somme.

Pour la dérivé du produit de deux polynômes, la fonction fait la dérivée des deux polynômes mis en paramètre via la fonction dérivée. Puis elle retourne la somme du produit de la dérivée du premier polynôme avec le deuxième et du produit de la dérivée du deuxième polynôme avec le premier.

La fonction qui fait la primitive d'un polynôme marche sur le même modèle que la fonction de dérivation. La seule différence est qu'elle ajoute la primitive de chaque monôme à la place de la dérivée.

Quels ont été les différents procédés utilisés ?

Notre projet a donc débuté par la partie avec la liste de structure.Toutes les fonctions de la première partie du projet se situent dans le fichier fonctions.c et les fonctions sont initialisées dans le fichier fonctions.h.

Pour commencer nous avons effectué la fonction initialisation, elle a un double pointeur en paramètre ce qui nous permet d'appliquer directement des changements sur le pointeur polynôme *p.

On part du principe que le pointeur à initialiser est un pointeur NULL. L'utilisation du malloc dans la première ligne du programme nous permet d'attribuer directement de la mémoire dans le tas, c'est de l'allocation de mémoire, cela permet au pointeur *p d'agir comme une structure.

En plus du pointeur polynôme, il faut allouer de la mémoire au pointeur s'appelant "monômes" afin qu'il agisse comme un tableau de structures monôme. Avant de l'allouer, on initialise le nombre de monômes à 1 puisque le but de la fonction est d'initialiser le polynome à un seul monome (0.0, 0).

```
(*p)->n = 1;
(*p)->monomes = malloc((*p)->n*sizeof(monome));
```

(initialisation du nombre de monôme et de la taille du tableau)

Une fois tout cela fait, la fonction peut enfin initialiser le polynome au monome (0.0, 0).

```
(*p)->monomes[0].degre = 0;
(*p)->monomes[0].coefficient = 0;

(*p)->degre = 0;
```

Afin de vérifier si l'allocation s'est bien déroulée, on retourne 0 si le pointeur est NULL et 1 si il a bien été initialisé.

La fonction de saisie est la plus importante du projet. Elle vérifie d'abord si le pointeur polynôme renseigné est NULL et si c'est le cas elle l'initialise.

Ensuite elle parcourt le tableau pour savoir si le degré renseigné est égale au degré d'un des monômes du polynôme. Si c'est le cas alors elle additionne le coefficient renseigné en paramètre avec celui de même degré. Cela nous permet d'avoir un polynôme réduit sans avoir plusieurs fois les mêmes degrés.

```
int test = 0;
for (int i=0; i < (*p)->n; i++)
{
    if ((*p)->monomes[i].degre == degre)
    {
        (*p)->monomes[i].coefficient += coeff;
        test = 1;
    }
}
```

L'entier test permet de savoir si la condition a été remplie. Si ce n'est pas le cas, alors la fonction réalloue la taille du tableau de monômes et ajoute le monôme renseigné en paramètre dans le tableau.

```
if(test == 0)
{
    (*p)->n += 1;
    monome* tp = NULL;
    tp = realloc((*p)->monomes, (*p)->n*sizeof(monome));
    (*p)->monomes = tp;

    if((*p)->monomes == NULL)
    {exit(EXIT_FAILURE);}
```

(réallocation du tableau + sécurité si l'allocation a échoué)

(ajout du nouveau monôme à la fin du tableau seulement si l'allocation n'as pas échoué)

```
if((*p)->monomes == NULL)
{exit(EXIT_FAILURE);}
else
{
    (*p)->monomes[(*p)->n - 1].degre = degre;
    (*p)->monomes[(*p)->n - 1].coefficient = coeff;
}
```

L'avant dernière fonctionnalité de cette fonction est de trier le tableau dans l'ordre décroissant selon le degré des monômes. Cela fonctionne avec une double boucle et un if qui check si le degré du monôme est inférieur au degré du monôme suivant. Si c'est le cas alors elle swap les deux monômes entre eux.

Pour finir, une dernière boucle traverse le tableau et repère le monôme de degrés le plus important. Elle attribue ensuite le degré en question au degré du polynôme même si cette valeur ne nous sert à rien.

La fonction de somme va retourner un pointeur polynôme appelé "somme" dans lequel on aura ajouter tous les monômes du premier pointeur en paramètre. On le fait via une boucle et via la fonction saisir puis on fait de même avec le deuxième pointeur..

```
for(int i = 0; i < p1->n; i++)
{
    saisir(&somme, p1->monomes[i].coefficient, p1->monomes[i].degre);
}
```

(la boucle utilisé pour les deux pointeurs)

Puisque la fonction saisir additionne déjà les monômes de même degré, il n'y a pas besoin de faire quoi que ce soit d'autre.

Pour le produit, la fonction va elle aussi retourner un pointeur polynôme appelé "produit".

Par exemple, si en paramètre on a un polynôme p1 et un polynôme p2, alors la fonction va multiplier le premier monôme de p1 avec le premier monôme de p2, puis le deuxième de p2 jusqu'au dernier monôme de p2. Ensuite elle va re-multiplier chacun des monômes de p2 avec le monôme suivant de p1 et ainsi de suite jusqu'au dernier monôme de p1.

Enfin elle va saisir le résultat de toutes ces multiplications de monômes dans le polynôme produit via la fonction saisir et une double boucle. En conclusion, l'action de cette double boucle est de réduire le polynôme qui résulte du produit des deux polynômes mis en paramètre.

La fonction de calcul retourne une variable double nommée résultat. Dans la fonction on initialise la variable à zéro puis on lui additionne,via une boucle, la valeurs de chacun des monômes du polynôme en paramètre pour une certaine valeur de x.

Dans cette boucle on multiplie le coefficient du monôme par la valeur de x en fonction du degré du monôme.

Pour la fonction dérivation, comme expliqué précédemment, on ajoute la dérivée de chacun des monômes du polynôme en paramètre dans un polynôme derive que l'on retourne à la fin.

```
if(p->monomes[i].degre == 0)
{
    saisir(&derive, coeff: 0, degre: 0);
}else
```

Il y a cependant une condition pour les monômes au degré zéro. Si le degré du monôme est égal à zéro alors il ajoute dans le polynôme derive un monôme nul.

Sans cette condition il aurait ajouté un monôme dont le degré est -1 ce qui est faux.

```
}else
{
    saisir(&derive, coeff: p->monomes[i].coefficient * p->monomes[i].degre, degre: p->monomes[i].degre - 1);
}
```

Maintenant en ce qui concerne les liste chaînée, toutes les fonctions de la deuxième partie du projet se situent dans le fichier "fonction_LC.c" et les fonctions sont initialisées dans le fichier "fonction_LC.h".

```
typedef struct Monome{
    double coefficient;
    int degre;
    struct Monome * suivant;
}monome;

typedef struct Polynome{
    struct Monome * monomes;
    int n;
}polynome;
```

Nos deux structures se présentent ainsi et la liste chaînée de monômes se trouve dans la structure polynôme.

La plupart des fonctions marchent avec le même principe que pour les tableaux de monômes avec comme seul différence la façon de faire des boucles pour parcourir un polynôme.
Au lieu de faire des boucles for on fait des boucles while et avant chaque boucle on fait une copie de la liste chaînée afin de ne pas perdre les nœuds précédent.

Néanmoins la fonction de saisie reste différente de celle qui fonctionne avec le tableau de monômes. Elle garde quand même le même principe que la précédente c'est à dire qu'elle ajoute le monôme renseigné en paramètre au pointeur polynôme lui aussi renseigné en paramètre puis range le polynôme. Cette fois la fonction retourne un polynôme.

Elle initialise le pointeur polynôme renseigné s'il est NULL.

Ensuite, elle crée une copie de la liste chainée de monome afin de parcourir cette dernière sans perdre de nœuds.

```
monome* copie = NULL;

copie = p->monomes;
```

Puis pour ranger la liste chaînée dans l'ordre décroissant selon les degrés, on part du principe qu'elle est déjà rangé (de toute façon elle le sera toujours car la seule façon d'ajouter des monômes est de passer par cette fonction) et on utilise donc des conditions permettant de ranger le nouveau monôme en fonction des monômes déjà présent dans la liste.

Si le degré renseigné est supérieur au degré du monôme en tête de liste, alors on ajoute le nouveau monôme en en-tête. On ajoute +1 au nombre de monômes dans la liste puis on retourne le polynôme.

```
if(degre > copie->degre)
{
    nouveau = (monome*) malloc(sizeof(monome));

    nouveau->suivant = copie;

    nouveau->coefficient = coeff;
    nouveau->degre = degre;

    copie = nouveau;

    p->monomes = copie;
    p->n += 1;

    return p;
```

Ensuite la copie de la liste chaînée prend tout son sens car on va devoir parcourir la liste.

```
if(copie->degre > degre && copie->suivant == NULL)
{
    copie->suivant->degre = degre;
    copie->suivant->coefficient = coeff;

    p->n += 1;
```

Si le degré renseigné est inférieur à un des monômes et

que le monôme suivant est NULL alors le monôme en paramètre devient le monôme suivant. On ajoute +1 au nombre de monômes dans la liste puis on retourne le polynôme

Si le degré renseigné est égal à un des degrés de la liste alors les coefficients s'ajoutent et le nombre de monômes dans la liste ne bouge pas. Enfin on retourne le polynôme.

```
if (copie->degre == degre)
{
    copie->coefficient += coeff;

    return p;
```

Enfin si le degré en paramètre est entre les degrés de deux monômes de la liste, alors le monôme en paramètre s'ajoute entre les deux monômes. On ajoute +1 au nombre de monômes dans la liste puis on retourne le polynôme.

```
if(copie->degre > degre && degre > copie->suivant->degre)
{
    nouveau = (monome*) malloc(sizeof(monome));

    nouveau->degre = degre;
    nouveau->coefficient = coeff;

    nouveau->suivant = copie->suivant;

    copie->suivant = nouveau;

    p->n += 1;

    return p;
```

Les trois dernières conditions sont dans un boucle while (copie != NULL) et dont la dernière instruction est copie = copie->suivant.
Le programme vérifie donc ces trois dernières conditions pour tous les nœuds de la liste.

Difficultés rencontrées ?

Les problèmes rencontrés ont surtout été aux niveaux de la consigne , plus précisément sur les bouts de code fournis. Nous ne savions pas si nous pouvions modifier les programmes donnés dans le cahier des charges dont la structure polynômes. Nous nous posions la question si c'était vraiment nécessaire d'utiliser le « int degre ; » fournie dans la structure car pour nous, nous n'en avions pas l'utilité car notre fonction saisie faisait déjà ce travail, mais nous l'avons quand même laissé. La seconde difficulté a été sur la difficulté sur la compréhension de la fonction « Horner » ainsi que sur les possibilités de modification à apporter sur la fonction donnée en consigne. Nous l'avons donc modifiée pour qu'elle s'adapte à notre polynôme tout en que la méthode d'Horner soit respectée, cette fonction a donc été la dernière à être effectuée. Nous ne connaissons donc pas les limites de modification.

```
typedef struct Polynome{
    monome* monomes;
    int n;
    int degre;
} polynome;
```

Ce projet nous a donc beaucoup aidé sur la compréhension des listes chaînées dont nous n'avions pas compris le fonctionnement, nous nous sommes donc inspirés des fichiers partagés par lors des cours en programmation pour effectuer la deuxième partie du projet. Nous avons eu du mal à avancer sur cette deuxième partie car nous voulions une méthode différente où la fonction « saisie » avait moins d'importance mais nous n'avons pas réussi. La fonction d'ajout donc nous a beaucoup aidé, elle a les mêmes propriétés que la fonction « saisie » mais pour les listes chaînées. Notre plus gros problème lors de cette deuxième partie a été le moyen d'avancer de monôme en monôme dans la liste chaînée de polynôme. Le système de copie pour avancer dans la liste nous a posé beaucoup de problèmes. Ce point là a été le plus important selon nous lors de cette deuxième partie car il est présent sur chaque fonction. La deuxième partie a donc les mêmes fonctions que la première partie mais avec un fonctionnement différent.Un autre problème lors de la conception notamment la finalisation du projet a été les problèmes de fuites mémoires, ne possédant pas Valgrind nous n'avons pas repérer s' il y en avait.

```
monome* copie = NULL;

copie = p->monomes;
```

Notre perspective sur l’avenir

L’organisation a été primordiale pour accomplir ce projet notamment la répartition des tâches pour éviter de travailler plusieurs fois le même point, ainsi que la communication sur le début du projet pour bien le comprendre. La partie des listes chainee du projet nous a bien aidé à la comprendre et à éclaircir les points qui étaient flous. Pour programmer les différentes fonctions du polynômes, nous avons donc suivi le cahier des charges dans l’ordre. Le projet est donc allé assez vite dans son ensemble. Cela nous a motivé à réaliser différents projets autour de la programmation et pourquoi pas les pousser plus loin en travaillant cette fois-ci sur le rendu graphique du projet.