TP6
SI4 – Liste chaînée

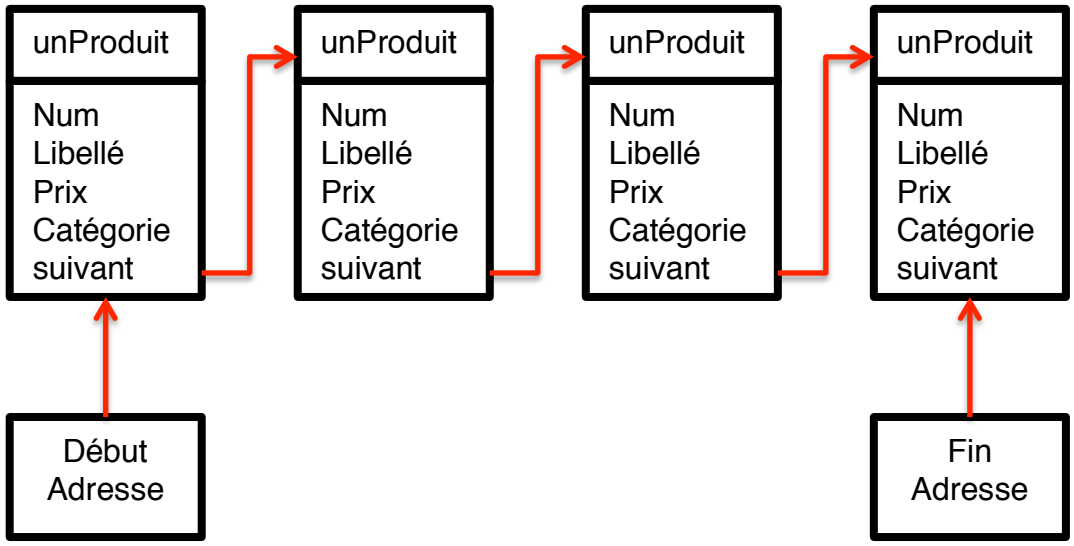
Dans ce nouvel exercice, nous allons aborder l'utilisation des pointeurs à travers le chaînage d'éléments.

Le code qui permet de réaliser le chaînage d'éléments en mémoire est disponible. Cela permettra de voir comment fonctionne le principe.

Le chaînage impose des méthodes spécifiques :

- Création dynamique d'éléments (malloc en C)
- Manipulation d'adresses (donc pointeurs *)
- Mémorisation des adresses (de pointeur à pointeur)

Le principe déjà présenté en cours et rappelé ici à travers ce schéma :



Le champ suivant, mémorise à chaque fois l'adresse du produit suivant. A chaque ajout d'un produit nouveau, c'est la fin qui change. Le début ne change jamais, sauf si la liste est vide. Un élément est associé à une structure.

Attention la liste ne fonctionne que dans un sens, celui des suivants. Dans l'autre sens, il faut ajouter un champ supplémentaire pour mémoriser l'adresse.

Une liste chaînée suppose des opérations de base :

- Création d'un élément
- Insertion de l'élément dans la liste
- Parcours de la liste

Pour compléter, on peut ajouter des opérations :

- De suppression d'un élément
- De dénombrement
- D'affichage d'un élément

Afin d'être plus concret je vous propose de mettre en pratique le code proposé plus loin. Attention, le code proposé a été réalisé avec Xcode en C. Cela suppose quelques adaptations à travers VS ou Code::Blocks.

Les standards présentés en cours sont mis en application dans ce code :

- Répartition de l'interface et de l'implémentation des fonctions dans des fichiers différents (spécifique au langage C/C++)
- Utilisation de procédures et de fonctions pour factoriser le code.
- Passage de paramètres entre programme appelant et sous-programme.
- Pas de variables globales au travers du programme et des sous-programmes.

Commençons par le fichier d'interface : (fonctions.h)

```
//
//  fonctions.h
//  ListeChaineSimple
//
//  Created by Sébastien Gagneur on 30/10/2013.
//  Copyright (c) 2013 Sébastien Gagneur. All rights reserved.
//
// INTERFACE DES STRUCTURES = DEFINITION DES TYPES PERSONNALISES
// INTERFACE DES FONCTIONS = MODE D'EMPLOI

#ifndef ListeChaineSimple_fonctions_h
#define ListeChaineSimple_fonctions_h

// structure d'un produit
typedef struct produit
{
    int numProduit;
    char libProduit[256];
    float priProduit;
    int catProduit;
    struct produit * suivProduit;

};

// structure des pointeurs afin de faire du passage de paramètre qui
fonctionne
typedef struct pointeurs
{
    struct produit * debut;
    struct produit * fin;
};
```

```
// interface des fonctions. Le mode d'emploi

// création d'un nouveau produit
struct produit * creerProduit();

// insertion du produit dans la liste
struct pointeurs * inserProduit(struct produit * unProduit, struct
produit * debut, struct produit * fin);

// affichage du produit courant
void afficProduit(struct produit * unProduit);

// affichage complet de la liste
void listeProduit();

// affichage du nombre de produit contenu dans la liste

#endif

// Séquence code couleur :
http://en.wikipedia.org/wiki/ANSI_escape_code
// Exemple d'affichage des couleurs :
http://www.developpez.net/forums/d1051487/c-cpp/c/debuter/printf-
couleur/
```

Voyons ensuite le fichier d'implémentation : (fonctions.c)

```
//
//  fonctions.c
//  ListeChaineSimple
//
//  Created by Sébastien Gagneur on 30/10/2013.
//  Copyright (c) 2013 Sébastien Gagneur. All rights reserved.
//
// IMPLEMENTATION DES FONCTIONS = CODE DES FONCTIONS

// Directives du préprocesseur pour gérer les inclusions multiples
#ifndef ListeChaineSimple_fonctions_c
#define ListeChaineSimple_fonctions_c

// Ce dont j'ai besoin ici !
#include "fonctions.h"
#include <stdlib.h>


//implémentation des fonctions

//créer un produit
struct produit * creerProduit()
{
    // allocation dynamique qui permet d'allouer un nouvel
emplacement pour chaque nouveau produit créé
    // si vous ne mettez pas l'allocation dynamique, le produit est
```

```
une variable statique, et l'emplacement,
    // l'adresse du produit ne change jamais, si bien, que chaque
produit créé remplace le précédent.
    // Dans ce cas les nouveaux produits créés ne remplacent pas les
précédents, ils sont placés à des
    // adresses différentes.

    struct produit * unProduit = malloc(sizeof(struct produit));

    printf("Code produit ?:");
    scanf("%d",&unProduit->numProduit);
    printf("Libellé produit ?:");
    scanf("%s",&unProduit->libProduit);
    printf("Prix produit ?:");
    scanf("%f",&unProduit->priProduit);
    printf("Catégorie produit ?:");
    scanf("%d",&unProduit->catProduit);
    // Le procuit suivant n'est pas encore connu !
    unProduit->suivProduit = NULL;
    return unProduit;
}

// Afficher un produit
void affichProduit(struct produit * unProduit)
{
    printf("num : %d lib : %s prix : %f cat : %d \n",unProduit-
>numProduit, unProduit->libProduit, unProduit->priProduit,
unProduit->catProduit);
}

// Insérer un produit
struct pointeurs * inserProduit(struct produit * unProduit, struct
produit * debut, struct produit * fin)
{
    // structure pour récupérer mes pointeurs, la fonction ne
retournant pas mes modifications
    struct pointeurs mesPointeurs;
    // Si la liste ne contient pas de produit
    if ( debut == NULL)
    {
        // le seul est unique produit, constitue la début et la fin
de la liste
        debut = unProduit;
        fin = unProduit;
    }
    else
    // La liste contient déjà au moins un produit, et donc cette
fois il faut faire le chaînage
    {
        // le dernier produit de liste à pour suivant le nouveau
produit
        fin->suivProduit = unProduit;
        // le dernier produit, constitue la nouvelle fin
        fin = unProduit;
```

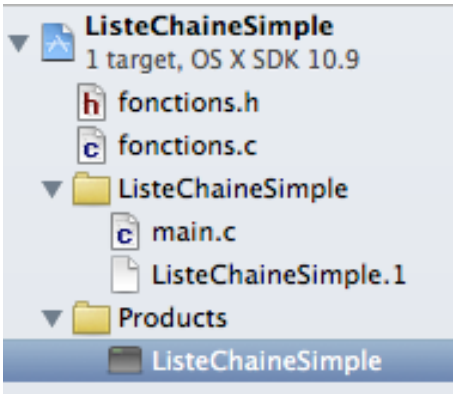
```
    }
    // je mets à jour ma structure pour faire remonter dans le
programme principal mes modifications sur
    // mes pointeurs
    mesPointeurs.debut = debut;
    mesPointeurs.fin = fin;
    return &mesPointeurs;
}

// Parcourir la liste
void listeProduit(struct produit * debut, struct produit * fin)
{
    // le pointeur sur le produit courant
    struct produit * courant;

    // le produit courant est le premier
    courant = debut;

    // tant que l'adresse du produit que je traite est différente de
l'adresse du dernier produit je traite
    while (courant != fin)
    {
        // j'affiche le produit courant
        affichProduit(courant);
        // le nouveau produit courant, est le produit suivant du
produit courant !
        courant = courant->suivProduit;
    }
    // Si vous voulez voir le dernier !
    affichProduit(courant);

}

#endif
```

Passons pour finir au programme appelant : (main.c)

```
//
//  main.c
//  ListeChaineSimple
//
//  Created by Sébastien Gagneur on 30/10/2013.
//  Copyright (c) 2013 Sébastien Gagneur. All rights reserved.
//

#include <stdio.h>
#include <string.h>
#include "fonctions.h"

// Programme principal
int main(int argc, const char * argv[])
{
    // les structures utilisées
    struct produit * unProduit;
```

```
    struct pointeurs * mesPointeurs;
    struct produit * debut = NULL;
    struct produit * fin = NULL;
    char reponse[4] = "oui";

    // Codes couleurs présentés dans le fichier d'interface
    printf ("\033[32;01mListe Chaine Simple\033[00m\n");

    // la boucle pour gérer n produit
    while (strcmp(reponse,"oui") == 0)
    {
        // création d'un produit
        unProduit = creerProduit();
        // insertion d'un produit
        mesPointeurs = inserProduit(unProduit, debut, fin);
        // mise à jour des pointeurs, pour avoir connaissance des
nouvelles valeurs
        debut = mesPointeurs->debut;
        fin = mesPointeurs->fin;
        printf("Autre produit (oui/non) : ?");
        scanf("%s",&reponse);
    }
    // Affichage de la liste constituée
    listeProduit(debut,fin);
    return 0;
}
```

La hiérarchie du projet est la suivante :



Pour réaliser le travail, vous pouvez utiliser Xcode, VS, Code::Blocks, pouvant traiter le langage C.

Le travail qui suit est à réaliser à partir du moment ou vous aurez pu reconstruire ce projet sur machine. Attention : le terme méthode présent dans les questions suivantes, correspond à une fonction. Une méthode est plus précisément une fonction utilisée dans une classe. Le terme est ici, utilisé dans le contexte d'une programmation procédurale.

Travail à faire :

1. Mettre au point une méthode pour compter le nombre d'éléments dans la liste.

2. Mettre au point une méthode pour supprimer un élément de la liste.

3. Mettre au point une méthode pour faire une insertion en début de liste.

4. Modifier le code du projet pour gérer la liste dans les deux sens. Cette modification va évidemment impacter votre travail des questions précédentes, mais aussi modifier l'existant qui vous est fournit.

5. Modifier le code de la méthode « listeProduit » qui permet de parcourir la liste, afin de pouvoir passer en paramètre le sens d'affichage de la liste (vrai : du début vers la fin, faux : de la fin vers le début).