PROBLEMATIQUE

Explication du choix

Nous avons choisi de faire une l'implémentation d'une pile car nous l'avons vu avant en TD. Cela nous permettra de travailler sur le sujet actuel ainsi qu'aidera à mieux comprendre ce-dernière.

Objectif

L'objectif du TP est de savoir implémenter un type abstrait dans une langage de programmation en utilisant le spécification et les opérations fournis par une spécification formelle. Le but de ce TP est également apprendre à savoir tester le programme en créant le fichier de preuve.

Problematique

Souvent il est très difficile pour un développeur de réussir de développer un programme qui correspondait à 100% d'attente d'utilisateur. Pour éviter l'ambiguïté, les informaticiens utilisent un cahier de charges avec toutes les données (spécifications, operations, structures) qui ne sont pas seulement facilitent le processus du développement mais aussi aident à mieux comprendre le fonctionnement du programme et le but final du projet. C'est pour cela que nous utilisons les fichiers *.casl* et les types abstrait.

REALISATION

Nous avons décidé de faire une implémentation de type abstrait pile en utilisant la liste chaînée au lieu d'un tableau car l'utilisation d'une liste chaînée nous permet de gérer la mémoire dynamiquement. En plus, la gestion d'une pile par une liste chaînée est plus pratique car nous ne travaillons qu'avec le premier élément de la liste.

Les étapes du cycle de développement

1) Création d'un fichier interface pile.h

Tout d'abord, à partir d'une spécification formelle du document *pile.casl* nous avons traduit les spécifications et les opérations dans le langage C en créant un fichier interface *pile.h*. Nous avons y déclaré des fonctions et une structure. Nous avons utilisé la protections contre la multiple inclusion en utilisant la structure suivant:

```
#ifndef PILE_H
#define PILE_H

/* corps du fichier pile.h */

#endif
```

Ensuite, nous avons décrit la structure de notre pile qui a été indiquée dans le TAD du fichier *pile.casl* en utilisant une liste chaînée :

```
typedef struct Cell {
  int elt;
  struct Cell *suivant;
} Cellule;

typedef Cellule *PILE;
```

Puis, dans notre fichier interface nous avons fait une représentation suivant pour les opérations de type pile:

```
void initialiser(PILE *pile);
int estVide(PILE pile);
int sommet(PILE pile);
void empiler(PILE *pile, int elt);
void depiler(PILE *pile);
```

L'étape de création d'un fichier *pile.h* est donc pour le but de la représentation des opérations, des structures et des objets du type pile.

2) Création d'un fichier implémentation pile.c

Cette étape est pour le but de création du fichier d'implémentation *pile.c* qui est un fichier source où nous implémentons les opérations décrites dans l'interface *pile.h*. Il est important de n'est pas oublier d'inclure l'interface *pile.h* dans le fichier implémentation *pile.c.*

Un exemple, d'une fonction qui retourne le sommet de la pile:

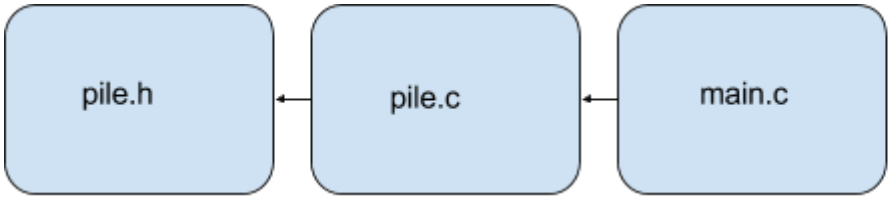
```
int sommet(PILE pile) {
  return(pile->elt);
}
```

Un autre exemple d'une fonction qui initialise la pile:

```
void initialiser(PILE *pile) {
  *pile = NULL;
}
```

3) Création d'un fichier de preuve main.c

Cette étape consiste à vérifier l'implémentation de toutes les opérations ainsi que permet de faire des exemples sur l'utilisation d'une pile. De cette manière nous vérifions aussi que tous les objets et les opérations sont construits correctement et que notre programme de pile fait bien ce que nous attendons. Pour faire fonctionner l'ensemble de tous les fichiers, nous avons fait les inclusions suivant:



Le fichier de preuve *main.c* utilise le fichier implémentation *pile.c* qui prends les opérations et les structures qui sont définies dans l'interface *pile.h*.

Un exemple de verification:

```
initialiser(&pile);

printf("La pile a été initialisée avec le succès et ");

pileVide = estVide(pile);

if (pileVide)
  printf("elle est vide par instant\n");
else
   printf("La pile n'est pas vide par instant\n");
```

CONCLUSION

Pour conclure, nous pouvons dire que l'utilisation et l'application des types abstraits permet aux développeurs d'avoir une vision plus générale et plus claire d'un projet. Cela aussi permet de choisir indépendamment une langage de programmation. Avec l'indication des spécifications et opérations le développeur sait ce qu'il doit attendre du programme sans ambiguïté.