Programmation C - Allocation dynamique – Piles

Christel DARTIGUES-PALLEZ

Déclaration automatique

- Pour chaque variable utilisée dans un programme, il faut réserver un emplacement mémoire correspondant

- À la définition des variables, la réservation de la mémoire se fait automatiquement

Variables dynamiques

- Pourquoi?
  - On ne sait pas toujours de combien de variables on a besoin
  - On ne peut pas toujours déclarer l'espace maximal prévisible
  - On doit alors allouer la mémoire et la libérer au moment où on en a besoin
- Allocation dynamique
  - Utilisation d'un pointeur

Variables dynamiques

- Correspondent à des zones mémoires temporaires
- Désignée uniquement par son adresse qui est contenue dans le pointeur associé
- Allouée quand on en a besoin
- Détruite après son utilisation

Allocation de la mémoire

- void * malloc (sizeof (type de données))

- sizeof : retourne la taille en octets nécessaire pour stocker un objet du type spécifié
- malloc : alloue une zone mémoire de la taille spécifiée

Allocation de la mémoire

- Trouve une zone libre de la taille voulue
- Retourne l'adresse de cette zone, null si il ne reste pas suffisamment de place
- Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "PremierProgramme.h"
#include <malloc.h>

void main () {

    int * pN ;
    pN = (int *)malloc (sizeof (int)) ;
}
```

Libération de la mémoire

- void free (void * pointeur)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "PremierProgramme.h"
#include <malloc.h>

void main () {

    int * pN ;
    pN = (int *)malloc (sizeof (int)) ;

    free (pN) ;
}
```

Allocation dynamique d'un tableau

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "PremierProgramme.h"
#include <malloc.h>

void main () {

    int * pN ;
    pN = (int *)malloc (sizeof (int)) ;


    int* pTab = (int*)malloc(sizeof(int)*3) ;
    pTab[0] = 2 ;
    int nb = 10 ;

    Point2D* pTabPt = (Point2D*) malloc(sizeof(Point2D)*nb) ;
    Point2D** pTabPt = (Point2D**) malloc(sizeof(Point2D*)*nb) ;

    free (pN) ;
}
```

Types de données "classiques"

- Pour chaque type de données on dispose des informations suivantes
  - Type
    - sert de modèle pour un objet (un moule dans lequel une variable va aller)
    - on lui associe une plage de valeurs possibles et un ensemble d'opérations permises sur ces valeurs
  - Plage de valeurs
    - valeurs possibles pour chaque instance du type. Cela implique qu'il faut gérer une représentation des données propres au type
  - Opérations permises
    - opérations, fonctions, procédures

Types de données "classiques"

- Exemples
  - int
  - float
  - char
  - int*
  - Etc.
  - Des types définis par l'utilisateur
- On peut regrouper un ensemble de types de données dans des tableaux
  - Les tableaux sont très pratiques mais ils sont bornés et on ne peut pas toujours savoir combien de données on va devoir manipuler

Types de données "classiques"

- Utilisation naturelle et simple des variables définies à partir des types simples
- Exemple
  - int a = 2, b = 5, c ;
  - c = a + (b*10) ;
- Ceci représente l'interface des entiers
  - Possibilité d'utiliser de façon naturelle les entiers
  - Plus facile à utiliser que la représentation interne des entiers (qui est une représentation binaire)
- Type de données + ensemble de méthodes qui lui sont associées = type abstrait de données

Type abstrait de données

- Définition
  - Un type abstrait de données (TAD) est la donnée d'un modèle et de l'ensemble des actions de base (en création, modification, consultation et destruction) que l'on souhaite possibles sur ce modèle
- Ces actions de base doivent être le seul moyen de manipuler ce modèle

Type abstrait de données

- Différents implémentation des TAD existent
  - Les piles
  - Les files
  - Les listes
  - Les tables de hachage
  - Les arbres
  - Les graphes
  - Etc.

Les piles … les derniers seront les premiers

- L'ajout et la suppression de nouveaux éléments dans la structure se fait toujours par le dessus de la pile
- LIFO: Last in First out
- Exemples d'utilisation
  - Calculs arithmétiques
  - Pile d'exécution des programmes récursifs

Représentation d'une pile

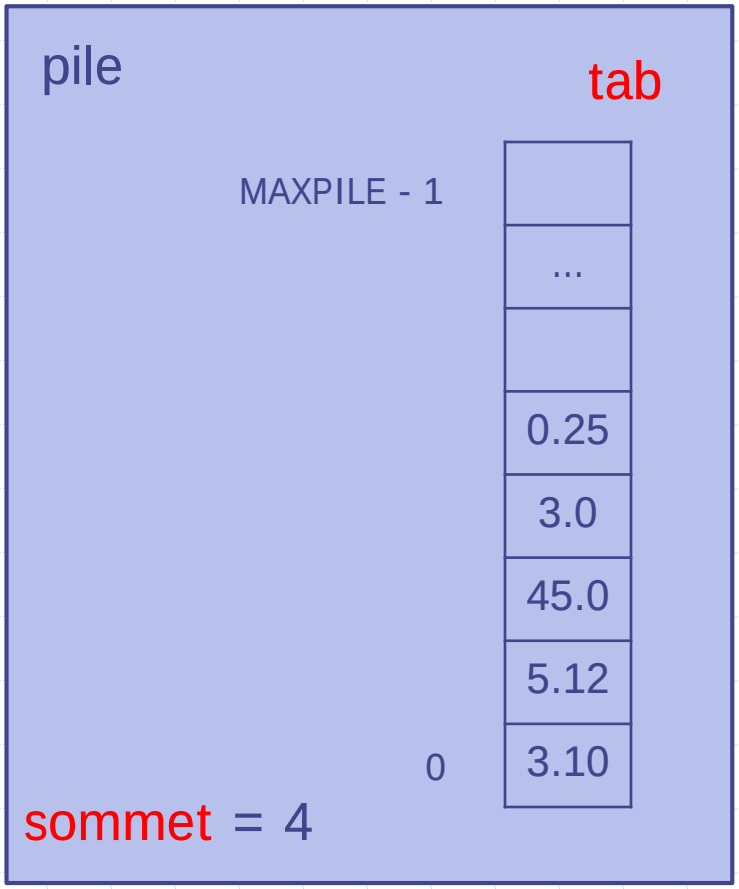
- Par un tableau
  - Avantage: les opérations sont faciles
  - Inconvénient: la hauteur est bornée
- Par des pointeurs
  - Avantages: certaines opérations sont plus faciles (en particulier le test de la pile vide)
  - Inconvénient: espace occupé par les pointeurs

Représentation d'une pile

- On veut gérer une pile de maximum 20 flottants

```
#define MAXPILE 20

typedef struct Spile {
     int sommet ;
     float tab [MAXPILE] ;
}pile ;
```



Opérations sur les piles

- Créer une pile vide
- Empiler un élément
- Dépiler
- Récupérer la valeur du sommet de la pile
- Tester si la pile est vide
- Tester si la pile est pleine
- Afficher une pile
- ...

Créer une pile

- Notation correcte mais peut générer une erreur
- La pile est retournée par valeur et le compilateur ne le gère pas

```
pile creerPile () {
    pile p ;
    p.sommet = -1 ;
    return p ;
}
```

```
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include "PremierProgramme.h"

void main () {
    int i ;
    float f ;
    pile p ;

    p = creerPile () ;
}
```

Créer une pile

- Autre solution

```
void creerPile (pile *p) {
    p->sommet = -1 ;
}
```

Est vide, est pleine

```
bool estVide (pile p) {
      return (p.sommet == -1) ;
 }

bool estPleine (pile p) {
      return (p.sommet == MAXPILE - 1) ;
 }

float sommetPile (pile p) {
      return (p.tab [p.sommet]) ;
 }
```

Empiler, dépiler

```
void empiler (float f, pile *p) {
     if (!estPleine (*p)) {
          p->sommet++ ;
          p->tab[p->sommet] = f ;
     }
}

void depiler (pile *p) {
     if (!estVide (*p))
          p->sommet-- ;
}
```

- Attention : on suppose que si la pile est pleine on ne fait rien... Pas forcément réaliste

Un exemple complet

```
#include "PremierProgramme.h"

void main () {
    int i ;
    float f ;
    pile p ;

    creerPile (&p) ;

    for (i=0; i<5; i++) {
        printf("Donnez une valeur pour la pile\n") ;
        scanf ("%f", &f) ;
        empiler (f, &p) ;
    }
    for (i=0; i<5; i++) {
        printf ("la pile contient %d valeurs\n", p.sommet) ;
        printf ("le sommet de la pile est %f\n", p.tab[p.sommet]) ;
        depiler (&p) ;
    }
    system ("PAUSE") ;
}
```