# APAD

# Algorithmique et programmation 2

# –

# Journée 2

# Allocation dynamique de mémoire

Xavier Crégut

<cregut@enseeiht.fr>

Septembre 2003

# 1 Introduction

## 1.1 Objectifs

Présenter les notions liées à la gestion de la mémoire, en particulier les pointeurs et la gestion de la mémoire dynamique.

## 1.2 Plan

1. L’adresse d’une variable
2. Les pointeurs
3. La mémoire dynamique
4. Types abstraits de données : la pile

# 2 Adresse d'une variable

**Définition :** L'opérateur `&` (adresse) permet d'obtenir l'adresse en mémoire d'une donnée (variable).

**Intérêt :** Obtenir la position en mémoire d'une donnée.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;
    int j;
    printf("&i = %p\n", &i);
    printf("&j = %p\n", &j);

    return EXIT_SUCCESS;
}
```

```
&i = 0xbfffeb78
&j = 0xbfffeb74
```

# 3 Pointeurs

**Définition :** On appelle pointeur une variable qui peut contenir l'adresse d'une donnée.

Par extension, on appelle également pointeur le type d'une telle variable.

```
int *pi;        /* pi est une variable de type pointeur sur int */
double *pd;     /* pd   "       "       "       "       sur double */
Date *d;        /* d    "       "       "       "       sur Date */
double **pd;    /* pd  est un pointeur sur pointeur sur double ! */
```

**Attention :** Il y a autant de types pointeurs que de données pointées !

**Constante :** Il existe une seule constante : `NULL`, compatible avec tous les pointeurs. Elle indique qu'un pointeur ne pointe sur aucune donnée.

**Attention :** La valeur `NULL` n'est pas une valeur indéterminée.

# 3.1 Opérations sur les pointeurs

```
int n, m;                /* deux variables entières */
int *p1, *p2, *p3;       /* trois pointeurs sur int */
Date *d;                 /* un pointeur sur une date */
```

## 3.1.1 affectation

```
p1 = NULL; /* pointeur sur rien */
p2 = &i;   /* pointeur sur la zone mémoire désignée par la variable i */
p3 = p2;   /* initialisation avec la valeur d'un pointeur */
```

Deux types pointeurs sont compatibles si les types pointés le sont.

## 3.1.2 Accès à la donnée pointée

```
*p2 = 10;        /* accès en modification */
m = *p2;         /* accès en lecture */
d->jour = 10;    /* équivalent à (*d).jour = 10 */
```

**Remarque :** Cette opération s'appelle le déréférencement (on obtient la donnée pointée, c'est-à-dire à l'adresse contenue dans le pointeur).

## 3.2 Exemple

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i = 1, j = 2;
    int *ptr = NULL;
    printf("&i = %p, i = %d\n", &i, i);
    printf("&j = %p, j = %d\n", &j, j);
    ptr = &i;
    printf("&ptr = %p, ptr = %p, *ptr = %d\n", &ptr, ptr, *ptr);
    ptr = &j;
    printf("&ptr = %p, ptr = %p, *ptr = %d\n", &ptr, ptr, *ptr);
    *ptr = 5;
    printf("&j = %p, j = %d\n", &j, j);

    return EXIT_SUCCESS;
}
```

```
&i = 0xbfffea34, i = 1
&j = 0xbfffea30, j = 2
&ptr = 0xbfffea2c, ptr = 0xbfffea34, *ptr = 1
&ptr = 0xbfffea2c, ptr = 0xbfffea30, *ptr = 2
&j = 0xbfffea30, j = 5
```

## 3.3 Quelques régles à respecter

**Règle :** Avant de déréférencer un pointeur, il faut s'assurer que son contenu correspond bien à une adresse en mémoire encore valide. La donnée pointée doit donc encore exister.

**Attention :** C'est au programmeur de le vérifier !

**Conséquence :** Dans un sous-programme, ne JAMAIS renvoyer un pointeur sur une variable locale.

**Conséquence 2 :** On ne peut pas accéder à la donnée pointée (déréférencer) si la valeur d'un pointeur est `NULL` !

## 3.4 Application 1 : Passage des paramètres

```
#include <stdio.h>
#include <stdlib.h>

void echanger(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main()
{
    int i = 1, j = 2;
    printf("i = %d et j = %d\n", i, j);
    echanger(&i, &j);
    printf("i = %d et j = %d\n", i, j);

    return EXIT_SUCCESS;
}
```

– Que donne l'exécution du programme précédent ?

## 3.5 Application 2 : Accès à une donnée

On considère un tableau de notes, une note étant définie par sa valeur et son coefficient. L'objectif de noter est d'enregistrer une nouvelle note dans le tableau qui on contient déjà nb.

```
struct Note {
    double valeur;
    double coefficient;
};
typedef struct Note Note;

void noter(Note tab[], int *nb, int valeur, int coefficient) {
    Note *n = &tab[*nb];        /* adresse de la note à renseigner */
    n->valeur = valeur;         /* initialisation de la note */
    n->coefficient = coefficient;
    (*nb)++;                    /* une note de plus ! */
}
```

Aurait-on pu prendre une variable locate du type `Note` (au lieu de `Note *`) ?

# 4 Mémoire dynamique

En plus de la pile d'exécution (mémoire automatique), il existe également de la mémoire dynamique, le **tas**.

Elle est dite dynamique car c'est le programmeur qui décide de réserver cette mémoire et c'est à lui de penser à la libérer.

**Remarque :** La mémoire vue jusqu'à présent est dite automatique car c'est le compilateur qui s'occupe entièrement de la réservation et de la libération de cette mémoire (en fonction de la portée d'une variable). Exemple : les paramètres et les variables locales d'un sous-programme.

## 4.1 Accès à la mémoire dynamique

La mémoire dynamique n'est accessible qu'à travers d'une variable de type pointeur.

## 4.2 Allocation dynamique de mémoire

Pour allouer dynamiquement de la mémoire dans le tas, on utilise la fonction `malloc` qui prend en paramètre la quantité de mémoire à réserver.

```
double *ptr;

ptr = (double *) malloc(sizeof(double));
            /* réservation de la mémoire dans le tas */
/* ptr contient l'adresse de la mémoire réservée (allouée) */
*ptr  = 10      /* initialisation de la mémoire réservée */
```

**Attention :** Ne pas oublier de faire l'initialisation après la réservation, sinon les données sont indéterminées !

**Conseil :** Toujours utiliser l'opérateur **`sizeof`** qui indique la quantité de mémoire nécessaire à la représentation d'un type.

## 4.3 Libération de la mémoire pointée

Pour libérer de la mémoire allouée dynamiquement (donc dans le tas), on utilise la fonction `free`.

```
free(ptr)                    /* libérer la mémoire associée à ptr */
```

**Attention :** Il est interdit :

– de libérer un pointeur de valeur `NULL` ;

– de libérer une zone mémoire déjà libérée ;

– de libérer de la mémoire allouée automatiquement par le compilateur.

Dans tous ces cas, il s'agit d'une erreur de programmation. Si vous avez de la chance, le programme plantera tout de suite à l'exécution (mais ce n'est pas toujours le cas !).

## 4.4 Exemple

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    double *ptr;                                    /* déclaration */
    ptr = (double *) malloc(sizeof(double));        /* allocation */
    printf("Donner un réel : ");                    /* utilisation */
    scanf("%lf", ptr);
    printf("Racine carrée : %f\n", sqrt(*ptr));
    free(ptr);                                      /* libération */
    return EXIT_SUCCESS;
}
```

**Attention :** Si ce programme montre bien l'utilisation de la mémoire dynamique, il constitue un **contre-exemple** de ce qu'il faut faire. Pourquoi ?

## 4.5 Intérêt de la mémoire dynamique

Il est possible d'allouer de la mémoire en fonction des besoins connus à l'exécution sans nécessité de majorer les besoins.

## 4.6 Inconvénients

– attention aux fuites de mémoire : perte d'accès (de pointeur) sur une zone de mémoire non libérée : elle ne pourra jamais plus l'être !

```
ptr = (double *) malloc(sizeof(double));
ptr = ...            -- fuite de mémoire !
```

– contrairement à la pile, le tas est une mémoire fragmentée ;

– c'est au programmeur de décider quand allouer de la mémoire (facile) mais aussi de décider quand elle peut être libérée ( difficile).

**Règle :** Il est recommandé de libérer la mémoire dynamique **dès** qu'on n'en a plus besoin. La difficulté est de savoir quand on n'en a plus besoin !

## 4.7 Retour sur l'initialisation d'un pointeur

**Règle :** Il existe trois manières d'initaliser une variable de type pointeur :

– initialiser à la constante `NULL` (le pointeur ne pointe alors sur rien) ;
– initialiser avec l'adresse d'une donnée existante (adresse d'une variable ou valeur d'un autre pointeur) ;
– initaliser par allocation dynamique de mémoire avec `malloc`.

**Attention :** Ce n'est pas parce que l'on déclare une variable de type pointeur que l'on doit forcément allouer de la mémoire !

Le pointeur peut servir :

– à référencer une donnée déjà existante (donc pas d'allocation !) ;
– accéder à une donnée que l'on va créer dynamiquement (donc allocation pour cette donnée).

## 4.8 La structure de données pile

**Exercice 1** On souhaite définir une réalisation du type abstrait pile qui n'a pas, a priori, de contrainte sur le nombre d'éléments qu'elle contiendra (pas de capacité maximale définie à la compilation).

**1.1** Proposer un module définissant une telle pile.

**1.2** Comment faire pour obtenir le nombre d'éléments de la pile ? Ajouter une opération qui permet de l'obtenir.

# 5 Mémoire automatique vs mémoire dynamique

- Gestion réalisée par le compilateur / par le programmeur ;
- Dimensionnement réalisé à la compilation / à l'exécution ;
- Mémoire gaspillée car surdimensionnement / moins de gaspi (attention, le chaînage utilise de la place !)
- Données contiguës en mémoire ==> rapidité d'accès à un élément mais insertion ou suppression coûteuse / mémoire chaînée ==> lenteur d'accès mais insertion suppression plus efficace.

# Table des matières