Structures de données (LU2IN006)

Nawal Benabbou

Licence Informatique - Sorbonne Université

2020-2021

Présentation et organisation de l'UE

Contenu de l'UE

Présenter de manière synthétique les structures de données couramment utilisées en informatique afin de connaître les clefs permettant de choisir optimalement les structures les plus adaptées pour résoudre un problème donné.

Quelques références bibliographiques :

- C. Cormen, C. Leiserson, R. Rivest. "Introduction à l'algorithmique", Dunod, 1994.
- A. Aho, J. Ullman. "Concepts fondamentaux de l'informatique", Dunod, 1993.
- M. Divay. "Algorithmes et structures de données génériques", Dunod, 1999.
- R. Malgouyres, R. Zrour, F. Feschet. "Initiation à l'algorithmique et aux structures de données en C", 2008.

Déroulement de l'UE

Cours : 11 séances de 1h45 (mercredi 16h-17h45)

TD/TME :

- 11 séances, en décalage d'une semaine avec le cours
- Chaque séance : TD de 1h45 et TME de 1h45 (en langage C)

Documents et informations : sur Moodle.

Équipe pédagogique

Responsable : N. Benabbou

Chargés de cours : N. Benabbou et P.-H. Wuillemin

Groupe	Chargé de TD	Doublure(s) de TME
1 (Mer. 10h45-12h30/14h00-15h45)	H. Martin	M. Fares
2 (Ma. 08h45-10h30/10h45-12h30)	M. Amoussou	M. Durand
3 (Jeu. 08h45-10h30/10h45-12h30)	L. Prosperi	N. El Madhoun
4 (Jeu. 08h45-10h30/10h45-12h30)	M. Lehaut	D. Wladdimiro
5 (Ma. 08h45-10h30/10h45-12h30)	N. Benabbou	M. Fares, C. Gainon de Forsan
6 (Lun. 10h45-12h30/14h00-15h45)	M. Tydrichova	D. Wladdimiro
7 (Lun. 10h45-12h30/Ma. 8h45-10h30)	M. Millet	M. Ghanem
8 (Mer. 10h45-12h30/14h00-15h45)	P. Civit	N. Gross–Humbert
9 (Mer. 10h45-12h30/14h00-15h45)	N. Benabbou	M. Tydrichova, R. Yehia
10 (Ven. 14h00-15h45/16h00-17h45)	D. Genius	J.-G. Ganascia

Contacts (dans l'ordre) : nawal.benabbou@lip6.fr, pierre-henri.wuillemin@lip6.fr, hugo.martin@lip6.fr, fares@isir.upmc.fr, manuel.amoussou@centralesupelec.fr, martin.durand@lip6.fr, laurent.prosperi@lip6.fr, nour.el-madhoun@epita.fr, mathieu.lehaut@lip6.fr, daniel.wladdimiro@lip6.fr, clara.gainon-de-forsan-de-gabriac@lip6.fr, tydrichova@ia.lip6.fr, maxime.millet@lip6.fr, marwan.t.ghanem@gmail.com, pierre.civit@lip6.fr, nathanael.gross–humbert@lip6.fr, raja.yehia@lip6.fr, daniela.genius@lip6.fr, jean-gabriel.ganascia@lip6.fr

Évaluation

Contrôle des connaissances

- Examen final : 50%
- Un projet long : 30%
- Contrôle continu : 20%

Sur le projet

Le projet est à réaliser pendant les séances 5 à 10 de TME, avec une soutenance prévue en séance 11.

Attention : la note du projet est conservée en deuxième session.

Sur le contrôle continu

Pas de partiel, mais deux mini-projets à réaliser pendant les premières séances de TME, comptant pour 50% chacun.

L'abstraction

Définition générale (Larousse)

Opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment de ses autres caractères.

Quelques types d'abstraction :

- L'abstraction **par simplification** permet de réduire la description d'un objet en ne conservant que les informations utiles ou importantes.
  Exemples : un plan de métro, une fiche personnage.
- L'abstraction **par généralisation** permet d'aller du particulier vers le général, d'un objet à une classe d'objets plus générale.
  Exemples : les hommes sont des vertébrés, un bureau est un meuble.

L'abstraction en informatique

Abstraction informatique

L'abstraction en informatique consiste à synthétiser des caractéristiques et traitements communs, applicables à des entités ou concepts variés, afin de simplifier et d'unifier leur manipulation.

Exemples

- Un ordinateur peut se définir comme un ensemble de composants électroniques, et les couches progressives d'abstraction (noyau, système d'exploitation...) permettent à l'utilisateur de ne pas avoir à gérer directement les aspects électroniques lorsqu'il manipule l'ordinateur.
- Les variables sont des abstractions, permettant de représenter par un symbole la valeur qu'elle contient, qui peut être une structure complexe.
- Les procédures sont des abstractions, permettant de résumer en une instruction des choses qui peuvent être complexes à être réaliser.
- Les langages informatiques sont des abstractions, faisant correspondre le langage machine à un langage plus compréhensible par l'homme.

Attention : l'abstraction est une bonne chose (codes généraux, faciles à lire), mais cela ne doit pas rendre le programme plus lent !

Type abstrait de données

Définition

Un type abstrait est une spécification mathématique d'un ensemble d'objets et d'un ensemble d'opérations applicables à ces objets. Il est défini indépendamment de sa représentation en mémoire.

Exemples :

- les nombres entiers (`int`), munis des opérations $+, *$ etc.
- les chaînes de caractères (`char`), munis des opérations de concaténation, d'insertion, etc.
- les listes, munis des opérations d'insertion, d'accès à des éléments, etc.

Remarques :

- Un type abstrait de données ne définit pas la façon dont les données sont stockées, ni la manière d'implémenter les méthodes.
- Définir un type abstrait permet de manipuler des objets sans connaître leur représentation dans la mémoire de l'ordinateur (codage, implémentation), de sorte à élaborer des algorithmes de manière abstraite, conduisant à des programmes compréhensibles et réutilisables.

Structure de données

Définition

Une structure de données est une représentation concrète des objets décrits par un type abstrait dans la mémoire d'un ordinateur, accompagnée de l'implémentation des opérations sur cette représentation.

Exemple : un booléen est représenté par 1 octet en C++, 1 bit en java...

Autre exemple : un dictionnaire

Type abstrait de données qui associe à un ensemble de clefs, un ensemble correspondant de valeurs. Chaque clef est associée à une seule valeur (au plus). Opérations usuelles : insertion, modification, recherche, suppression etc.

Plusieurs implémentations possibles :

- Tableau (ensemble de cellules situées dans un espace contigu en mémoire),
- Liste chaînée (ensemble de cellules contenant l'adresse d'une autre cellule),
- Arbre binaire de recherche (hiérarchie de cellules définie avec l'adresse des deux cellules "filles"),
- Table de hachage (tableau non ordonné permettant un accès aux éléments par leur clé), etc.

(On détaillera toutes ces structures de données plus tard)

Choix de la structure de données

Coût associé au choix d'une structure de données

Le choix de la structure de données a des conséquences sur les performances des programmes :

- la déclaration d'un objet à un coût en mémoire (en octets). On parlera de complexité spatiale.
- les opérations sur les objets ont un coût en temps de calcul. On parlera de complexité temporelle, mesurée en nombre d'opérations élémentaires.

Exemple

- `int i;` $\Rightarrow$ déclare l'utilisation de x octets en mémoire.
- `i = i + 1;` $\Rightarrow$ 2 opérations élémentaires (addition puis affectation).

Observation

Toutes les structures de données n'ont pas les mêmes performances, ni les mêmes cas d'utilisation.

$\Rightarrow$ Cette UE a pour but d'apporter une connaissance des principales structures de données, afin d'apprendre dans quel cas il convient d'utiliser une structure de données plutôt qu'une autre, et de savoir comment les implémenter.

Quelques définitions de base

Variable

Une variable est une “boîte” ayant un nom et contenant une valeur appartenant à un domaine. Une variable peut correspondre à un type simple (entier, flottant), ou à une structure plus élaborée.

Opération élémentaire

Une opération élémentaire est une commande informatique d'un langage portant sur les variables et réalisant une opération simple (arithmétique, logique, affectation, etc.) qu'un ordinateur réalise en un nombre connu d'appels au processeur (CPU).

Exemples : `a = b ; i > j ; a = a + b ;`

Structure de contrôle

Une structure de contrôle est une instruction pouvant dévier le flot de contrôle du programme lorsqu'elle est exécutée. En programmation impérative, elle est construite à partir de structures élémentaires :

- Séquence (structure de contrôle implicite)
- Tests, Boucles...

Quelques définitions de base

Programme

Un programme est composé de :

- un ensemble fini de variables, dont des variables d'entrée
- un ensemble fini d'opérations élémentaires
- un ensemble fini de structures de contrôle

Observation

Une exécution d'un programme est donc une suite d'opérations élémentaires. Plus précisément, pour des valeurs données des variables d'entrée, le programme (et ses structures de contrôle) donne une suite d'opérations élémentaires qui peut être finie ou infinie.

Remarque : une opération peut être exécutée plusieurs fois dans un programme.

Quelques définitions de base

Problème

Un problème se définit à partir de paramètres, qui prennent des valeurs dans un domaine, et d'une question.

Exemples :

- À partir d'un tableau, construire un tableau trié.
- Trouver le plus court chemin dans une carte routière.
- Découper du tissu pour faire des vêtements en minimisant les chutes.
- Identifier les contours d'un visage sur une photo.

Algorithme

Un algorithme est un programme résolvant un problème donné :

- en un nombre fini d'opérations élémentaires (on parle de terminaison).
- en donnant une réponse juste à la question du problème, quelque soit les valeurs des variables d'entrée (on parle de validité).

Exemple : recherche dans un tableau

Pour la plupart des algorithmes, le nombre d'opérations réalisées dépend de la valeur des paramètres d'entrée. Prenons l'exemple de la recherche d'une valeur dans un tableau.

```
1  #include <stdio.h>
2
3  void recherche_tab(int val, int* tab, int n){
4      int i=0;
5
6      while ((i<n) && (tab[i]!=val)){
7          i=i+1; //ou encore i++;
8      }
9
10     if (i<n){
11         printf("Valeur dans le tableau en position \%d",i);
12     } else {
13         printf("Valeur non pr'esente dans le tableau");
14     }
15 }
```

Si la valeur recherchée est dans la première case du tableau, alors un seul tour de boucle est effectué. Par contre, si la valeur recherchée n'est pas dans le tableau, il faut faire autant de tours de boucle que la taille du tableau.

Performances d'un algorithme : complexité temporelle

Évaluation d'un algorithme en temps de calcul

Pour évaluer les performances d'un algorithme (ou programme), on considère souvent le nombre maximum d'opérations élémentaires que celui-ci peut effectuer lors d'une exécution.

Définition : complexité temporelle

Le cas où un algorithme effectue le nombre maximum d'opérations est communément appelé le "pire des cas". Le nombre d'opérations élémentaires que l'algorithme effectue dans le pire des cas est appelé "complexité-temps pire-cas" ou "complexité temporelle".

Retour à l'exemple de recherche dans un tableau

Pour cet algorithme, le pire des cas est la recherche d'un élément non présent dans le tableau (n tours de boucle). À chaque tour, 5 opérations élémentaires sont réalisées :

- 3 tests (ligne 7)
- 1 addition et 1 affectation (ligne 8)

Au total, il faut donc $5n+3$ opérations dans le pire des cas (en comptant l'initialisation de la variable `i` et l'affichage du résultat).

Performances d'un algorithme : complexité temporelle

Observation

Il est fréquent que l'on ne puisse donner précisément la complexité d'un algorithme, car le nombre d'opérations peut dépendre de beaucoup de paramètres. Dans ce cas, on indique des bornes (supérieures et/ou inférieures) sur la complexité temporelle, en utilisant les notations de Landau.

Définition : notation de Landau O (borne supérieure)

Soit $f(n)$ et $g(n)$ deux fonctions positives définies sur $\mathbb{N}$. On dit que $f(n)$ est en $O(g(n))$ si la fonction f est asymptotiquement bornée supérieurement par la fonction g à un facteur près. Plus formellement, $f(n)$ est en $O(g(n))$ s'il existe un entier n_0 et un réel $k > 0$ tels que pour tout $n \geq n_0$:

$$f(n) \leq kg(n).$$

Retour à l'exemple de recherche dans un tableau

Pour cet algorithme, notons $f(n)$ le nombre d'opérations effectuées dans le pire cas pour un tableau de taille n. On peut dire que $f(n)$ est en $O(n)$ puisqu'il faut **au plus** $5n+3$ opérations élémentaires dans le pire des cas pour afficher la valeur recherchée. En effet, avec $k = 6$ et $n_0 = 3$, on a bien $f(n) \leq k \times n$ pour tout $n \geq n_0$. On dit aussi que l'algorithme a une complexité temporelle en $O(n)$.

Performances d'un algorithme : complexité temporelle

Soit $f(n)$ et $g(n)$ deux fonctions positives définies sur $\mathbb{N}$.

Définition : notation de Landau Ω (borne inférieure)

On dit que $f(n)$ est en $\Omega(g(n))$ si la fonction f est asymptotiquement bornée inférieurement par la fonction g à un facteur près. Plus formellement, $f(n)$ est en $\Omega(g(n))$ s'il existe un entier n_0 et un réel $k > 0$ tels que pour tout $n \geq n_0$:

$$kg(n) \leq f(n).$$

Définition : notation de Landau Θ (borne inférieure et supérieure)

On dit que $f(n)$ est en $\Theta(g(n))$ si la fonction f est dominée et soumise à g asymptotiquement. Plus formellement, $f(n)$ est en $\Theta(g(n))$ s'il existe un entier n_0 et deux réels $k_1, k_2 > 0$ tels que pour tout $n \geq n_0$:

$$k_1 g(n) \leq f(n) \leq k_2 g(n).$$

Retour à l'exemple de recherche dans un tableau

On sait que la complexité temporelle de cet algorithme est en fait en $\Theta(n)$ puisqu'il faut **exactement** $5n+3$ opérations élémentaires dans le pire des cas. En effet, avec par exemple $k_1 = 5, k_2 = 6$ et $n_0 = 3$, on a bien $k_1 \times n \leq f(n) \leq k_2 \times n$ pour tout $n \geq n_0$.

Exemple : Calcul de la somme des factorielles ($\sum_{i=1}^{n} i!$)

```
1  double somme_factorielles(int n){
2      int i,j;
3      double f,s;
4      s=0;
5      for (i=1;i<=n;i++){
6          f=1;
7          for (j=1;j<=i;j++){
8                  f=f*j;
9          }
10         s=s+f;
11     }
12     return s;
13 }
```

Quelle est sa complexité temporelle ?

Sa complexité est en $\Theta(n^2)$ car le nombre d'opérations élémentaires est :

$$3+\sum_{i=1}^{n}\left(6+\sum_{j=1}^{i}4\right)=3+\sum_{i=1}^{n}\left(6+4i\right)=3+6n+\frac{4n\times(n+1)}{2}=2n^2+8n+3=\Theta(n^2).$$

Remarque : Avec les notations de Landau, on peut compter 1 par instruction élémentaire et ignorer les tests de boucles, afin d'obtenir la même bonne borne :

$$2+\sum_{i=1}^{n}\left(2+\sum_{j=1}^{i}1\right)=2+\sum_{i=1}^{n}\left(2+i\right)=2+2n+\frac{n\times(n+1)}{2}=\frac{1}{2}n^2+\frac{5}{2}n+2=\Theta(n^2).$$

Autre exemple : Calcul de la somme des factorielles

Voici un autre algorithme pour résoudre le même problème :

```
1  double somme_factorielles_v2(int n){
2      int j;
3      double f,s;
4      s=0;f=1;
5
6      for (i=1;i<=n;i++){
7          f=f*i;
8          s=s+f;
9      }
10     return s;
11 }
```

Quelle est sa complexité temporelle ?

Sa complexité est en $\Theta(n)$ car son nombre d'instructions élémentaires est exactement égal à $3+2n$ (on prend par exemple $k_1 = 2$, $k_2 = 3$, $n_0 = 3$).

$\Rightarrow$ On préférera donc cet algorithme au précédent, pour avoir un programme plus efficace.

Performances d'un algorithme

Pourquoi est-ce que la complexité d'un algorithme est important ?

Un petit exemple

Supposons que l'ordinateur soit capable de réaliser 10^9 instructions élémentaires par seconde (1 GigaHertz). Notons $f(n)$ le nombre d'instructions élémentaires réalisées par un programme, où n est la taille de l'instance (par exemple, la taille d'un tableau). Quand $n = 1000$, combien de temps faut-il pour exécuter un programme de complexité :

- $f(n) = \Theta(n^2)$? 1×10^{-3} seconde
- $f(n) = \Theta(n^3)$? 1 seconde
- $f(n) = \Theta(n^4)$? 16.7 secondes
- $f(n) = \Theta(n^5)$? 11.6 jours
- $f(n) = \Theta(2^n)$? 3.39×10^{282} siècles

$\Rightarrow$ Comme le choix des structures de données a un impact sur le nombre d'instructions élémentaires réalisées par votre programme pour effectuer des opérations sur vos données, il est primordial de savoir comment choisir les structures de données selon l'application considérée.

Une autre mesure de performance : la complexité spatiale

Observation

On peut aussi vouloir évaluer l'efficacité d'un algorithme selon son occupation mémoire. En effet, il faut pouvoir déterminer une implémentation permettant de ne pas dépasser l'espace mémoire d'un ordinateur.

Définition : complexité spatiale

L'espace mémoire réservé pour l'exécution d'un algorithme dans le pire des scénarios est appelé "complexité spatiale" ou "complexité-espace".

$\Rightarrow$ La complexité spatiale peut être indiquée avec les notations de Landau, comme la vitesse d'un algorithme, par rapport à la valeur des variables d'entrée.

Remarque

Les complexités temporelles et spatiales sont fréquemment en opposition, et il faut déterminer un compromis entre les deux.

Exemple : calcul de la factorielle d'un nombre

```
1  double factorielle_iteratif(int n){
2      int fact= 1;
3      for (i=2;i<=n;i++){
4          fact=fact*i;
5      }
6      return fact;
7  }
```

```
1  double factorielle_recursif(int n){
2      if (n <= 1){
3          return 1;
4      }else{
5          return n*factorielle_recursif(n-1);
6      }
7  }
```

Comparaison

Ces deux algorithmes ont la même complexité temporelle : $\Theta(n)$. Cependant, leur complexité spatiale est différente :

- La version itérative utilise un nombre constant de cases mémoire (au plus 4).

$\Rightarrow$ Complexité spatiale : $\Theta(1)$.

- La version récursive a besoin de 2 cases mémoire par appel, donc $2n$ au total.

$\Rightarrow$ Complexité spatiale : $\Theta(n)$.

Pour résumer

Objectif de l'UE

L'objectif de ce module est tout à la fois :

- Étudier les différents types et structures de données.
- Comprendre leur vitesse et leur occupation mémoire.
- Apprendre à les implémenter correctement (en langage C).
- Savoir identifier dans quelles situations une structure de données est préférable à une autre.

Remarque

Cette étude est réalisée avec le langage C, mais elle pourrait être faite dans d'autres langages, et elle vous sera utile quel que soit le langage !

$\Rightarrow$ Commençons par quelques rappels de C.

Adresse mémoire, type et pointeur

Adresse mémoire et type

La mémoire d'un ordinateur peut être vue comme une séquence de bits (0 ou 1) mis les uns après les autres. Un emplacement dans la mémoire est ainsi repéré par son **adresse physique** et sa **taille** (en bits, octets...). Si l'on ne connaît pas le codage utilisé à cet emplacement, on ne pas lire/utiliser le contenu stocké à cet emplacement. On appellera **type** le codage d'un emplacement (qui correspond à une taille pré-définie). On appellera **case mémoire** un emplacement typé.

Pointeur

Un pointeur est une variable contenant l'adresse d'un emplacement mémoire (on dit qu'il "pointe" vers un emplacement mémoire). Un **pointeur sur un type** est un pointeur dont la valeur correspond à une case mémoire (c'est-à-dire un emplacement typé). Un pointeur sur un type se déclare en ajoutant une étoile derrière le type.

Exemple : l'instruction `int* p;` déclare un pointeur sur un entier.

Remarque : l'accès à une case mémoire est en $\Theta(1)$.

Manipulation de pointeurs

Les opérateur & et $*$

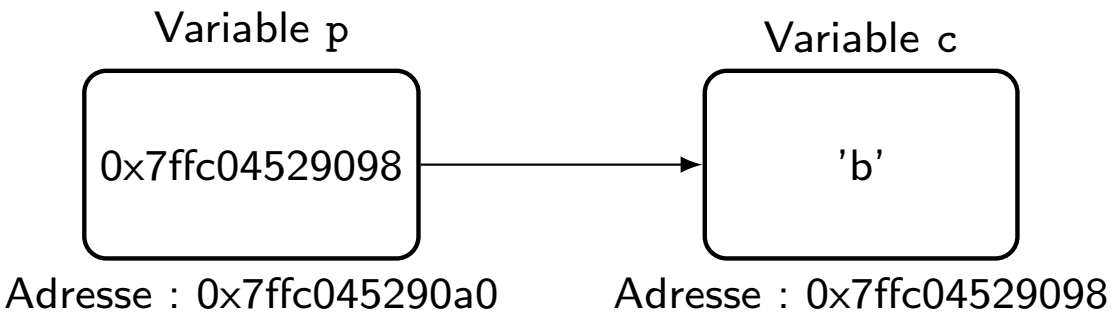
En langage C :
- l'opérateur & permet de récupérer l'adresse de la case mémoire d'une variable.
- l'opérateur $*$ permet d'accéder à la case mémoire vers laquelle pointe un pointeur (on parle de déréférencement).

Exemple :

- `char c = 'a';` permet de réserver une case mémoire de la taille d'un `char` (1 octet), contenant le caractère 'a' (codé en binaire).
- `char* p;` permet de réserver une case mémoire de la taille d'un pointeur sur un caractère.
- `p = &c;` permet de récupérer l'adresse de la variable `c` et de la stocker dans `p` (on dit que `p` pointe vers la case mémoire de `c`).
- `printf("%c\n",*p);` permet d'afficher la valeur stockée à la case mémoire vers laquelle pointe `p` (ici affiche le caractère 'a').
- `*p = 'b';` permet de changer la valeur de la case mémoire pointée par `p`.

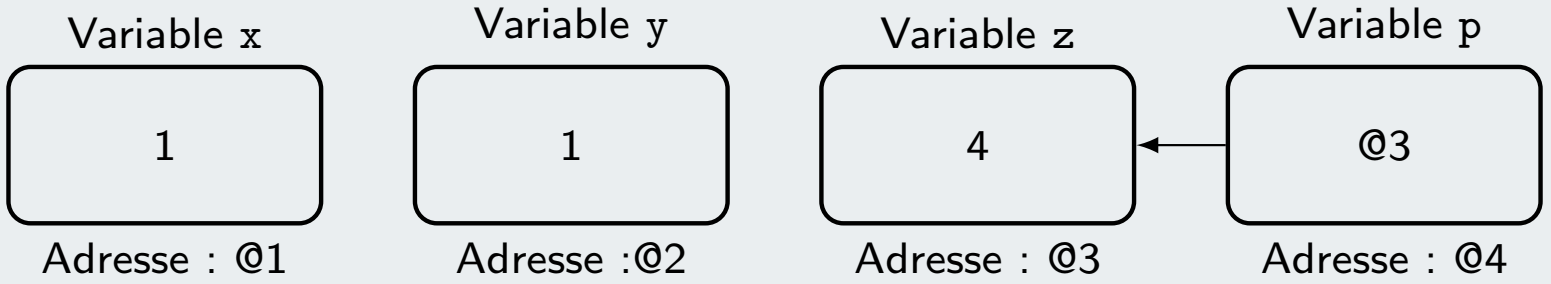
Représentation graphique de la mémoire

On a l'habitude de représenter les cases mémoires par des boîtes (contenant les valeurs des variables), et les pointeurs par des flèches. Pour l'exemple précédent, on obtient à la fin la représentation suivante :



Exercice : Donner la représentation graphique

```
1  int x=1,y=2,z=3;
2  int* p = &x;
3  y = *p;
4  p = &z;
5  *p = 4;
```



Mémoire contiguë et tableaux

Définition : tableau

En langage C, un tableau est un pointeur sur une **zone contiguë** de la mémoire composée d'éléments **du même type**. Plus précisément, un tableau est représenté par un pointeur sur le premier élément, les autres éléments pouvant être obtenues par l'arithmétique des pointeurs.

Remarque : l'instruction `int* t;` permet de déclarer un pointeur sur un entier, ou un tableau d'entiers... la différence se fera au moment de son initialisation.

Sur l'arithmétique des pointeurs

Si t est un tableau de type `int *`, alors `t+i` est un pointeur sur le `i`ème entier du tableau `t`. En effet, l'opération `+i` correspond à se déplacer dans la mémoire de `i` fois le nombre d'octets permettant de représenter un entier. Ainsi, pour accéder à la valeur du `i`ème élément, on peut écrire `t[i]` ou encore `*(t+i)`.

Exercice : donner la représentation graphique du tableau d'entiers `t` = {2,5,1}

Allocation mémoire d'un tableau

Attention : écrire `int* t;` déclare un tableau d'entiers, mais ne réserve pas l'espace mémoire pour les éléments du tableau. Il faut réserver de la mémoire, ce qui s'appelle l'allocation mémoire.

Allocation statique

Un tableau peut être alloué statiquement de la manière suivante :

- `int t[4];` alloue l'espace mémoire pour un tableau de 4 entiers, sans initialiser les valeurs.
- `int t[4]={2,4,7,1};` permet de faire les deux.
- `int t[4]={2,4};` alloue le tableau en entier et initialise seulement les deux premières valeurs (les autres sont mises à zéro).
- `int t[4]={};` alloue le tableau en entier et le remplit avec des zéros.

Remarques : On ne peut pas écrire `int[n] t;` si `n` est une variable, car l'allocation statique réserve l'espace mémoire lors de la compilation (or la valeur de `n` est connue à l'exécution). Mais on peut écrire `int[N_MAX] t;` quand N_MAX est une constante définie avec la commande #define. En effet, cette commande préprocesseur remplace toutes les instances de N_MAX par sa valeur avant la compilation du programme. Par contre, cela ne fonctionne pas avec les constantes déclarées avec le modificateur `const`...

Allocation mémoire d'un tableau

Allocation dynamique

Un tableau peut être alloué dynamiquement en utilisant la fonction `malloc` qui prend en paramètre le nombre d'octets à réserver. Par exemple :

- `malloc(12);` réserve un espace de 12 octets.
- `malloc(5*sizeof(int));` réserve l'espace nécessaire pour stocker 5 entiers de type `int`.

Attention : la fonction malloc alloue de l'espace non typé (elle retourne un pointeur de type `void*`). Pour donner un type à la mémoire allouée, il faut "caster" le retour de la fonction `malloc`. Par exemple, on peut écrire :

- `int* t1 = (int*) malloc(5*sizeof(int));`
- `double* t1 = (double*) malloc(n*sizeof(double));`

Remarques :

- Dans le dernier exemple, la taille de `t1` est définie à partir d'une variable (nommée `n`), ce qui n'aurait pas été possible avec un allocation statique. Par exemple, cela permet à un utilisateur de saisir sa taille au clavier.
- Par ailleurs, allouer la mémoire dynamiquement permet de changer la taille d'un tableau en cours d'exécution (une ou plusieurs fois, avec la fonction `realloc`), et donc permet une meilleure gestion de la mémoire.

Allocation mémoire d'un tableau

Quand un espace alloué dynamiquement n'est plus utilisé, il faut libérer la mémoire pour permettre au programme (ou à d'autres programmes) de l'utiliser. Dans certains langages, ce travail est réalisé par un "ramasse-miette" (garbage collector), ce qui n'est pas optimal car seul le programmeur sait exactement ce qui est à libérer et quand.

Désallocation (libération de la mémoire)

En langage C, on désalloue avec la fonction `free` :

- `free(p);` permet de libérer l'espace mémoire pointé par `p`. Il peut contenir un type simple, ou une structure plus élaborée.
- `free(t);` permet de libérer la zone contiguë (tableau) pointéd par `t`.

Gestion des problèmes d'allocation

Comme l'allocation mémoire a lieu pendant l'exécution, des problèmes peuvent surgir si le système ne peut plus allouer de mémoire à un moment donné. Cela peut arriver quand l'espace demandé est trop grand ou que des zones mémoires n'ont pas été libérées. Pour éviter les *segfault*, on traite les éventuelles erreurs :

```
1 double* p = (double*) malloc(n*sizeof(double));
2 if (p == NULL){ // ou p == 0
3     printf("Erreur durant l'allocation m'emoire");
4     return NULL;
5 }
```

Les tableaux à deux dimensions

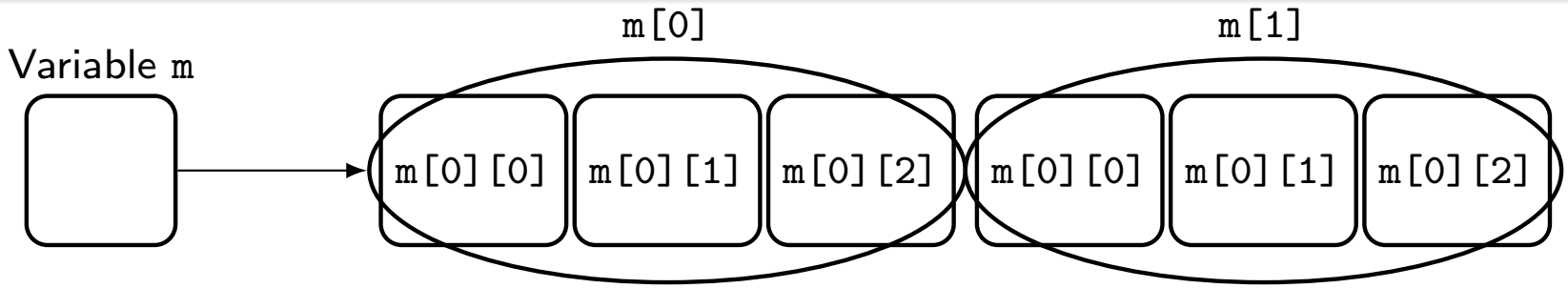
En langage C, un tableau à deux dimensions peut être vu comme un tableau de tableaux, et donc représenté par un pointeur sur pointeur.

Allocation statique

En mémoire, les données sont stockées les unes après les autres, ligne par ligne.

- int m[2][3] ; permet d'allouer l'espace mémoire pour un tableau 2D de taille 2×3 contenant des entiers, sans initialiser ses valeurs.
- int m[2][3] = {{1,5,4}, {3,2,1}} ; permet de faire les deux.
- int m[2][3] = {1,5,4,3,2,1} ; permet de faire la même chose.

Remarque : `m` est de type `int**`.



Remarque : on pourrait "aplatir" le tableau à deux dimensions, en le codant avec un tableau à une dimension. Par exemple, l'instruction `int t[6] = {1,5,4,3,2,1};` permet de représenter le même tableau 2D, en faisant correspondre la case `m[i][j]` avec la case `t[i*n+j]`.

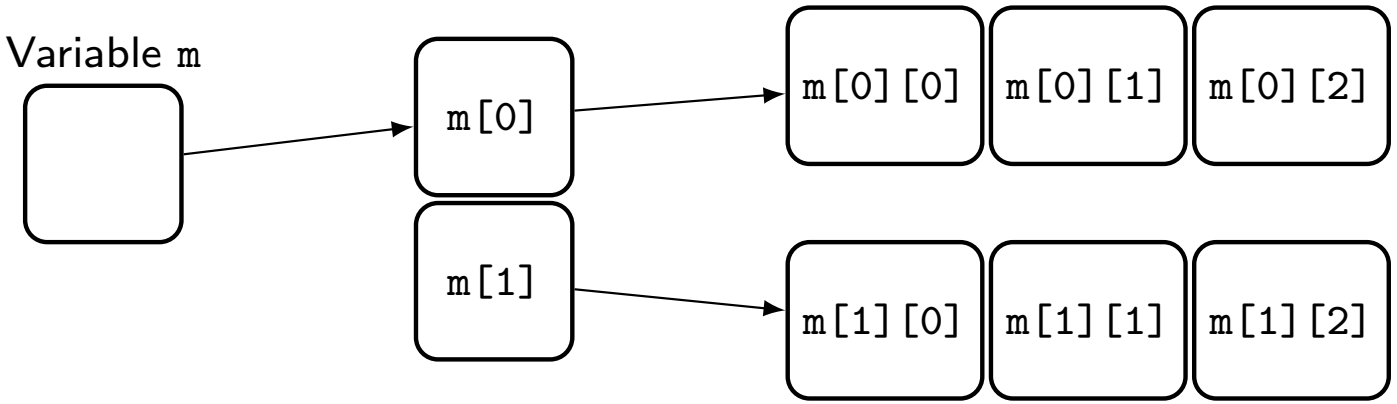
Les tableaux à deux dimensions

On préfère réaliser une allocation dynamique quand la mémoire ne possède pas de zone espace contiguë assez grande.

Allocation dynamique

Allouer un tableau à une dimension, dont chaque case pointe sur un tableau à une dimension. Exemple avec le même tableau à deux dimensions :

```
1 int i;
2 int** m = (int**) malloc(2*sizeof(int*));
3 for (i=0;i<3;i++){
4     m[i] = (int*) malloc(3*sizeof(int));
5 }
```



Remarque : on pourrait ici aussi "aplatir" le tableau à deux dimensions en faisant un seul appel à `malloc` pour allouer un tableau de 6 entiers, mais ce n'est pas vraiment intéressant du point de vue de l'occupation mémoire.

Les tableaux à deux dimensions

Désallocation : libérer les sous-tableaux, avant de libérer le tableau principal.

```
1  int i;
2  for (i=0;i<3;i++){
3      free(m[i]);
4  }
5  free(m);
```

Gestion des problèmes d'allocation : libérer la mémoire allouée avant l'erreur.

```
1  int i,j;
2  int** m = (int**) malloc(2*sizeof(int*));
3  if (m == NULL){
4      return NULL;
5  }
6  for (i=0;i<3;i++){
7      m[i] = (int*) malloc(3*sizeof(int));
8      if (m[i]==NULL){
9          for (j=0;j<i;j++){
10             free(m[j]);
11         }
12         free(m);
13         return NULL;
14     }
15 }
```

Le type `struct` en C

Question

Les tableaux permettent de stocker des données de même type. Comment faire pour rassembler des variables qui peuvent avoir des types différents ?

Une réponse : le type `struct`

En langage C, le type `struct` permet de grouper dans une case mémoire des variables appelées *champs* ou *membres*, potentiellement de types différents. En particulier, cela permet au programmeur de créer un nouveau type de variable, composé de variables de types différents, mais qui ont un lien sémantique fort pour le programmeur. Par exemple, on peut définir le type `struct personne` :

```
1 typedef struct personne {
2     char* nom;
3     char* prenom;
4     int* dateDeNaissance;
5     int sexe;
6     struct personne* conjoint;//un pointeur sur le meme type
7 } Personne;
```

Remarque : Le mot clé `typedef` permet de créer des types synonymes. Ce n'est pas nécessaire, mais sans lui, il faudrait écrire "`struct personne`" à chaque définition de variable, ce qui est un peu lourd... En ajoutant "`typedef [...] Personne;`", on peut simplement écrire "`Personne`" à la place.

Manipulation de struct et allocation mémoire

Accès aux champs

Pour une variable s de type struct, l'accès aux champs se fait avec un point :

```
1 Personne s; // declaration d'une variable de type personne
2 s.sexe = 0; // initialisation du champ ''sexe''
```

Pour une variable p de type pointeur sur struct, on utilise une flèche :

```
1 Personne* p = &s; // recuperation de l'adresse de s
2 p->sexe = 1; // modification du champ ''sexe''
```

Allocation statique VS allocation dynamique

Pour faire une allocation entièrement statique, il faut que dans la déclaration du nouveau type, les tableaux soient alloués statiquement.

```
1 typedef struct personne {
2     char nom[20];
3     char prenom[20];
4     int dateDeNaissance[3];
5     int sexe;
6     struct personne* conjoint;
7 } Personne;
```

Pas besoin pour les allocations dynamiques.

Allocation mémoire de struct

Allocation statique

On peut faire des allocations mémoire et initialiser les champs comme suit :

```
1  struct personne p1 = {"A", "B", {12,5,1995}, 0, NULL};
2  Personne p2 = {"D", "E", {}, 1, &p1};
3  memcpy(p2.dateDeNaissance, dateAcopier, 3*sizeof(int));
4  p1.conjoint = &p2;
5
6  Personne p3;
7  (&p3)->sexe = 1; //on pourrait ecrire p3.sexe = 1;
8  strcpy(p3.nom,"F");
9  strcpy(p3.prenom,"G");
10 p3.dateDeNaissance[0]= 15;
11 p3.dateDeNaissance[1]= 10;
12 p3.dateDeNaissance[2]= 1998;
13 p3.conjoint = NULL;
```

Cette solution n'étant pas très souple au niveau de la gestion mémoire, on peut préférer l'allocation dynamique pour allouer des types élaborées.

Allocation mémoire de struct

```
1  typedef struct personne {
2      char* nom;
3      char* prenom;
4      int* dateDeNaissance;
5      int sexe;
6      struct personne* conjoint;
7  } Personne;
```

Allocation dynamique : allouer aussi les champs contenant des tableaux

On peut faire des allocations mémoire et initialiser les champs comme suit :

```
1  Personne* p = (Personne*) malloc (sizeof(Personne));
2  p->nom = (char*) malloc (sizeof(char)*tailleChaine);
3  strcpy(p->nom,chaineACopier);
4  p->prenom = "B"; //plus court, quand la chaine est donnee
5  p->dateDeNaissance = (int*) malloc (3*sizeof(int));
6  memcpy(p->dateDeNaissance, dateACopier, 3*sizeof(int));
7  p->sexe = 0;
8  p->conjoint = NULL;
```

Remarque : on peut mixer allocation statique et dynamique selon les besoins. Par exemple, le tableau dateDeNaissance sera toujours de taille 3. L'allouer statiquement (en écrivant int dateDeNaissance[3] ;) permet de retirer la ligne 5.

Tableau de struct

Grouper des struct

On a vu que le type `struct` permet de créer des nouveaux types de variables, comme par exemple le type `Personne`. On peut maintenant grouper des variables de ce type dans des tableaux, par exemple pour créer une base de données contenant les personnes inscrites sur un site de rencontre en ligne.

Un exemple de tableau de `struct` :

```
1 Personne tab[2]={{"A","B",{},0,NULL}, {"C","D",{},1,NULL}};
```

Observation

Si chaque élément `struct` est consommateur de mémoire, on ne choisira pas d'allouer de la mémoire contiguë pour ce tableau. On préférera un "tableau de pointeurs sur struct" (le tableau ne contenant que des pointeurs).

Un exemple de tableau de pointeurs sur `struct` :

```
1 Personne p1 = {"A","B",{},0,NULL};
2 Personne p2 = {"C","D",{},1,NULL};
3 Personne** tab = (Personne **) malloc(2*sizeof(Personne*));
4 tab[0]=&p1;
5 tab{1]=&p2;
```