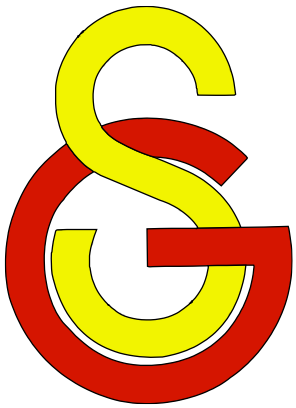
Algorithmique & programmation en langage C - vol.1

Damien Berthet, Vincent Labatut

Université Galatasaray
Faculté d'ingénierie et de technologie



Algorithmique & programmation en langage C

Damien Berthet & Vincent Labatut

Notes de cours

Supports de cours – Volume 1
Période 2005-2014

 Damien Berthet & Vincent Labatut 2005-2014

Galatasaray Üniversitesi
Mühendislik ve Teknoloji Fakültesi
Çırağan Cad. No:36
Ortaköy 34349, İstanbul
Turquie

version 0.9
12/09/2014

Sommaire

Ce document constitue le support de cours écrit pour différents enseignements d'algorithmique et de programmation en langage C donnés à la Faculté d'ingénierie de l'Université Galatasaray (Istanbul, Turquie), entre 2005 et 2014. Il s'agit du premier volume d'une série de 3 documents, comprenant également les recueils des sujets de TP (volume 2) et de leurs corrigés (volume 3).

Bien que ceci n'apparaisse pas spécialement dans ces notes de cours, l'accent a été mis en TP sur l'utilisation de la bibliothèque SDL[1] (*Simple DirectMedia Layer*) afin d'aborder l'algorithmique via une approche graphique. Les outils utilisés en TP (GCC[2] et Eclipse[3]), ainsi que les instructions concernant leur installation, configuration et utilisation, dont décrits dans le volume 2.

Malgré tout le soin apporté à la rédaction de ce support de cours, il est probable que des erreurs s'y soient glissées. Merci de nous contacter afin de nous indiquer tout problème détecté dans ce document. Il faut également remarquer qu'il s'agit d'un cours d'introduction, aussi les notions abordées le sont parfois de façon simplifiée et/ou incomplète.

À titre d'information, le cours se composait en pratique de 24 séances de 2 heures réparties de la façon suivante :

Séance	Sujet abordé	Sections traitées
01	Introduction	1
02	Types simples	2
03	Variables, expressions et opérateurs	3-4
04	Structures de contrôle	5
05	Tableaux	6
06	Définition de fonctions	7-7.2.1
07	Passage de paramètres	7.2.2-7.4
08	Structures et unions	8-8.3
09	Types personnalisés & classes de mémorisation	8.4-9
10	Pointeurs	10
11	Allocation dynamique de mémoire	11
12	Fichiers	12
13	Fonctions récursives	13
14	Listes simplement chaînées	14-14.2
15	Listes doublement chaînées	14.3-14.3.5
16	Piles de données	15
17	Files de données	16
18	Introduction à la complexité	17-17.1
19	Complexité d'algorithmes itératifs	17.2
20	Complexité d'algorithmes récursifs	17.3
21	Tris quadratiques	18-18.4
22	Tris log-linéaires	18.5-18.4
23	Arbres binaires	19-19.2
24	Arbres binaires de recherche	19.3

La gestion des entrées-sorties (affichage écran et saisie clavier) sont absents du cours magistral, et étaient plutôt traités lors du tout premier TP.

[1] https://www.libsdl.org/
[2] http://gcc.gnu.org/
[3] http://www.eclipse.org/

Les principales références bibliographiques utilisées pour préparer ce cours sont les suivantes :

- *Introduction à l'algorithmique*, Thomas Cormen, Charles Leiserson & Ronald Rivest, Dunod, 1994.
- *Méthodologie de la programmation en C, Achille Braquelaire*, Dunod, 4ème édition, 2005.
- *Langage C*, Gerhard Willms, MicroApplication, 1996.

Damien Berthet & Vincent Labatut
le 7 juillet 2014

Conventions

Ce document utilise différentes conventions, dont certaines sont aussi appliquées dans les deux autres volumes de cette série de documents. Tout d'abord, les identificateurs (noms de fonctions, variables, constantes, etc.) sont indiqués en utilisant la police `Courier`.

Le contenu de la console est représenté en utilisant `Courier` sur fond rouge. Lorsque l'utilisateur doit saisir des valeurs, celles-ci sont surlignées en jaune.

```
Entrez une valeur : 12
Vous avez entré la valeur 12.
```

Lorsque du code source est cité, nous employons un fond bleu. Si une partie du code source en particulier doit être désignée, elle est surlignée en vert.

```
int ma_function(int x)
{  int une_variable_notable;
   ...
```

Les points de suspensions présents dans le code source, comme dans l'exemple ci-dessus, indiquent la présence d'instructions dont la nature n'a pas d'importance.

Les définitions données dans le cours sont mises en avant grâce à la mise en forme suivante :

Concept : la définition de ce concept est la suivante…

1 Introduction

Dans cette section introductive, nous abordons d'abord quelques concepts généraux relatifs à l'algorithmique et aux langages de programmations (pas forcément le langage C). Nous présentons ensuite plus particulièrement le langage C, avant d'aborder ses caractéristiques plus en détails dans les sections suivantes.

1.1 Définitions

1.1.1 Notion d'algorithme

Le mot *algorithme* est issu de la déformation du nom d'un savant perse du IX^{ème} siècle appelé *Al Khuwarizmi*[4]. Donc, il n'y a aucun rapport avec le mot *rythme*, ce qui explique l'absence de *y* dans le mot algorithme.

Algorithme : séquence finie d'actions permettant de résoudre un problème donné.

La notion d'algorithmique est directement dérivée du concept d'algorithme :

Algorithmique : ensemble des méthodes permettant de définir et/ou d'étudier des algorithmes.

Les premiers algorithmes sont destinés à résoudre certains problèmes mathématiques simples, par exemple multiplier ou diviser des nombres. Ils étaient appliqués manuellement, et sont antérieurs de plusieurs siècles (voire millénaires) à l'invention des ordinateurs. Ceci permet d'ores et déjà d'établir l'indépendance entre un algorithme et sa mise en œuvre, c'est à dire (dans le cadre informatique) son implémentation.

Il existe également des algorithmes qui n'ont rien à voir avec les mathématiques, comme par exemple les recettes de cuisine. Une recette est une séquence d'instructions à suivre pour réaliser un plat donné, elle est mise en œuvre manuellement par un cuisinier.

1.1.2 Représentation d'un algorithme

Un algorithme est généralement exprimé dans un langage *informel*, ou incomplètement formalisé : texte libre (i.e. description des différentes étapes en français), organigramme (diagramme représentant les étapes), pseudo-code (version simplifiée d'un langage informatique) ou autres. Par opposition, une démonstration mathématique ou un programme informatique sont exprimés en utilisant des langages *formels*[5]. Cela signifie donc que l'écriture d'un algorithme est souple, elle vise à exprimer une méthode de résolution de façon compréhensible à un être humain. Mais pour la même raison, un algorithme ne peut pas être traité directement par un ordinateur : il doit être formalisé, i.e. transformé en un programme.

Il n'existe pas vraiment de norme pour les organigrammes représentant des algorithmes. On peut tout de même mentionner certains points qui font consensus[6] :

- Les étapes sont représentées par des *nœuds* et les transitions par des *liens* orientés entre ces nœuds ;
- Les étapes de *test* sont représentées par des losanges ;
- Les étapes de *début* et de *fin* sont représentées par des rectangles aux coins arrondis ;
- Les étapes de *traitement* sont représentées par des rectangles ;
- Les appels à des *fonctions* ou procédures (aussi appelées *sous-routines*) sont représentés par des rectangles dont les côtés sont dédoublés ;

[4] Ce même savant est aussi à l'origine du mot *algèbre*.
[5] La théorie des langages sera étudiée plus tard dans un cours dédié.
[6] Voir http://en.wikipedia.org/wiki/Flowchart pour plus de détails.

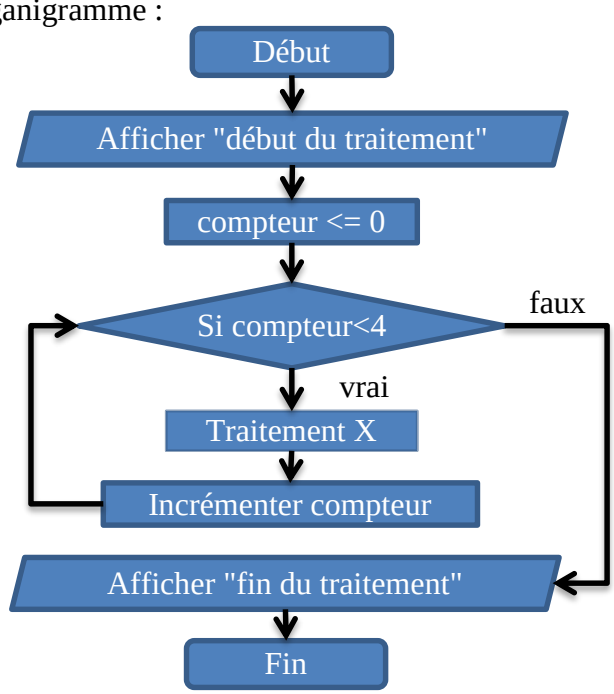
- Les étapes d'*entrée/sortie* sont représentées par des parallélogrammes.

L'inconvénient de cette représentation est qu'il est difficile de décrire des algorithmes complexes tout en gardant un diagramme lisible.

Comme son nom l'indique, le pseudo-code est une façon de décrire un algorithme en utilisant du texte ressemblant fortement à un langage de programmation, mais sans en être un. L'intérêt est de rester informel et indépendant d'un langage de programmation en particulier. Cette représentation permet aussi de ne pas représenter explicitement les détails triviaux de l'algorithme pour se concentrer sur l'essentiel.

exemple : le même algorithme représenté sous forme d'organigramme puis de pseudo-code

- Version organigramme :



- Version pseudo-code :

```
debut
   afficher "début du traitement "
   compteur <- 0
   tant que compteur < 4
      faire un traitement X
      incrementer le compteur
   fin tant que
   afficher "fin du traitement"
fin
```

1.1.3 Diviser pour régner

Dans notre cas, le problème à résoudre est souvent *compliqué*, et l'algorithme permettant de le résoudre n'est pas évident à définir. Pour cela, on utilise en général l'approche appelée *diviser pour régner*. Cette méthode hiérarchique consiste à diviser notre problème complexe en plusieurs sous-problèmes plus simples à résoudre. L'approche s'applique *récursivement* aux sous-problèmes s'ils sont eux-mêmes trop complexes, i.e. on les divise eux-mêmes en sous-problèmes plus simples si nécessaire. On aboutit finalement à des sous-problèmes *élémentaires*, c'est-à-dire qu'on ne peut plus les réduire à des sous-problèmes plus simples. La résolution de ces sous-problèmes est supposée facile, et il en est de même pour la définition de l'algorithme correspondant.

exemple : la table traçante

- **Énoncé du problème :**
  - Une table traçante est un dispositif mécanique constitué d'une table de dessin, d'un stylo et d'un moteur capable de déplacer le stylo.
  - La table traçante possède un `compteur`.
  - Elle est capable d'effectuer les actions suivantes :
    - `lever`/`baisser` : lever le stylo, pour qu'il ne se trouve pas en contact avec la feuille de papier posée sur la table de dessin, ou au contraire le baisser pour pouvoir dessiner sur la feuille.
    - `centrer` : positionner le stylo au centre de la table de dessin.
    - `haut`/`bas`/`gauche`/`droite` : déplacer le stylo d'1 cm dans la direction indiquée.
    - `initialiser`/`incrementer` : initialiser/incrémenter le compteur.
  - Le problème est de dessiner une croix centrée sur la feuille, et dont chaque branche mesure 10 cm.
- **Résolution du problème :**
  - **Algorithme de niveau 1 :**
    - On reprend le problème tel qu'il est posé.

```
début
   dessiner une croix centrée dont les branches mesurent 10 cm
fin
```

  - **Algorithme de niveau 2 :**
    - On décompose l'algorithme précédent.

```
debut
   aller au centre sans tracer
   tracer la branche du haut
   aller au centre sans tracer
   tracer la branche de droite
   aller au centre sans tracer
   tracer la branche du bas
   aller au centre sans tracer
   tracer la branche de gauche
fin
```

  - **Algorithme de niveau 3 :**
    - On décompose `aller au centre sans tracer` en plusieurs sous-problèmes :

```
debut
   lever
   centrer
   baisser
fin
```

    - On décompose `tracer la branche du haut` en plusieurs sous-problèmes :

```
debut
   initialiser
   tant que compteur < 10
      monter
      incrementer
   fin tant que
fin
```

    - On recommence avec les autres actions du niveau 2, et en remplaçant chacune de ces actions par la séquence de niveau 3 correspondante, on obtient l'algorithme de niveau 3.

1.1.4 Programme et compilation

On peut considérer un programme comme la traduction d'un algorithme dans un langage de programmation, i.e. un langage formel compréhensible par un ordinateur. On dit alors que ce programme est l'*implémentation* de cet algorithme.

Programme : séquence d'instructions destinées à être exécutées par un ordinateur.

Instruction : action que l'ordinateur connait et peut réaliser.

Un langage de programmation (y compris le C) est caractérisé par son jeu d'instructions :

Jeu d'instructions : ensemble des instructions autorisées dans un langage de programmation.

Un programme est stocké dans un fichier (un ensemble de données stockées sur un support persistant). On distingue deux types de fichiers, correspondant à deux versions différentes d'un programme : fichier *source* et fichier *exécutable*. Le premier contient ce que l'on appelle le code *source*, il s'agit d'un fichier texte, alors que le second contient le code *binaire*, aussi appelé code *machine* ou code *objet*.

Code source : programme exprimé dans un langage de programmation compréhensible par un être humain, et ne pouvant pas être exécuté directement par l'ordinateur.

Code binaire : programme exprimé en langage machine, pouvant être directement exécuté par l'ordinateur.

Remarque : on utilise souvent le seul terme *programme* pour désigner le code source ou le code binaire. Son interprétation dépend alors du contexte.

Le code binaire est obtenu à partir du code source, en réalisant un traitement appelé *compilation*. Ce traitement est réalisé par un programme spécial appelé le *compilateur*.

Compilation : action de transformer un code source en code binaire, de manière à obtenir un fichier exécutable à partir d'un fichier source.

Le code binaire est spécifique au processeur qui va exécuter le programme. Par contre, le code source en est généralement indépendant. Cela signifie qu'un même code source sera compilé différemment en fonction du processeur (et/ou du système d'exploitation) que l'on veut utiliser pour exécuter le programme.

Remarque : il est important de bien comprendre qu'un code source et l'algorithme qu'il implémente sont deux choses bien distinctes. En effet, le même algorithme peut être implémenté différemment, en fonction de :

- La personne qui écrit le programme ;
- Le langage de programmation employé ;
- La machine et/ou le système d'exploitation utilisés ;
- Le niveau de précision de l'algorithme ;
- etc.

La procédure de compilation varie en fonction du langage de programmation. On s'intéresse ici au cas du langage C, décrit dans la figure ci-dessous. Notez que le processus de compilation sera étudié plus en détail dans un cours dédié.

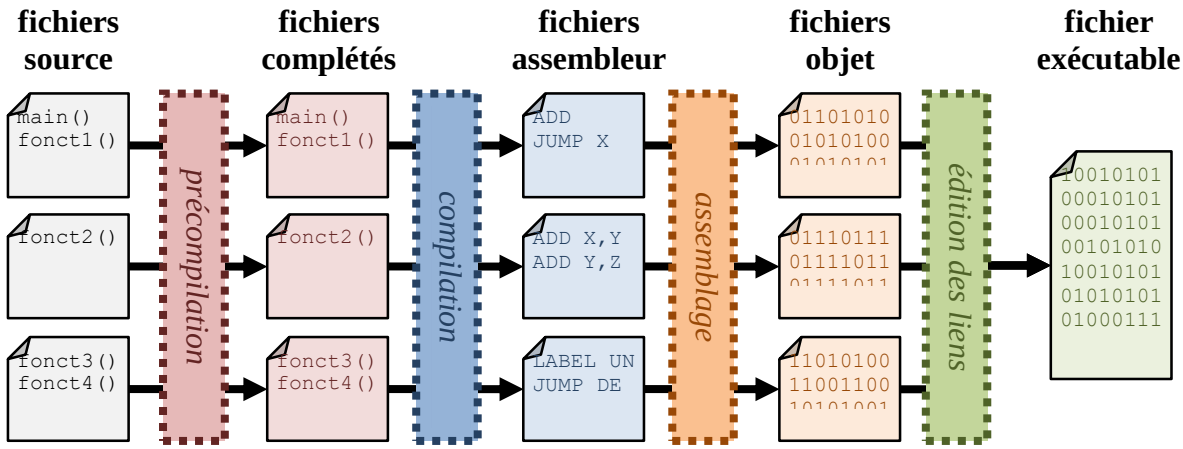
Un programme C prend généralement la forme de plusieurs fichiers source distincts, que l'on veut compiler de manière à obtenir un seul fichier exécutable. La décomposition d'un programme en plusieurs fichiers séparés permet de le rendre modulaire, ce qui facilite la maintenance, le débogage et l'évolution du programme.

La *première* étape (en rouge) consiste à *précompiler* les fichiers source, en appliquant les directives de pré-compilation indiquées dans le code source au moyen du caractère `#`. Ces directives seront décrites plus tard, mais on peut déjà dire que le travail de pré-compilation consiste à transformer et compléter le code source. On obtient ainsi une *version complétée* de chaque fichier source traité. Les fichiers obtenus contiennent toujours du code source.

La *deuxième* étape (en bleu) est la *compilation* proprement dite. Cependant, elle ne produit pas directement du code bianire, mais un code source intermédiaire, utilisant le langage *assembleur*. Ce langage est qualifié de bas-niveau, car son jeu d'instructions correspond aux actions les plus simples qu'un processeur peut effectuer. Par conséquent, un programme assembleur se traduit très facilement en un programme binaire. À ce stade de la procédure, on obtient autant de fichiers assembleur qu'on avait initialement de fichiers source.

La *troisième* étape (en orange) est appelée l'*assemblage*, et consiste à transformer chaque fichier assembleur en un fichier binaire. Là encore, on obtient autant de fichiers binaires que de fichiers source originaux. Ces fichiers prennent généralement l'extension `.o`.

La *quatrième* et dernière étape (en vert) est l'*édition des liens*. Elle consiste à rassembler dans un seul fichier binaire tout le code binaire nécessaire à l'exécution du programme. Ceci inclut à la fois la totalité ou une partie des fichiers objets de l'étape précédente, mais aussi éventuellement de bibliothèques statiques préexistantes. On obtient finalement un seul fichier que l'on peut exécuter.



Remarque : par *abus de langage*, on appelle *compilation* l'ensemble de la procédure permettant d'obtenir un fichier binaire à partir d'un ou plusieurs fichiers sources, alors qu'en réalité, la compilation proprement dite n'est qu'une étape de cette procédure (ici, la deuxième étape).

1.1.5 Erreurs de compilation et d'exécution

On distingue deux types de problèmes pouvant survenir lors de l'écriture d'un programme, en fonction du moment où elles sont détectées : compilation ou exécution.

Erreur de compilation : problème détecté lors du processus de compilation.

Cette erreur peut être détectée lors de l'étape de compilation proprement dite, mais elle peut aussi apparaitre lors de la pré-compilation (auquel cas il y a probablement un problème avec les directives, par exemple un `;` placé incorrectement à la fin d'une ligne de directive) ou d'une autre étape du processus.

Le compilateur peut indiquer un simple *avertissement*, auquel cas le problème détecté n'est que potentiel, et n'empêche pas la création d'un fichier binaire qui pourra ensuite être exécuté.

Ce type de problème est généralement indiqué par l'IDE[7] en soulignant la portion de code source concernée en jaune ou orange. Dans le cadre de nos TP, il est obligatoire de corriger les simples avertissements, au même titre que les erreurs.

Le compilateur peut indiquer une erreur proprement dite, qui empêche de finaliser la compilation. Dans ce cas-là, il n'est pas possible de créer le fichier exécutable correspondant à votre programme. *Il est donc inutile de tenter d'exécuter votre programme si la compilation a échoué*. Ces erreurs-là sont indiquées en rouge par l'IDE. À noter que des options du compilateur permettent de considérer certains problèmes soit comme des avertissements, soit comme de vraies erreurs.

Des messages d'information sont indiqués pour chaque avertissement et erreur de compilation, afin d'aider le programmeur à résoudre le problème identifié. Cependant, il faut savoir que ces messages ne sont pas toujours très explicites, et peuvent parfois aussi être trompeurs (cela dépend du compilateur). Il est nécessaire d'avoir une certaine expérience du compilateur utilisé pour être capable de les interpréter correctement. De plus, une erreur de compilation peut en provoquer d'autres. Pour cette raison, il faut absolument commencer par corriger la *toute première* erreur détectée par le compilateur, puisqu'elle est susceptible de causer les erreurs ultérieures.

Erreur d'exécution : erreur se produisant quand le programme n'a pas le comportement attendu lors de son exécution.

Les erreurs d'exécution ne sont pas détectées par le compilateur, mais par l'utilisateur. Autrement dit, leur détection n'est pas automatique, mais manuelle. Par conséquent, rien n'indique précisément la partie du code source qui pose problème : le débogage se fait complètement manuellement. Ces erreurs sont donc beaucoup plus difficiles à corriger que les erreurs de compilations.

On distingue deux types d'erreurs d'exécution : les erreurs *fatales* et les erreurs *non-fatales*. Les premières provoquent la terminaison non-naturelle du programme. Autrement dit, votre programme se termine alors qu'il ne devrait pas. C'est par exemple le cas si vous effectuez une division par zéro. Dans ce cas-là, il est facile de détecter la présence d'une erreur, puisque le programme se termine intempestivement. Par contre, localiser la partie du code source concernée n'est pas forcément simple. Les erreurs *non-fatales* sont beaucoup plus difficiles à détecter, car l'exécution continue, et la conséquence de ce type d'erreur peut se manifester bien après l'apparition de l'erreur elle-même.

Remarque : le volume 2 décrit des méthodes de débogages permettant d'identifier les erreurs d'exécution.

1.2 Présentation du langage C

L'écriture de programmes en C nécessite de comprendre un grand nombre de concepts tous liés les uns aux autres. Pour cette raison, il n'est pas possible de tous les étudier en détails d'entrée. Dans cette partie, on se contentera donc d'introduire les notions essentielles de façon superficielle, afin de pouvoir progresser dans le cours. La plupart de ces notions seront ensuite revues plus tard, de façon plus approfondie.

1.2.1 Historique

Le langage C est un langage de programmation développé à partir de 1972 dans les laboratoires Bell, dans le but d'écrire le système d'exploitation Unix[8]. Sa paternité est attribuée

[7] *Integrated development environment* (environnement de développement intégré), il s'agit du logiciel utilisé pour écrire le code source. Pour nos TP, nous utiliserons Eclipse (cf. le volume 2).

[8] Qui sera étudié plus tard dans le cours portant sur les systèmes d'exploitation.

à Dennis Ritchie et Ken Thompson, tandis que Brian Kernighan est intervenu plus tard dans son évolution. L'un des objectifs était de définir un langage portable.

Portabilité : un langage de programmation est dit *portable* s'il peut être compilé sur des machines et de systèmes différents, *sans avoir besoin d'être modifié*.

Cependant, on ne peut pas vraiment considérer que cet objectif a été atteint, car différentes versions du C sont apparues, qui ne sont pas complètement compatibles entre elles. Les principales sont les suivantes :

- La version originale s'est stabilisée en 1978 et est appelée *Kernighan & Ritchie* ou *C K&R* ;
- L'ANSI[9] a défini sa propre version en 1989, basée sur le C K&R et appelée *C89* ou *C ANSI* ;
- L'ISO[10] a défini sa version basée sur le C ANSI en 1999, qui est appelée C99

Dans ce cours, nous nous concentrerons sur la version C89, qui est d'après nous la plus répandue. Nous indiquerons occasionnellement les points traités différemment dans les versions ultérieures.

On qualifie parfois le C de langage *bas niveau* car il est relativement proche de l'assembleur. Toutefois, il est de plus haut niveau que ce dernier. Il s'agit du langage le plus utilisé au monde[11], ou du moins figure-t-il parmi les trois premiers, en fonction du classement considéré. Sa syntaxe a fortement influencé d'autres langages de plus haut niveau apparus plus tard, tels que C++, Java, C#, JavaScript ou PHP. Un grand nombre de systèmes d'exploitation et de jeux vidéo sont encore (complètement ou partiellement) écrits en C aujourd'hui.

Remarque : l'utilisation de lettres pour nommer un langage est une pratique courante en informatique. Ainsi, le langage C est basé sur B, et il existe aussi les langages D, F#, G, J, K, M, S, R, X++, Z….

1.2.2 Notion de fonction

Par définition, un programme est une séquence d'instructions. À noter qu'en langage C, chaque instruction se termine par un point-virgule (i.e. `;`). Pour améliorer la lisibilité du code source et sa gestion, il peut être pratique de découper un programme en plusieurs parties relativement indépendantes. Cela se fait grâce à la notion de *bloc*.

Bloc d'instructions : sous-séquence d'instructions, délimitée par des accolades (i.e. `{` et `}`).

exemple : les instructions 2, 3 et 4 suivantes sont contenues dans un bloc distinct

```
instruction 1;
{  instruction 2;
   instruction 3;
   instruction 4;
}
instruction 2;
```

Il est aussi possible de définir un bloc *à l'intérieur* d'un autre bloc, et de répéter cela récursivement. On peut alors parler de *sous-bloc*.

exemple : les instructions 3 et 4 sont maintenant dans un sous-bloc :

[9] Institution américaine chargée de la création de normes.
[10] Comme l'ANSI, mais au niveau mondial.
[11] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

```
instruction 1;
{  instruction 2;
   {  instruction 3;
      instruction 4;
   }
}
instruction 2;
```

Lorsqu'on a besoin d'utiliser plusieurs fois un bloc donné, on peut lui donner un nom, afin de le désigner facilement. On obtient alors ce qu'on appelle une *fonction*. Outre son nom, une fonction est aussi caractérisée par ses entrées/sorties, i.e. les informations qu'elle reçoit pour effectuer son traitement (entrées), et celles qu'elle renvoie à la fin de ce traitement (sorties).

Fonction : bloc d'instructions identifié par un nom, des entrées et des sorties.

Le nom de la fonction est aussi appelé *identificateur*. À noter que cette notion d'identificateur est approfondie en section 1.2.5. Le bloc d'instructions qui décrit le traitement effectué par la fonction est appelé *corps de la fonction*, alors que son identificateur et ses entrées-sorties sont décrits sous la forme d'un *en-tête*.

Corps de fonction : bloc d'instructions implémentant le traitement réalisé par la fonction.

En-tête de fonction : type de retour, identificateur (nom) et paramètres de la fonction.

Le *type de retour* décrit quel genre d'information la fonction calcule : cela peut être rien du tout (noté `void`) si la fonction ne renvoie rien, ou bien une valeur entière, une valeur réelle, ou encore d'autres données. La notion de *type* sera abordée plus en détail dans la section 2. Les *paramètres* de la fonction sont des *variables* que la fonction reçoit et sur lesquelles elle peut travailler. Elles sont indiquées entre parenthèses (i.e. `(` et `)`) et séparées par des virgules. Pour chacune, on doit préciser son type (comme pour le type de retour) et son identificateur (i.e. son nom). La notion de variable sera approfondie en sections 2 et 3. Le nombre de paramètres n'est pas limité, il peut même ne pas y avoir de paramètre du tout, et on a alors des parenthèses vides : `()`.

exemple : une fonction appelée **`ma_fonction`**, dont le type de retour est **`type_r`** et qui possède trois paramètres **`p1`** à **`p3`** de types respectifs **`type_p1`** à **`type_p3`**. Les couleurs sont destinées à faciliter l'identification de ces différentes composantes.

```
type_r ma_fonction(type_1 p1, type_2 p2, type_3 p3)
{  instruction_1;
   instruction_2;
   ...
}
```

1.2.3 Fonctions principales et secondaires

Un programme en langage C prend essentiellement la forme d'une *séquence de fonctions* (plus quelques autres choses que nous verrons plus tard). La notion de fonction sera explorée en détails plus tard, en section 7.

Ce programme doit obligatoirement contenir une *fonction principale*, dont le nom est obligatoirement `main`. En règle générale, un programme ne peut pas contenir plusieurs fonctions portant le même nom. La fonction `main` a, de plus, un type de retour et des paramètres imposés :

```
int main(int argc, char** argv)
```

```
{  instruction 1;
   instruction 2;
   ...
}
```

Le type de retour (`int`) et les deux paramètres (`int argc` et `char** argv`) seront expliqués plus tard en cours.

On appelle *programme minimal* le programme le plus court possible autorisé par les règles d'un langage de programmation donné. Dans le cas du langage C, il s'agit du programme suivant, que vous pouvez copier-coller dans votre IDE puis compiler :

```
int main(int argc, char** argv)
{
}
```

Notez que ce programme ne fait absolument rien du tout !

exemple : un programme composé de trois fonctions secondaires `f1`, `f2`, `f3` et d'une fonction principale :

```
t_a f1(t_b p1, t_c p2)
{  ...
}

void f2(t_a p1)
{  ...
}
```

```
t_c f3()
{  ...
}

int main(int argc, char** argv)
{  ...
}
```

1.2.4 Commentaires et lisibilité

Comme dans la plupart des langages de programmation, il est possible de définir des *commentaires* en langage C.

Commentaire : un texte exprimé en langage naturel (par exemple en français ou en anglais) et dont le but est d'expliquer le fonctionnement du programme à un programmeur.

Ce programmeur peut être celui qui écrit le code source, ou une autre personne. Les commentaires permettent d'insérer dans le programme des indications précieuses destinées à un être humain, et ils sont par conséquent complètement ignorés par le compilateur. Les IDE représentent généralement les commentaires d'une couleur différente, pour les mettre en valeur.

En C, un commentaire est délimité par `/*` et `*/`, et peut occuper plusieurs lignes. On peut aussi faire des commentaires d'une seule ligne en insérant `//` en début de ligne.

exemples :

- Commentaire de plusieurs lignes :

```
/*
   Bla bla bla bla
   bla bla bla bla
   bla bla
*/
```

- Commentaire d'une seule ligne (seulement à partir de la version C99) :

```
// Bla bla bla bla bla bla
```

De manière générale, il est très important de mettre des commentaires lorsqu'on programme, car cela améliore la lisibilité du code.

Lisibilité du code source : propriété d'un code source à être facilement compris par une personne qui ne l'a pas forcément écrit.

En TP (ainsi qu'en examen de TP), nous utiliserons également les commentaires pour indiquer à quel exercice correspond quelle partie du programme, et pour répondre à certaines questions[12]. L'*indentation* du code est un autre facteur affectant la lisibilité du code.

Indentation : décalage introduit en début de ligne pour aligner horizontalement certaines lignes du code source.

Différentes conventions existent en ce qui concerne l'indentation. Dans ce cours, nous avons opté pour l'alignement des accolades :

- Toute accolade fermante `}` doit être alignée avec l'accolade ouvrante `{` correspondante ;
- Toute instruction doit être alignée avec la première instruction appartenant au même bloc.

exemple : tous les exemples précédents sont corrects. Voici un exemple incorrect à gauche (les erreurs sont indiquées en rouge) et le même programme correctement indenté à droite :

```
instruction 1;{
   instruction 2;
   {  instruction 3;
instruction 4;
      instruction 5;
}
   instruction 6;
}
```

```
instruction 1;
{  instruction 2;
   {  instruction 3;
      instruction 4;
      instruction 5;
   }
   instruction 6;
}
```

Une bonne méthode générale de programmation consiste à procéder de la façon suivante :

1) Définir un algorithme pour résoudre le problème à traiter. On utiliser l'approche *diviser-pour-régner* (cf. section 1.1.3) jusqu'à atteindre un niveau de détail satisfaisant.
2) On recopie cet algorithme sous forme de commentaires dans le code source, qui ne contient pour l'instant aucune instruction.
3) On complète le code source en ajoutant, en-dessous de chaque commentaire, les instructions permettant d'effectuer l'action qu'il décrit.

1.2.5 Identificateurs et mots-clés

Nous revenons ici sur la notion d'identificateur, que l'on a déjà mentionnée pour les fonctions. Il s'agit cependant d'un concept plus général.

Identificateur : nom donné par le programmeur à un élément de programmation : fonction, variable, constante symbolique, type, etc.

La règle principale pour un identificateur, est qu'il doit être *unique*, dans une certaine mesure qui dépend de l'élément concerné : variable, constante, fonction, etc. Pour une fonction, l'identificateur doit être unique pour le programme entier, en considérant tous les fichiers source qui constituent le programme : fichier principal et bibliothèques (cf. section 1.2.7). Pour une variable, l'identificateur doit être unique dans le bloc qui la contient et dans tous ses sous-blocs.

Un identificateur doit aussi respecter un certain nombre de contraintes pour être valide. Pour simplifier, on utilisera les règles suivantes dans le cadre de ce cours :

- On utilisera seulement des lettres (minuscules et majuscules), des chiffres et le caractère tiret-bas (aussi appelé underscore : _) ;
- On n'utilisera pas de signes diacritiques sur les lettres (accents, trémas, tilde, etc.)
- Le premier caractère de l'identificateur sera toujours une lettre.

[12] Ceci sera expliqué en TP, cf. le volume 2.

Remarque : le compilateur C fait la distinction entre lettres minuscules et majuscules. Par exemple, les identificateurs `var` et `VAR` sont considérés comme différents.

Les conventions suivantes sont généralement utilisées en C (ce qui n'est pas forcément le cas d'autres langages). Nous les utiliserons nous aussi, en particulier en TP :

- Les noms des variables et fonctions ne contiennent pas de majuscules (ex. : `var1`)
- Les constantes symboliques ne contiennent pas de minuscules (ex. : `CST`)
- On utilise le caractère underscore pour séparer les mots (ex : `ex_de_var`)

Quand on choisit le nom d'un identificateur, il faut que ce nom ait un sens, ce qui permet d'améliorer la lisibilité du code source. Le but de l'identificateur est d'indiquer à une personne qui lit le programme quelle est la signification de la variable ou fonction concernée. Un identificateur sans aucun sens rendra votre code source plus difficile à comprendre. Or, une bonne évaluation de votre travail passe nécessairement par une bonne compréhension de votre code source par le correcteur qui vous lit. Il est donc dans votre intérêt de faire un effort pour faciliter son travail.

Un certain nombre de mots ne peuvent pas être utilisés comme identificateurs. On les appelle *mots protégés* ou *mots réservés*. Il s'agit essentiellement des *mots-clés* du langage C.

Mot-clé d'un langage informatique : mot associé à une action particulière dans le cadre de ce langage.

exemples : quelques mots clés du C : `for`, `if`, `else`, `return`…

Puisque le mot clé est déjà associé à une sémantique précise, il est normal qu'on interdise son emploi à un autre effet, comme par exemple en tant qu'identificateur. À noter que les mots clés sont généralement représentés d'une couleur spécifique par les IDE.

1.2.6 Directives de pré-compilation

En dehors des fonctions mentionnées, il est possible de placer dans le programme ce que l'on appelle des *directives de pré-compilation*. Comme expliqué en section 1.1.4, il s'agit d'instructions traitées lors de la première étape du processus de compilation. Pour cette raison, elles ne suivent pas la même syntaxe que les instructions qui s'adressent au compilateur.

Chaque directive doit :

- Être précédée du caractère dièse (i.e. `#`).
- Ne *pas* être terminée par un point-virgule (i.e. `;`).

Dans le cadre de ce cours, on s'intéressera essentiellement aux deux directives `#include` et `#define`. La première permet de faire référence à des programmes écrits dans d'autres fichiers. Il peut s'agir de programmes écrits par la même personne, ou bien par d'autres. Les noms de ces programmes doivent prendre la forme `xxxxx.h`, telle qu'elle est décrite plus en détails dans la section 1.2.7. On distingue deux formes pour cette directive :

- Une forme *locale*, dans laquelle le nom du fichier est entouré de *guillemets* (i.e. `"` et `"`) :

```
#include "xxxxx.h"
```

- Une forme *système*, dans laquelle le nom du fichier est entouré des signes mathématiques < et >.

```
#include <xxxxx.h>
```

Dans le premier cas, le fichier sera recherché dans le même dossier que votre propre programme. On dit qu'il est *local* à votre programme. Dans le second cas, le fichier sera

recherché dans les chemins définis au niveau de votre *système d'exploitation* (cf. la section décrivant l'installation du C dans le volume 2).

La seconde directive permet de définir des *constantes symboliques*, que l'on appelle aussi parfois des *macros*.

Constante symbolique : identificateur associé à une valeur qui ne change pas au cours de l'exécution du programme.

Pour rappel, l'identificateur utilisé pour désigner une constante ne doit pas contenir de lettres minuscules. La définition se fait en utilisant la syntaxe suivante :

```
#define XXXXX valeur
```

Où `XXXXX` est l'identificateur et valeur est la valeur associée. Par exemple, pour définir une constante `C` de valeur 5, on écrira :

```
#define C 5
```

Cette directive fonctionne de la façon suivante : lors de la pré-compilation, à chaque fois qu'on trouve `XXXXX` dans le code source, cette expression est remplacée par `valeur`. Il est donc complètement impossible de modifier `XXXXX` lors de l'exécution.

Encore une fois, notez bien l'absence du signe `;` à la fin de la ligne. Remarquez aussi qu'à la différence des variables, on n'utilise pas ici l'opérateur d'affectation =.

Ces deux directives sont à placer au tout début de votre programme.

exemple : la fonction `printf` permet d'afficher du texte à l'écran[13]. Pour l'utiliser, il faut mentionner que le programme utilise la bibliothèque `stdio.h` (Standard Input/Output = entrées/sorties standards). Supposons également que l'on veut définir une constante `CSTE` de valeur 99. Alors on écrira (**en gras**) :

```
#include <stdio.h>
#define CSTE 5

int ma_fonction()
{  ...
}

int main(int argc, char** argv)
{  printf("j'affiche du texte");
   ...
}
```

1.2.7 Modularité des programmes

Comme expliqué précédemment, le langage C permet de décomposer un programme en plusieurs fichiers relativement indépendants, qui sont rassemblés lors de la compilation (cf. section 1.1.4).

On a tout d'abord un fichier principal, qui contient la fonction principale `main`. Celui-ci doit obligatoirement être défini. Son nom n'est pas contraint, mais par convention nous l'appellerons `main.c`. Outre la fonction `main`, il peut contenir des fonctions secondaires.

Le reste des fichiers formant le programme sont ce que l'on appelle des *bibliothèques*.

Bibliothèque : paire de fichiers (`.h` et `.c`) définissant un ensemble de fonctionnalités thématiquement liées.

Une bibliothèque nommée `xxxxx` est formée de deux fichiers :

- `xxxxx.c`, qui contient essentiellement les fonctions complètes ;

[13] Tout cela sera étudié plus en détails lors du 1er TP.

- `xxxxx.h` qui contient essentiellement les en-têtes des fonctions (i.e. pas leurs corps).

Le second est d'ailleurs appelé *fichier d'en-tête*, ce qui se traduit en *header file* en anglais, d'où l'extension en `.h`. Outre les en-têtes des fonctions, le fichier `.h` peut également contenir des types, des variables, des constantes destinées à être utilisées dans d'autres programmes (ces notions seront abordées plus tard dans ce cours). Le fichier `.c` contient non seulement les fonctions complètes (avec leur corps), mais aussi tous les autres éléments qui ne sont pas destinés à être utilisés à l'extérieur de la bibliothèque.

Comme nous l'avons indiqué précédemment, pour pouvoir utiliser une bibliothèque dans un autre programme, il est nécessaire d'utiliser la directive de pré-compilation `#include`. Mais cette directive sert aussi à lier les fichiers `.c` et `.h` composant une bibliothèque : elle doit apparaître au début du fichier `.c`.

exemple : une bibliothèque très simple, appelée `xxxx`, avec :

- Le fichier `xxxx.h` :

```
int ma_fonction(int x);
```

- Et le fichier `xxxx.c` :

```
#include "xxxx.h"

int ma_fonction(int x)
{  instruction 1;
   instruction 2;
   ...
}
```

Vous remarquerez que dans le fichier d'en-tête `.h`, l'en-tête de la fonction `ma_fonction` est terminé par un point-virgule (`;`), alors que ce n'est pas le cas dans le fichier `.c`.

Si on veut utiliser cette bibliothèque dans un programme principal `main.c`, alors il faut rajouter la directive `#include "xxxx.h"` au début de ce programme `main.c`.

Le principe d'inclusion est *récursif*, ce qui signifie qu'il est tout à fait possible de définir une bibliothèque qui utilise une autre bibliothèque, qui en utilise une troisième, et ainsi de suite. Cela peut produire des situations assez complexes, où la même bibliothèque est utilisée par plusieurs autres bibliothèques utilisées simultanément. Or, si on réalise plusieurs `#include` de la même bibliothèque, cela signifie que ses fonctions apparaitront plusieurs fois dans notre programme à la suite de la pré-compilation. Et nous savons déjà (cf. section 1.2.5) que l'identificateur d'une fonction doit être *unique* pour le programme entier, ce qui n'est pas le cas ici. Il faut donc éviter de faire plusieurs `#include` de la même bibliothèque.

On utilise pour cela des directives particulières, qualifiées de *conditionnelles* : `#ifndef` et `#endif`. La première permet de tester si une certaine constante existe. Si c'e n'est pas le cas, alors toute la portion de code source qui suit, jusqu'à `#endif`, est traitée normalement par le pré-compilateur. Sinon (i.e. la constate existe déjà), cette portion de code source est ignorée par le pré-compilateur. Ces directives sont utilisées dans le fichier d'en-tête d'une bibliothèque pour éviter de la traiter plusieurs fois si elle est concernée par plusieurs `#include`. Le principe est le suivant : avec `#ifndef`, on teste si une certaine constante caractéristique de la bibliothèque existe (en général, on utilise le nom de la bibliothèque). Si elle n'existe pas, alors on définit cette constante, puis traite la bibliothèque. Au contraire, si la constante existe déjà, cela signifie que la bibliothèque a déjà fait l'objet d'un `#include`, et on ne la traite pas.

exemple : pour notre bibliothèque `xxxx` précédente :

```
// est-ce que la constante de la bibliotheque existe déjà ?
#ifndef XXXX_H
// si non, on la définit
#define XXXX_H
```

```
// puis on définit notre bibliothèque normalement
// (y compris avec d'autres #include, si c'est necessaire)
int ma_fonction(int x);

// reste de la bibliothèque
...

// pour terminer, on indique la fin de #ifndef
#endif
```

Vous noterez que la directive `#define` définit ici une constate sans valeur : en effet, ici (à la différence de l'utilisation de `#define` faite en section 1.2.6), on n'a pas besoin d'affecter une valeur particulière à `XXXX_H`, mais juste de la créer.

On appelle *bibliothèque standard* une bibliothèque qui fait partie de la norme définissant le C. Ces bibliothèques contiennent donc des fonctions et types supposées accessibles à n'importe quel compilateur C.

Celles que nous allons utiliser en TP sont les suivantes :

- `stdlib.h` : (standard library) fonctions diverses ;
- `stdio.h` : (standard input-output) afficher du texte à l'écran, saisir du texte au clavier ;
- `math.h` : fonctions mathématiques (valeur absolue, puissance, cosinus, etc.) ;
- `time.h` : fonctions liées aux temps (heure, date, etc.).

Bien que standard, ces bibliothèques doivent faire l'objet d'un `#include` avant de pouvoir être utilisées dans un programme.

2 Types de données

La notion de type de données est fondamentale en programmation, et plus généralement en informatique. Elle est liée à la façon dont l'information est représentée dans la mémoire d'un ordinateur. La première sous-section introduit des notions proche de l'architecture matérielles, nécessaires pour comprendre le fonctionnement de base de la mémoire et les contraintes qui en découlent. Les sous-sections suivantes se concentrent sur différents types de données : d'abord les valeurs numériques, avec les entiers et les réels, puis les informations qui ne sont pas numériques par nature, avec le texte et les images. Pour chacune, nous décrivons différents codages génériques et les type formels utilisés plus particulièrement en langage C.

2.1 Représentation de l'information

Dans cette sous-section, nous allons d'abord étudier comment la mémoire d'un ordinateur est structurée, et comment on peut y stocker des données, de façon générale. Puis, nous aborderons la notion de type de données, particulièrement dans le cas du langage C.

2.1.1 Organisation de la mémoire

L'unité atomique d'information dans un ordinateur est le bit :

Bit : binary digit, i.e. en français : *nombre binaire*. Il s'agit de la plus petite division de la mémoire, qui a deux états possibles : soit la valeur 0, soit la valeur 1.

La mémoire de l'ordinateur est composée d'une séquence de bits, donc son état est lui-même une séquence de valeurs binaires.

Dans les ordinateurs modernes, les bits sont manipulés par 8, et non pas un par un :

Octet : groupe de 8 bits *adjacents* correspondant à un emplacement mémoire désigné par une adresse unique.

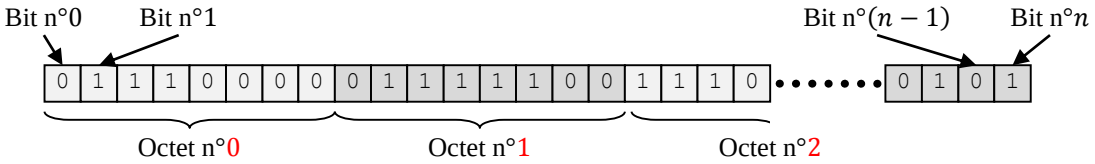
Par conséquent, le nombre total de bits constituant la mémoire est forcément un multiple de 8.

Adresse : numéro unique indiquant la position d'un octet dans la séquence d'octets constituant la mémoire de l'ordinateur.

L'adresse d'un octet permet de le distinguer de tous les autres octets constituant la mémoire. Elle sert à localiser l'octet, et à y accéder, c'est-à-dire :

- Lire son contenu, i.e. connaitre les valeurs des 8 bits qui le constituent ;
- Modifier son contenu, i.e. changer la valeur de ces mêmes bits.

Le schéma suivant illustre les notions de bit, octet et adresse :



Une information représentée dans la mémoire d'un ordinateur occupera donc un certain nombre d'octets. Si on veut par exemple représenter une valeur nécessitant 10 bits, on devra utiliser 2 octets (soit 16 bits). En effet, on ne peut pas occuper seulement 10 bits (les 8 bits d'un premier octet et 2 bits d'un autre octet), car le système d'adressage ne permet pas de distinguer les bits à l'intérieur d'un octet.

L'octet (noté o) sert d'unité de base pour mesurer la taille d'une mémoire ou des données qu'elle contient. Un octet vaut 8 bits (b), i.e. $1\,o = 8\,b$. Les unités supérieures sont

traditionnellement des puissances de 2 : un kilo-octet (*ko*) correspond à $2^{10} = 1024$ octets, un méga-octet (*Mo*) à 2^{10} *ko*, soit 2^{20} octets, un giga-octet à 2^{10} $Mo = 2^{20}$ $ko = 2^{30}$ o, etc.

Cependant, cette utilisation des préfixes du système métrique est erronée, puisque 2^{10} est seulement approximativement égal à 1000. Pour cette raison, ces unités ont été officiellement redéfinies de manière à utiliser des puissances de 10. De nouveaux noms ont été donné aux unités préexistantes, basées sur des puissance de 2, comme indiqué dans le tableau suivant[14]. Il faut néanmoins noter que l'usage de ces nouveaux noms n'est pas répandu.

Nom	Symbole	Taille
Kilo-octet	ko	10^3
Méga-octet	Mo	10^6
Giga-octet	Go	10^9
Tera-octet	To	10^{12}
Peta-octet	Po	10^{15}

Nom	Symbole	Taille
Kibi-octet	kio	2^{10}
Mébi-octet	Mio	2^{20}
Gibi-octet	Gio	2^{30}
Tebi-octet	Tio	2^{40}
Pebi-octet	Pio	2^{50}

Remarque : l'étude du fonctionnement de la mémoire d'un ordinateur sera réalisée de façon plus détaillées dans les cours d'architecture et de systèmes d'exploitations.

2.1.2 Notion de type de données

Puisqu'une mémoire d'ordinateur ne peut contenir que des données binaires, il faut maintenant décider comment représenter des informations en mémoire. On s'appuie pour cela sur la notion de *type de données*, qui définit comment une certaine catégorie d'information est représentée en mémoire, comment on y accède et comment on la modifie.

Type de données : association d'un identificateur, d'un ensemble de règles de représentation, et de manipulation.

Comme nous l'avons vu précédemment, un identificateur est un nom permettant de désigner facilement un concept dans un programme (et ici, ce concept est un type). Par exemple, le langage C contient un type appelé `int` et utilisé pour représenter des nombres entiers.

Les règles de représentation décrivent la *forme* que l'information doit prendre en mémoire. Cette forme est appelée un *code*. La règle permettant de transformer l'information en un code est appelée le *codage*.

Code : représentation d'une information en mémoire, conformément aux règles définies pour un type donné.

Nous allons voir plus tard qu'il existe plusieurs façons de représenter une information donnée. Par exemple une simple valeur numérique comme 66 peut être représentées par plusieurs codes binaires différents. Certaines de ces représentations sont plus pratiques ou efficaces que d'autres.

Les règles de manipulation indiquent comment les codes d'un certain type doivent être manipulés. Par exemple, on ne multiplie pas deux nombres *entiers* de la même façon qu'on multiplie deux nombres *réels*.

Remarque : la notion de type est au cœur de la programmation. Il s'agit un concept de première importance qui reviendra souvent, à la fois en cours et en TP, tout le long du semestre, et même dans d'autres cours que ceux de programmation.

À noter que le type définit aussi, à travers les règles de représentation, l'*espace* qu'une information prend en mémoire (i.e. combien d'octets elle occupe). L'opérateur `sizeof(xxxx)`

[14] Voir aussi http://fr.wikipedia.org/wiki/Octet

permet de déterminer la taille d'une valeur de type `xxxx`, exprimée en octets (cf. section 4.2.6). Toute information manipulée par le langage C possède forcément un type, même s'il est parfois implicite.

On distingue deux sortes de types en C : les types *simples* et les types *complexes*. Dans cette section, nous nous concentrons sur les premiers (et encore, pas tous mais seulement certains d'entre eux). Les types complexes seront abordés plus tard.

Type simple : type de données décrivant une information atomique.

Type complexe : type de données décrivant une information *composite*, i.e. constituée de plusieurs valeurs, elles-mêmes de types simples ou complexes.

Intuitivement, un type simple permet de manipuler une information correspondant à une seule valeur (par opposition à une collection de plusieurs valeurs), comme par exemple un nombre entier, un nombre réel, une lettre de l'alphabet, etc. Ces types simples sont décrits dans cette section, au fur et à mesure que les différents codages existants sont abordés.

2.2 Nombres entiers

Nous allons d'abord nous intéresser aux façons d'encoder des nombres entiers, qui sont les plus simples, en commençant par les entiers naturels avant d'aborder les entiers relatifs.

2.2.1 Notion de base

On peut décomposer un entier naturel x de manière à l'exprimer dans une base $b > 1$:

$$x = \sum_{i=0}^{n-1} x_i b^i$$

Chaque valeur $x_i \in \{0, 1, \dots, b-1\}$ correspond à l'un des n chiffres de la représentation de x en base b. On précise la base relativement à laquelle est décomposé un nombre en utilisant des parenthèses et un indice.

exemple : $(10100111)_2 = (247)_8 = (167)_{10} = (A7)_{16}$

Dans le monde moderne[15], la base usuelle des humains est $b = 10$, qui correspond à la décomposition *décimale*. En informatique, on s'intéresse surtout à la base 2 (décomposition *binaire*), la base 8 (décomposition octale) et la base 16 (décomposition *hexadécimale*). La base 2 est pertinente puisqu'elle permet d'exprimer une valeur numérique en n'utilisant que les chiffres 0 et 1, comme dans la mémoire d'un ordinateur. Elle a cependant l'inconvénient de ne pas être très pratique en termes d'écriture, car les nombres sont très longs (ils contiennent beaucoup de chiffres). Les bases 8 et 16 permettent d'obtenir une écriture plus compacte, tout en restant compatibles puisqu'il s'agit de multiples de 2.

La base hexadécimale utilise les chiffres de 0 à 9 puis les lettres de A à F :

$b = 10$	$b = 2$	$b = 8$	$b = 16$	$b = 10$	$b = 2$	$b = 8$	$b = 16$
0	0	0	0	10	1010	12	A
1	1	1	1	11	1011	13	B
2	10	2	2	12	1100	14	C
3	11	3	3	13	1101	15	D
4	100	4	4	14	1110	16	E
5	101	5	5	15	1111	17	F
6	110	6	6	16	10000	20	10

[15] Cela n'a pas toujours été le cas, cf. http://fr.wikipedia.org/wiki/Syst%C3%A8me_sexag%C3%A9simal (par exemple)

7	111	7	7	17	10001	21	11
8	1000	10	8	18	10010	22	12
9	1001	11	9	19	10011	23	13

Pour passer de l'écriture en base 10 à l'écriture en base b, l'approche naïve consiste à effectuer des divisions euclidiennes successives par b.

exemple : conversion de $(69)_{10}$ en une valeur binaire (base 2) :

$69/2 = 34$ reste $\mathbf{1}$ ↑
$34/2 = 17$ reste $\mathbf{0}$ ↑
$17/2 = 8$ reste $\mathbf{1}$ ↑
$8/2 = 4$ reste $\mathbf{0}$ ↑
$4/2 = 2$ reste $\mathbf{0}$ ↑
$2/2 = 1$ reste $\mathbf{0}$ ↑
$1/2 = 0$ reste $\mathbf{1}$ ↑

$$(69)_{10} = (\mathbf{1} \times 2^6 + \mathbf{0} \times 2^5 + \mathbf{0} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0)_{10}$$
$$= (\mathbf{1000101})_2$$

Les couleurs et les flèches indiquent l'ordre dans lequel les chiffres obtenus doivent être lus pour obtenir l'écriture en base 2.

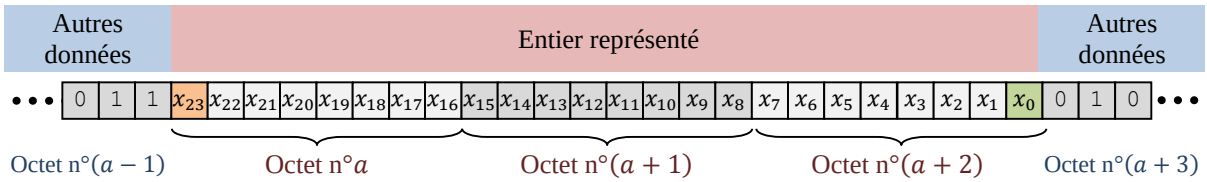
2.2.2 Représentation binaire

La méthode la plus simple pour représenter un entier naturel consiste à l'exprimer en base 2 et à placer en mémoire cette forme ne contenant que les chiffres 0 et 1. Pour cela, on doit d'abord décider du nombre de bits k qui vont être utilisés pour la représentation. Pour mémoire, on a alors la relation suivante :

$$x = \sum_{i=0}^{k-1} x_i 2^i$$

Le *poids* d'un bit correspond à la position i (dans le nombre) du chiffre qu'il contient. Le bit contenant x_0 est appelé bit de *poids faible*, tandis que x_{k-1} est le bit de *poids fort*. Comme expliqué précédemment, sur un ordinateur moderne, la valeur k doit être un multiple de 8, puisqu'on accède à des octets, i.e. des groupes de 8 bits. Par extension, on parle d'*octet de poids faible* et d'*octets de poids fort* pour les octets contenant respectivement les bits de poids faible et de poids fort.

Supposons qu'on a $k = 24$ (i.e. on représente l'entier sur 3 octets), l'organisation en mémoire est alors la suivante :



La zone rouge représente les 3 octets correspondant à l'entier, les zones bleues sont le reste de la mémoire (utilisé pour stocker d'autres informations). La valeur a est l'adresse de l'entier stocké en mémoire, i.e. le numéro de son premier octet. Le bit de poids faible est représenté en vert et celui de poids fort en orange.

Remarque : l'ordre dans lequel les octets sont placés peut varier. Nous avons adopté ici le cas le plus simple.

exemple : représentation de $23 = ((10111)_2$

- Sur 1 octet : `0001 0111`
- Sur 2 octets : `0000 0000 0001 0111`

La valeur k va directement affecter l'intervalle des valeurs que l'on peut représenter en mémoire, qui est $[0; 2^k - 1]$.

exemples :

- Sur 1 octet, le plus grand entier qui peut être codé est $(11111111)_2 = 2^8 - 1 = 255$;
- Sur 2 octets, c'est $(1111\ 1111\ 1111\ 1111)_2 = 2^{16} - 1 = 65535$.

Si jamais, lors d'un calcul, la valeur manipulée sort de l'intervalle prévu, on dit qu'il y a *dépassement de capacité*.

Dépassement de capacité : le nombre de bits utilisé pour coder un nombre n'est pas suffisant en raison de sa valeur qui se situe au-delà de l'intervalle prévu.

En cas de dépassement de capacité, les bits de poids fort du nombre qui ne contiennent pas dans l'espace mémoire alloué sont perdus.

exemple : si on veut représenter 256 sur un octet, on obtient `1 0000 0000`. Ce code demande 9 bits, or on n'en a que 8. Donc, il y un dépassement de capacité et on obtient le code `0000 0000`, correspondant à la valeur (erronée) 0.

2.2.3 Complément à 2

On s'intéresse maintenant aux entiers relatifs, qui par rapport aux entiers naturels, nécessitent en plus de représenter un signe (positif ou négatif). L'approche la plus naturelle consiste à réserver un bit pour cela, par exemple le bit de poids fort : 0 représente un signe $+$ et 1 un signe $-$. Le reste des bits disponible est utilisé pour représenter la valeur absolue du nombre en binaire, comme pour les entiers naturels.

Cependant, cette méthode présente deux inconvénients importants :

- La valeur zéro peut être représentée de deux façon différentes : `00…0` (i.e. $+0$) et `10…0` (i.e. -0), ce qui complique les calculs.
- On a besoin de circuits électroniques différents pour effectuer les additions et les soustractions d'entiers.

exemple : calculons la somme de 1 et -1 codés sur 4 bits avec cette approche

- Code de 1 : `0001`
- Code de -1 : `1001`
- $1 + (-1)$: `0001 + 1001 = 1010 ≠ 0000 ≠ 1000`

Le résultat obtenu ne correspond à aucun des deux codes permettant de représenter zéro, ce qui constitue un gros problème.

La méthode du *complément à deux* permet d'éviter ces inconvénients. Le complément à deux est basé sur le *complément à un* :

Complément à un d'un nombre binaire : transformation du nombre obtenue en remplaçant simultanément tous ses 0 par des 1, et tous ses 1 par des 0.

Complément à deux d'un nombre binaire : transformation du nombre obtenue en calculant le complément à un et en lui additionnant 1.

exemple : calcul du complément à deux de $92 = (1011100)_2$

- On calcule le complément à un : $(100011)_2$
- On rajoute 1 pour obtenir me complément à deux : $(100011 + 1)_2 = (100100)_2$

Dans le cadre informatique, le nombre binaire est représenté en utilisant un nombre fixé de chiffres, noté k. Cette valeur va influencer le calcul du complément à deux, puisque les chiffres 0 éventuellement placés au début du nombre vont devenir autant de chiffres 1.

exemple : si 92 est représenté sur 1 octet, alors on a le nombre binaire `0101 1100` (notez le zéro placé au début du nombre), donc :

- Le complément à un est : `1010 0011`
- Le complément à deux est : `1010 0100`
- La valeur est différente de celle obtenue précédemment (chiffre rouge)

Le complément à deux présente la propriété suivante :

> **P_1 :** le complément à deux sur k bits d'un entier $x \in [0; 2^{k-1} - 1]$ est également la décomposition binaire de l'entier $2^k - x$.

Cette propriété permet de calculer facilement le complémentaire d'un nombre x.

exemple : considérons $k = 8$ et $x = 92 = (0101\ 1100)_2$

- Le complément à un est : `1010 0011`
- Le complément à deux est : `1010 0100`
- Si on interprète ce code comme si c'était simplement une décomposition binaire, on obtient : $(1010\ 0100)_2 = 4 + 32 + 128 = 164$
- Calculons $2^k - x$: $2^8 - 92 = 256 - 92 = 164$ (la propriété est bien respectée)

preuve :

Notons respectivement $C_1(x)$ et $C_2(x)$ les compléments à un et à deux de x sur k bits.

Par définition, on a :

$$x + C_1(x) = 2^k - 1$$

car la somme de ces deux nombres produit un nombre binaire formé de k chiffres 1.

Le complément à deux est défini par : $C_2(x) = C_1(x) + 1$, donc en remplaçant dans l'expression précédente, on obtient :

$$x + C_2(x) - 1 = 2^k - 1$$

Par conséquent, on a bien la propriété P_1 :

$$C_2(x) = 2^k - x$$

Ceci a pour conséquence la propriété suivante :

> **P_2 :** pour tout entier $x \in [0; 2^{k-1} - 1]$, le complément à deux sur k bits de son complément à deux sur k bits est égal à l'entier x lui-même.

Cela signifie que quand on connait seulement le complément à deux de x, il suffit de calculer à nouveau le complément à deux de ce nombre pour retrouver x. Autrement dit, cette transformation est sa propre transformation réciproque[16].

exemple :

- On a calculé le complément à deux de 92 sur 8 bits, qui est `1010 0100`
- Calculons le complément à deux de `1010 0100` :
  - Complément à un : `0101 1011`
  - Complément à deux : `0101 1100`
- Ce nombre `0101 1100` correspond bien à la décomposition binaire de 92

preuve :

D'après P_1, on a $C_2(x) = 2^k - x$, donc on peut poser :

$$C_2\big(C_2(x)\big) = C_2(2^k - x)$$
$$C_2\big(C_2(x)\big) = 2^k - (2^k - x)$$

Par conséquent, on a bien la propriété P_2 :

$$C_2\big(C_2(x)\big) = x$$

[16] On appelle ça une involution.

2.2.4 Entiers relatifs

Pour représenter un entier relatif, on utilise la méthode suivante :

- Un entier *positif* est représenté simplement en base 2, comme un entier naturel.
- Un entier *négatif* est représenté par le complément à deux de sa valeur absolue.

Vous remarquerez que la valeur zéro est représentée de la même façon, que l'on considère qu'il s'agit d'un nombre négatif ou positif :

exemple : complément à deux de zéro sur 1 octet (donc de `0000 0000`) :

- Complément à un : `1111 1111`
- Complément à deux : `(1) 0000 0000`
- Le 1 entre parenthèses n'est pas placé en mémoire, puisqu'il ne peut pas être contenu dans les 8 bits qui constituent l'octet (on a un *dépassement de capacité*).

Comme on l'a vu précédemment, puisqu'on utilise k bits pour coder le nombre, on dispose de 2^k combinaisons possibles. Pour les entiers naturels, cela nous a permis de représenter des valeurs allant de 0 à $2^k - 1$. Ici, puisqu'on a le même nombre de combinaisons mais qu'on doit représenter des nombres négatifs et positifs, il faut diviser cet intervalle en deux parties (une pour les nombres positifs et l'autre pour les négatifs) de tailles à peu près égales. Les bornes utilisées sont les suivantes :

- Borne inférieure : -2^{k-1}, qui correspond au code `1...1`
- Borne supérieure : $+2^{k-1} - 1$, qui correspond au code `01...1`

On peut vérifier que si l'on compte le nombre total de combinaisons (valeur zéro, négatifs et positifs) on a bien $1 + 2^{k-1} + 2^{k-1} - 1 = 2^k$. À noter qu'ici, le dépassement de capacité peut maintenant apparaitre non seulement parce qu'une valeur est trop grande (i.e. supérieure à $2^{k-1} - 1$), comme pour les entiers naturels, mais aussi parce qu'elle est trop petite (i.e. inférieure à -2^{k-1}).

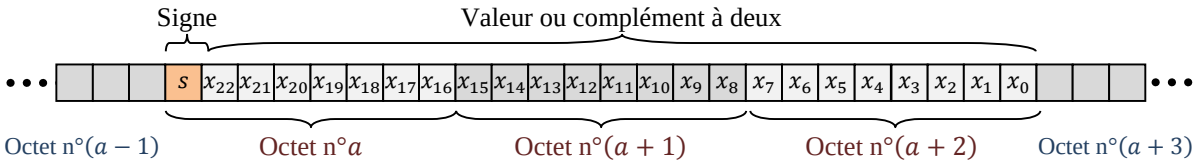
L'intérêt de ces bornes est que les codes commençant par 1 représentent toujours des nombres négatifs, et ceux commençant par 0 correspondent toujours à des nombres positifs (ou à zéro). Le code peut donc être interprété sans ambiguïté.

exemples : codage de quelques entiers relatifs sur 1 octet

- $2^7 - 1$: `0111 1111`
- 92 : `0101 1100`
- 0 : `0000 0000`

- -1 : `1111 1111`
- -92 : `1010 0100`
- -2^7 : `1000 0000`

Reprenons la figure précédente représentant l'occupation mémoire, dans laquelle un entier naturel était codé sur 3 octets. Pour un entier relatif, on aura quelque chose de la forme suivante :



On peut vérifier que le problème rencontré précédemment pour $1 + (-1)$ est résolu par cette représentation duale :

- Code de 1 : `0001`
- Code de -1 : `1111`
- Code de $1 + (-1)$: `0001 + 1111 = (1) 0000 = 0000` (il y a un dépassement de capacité)

Ceci est peut être considéré comme un corollaire de la propriété $P_1 : x + C_2(x) = x + 2^k - x = 2^k$. Or 2^k provoque un dépassement de capacité qui ne laisse que des chiffres 0 dans les bits concernés, ce qui est interprété comme la valeur zéro.

La propriété P_1 amène aussi une réflexion intéressante sur l'interprétation des données stockées en mémoire. Supposons qu'on a un octet contenant le code `1010 0100`, mais qu'on ne connait pas le type de ce code. Autrement dit, on ne sait pas s'il s'agit d'un entier naturel ou d'un entier relatif. Alors, on ne peut pas interpréter correctement le code, car les deux interprétations divergent. En effet, le code représente :

- La valeur -92 quand il est interprété par la méthode du complément à deux ;
- La valeur 164 quand il est interprété comme une simple décomposition binaire.

Ceci illustre l'importance du type dans la manipulation des données, et ce point sera abordé de nouveau plus tard dans le cours et en TP.

On peut aussi présenter la propriété suivante liée au complément à deux :

> **P_3 :** tout entier $x \in [-2^{k-1}; 2^{k-1} - 1]$ peut être exprimé sous la forme $x = -2^{k-1}s + \sum_{i=0}^{k-2} x_i 2^i$ et son code sur k bits correspond alors à la séquence de bits $(s, x_{k-2}, \ldots, x_0)$.

L'intérêt de cette propriété est qu'elle permet de calculer rapidement le code d'une valeur négative.

vérification :

- Le code `0101 1100` représente l'entier positif 92 ;
- Le code `1101 1100` (la seule différence étant le bit de poids fort) représente un entier négatif :
  - Complément à un : `0010 0011`
  - Complément à deux : `0010 0100`
  - Valeur obtenue : $-(4 + 32) = -36$
- On a bien, d'après la propriété P_3 : $-2^7 + 92 = -128 + 92 = -36$

exemple : on veut calculer le code représentant la valeur -36

- $-36 = -2^7 + 92$
- Donc il suffit de calculer la représentation binaire de 92 (i.e. `0101 1100`) et de mettre un 1 dans le bit de poids fort (on obtient bien `1101 1100`).

preuve :

Si $s = 0$, alors $-2^{k-1}s = 0$, et on a donc $x = \sum_{i=0}^{k-2} x_i 2^i$.

Comme les valeurs positives sont représentées directement par leur décomposition binaire, on obtient bien la séquence $(s, x_{k-2}, \ldots, x_0) = (0, x_{k-2}, \ldots, x_0)$ décrite par P_3.

Sinon, i.e. on a $x_{k-1} = 1$, alors $x = -2^{k-1} + \sum_{i=0}^{k-2} x_i 2^i$.

Cette valeur est négative car $2^{k-1} > \sum_{i=0}^{k-2} x_i 2^i$, donc son code est par définition le complément à deux de $|x| = -x$ sur k bits.

D'après la propriété P_1, on a alors :

$$C_2(-x) = 2^k - (-x) = 2^k + x$$

En remplaçant x dans l'expression précédente, on obtient :

$$C_2(-x) = 2^k - 2^{k-1} + \sum_{i=0}^{k-2} x_i 2^i$$

$$C_2(-x) = 2^{k-1} + \sum_{i=0}^{k-2} x_i 2^i$$

On obtient bien la séquence décrite par P_3 : $(s, x_{k-2}, \ldots, x_0) = (1, x_{k-2}, \ldots, x_0)$

2.2.5 Types entiers du C

Le langage C comprend principalement 4 types entiers, différenciés par la quantité d'espace mémoire utilisée pour représenter la valeur numérique considérée :

- `char` : 1 octet
- `short` : 2 octets
- `int` : en général 2 octets, parfois 4 ou plus (ça dépend du compilateur)
- `long` : 4 octets

De plus, 2 *modificateurs* peuvent être appliqués à ces types :

- `unsigned` : entiers naturels
- `signed` : entiers relatifs

Si on ne précise *pas* de modificateur, le type est `signed` par défaut. Un type `unsigned` est représenté en décomposition binaire, alors qu'un type `signed` utilise le complément à deux pour les valeurs négatives. Ceci affecte donc l'intervalle des valeurs que l'on peut représenter avec un type donné.

type	octets	nombre de valeurs possibles	modificateur	intervalle de définition
`char`	1	$2^8 = 256$	`unsigned`	$[0, 2^8 - 1]$
			`signed`	$[-2^7, 2^7 - 1]$
`short`	2	$2^{16} = 2^{2\times8} = 65\,536$	`unsigned`	$[0, 2^{16} - 1]$
			`signed`	$[-2^{15}, 2^{15} - 1]$
`long`	4	$2^{4\times8} = 4\,294\,967\,296$	`unsigned`	$[0, 2^{32} - 1]$
			`signed`	$[-2^{31}, 2^{31} - 1]$

Le type `char` est en général utilisé pour stocker des valeurs numériques qui représentent des caractères alphanumériques (i.e. des lettres, chiffres, symboles de ponctuation, etc.). Il est approfondi en section 2.4.

Il n'existe *pas* de type *booléen* à proprement parler en C89. A la place, on utilise des entiers :

- La valeur *zéro* représente la valeur *faux* ;
- Une valeur *non-nulle* représente la valeur *vrai*.

Remarque : un type booléen explicite a ensuite été introduit dans C99, mais nous ne l'utiliserons pas.

exercices :

1) Donnez la représentation de la valeur 58 pour un entier de type `unsigned char` :
   - $(58)_{10} = (2^5 + 2^4 + 2^3 + 2^1)_{10} = (111010)_2$
   - → `00111010`

2) Donnez la représentation de la valeur -58 pour un entier de type `signed char` :
   - $(58)_{10} = (111010)_2$
   - complément à *2* : $(000101)_2$
   - → `00000110`

2.3 Nombres réels

La représentation utilisée pour les nombres entiers pourrait s'appliquer assez directement aux nombres réels, mais nous allons voir que cela présente certains inconvénients. Pour cette raison, ceux-ci sont représentés selon un codage assez différent appelé *virgule flottante*.

2.3.1 Représentation binaire

La décomposition binaire des entiers peut se généraliser à l'ensemble des réels, grâce au théorème suivant :

> **T_1 :** tout nombre réel $x \in [0;1[$ admet un développement en base 2 (binaire) de la forme :
>
> $$x = \sum_{i=0}^{+\infty} \frac{x_i}{2^i} \text{ avec } x_i \in \{0,1\}$$
>
> Le développement en base 2 de x est dit *fini* s'il existe un rang i_0 à partir duquel on a : $\forall i > i_0, x_i = 0$.

L'algorithme permettant de calculer le développement en base 2 d'un réel $x \in [0,1[$ à partir de sa forme décimale est basé sur des multiplications successives par $b = 2$.

exemple : convertissons $x = 0{,}59375$:

$0{,}59375 \times 2 = \mathbf{1}{,}1875$ ↓
$0{,}1875 \times 2 = \mathbf{0}{,}375$ ↓
$0{,}375 \times 2 = \mathbf{0}{,}75$ ↓
$0{,}75 \times 2 = \mathbf{1}{,}5$ ↓
$0{,}5 \times 2 = \mathbf{1}{,}0$ ↓

$$(0{,}59375)_{10} = (\mathbf{1} \times 2^{-1} + \mathbf{0} \times 2^{-2} + \mathbf{0} \times 2^{-3} + \mathbf{1} \times 2^{-4} + \mathbf{1} \times 2^{-5})_{10}$$
$$= (0, \mathbf{1}\mathbf{0}\mathbf{0}\mathbf{1}\mathbf{1})_2$$

Les couleurs et les flèches indiquent l'ordre dans lequel les chiffres obtenus doivent être lus pour obtenir l'écriture en base 2. Le nombre $(0{,}59375)_{10}$ admet un développement fini en base 2 : $(0{,}10011)_2$.

Certains nombres, même simples, comme 0,3 n'admettent pas de développement binaire fini, et l'algorithme précédent devient alors cyclique.

exemple : convertissons $x = 0{,}3$ en binaire :

$0{,}3 \times 2 = \mathbf{0}{,}6$ ↓
$0{,}6 \times 2 = \mathbf{1}{,}2$ ↓
$0{,}2 \times 2 = \mathbf{0}{,}4$ ↓
$0{,}4 \times 2 = \mathbf{0}{,}8$ ↓
$0{,}8 \times 2 = \mathbf{1}{,}6$ ↓
$0{,}6 \times 2 = \mathbf{1}{,}2$ ↓

… on recommence à partir de la 3ème ligne et on boucle à l'infini.

Donc 0,3 admet un développement infini en base 2 de la forme 0,0 1001 1001 1001…

On a la propriété suivante :

> **P_4 :** Un nombre réel $x \in [0;1[$ possède une écriture binaire finie si et seulement s'il existe un entier $m > 0$ tel que $2^m x \in \mathbb{N}$ est un entier.

Si m, alors l'algorithme proposé ci-dessus permet de déterminer les m chiffres situés après la virgule, puisque les multiplications successives par 2 reviennent à multiplier par 2^m.

Pour 0,3, quel que soit $m \in \mathbb{N}$, $3 \times 2^m/10$ n'est pas entier. Donc, ce nombre ne possède pas de développement fini en base 2.

Pour écrire le développement d'un nombre réel x quelconque (pouvant être supérieur à 1), on le décompose sous la forme :

$$x = \mathrm{E}[x] + (x - \mathrm{E}[x])$$

où $E[x] \in \mathbb{N}$ est la partie entière de x, et $x - \mathrm{E}[x] \in [0; 1[$. Le premier terme est obtenu en utilisant la méthode vue pour les entiers. Le second terme avec la méthode vue pour les nombres réels dans $[0; 1[$. Le nombre complet est obtenu en sommant ces deux termes.

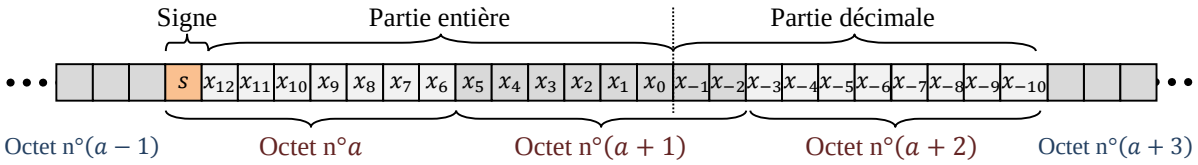
exemples :

- On veut convertir 69,59375 en binaire :
  - On décompose : $69{,}59375 = 69 + 0{,}59375$
  - On traite 69 : $(1000101)_2$ (déjà fait en section 2.2.1)
  - On traite 0,59375 : $(0{,}10011)_2$ (déjà fait en section 2.3.1)
  - On a donc au total : $(1000101{,}10011)_2$
- On veut convertir 17,1 en binaire :
  - On décompose : $17{,}1 = 17 + 0{,}1$
  - On traite 17 : $17 = (10001)_2$
  - On traite 0,1 : $0{,}1 = (0{,}0001\ 1001\ 1001\ \ldots)_2$
  - Donc, on a $17{,}1 = (10001{,}0001\ 1001\ 1001\ \ldots)_2$

2.3.2 Virgules fixe et flottante

Comme pour les entiers, une fois qu'on a déterminé la décomposition binaire d'un nombre, la question se pose de savoir comment le *coder* sur un nombre limité de bits. La méthode la plus simple consiste à utiliser une *virgule fixe* : un nombre p de bits sert à coder la partie entière et un nombre q de bits de bits est dédié à la partie décimale. Pour distinguer entre positifs et négatifs, on peut utiliser la méthode du complément à deux, comme pour les entiers relatifs.

exemple : supposons qu'on utilise 3 octets, dont $q = 10$ bits pour la partie décimale (et donc $p = 13$ pour la partie entière). La figure précédente de l'occupation mémoire devient alors la suivante :



Avec ce codage, on peut représenter des nombres définis dans $[-2^p; 2^p - 2^{-q}]$. La borne inférieure est attente quand seul le bit s est 1 ; la borne supérieure quand seul le bit s est 0. La décomposition est maintenant la suivante :

$$x = -2^p s + \sum_{i=-q}^{p-1} b_i 2^i$$

Avec ce codage en virgule fixe, le *pas de quantification* est constant et égal à 2^{-q}.

Pas de quantification : écart entre deux valeurs successives représentables avec le codage considéré.

Cette méthode de représentation en virgule fixe est utilisée dans certains systèmes car elle permet une implémentation simple des opérations arithmétiques. Mais elle n'est pas efficace pour coder des grandes valeurs ou des valeurs proches de zéro.

exemple : supposons qu'on utilise 1 octet pour représenter le nombre, dont $p = 4$ bits pour la partie entière :

- Pas de quantification : 2^{-3}
- Plus grande valeur représentable : $2^4 - 2^{-3} = 16 - 0{,}125 = 15{,}875$
- Plus petite valeur représentable : $-2^4 = -16$

- Plus petite valeur positive codée : $2^{-3} = 0{,}125$ (i.e. le pas de quantification)

Les valeurs représentables sont alors (dans l'ordre décroissant) :

$$15{,}875; 15{,}750; 15{,}625; 15{,}500; \ldots; 0{,}250; 0{,}125; 0; \ldots; -15{,}875; -16$$

Pour la *même quantité* d'espace mémoire, on aimerait pouvoir coder des valeurs *plus grandes* en *augmentant* le pas de quantification pour les grandes valeurs ; et coder des valeurs *plus petites* en *diminuant* le pas de quantification pour les petites valeurs. Cette possibilité est offerte par le codage des réels en *virgule flottante*. Ce principe permet de faire varier les valeurs p et q, de façon à les adapter à la grandeur du nombre représenter, et ainsi de faire varier la précision du codage.

Un nombre x est alors décomposé de la façon suivante :

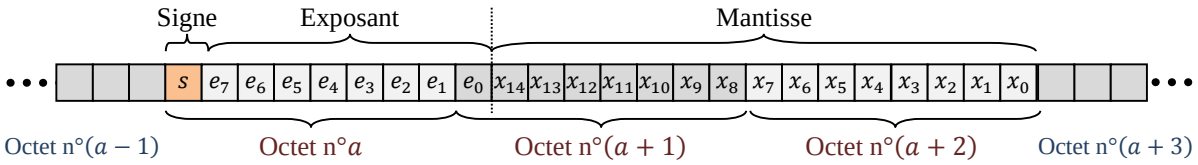
$$x = (-1)^s m b^e$$

où $b = 2$ dans notre cas.

exemple : $x = (100{,}0101)_2 = (1000101)_2 \times 2^{-4} = (1{,}000101)_2 \times 2^2$

Comme précédemment, s correspond au signe du nombre. La variable m est appelée la *mantisse*, tandis que e est l'*exposant*. C'est ce dernier paramètre qui détermine la position de la virgule dans la représentation de x (d'où le nom de *virgule flottante*). Les nombres de bits p et q utilisés pour représenter respectivement m et e déterminent le codage.

Par exemple, si on représente un réel sur 3 octets en utilisant 8 bits pour l'exposant, la représentation graphique de la mémoire prend alors la forme suivante :



2.3.3 Types réels du C

En langage C (du moins dans la version C89), on trouve trois types de réels, qui diffèrent sur :

- Le nombre d'octets k utilisés pour représenter le nombre.
- Les nombres de bits p et q consacrés à la mantisse m et à l'exposant e.

Le tableau ci-dessous décrit ces trois types :

type	octets	p	q	intervalle de définition	chiffres significatifs
`float`	4	23	8	$[1{,}5 \times 10^{-45}\,;3{,}4 \times 10^{38}]$	7-8
`double`	8	52	11	$[5 \times 10^{-324}\,;1{,}7 \times 10^{308}]$	15-16
`long double`	10	64	15	$[3{,}4 \times 10^{-4932}\,;1{,}1 \times 10^{4932}]$	19-20

Cependant, pour ces trois types, la forme de la mantisse est fixée à : 1,...

exemple : reprenons la valeur valeur 69,59375, dont on a vu précédemment que la décomposition binaire était $(1000101{,}10011)_2$. Avec ce codage, on devra la représenter sous la forme $(1{,}00010110011)_2 \times 2^6$. On a donc la décomposition suivante :

- La mantisse est $m = \mathbf{1{,}00010110011}$
- L'exposant est $e = (6)_{10} = (\mathbf{110})_2$
- Le signe est $s = \mathbf{0}$, puisqu'on a un nombre positif.

Si on représente cette valeur 69,59375 sur autant de bits qu'un `float`, c'est-à-dire sur 4 octets dont 8 bits pour l'exposant et 23 pour la mantisse, on obtient le code suivant :

```
0 00000110 10001011001100000000000
```

Mais puisque, dans ce codage, *tous* les nombres ont une partie entière égale à 1, alors il est inutile de représenter ce 1. On parle de *bit caché*, et cette astuce permet d'économiser un bit dans la représentation, et donc d'augmenter sa précision d'autant. Bien sûr, ce 1 est néanmoins considéré lors des calculs.

exemple : pour 69,59375, après cette correction le code devient :

```
0 00000110 00010110011000000000000
```

De plus, il faut aussi pouvoir représenter des exposants négatifs dans notre codage. Pour cela, on pourrait utiliser le complément à deux, comme on l'avait fait pour représenter les entiers relatifs (section 2.2.3). Cependant, on est soumis ici à des contraintes différentes des précédentes. Pour un exposant e exprimé sur q bits, la solution qui a été retenue est de représenter non pas directement e, mais $e + 2^{q-1} - 1$. On peut ainsi représenter des exposants négatifs, pourvu que leur valeur absolue soit inférieure à $2^{q-1} - 1$.

exemple : pour coder la valeur 69,59375, on voulait représenter l'exposant $e = (6)_{10} = (110)_2$ sur $q = 8$ bits. Si on applique la formule précédente, on a donc : $(6 + 2^7 - 1)_{10} = = (110 + 1111111)_2 = (10000101)_2$. Le code devient alors :

```
0 10000100 00010110011000000000000
```

Remarque :

Le pas de quantification, c'est-à-dire l'écart entre 2 valeurs successives représentables n'est pas constant contrairement à la représentation en virgule fixe, il est proportionnel à l'exposant.

Exemple :

- Entre l'entier 2101987328 codé par `0 10011101 11110101001001110010000` et la valeur représentable suivante codée par `0 10011101 11110101001001110010001` (on a ajouté 1 à la mantisse), l'écart est égal à $2^e \times 2^{-23} = 2^7$=128.
- Entre 1 codé par `0 01111111 00000000000000000000000` et le nombre suivant représenté par `0 01111111 00000000000000000000001` l'écart est $2^e \times 2^{-23} = 2^{-23} \approx 1{,}2 \times 10^{-7}$.

exercices :

1) Donnez la représentation de la valeur $-3{,}5$ pour un réel de type `float` :
   - $(3{,}5)_{10} = (11{,}1)_2$
   - on donne la forme 1, ... à la mantisse : $11{,}1 = 1{,}11 \times 2^1$
     - on représentera `110...0`.
   - donc $p = 1$. On a $(1 + 2^{8-1} - 1)_{10} = (2^7)_{10} = (10000000)_2$
     - on représentera `10000000`.
   - le bit de signe a la valeur 1
     - on représentera `1`.
   - au final, on a :

     ```
     1 10000000 11000000000000000000000
     ```

2) Donnez la représentation de la valeur -58 pour un réel de type `float` :
   - $(58)_{10} = (111010)_2 = (1{,}11010)_2 \times (2^5)_{10}$
     - représentation de la mantisse : `11010...0`
   - $(101 + 1111111)_2 = (10000100)_2$
     - représentation de l'exposant : `10000100`
   - bit de signe : `1`

     ```
     1 10000100 11010000000000000000000
     ```

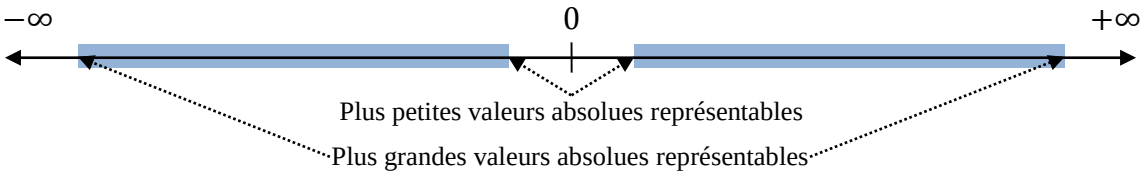
2.3.4 Erreurs d'amplitude et de précision

Sur k bits, on peut coder au maximum 2^k réels différents de manière exacte. La probabilité d'un réel d'être codé de manière exacte est donc *quasi-nulle* (puisque sur l'intervalle de codage, on va trouver une infinité de réels). On distingue deux types d'erreurs de représentation en virgule flottante : celles liées à la taille des nombres (erreur d'amplitude) et celles liées à l'écriture de la mantisse (erreur de précision).

La limitation concernant la taille des nombres représentables est proportionnelle au nombre de bits q réservé au codage de l'exposant. Si l'exposant d'un nombre est *trop grand*, ce nombre n'est pas représentable, et comme pour les entiers, on dit qu'il y a un *dépassement de capacité*. Mais au contraire des entiers, il est aussi possible ici que l'exposant soit *trop petit* pour que le nombre soit représentable. On parle alors de sous-passement de capacité :

Sous-passement de capacité : le nombre de bits utilisé pour coder un nombre n'est pas suffisant en raison de sa valeur qui se situe en-deçà de l'intervalle prévu.

La figure suivante illustre la différence entre dépassement et sous-passement de capacité :



La limite concernant la précision de l'écriture du nombre est liée au nombre de bits utilisés pour coder la mantisse. En effet un nombre sera représenté en machine de manière *approchée* si :

- Soit son développement en base 2 est fini, mais le nombre de bits significatifs de sa mantisse est supérieur au nombre de bits alloués au codage de la mantisse.
- Soit il n'admet pas de développement fini en base 2.

exemple 1 : considérons le codage de $x = 2101987361$ avec le type `float` :

- La décomposition binaire du nombre est la suivante : $2101987361 = (1{,}11\ 1101\ 0100\ 1001\ 1100\ 1000\ 0010\ 0001)_2 \times 2^{30}$
- On retient les 23 premiers bits écrits après la virgule pour former la mantisse, qui prendra donc la forme : `11110101001001110010000`
- On a l'exposant $e = (30)_{10}$, donc $e + 2^{8-1} - 1 = (175)_{10} = (1001\ 1101)_2$, et il sera codé `10011101`
- Le nombre est positif, donc le bit de signe a la valeur 0, qui sera codée `0`
- Au final, on a le code suivant :

```
0 10011101 11110101001001110010000
```

La valeur affichée par l'ordinateur est `2101987328.000000`. La différence entre x et le nombre codé en machine, appelé *nombre machine*, correspond à la troncature effectuée lors du codage. En effet, on a ici :

$$2101987361 - 2101987328 = 33$$
$$= (0{,}0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000\ 01)_2 \times 2^{30}$$
$$= (10\ 0001)_2$$

exemple 2 : considérons le codage de $x = 0{,}1$ toujours en utilisant le type `float` :

- La décomposition binaire est : $0{,}1 = (1{,}1001\ 1001\ \ldots)_2 \times 2^{-4}$

- On retient les 23 premiers chiffres après la virgule pour obtenir la mantisse, ce qui donne le code : `10011001100110011001100`
- Pour l'exposant, on a $e = (-4)_{10}$, donc on utiliser comme code la valeur : $e + 2^{8-1} - 1 = (123)_{10} = (0111\ 1011)_2$, i.e. `01111011`
- Le bit de signe est représenté par le code `0`
- On obtient finalement le code suivant :

`0 01111011 10011001100110011001100`

L'affichage de cette valeur par l'ordinateur donnera bien entendu une valeur approchée, en l'occurrence : `0.1000000015`.

Plutôt que de faire une *troncature*, il est aussi possible d'effectuer un *arrondi* lorsque le nombre de chiffres significatifs est supérieur à celui autorisé par le codage.

exemple : pour $x = 0{,}1$, les deux chiffres suivant la troncature sont des 1, donc on peut remplacer le dernier chiffre de la mantisse (qui est un 0) par un 1. On obtient alors une meilleure approximation, qui prend la forme du code suivant :

`0 01111011 10011001100110011001101`

2.3.5 Erreurs relatives

Considérons un nombre $x = (-1)^s m 2^e$ et le code utilisé pour sa représentation en mémoire $x' = (-1)^s m' 2^e$. L'*erreur de représentation* est alors : $\Delta x = |x - x'| = (-1)^s \Delta m 2^e$, avec $\Delta m = |m - m'|$.

L'erreur Δx, ne nous dit rien sur la précision de la représentation si nous ne connaissons pas la valeur de x. Par exemple une erreur $\Delta x < 0{,}1$ peut paraître faible si $x = 10^6$ mais importante si $x = 2$. Il est donc plus significatif de connaître une majoration de l'*erreur relative* $\Delta x / x$ de la représentation.

On a la propriété suivante :

P_5 : Si m' est normalisée sous la forme $m' = (1, b_1 b_2\ b_3 \ldots\ b_n)_2$, et codé sur n bits, alors la majoration de l'erreur relative ne dépend que de n :

$$\frac{\Delta x}{x} = \frac{\Delta m}{m} < 2^{-n}$$

exemple :

- Pour le type `float`, l'erreur relative est majorée par 2^{-23}
- Pour le type `double`, l'erreur relative est donc majorée par 2^{-52}

preuve :

La mantisse m s'écrit sous la forme $m = (1, b_1 b_2\ b_3 \ldots\ b_n b_{n+1}\ \ldots)_2$, ce développement pouvant être fini ou infini. Alors, en supposant que m' est obtenu par troncature à partir de m :

$$\frac{\Delta m}{m} = \frac{m - m'}{m} = \frac{(0{,}0 \ldots . 0 b_{n+1} \ldots .)_2}{(1, b_1 b_2 \ldots . b_{n+1} \ldots)_2} \leq \frac{2^{-n}}{1} = 2^{-n}$$

Si m' est obtenue par arrondi à partir de m, l'erreur Δm obtenue est inférieure ou égale à l'erreur obtenue par troncature, donc l'inégalité obtenue reste valable.

2.4 Caractères

Par définition, les données stockées dans la mémoire d'un ordinateur sont numériques. Cependant, ces codes ne représentent pas forcément des *informations numériques*. En particulier, dans le reste de cette section, nous allons nous intéresser aux caractères (lettres, chiffres, symboles de ponctuation, etc.) et aux images. Ces informations-là ne sont pas numériques *par nature*, mais bien sûr elles sont représentées de façon numérique dans la mémoire.

2.4.1 Codage ASCII

Chaque caractère manipulé par un programme informatique est représenté par une valeur numérique spécifique. La correspondance entre caractère et valeur numérique se fait à l'aide d'une *table d'encodage* qui associe à chaque caractère une valeur entière. Cette table dépend du système d'exploitation utilisé, notamment de sa langue (français, anglais, turc, etc.).

En langage C, on utilise la norme ASCII[17] à cet effet. Dans sa version de base, elle associe un code numérique à $128 = 2^7$ caractères différents. Il s'agit principalement de lettres majuscules et minuscules, de chiffres, de signes de ponctuation, et de caractères *non-imprimables*.

En effet, on distingue deux types de caractères :

Caractère imprimable : code représentant un caractère apparaissant à l'écran ou lors d'une impression papier.

exemples : lettres, chiffres, signes de ponctuations, mais aussi le caractère *espace* (ou caractère d'*espacement*) qui permet de séparer les mots.

Caractère non-imprimable (ou caractère de contrôle) : code ne représentant par un caractère, mais une *commande* utilisée notamment pour la *mise en forme*.

Ces commandes sont destinées au périphérique qui traite le texte, par exemple qui l'affiche ou qui l'imprime : sauter une ligne, signaler la fin du texte, etc.

exemples : tabulation, retour à la ligne, etc.

La table ci-après contient tous les caractères *imprimables* de la norme ASCII de base :

32		46	.	60	<	74	J	88	X	102	f	115	t
33	!	47	/	61	=	75	K	89	Y	103	g	116	u
34	"	48	0	62	>	76	L	90	Z	104	h	117	v
35	#	49	1	63	?	77	M	91	[	105	i	118	w
36	$	50	2	64	@	78	N	92	\	106	j	119	x
37	%	51	3	65	A	79	O	93	]	107	k	120	y
38	&	52	4	66	B	80	P	94	^	108	l	121	z
39	'	53	5	67	C	81	Q	95	_	109	m		
40	(	54	6	68	D	82	R	96	`	111	n		
41	)	55	7	69	E	83	S	97	a	110	o		
42	*	56	8	70	F	84	T	98	b	111	p		
43	+	57	9	71	G	85	U	99	c	112	q		
44	,	58	:	72	H	86	V	100	d	113	r		
45	-	59	;	73	I	87	W	101	e	114	s		

Remarque : notez que la table ASCII range les chiffres dans l'ordre croissant et les caractères majuscules ou minuscules dans l'ordre alphabétique. Cette propriété sera utile lorsqu'on écrira des programmes manipulant les caractères, lors des TP.

La norme ayant été initialement définie pour la langue anglaise, elle ne contient pas de lettre comportant des signes diacritiques. Elle ne permet donc pas de représenter correctement des textes en français ou en turc.

Pour permettre de traiter les langues de ce type, des extensions de la norme ASCII ont été développées par la suite, permettant de coder $2^8 = 256$ caractères. Par exemple, les lettres accentuées (é, è, ê, ü, ö, ù) ou les caractères avec cédille (comme le ç) peuvent être codés dans cette table. Chaque extension cible une langue (ou un groupe de langues) en particulier, et ne permettent pas de coder les caractères de toutes les langues en usage dans le monde. Un code

[17] *American Standard Code for Information Interchange*, norme datant de 1963.

de la norme ASCII étendue peut être codé sur un seul octet. En C, ces codes sont représentés grâce au type `char`, qui permet de stocker un entier de 1 octet, comme on l'a vu précédemment.

2.4.2 Autre codages

Il existe d'autres solutions d'encodage des caractères plus récentes. Elles sont basées principalement sur le standard Unicode, normalisé par l'ISO. Unicode recense les caractères des très nombreuses langues en usage dans le monde, il les catégorise par langue ou par région et leur associe un nom et une valeur (appelée *point de code*). Il ne définit pas la graphie de ces caractères.

exemple : dans la table *Latin Extended-A* d'Unicode, le caractère ğ est dénommé *latin small letter G with breve* et son point de code (en hexadécimal) est `011F`, noté *U+011F*. Il est aussi précisé que ce caractère est associé aux langues turque et azérie (voir http://www.unicode.org/charts/PDF/U0100.pdf).

http://www.unicode.org/charts/PDF/U0100.pdf

À partir du standard Unicode, sont définies des méthodes d'encodage des caractères comme l'UTF-8 (*Unicode Transformation Format*) qui permet d'encoder des caractères définis dans Unicode. L'UTF-8 représente les caractères sur un nombre d'octets variant entre 1 et 4. Les caractères codés sur 1 seul octet, de valeur comprise entre 1 et 127, sont identiques à ceux de la norme ASCII, ce qui rend l'UTF-8 compatible avec ASCII.

exemple : le point de code de ğ est *U+011F*, soit en binaire `U+0000 0001 0001 1111`. Le caractère ğ sera alors codé sur 2 octets sous la forme `110* **** 10** ****`, les étoiles étant remplacé par les $5 + 6 = 11$ bits de poids faible du point de code. Ce qui donne un encodage sur 2 octets :

1100 0100 **10**01 1111 ou C49F

Dans le cadre des TP, on se limitera à l'utilisation du code ASCII. Mais notez que l'Unicode est répandu dans les langages plus récents tels que Java. Pour plus de détails sur l'UTF-8, cf. http://www.rfc-editor.org/rfc/rfc3629.txt.

2.5 Images

Lors de TP, nous allons utiliser une approche graphique pour illustrer certains concepts de l'algorithmique et du C. Pour cette raison, nous donnons ici une description succincte de la façon dont les images peuvent être représentées dans la mémoire d'un ordinateur.

2.5.1 Images monochromes

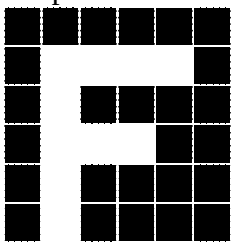
Une image numérique est découpée de manière régulière en petits carrés appelés *pixels :*

Pixel : *picture element*, i.e. plus petit élément constitutif d'une image.

Pour une image en noir & blanc, un pixel peut être soit noir, soit blanc. On peut donc le coder au moyen d'un simple bit. Par exemple : 0 pour un pixel éteint, et 1 pour un pixel allumé. Une image complète sera alors représentée par une matrice de bits, chacun de ces éléments correspondant à un pixel.

exemple : Dessin d'un caractère sur 36 pixels (matrice 6 × 6) :

Lettre F représentée sur 36 pixels :



Matrice codant l'image :

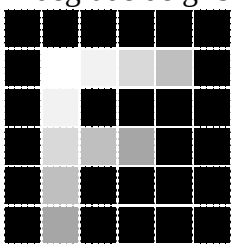
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$ Niveaux de gris

L'œil humain perçoit en moyenne 200 niveaux d'intensité entre le noir et le blanc. On peut donc étendre le codage noir & blanc de manière à représenter des valeurs intermédiaires : on parle alors de codage en *niveaux de gris*.

Puisqu'on n'a besoin que de 200 valeurs intermédiaires, il est possible de coder l'intensité de chaque pixel sur un seul octet : de 0 (pixel noir) à 255 (pixel allumé).

exemple :

dégradé de gris :

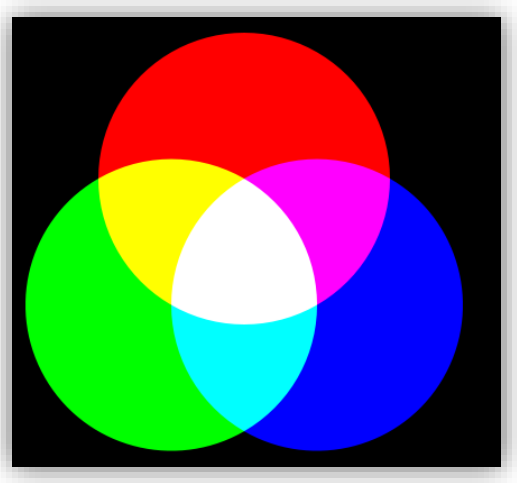


Matrice codant l'image :

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 255 & 240 & 200 & 160 & 0 \\ 0 & 240 & 0 & 0 & 0 & 0 \\ 0 & 200 & 150 & 120 & 0 & 0 \\ 0 & 160 & 0 & 0 & 0 & 0 \\ 0 & 120 & 0 & 0 & 0 & 0 \end{pmatrix}$$

2.5.2 Images en couleur

On peut faire évoluer ce codage pour représenter des images en couleur. La combinaison de trois sources émettrices de couleurs rouge, vert et bleu permet de restituer les différentes teintes. C'est ce que l'on appelle la *synthèse additive des couleurs*.



Une teinte t est ainsi obtenue par combinaison linéaire des couleurs primaires :

$$t = \alpha R + \beta V + \gamma B.$$

On peut donc reconstituer les couleurs sur un écran vidéo en combinant 3 faisceaux de couleurs rouge, vert et bleu sur chaque pixel de l'écran. On code la teinte t d'un pixel en codant les intensités α, β et γ de chaque faisceau. Le nombre de couleurs dépendra donc de la taille allouée au codage des intensités α, β et γ. Une image de *profondeur* 24 bits est une image où chacune des intensités est codée sur 1 octet ce qui permet de coder 256^3 couleurs différentes (soit environ 16 millions de couleurs ce qui est largement suffisant pour la perception humaine).

Dans le cadre des TP, nous allons utiliser une bibliothèque appelée SDL (*Simple DirectMedia Layer*) pour traiter les graphismes[18]. Les couleurs y sont codées en mode RVB (*Red Green Blue* ou *RGB*) sur une profondeur de 24 bits, chaque intensité est codée en hexadécimal, ce qui est indiqué par le préfixe `0x` devant chaque valeur. L'extrait de code source ci-dessous définit des macros utilisées pour représenter des couleurs :

```
#define C_NOIR              SDL_MapRGB(affichage->format,  0x00, 0x00,  0x00)
#define C_BLANC             SDL_MapRGB(affichage->format,  0xFF, 0xFF,  0xFF)
#define C_ROUGE             SDL_MapRGB(affichage->format,  0xFF, 0x00,  0x00)
#define C_BLEU              SDL_MapRGB(affichage->format,  0x00, 0x00,  0xFF)
#define C_VERT              SDL_MapRGB(affichage->format,  0x00, 0xFF,  0x00)
#define C_JAUNE             SDL_MapRGB(affichage->format,  0xFF, 0xFF,  0x00)
```

- Pour la macro `C_NOIR`, on a $(\alpha, \beta, \gamma) = (0,0,0)$ puisque le noir résulte de l'absence de couleurs émises.
- La macro `C_BLANC` correspond au triplet $(\alpha, \beta, \gamma) = (255, 255, 255)$, car le blanc est au contraire obtenu par mélange des trois couleurs primaires.
- La macro `C_JAUNE` correspond au triplet $(\alpha, \beta, \gamma) = (255, 255, 0)$, car le jaune est obtenu par mélange du rouge et du vert.

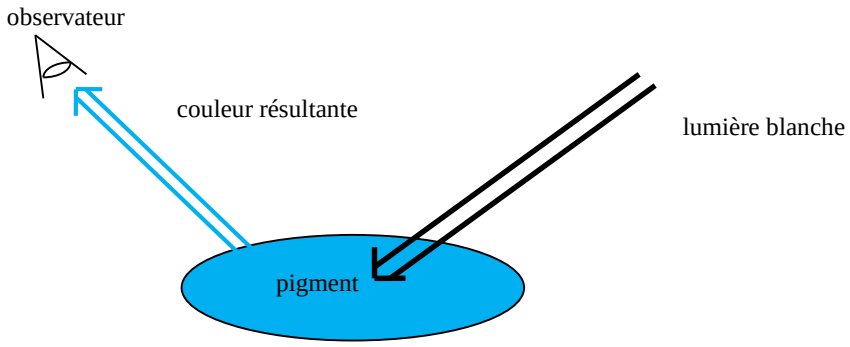
Les niveaux de gris sont obtenus par des triplets (α, β, γ) tels que $\alpha = \beta = \gamma$. La couleur obtenue se rapproche du noir lorsque les valeurs de α, β et γ se rapprochent de 0. Les couleurs secondaires Cyan, Magenta et Jaune sont celles obtenues par mélange des trois couleurs primaires. Elles correspondent aux triplets $(255, 255, 0)$ pour le jaune $(255, 0, 255)$ pour le magenta et $(0, 255, 255)$ pour le cyan.

Ce codage des couleurs n'est cependant pas unique. La lumière blanche est la composée des trois couleurs primaires, qui chacune correspond à une longueur d'onde dans le spectre du visible. Lorsqu'une lumière blanche éclaire un support d'impression, l'observateur qui regarde le support perçoit seulement les couleurs qui n'ont pas été absorbées par ce support. La couleur

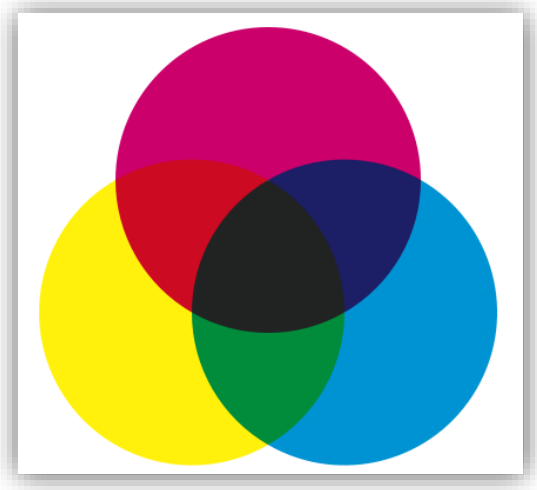
[18] Cf. le volume 2 pour plus de détails sur la SDL (installation, utilisation…).

d'un pigment est donc la partie du spectre de la lumière blanche qui n'a pas été absorbé par ce pigment.

Couleur en éclairage indirect :



Si on mélange deux pigments, la couleur résultante correspondra alors aux longueurs d'ondes absorbées par aucun de ces deux pigments. On parle donc, dans le cas d'un éclairage indirect, de *synthèse soustractive* des couleurs contrairement au cas de l'éclairage direct où la *synthèse* est dite *additive*.



La synthèse soustractive a notamment les conséquences suivantes :

- Le noir est la résultante du mélange des trois couleurs : cyan, magenta et jaune, toutes les longueurs d'onde sont absorbées.
- Le blanc correspond à l'absence de couleurs.
- Le jaune est la couleur complémentaire du bleu. Elle est obtenue en éclairage direct par mélange du vert et du rouge. En éclairage indirect elle est obtenue par absorption du bleu.
- Le cyan et le magenta sont les couleurs complémentaires du rouge et du vert. Le cyan absorbe le rouge et le magenta le vert en éclairage indirect. Le mélange de ces deux couleurs est donc la couleur bleue.

Ainsi pour coder une image d'impression on peut utiliser le système cyan, magenta, jaune (*Cyan Magenta Yellow* ou *CMY*), et chaque teinte t est obtenue par combinaison linéaire de ces trois couleurs :

$$t = \alpha' C + \beta' M + \gamma' J.$$

Avec ce système, les couleurs déjà mentionnées sont représentées par les codes suivants (toujours sur 24 bits) :

couleur	RVB	CMJ
Blanc	(255,255,255)	(0,0,0)
Noir	(0,0,0)	(255,255,255)
Rouge	(255,0,0)	(0,255,255)
Vert	(0,255,0)	(255,0,255)

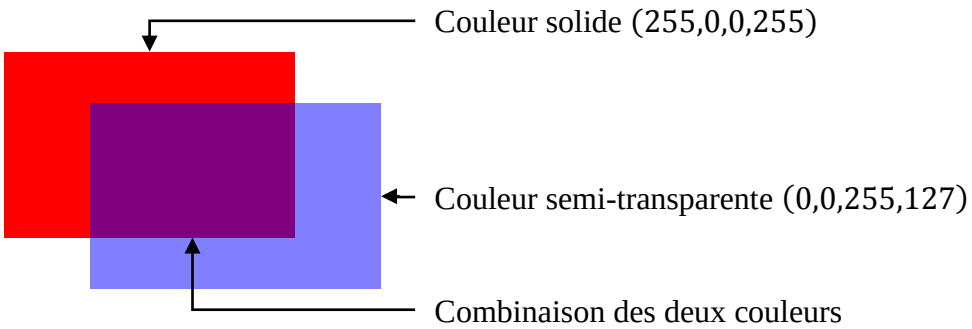
couleur	RVB	CMJ
Bleu	(0,0,255)	(255,255,0)
Cyan	(0,255,255)	(255,0,0)
Magenta	(255,0,255)	(0,255,0)
Jaune	(255,255,0)	(0,0,255)

Le système CMJ est complémentaire (ou duale) du système RGB. Il est utilisé pour coder les pixels d'une image destinée à être imprimée.

D'autres informations que la couleur peuvent être nécessaires pour coder un pixel. On a par exemple souvent besoin de représenter son niveau de transparence, lorsqu'on on veut combiner plusieurs couches séparées (parfois appelées calques) pour former l'image finale.

Le niveau de transparence est généralement représenté comme une 4ème valeur que l'on ajoute au triplet RVB, et que l'on appelle *canal alpha*.

Canal alpha : valeur complétant le codage des couleurs d'un pixel, pour indiquer son niveau de transparence.



Typiquement, si la couleur est codée sur 3 octets, on va utiliser un octet supplémentaire pour le canal alpha, et ainsi représenter le pixel sur 4 octets RVBα. L'objet sera complètement transparent (donc invisible) pour une valeur 0 et complètement opaque pour 255.

3 Variables et constantes littérales

Dans cette section, on s'intéresse aux notions de variables et de constantes, qui sont elles aussi au cœur de la programmation, et seront utilisées dans tous les cours et TP suivants.

3.1 Constantes littérales

Nous avons déjà vu comment définir une constante *symbolique*, i.e. une association entre un identificateur et une valeur qui ne change pas (section 1.2.6). Une constante littérale ne possède pas de nom, il s'agit simplement d'une valeur :

Valeur littérale : (ou *constante littérale*) en programmation, il s'agit d'une valeur donnée *explicitement* dans le code source, par opposition à une variable ou une constante symbolique, qui sont, elles, représentées par un identificateur.

Nous allons distinguer les valeurs numériques des valeurs textuelles.

3.1.1 Valeurs numériques

En langage C, les constantes *entières* peuvent être exprimées en bases 8, 10 et 16 :

- Octal (base 8) : le nombre doit commencer par `0`
- Décimal (base 10) : écriture classique
- Hexadécimal (base 16) : le nombre doit commencer par `0x` (ou `0X`)

exemple : valeur 123

- Octal : `0123`
- Décimal : `123`
- Hexadécimal : `0x123` (ou `0X123`)

Le type d'une constante littérale entière dépend de sa valeur : si elle n'est pas trop grande, alors il s'agit d'un `int`, sinon d'un `long`. À noter que l'on peut forcer le compilateur à considérer la constante comme un entier naturel (`unsigned`) en utilisant le suffixe `u` (ou `U`). Par exemple, `10u` force le compilateur à utiliser le type `unsigned int` au lieu de `int` (i.e. `signed int`) pour représenter 10. De la même façon, le suffixe `l` (ou `L`) permet de forcer le compilateur à utiliser le type `long`. Donc `10l` correspond à la valeur 10 représentée par un `long`. Enfin, on peut combiner les deux suffixes : `10ul` sera la valeur 10 représentée par un `unsigned long`.

Les constantes *réelles* sont exprimées en utilisant un point décimal, par exemple : 10.6. On peut omettre la partie entière ou la partie décimale, mais pas les deux : 10. correspond à 10.0 et .6 correspond à 0.6.

On peut aussi utiliser la notation scientifique en séparant les chiffres significatifs et l'exposant par la lettre `e` (ou `E`). Les expressions suivantes sont équivalentes :

- `12.5`
- `1.25e1`
- `0.125e2`
- `.125e2`

Par défaut, une valeur réelle est de type `double`. Comme pour les entiers, on peut influencer le compilateur à l'aide de suffixes. Pour obtenir un `float`, il faut faire suivre le nombre de la lettre `f` (ou `F`) par exemple : `12.5f`. Pour un long double, on utilise `l` (ou `L`) : `12.5l`.

3.1.2 Valeurs textuelles

Il est possible d'utiliser un *caractère* pour représenter une valeur comprise entre 0 et 255. La valeur numérique correspond alors au code ASCII du caractère indiqué. Pour être *interprété littéralement* par le compilateur, i.e. pour ne pas être confondu avec un identificateur ou un mot-clé, le caractère doit être entouré de deux apostrophes (i.e. `'`).

exemple : le caractère `'A'` correspond à la valeur numérique (code ASCII) 65

Le type d'une constante littérale prenant la forme d'un caractère est `char`. Le caractère *backslash* (i.e. `\`) permet d'exprimer les caractères spéciaux, notamment les caractères non-imprimables :

- `'\''` pour l'apostrophe
- `'\\'` pour le backslash
- `'\n'` pour le retour chariot (i.e. retour à la ligne)
- `'\t'` pour la tabulation
- Etc.

Remarque : afin d'améliorer la lisibilité de vos programmes, il est recommandé d'utiliser un caractère littéral quand on veut manipuler une lettre, et une valeur numérique quand on veut manipuler un entier. Autrement dit : ne pas écrire explicitement le code ASCII d'un caractère pour manipuler celui-ci, mais plutôt utiliser le caractère lui-même.

Cette dualité dans la façon d'interpréter une valeur numérique (soit comme un nombre en soit, soit comment le code ASCII d'un caractère) se retrouve en C dans les formats utilisés pour afficher et saisir des données.

exemples :

- Le programme suivant utilise le format entier %d pour afficher le code ASCII des caractères passés en paramètres de `printf` :

```
printf("%d\t %d\t %d \n",'a','b','c');
97       98      99
```

  Les entiers 97, 98 et 99 sont les codes des caractères `a`, `b` et `c` dans la table ASCII.

- Une boucle affichant l'alphabet en minuscules :

```
for(i='a';i<='z'; i++)
   printf("%c ",i);
```

  Cette fois le spécificateur de format `%c` provoque l'affichage de la valeur de `i` sous forme d'un caractère et non pas d'un entier.

Puisque les caractères peuvent être interprétés comme des entiers, on peut aussi leur appliquer les opérateurs arithmétiques prévu pour ceux-ci. Par exemple, l'expression `'a'-'c'` a un sens : sa valeur est -2, ce qui signifie ici que le caractère `a` est placé deux rangs avant `c` dans la table ASCII. L'expression `'a'+2` correspond donc au caractère `c`, i.e. le caractère situé 2 rangs après `a` dans la table ASCII.

Les constantes *chaînes de caractères* sont des séquences de `char` délimitées par des guillemets (i.e. `"`), par exemple `"qsjkqsdh"`. On peut y inclure des caractères spéciaux, par exemple `"abcdef\nghijkl"` correspond au texte suivant (en raison du `\n`) :

```
abcdef
ghijkl
```

3.2 Variables

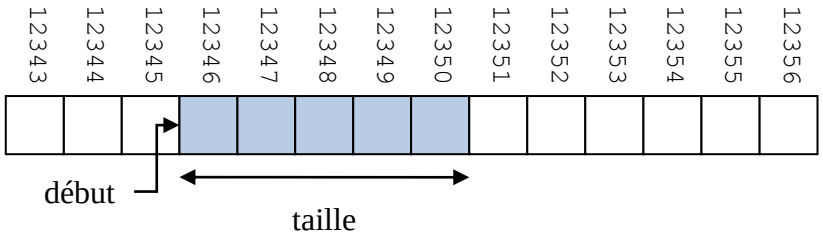
Cette sous-section présente les principaux concepts relatifs aux variables. À noter que des notions plus avancées seront abordées plus tard, cf. section 9.

3.2.1 Définition

La notion de variable est centrale en programmation, car il s'agit de l'objet de base manipulé par un programme.

Variable : association entre une *partie* de la mémoire, un *identificateur* et un *type*.

Une variable est fondamentalement un *morceau de mémoire* permettant de stocker une information. Ce morceau de mémoire est constitué d'un ensemble d'octets voisins, il peut donc être décrit par une *position de départ* et une *taille* (exprimée en octets). Dans la figure ci-dessous, le morceau est représenté en bleu dans la séquence d'octets correspondant à la mémoire.



Comme on le sait, un emplacement de la mémoire est décrit à travers une adresse (i.e. le numéro de l'octet dans la mémoire). Cette adresse est complètement indépendante de ce que la variable représente. Par contre, la taille du morceau de mémoire dépend du *type* de la donnée à stocker : on n'utilise pas le même nombre d'octets pour représenter un caractère que pour un réel.

Le type, à son tour, affecte également la *façon* dont les données sont stockées dans la variable, i.e. le codage utilisé. Le type est donc primordial dans la définition d'une variable, car il indique comment interpréter les bits qu'elle contient pour obtenir sa valeur.

Pour des raisons pratiques, on associe également à notre morceau de mémoire un *identificateur*. Il s'agit d'un nom destiné à être manipulé par un être humain quand il écrit ou lit un programme : c'est bien plus pratique à utiliser que l'adresse numérique de la variable.

L'information placée dans une variable est appelée sa *valeur* :

Valeur d'une variable : contenu de la variable, i.e. séquence des bits présents dans le morceau de mémoire associé à la variable, interprétée en utilisant le type de la variable.

3.2.2 Déclaration

Pour créer une variable, il est nécessaire de la *déclarer* :

Déclaration d'une variable : action de définir une variable en précisant son *identificateur* et son *type*.

En langage C, on réalise cette opération simplement en précisant d'abord le type puis le nom de la variable.

exemple : déclaration d'une variable `x` représentant un entier :

```
int x;
```

Concrètement, l'association constitutive des variables prend la forme d'une structure de données appelée *table des symboles*[19] et gérée par le compilateur. Elle contient la position et le type associés à chaque identificateur déclaré.

[19] Concept qui sera étudié en détail en cours de compilation.

On peut remarquer que la déclaration ne nécessite pas qu'on définisse le morceau de mémoire utilisé par la variable. On a vu que ce morceau était décrit par deux informations : d'abord l'adresse de départ, et ensuite la taille (en octets). Cette adresse est appelée adresse de la variable.

Adresse d'une variable : adresse du premier octet du morceau de mémoire alloué à la variable.

La taille en octets dépend uniquement du type, donc il est normal qu'on n'ait pas besoin de la préciser lors de la déclaration. L'adresse du morceau, elle, est déterminée *automatiquement* : le programmeur ne peut pas la contrôler.

Allocation : action de réserver un morceau de mémoire pour une utilisation particulière.

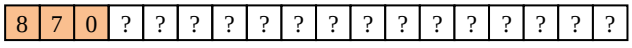
Le fait de réserver une partie de la mémoire pour définir une variable est compatible avec la définition ci-dessus. On dit que cette partie de la mémoire est *allouée* à la variable, et cette opération est appelée l'*allocation*. Lors de la déclaration d'une variable, l'allocation est réalisée automatiquement.

3.2.3 Initialisation

Une mémoire informatique contient *toujours* quelque chose. En effet, chaque bit a une valeur 0 ou 1, même s'il appartient à une partie de la mémoire qui n'a jamais été allouée. Par conséquent, contrairement à ce qu'on pourrait intuitivement supposer, on ne peut pas *effacer* une partie de la mémoire : on peut seulement écrire quelque chose de nouveau à la place de l'ancien contenu.

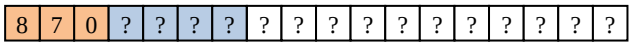
Lors de la déclaration, on se contente de *réserver* de la mémoire : le *contenu* de la mémoire, lui, n'est pas modifié. Cela signifie que la valeur d'une variable juste après sa création est *inconnue* : elle dépend des opérations qui ont été réalisées précédemment dans le morceau de mémoire qui lui a été alloué.

exemple : supposons qu'on a la mémoire suivante :



La partie rouge est déjà utilisée par une variable, qui contient une valeur qu'on y a placé. La partie blanche est inutilisée, et on ne connait pas son contenu.

Supposons maintenant qu'on déclare une nouvelle variable, représentée en bleu :



Alors la valeur de cette variable est lui-même inconnu, car le contenu de la mémoire qui lui a été allouée est lui-même inconnu.

Pour éviter tout problème, il faut systématiquement *initialiser* la variable en même temps qu'on la déclare, ou bien juste après la déclaration.

Initialisation : toute première affectation.

Affectation : opération consistant à modifier la valeur d'une variable.

L'initialisation consiste donc à donner à une variable sa toute première valeur. Il est primordial de prendre l'habitude de toujours initialiser ses variables, car ce type d'oubli provoque fréquemment des erreurs d'exécution difficiles à détecter et à corriger (cf. la section 1.1.5).

L'affectation d'une valeur à une variable se fait en utilisant l'opérateur `=`. On place l'identificateur de la variable toujours à gauche de l'opérateur, et la valeur toujours à droite. On appelle parfois ces deux termes *l-value* (ou left-value) et *r-value* (ou right-value), notamment dans certains messages d'erreur de compilation.

exemple : affecter la valeur 5 à la variable `x` :

```
x = 5;
```

À noter que l'initialisation peut s'écrire en même temps que la déclaration, en procédant de la façon suivante :

```
int x = 5;
```

Remarque : il ne faut surtout pas confondre l'opérateur d'affectation = du langage C avec l'opérateur d'égalité = utilisé en mathématiques. Les deux sont associés au même symbole, mais leurs sémantiques sont complètement différentes :

- L'égalité mathématique est une relation symétrique indiquant le fait que deux objets sont équivalents dans une certaine mesure.
- L'affectation est une action asymétrique consistant à recopier une valeur dans une variable.

L'opérateur d'égalité est représenté en C par le symbole ==, qui sera présenté en section 0. La confusion entre = et == est une erreur très fréquente chez les débutants.

Pour être complet, notez qu'il est possible de rajouter le mot-clé `const` devant une variable pour indiquer que sa valeur ne peut pas être modifiée. On obtient alors une variable constante, à ne pas confondre avec les macros utilisées pour définir des constantes symboliques avec `#define` en section 1.2.6. En effet, les macros sont traitées par le précompilateur et n'existent plus pendant la compilation.

3.2.4 Portée

L'endroit, dans le programme, où la déclaration est faite, influence directement la *portée* de la variable (on dit aussi *scope* de la variable).

Portée d'une variable : portion du code source dans laquelle il est possible d'accéder à la variable.

Si la variable est déclarée à l'extérieur de tout bloc, elle est dite *globale*, et elle sera accessible de n'importe quel point du code source. Si au contraire elle est déclarée à l'intérieur d'un bloc, elle est dite *locale* (à ce bloc) ne sera accessible que dans ce bloc (et les blocs qu'il contient). Dans le premier cas, la portée de la variable est le programme entier, alors que dans le second il s'agit du bloc concerné.

exemple :

```
int a;

void fonction(...)
{  int b;
   ...
}
```

```
int main(int argc, char** argv)
{  char c;
   ...
   {  int d;
      ...
   }
   ...
}
```

La variable `a` est globale, la variable `b` est locale à la fonction `fonction`, la variable `c` est locale à la fonction `main` et la variable `d` est locale au bloc contenu dans la fonction `main`.

Remarque : il est recommandé d'éviter d'utiliser des variables globales. Les variables locales permettent une meilleure gestion des données, et de mieux modulariser les programmes (i.e. de réaliser un meilleur découpage du programme).

Il n'est pas interdit de déclarer plusieurs variables de même nom, du moment que leurs portées sont différentes.

Si on se trouve dans la situation où la portée d'une variable inclut celle d'une autre variable de même nom, la variable la plus locale (déclarée en dernier) masque la première occurrence.

exemple : le programme suivant affichera la valeur 2 :

```
int x=1;
{  int x=2;
   printf("x=%d",x);
}
```

3.2.5 Gestion de la mémoire

En fonction de la portée de la variable déclarée, l'allocation se fait différemment. À ce stade, il est nécessaire de préciser que le C décompose la mémoire en trois parties, utilisées avec des finalités bien précises.

- *Segment de données* : variables globales ;
- *Pile* : variables locales ;
- *Tas* : allocation dynamique, qui sera étudiée en section 11.

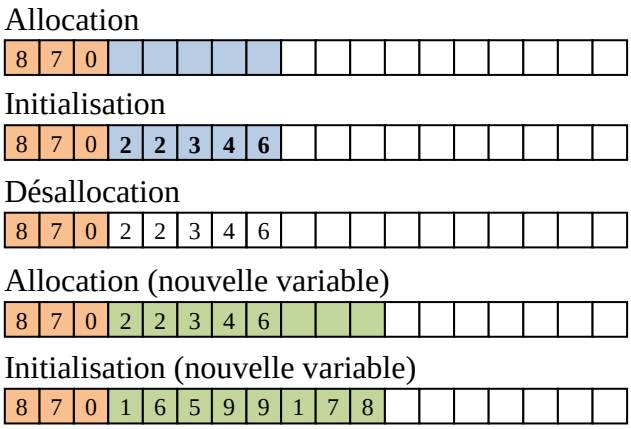
Remarque : cette classification n'est pas tout à fait exacte, mais nous reviendrons dessus plus tard, et la préciserons grâce à des concepts étudiés plus loin dans le cours (cf. section 9).

À chaque fois qu'on exécute une fonction, une certaine quantité de mémoire lui est allouée dans la pile. Cette mémoire est utilisée pour stocker toutes les variables locales à cette fonction. Donc, quand on déclare une variable *locale* à la fonction, l'espace mémoire qu'elle occupe appartient à cette partie de la pile. Quand une fonction se termine, son espace mémoire et toutes les variables qu'il contient sont détruites.

Cette destruction consiste à briser l'association précédemment définie entre le morceau de mémoire alloué à la variable, son identificateur et son type. On dit que la mémoire est *désallouée* :

Désallocation : opération inverse de l'allocation, consistant à libérer une partie de la mémoire qui était jusque-là réservée à une utilisation spécifique.

Remarque : il faut noter que la valeur de la variable n'est pas effacée pour autant : elle reste présente en mémoire. Elle ne sera perdue que si le morceau de mémoire est alloué à une autre variable puis modifié, comme illustré dans la figure suivante :



Dans cette figure, chaque octet contient un chiffre, mais il s'agit bien entendu d'une simplification, puisqu'on sait que la valeur d'une variable dépend de son type, i.e. du codage de l'information sur lequel se base ce type. Dans cet exemple, on déclare d'abord une variable occupant 5 octets (en bleu). La partie orange représente l'espace alloué à une autre variable. Le reste de la mémoire n'est pas utilisé. On initialise ensuite notre variable bleue. Puis on la désalloue : son espace mémoire n'est plus réservé, mais les valeurs de l'initialisation ne sont pas supprimées. On déclare ensuite une nouvelle variable (en vert), visiblement d'un type différent puisqu'elle occupe 8 octets. Remarquez que les valeurs précédentes sont toujours en mémoire : cela montre bien l'importance de toujours initialiser une variable. L'initialisation

réalisée dans la dernière étape permet de remplacer les anciennes valeurs (qui n'utilisaient probablement pas un codage approprié, puisque la variable bleue utilisait un type différent) par de nouvelles valeurs cohérentes avec la nouvelle variable.

Quand on déclare une variable *globale*, l'espace mémoire réservé est pris dans une partie de la mémoire appelée *segment de données*. À la différence de la pile, cette partie de la mémoire est *persistante,* c'est-à-dire qu'elle ne disparait que lorsque le programme lui-même se termine. On ne peut donc pas désallouer ces variables-là.

4 Expressions et opérateurs

Une grande partie d'un code source consiste à effectuer des calculs. Ceux-ci sont réalisés en évaluant des expressions, qui contiennent généralement des opérateurs. Cette section définit d'abord les notions d'expression, d'opérateur et d'opérande, puis passe en revue les principaux opérateurs du langage C et les notions qui leur sont relatives.

4.1 Expressions

Dans le contexte d'un langage de programmation, on appelle *expression* toute représentation d'une valeur.

Expression : élément ou combinaison d'éléments possédant une valeur.

Puisqu'une expression possède une valeur, elle aussi un *type de données*. On distingue deux types d'expressions : *simples* ou *complexes*.

Expression simple : expression composée d'un seul élément.

Une variable seule ou une constante seule (qu'elle soit symbolique ou littérale) sont des expressions simples. L'évaluation d'une expression simple est directe, car on peut obtenir sans aucun calcul la valeur associée.

Expression complexe : expression composée de la combinaison de plusieurs expressions.

L'évaluation d'une expression complexe est indirecte, puisqu'il faut d'abord évaluer les expressions qu'elle contient, puis les combiner, avant d'obtenir sa propre valeur. La valeur et le type d'une expression complexe dépendent non seulement des valeurs des expressions qu'elle contient, mais aussi de la façon dont ces valeurs sont combinées. La combinaison se fait via des fonctions et/ou des opérateurs.

La notion de fonction a été abordée en section 1.2.2. On sait déjà qu'une fonction est un bloc d'instructions auquel on a donné un nom, et associé des paramètres d'entrée et un type de retour. Soit par exemple la fonction `ma_fonction` suivante, qui prend deux entiers en paramètres et renvoie une valeur entière en résultat.

```
int ma_fonction(int a, int b)
{  return (a+b)*2
}
```

La syntaxe de l'appel d'une fonction consiste à préciser le nom de la fonction, puis les paramètres entre parenthèses. Ces paramètres sont des expressions, dont le type doit correspondre à ceux indiqués dans l'en-tête de la fonction. Ainsi, pour l'exemple précédent, on peut effectuer l'appel `ma_fonction(1,2)` : celui-ci constitue une expression de type `int` et de valeur 6. Par contre, l'appel `ma_fonction(1.1,2)` pose problème, car le premier paramètre est un réel.

Un opérateur peut être considéré comme une fonction particulière :

Opérateur : fonction ayant la particularité d'être prédéfinie dans le langage de programmation, et de ne pas respecter la syntaxe des fonctions classiques.

En particulier : le nom d'un opérateur est généralement un signe de ponctuation, son utilisation ne nécessite pas l'utilisation de parenthèses, et il peut être *infixe* (i.e. placé entre ses paramètres, au lieu de les précéder comme pour une fonction). Les paramètres d'un opérateur sont appelés des *opérandes*.

Opérande : paramètre d'un opérateur.

exemple : pour l'expression `1+2` :

- L'opérateur est `+`
- Les opérandes sont `1` et `2`
- La valeur est `3`
- Le type est `int`

Il est bien sûr possible de placer *plusieurs* fonctions ou opérateurs dans une expression, par exemple :

- Plusieurs fonctions : `ma_fonction(ma_fonction(1,2),3)`, dont la valeur est 18
- Plusieurs opérateurs : `(1+2)+(3+4)`, dont la valeur est 10

On peut aussi mélanger fonctions et opérateurs dans une même expression, par exemple : `1+ma_fonction(1,2)` a la valeur 7.

4.2 Opérateurs

Dans cette sous-section, nous présentons d'abord des propriétés communes à tous les opérateurs, puis nous détaillons les principaux opérateurs du langage C. Le but n'est pas d'être exhaustif, et d'autres opérateurs seront introduits dans des cours futurs, en fonction des besoins.

4.2.1 Généralités

Un opérateur est notamment caractérisé par son arité, sa priorité et son sens d'associativité.

Arité : nombre de paramètres d'un opérateur. L'opérateur peut être unaire (1 paramètre), binaire (2 paramètres), ternaire (3 paramètres)…, *n*-aire.

La plupart des opérateurs sont *unaires* ou *binaires*, ce qui signifie qu'ils prennent respectivement un ou deux paramètres.

exemple : l'opérateur d'addition est un opérateur binaire, car il prend deux opérandes : `1+2`

Remarque : la notion d'arité s'applique aussi aux fonctions, en considérant le nombre de paramètres.

Priorité : valeur numérique indiquant dans quel ordre les opérateurs doivent être appliqués dans le cas où une expression en contient plusieurs.

Les opérateurs du C sont classés selon 16 niveaux de priorité, le plus haut niveau étant 15 (et donc le plus bas le 0).

exemple : l'addition `+` a une priorité inférieure à la multiplication `*`, donc `1+2*3` est égal à `1+(2*3)`

Sens d'associativité : indique si les expressions contenant plusieurs opérateurs de même priorité doivent être évaluées en considérant les opérandes de gauche à droite ou de droite à gauche.

exemple : l'addition est évaluée *de gauche à droite*, ce qui signifie que pour l'expression `1+2+3`, on va d'abord évaluer `1+2` (qui vaut `3`) puis `3+3` (ce qui donne un total de `6`).

Remarque : il est recommandé d'utiliser des parenthèses en cas d'incertitude sur les priorités des opérateurs utilisés, voire simplement pour améliorer la lisibilité de l'expression.

Attention de ne pas confondre un *opérateur* et le *symbole* utilisé pour le représenter. En langage C, plusieurs opérateurs distincts sont représentés par le même symbole.

exemples :

- Symbole `-` : correspond à la fois à l'opérateur *opposé*, qui est unaire (ex. : `-1`) et à l'opérateur *soustraction*, qui est binaire (ex. : `1-2`)

- Symbole `/` : correspond à la fois à l'opérateur de division réelle si on l'applique à au moins un réel, et à la division entière (ou euclidienne) si ses deux opérandes sont entiers.

4.2.2 Opérateurs arithmétiques

Les opérateurs arithmétiques respectent les règles usuelles des mathématiques. À noter que lorsqu'on mélange différents types numériques, des conversions implicites peuvent avoir lieu (cf. section 4.2.7).

Ces opérateurs s'appliquent à des valeurs numériques (y compris les caractères, qui comme on l'a vu sont représentés par leur code ASCII) et renvoient un résultat lui aussi numérique.

opérateur	description	priorité	associativité
`-`	opposé	14	DàG
`+`	addition	12	GàD
`-`	soustraction	12	GàD
`*`	multiplication	13	GàD
`/`	division (si les deux opérandes sont entiers, il s'agit d'une division entière)	13	GàD
`%`	modulo : reste de la division entière (les deux opérandes doivent être entiers)	13	GàD

exemples :

- `1+2` est une expression de type `int` dont la valeur est `3`
- `1.0*4.5` est une expression de type `double` dont la valeur est `4.5`

4.2.3 Opérateurs de comparaison

Les opérateurs de comparaison prennent comme opérandes des expressions numériques, et produisent un résultat *booléen*, i.e. vrai ou faux. Comme on l'a déjà vu, il n'y a pas de type booléen en C. On utilise des valeurs numériques à la place :

- Zéro représente *faux* ;
- Une valeur non-nulle représente *vrai*.

opérateur	description	priorité	associativité
`==`	égal	9	GàD
`!=`	différent	9	GàD
`<=`	inférieur ou égal	10	GàD
`<`	strictement inférieur	10	GàD
`>`	strictement supérieur	10	GàD
`>=`	supérieur ou égal	10	GàD

exemples :

- `0>5` est une expression de type `int` dont la valeur est `0` (faux)
- `1.0>0.5` est une expression de type `int` dont la valeur est `1` (vrai)

Remarque : les débutants font souvent l'erreur de confondre les opérateurs d'affectation (=) et d'égalité (==), car le symbole utilisé en mathématique pour l'égalité est utilisé en C pour l'affectation. Cette erreur est difficile à détecter, car elle ne donne pas lieu à une erreur de compilation : elle provoque une erreur d'exécution (cf. section 1.1.5). Il est donc nécessaire de faire bien attention lorsqu'on utilise ces opérateurs.

Remarque : il faut faire attention lors de la comparaison de réels, car le codage employé étant approché (comme expliqué en section 2.3), il est possible que le résultat de la comparaison soit inattendu. Considérons par exemple le programme suivant :

```
float x= 0.1;
if(3*x == 0.3)
   printf("oui\n");
else
   printf("non\n");
```

On s'attends à ce que le programme affiche `oui`, car *mathématiquement*, on a bien $3x = 3 \times 0{,}1 = 0{,}3$. Mais *informatiquement*, on sait que la valeur $0{,}1$ ne peut pas être codée en base 2 de façon *exacte*, et par conséquent le programme affichera `non`.

4.2.4 Opérateurs logiques

Les opérateurs logiques sont supposés s'appliquer à des valeurs *booléennes* (vrai ou faux) et renvoyer un résultat lui aussi booléen. Puisqu'il n'y a pas de type booléen en C, et qu'on représente vrai et faux par des entiers, alors ces opérateurs s'appliquent aussi à des valeurs entières et renvoient une valeur entière.

opérateur	description	priorité	associativité
!	non (ou négation)	14	DàG
&&	et (ou conjonction)	5	GàD
||	ou (ou disjonction)	4	GàD

exemples :

- `0 && 1` est une expression de type `int` dont la valeur est `0` (faux)
- `0 || 5` est une expression de type `int` dont la valeur est `1` (vrai)

Remarque : l'évaluation d'une expression correspondant à une combinaison d'opérateurs logique est soumise à condition :

- L'opérateur et est appliqué tant qu'on ne rencontre pas d'opérande faux (auquel cas la valeur de l'expression est faux) ;
- L'opérateur ou est appliqué tant qu'on ne rencontre pas d'opérande vrai (auquel cas la valeur de l'expression est vrai).

exemples :

- **`1 && 1 && 0 && 1 && 1`** : on évalue seulement la partie rouge de l'expression, car dès qu'on rencontre le `0`, on sait que l'expression est fausse sans avoir besoin d'en connaitre la suite.
- **`0 || 1 || 0 || 0`** : là-aussi, on évalue seulement la partie rouge de l'expression, car dès qu'on rencontre le `1`, on sait que l'expression est vraie sans avoir besoin d'en connaitre la suite.

Conclusion : quand on écrit une expression logique, les termes susceptibles de provoquer des erreurs doivent être placés après le test concernant l'erreur.

exemple : on veut tester si un rapport $10/x$ est supérieur à 1. Pour cela, on doit s'assurer que x n'est pas nul. On peut le faire des deux façons suivantes :

- `10/x>1 && x !=0` : avec cette méthode, si $x = 0$, alors on provoquera une *erreur d'exécution fatale* quand ont effectuera une *division par zéro* en calculant $10/x$.
- `x !=0 && 10/x>1` : avec cette méthode, *seul* le premier terme de l'expression sera évalué si $x = 0$, et on n'effectuera donc *pas* de division par zéro.

Remarque : cette propriété ne s'applique pas aux autres opérateurs. Pour ceux-là, l'ordre d'évaluation des opérandes d'un opérateur n'est pas défini par le C. Ainsi la valeur de l'expression `(x=0)*(x=3)` sera `0` mais la valeur de `x` après l'évaluation dépendra du compilateur (ce sera donc soit `0`, soit `3`).

4.2.5 Opérateurs bit-à-bit

Ces opérateurs permettent de travailler directement sur la représentation binaire des données, bit par bit. On ne les utilisera pas en cours ni en TP.

opérateur	description	priorité	associativité
`~`	complément à un	14	DàG
`&`	et bit-à-bit	8	GàD
`^`	ou exclusif bit-à-bit	7	GàD
`|`	ou bit-à-bit	6	GàD
`>>`	décalage à droite	11	GàD
`<<`	décalage à gauche	11	GàD

4.2.6 Opérateurs divers

La plupart des opérateurs suivants seront utilisés plus tard dans ce cours.

opérateur	description	priorité	associativité
`( )`	parenthèses	15	GàD
`[ ]`	tableaux	15	GàD
`.`	champ	15	GàD
`->`	pointeur de champ	15	GàD
`( )`	transtypage (cast)	14	DàG
`sizeof()`	obtenir le nombre d'octet occupé par une variable ou un type de données	14	DàG
`&`	référençage : obtenir l'adresse d'une variable	14	DàG
`*`	déréférençage ou indirection : obtenir la valeur stockée à une adresse	14	DàG
`=`	affectation	2	DàG

L'opérateur `sizeof(t)` permet de connaitre l'espace mémoire occupé par une valeur du type `t` spécifié. Il est donc relatif au codage utilisé pour représenter les données (cf. section 2.1.2). Il peut aussi être appliqué directement à une expression, pour obtenir l'espace qu'elle occupe en mémoire. Appliqué à un tableau (section 6), `sizeof` renvoie l'espace mémoire total occupé par tous les éléments du tableau.

Le résultat de sizeof est unentier de type `size_t`. Ce type a été défini spécialement pour cet usage, i.e. représenter des tailles (comme son nom l'indique). En pratique, dans le cadre de nos TP, on le considérera comme un entier classique.

L'opérateur de référençage, aussi appelé opérateur d'adressage, permet de connaitre l'adresse d'une variable. Il faut le placer devant la variable considérée. Par exemple, supposons qu'on a une variable `x` déjà déclarée. Alors l'expression `&x` correspond à l'adresse de cette variable.

L'opérateur d'affectation est intéressant, car on pourrait supposer intuitivement qu'il ne produit pas de valeur. Par exemple, dans `x=4`, on place la valeur `4` dans `x`, mais quelle est la valeur et le type de l'expression ? Eh bien le type est celui de la *left-value* (cf. section 3.2.3), c'est-à-dire la variable concernée par l'affectation, et la valeur est celle contenue dans cette variable après l'affectation, donc ici : `4`. À noter que des mécanismes de conversion automatique peuvent intervenir lors des affectations, cf. la sous-section 4.2.7.

À noter également que contrairement à ce que l'utilisation du symbole *égale* pourrait laisser supposer, l'opérateur d'affectation n'est *pas commutatif*. Autrement dit, `x=y` et `y=x` ne sont pas équivalents : dans le premier cas, on recopie la valeur de `y` dans `x`, alors que dans le second on recopie la valeur de `x` dans `y`.

Il existe d'autres opérateurs qui ne sont pas mentionnés ici, car leur utilisation est interdite dans le cadre de ce cours (et des TP qui vont avec). Il s'agit en particulier de l'opérateur ternaire `?`, qui parait pratique mais tend à rendre le code source illisible. Même chose pour les opérateurs qui combinent arithmétique et affectation, tels que `+=`, `*=`, etc. On autorise seulement l'utilisation de `++` et `--`, qui sont respectivement équivalents à :

- `x++` est équivalent à `x=x+1`
- `x--` est équivalent à `x=x-1`

4.2.7 Exercices

`x=y=z=7` : équivalente à `x=(y=(z=7))` car le sens d'associativité de l'opérateur d'affectation est de droite à gauche. Donc `7` sera affecté d'abord à `z`, puis la valeur de cette expression `z=7` (qui est aussi `7`) sera affectée à `y`, et ainsi de suite.

`x==y==1` : équivalente à `(x==y)==1` car le sens d'associativité de l'opérateur d'égalité est de gauche à droite. On teste donc d'abord si `x` et `y` sont égaux, puis on compare l'entier résultant de ce test avec la valeur `1`.

`7*4/3` : équivalente à `(7*4)/3` car les opérateurs `*` et `/` ont la même priorité, et l'ordre d'évaluation de l'expression est donc déterminé par leur sens d'associativité (qui est de la gauche vers la droite).

`7/4*3` : équivalente à `(7/4)*3`, pour les mêmes raisons.

4.3 Conversions

Lorsqu'on veut appliquer des opérateurs ou des fonctions à des valeurs *hétérogènes* (i.e. de différents types), il est parfois nécessaire de *convertir* certaines de ces valeurs. Par exemple, il existe un opérateur `+` permettant d'additionner deux réels, et un autre opérateur `+` permettant d'additionner deux entiers, mais aucun permettant d'additionner un réel et un entier.

Pour effectuer cette opération, on a deux possibilités :

- Soit transformer le réel en entier, puis appliquer l'addition entière, et on obtient alors un entier pour résultat ;
- Soit transformer l'entier en réel, puis appliquer l'addition réelle, et on obtient alors un réel pour résultat.

Conversion : action de changer le type de données d'une expression.

exemple : changer un `int` de valeur `10` en un `double` de valeur `10.0`

On distingue deux types de conversions : implicite ou explicite.

4.3.1 Conversion implicite

Les conversions implicites sont généralement effectuées lorsqu'il est impossible d'appliquer directement un opérateur, en raison du type de ses opérandes.

Conversion implicite : effectuée *automatiquement* sans aucune intervention du programmeur. Le compilateur la déduit du code source.

Le compilateur va réaliser une conversion tout seul, quand c'est possible, afin de permettre d'appliquer l'opérateur. La conversion se fait en suivant les règles suivantes :

- Les opérandes de type (`unsigned`) `char` ou (`unsigned`) `short` sont systématiquement convertis en (`unsigned`) `int`.
- Si un des opérandes est un `long double`, les autres opérandes sont convertis en `long double`, et le résultat est un `long double`.
- Sinon, si un des opérandes est un `double`, les autres opérandes sont convertis en `double`, et le résultat est un `double`.
- Sinon, si un des opérandes est un `float`, les autres opérandes sont convertis en `float`, et le résultat est un `float`.
- Sinon, si un des opérandes est un `unsigned long`, les autres opérandes sont convertis en `unsigned long`, et le résultat est un `unsigned long`.
- Sinon, si un des opérandes est un `long`, les autres opérandes sont convertis en `long`, et le résultat est un `long`.
- Sinon, si un des opérandes est un `unsigned int`, les autres opérandes sont convertis en `unsigned int`, et le résultat est un `unsigned int`.
- Sinon, tous les opérandes sont forcément des `int`.

Donc, on peut en déduire que la conversion automatique se fait de manière à préserver au maximum l'information contenue dans la variable. Si on a l'expression `10.5+1`, le résultat sera un réel de valeur `11.5`, et non pas un entier de valeur `11`.

Pour une affectation, la valeur qui est affectée est convertie au type de la variable qui va la contenir. Dans ce cas-là, on peut donc avoir une perte de précision. Par exemple, la valeur réelle 10,5 devient 10 si on la place dans une variable entière.

exemple : Soit le programme suivant :

```
float a=2.1, b=-2.1;
int x;
x = a;
printf("x = %d\n",x);
x = b;
printf("x = %d\n",x);
```

Il affiche le résultat suivant :

```
x = 2
x = -2
```

On observe que la conversion implicite des réels vers les entiers supprime la partie décimale (troncature).

4.3.2 Conversion explicite

La conversion explicite est utilisée par le programmeur pour certains calculs particuliers, dans lequel il veut contrôler manuellement la façon dont les types sont manipulés.

Conversion explicite : conversion *demandée* par le programmeur grâce à l'opérateur de *transtypage* (aussi appelé cast).

La conversion explicite est effectuée au moyen de l'opérateur de transtypage (`cast`). La syntaxe est la suivante :

```
(type) expression
```

Par exemple, si on veut transformer un réel en entier pour pouvoir appliquer l'opérateur modulo :

```
int x=10,resultat;
float y=3;
resultat = x%(int)y;
```

Supposons, au contraire, que l'on veut diviser un entier par un autre entier, mais en utilisant la division réelle (par opposition à la division euclidienne). Considérons le programme suivant :

```
int x=10,y=3;
float resultat;
resultat = x/y;
```

La valeur contenue dans `resultat` sera alors `3.0`, et non pas `3.333...`. En effet, puisque `x` et `y` sont toutes les deux des variables entières, c'est la division entière qui sera appliquée quand on utilisera `/`, et son résultat est `3` (reste `1`). Cette valeur entière est ensuite transformée en un réel pour être stockée dans la variable réelle `resultat`, et on obtient finalement `3.0`.

Donc ce programme ne fait pas ce que l'on veut (il s'agit typiquement d'une erreur d'exécution, cf. section 1.1.5). Considérons plutôt le programme suivant :

```
int x=10,y=3;
float resultat;
resultat = x/(float)y;
```

Le fait de rajouter l'opérateur de conversion explicite devant `y` permet de transformer sa valeur en un réel. La division devient alors elle aussi réelle, et le résultat de `x/y` est `3.333...`. La variable `resultat` contient finalement bien la valeur `3.333`..., comme désiré.

5 Conditions et boucles

La plupart des langages de programmation contiennent des instructions capables de réaliser différents types de tests et d'autres permettant de répéter le même traitement plusieurs fois. Ce sont ces instructions qui sont décrites dans ce chapitre, pour le langage C.

Remarque : il existe en C d'autres instructions de contrôle, en plus de celles abordées ici, comme par exemple `goto`. Nous avons choisi de ne pas les présenter car leur utilisation diminue la lisibilité des programmes. Il est interdit de les utiliser dans le cadre de ce cours (y compris en TP, examen de TP et examen écrit).

5.1 Instructions de test

Le langage C permet trois types de tests différents : simple, avec alternative, et multiple. Pour chacun, on distingue deux parties :

- Une ou plusieurs *conditions* ;
- Un ou plusieurs blocs.

Condition : expression dont la valeur est interprétée de façon *booléenne* : la condition peut être soit *vraie*, soit *fausse*.

Le principe est d'associer une condition à un bloc. Autrement, si une condition donnée est vraie, alors une certaine partie du programme sera exécutée. Sinon, c'est une autre partie qui sera exécutée, voire aucune partie.

Remarque : une fois l'instruction de test exécutée, le programme reprend son cours normalement.

5.1.1 Test simple

Avec le test *simple*, on associe une seule condition à un seul bloc de code. Si la condition est vraie, ce bloc est exécuté. Sinon, il n'est pas exécuté.

Cette instruction est notée `if`, et sa syntaxe est la suivante :

```
if(condition)
{  instruction 1;
   ...
   instruction n;
}
```

Notez l'*absence* de point-virgule (i.e. `;`) après la condition. Remarquez aussi l'indentation utilisée (cf. section 1.2.4) : vous devez la reproduire dans vos propres programmes.

Le fonctionnement est le suivant. Tout d'abord, l'expression `condition` est évaluée :

- Si elle est vraie (i.e. si sa valeur est un entier non-nul) alors le bloc est exécuté
- Si elle est fausse (i.e. si sa valeur est zéro) alors le bloc n'est *pas* exécuté.

Comme indiqué précédemment, dans les deux cas, l'exécution du programme continue normalement ensuite.

exemple : considérons le programme suivant :

```
int x;
...
if(x>0)
{  printf("La valeur de x est positive\n");
}
printf("Le test est terminé\n");
```

Pour `x=10` on va obtenir l'affichage suivant :

```
La valeur de x est positive
Le test est terminé
```

Et pour `x=-3`, on aura l'affichage suivant :

```
Le test est terminé
```

Remarque : si le bloc ne contient qu'une seule instruction, on peut omettre les accolades.

exemple : le `if` de l'exemple précédent peut se simplifier de la façon suivante :

```
if(x>0)
   printf("La valeur de x est positive\n");
```

Toutefois, il faut faire très attention avec cette syntaxe, car si vous l'utilisez, toute instruction supplémentaire sera considérée comme extérieure au `if`.

exemple : supposons qu'on a le code source suivant :

```
if(x>0)
   printf("La valeur de x n'est pas négative\n");
   printf("Et je dirais même qu'elle est positive\n");
printf("Le test est terminé\n");
```

Alors, une valeur `x=10` produira bien le résultat attendu :

```
La valeur de x n'est pas négative
Et je dirais même qu'elle est positive
Le test est terminé
```

Mais une valeur négative comme `x=-3` affichera également la ligne supplémentaire (en italique dans le code source) :

```
Et je dirais même qu'elle est positive
Le test est terminé
```

En effet, il s'agit de la deuxième instruction après le `if`, et aucune accolade ne vient délimiter le bloc, donc cette ligne n'appartient pas au `if`.

Une autre erreur classique des programmeurs débutant consiste à placer un point-virgule juste après la condition, de la façon suivante :

```
if(x>0);
   printf("La valeur de x est positive\n");
```

Il ne s'agit pas d'une erreur de compilation, la syntaxe est correcte. En effet, le compilateur considère que le bloc du `if` est constitué d'une seule instruction, qui est… rien du tout. Donc, si `x>0`, ici on ne fera rien de spécial. Cela signifie aussi que le `printf` ne se trouve pas dans le `if`, mais à l'extérieur. Et par conséquent, il sera exécuté quelle que soit la valeur de `x`.

Il est possible de combiner plusieurs `if`, de manière à effectuer des séquences de tests. On dit alors qu'on *imbrique* les `if` :

Imbrication : fait d'utiliser une instruction dans le bloc d'une autre instruction.

exemple :

```
if(x>0)
{  printf("La valeur de x est positive\n");
   if(x>10)
      printf("Je dirais même que la valeur de x est supérieure à 10\n");
}
```

Pour `x=5`, on a l'affichage suivant :

```
La valeur de x est positive
```

Alors que pour `x=15`, on a l'affichage suivant :

```
La valeur de x est positive
Je dirais même que la valeur de x est supérieure à 10
```

5.1.2 Test avec alternative

Avec le test simple, on exécute un bloc seulement si une condition est vraie, et on ne l'exécute pas si la condition n'est pas vraie. Il est possible de proposer un bloc alternatif, à exécuter quand la condition n'est pas vraie (plutôt que de ne rien faire). On parle alors de test *avec alternative.*

Ce bloc additionnel est introduit par le mot clé else, en respectant la syntaxe suivante :

```
if(condition)
{  instruction 1a;
   ...
   Instruction na;
}
else
{  instruction 1b;
   ...
   instruction nb;
}
```

exemple :

```
int x;
...
if(x>0)
   printf("La valeur de x est positive\n");
else
{  printf("La valeur de x n'est pas positive\n");
}
printf("Le test est terminé\n");
```

Pour `x=10`, on aura exactement le même affichage que pour le test simple. Par contre, pour `x=-3`, on aura l'affichage suivant :

```
La valeur de x n'est pas positive
Le test est terminé
```

Comme pour le `if`, il est possible d'omettre les accolade si le bloc du `else` ne contient qu'une seule instruction. On peut donc simplifier l'exemple précédent de la façon suivante :

```
if(x>0)
   printf("La valeur de x est positive\n");
else
   printf("La valeur de x n'est pas positive\n");
```

Il est aussi possible d'imbriquer d'autres `if` dans le bloc du `else`.

5.1.3 Test multiple

Dans certains cas, on a besoin de comparer une variable donnée à plusieurs valeurs différentes, et non pas une seule comme précédemment.

exemple : soit une variable entière `x`. On veut effectuer un traitement différent suivant que la valeur de la variable est 1, 2, 3 ou plus. Si on utilise `if`, on va devoir procéder par *imbrications successives* :

```
if(x==1)
{  ...
}
else
{  if(x==2)
   {  ...
   }
   else
   {  if(x==3)
      {  ...
      }
      else
      {  ...
      }
   }
}
```

L'instruction `switch` permet d'effectuer le même traitement, mais avec une syntaxe plus compacte. Le mot-clé `switch` reçoit entre parenthèses la variable à tester. Chaque valeur à comparer est ensuite indiquée dans le bloc du `switch`, en utilisant le mot-clé `case`, de la façon suivante :

```
switch(variable)
{  case valeur_a:
      instruction 1a;
      ...
      instruction na;
   case valeur_b:
      instruction 1b;
      ...
      instruction nb;
   ...
}
```

Le fonctionnement est le suivant. La variable spécifiée va être comparée successivement à chaque valeur proposée, jusqu'à ce qu'il y ait égalité. Si la variable n'est égale à aucune valeur, alors on sort simplement du `switch` sans rien faire de plus. Si on trouve une valeur égale, alors on exécute toutes les instructions situées en-dessous du `case` correspondant. Attention : cela inclut non seulement les instruction du case associé à la valeur, mais aussi celles de *tous les autres `case` suivants* !

Or, il arrive fréquemment qu'on désire que seules les instructions placées directement en dessous du case correspondant soient exécutées, mais pas les suivantes. Pour arriver à ce comportement, il faut ajouter l'instruction `break` juste avant le case suivant, comme indiqué ci-dessous :

```
switch(variable)
{  case valeur_a:
      instruction 1a;
      ...
      instruction na;
      break;
   case valeur_b:
      instruction 1b;
      ...
      instruction nb;
      break;
   ...
}
```

exemple : on veut que le traitement soit le même si x a la valeur 2 ou la valeur 3, mais on veut aussi un traitement spécifique pour la valeur 1 :

```
switch(x)
{  case 1:
      instruction1-1;
      instruction1-2;
      break;
   case 2:
   case 3:
      instruction23-1;
      instruction23-2;
      break;
}
```

Il est possible de définir un cas *par défaut*, c'est-à-dire de préciser un comportement si la variable ne correspond à *aucune* des valeurs spécifiées dans les `case`. On utilise pour cela le mot-clé `default` à la place de `case`, de la façon suivante :

```
   ...
   case ...:
      ...
      break;
   default:
      ...
}
```

À noter que ce cas par défaut se place tout à la fin du `switch`.

exemple : on reprend le programme qui utilisait des `if`, au début de cette sous-section, et on les remplace par un `switch` équivalent :

```
switch(x)
{  case 1:
      ...
      break;
   case 2:
      ...
      break;
   case 2:
      ...
      break;
   default:
      ...
}
```

Remarque : la structure de contrôle `switch` s'utilise uniquement sur des valeurs entières (donc aussi sur des caractères).

Notez que l'instruction `break` est utilisable dans d'autres situations, comme par exemple dans des boucles. Mais cela rend les programmes peu lisibles et difficiles à comprendre. Par conséquent, dans le cadre de ce cours, vous n'avez le droit d'utiliser `break` que dans les `switch`.

5.1.4 Exercices

1) En utilisant `if`, écrivez un programme qui teste la valeur d'une variable entière `x` contenant un chiffre entre 1 et 4, et affiche ce chiffre en toutes lettres. Pour les autres chiffres, le programme doit afficher `"autre"`.

```
if(x==1)
   printf("un\n");
else
   if(x==2)
      printf("deux\n");
   else
      if(x==3)
         printf("trois\n");
      else
         if(x==4)
            printf("quatre\n");
      else
         printf("autre\n");
```

2) Même question en utilisant `switch`.

```
switch(x)
{  case 1:
      printf("un\n");
      break;
   case 2:
      printf("deux\n");
      break;
   case 3:
      printf("trois\n");
      break;
   case 4:
      printf("quatre\n");
      break;
   default:
      printf("autre\n");
}
```

5.2 Instructions de répétition

On distingue trois boucles différentes en C : *tant que*, *répéter* et *pour*. Nous allons voir qu'en termes d'expressivité, elles ont strictement le même pouvoir (i.e. elles permettent de faire exactement la même chose). Cependant, la convention veut qu'on les utilise dans des contextes différents.

Une boucle se compose de quatre éléments essentiels :

- Un *bloc* d'instruction, qui sera exécuté un certain nombre de fois ;
- Une *condition*, comme pour les instructions de test. Cette condition porte sur au moins une variable, que l'on appellera la variable de boucle. Il peut y avoir plusieurs variables de boucle pour une seule boucle.
- Une *initialisation*, qui concerne la variable de boucle. Cette initialisation peut être directement réalisée par l'instruction de répétition, ou bien laissée à la charge du programmeur.
- Une *modification*, qui concerne elle aussi la variable de boucle. Comme pour l'initialisation, elle peut être intégrée à l'instruction de répétition, ou bien laissée à la charge du programmeur.

Si l'un de ces 4 éléments manque, alors la boucle est incorrecte. Cela provoque soit une erreur de compilation, soit une erreur d'exécution, en fonction de l'élément manquant et du type de boucle.

Le principe d'une boucle est le suivant :

1) La variable de boucle est initialisée.
2) Le bloc d'instructions est exécuté.
3) La condition est évaluée. En fonction de son résultat :
   a. Soit on sort de la boucle.

b. Soit on recommence à partir du point 2.

Remarque : les points 2 et 3 peuvent être inversés, en fonction du type de boucle considéré, comme on va le voir plus loin.

Si l'initialisation manque (point 1) alors une erreur d'exécution se produira. Pensez à toujours initialiser une variable.

Il est absolument nécessaire qu'à chaque itération, la valeur de la variable de boucle soit modifiée. En fonction du type de boucle, cette modification est soit intégrée dans l'instruction de boucle, soit à réaliser dans le bloc d'instruction. De plus, la modification réalisée doit impérativement provoquer, au bout d'un nombre *fini* de répétitions, un changement de la valeur de la condition. Si ces deux contraintes ne sont pas respectées, alors la condition reste toujours la même, et on ne sortira jamais de la boucle.

Boucle infinie : boucle dont la condition ne change jamais, ce qui provoque un nombre infini de répétitions. Il s'agit généralement d'une erreur d'exécution.

Autrement dit : la raison pour laquelle la boucle s'arrête, c'est que la variable de boucle a été modifiée de façon à ce que la condition change de valeur.

On emploie le mot *itérer* pour indiquer qu'on exécute une fois la boucle :

Itération : réalisation d'un *cycle complet* de boucle. Ceci inclut forcément l'exécution du bloc d'itération de la boucle, mais aussi éventuellement le test de condition, et aussi la modification, en fonction du type de boucle considéré.

5.2.1 Boucle *tant que*

La boucle *tant que* permet de répéter l'exécution d'un bloc d'instructions tant qu'une condition est vraie. Elle repose sur l'instruction `while`. La condition est exprimée sous la forme d'une expression placée entre parenthèses juste après l'instruction `while`. La condition est elle-même suivie du bloc d'instructions à répéter.

```
while(condition)
{  instruction 1;
   ...
   instruction n;
}
```

Notez l'absence de point-virgule (i.e. `;`) après la condition.

Remarque : comme indiqué précédemment, la condition doit forcément porter sur la variable de boucle. Celle-ci doit impérativement être initialisée avant le `while`. De plus, cette variable doit forcément être modifiée dans le bloc d'instructions.

Un `while` fonctionne de la façon suivante. Tout d'abord, la condition est évaluée. Si elle est vraie (i.e. valeur non-nulle), alors le bloc d'instructions est exécuté. La variable de boucle étant supposée être modifiée dans ce bloc, la valeur de la condition est susceptible d'avoir changé. Donc, après avoir exécuté le bloc, on re-teste la condition. Si elle est toujours vraie, on exécute le bloc une autre fois. On répète ce traitement jusqu'à ce que la condition soit fausse, et on continue alors l'exécution du reste du programme.

Remarque : si la condition est fausse dès le début, le bloc n'est même pas exécuté une seule fois.

exemple : considérons le programme suivant :

```
int x;
...
x = 0;          // initialisation
while(x<10)
{  printf("x=%d\n",x);
   x = x + 1;   // modification
}
```

Il provoquera l'affichage suivant :

```
x=0
x=1
x=2
x=3
x=4
```

```
x=5
x=6
x=7
x=8
x=9
```

Si on remplaçait l'initialisation par `x=5`, on n'aurait que la colonne de droite. Si on la remplaçait par `x=20`, alors rien ne serait affiché du tout.

Par convention, on utilise les boucles *tant que* pour effectuer des répétitions dans le contexte suivant :

- On ne *sait pas* à l'avance *combien* de fois on devra répéter le bloc (par exemple parce que le nombre de répétition dépend du comportement de l'utilisateur).
- Et on ne *sait pas* à l'avance *si* on doit exécuter le bloc ne serait-ce qu'une seule fois.

Dans le cadre de ce cours, il est interdit aux étudiants d'utiliser `while` dans un autre contexte, sauf si c'est demandé explicitement.

5.2.2 Boucle *répéter*

La boucle *répéter* permet d'exécuter un bloc d'instructions d'abord, de façon inconditionnelle, puis de répéter son traitement tant qu'une condition est vraie. Une boucle de ce type est introduite par l'instruction `do`, suivie du bloc d'instructions. La condition est exprimée sous la forme d'une expression entre parenthèses, placée après un `while`, comme pour la boucle *tant que*. La différence ici, est que le `while` est placé *après* le bloc, et qu'elle est suivie d'un point-virgule (i.e. `;`). N'oubliez surtout pas ce point-virgule.

```
do
{  instruction 1;
   ...
   instruction n;
}
while(condition);
```

Un `do...while` se comporte comme un `while`, à l'exception du fait que le bloc sera *toujours* exécuté au moins *une* fois, même si la condition est fausse.

Son principe de fonctionnement est le suivant. Le bloc d'instructions est exécuté. Puis, la condition est testée. Si elle est fausse, on sort de la boucle et on continue l'exécution normale du programme. Si elle est vraie, on exécute de nouveau le bloc, puis on re-teste la condition. Comme pour le `while`, il est donc nécessaire d'initialiser manuellement la variable de boucle avant le `do`. De plus, cette variable doit apparaître dans la condition, et doit être modifiée dans le bloc, sinon la boucle sera infinie.

exemple : on adapte le programme précédent :

```
int x;
...
x = 0;          // initialisation
do
{  printf("x=%d\n",x);
   x = x + 1;   // modification
}
while(x<10);
```

Il provoquera exactement le même affichage qu'avec le `while` :

```
x=0
x=1
x=2
x=3
x=4
```

```
x=5
x=6
x=7
x=8
x=9
```

Comme avec le `while`, si on remplaçait l'initialisation par `x=5`, on n'aurait que la colonne de droite. Par contre, i on la remplaçait par `x=20`, alors on aurait l'affichage suivant (alors que `while` n'affichait rien du tout) :

```
x=20
```

Comme pour le `while`, la boucle `do...while` s'utilise par convention dans une situation bien spécifique :

- On ne *sait pas* à l'avance *combien* de fois on devra répéter le bloc.
- On *sait* à l'avance qu'on doit exécuter *au moins une fois* le bloc.

C'est donc le deuxième point qui est différent, par rapport à la situation décrite pour le `while`.

5.2.3 Boucle *pour*

La boucle *pour* permet de répéter l'exécution d'un bloc en automatisant les phases d'initialisation et de modification de la variable de boucle. En effet, le mot-clé `for`, qui précède le bloc, est suivi de trois expressions séparées par des points-virgules :

- La première permet d'effectuer l'initialisation : elle est exécutée une seule fois, au début de la boucle ;
- La deuxième est la condition : elle est évaluée au début de chaque itération ;
- La troisième est la modification : elle est exécuté à la fin de chaque itération.

```
for(intialisation; condition; modification)
{  instruction 1;
   ...
   instruction n;
}
```

À noter qu'il est possible d'initialiser/modifier plusieurs variables de boucle en même temps en utilisant des virgules dans les expressions :

```
for(x=1,y=10; x=y; x++,y--)
```

Le principe de fonctionnement du `for` est le suivant. Tout d'abord, à la différence de `while` et `do...while`, l'initialisation est intégrée à l'instruction. Mais comme pour les autres boucles, cette opération est réalisée une seule fois, au tout début du traitement de la boucle. De plus, il reste nécessaire de déclarer la variable avant la boucle.

Après l'initialisation, la condition est évaluée. Si elle est vraie, alors le bloc d'instructions est exécuté. À la différence de `while` et `do...while`, *il ne faut surtout pas* modifier la variable de boucle dans ce bloc. En effet, pour le `for` la modification est intégrée à l'instruction. Cette modification est effectuée *automatiquement* juste après que le bloc a été exécuté.

Après exécution du bloc et application de la modification, la condition est évaluée. Si elle est toujours vraie, on recommence le même traitement (bloc puis modification puis condition). Sinon, on sort de la boucle.

exemple : programme équivalent au précédent basé sur `while` et `do...while` :

```
int x;
...
for(x=0;x<10;x++)
{  printf("x=%d\n",x);
}
```

L'affichage sera le même que précédemment. Si on initialise `x` à `5` dans la boucle, l'affichage sera là-aussi le même. Si on initialise `x` à `10`, alors la condition sera fausse lors du premier test, et le bloc ne sera pas exécuté (comme pour le `while`, et à la différence du `do...while`).

Nous utiliserons les boucles *pour* uniquement quand on *saura* à l'avance *combien* d'itérations devront être effectuées. Il est interdit d'utiliser un for quand le nombre de répétitions ne peut pas être prévu.

5.2.4 Exercices

1) Écrivez un programme calculant la somme des `N` premiers entiers grâce à une boucle `for`. Tracez le déroulement en donnant le tableau de situation pour `N=5`.

```
int i,somme=0;
for(i=1;i<=N;i++)
   somme=somme+i;
```

2) Mêmes questions avec une boucle `while`.

```
int i=1,somme=0;
while(i<=N)
{  somme=somme+i;
   i++;
}
```

3) Mêmes questions avec une boucle `do...while`.

```
int i=1,somme=0;
do
{  somme=somme+i;
   i++;
}
while(i<=N);
```

5.3 Analyse d'un programme

Dans cette sous-section, on s'intéresse à deux problèmes : comprendre un programme qu'on n'a pas écrit, et montrer formellement qu'un programme fonctionne. En effet, écrire un programme est une chose, mais comprendre un programme écrit par un tiers en est une autre. La méthode la plus efficace est alors d'exécuter pas-à-pas l'algorithme, et c'est ce qu'un *tableau de situation* permet de faire.

Tester un programme de façon empirique (i.e. en l'exécutant avec différents paramètres) permet de montrer qu'il fonctionne dans certains cas. Mais si on veut une preuve absolue, cette approche est peu efficace car on devrait envisager toutes les combinaisons de valeurs de paramètres possibles. Or, celles-ci peuvent être quasi-infinies. Une méthode plus efficace consiste à effectuer une preuve générale, indépendamment des valeurs particulières des paramètres. C'est ce que nous ferons en utilisant la notion de *correction d'un programme*.

5.3.1 Tableau de situation

Pour visualiser le traitement réalisé par un programme, on peut utiliser un *tableau de situation* :

Tableau de situation : tableau dont chaque colonne correspond à une variable utilisée dans un programme, et chaque ligne à une modification de la valeur d'une variable.

Ce type de tableau permet de tracer l'évolution des valeurs des variables au cours de l'exécution du programme.

exemple :

```
1  int x=4;
2  int y=10;
3  y=y+x;
4  x=y-x;
5  y=y-x;
```

ligne	x	y
1	**4**	-
2	4	**10**
3	4	**14**
4	**10**	14
4	**10**	4

Remarque : lorsqu'on se sera familiarisé avec la notion d'adresse d'une variable, ou bien lorsque nous manipulerons des pointeurs (cf. section 10), nous rajouterons parfois une colonne à ce tableau : l'adresse de la variable considérée.

exercices : reprenez les programmes des exercices de la section 5.2.4, et donnez leurs tableaux de situation.

1) Boucle `for` :

```
1  int i,somme=0;
2  for(i=1;i<=N;i++)
3     somme=somme+i;
4  ...
```

ligne	i	somme
1	_	**0**
2	**1**	0
3	1	**1**
2	**2**	1
3	2	**3**
2	**3**	3
3	3	**6**

ligne	i	somme
2	**4**	6
3	4	**10**
2	**5**	10
3	5	**15**
2	**6**	15
4	6	15

2) Boucle `while` :

```
1  int i=1,somme=0;
2  while(i<=N)
3  {  somme=somme+i;
4     i++;
5  }
6  ...
```

ligne	i	somme
1	**1**	**0**
2	1	0
3	1	**1**
4	**2**	1
2	2	1
3	2	**3**

ligne	i	somme
4	**4**	6
2	4	6
3	4	**10**
4	**5**	10
2	5	10
3	5	**15**

4	**3**	3
2	3	3
3	3	**6**

4	6	15
2	6	15
6	6	15

3) Boucle `do...while` :

```
1  int i=1,somme=0;
2  do
3  {   somme=somme+i;
4      i++;
5  }
6  while(i<=N);
7  ...
```

ligne	i	somme
1	**1**	**0**
3	1	**1**
4	**2**	1
6	2	1
3	2	**3**
4	**3**	3
6	3	3
3	3	**6**
4	**4**	6

ligne	i	somme
6	4	6
3	4	**10**
4	**5**	10
6	5	10
3	5	**15**
4	6	15
6	6	15
7	6	15

5.3.2 Correction d'un programme itératif

Pour vérifier théoriquement la correction d'un algorithme itératif, il faut s'assurer que :

- L'algorithme n'effectue qu'un nombre *fini* d'itérations (i.e. on n'a pas une boucle infinie).
- L'algorithme calcule le *résultat correct*.

Pour effectuer ces vérifications, on utilise le concept d'*invariant de boucle* :

Invariant de boucle : propriété vraie à la sortie de la boucle *après chaque itération*. La propriété doit être vraie même si on n'effectue aucune itération.

Cette une propriété dépend en général du nombre d'itérations k effectuées. Pour prouver que la propriété d'invariant $P(k)$ est vraie après chaque itération on utilise le *principe de récurrence* :

Principe de récurrence : soient un entier k_0 et $P(k)$ une propriété dépendant d'un entier $k \geq k_0$. Si :

i) $P(k_0)$ est vraie ;

ii) $\forall k : (k \geq k_0 \text{ et } P(k) \text{ vraie}) \Rightarrow (P(k+1) \text{ vraie})$

Alors :

$\forall k \geq k_0 : P(k)$ est vraie.

La propriété d'invariance doit être choisie de manière à montrer qu'après la dernière itération, on obtient bien le résultat souhaité.

exemple 1 : calcul de $som = 1 + x + x^2 + \cdots + x^n$, pour n entier, $n \geq 0$.
On propose le programme suivant :

```
som = 0 ;
for(i=0;i<=n;i++)
   som = 1 + som*x;
```

L'indice `i`, qui est la *variable de boucle*, est incrémenté de 1 à chaque itération, en partant de 0 et jusqu'à n. Donc, le nombre d'itérations sera exactement $n+1$. En particulier, le bloc est toujours exécuté au moins une fois, car $n \geq 0$, donc la récurrence peut commencer à partir du rang 1.

On note som_k la valeur de la variable som après k itérations. On choisit comme invariant de boucle de ce programme la propriété $P(k)$ suivante :

$$P(k) : som_k = 1 + x + \cdots + x^{k-1}$$

où $k \geq 1$ est le nombre d'itérations effectuées. On va maintenant la vérifier par récurrence.

- Pour $k = 1 : som_1 = 1 + 0 = x^0$
- On suppose l'invariant P(k) : $som_k = 1 + x + \cdots + x^{k-1}$ vérifié après k itérations, vérifions sa correction après $k+1$ itérations :
  - Après l'exécution du bloc de la boucle, on a :
    $$som_{k+1} = 1 + (1 + x + \cdots + x^{k-1})x = 1 + x + \cdots + x^k$$
  - Donc $P(k+1)$ est vraie.
- D'après le principe de récurrence, $P(k)$ est vraie quel que soit le nombre d'itérations k.
- Le programme se termine après $n+1$ itérations, donc la valeur de som à la fin de l'exécution est bien : $1 + x + \cdots + x^n$.

exemple 2 : calcul du PGCD (Plus Grand Commun Diviseur) de 2 entiers $n > 0$ et $m > 0$

On propose le programme suivant :

```
int reste, a=m, b=n, pgcd;
while(b!=0)
{  reste = a % b;
   a = b;
   b = reste;
}
pgcd = a;
```

Pour montrer la correction de cet algorithme, on a besoin de la propriété arithmétique suivante :

P_6 : Pour tous entiers $m > 0$ et $n > 0$:

$$\text{pgcd}(m,n) = \text{pgcd}(n, m \bmod n)$$

où $m \bmod n$ désigne le reste de la division Euclidienne de m par n.

Vérifions alors la correction de l'algorithme. On note a_k et b_k les valeurs des variables a et b après k itérations. $0 \leq b_{k+1} = a_k \bmod b_k < b_k$, donc la valeur de b diminue strictement à chaque itération. La valeur de b sera donc nulle ou bout d'un nombre fini d'itérations (nombre qui dépend de m et n).

On choisit pour invariant de boucle la propriété suivante :

$$P(k) : \text{pgcd}(a_k,b_k) = \text{pgcd}(m,n)$$

On va maintenant la vérifier par récurrence.

- Avant la première itération, $a_0 = m$ et $b_0 = n$, donc $P(0)$ est vraie.
- On suppose $P(k)$ vraie, et on étudie l'exécution d'une $(k+1)^{\text{ème}}$ itération :
  - Après l'exécution du bloc de la boucle on a :
    $$a_{k+1} = b_k \text{ et } b_{k+1} = a_k \bmod b_k$$
  - D'après la propriété P_6 énoncée plus haut, on a donc :
    $$\text{pgcd}(a_{k+1}, b_{k+1}) = \text{pgcd}(a_k, b_k)$$
  - D'après l'hypothèse de récurrence, on en déduit :
    $$\text{pgcd}(a_{k+1},b_{k+1}) = \text{pgcd}(m,n)$$
  - Finalement, $P(k+1)$ est vraie.

- D'après le principe de récurrence, $P(k)$ est vraie quel que soit le nombre d'itérations k.
- Le programme s'arrête lorsque $b_k = 0$, et on a alors :
$$a_k = \text{pgcd}(a_k, 0) = \text{ppgcd}(m, n)$$
- Ceci justifie l'affectation `pgcd=a` à la fin du programme.

exemple 3 : méthode d'exponentiation rapide pour calculer x^n, pour deux entiers $x \geq 0$ et $n > 0$

On propose le programme suivant :

```
int i=n, y=x, result=1;
while(i!=0)
{  if(i%2)
      result = result * y;
   y = y * y;
   i = i / 2;
}
```

On note $n = (b_l b_{l-1} \dots b_0)_2$ la décomposition binaire de n. On note i_k, y_k et r_k les valeurs des variables i, y et r après k itérations.

Pour vérifier la terminaison de l'algorithme, il faut montrer que $\forall k : i_k = (b_l b_{l-1} \dots b_k)_2$ (laissé à faire). L'algorithme effectue exactement $l + 1$ itérations, où $l + 1$ est égal à la longueur de la décomposition binaire de n.

On choisit l'invariant de boucle suivant : $\forall k : y_k = 2^{2^k}$. On peut aussi l'exprimer sous la forme :

$$P(0) : r_0 = 1$$
$$P(k) : r_0 = r_k = \left(x^{2^0}\right)^{b_0}\left(x^{2^1}\right)^{b_1} \dots \left(x^{2^{k-1}}\right)^{b_{k-1}} \text{ pour } k > 0$$

Vérifions cette propriété par récurrence :

- Par définition, $P(0)$ est vraie.
- $P(1)$ est vraie, puisque $r_1 = r_0 \times y_0 = 1 \times x = x$
- Supposons la propriété $P(k)$ vraie après $k \geq 1$ itérations.
  - Si $i_k \bmod 2 = b_k = 1$:
  $$r_{k+1} = r_k \times y_k = \left(x^{2^0}\right)^{b_0}\left(x^{2^1}\right)^{b_1} \dots \left(x^{2^{k-1}}\right)^{b_{k-1}} \times x^{2^k}$$
  - Si $i_k \bmod 2 = b_k = 0$:
  $$r_{k+1} = r_k = \left(x^{2^0}\right)^{b_0}\left(x^{2^1}\right)^{b_1} \dots \left(x^{2^{k-1}}\right)^{b_{k-1}} \times 1$$
  - Dans tous les cas :
  $$r_{k+1} = \left(x^{2^k}\right)^{b_k}\left(x^{2^{k-1}}\right)^{b_{k-1}} \dots \left(x^{2^1}\right)^{b_1}\left(x^{2^0}\right)^{b_0}$$
  - La propriété $P(k+1)$ est donc vérifiée.
- D'après le principe de récurrence, $P(k)$ est vraie quelque soit le nombre k d'itérations.

L'algorithme donne le résultat attendu :

- Si $n = 0 : r_0 = 1 = x^0$
- Si $n = (b_l b_{l-1} \dots b_0)_2 > 0$, le nombre d'itérations est $l + 1$ (on a $l \sim \log_2(n)$)

Remarque : le code d'un algorithme naïf calculant x^n ressemble à :

```
r=1;
for(i=1;i<=n;i++)
   r=r*x;
```

Le nombre d'itérations de cet algorithme est donc exactement égal à n. Comparons le nombre d'itérations de cet algorithme avec l'algorithme étudié pour certaines valeurs de n :

n	Itérations pour l'algorithme naïf	Itérations pour l'algorithme étudié
$255 = 2^8 - 1$	255	8
$65\,535 = 2^{16} - 1$	$65\,535$	16
$2^{40} - 1$	2^{40}	$> 10^{10}$

On constate que pour les grandes valeurs de n, l'algorithme étudié effectue incomparablement moins d'itérations que l'algorithme naïf, ce qui justifie son nom d'algorithme d'exponentiation rapide.

6 Tableaux

Les tableaux sont des types de données complexes du langage C, par opposition aux types simples utilisés jusqu'à présent (cf. section 2). Pour mémoire, une valeur de *type simple* correspond à une seule information : un entier, ou un réel, ou un caractère, etc. Par opposition, un *type complexe* correspond à une collection de plusieurs valeurs.

Dans cette section, nous introduisons la notion de tableau et décrivons comment elle s'utilise avec les notions déjà étudiées. Par la suite, nous reviendrons sur les tableaux lorsqu'ils seront liés aux nouveaux concepts vus en cours (notamment les fonctions et les pointeurs). Cette section n'est donc qu'introductive.

6.1 Définition

Les tableaux constituent un type de données complexe *homogène* :

Type complexe homogène : type de données décrivant une information composite, constituée de plusieurs valeurs qui sont toutes elles-mêmes du même type (qui peut être lui-même simple ou complexe).

Autrement dit, un tableau est un type lui-même basé sur un autre type. L'intérêt des tableaux est de permettre de désigner un ensemble de valeurs à l'aide d'un seul identificateur. En effet, une seule variable de type tableau représentera toute une collection de valeurs (d'où l'appellation de type complexe), au lieu d'une seule pour les types simples.

Élément d'un tableau : désigne l'une des valeurs contenues dans le tableau.

Comme indiqué dans la définition, toutes ces valeurs sont homogènes, c'est-à-dire de même type de données. Il peut s'agir de n'importe quel type du langage C, que ce soit un type simple ou un type complexe. Si ce type est un tableau, on a un tableau de tableaux et on parle de tableau *multidimensionnel*, sinon, on parle de tableau *unidimensionnel*.

Lorsqu'on *déclare* une variable de type tableau, on précise le *type* des données contenues dans le tableau (i.e. le type de ses éléments), et sa *taille*.

Taille d'un tableau : nombre d'éléments que le tableau peut contenir *au plus*.

exemple : définition d'un tableau de 10 entiers appelé `tab`

```
short tab[10];
```

Comme on le voit ci-dessus, il y a 3 éléments dans la déclaration d'un tableau :

- D'abord le type de ses éléments (ici : `short`) ;
- Ensuite l'identificateur désignant le tableau (ici : `tab`) ;
- Et enfin, la taille du tableau, toujours entre crochets `[]` (ici : 10).

Remarque : une fois qu'un tableau a été déclaré, *on ne peut plus changer sa taille*. En effet, comme pour les déclarations de variables étudiées jusqu'à présent, il s'agit d'une allocation *statique* de la mémoire. Il faut donc définir un tableau suffisamment grand pour la tâche à réaliser.

Parfois, on veut parler d'un type de tableau *en général* (par opposition à une variable de type tableau en particulier), qui est caractérisée par une taille donnée. Cette généralisation est notée en rajoutant les crochets (soit vides `[]` , soit avec la taille du tableau si on la conait) à la fin du type des éléments. Par exemple, quand on veut parler des tableaux d'entiers de type `int` en général, sans en connaître la taille, on écrit simplement `int[]`. Si on veut parler des tableaux de réels `float` de taille 10, alors on note le type `float[10]`.

6.2 Mémoire

Un tableau est représenté en mémoire de façon *contigüe*, c'est-à-dire que le premier élément est suivi du second, qui est directement suivi du troisième, etc.

Représentation contigüe : tous les éléments constituant le tableau sont placés en mémoire les uns après les autres.

Pour un type de données occupant s octets en mémoire, un tableau de taille n occupera $n \times s$ octets en mémoire. Attention de ne pas confondre la *taille* du tableau (nombre d'éléments) avec l'*espace* qu'il occupe en mémoire. L'exemple suivant représente le tableau déclaré précédemment, qui contient 10 entiers de type `short`. Remarquez comme ces entiers sont placés les uns après les autres dans la mémoire (chacun d'eux occupant 2 octets consécutifs).

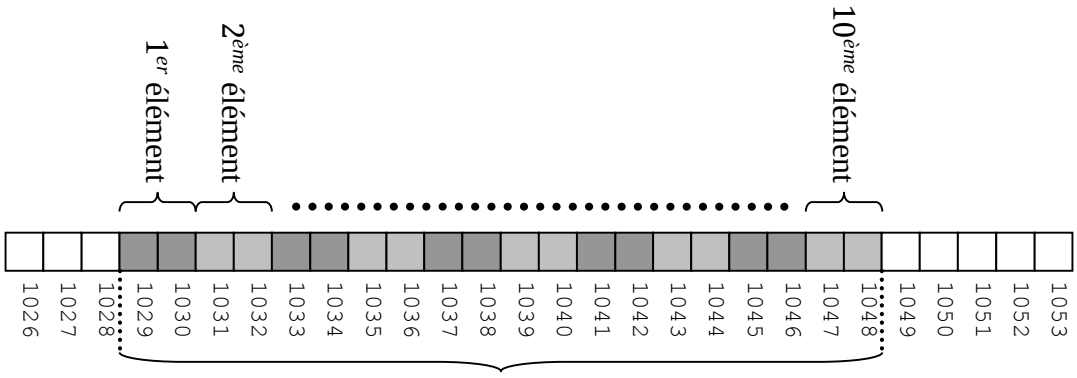


tableau de 10 `short`

Si le premier élément du tableau est stocké à partir de l'adresse a, alors le deuxième sera stocké à partir de l'adresse $a + s$, le troisième à partir de l'adresse $a + 2s$, etc.

En raison de l'homogénéité des types des valeurs contenues, il est donc très simple de calculer l'adresse de n'importe lequel des éléments d'un tableau. En effet, on sait que l'opérateur `sizeof(xxxx)` renvoie l'espace mémoire occupé par un type `xxxx`, exprimé en octets. Donc, pour un tableau de type `xxxx[]` dont le premier élément est situé à l'adresse a, l'élément `xxxx[k]` aura pour adresse l'octet $a + \text{sizeof(xxxx)} \times k$.

Remarque : lorsque l'on applique `sizeof` à une variable de type tableau, l'opérateur renvoie l'espace mémoire total qu'il occupe, i.e. son nombre d'éléments multiplié par l'espace mémoire occupé par un élément. Si on appelle `tab` le tableau de l'exemple ci-dessus, alors `sizeof(tab)` renvoie donc $10 \times 2 = 20$ octets.

Lorsque le nom d'un tableau apparait dans une expression, son évaluation donne *l'adresse de son premier élément*[20]. Par abus de langage, on appelle aussi cette adresse l'*adresse du tableau* : comme les éléments du tableaux sont stockés de façon contigüe, on peut considérer que les deux sont équivalentes.

Adresse d'un tableau : adresse du premier élément du tableau.

Ainsi, pour le tableau des exemples précédents, dont l'identificateur est `tab`, alors l'expression `tab` est une adresse qui vaut 1029 (puisque dans la figure précédente, cette adresse correspond au premier des octets constituant le tableau).

Il faut bien comprendre que l'identificateur d'un tableau n'est pas une adresse *en soit*. Il représente bien le tableau, de la même façon qu'un identificateur de variable classique représente la variable et non pas son adresse. La différence est que dans la plupart des expressions, cet identificateur est remplacé par l'adresse du tableau.

[20] En réalité, il existe trois exceptions à cette règle, sur lesquelles nous reviendrons plus tard.

6.3 Manipulation

Puisqu'un tableau est une séquence d'éléments, il nous faut un moyen d'accéder à ces éléments individuellement. Cet accès individuel se fait en précisant son *index* (ou sa *position*).

Index d'un élément : numéro de l'élément dans la séquence constituant le tableau.

Attention, il s'agit bien de la position de l'élément dans le tableau, et non pas de son adresse dans la mémoire. Dans l'exemple précédent, on a donné la formule pour calculer l'adresse d'un élément, qui était $a + \text{sizeof(xxxx)} \times k$. Ici, l'index de cet élément correspond à la valeur k.

Remarque : en langage C, l'indexation (i.e. la numérotation) des éléments des tableaux commence à *la valeur zéro*, et non pas à la valeur 1 comme en mathématiques. Notez que ce n'est pas le cas pour tous les langages informatiques.

exemple : accès au $4^{ème}$ élément du tableau `tab`

```
tab[3]
```

Un élément de tableau se manipule comme une variable classique : on peut l'introduire dans des expressions, exactement comme on le ferait habituellement.

exemples :

- On veut stocker la valeur 52 dans le $4^{ème}$ élément du tableau

```
tab[3] = 52;
```

- On veut afficher la valeur du $4^{ème}$ élément du tableau

```
printf("%d",tab[3]);
```

Comme on l'a vu précédemment, si l'identificateur désignant le tableau est employé dans une expression, il représente l'*adresse* du début du tableau. Donc, les expressions `tab` et `&tab[0]` désignent la *même adresse* : celle du premier élément du tableau. De plus, l'expression `&tab[k]` sera égale à l'adresse de l'élément numéro k du tableau.

Comme l'allocation mémoire d'un tableau est *statique*, son adresse ne *peut pas* être modifiée. Par conséquent, l'indentificateur d'un tableau n'est pas une left-value (cf. section 4.2.6). Autrement dit, on ne peut pas utiliser le nom d'un tableau comme opérande de gauche dans une affectation. Ainsi, il est interdit d'écrire quelque chose comme : `tab = ...`.

La conséquence de cette observation est qu'il n'est pas possible de modifier un tableau de manière globale, i.e. tous les éléments à la fois : il faut procéder élément par élément. Par exemple, même si on veut affecter la valeur 0 à tous les éléments d'un tableau, il faut le faire un élément à la fois.

exemple : on veut remplir notre tableau `tab` avec les 10 premiers nombres pairs

```
tab[0]=0;
tab[1]=2;
tab[2]=4;
tab[3]=6;
tab[4]=8;
```

```
tab[5]=10;
tab[6]=12;
tab[7]=14;
tab[8]=16;
tab[9]=18;
```

Bien entendu, de par leur fonctionnement, les tableaux se prêtent particulièrement à l'utilisation des boucles.

exercice : initialisez `tab` comme dans l'exemple précédent, mais en utilisant une boucle `for`.

```
short i,tab[10];
for(i=0;i<10;i++)
   tab[i]=2*i;
```

Une autre façon d'initialiser un tableau est de la faire lors de sa *déclaration*, en indiquant les valeurs d'initialisation entre *accolades* (i.e. `{` et `}`), séparées par des *virgules* (i.e. `,`).

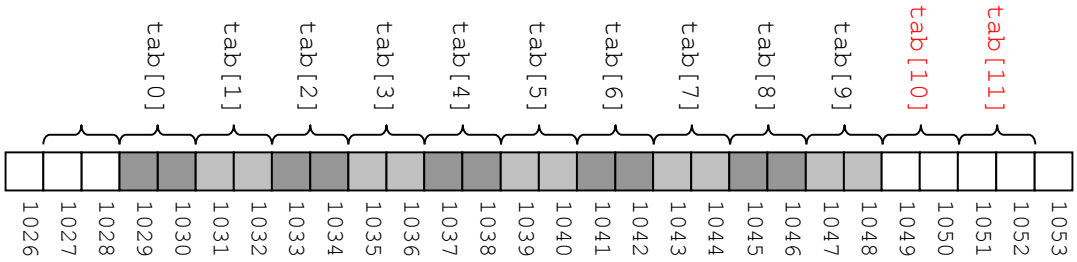
exemple : on initialise un tableau avec les 10 premiers nombres pairs

```
short tab[]={0,2,4,6,8,10,12,14,16,18};
```

Dans ce cas-là, il n'est *pas nécessaire* d'indiquer la *taille* du tableau entre crochets, car elle est spécifiée par le nombre d'éléments présents entre les accolades.

Lors de l'accès à un élément du tableau, le compilateur C ne vérifie pas si l'index utilisé est *valide*, c'est-à-dire compris entre 0 et la taille du tableau moins un. Si un index invalide est utilisé, on provoque ce que l'on appelle un *débordement de tableau* :

Débordement de tableau : tentative d'accès à des éléments situés hors du tableau, comme indiqué dans la figure suivante.



Cette observation est importante, car elle provoque souvent des erreurs d'exécution (difficiles à corriger, cf. la section 1.1.5). Considérons le tableau représenté dans la figure, qui contient 10 valeurs de type `short`. Seul l'espace nécessaire pour ces 10 valeurs a été réservé pour le tableau, soit l'espace mémoire allant de l'adresse 1029 à l'adresse 1048 (incluses). Le reste de la mémoire est probablement réservé pour stocker d'autres variables. Supposons qu'on a une variable de type `char` stockée à l'adresse 1049. Alors, si on effectue l'opération `tab[10]=5`, le programme va écrire un code représentant 5 sur deux octets, aux adresses 1049 et 1050. Il va donc écraser la valeur de notre caractère. C'est la perte de cette valeur qui peut provoquer une erreur d'exécution.

6.4 Chaînes de caractères

En langage C, les tableaux sont utilisés pour représenter les chaînes de caractères :

Chaîne de caractères : *séquence* de caractères formant du texte.

Une chaîne peut contenir aussi bien des caractères imprimables que non-imprimables (cf. section 2.4). Une chaîne définie sous la forme d'une *constante littérale* (cf. section 3.1) est notée à l'aide des guillemets (i.e. `"`).

exemple : une chaîne de caractère composée des 5 premières lettres de l'alphabet sera notée `"abcde"`

Les tableaux utilisés en C pour représenter des chaînes sont toujours de type `char`. Donc, formellement, le type d'une chaîne de caractère est noté `char[]`. De plus, un *caractère spécial* est utilisé pour indiquer la fin de la chaîne :

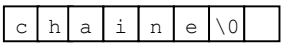
Caractère de fin de chaîne : noté `'\0'`, ce caractère non-imprimable indique que la chaîne se termine au caractère précédent.

exemples :

- La chaîne de caractère `"chaines"` dans un tableau de taille 8 :

c	h	a	i	n	e	s	\0

- La chaîne de caractère "chaine" dans un tableau lui aussi de taille 8 :



Remarquez que dans le second cas, le dernier élément du tableau n'est pas utilisé. L'intérêt du caractère de fin de chaîne est qu'il permet de stocker dans un tableau de taille *fixe* des chaînes de caractères de taille *variable*. En effet, on a vu que la mémoire allouée à un tableau est réservée de façon statique, et que donc on ne peut pas changer sa taille. Mais on peut changer son contenu, i.e. modifier les caractères qu'il contient. Et en particulier, on peut raccourcir ou rallonger ce texte. Il est donc nécessaire d'utiliser un système permettant cela : c'est le but de `'\0'`.

Comme on l'a déjà vu en section 2.4, une valeur de type `char` peut aussi bien être interprétée, en C, comme un caractère que comme un entier (le code ASCII associé au caractère). Cela signifie qu'un tableau de `char` peut lui-même être interprété comme un tableau d'entiers ou comme une chaîne de caractères. Autrement dit, une chaîne de caractères est *forcément* représentée par un tableau de `char`, mais un tableau de `char` ne correspond *pas* forcément à une chaîne de caractères.

Remarque : pour être valide, une chaîne doit forcément se terminer par un `'\0'`. Si ce caractère de fin de chaîne manque, il est très probable qu'une *erreur d'exécution* (cf. section 1.1.5) se produise (l'erreur ne sera pas détectée à la compilation).

L'initialisation d'une chaîne de caractère *à la déclaration* peut être effectuée en utilisant la syntaxe suivante classique des tableaux :

```
char s[] = {'c', 'h', 'a', 'i', 'n', 'e', '\0'};
```

Remarquez que dans ce cas-là il faut bien penser à inclure le caractère de fin de chaîne (représenté en rouge ci-dessus) dans la séquence de caractères. En effet, il y a une ambiguïté qui ne peut pas être résolue automatiquement. Comme on l'a vu, un tableau de `char` peut aussi être utilisé pour manipuler des séquences d'entiers, et rien ne permet de distinguer ici cette utilisation de l'initialisation d'une chaîne.

Il existe également une méthode *spécifique* aux chaînes de caractères, utilisant la notation des guillemets :

```
char s[] = "chaine";
```

Notez qu'avec cette méthode, il ne faut pas inclure de `'\0'` final : cette opération est réalisée automatiquement, car les guillemets indiquant sans ambiguïté qu'il s'agit bien d'une chaîne (et non pas d'un tableau d'entiers).

De la même façon, les fonctions spécialisées dans le traitement des chaînes de caractères rajoutent automatiquement le caractère de fin de chaîne lors de la saisie ou de l'initialisation d'une chaîne.

Les fonctions `scanf` et `printf` manipulent les chaînes au moyen du code de format `%s`.

exemple :

```
char ch[99];
scanf("%s",ch);
printf("La chaîne est %s.",ch);
```

Remarquez qu'à la différence de ce qu'on faisait pour les variables de type simple (`int`, `char`, etc.), on n'utilise pas ici l'opérateur `&` sur notre variable dans le `scanf`. Ceci est dû au fait que le paramètre passé au `scanf` est `ch`, qui est de type `char[]`, et qui est donc *déjà* une adresse. Autrement dit : pas la peine d'obtenir son adresse avec `&`, puisque c'est déjà une adresse.

En supposant que l'utilisateur saisisse la chaine "abcdef", on obtiendra l'affichage suivant :

```
La chaîne est abcdef.
```

À noter que par défaut, `scanf` arrête la saisie dès qu'il rencontre un *caractère de séparation* : retour à la ligne, espace, tabulation, etc. Ainsi, si l'utilisateur entre le texte `abcd efgh ijkl`, alors seulement `abcd` sera placé dans le tableau `ch`.

On peut aussi manipuler `gets` et `puts` pour réaliser des opérations similaires, comme on utilisait `putc` et `getc` pour les simples caractères.

L'utilisation du caractère de fin de chaîne permet, comme on l'a vu, de stocker des chaînes de tailles variables dans un tableau de taille fixe. Cependant, il faut s'assurer de ne pas effectuer de débordement de tableau. Par exemple, si on déclare un tableau de taille 5 mais qu'on y stocke la chaîne `"abcdefgh"`, alors la fin de la chaîne sera écrite dans une partie de la mémoire qui n'a pas été réservée à ce tableau. On rencontrera alors certainement des erreurs d'exécution.

Remarque : les débutants font couramment l'erreur de ne pas compter le `'\0'` dans la taille du tableau, ce qui risque aussi de provoquer un débordement. Pour reprendre l'exemple précédent, un tableau de taille 5 ne pourra pas stocker la chaine `"abcde"`, car il n'y aura plus de place pour le `'\0'`. On pourra par contre stocker la chaine `"abcd"`, qui prendra la forme `{'a', 'b', 'c', 'd', '\0'}` en mémoire, soit 5 valeurs de type `char`.

6.5 Tableaux multidimensionnels

Les tableaux qu'on a étudiés jusqu'à présent sont dits *unidimensionnels*, et ils se rapprochent des vecteurs utilisés en mathématiques. Mais on peut aussi avoir besoin de manipuler des matrices, i.e. des structures possédant plus de deux dimensions. On utilise pour cela en C des tableaux multidimensionnels.

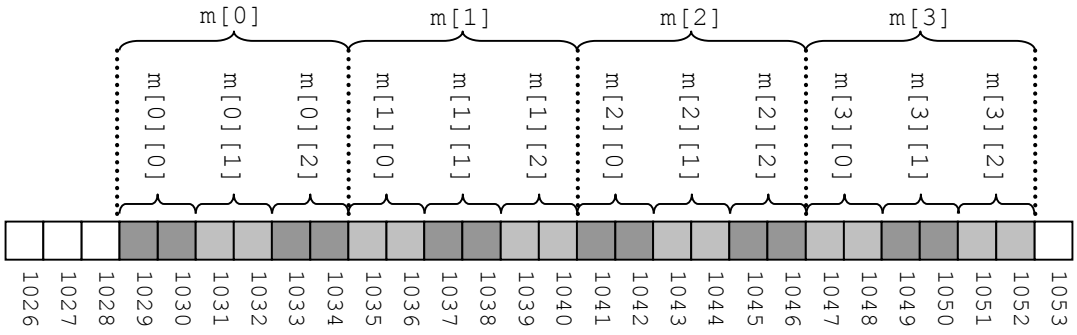
Cependant, un problème de représentation en mémoire se pose. On sait que la mémoire d'un ordinateur est *linéaire* par définition : il s'agit d'une séquence d'octets. Or, un tableau multidimensionnel est *non-linéaire*, puisqu'il possède plusieurs dimensions. La solution qui a été retenue pour le C consiste à linéariser le tableau multidimensionnel, en le représentant sous la forme d'un tableau dont chaque éléments sont eux-mêmes des tableaux : un tableau de tableaux.

Comme on l'a vu, un tableau C est une séquence d'éléments d'un certain type. Et ce type peut être *simple* (par exemple un tableau d'entiers de type `int`), mais aussi *complexe*. Un tableau est un type complexe. Il est donc bien possible de déclarer des tableaux de tableaux, ce qui permet de manipuler des tableaux multidimensionnels.

exemple : une matrice d'entiers de taille $N \times P$:

```
int m[N][P];
```

Ici, on déclare en fait un tableau de taille N, dont chaque élément correspond à un tableau de taille P. Pour $N = 4$ et $P = 3$, ce tableau serait représenté ainsi :



Si on note a l'adresse du premier élément du tableau, et s le nombre d'octets occupés par un élément du tableau (comme précédemment) alors l'adresse d'un élément `m[i][j]` est obtenue avec la formule $a + s \times i \times P + s \times j$.

L'accès à un élément est effectué comme pour un tableau unidimensionnel, à l'exception du fait qu'il faut préciser un index pour *chacune* des dimensions.

exemple : pour accéder à l'élément 1 du sous-tableau 2 :

```
m[2][1];
```

Notez que ceci est consistant avec le fait que l'opérateur `[]` est associatif de gauche à droite, comme indiqué en section 4.2. On évalue par conséquent le premier index (à gauche) avant d'évaluer le second (à droite).

Comme pour les tableaux unidimensionnels, on peut réaliser une initialisation lors de la déclaration.

exemple : pour initialiser entièrement le tableau :

```
int matrice[][P]={{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
```

Ici, on n'a pas précisé la première dimension, car c'est optionnel. Par contre, on a forcément *besoin* de la seconde pour pouvoir déterminer l'adresse d'un élément du tableau.

6.6 Exercices

1) Écrivez un programme qui initialise une matrice identité de taille $N \times N$.

```
int main(int argc, char** argv)
{  int i,j,mat[N][P];
   for(i=0;i<10;i++)
    {  for(j=0;j<10;j++)
      {  if(i==j)
               mat[i][j] = 1;
         else
            mat[i][j] = 0;
      }
   }
}
```

On peut éviter de faire 100 fois le test `if(i==j)`, ce qui donne la version alternative suivante :

```
int main(int argc, char** argv)
{  int i,j,mat[N][P];
   for(i=0;i<10;i++)
   {  for(j=0;j<10;j++)
         mat[i][j]=0;
      mat[i][i]=1;
   }
}
```

2) Écrivez un programme qui affiche une matrice d'entiers de dimensions $N \times P$. On suppose que la matrice est initialisée d'une façon ou d'une autre.

```
int main(int argc, char** argv)
{  int i,j,tab[N][P]={...};
   for(i=0;i<N;i++)
   {  for(j=0;j<P;j++)
         printf("\t%d",tab[i][j]);
      printf("\n");
   }
}
```

3) Écrivez un programme qui additionne deux matrices d'entiers de dimensions $N \times P$. Le résultat est stocké dans une troisième matrice.

```
int main(int argc, char** argv)
{  int i,j,tab1[N][P]={...},tab2[N][P]={...},tab3[N][P];
   for(i=0;i<N;i++)
      for(j=0;j<P;j++)
         tab3[i][j]=tab1[i][j]+tab2[i][j];
}
```

4) Écrivez un programme qui multiplie deux matrices d'entiers de dimensions respectives $N \times P$ et $P \times Q$. Le résultat est stocké dans une troisième matrice.

Rappel : la formule est $tab_3[i][k] = \sum_{j=0}^{P} tab_1[i][j] \times tab_2[j][k]$ avec $0 \leq i \leq N$ et $0 \leq k \leq Q$.

- **Algorithme niveau 1 :**

```
Pour tout i,k
   on calcule tab3[i,k]
```

- **Algorithme niveau 2 :**

```
Pour tout i
   Pour tout k
      on calcule tab3[i,k]
```

- **Algorithme niveau 3 :**

```
Pour tout i
   Pour tout k
      Pour tout j
         on somme les tab1[i,j]tab2[j,k]
```

- **Programme :**

```
int main(int argc, char** argv)
{  int i,j,k,tab1[N][P]={...},tab2[P][Q]={...},tab3[N][Q];
   for(i=0;i<N;i++)
      for(k=0;k<Q;k++)
      {  tab3[i][k]=0;
         for(j=0;j<P;j++)
            tab3[i][k]=tab3[i][k]+tab1[i][j]*tab2[j][k];
      }
}
```

7 Fonctions

La notion de fonction a déjà été introduite en section 1.2.2. Elle permet de décomposer un programme complexe en plusieurs parties plus simples et relativement indépendantes. Ceci a de nombreux avantages :

- Chaque fonction peut être implémentée et testée séparément ;
- Si le même traitement est nécessaire en plusieurs point du programme, on se contente d'appeler la fonction correspondante plusieurs fois au lieu de dupliquer le code source ;
- Une même fonction peut être utilisée dans plusieurs programmes différents, grâce au concept de bibliothèque (section 1.2.7).

Nous allons d'abord revenir sur la définition de ce qu'est une fonction, en profitant des concepts abordés depuis l'introduction de la notion de fonction (notamment les types) pour approfondir certains points déjà présentés superficiellement auparavant, puis nous nous concentrerons sur le passage de paramètres.

7.1 Présentation

La notion de fonction apparait dans deux contextes différents : d'abord lorsqu'on crée une fonction, puis lorsqu'on utilise une fonction déjà créée.

7.1.1 Création

On a déjà vu en section 1.2.2 qu'une fonction correspond à un bloc de code source portant un nom, et se compose des parties suivantes :

- Un *en-tête* qui comprend :
  - Le type de retour ;
  - L'identificateur ;
  - Les paramètres.

```
type nom_fonction(type paramètre1, type2 paramètre2...)
```

- Le *corps*, qui décrit le traitement effectué par la fonction, et qui est constitué d'un *bloc* de code source.

Dans le corps, l'instruction `return` permet de renvoyer la valeur d'une expression en tant que résultat de la fonction :

```
return expression;
```

Cette valeur doit posséder le type qui est précisé dans l'en-tête. Si ce n'est pas le cas, une conversion implicite (cf. section 4.3.1) peut être réalisée s'il s'agit de types simples. Par exemple, si la fonction est supposée renvoyer un `int` et qu'on applique `return` à une expression de type `float`, la valeur renvoyée sera la troncature de l'expression (sauf si un dépassement de capacité se produit).

Il est également possible de préciser qu'une fonction ne *renvoie pas* de résultat par `return`, en indiquant dans son en-tête que son type est `void`. Par défaut (i.e. si on ne précise pas de type de retour), une fonction est supposée être de type `int`.

Si la fonction se termine sans renvoyer explicitement de résultat (`return` oublié) alors que l'en-tête spécifie un type différent de `void`, une valeur *indéfinie* est renvoyée automatiquement (i.e. on ne sait pas ce que la fonction renvoie vraiment, mais elle renvoie quelque chose).

Il est donc important d'éviter de ne pas spécifier le type ou la valeur de retour, car sinon il est probable qu'une erreur de compilation se produira (cf. section 1.1.5).

Remarque : rappelons que la définition d'une fonction est placée *à l'extérieur* de toute autre fonction. Certains compilateurs ne détectent pas ce problème-là, mais il s'agit bel et bien d'une erreur qui empêchera le programme de fonctionner.

exemples :

- Une fonction qui affiche `bonjour` et passe à la ligne :

```
void bonjour(void)
{  printf("bonjour\n");
}
```

Remarquez que cette fonction n'a pas de paramètre et ne retourne aucune valeur au reste du programme.

- Une fonction qui affiche `n` étoiles `*` et passe à la ligne :

```
void etoiles (int n)
{  int i;
   for (i=1;i<=n;i++)
      printf("*\t");
   printf("\n");
}
```

Cette fonction a un seul paramètre (l'entier `n`) et ne retourne rien

- Une fonction implémentant une application linéaire (au sens mathématique) :

```
float f(float x)
{  int y = 3 * x + 1;
   return y;
}
```

Cette fonction reçoit en paramètre un réel x et retourne le réel $3x + 1$.

7.1.2 Appel

Pour utiliser une fonction, on doit effectuer un *appel* en lui passant des *expressions* correspondant aux paramètres dont elle a besoin. La syntaxe est la suivante :

```
nom_fonction(p1,p2,...);
```

exemples : appels correspondant aux fonction définies dans l'exemple précédent

- Appel de la fonction `bonjour` :

```
bonjour();
```

Il ne faut pas oublier les parenthèses, sinon cela provoque une erreur de compilation.

- L'instruction :

```
etoiles(7);
```

provoque l'affichage de 7 étoiles puis un passage à la ligne. L'argument peut aussi être une variable ou une expression qui doit être du type `int`. Par exemple si `n` est une variable de type entier, l'instruction :

```
etoiles(n%2);
```

provoquera l'affichage d'une étoile et d'un passage à la ligne si la valeur de la variable n est paire et un simple saut à la ligne si elle est impaire.

- L'instruction :

```
y = f(2.3);
```

calcule l'image de 2,3 par f et place sa valeur dans la variable `y`. L'instruction :

```
y = f(f(2.3));
```

calcule, elle, l'image de 2,3 par $f \circ f$.

Remarque : la fonction `main` est particulière, puisqu'elle est appelée *automatiquement* au début de l'exécution du programme.

L'appel de fonction est une instruction. Une instruction appartient forcément à un bloc. Un bloc appartient forcément à une fonction. Donc, quand on réalise un appel de fonction, c'est forcément à partir d'une autre fonction. Cela nous amène à distinguer deux sortes de fonctions :

Fonctions appelée : la fonction qui est explicitement nommée dans l'appel, et que l'on veut exécuter.

Fonction appelante : la fonction qui effectue l'appel, i.e. la fonction contenant l'instruction d'appel.

exemple : considérons le code source suivant

```
int f1(int x, int y)
{  ...
   return ...
}

int f2(float a)
{  int b;
   ...
   b = f1(a,99);
   ...
   return b;
}

int main(int argc, char** argv)
{  int z;
   z = f2(23);
   ...
}
```

On peut dire que :

- Pour l'instruction `b = f1(a,99)` :
  - `f2` est la fonction appelante ;
  - `f1` est la fonction appelée.
- Pour l'instruction `z = f2(23)` :
  - `main` est la fonction appelante ;
  - `f2` est la fonction appelée.

7.1.3 Paramètres

On distingue deux sortes de paramètres, en fonction de leur emplacement dans le code source :

Paramètre formel : *variable déclarée* dans l'*en-tête* de la fonction.

Notez qu'il s'agit de *variables*. Ces paramètres-là sont caractérisés par un type attendu (i.e. ces variables doivent être d'un certain type). Ce sont des variables *locales* à la fonction. Autrement dit : 1) elles ne sont accessibles que dans le bloc de la fonction, et pas à l'extérieur ; et 2) l'espace mémoire qui leur est alloué est réservé au début de l'exécution de la fonction, et libéré à la fin de son exécution. Dans l'exemple de la sous-section précédente, les paramètres formels sont représentés en rouge.

Paramètre effectif : *expression passée* à la fonction lors d'un appel.

Cette fois, il s'agit d'expressions. Autrement dit, lors d'un appel de fonction, on n'a pas forcément besoin de passer des *variables* en tant que paramètres. En effet, une variable est une expression, mais une expression n'est pas forcément une variable (cf. section 4.1).

Lors d'un appel, les contraintes suivantes doivent être respectées absolument :

1) Le nombre d'expressions doit correspondre au nombre de paramètres formels ;
2) Les types de ces expressions doivent correspondre à ceux des paramètres formels

La 1ère expression passée doit donc avoir le type du 1er paramètre formel déclaré, la 2ème doit avoir le type du 2ème paramètre formel, etc. Si les types ne correspondent pas, mais qu'il s'agit de types simples, alors une conversion implicite peut avoir lieu. Par exemple, si la fonction attend un entier et qu'on lui passe un réel, l'entier sera tronqué.

Remarque : ce point est particulièrement important. Quand vous appelez une fonction, vérifiez bien que les types des paramètres formels et effectifs correspondent. De plus, le type du paramètre formel peut vous aider à déterminer quel calcul effectuer pour obtenir le paramètre effectif. C'est particulièrement vrai lorsque des pointeurs sont manipulés (cf. section 10).

7.1.4 Résultat

Si la fonction renvoie une valeur, c'est-à-dire si son type de retour n'est pas `void`, alors cela signifie qu'elle a été conçue pour calculer un résultat et le rendre accessible à la fonction qui a réalisé l'appel. La plupart du temps, cela veut dire que la fonction appelante veut utiliser ce résultat. Il faut donc en tenir compte lors de l'appel.

exemple : supposons qu'on a une fonction qui double la valeur de l'entier qu'elle reçoit en paramètre :

```
int double(int a)
{  int resultat = a * 2;
   return resultat;
}
```

Supposons qu'on veut utiliser cette fonction dans la fonction `main`, par exemple pour doubler la valeur 14 et afficher le résultat. Alors *on ne peut pas* effectuer l'appel suivant :

```
double(14)
```

En effet, avec cet appel, la fonction `double` va bien calculer le double de 14 et le renvoyer à la fonction `main`, mais ce résultat sera perdu car il n'est pas utilisé. Pour être utilisé, ce résultat doit apparaître dans une *expression*. On peut par exemple le faire apparaître dans un `printf`, puisqu'on voulait initialement afficher le double de 14 :

```
int main(int argc, char** argv)
{  printf("Le résultat est : %d\n",double(14));
}
```

Cependant, cette approche peut poser problème si on veut utiliser le résultat plusieurs fois. Par exemple, supposons qu'on veuille afficher une fois le résultat, puis une fois le résultat plus 1 :

```
int main(int argc, char** argv)
{  printf("Le résultat est : %d\n",double(14));
   printf("Le résultat plus 1 est : %d\n",double(14)+1);
}
```

Alors on appelle *deux fois* la même fonction avec le même paramètre, et on réalise deux fois *exactement le même calcul*, ce qui est peu efficace. Évidemment, dans cet exemple le calcul est extrêmement rapide, en pratique. Mais imaginez qu'il s'agisse d'une fonction prenant 1 heure à s'exécuter : notre programme durera alors 2 heures au lieu d'une seule !

Une méthode plus efficace consiste à stocker le résultat dans une variable et à utiliser cette variable plusieurs fois. Non seulement c'est plus efficace en termes de temps de calcul, mais aussi et surtout en termes de lisibilité :

```
int main(int argc, char** argv)
{  int res;
   res = double(14);
   printf("Le résultat est : %d\n",res);
   printf("Le résultat plus 1 est : %d\n",res+1);
}
```

On sait que certaines fonctions ont `void` pour type de retour, ce qui signifie qu'elles ne renvoient aucune valeur. On peut alors se demander à quoi ces fonctions peuvent bien servir,

puisque la fonction appelante ne reçoit rien du tout à utiliser après l'appel. Mais ces fonctions profitent en réalité du concept d'*effet de bord* :

Fonction à effet de bord : fonction dont l'exécution a une conséquence *hors* de la fonction elle-même et de sa valeur de retour.

Une fonction à effet de bord va réaliser un traitement qui affectera par exemple une variable globale (pas passée en paramètre, pas renvoyée en résultat) ou un périphérique. L'exemple le plus classique est la fonction `printf`, dont l'effet de bord est d'afficher quelque chose à l'écran.

Remarque : une fonction peut *à la fois* renvoyer un résultat (i.e. son type de retour n'est pas `void`) *et* avoir simultanément un effet de bord. Par contre, une fonction qui ne renvoie rien du tout ni n'a aucun effet de bord ne sert pas à grand-chose.

7.1.5 Emplacement

Une fonction doit être *connue* (mais pas forcément *complètement* définie) du compilateur avant sa première utilisation dans le programme. Cela signifie que sa *déclaration* doit précéder son *appel*, mais que sa *définition* peut lui succéder.

Déclaration d'une fonction : action d'informer le compilateur qu'une fonction existe, en indiquant son en-tête suivi d'un point-virgule (i.e. `;`).

exemple : déclaration d'une fonction `multiplication` :

```
int multiplication(int x, int y);
```

Remarque : lorsqu'on déclare une fonction, le C n'oblige pas à préciser les noms des paramètres. Cependant, il est recommandé de le faire, car cela améliore la lisibilité du programme. Pour cette raison, dans le cadre de ce cours, il est obligatoire de nommer les paramètres dans les déclarations de fonctions.

Définition d'une fonction : action d'associer un en-tête à un corps de fonction. La fonction peut avoir été déclarée avant, mais ce n'est pas obligatoire.

Bien entendu, si la fonction a été déclarée avant, l'en-tête de sa définition doit être équivalent à celui de sa déclaration. Cela veut dire que le type de retour, le nom de la fonction, le nombre de paramètres et leurs types doivent être exactement les mêmes. Seul les noms des paramètres peuvent être différent (mais ce n'est pas recommandé, pour des raisons de lisibilité).

exemple : définition de la même fonction `multiplication` :

```
int multiplication(int x, int y)
{  int resultat;
   resultat = x * y;
   return resultat;
}
```

On a vu précédemment, lorsqu'on a étudié la notion de bibliothèque (section 1.2.7) qu'il était possible de placer la déclaration d'une fonction dans un fichier d'en-tête (dont le nom finit par `.h`) et sa définition dans un autre fichier (dont le nom finit par `.c`). Mais il est aussi possible d'inclure les deux dans le même fichier. Ceci n'est pas recommandé lorsqu'on crée une bibliothèque, mais c'est la marche à suivre quand on travaille sans bibliothèque, par exemple quand on écrit une fonction dans le fichier principal `main.c`.

Il y a alors deux possibilités pour placer la déclaration et la définition de la fonction dans le programme :

- Soit on ne déclare pas la fonction, et on la définit directement.
- Soit on déclare d'abord la fonction, puis on la définit plus tard.

La première méthode ne peut être utilisée que si la fonction n'a pas été appelée avant sa définition. Considérons l'exemple suivant :

```
int multiplication(int x, int y)
{  int resultat;
   resultat = x * y;
   return resultat;
}

int main(int argc, char** argv)
{  int r = multiplication(12,13);
   ...
}
```

Ce programme est correct, car on a bien défini la fonction `multiplication` *avant* de l'appeler dans la fonction `main`. Par contre, soit le programme suivant :

```
int main(int argc, char** argv)
{  int r = multiplication(12,13);
   ...
}

int multiplication(int x, int y)
{  int resultat;
   resultat = x * y;
   return resultat;
}
```

Le compilateur va indiquer une erreur dans la fonction `main`, car on veut utiliser une fonction `multiplication` qui n'a jamais été ni déclarée ni définie. Il faut alors utiliser la seconde méthode, c'est-à-dire déclarer la fonction avant l'appel, puis la définir :

```
int multiplication(int x, int y);

int main(int argc, char** argv)
{  int r = multiplication(12,13);
   ...
}

int multiplication(int x, int y)
{  int resultat;
   resultat = x * y;
   return resultat;
}
```

7.2 Passage des paramètres

En langage C, il est possible de passer les paramètres d'une fonction de deux manières différentes : par valeur ou par adresse. Bien qu'elles n'aient pas été expliquées jusqu'à présent, ces deux méthodes ont été utilisées en TP, notamment lors de l'affichage (`printf`) et de la saisie (`scanf`) de données.

7.2.1 Passage par valeur

Définissons d'abord l'exemple suivant, qui sera utilisé dans le reste de cette section. Soit la fonction `carre`, qui calcule le carré d'un entier et renvoie le résultat :

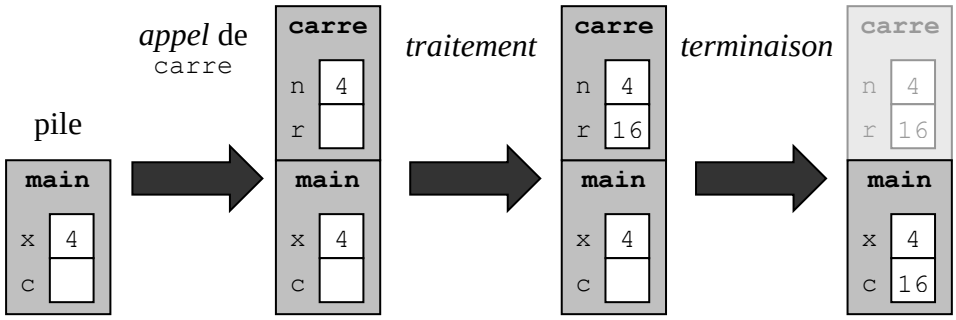
```
int carre(int n)
{  int r = n * n;
   return r;
}
```

Supposons qu'on appelle cette fonction à partir de la fonction `main`, en utilisant une variable `x`, et que le résultat de la fonction `carre` est stocké dans une variable `c` pour être utilisé plus tard :

```
int main(int argc, char** argv)
{  int x=4,c;
   c = carre(x);
   ...
}
```

Alors ici, `n` est le paramètre formel et `x` est le paramètre effectif.

Quand une fonction est appelée, ses paramètres et les variables déclarées dans son corps sont définis comme des variables locales à la fonction. Or, comme nous l'avons déjà vu, les variables locales à une fonction sont stockées dans une partie de la mémoire appelée la pile (cf. section 3.2.5). De plus, on sait aussi que la partie de la pile allouée à une fonction est désallouée quand la fonction se termine (comme illustré ci-dessous).

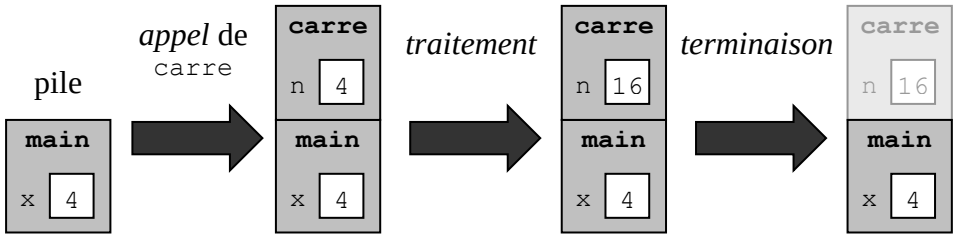


Lors d'un appel de fonction, les paramètres formels sont initialisés grâce aux valeurs correspondant aux paramètres effectifs. En effet, comme on l'a déjà indiqué, un paramètre effectif n'est pas forcément une variable, mais peut être plus généralement une expression (par exemple une constante, ou bien le résultat d'une opération, etc.). Puisque les variables locales sont stockées dans une partie de la pile qui est elle-même désallouée à la fin de la fonction, cela signifie donc que les paramètres formels disparaissent eux-aussi quand la fonction se termine. Par conséquent, toute modification d'un paramètre formel est locale à la fonction. Autrement dit, quand on change la valeur d'un paramètre, cette modification n'est pas répercutée dans la fonction appelante.

exemple : tentons de modifier les fonctions précédentes de manière à ce que `carre` modifie directement le paramètre qui lui est passé :

```
void carre(int n)
{  n = n * n;
}

int main(int argc, char** argv)
{  int x = 4;
   carre(x);
   ...
}
```

Alors la fonction `carre` ne va *pas* modifier `x`, mais seulement la copie de `x` à laquelle elle a accès, c'est-à-dire la variable `n` :



Cela signifie qu'en utilisant cette méthode, la seule façon de renvoyer un résultat est d'utiliser `return`. Or, `return` ne peut renvoyer qu'une seule valeur. Donc, avec cette méthode, on ne peut renvoyer qu'une seule valeur à la fois.

Ceci peut être considéré comme une limitation forte. Par exemple, supposons qu'on veuille écrire une fonction calculant à la fois le quotient et le reste de la division entière : on veut alors

renvoyer deux valeurs. Cependant, il est impossible d'atteindre cet objectif avec le passage de paramètre par valeur.

7.2.2 Passage par adresse

Pour résoudre le problème identifié précédemment, i.e. l'impossibilité de renvoyer plusieurs valeurs en utilisant `return`, on utilise l'*effet de bord*. On manipule pour cela non pas la valeur des paramètres, mais leur *adresse*, d'où le terme de *passage par adresse*. Le principe est le suivant : puisqu'on ne peut pas utiliser `return` pour renvoyer plusieurs valeurs, on va directement modifier les valeurs des paramètres. Ainsi, si on veut renvoyer deux valeurs à la fois, il suffira de modifier les valeurs de deux paramètres. Si on en veut trois, il faudra utiliser trois paramètres, et ainsi de suite.

Cependant, comme on l'a vu précédemment, il est inutile de modifier la valeur d'un paramètre formel : cela n'a aucun effet sur le paramètre effectif, car le paramètre formel n'est qu'une copie locale et temporaire. La solution consiste à travailler directement sur le paramètre effectif. Mais pour cela, on a besoin de savoir où il se trouve dans la mémoire : ceci est indiqué par son adresse. Donc, si on passe l'adresse du paramètre effectif au lieu de sa valeur, il sera possible de renvoyer (indirectement, i.e. sans passer par `return`) plusieurs données *simultanément*.

À titre d'exemple, nous allons reprendre notre fonction `carre`, et la modifier de manière passer son résultat par adresse au lieu d'utiliser `return`. Il est nécessaire de modifier la fonction `carre` en rajoutant un nouveau paramètre qui contiendra le résultat : appelons-le `res`. Le type de ce paramètre ne peut pas être `int`, car on veut que la fonction reçoive non pas un entier `int`, mais *l'adresse d'un entier* `int` en paramètre. Il est important de distinguer ces deux informations. Formellement, cela se fait au moyen des types : les adresses sont représentées en C au moyen de types spécifiques. Le type associé à l'adresse d'une donnée d'un certain type `xxxxx` est noté en ajoutant un astérisque (i.e. `*`) après le nom du type : `xxxxx*`. Dans notre cas, le paramètre reçu sera donc de type `int*`. De plus, comme la fonction ne renverra plus rien par `return`, sont type de retour devient `void`. Au final, l'en-tête de la fonction sera le suivant ;

```
void carre(int n, int* res)
```

Remarque : ici, `*` ne représente pas l'opérateur de multiplication. Il a une sémantique spécifique aux types, qui sera approfondie plus tard dans le cours.

Dans le corps de la fonction, on veut affecter à `res` la valeur de l'expression `n*n`. On pourrait être tenté de le faire de la façon suivante :

```
res = n * n;
```

Cependant, ceci est faux pour plusieurs raisons. Tout d'abord, les types de l'affectation ne sont pas compatibles : la left-value `res` est de type `int*` et correspond à une adresse, alors que la right-value `n*n` est de type `int` et correspond à une valeur entière. De plus, on sait qu'il est inutile de modifier un paramètre dans une fonction, car cette modification sera locale à la fonction, et n'aura pas d'effet en dehors, en particulier dans la fonction appelante.

Rappelons-nous que le but du passage par adresse est justement d'éviter cela ! On veut travailler directement sur la variable dont l'adresse (i.e. l'emplacement) a été passée en paramètre. On a pour cela besoin de l'opérateur d'indirection, noté `*`. Là encore, il ne s'agit pas de l'opérateur de multiplication. Celui-ci réalise l'opération inverse de l'opérateur `&`. Autrement dit, quand on l'applique à une adresse, cet opérateur permet d'accéder au contenu situé à cette adresse. Par exemple, si on applique `*` à une adresse de type `int*`, alors on accède à une valeur

de type `int`. Remarquez comme le type de l'adresse permet à cet opérateur de déterminer le type de la donnée située à cette adresse.

Dans notre exemple, on ne veut pas modifier `res`, mais la valeur située à l'adresse `res`. Pour accéder à cette valeur, on utilisera donc `*res` de la façon suivante :

```
void carre(int n, int* r)
{   *r = n * n;
}
```

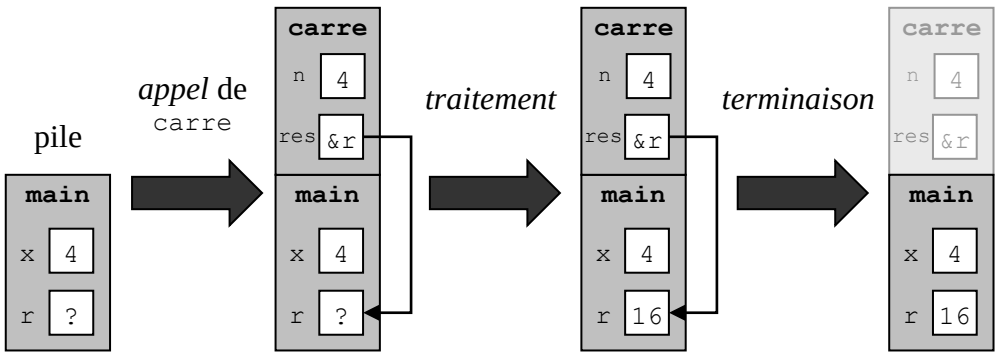
Notez que cette fois, il n'y a pas d'incompatibilité de types pour l'opérateur d'affectation : la left-value `*res` et la right-value `n*n` sont toutes les deux de type `int`. De plus, remarquez aussi l'absence de `return` à la fin de la fonction : ceci est consistant avec son type de retour `void`, et avec le fait qu'on renvoie le résultat par effet de bord, en modifiant le paramètre `res`.

Intéressons-nous maintenant à la fonction `main`, qui est la fonction appelante dans notre exemple. Notre fonction `carre` modifiée attend de recevoir deux paramètres : un entier et l'adresse d'un entier. Son appel dans `main` doit donc être modifié de façon appropriée. On a besoin d'une variable supplémentaire pour le deuxième paramètre, notons la `r`. Le premier paramètre reste une valeur, donc il sera passé comme précédemment. Le second est une adresse. On sait qu'on obtient l'adresse d'une variable `variable` en utilisant l'opérateur `&` de la façon suivante : `&variable` (cf. section 4.2.6). L'adresse de `r` s'obtient donc par l'expression `&r`, et on a la fonction `main` suivante :

```
int main(int argc, char** argv)
{   int x = 4;
    int r;
    carre(x,&r);
}
```

La figure ci-dessous résume la manipulation effectuée en mémoire. On a initialement deux variables de type `int` dans la fonction `main` : `x` et `r`. La première est initialisée à 4, la seconde n'est pas initialisée (on ne connait donc pas sa valeur).

Lorsqu'on appelle `carre`, le paramètre formel `n` est une copie de `x`, tandis que le paramètre `res` est une copie de l'adresse de `r`. Lorsqu'on modifie `*res` dans `carre`, on modifie en réalité `r` dans `main`. Puis la fonction `carre` se termine, les variables `n` et `res` sont désallouées, mais la valeur 16 qui a été stockée dans `r` n'est pas affectée.



On a vu que le passage par adresse est nécessaire quand on a besoin de modifier un paramètre, et/ou quand on a besoin de renvoyer plusieurs résultats à la fois. Mais il existe un autre cas où, s'il n'est pas nécessaire, le passage par adresse est quand même plus efficace que le passage par valeur. Supposons qu'un ou plusieurs paramètres occupe une grande place en mémoire. Par exemple, une image de grande taille (avec la SDL, il s'agirait d'une variable de type `SDL_Surface`). Un passage par valeur nécessite de recopier l'image dans une nouvelle

variable, ce qui demande à la fois de la mémoire et du temps de calcul. Le passage par adresse permet d'éviter cela.

7.3 Tableaux

Les tableaux constituent un type complexe (cf. section 6), et sont donc traités différemment des types simples lors du passage de paramètres.

7.3.1 Tableaux unidimensionnels

Le passage par valeur consiste à créer une variable locale à la fonction, appelée paramètre formel, et à l'initialiser en utilisant la valeur passée lors de l'appel de la fonction, qui est appelée paramètre effectif. Si cette méthode ne pose pas de problème pour une variable de type simple, en revanche elle peut être critiquée quand il s'agit d'un tableau. En effet, il faut alors créer un nouveau tableau, aussi grand que le tableau existant, ce qui occupe de la mémoire, puis copier chacun des éléments du tableau original dans le tableau copie, ce qui peut prendre du temps.

Pour éviter cela, en langage C les tableaux sont systématiquement passés par adresse. Cela signifie que quand on déclare un paramètre de type tableau dans l'en-tête d'une fonction, on ne va pas créer un nouveau tableau, mais plutôt créer une variable qui est connectée à un tableau existant. Pour cela, on va utiliser l'adresse du tableau, i.e. l'adresse de son premier élément (tous les éléments étant stockés de façon contigüe, on n'a besoin que de l'adresse du premier d'entre eux pour localiser le tableau en mémoire). Comme on l'a déjà vu (section 6), l'identificateur d'un tableau correspond à cette adresse. Donc, à la différence des variables de types simples, le fait de manipuler un tableau via son adresse n'a aucun effet sur la syntaxe de la fonction. Alors qu'on devait manipuler des variables de types simples en utilisant les opérateurs `&` et `*`, ce n'est pas nécessaire pour les tableaux car on les manipule déjà habituellement sous forme d'adresses.

Par exemple, supposons que l'on veut écrire une fonction `init` qui initialise un tableau de taille *N* avec la valeur zéro. La fonction peut prendre la forme suivante :

```
void init(int tableau[])
{  short i;
   for(i=0;i<N;i++)
      tableau[i] = 0;
}
```

Remarquez que le tableau est utilisé normalement (pas de `&` ou `*`). Notez aussi qu'on n'a pas précisé la taille du tableau (qui est *N*) dans la déclaration du paramètre. En effet, comme indiqué précédemment, il ne s'agit pas ici de créer un nouveau tableau, mais de faire référence à un tableau existant. Or, le compilateur n'a besoin de la taille d'un tableau que lors de sa création, pour savoir combien d'octets doivent être réservés pour stocker les éléments du tableau.

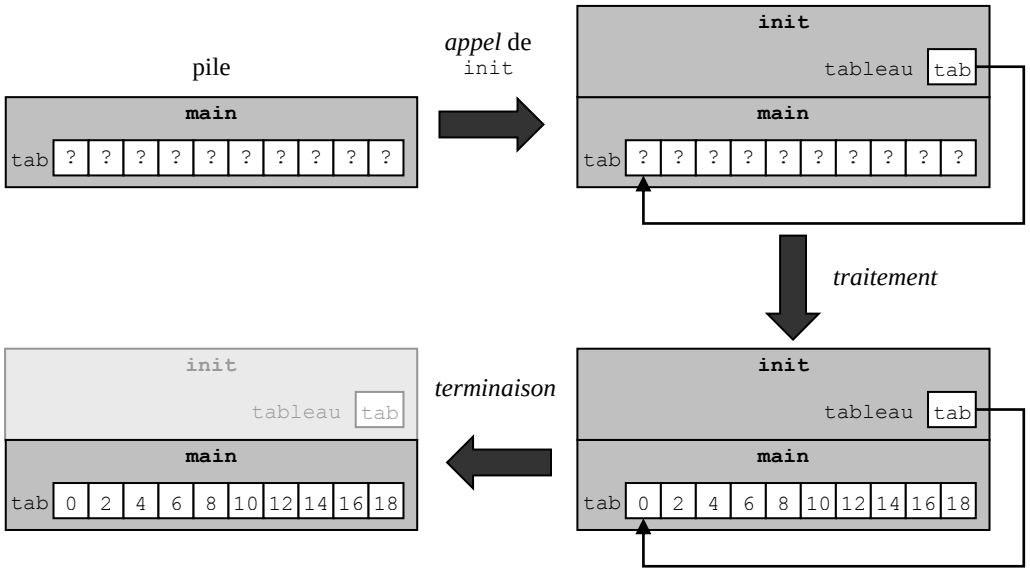
Remarque : il est possible, optionnellement, d'indiquer la taille du tableau. Nous recommandons de le faire chaque fois que c'est possible (ce qui n'est pas toujours le cas), car cela permet d'améliorer la lisibilité du code source.

L'appel de la fonction `init` se fait de la façon suivante :

```
int main(int argc, char** argv)
{  int tab[N];
   init(tab);
}
```

La déclaration du tableau se fait normalement. On passe en paramètre à la fonction `init` l'expression `tab`, dont le type est `int[]`. Ce type est bien compatible avec celui du paramètre formel de la fonction `init`, qui est lui aussi `int[]`. La fonction `init` accède ensuite

directement aux valeurs contenues dans le tableau `tab` qui existe dans la fonction `main`. La figure ci-dessous illustre le traitement réalisé en mémoire.



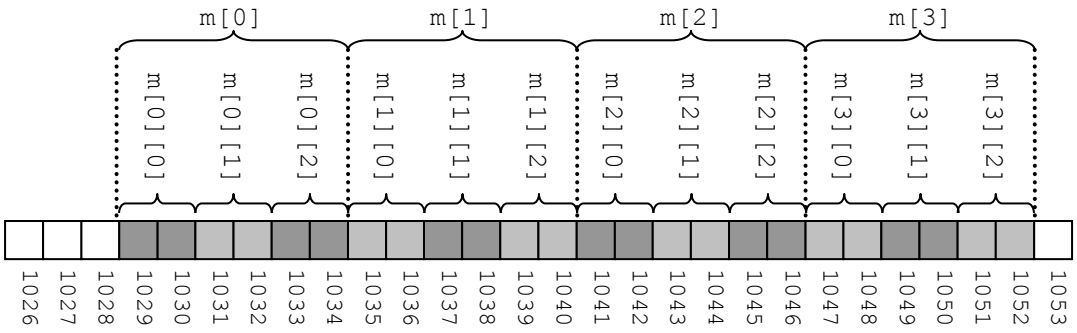
Remarque : le compilateur n'effectue aucun contrôle sur la taille du tableau passé en paramètre. Autrement dit, si on effectue l'appel à `init` avec un tableau dont la taille n'est pas N, cela ne produira aucune erreur de compilation ni aucun avertissement. Si une erreur se produit (ce qui est probable), elle aura lieu à l'exécution, quand `init` modifiera un octet qu'elle n'est pas supposée modifier car il ne se situe pas dans le tableau reçu.

7.3.2 Tableaux multidimensionnels

Le principe est le même pour les tableaux multidimensionnels passés en paramètres : on utilise là-aussi le passage paramètre. Cependant, si la taille de la première dimension du tableau reste optionnelle, il est par contre nécessaire de spécifier les tailles des dimensions suivantes. En effet, le compilateur a *besoin* de connaître ces dimensions pour pouvoir faire correspondre un élément du tableau à une adresse en mémoire.

exemple : passage d'un tableau d'entiers de dimensions 4×3

```
void fonction(int tab[][3])
{  ...
}
```



On a déjà vu (cf. section 6.5) que l'adresse d'un élément `m[i][j]` dans un tableau de dimensions $N \times P$ est calculée avec la formule $a + s \times i \times P + s \times j$, où a est l'adresse du premier élément du tableau et s le nombre d'octets occupés par un élément du tableau. On voit bien que N n'est pas nécessaire à ce calcul, à la différence de P.

7.4 Exercices

1) Écrivez une fonction `int puissance(int x, int y)` qui prend deux paramètres entiers x et y et renvoie pour résultat x^y. Donnez également un exemple d'appel de cette fonction en écrivant une fonctin `main` dans laquelle vous affichez le résultat du calcul.

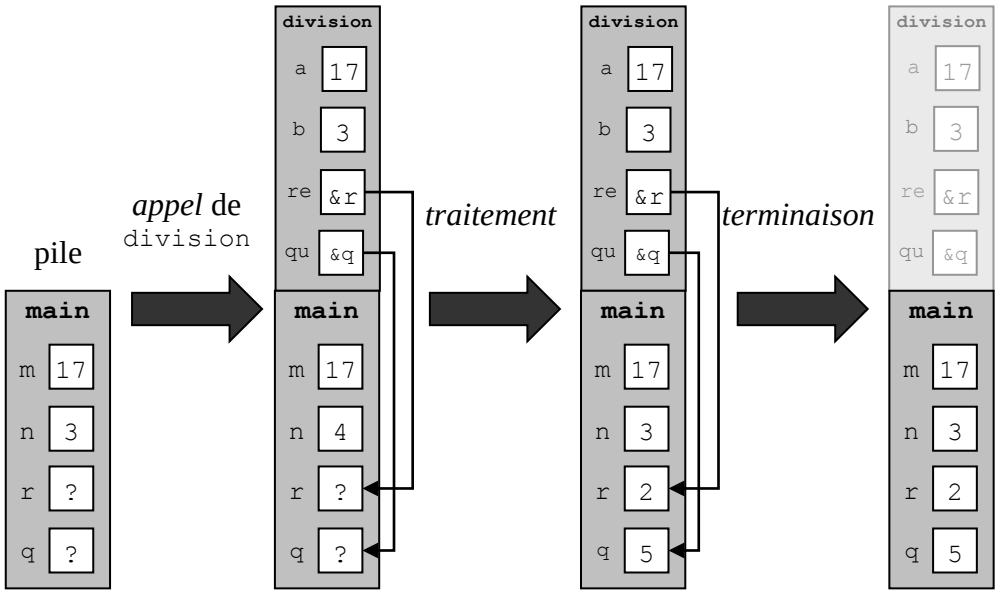
```
int puissance(int x, int y)
{  int resultat=1,i;
   for(i=0;i<y;i++)
      resultat = resultat*x;
   return resultat;
}

int main(int argc, char** argv)
{  int x=4,y=5,r;
   r=puissance(x,y);
   printf("%d puissance %d = %d\n",x,y,r);
}
```

Même question, mais cette fois on veut passer le résultat par adresse. L'en-tête de la fonction devient donc `void puissance(int x, int y, int *r)`.

```
void puissance(int x, int y, int *r)
{  int i;
   r=1;
   for(i=0;i<y;i++)
      *r = *r*x;
}

int main(int argc, char** argv)
{  int x=4,y=5,r;
   puissance(x,y,&r);
   printf("%d puissance %d = %d\n",x,y,r);
}
```

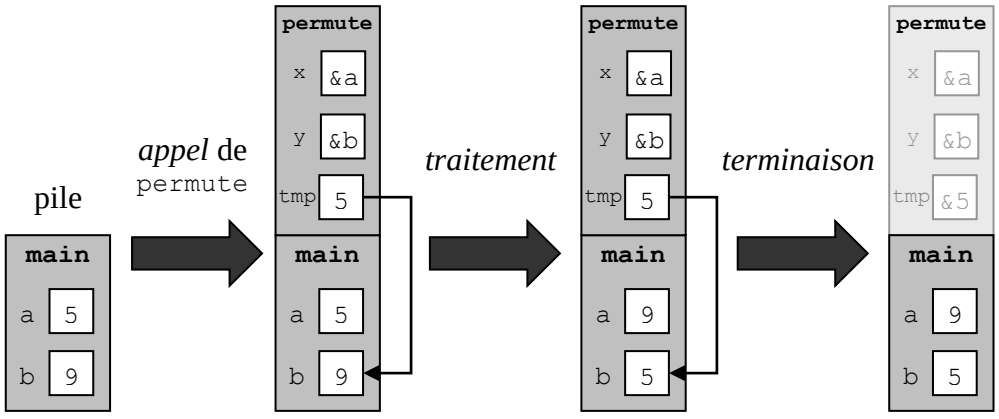
2) Écrivez une fonction `divise` qui effectue une division entière et renvoie à la fois le quotient et le reste par adresse, et la fonction `main` qui l'appelle. Donnez la représentation graphique de la mémoire au cours de l'appel correspondant à la division de 17 par 3.

```
void division(int a, int b, int* q, int* r)
{  *r = a % b;
   *q = a / b;
}

int main(int argc, char** argv)
{  int a=17, b=3, q, r;

   division(a,b,&q,&r);
   printf("%d=%d*%d+%d\n",a,b,q,r);
}
```



3) Écrivez une fonction `permute` qui permute les valeurs de deux variables (i.e. qui les échange), et la fonction `main` qui l'appelle. Donnez la représentation graphique de l'évolution de la mémoire au cours de l'appel pour 5 et 9.

```
void permute(int *x, int *y)
{   int temp = *x;
    *x = *y;
    *y = temp;
}

int main(int argc, char** argv)
{   int a=5, b=9;
    print("Avant : a=%d et b=%d\n",a,b);
    permute(&a,&b);
    print("Après : a=%d et b=%d\n",a,b);
}
```



8 Types personnalisés

Le langage C permet au programmeur de définir ses propres types afin de pouvoir manipuler des valeurs adaptées au problème à résoudre. Ceci concerne aussi bien les types complexes que les types simples.

Type personnalisé : type défini par le programmeur, par opposition aux types *par défaut*, qui font partie du standard du langage C (`int`, `char`, etc.).

Dans cette section, nous aborderons les *structures*, qui sont des types complexes hétérogènes ; les unions, qui sont des types simples hétérogènes ; et les énumérations, qui sont des types simples homogènes. Nous verrons également comment associer un nouvel identificateur à un type personnalisé.

8.1 Généralités

La déclaration d'un type personnalisé doit forcément précéder la déclaration de variables, paramètres ou fonctions utilisant ce type. Si le programme ne contient qu'un seul fichier `main.c`, on place la déclaration du type au début du programme, juste après les directives de précompilation comme `#include` ou `#define` :

```
#include <stdio.h>

#define N 99

struct mon_type
{ // définition d'un type ici
   ...
}

// reste du programme : fonctions, main, etc.
```

Si jamais le programme contient des bibliothèques, alors les types doivent être déclarés au début du fichier d'en-tête (celui qui finit par `.h`). On les place, là aussi, après les directives de précompilation. Le type pourra ensuite être utilisé dans le fichier `.c` associé au `.h`, et dans tous les fichiers qui incluent directement ou indirectement ce `.h`.

Par exemple, supposons qu'on veuille définir une bibliothèque utilisant le type date donné en exemple précédemment. Alors on fera :

- Fichier `date.h` : (cf. la section 1.2.7 pour ce qui concerne les bibliothèques)

```
#ifndef DATE_H
#define DATE_H

struct date
{  int jour;
   int mois;
   int annee;
};

void affiche_date(struct date d);
void saisir_date(struct date* d);
...
#endif
```

- Fichier `date.c` :

```
#include "date.h"

void affiche_date(struct date d)
{  ...
}

void saisir_date(struct date* d)
```

```
{   ...
}
...
```

8.2 Structures

8.2.1 Présentation

On a précédemment étudié les tableaux, qui sont des types permettant de regrouper sous un même identificateur plusieurs données de même type. Un type tableau est un type complexe homogène. Par comparaison, une structure est un type complexe *hétérogène* :

Type complexe hétérogène : type de données décrivant une information composite, constituée de plusieurs valeurs qui peuvent être elles-mêmes de *types différents* (qui peuvent être eux-mêmes simples ou complexes).

Autrement dit, une structure permet de regrouper sous un même identificateur plusieurs données qui peuvent être complètement différentes. Ces données sont appelées les *champs* de la structure.

Champ de structure : élément constituant la structure, caractérisé par un identificateur et un type.

Si on considère une structure comme un *groupe de variables*, alors un champ peut être comparé à *une variable*. En effet, comme une variable, un champ possède un type, un identificateur, une valeur et une adresse. Cependant, à la différence d'une variable, un champ n'est pas indépendant, car il appartient à un groupe de champs qui forme sa structure.

Le fait d'associer un identificateur à un champ est important, car cela signifie que chaque élément d'une structure possède un nom spécifique. Ce nom est unique dans la structure, et permet d'identifier le champ sans ambiguïté. Par opposition, les éléments qui constituent un tableau ne possèdent pas de nom : ils sont simplement *numérotés*, grâce à leur index qui indique leur position dans le tableau.

Un champ peut être de n'importe quel type, qu'il soit simple ou complexe. Il peut donc s'agit d'un tableau, ou même d'une structure.

8.2.2 Déclaration

La déclaration d'une structure se fait grâce au mot-clé `struct`, et en précisant :

- Son nom ;
- La liste de ses champs, avec pour chacun son nom et son type.

exemple : on veut représenter des personnes par leurs noms, prénoms et âge.

```
struct personne
{   char nom[10];
    char prenom[10];
    short age;
};
```

On crée une structure appelée `personne`, contenant une chaîne de caractères `nom` pour représenter le nom de famille, une autre `prenom` pour le prénom, et un entier `age` pour l'âge.

Notez bien les points-virgules qui terminent la déclaration de chaque champ (en rouge), et le point-virgule final qui conclut la déclaration de la structure (en orange).

La déclaration d'une variable structure se fait de façon classique, i.e. en précisant son type puis son identificateur :

```
struct personne x;
```

On déclare une variable appelée `x` et dont le type est la structure `personne` définie précédemment.

Notez qu'ici, le mot-clé `struct` fait partie du nom du type et doit toujours être répété. Nous verrons plus tard comment éviter cela, ce qui rendra le code source plus compact et lisible.

Il n'est pas obligatoire de donner un nom à la structure : on parle alors de *structure anonyme*.

Structure anonyme : structure à laquelle on n'a pas associé d'identificateur.

exemple : on peut déclarer la structure précédente de la façon suivante :

```
struct
{  char nom[10];
   char prenom[10];
   short age;
};
```

Mais puisque cette structure n'a pas de nom, on ne peut pas y faire référence par la suite, par exemple pour déclarer une variable. Nous verrons plus tard que cette syntaxe n'est utilisée, en pratique, que dans un cas bien précis.

Quelques autres exemples de structures :

- Représentation d'une date :

```
struct date
{  int jour;
   int mois;
   int annee;
};
```

- Utilisation d'une structure dans une autre structure :

```
struct identite
{  char nom[N_MAX];
   char prenom[N_MAX];
   struct date date_naissance;
};
```

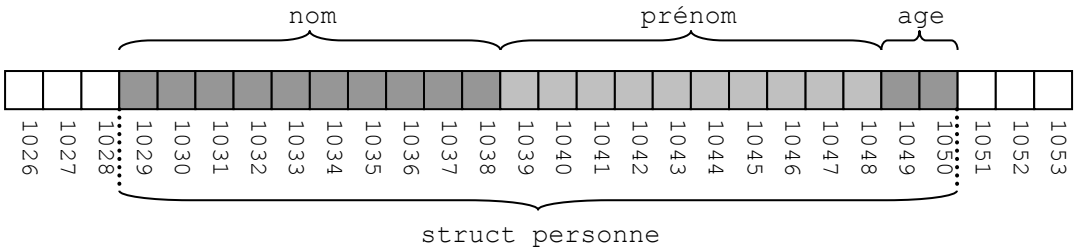
- Représentation d'une matrice avec ses dimensions :

```
struct matrice
{  double mat[TAILLE_MAX][TAILLE_MAX];
   int nb_lignes;
   int nb_colonnes;
};
```

8.2.3 Mémoire

En mémoire, les structures présentent également des similitudes avec les tableaux. Les champs qui composent une structure sont en effet placés de façon *contigüe* en mémoire, exactement comme les éléments formant un tableau.

exemple : pour la structure de l'exemple précédent, on a :



La différence avec les tableaux est qu'ici, les éléments sont hétérogènes, dans le sens où ils peuvent avoir des types différents. Donc, l'espace occupé par chaque élément (champ) n'est pas forcément le même. Dans l'exemple ci-dessus, le nom et le prénom occupent ainsi chacun 10 octets car il s'agit de tableaux de 10 caractères, mais l'âge n'en occupe que 2, car c'est un entier `short`.

8.2.4 Manipulation

Comme pour les tableaux, l'accès au contenu d'une variable structure se fait un élément à la fois. En l'occurrence, cela se fait donc *champ par champ*. Cependant, alors que pour les tableaux on utilise l'opérateur `[]` et l'index d'un élément, en revanche pour les structures on utilise l'opérateur `.` (un simple point) et le *nom* du champ. La syntaxe est la suivante :

```
variable.champ;
```

Cette expression se manipule comme une variable classique.

exemple : modifier l'âge d'une personne `x` :

```
x.age = 25;
```

Au niveau des types : l'expression prend le type du champ concerné. Donc ici, la variable `x` est de type `struct personne`, mais `x.age` est de type `short`.

Comme pour les tableaux, l'accès aux éléments d'une structure se fait champ par champ, et non pas globalement. Ainsi, sion veut comparer deux variables de type structure afin de savoir si elles sont égales, alors cette comparaison doit être faite pour chaque champ séparément. Par exemple, si on veut savoir si les variables `x` et `y` de type `struct personne` représentent la même personne, il faut comparer `x.nom` et `y.nom`, puis `x.prenom` et `y.prenom`, et enfin `x.age` et `y.age`.

Pour effectuer l'opération décrite ci-dessus, certains compilateurs acceptent l'expression `x=y`. Cependant, la copie qui est alors réalisée est superficielle. Ceci peut poser problème quand la structure contient des adresses (pointeurs, section 10).

À noter qu'il n'est pas possible de faire de conversion explicite (cf. section 4.3.2) sur une structure.

L'initialisation d'une structure peut être effectuée à la *déclaration*, comme pour les tableaux :

```
struct personne z = {"vincent", "labatut", 37};
```

Les valeurs des champs doivent être indiquées dans l'ordre de leur déclaration dans le type structure. Bien sûr, les types doivent correspondre.

8.2.5 Tableaux

On a déjà vu qu'un tableau pouvait être défini sur n'importe quel type, qu'il soit simple ou complexe. Par conséquent, il est possible de manipuler des *tableaux de structures*. La déclaration se fait comme pour un tableau classique, à l'exception du fait qu'on précise un type structure.

exemple : on déclare un tableau pouvant contenir 10 valeurs de type `struct personne` :

```
struct personne tab[10];
```

Pour accéder à l'élément *k* du tableau on utilise l'expression `tab[k]`, dont le type est `struct personne`. Pour accéder au champ `age` de l'élément *k* du tableau, on utilise l'expression `tab[k].age`, qui est de type `short`. Pour accéder à la première lettre du champ `prenom` de l'élément *k* du tableau, on utilise l'expression `tab[n].prenom[0]`, qui est de type `char`.

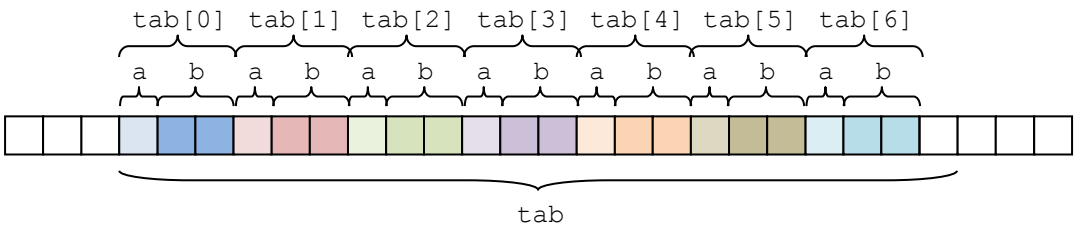
exemple : considérons la structure suivante :

```
struct ma_struct
{  char a;
   short b;
};
```

Et déclarons le tableau `tab` suivant, contenant 7 valeurs de ce type :

```
struct ma_struct tab[7]
```

Alors la représentation de la mémoire est la suivante :



8.2.6 Paramètres

Une variable de type structure peut bien entendu être utilisée comme paramètre d'une fonction. Le passage se déroule alors comme pour une variable de type simple : on doit choisir entre un passage par valeur ou par adresse.

Remarque : il est important de bien remarquer que sur ce point, la structure diffère du tableau, puisque le tableau est automatiquement passé par adresse (cf. section 7.3).

Supposons qu'on a une fonction recevant en paramètre l'adresse d'une variable de type `struct personne` :

```
void ma_fonction(struct personne* x);
```

Alors la modification d'un de ses champs passe par l'utilisation de l'opérateur `*`, comme on l'a déjà vu dans la section 7.2.2. Par exemple, si on veut changer l'âge de la personne `x`, on utilisera l'expression :

```
(*x).age = 99;
```

Notez bien l'utilisation des parenthèses. En effet, on veut appliquer l'opérateur `*` seulement à l'expression `x`, qui est de type `struct personne*`. On obtient alors l'expression `*x`, qui est de type `struct personne`. On peut alors appliquer l'opérateur `.` à cette expression `*x`, afin d'obtenir son champ `age`. Il serait impossible d'appliquer directement l'opérateur `.` à `x`, car cet opérateur ne s'applique qu'à une structure. Or, `x` n'est pas une structure, mais l'*adresse* d'une structure.

Cependant, un opérateur spécial peut être utilisé pour accéder à un champ à partir de l'adresse d'une structure. Il s'agit de l'opérateur `->`, qu'on utilise de la façon suivante :

```
x->age = 99;
```

Remarquez l'absence de parenthèses et de `*`. L'opérateur `->` s'applique à l'adresse d'une structure, ici `x` qui est de type `struct personne*`, et à un champ de la structure indiqué par cette adresse, ici `age` qui est de type `short`. L'expression obtenue a pour valeur et pour type ceux du champ concerné (ici, `short`).

Remarque : les deux formes `(*variable).champ` et `variable->champ` sont équivalentes.

8.2.7 Exercices

Question du partiel 2005/2006 : l'université Galatasaray organise un tournoi de tennis de table en double. Chaque équipe engagée se compose 2 joueurs et possède un nom. Chaque joueur est caractérisé par un nom, un prénom, un âge et un score allant de 0 à 100 indiquant son niveau.

1) Définissez les types structures `joueur` et `equipe` correspondants.

```
struct joueur
{   char prenom[10];
    char nom[10];
    short age;
    short niveau;
};
```

```
struct equipe
{  char nom[10];
   struct joueur joueur1;
   struct joueur joueur2;
};
```

2) Écrivez une fonction `saisis_joueur` permettant de saisir les caractéristiques d'un joueur. Le paramètre doit être passé par adresse.

```
void saisis_joueur(struct joueur *j)
{  printf("entrez le prenom : ");
   scanf("%s",j->prenom);
   printf("entrez le nom : ");
   scanf("%s",j->nom);
   printf("entrez l'age : ");
   scanf("%d",&j->age);
   printf("entrez le niveau : ");
   scanf("%d",&j->niveau);
}
```

3) En utilisant la fonction précédente, écrivez une fonction `saisis_equipe` permettant de saisir les caractéristiques d'une équipe. Le paramètre doit être passé par adresse.

```
void saisis_equipe(struct equipe *e)
{  printf("entrez le nom : ");
   scanf("%s",e->nom);
   printf("joueur 1 : \n");
   saisir_joueur(&e->joueur1);
   printf("joueur 2 : \n");
   saisir_joueur(&e->joueur2);
}
```

4) Écrivez une fonction `main` qui crée et remplit un tableau de `N` équipes en utilisant la fonction `saisis_équipe`.

```
int main()
{  struct equipe tab[N];
   int i;
   for(i=0;i<N;i++)
   {  printf(".: equipe %d :\n",i);
      saisir_equipe(&tab[i]);
   }
}
```

5) Écrivez des fonctions `affiche_joueur` et `affiche_equipe` pour afficher à l'écran les équipes et leurs joueurs.

```
void affiche_joueur(struct joueur j)
{  printf("\tprenom : %s\n",j.prenom);
   printf("\tnom : %s\n",j.nom);
   printf("\tage : %d\n",j.age);
   printf("\tniveau : %d\n",j.niveau);
}

void affiche_equipe(struct equipe e)
{  printf("nom : %s\n",e.nom);
   printf("joueur 1 :\n");
   affiche_joueur(e.joueur1);
   printf("joueur 2 :\n");
   affiche_joueur(e.joueur2);
}
```

6) Complétez la fonction `main` précédente de manière à afficher les équipes après la saisie.

```
int main()
{  struct equipe tab[N];
   int i;
   for(i=0;i<N;i++)
   {  printf(".: equipe %d :\n",i);
```

```
        saisir_equipe(&tab[i]);
    }
    //
    for(i=0;i<N;i++)
    {  printf(".: equipe %d :\n",i);
       afficher_equipe(tab[i]);
    }
}
```

8.3 Unions

8.3.1 Déclaration

Une union est une autre forme de type complexe *hétérogène*. À la différence d'une structure, une union ne peut contenir qu'une seule valeur simultanément. Cependant, cette valeur peut être de plusieurs types différents, c'est pourquoi une union est un type hétérogène. De plus, les types de cette valeur peuvent être simples ou complexes, c'est pourquoi l'union est qualifiée de type complexe.

Remarque : les unions sont bien moins utilisées que les structures, on les présente ici dans un souci d'exhaustivité. Mais nous ne nous en servirons pas du tout en TP.

La syntaxe utilisée pour déclarer un type union est très similaire à celle des structures, à la différence qu'on utilise le mot-clé `union` au lieu de `struct`. En dehors de ça, une union est, comme une structure, une séquence de champs.

exemple : on veut définir une union dont la valeur est une taille, qui peut être représentée soit sous forme d'un nombre de centimètres représenté par un entier, soit d'un nombre de mètres représenté par un réel :

```
union taille
{  short centimetres;
   float metres;
};
```

Il ne faut pas se laisser tromper par la syntaxe. Même s'il y a deux champs différents dans cet exemple, la donnée stockée est unique. Ces deux champs correspondent à des alternatives, qui sont mutuellement exclusives, et non pas à deux valeurs distinctes comme pour une structure. La taille sera représentée soit en *cm*, soit en *m*, mais pas les deux en même temps.

La déclaration d'une variable de type union est comparable à celle d'une variable de type structure, à l'exception du mot `struct` qui est remplacé par `union` :

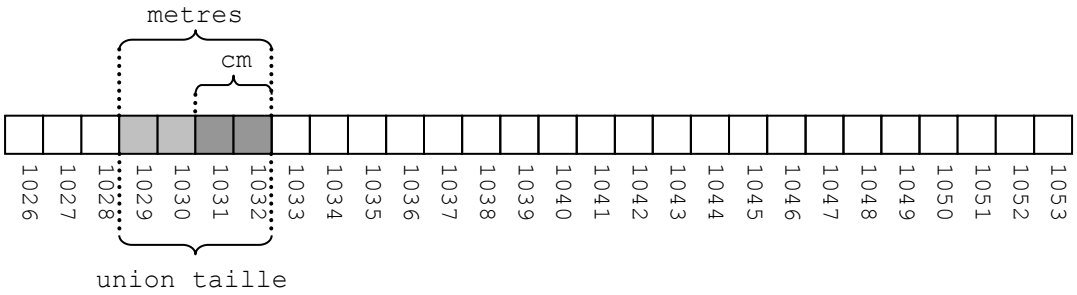
```
union taille t;
```

Notez que, comme pour les structures, on ne se contente pas de préciser le nom de l'union : on doit explicitement indiquer qu'il s'agit d'une union grâce au mot-clé `union`.

8.3.2 Mémoire

La place occupée en mémoire par une variable de type union correspond au champ occupant le plus d'espace dans ce type.

exemple : dans l'union `taille`, il s'agit du champ `metres`, qui est un `float` et prend donc 4 octets) alors que `centimetres` est un entier `short`, et ne prend donc que 2 octets:



On ne peut *pas* qualifier la représentation en mémoire d'une union de contigüe, puisque les champs ne se suivent pas, mais plutôt de *superposée*, puisque les champs sont représentés les uns sur les autres.

8.3.3 Manipulation

Chaque champ déclaré dans une union représente une modalité d'accès à l'information stockée. Par exemple, pour l'union taille, on peut accéder à la donnée contenue dans les 4 octets de la variable soit comme un entier `short`, soit comme un réel `float`. Si on veut accéder à une taille exprimée en nombre de *cm*, on utilisera le champ `centimetres` :

```
t.centimetres = 172;
printf("taille en cm : %d\n",t.centimetres);
```

Si on veut accéder à une taille exprimée en nombre de *m*, on utilisera le champ `metres` :

```
t.metres = 1.72;
printf("taille en m : %f\n",t.metres);
```

Bien sûr, si la valeur a été stockée dans un certain champ, il faut y accéder avec ce même champ. Si on stocke une taille en centimètres et qu'on tente d'afficher une taille en mètres, on obtiendra un affichage erroné, puisqu'on tentera d'afficher un `float` alors que c'est un `short` qui a été stocké. Or, nous avons vu que ces deux types de données n'avaient pas du tout la même représentation en mémoire. Et que le compilateur ne considérait pas ce genre de situation comme des erreurs. Autrement dit, cela provoquera des erreurs seulement à l'exécution.

Remarquez qu'à la différence des structures, l'utilisation de l'opérateur d'affectation sur les unions ne pose aucun problème.

Pour les tableaux et les paramètres de fonction, les unions se manipulent de la même façon que les structures.

8.4 Énumérations

8.4.1 Déclaration

Un type énuméré est un type simple correspondant à une liste de valeurs symboliques. On définit une liste de symboles (i.e. des noms) et une variable de ce type ne peut prendre sa valeur que parmi ces symboles.

On définit un type énuméré grâce au mot-clé `enum`, suivi de la liste de symboles séparés par des virgules :

```
enum nom_du_type
{  valeur_1,
   valeur_2,
   ...
   valeur_n
};
```

Par la suite, quand on déclare une variable de type énuméré, celle-ci peut prendre pour valeurs uniquement l'un des symboles `valeur_1`, `valeur_2`, …, `valeur_n` définis dans le type.

exemple :

```
enum istanbul
{  besiktas,
   sisli,
   beyoglu,
   uskudar,
   kadikoy,
   eminonu
};
enum istanbul x = besiktas;
```

Ici, on déclare d'abord un type `enum istanbul` énumérant les communes d'İstanbul, puis on déclare une variable `x` de ce type, et on lui affecte la valeur représentant Beşiktaş. Notez qu'à l'instar de `struct` et `union`, il est nécessaire de préciser le mot-clé `enum` quand utilise une énumération.

On peut se poser la question de savoir comment ces valeurs symboliques sont représentées en mémoire. En réalité, elles sont associées à une valeur entière, et le programme manipule ces valeurs entières lors de l'exécution. Autrement dit, les symboles eux-mêmes n'apparaissent que dans le code source. Ceci reflète l'idée que les types énumérés sont là pour rendre explicite la volonté du programmeur de limiter le nombre de valeurs possibles pour une variable à un ensemble de concepts clairement identifiés. En d'autres termes, les types énumérés permettent d'améliorer la lisibilité du code source.

Par défaut, le premier symbole est associé à la valeur 0, le deuxième à la valeur 1, etc. Dans notre exemple, `besiktas` vaut 0, `sisli` vaut 1, etc. Cependant, il est possible de modifier ces valeurs par défaut, au moyen de l'opérateur d'affectation : il suffit pour cela d'affecter à un symbole la valeur désirée. Par exemple, si on veut que `sisli` corresponde à la valeur 100, on définit le type de la façon suivante :

```
typedef enum
{  besiktas,
   sisli=100,
   beyoglu,
   uskudar,
   kadikoy,
   eminonu
} istanbul;
```

Alors `besiktas` vaudra 0, mais `sisli` vaudra 100. Mais du coup, `beyoglu` vaudra 101, `uskudar` 102, etc.

Le fait que les symboles soient codés par des entiers signifie qu'il est possible d'utiliser un symbole d'un type énuméré comme s'il s'agissait d'une valeur entière. Par contre, le compilateur empêchera de faire le contraire : impossible d'utiliser *directement* une valeur entière comme un symbole. Pour cela, il faut d'abord effectuer une conversation explicite à l'aide de l'opérateur de transtypage (cf. section 4.3.2).

Si on utilise plusieurs types énumérés, il est interdit :

- D'utiliser plusieurs fois le même symbole dans différents types (et bien sûr dans un seul type), pour des raisons d'ambigüité ;
- D'affecter à une variable un symbole qui n'est pas défini dans son type énuméré.

Il est par contre possible de définir dans un même type plusieurs symboles représentant la même valeur numérique.

8.4.2 Exercices

Une entreprise veut stocker dans un tableau ses recettes mensuelles pour une année.

1) Définissez un type énuméré `mois` permettant de représenter les 12 mois de l'année.

```
typedef enum
{  jan=1,
   fev,mars,avr,mai,juin,juil,aout,sept,oct,nov,dec
} mois;
```

2) Écrivez une fonction `affiche_mois` permettant d'afficher le mot correspondant au numéro du mois passé en paramètre.

```
void affiche_mois(mois m)
{  switch(m)
   {  case jan:
         printf("janvier");
         break;
      case fev:
         printf("fevrier");
         break;
      case mars:
         printf("mars");
         break;
      case avr:
         printf("avril");
         break;
      case mai:
         printf("mai");
         break;
      case juin:
         printf("juin");
         break;
      case juil:
         printf("juillet");
         break;
      case aout:
         printf("aout");
         break;
      case sept:
         printf("septembre");
         break;
      case oct:
         printf("octobre");
         break;
      case nov:
         printf("novembre");
         break;
      case dec:
         printf("decembre");
         break;
   }
}
```

3) Écrivez une fonction `saisis_recettes` demandant à un utilisateur de remplir le tableau des recettes mensuelles.

```
void saisis_recettes(int recettes[])
{  mois m;
   for(m=jan;m<=dec;m++)
   {  printf("entrez les recettes pour ");
      affiche_mois(m);
      printf(" : ");
      scanf("%d",&recettes[m-1]);
   }
}
```

8.5 Nommage de types

Il est possible de définir un nouvel identificateur pour un type, en utilisant l'instruction `typedef`, avec la syntaxe suivante :

```
typedef type nom_du_type;
```

Si le type est anonyme, alors il le fait de le nommer le rend plus facile à manipuler. S'il a déjà un nom, alors ce nouveau nom ne supprime pas l'ancien.

La principale utilisation de `typdef` concerne les types structurés, unions et énumérés, car elle évite de devoir répéter les mots clés `struct`, `union` ou `enum` en permanence.

exemple :

```
struct s_joueur
{  char prenom[10];
   char nom[10];
   short age;
   short niveau;
};
```

```
typedef struct s_joueur joueur;
```

Ici, on déclare d'abord un type structure, dont le nom complet est `struct s_joueur`. On définit ensuite un nouveau pour ce type en utilisant `typedef`. Ce nouveau nom est simplement `joueur`. Par la suite, on peut donc utiliser n'importe lequel de ces deux noms, qui font référence au même type : soit `struct s_joueur`, soit `joueur`. Pour déclarer une nouvelle variable x de ce type, on peut donc procéder ainsi :

```
joueur x;
```

Mais on peut aussi continuer à faire :

```
struct s_joueur x;
```

Remarque : on ne peut pas assigner à un type un nom qui est déjà utilisé pour un autre type.

Il est possible de définir le type que l'on veut nommer directement *dans* l'instruction `typedef`, de la façon suivante :

```
typedef struct s_joueur
{  char prenom[10];
   char nom[10];
   short age;
   short niveau;
} joueur;
```

On a déjà vu qu'il était possible définir des structures, unions et énumérations de façon anonyme. Il est donc possible de simplifier encore plus l'écriture : puisqu'il n'est pas obligatoire de nommer les types structures, on peut supprimer le mot `s_joueur` :

```
typedef struct
{  char prenom[10];
   char nom[10];
   short age;
   short niveau;
} joueur;
```

En dehors de cette utilisation concernant les structures, unions et énumérations, il est aussi possible d'utiliser typedef pour renommer les types simples.

exemple : donner le nom `entier` au type `int` :

```
typedef int entier;
```

Enfin, on peut même donner un nom aux tableaux d'une taille particulière :

exemple : on veut appeler `ligne` les tableaux de 80 `char` :

```
typedef char ligne [80];
```

9 Classes de mémorisation

On avait vu en section 3.2.5 que la mémoire allouée à un programme C est décomposée en 3 parties :

- *Segment de données* : variables globales ;
- *Pile* : variables locales ;
- *Tas* : allocation dynamique.

En réalité, la distinction est plus complexe que cela, et nous allons la préciser dans cette section.

9.1 Persistance d'une variable

Comme on l'a vu en section 3.2, une variable est caractérisée par les propriétés suivantes :

- Son identificateur ;
- Son type ;
- Sa portée ;
- Sa valeur.

On a aussi vu que la portée de la variable déterminait la zone de la mémoire dans laquelle elle est stockée : segment de données pour une variable globale et pile pour une variable locale.

Cependant, il existe une autre propriété, qui est la persistance d'une variable.

Persistance d'une variable : limites temporelles pendant lesquelles une variable est accessible.

On peut voir la portée d'une variable comme ses limites spatiales (dans l'espace du code source) et sa persistance comme ses limites temporelles (dans le déroulement du programme). Deux situations sont possibles : une variable peut-être persistante ou non-persistante.

Variable non-persistante : variable qui n'est accessible que pendant l'exécution du code source contenu dans sa portée.

Variable persistante : variable qui reste accessible jusqu'à la fin du programme, même après l'exécution du code source situé dans sa portée.

Si la portée d'une variable est déterminée par l'emplacement de sa déclaration dans le programme, en revanche sa persistance est spécifiée via un *modificateur*. Un modificateur est un mot-clé que l'on place devant la déclaration de la variable, pour préciser ce que l'on appelle sa classe de mémorisation.

Classe de mémorisation : méthode utilisée par le programme pour manipuler une variable donnée, et influençant en particulier l'emplacement de la mémoire utilisée pour stocker sa valeur.

Le concept peut également être étendu aux *fonctions*, comme nous allons le voir à la fin de cette section.

Remarque : le concept de classe de mémorisation étudiée ici n'a rien à voir avec la notion de classe en programmation orientée objet (POO), veuillez à ne pas les confondre.

9.2 Déclarations locales

On peut appliquer 4 modificateurs différents à une déclaration locale, i.e. une déclaration de variable contenue dans un bloc (par opposition à une déclaration globale, qui n'est contenue dans aucun bloc).

9.2.1 Classe `auto`

La classe `auto` est le défaut pour les variables locales. Cela signifie que c'est la classe à laquelle la variable est affectée si on ne précise aucune classe. C'est donc la classe que nous avons utilisée jusqu'à présent.

Une variable de classe `auto` est stockée dans la *pile*, et sa portée se limite au bloc dans lequel elle est déclarée. La variable est donc désallouée dès que le bloc a été exécuté. Si le bloc est exécuté plusieurs fois, la variable est réallouée à chaque fois. Il ne faut pas supposer que la valeur de la variable reste la même entre de allocations, car il est possible que la partie de la mémoire qui lui est allouée ne soit pas la même à chaque fois. Il faut donc bien veiller à ce que la variable soit toujours correctement initialisée.

exemple :

```
void ma_fonction()
{  int i;
   for(i=0;i<5;i++)
   {  auto int x = 0;
      printf("%d: x=%d\n",i,x);
      x++;
   }
}
```

Dans cet exemple, la valeur de `x` sera toujours 0 car la variable est réallouée et réinitialisée à chaque fois que le bloc du `for` est exécuté. Notez également que la variable `i` est elle aussi de classe `auto`, car aucune classe de mémorisation n'est précisée lors de sa déclaration, et qu'`auto` est la classe par défaut pour les variable locales.

9.2.2 Classe `static`

Les variables de classe `static` sont stockées dans le *segment de données*. Comme il s'agit de variables locales, leur portée reste limitée au bloc qui les contient, exactement comme pour les variables de classe `auto`.

Cependant, comme une variable `static` est persistante, elle n'est allouée qu'une seule fois, et continue d'exister même quand son bloc a été exécuté. Si le bloc est exécuté à nouveau, la variable au début de la nouvelle exécution possède la même valeur qu'elle avait à la fin de l'exécution précédente.

exemple :

```
void ma_fonction()
{  int i;
   for(i=0;i<5;i++)
   {  static int x=0;
      printf("%d: x=%d\n",i,x);
      x++;
   }
}
```

Au contraire de l'exemple précédent, ici `x` prendra successivement les valeurs 0, 1, 2, 3 et 4. Sa valeur n'est pas accessible à l'extérieur du `for`, mais elle continue à exister en mémoire malgré cela.

exemple : une fonction qui compte le nombre de fois qu'elle est appelée

```
int compteur(void)
{  static int nb_appels = 0;
   nb_appel++;
   return nb_appels;
}
```

9.2.3 Classe `register`

Une variable de classe `register` est, dans la mesure du possible, stockée directement dans un registre du processeur (une sorte de mémoire intégrée dans le processeur, par opposition à la mémoire centrale de l'ordinateur). Ceci qui permet d'accélérer l'accès à cette variable, car les registres sont plus rapides que la mémoire classique.

La portée et la persistance sont les mêmes que pour une variable de classe `auto`. En ce qui nous concerne, il vaut mieux ignorer cette classe et laisser le compilateur décider comment utiliser les registres. Nous ne l'utiliserons donc pas dans le cadre de ce cours.

9.2.4 Classe `extern`

La classe `extern` ne considère en réalité par les variables locales, mais les variables locales. Cependant, on la considère ici dans le cas où elle s'applique à une variable déclarée dans un bloc, et qui pourrait donc passer pour une variable locale.

La classe `extern` permet d'utiliser dans un bloc donné (par exemple une fonction) une variable globale qui a été définie *ailleurs* dans le programme. Une variable `extern` est donc stockée dans le segment de données. On l'utilise dans deux cas.

Tout d'abord, pour utiliser une variable globale déclarée dans le même fichier, mais plus loin dans le code. Le compilateur ne la connait donc pas encore quand il arrive à la variable de classe `extern` qui nous intéresse, et c'est un moyen de lui faire comprendre qu'on fait référence à une variable qui est en réalité déclarée ailleurs dans le même programme.

exemple :

```
int ma_fonction()
{  extern int x;
   int resultat = 2 * x;
   x++;
   return x;
}
...
int x = 4;
```

Ici, la variable `x` est une variable globale déclarée après la fonction `ma_fonction`. Pourtant, on veut utiliser `x` dans `ma_fonction`, car son rôle est de renvoyer le double de `x` puis de l'incrémenter. Si jamais la variable `x` était déclarée avant `ma_fonction`, on pourrait l'utiliser directement sans problème. Mais ici, ce n'est pas le cas. C'est pour cela que l'on doit pré-déclarer `x` dans `ma_fonction`, en précisant qu'il s'agit d'une variable de classe `externe`, ce qui permet au compilateur de faire le lien avec la déclaration de `x` placée après cette fonction.

On utilise `extern` aussi pour faire référence à une variable globale qui est déclarée dans un *autre fichier*, typiquement : une bibliothèque. Le principe est le même : on veut indiquer au compilateur que l'on veut simplement utiliser une variable qui est déjà définie ailleurs et non pas en créer une nouvelle.

Comme on va le voir dans la section suivante, pour qu'on puisse faire référence à une variable globale existante en utilisant `extern` dans un bloc, il faut que la déclaration de la variable globale soit elle-même réalisée avec la classe `extern` (qui est la classe par défaut pour les variables globales).

exemple : on représente à gauche un programme `main.c` et à droite une bibliothèque `xxxx` (plus exactement, son fichier `xxxx.c`) :

```
#include "xxxx.h"
...
int ma_fonction()
{  extern int x;
   int resultat = 2 * x;
```

```
   x++;
   return x;
}
...
```

```
extern int x = 4;
```

9.3 Déclarations globales

Pour les déclarations globales, on distingue seulement deux classes de mémorisation : `extern` et `static`. Comme nous allons le voir, si ces noms correspondent à des classes déjà présentées pour les variables locales, en revanche leurs sémantiques sont différentes.

9.3.1 Classe `extern`

La classe `extern` est la classe *par défaut* pour les variables globales. Cela signifie donc que si on ne précise pas de classe de mémorisation lorsqu'on déclare une variable globale, c'est `extern` qui est utilisée.

Une variable de classe `extern` est stockée dans le segment de données. Sa portée recouvre tout le programme, et la variable est persistante, i.e. elle existe jusqu'à la fin du programme.

Une variable `extern` peut être utilisée directement dans la partie du fichier qui suit sa déclaration. Pour la partie du fichier qui précède sa déclaration, il faut d'abord effectuer une pré-déclaration locale, en utilisant `extern` comme expliqué en section 9.2.4. Pour utiliser la variable globale `extern` dans un autre fichier, il faut aussi effectuer une pré-déclaration en utilisant `extern`, et celle-ci peut être soit locale, soit globale.

exemple : dans les TP utilisant la SDL, on utilise une variable globale `ecran`, qui est de type adresse d'une `SDL_Surface`. Pour simplifier, cette surface est la matrice de pixels représentant la fenêtre graphique dans laquelle on dessine pour modifier ce qui est affiché à l'écran.

Dans la bibliothèque graphisme utilisée pour manipuler la SDL en TP, la déclaration de cette variable se fait de la façon suivante :

```
SDL_Surface* ecran;
```

Dans les autres fichiers constituant le programme, par exemple `main.c`, cette variable doit être redéclarée de la façon suivante pour pouvoir être utilisée :

```
extern SDL_Surface* ecran;
```

9.3.2 Classe `static`

Pour une déclaration globale, la classe `static` permet de créer une variable globale persistante (comme pour `extern`), mais dont la portée est limitée au fichier dans lequel elle est déclarée. Autrement dit, si cette variable est déclarée dans une certaine bibliothèque, elle ne pourra pas être utilisée dans une autre bibliothèque, ni dans le programme principal `main.c`, même si on procède à une pré-déclaration.

9.4 Fonctions

Bien qu'on ne puisse plus vraiment parler de classe de mémorisation, les modificateurs utilisés pour les déclaration globales peuvent aussi s'appliquer à des fonctions.

9.4.1 Classe `extern`

La classe `extern` est la classe par défaut pour les fonctions, comme pour les variables globales. Une fonction de cette classe est accessible depuis n'importe quel point du programme, que ce soit le même fichier ou un autre fichier.

9.4.2 Classe `static`

Au contraire, une fonction de classe `static` n'est accessible que dans le fichier dans lequel elle est définie. On peut voir ça comme une façon de définir des fonctions internes à une bibliothèque, qui ne sont pas accessibles à l'extérieur.

10 Pointeurs

La notion de pointeur est centrale en langage C, et dans la plupart des langages de programmation, bien qu'elle n'apparaisse pas toujours aussi explicitement qu'en langage C. Elle touche différents concepts que nous avons déjà abordés dans les section précédentes, comme la notion de variable (section 3.2) et le passage de paramètre par adresse (section 7.2.2). Dans cette section, nous introduisons d'abord les pointeurs par ce biais, puis développons leurs caractéristiques.

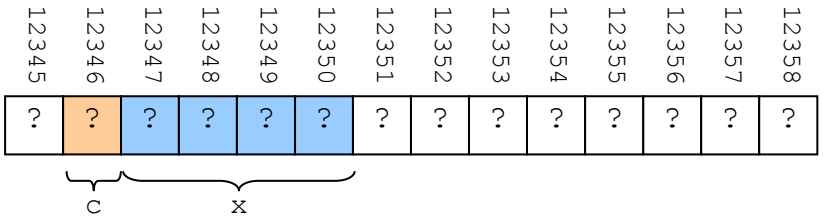
10.1 Présentation

10.1.1 Présentation

On a vu précédemment que le compilateur maintenait une *table des symboles*, lui permettant de représenter la liste des variables déclarées, chacun étant décrite par des propriétés telles que leur identificateur, leur type et leur adresse. Pour accéder au contenu d'une variable, on précise par la suite son identificateur, et son adresse est alors obtenue automatiquement.

exemple :

```
char c;
long x;
```

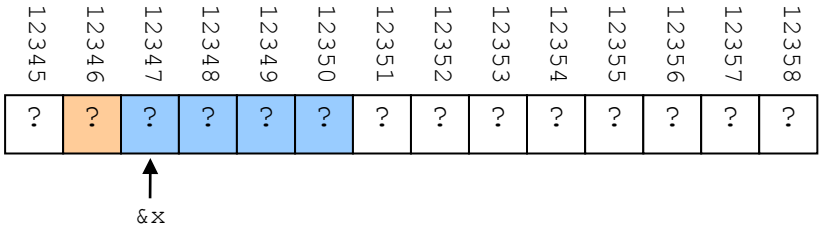


Ici, la variable `c` est de type `char`, donc elle occupe un seul octet, et le schéma indique que son adresse est 12345. La variable `x` est de type long, donc elle occupe 4 octets, et son adresse est 12346.

Grâce à l'adresse, on connait l'emplacement de la variable dans la mémoire, et grâce à son type, on connait l'espace qu'elle occupe et le codage des données qu'elle contient. Par exemple, on sait que `x` occupe 4 octets et contient un entier relatif codé selon le principe du complément à 2 (cf. section 2.2.3). Ce sont les seules informations nécessaires pour récupérer ou modifier le contenu de la variable.

Mais plutôt que d'obtenir ces informations via le nom de la variable, on pourrait aussi bien les obtenir manuellement et accéder quand même à son contenu. Si on connait l'adresse et le type de données, peu importe le nom de la variable : on peut accéder à son contenu même sans connaitre son identificateur.

Considérons la variable `x` dans l'exemple précédent. On sait que l'on peut obtenir son adresse grâce à l'opérateur `&`. De plus, on sait que le type de cette expression `&x` est `long*`. On peut donc facilement obtenir les deux informations dont on a besoin pour accéder au contenu de `x` (son adresse et son type).



C'est précisément ce que l'on fait quand on passe un paramètre *par adresse*. Considérons par exemple le programme suivant :

```
void fonction(long *y)
{  *y = 999;
}
...
int main(int argc, char** argv)
{  char c;
   long x;
   fonction(&x);
   ...
```

L'en-tête de la fonction indique que le paramètre `y` contient l'adresse d'une variable de type `long`. La fonction ne sait pas que le paramètre qu'elle va recevoir s'appelle en réalité `x` dans la fonction `main`. Toutes les informations dont elle dispose pour travailler sur cette variable, c'est son adresse, qui est contenue dans `y`, et son type, qui est implicitement indiqué par le type de `y` : puisque `y` est de type `long*`, alors la variable correspondante est forcément de type `long`.

Dans la fonction, on sait donc où est la variable, et comment elle est codée. On ne connait pas son identificateur, mais ça n'a aucune importance car cette information n'est pas nécessaire.

10.1.2 Définition

Le paramètre `y` utilisé dans l'exemple précédent est appelé un *pointeur* :

Pointeur : variable dont la valeur est une adresse.

Un pointeur est donc une variable qui la particularité de contenir une adresse. Comme une adresse n'est rien d'autre que le numéro d'un octet, il s'agit d'une variable *entière*. Cependant, son type ne peut pas être l'un de ceux utilisés habituellement pour les entiers (`char`, `short`, `long`…) car une adresse n'est pas n'importe quel entier : cette valeur a une sémantique bien particulière. C'est l'une des raisons pour lesquelles un pointeur a un type spécial, dont le nom se termine par `*`.

Comme on l'a vu auparavant, le type du pointeur permet de déterminer le type de la variable pointée : un pointeur de type `int*` pointe forcément vers un entier de type `int`.

Variable pointée : variable dont l'adresse est contenue dans un pointeur.

La variable pointée peut être de *n'importe quel type* : type simple, comme on l'a déjà vu, mais aussi types complexes, ou même autres pointeurs. Par exemple, le type `int**` correspond à l'adresse de l'adresse d'un pointeur.

Il existe même un type `void*`, qui correspond à l'adresse d'un type indéterminé, et que nous utiliserons quand nous étudierons l'allocation dynamique (en section 11).

10.1.3 Déclaration et initialisation

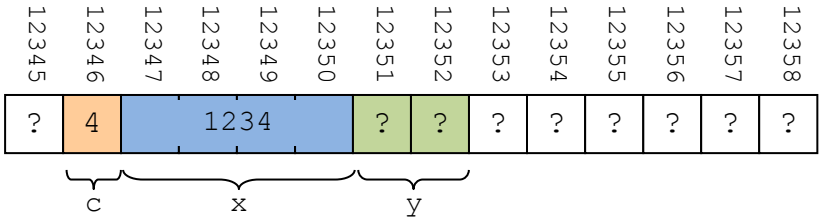
On déclare un pointeur comme une variable classique, à l'exception du fait qu'on fait précéder le nom de la variable de `*` :

```
char c = 4;
long x = 1234;
long *y;
```

Lorsqu'on déclare un pointeur, comme pour n'importe quelle variable, le compilateur réserve une zone mémoire suffisamment grande pour pouvoir y stocker sa valeur. Cette valeur étant ici une adresse. La taille d'une adresse dépend du système utilisé. Elle peut varier en fonction du matériel et du système d'exploitation. On ne peut pas savoir à l'avance combien d'octets sont nécessaire à la représentation d'un pointeur. Par contre, on sait que tous les pointeurs, quel que soit leur type, occupent exactement le même nombre d'octets. En effet,

indépendamment du type des données pointées, le pointeur contient une simple adresse, et donc occupe toujours le même nombre d'octets (ce qui n'est pas le cas de la donnée pointée).

Dans les exemples qui suivent, pour simplifier les schémas, on supposera que l'adresse est codée sur 2 octets.

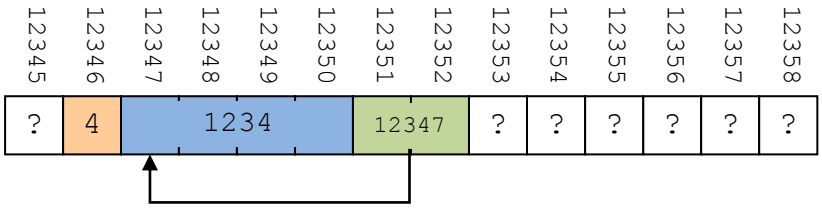


Dans notre code source d'exemple, le pointeur y a été déclaré, mais pas initialisé. Par conséquent, son contenu est inconnu, comme pour n'importe quelle variable non-initialisée. Cependant, ici ce contenu est une adresse. Cela signifie donc qu'en l'absence d'initialisation, un pointeur potentiellement sur n'importe quel octet de la mémoire.

Avant d'utiliser un pointeur, il faut donc *obligatoirement* l'initialiser, sinon on risque d'aller modifier n'importe quoi dans la mémoire, et de provoquer une erreur d'exécution (peut-être même une erreur d'exécution *fatale*). Comme pour les variables classiques, ce genre d'oubli n'est pas détecté à la compilation.

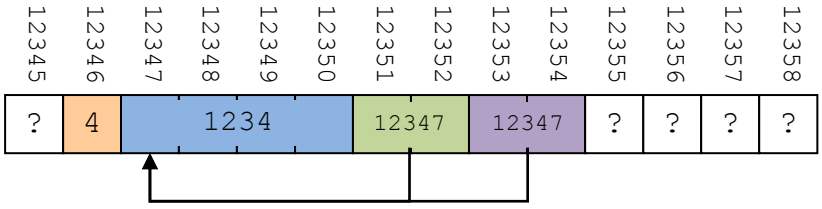
Pour utiliser le pointeur, il faut le faire pointer sur une adresse. Pour cela, on utilise une expression appliquant l'opérateur `&` à une variable. Par exemple, on peut faire pointer `y` sur `x` :

```
y = &x;
```



Mais on peut aussi utiliser un autre pointeur qui a déjà été initialisé. Ainsi, on peut déclarer un nouveau pointeur `z` pointant sur la même adresse que `y`, donc : `x`.

```
long *z = y;
```



Remarque : on ne doit affecter à un pointeur que des adresses de données correspondant au type déclaré, c'est-à-dire dans cet exemple des entiers de type `long`. Ainsi, on ne peut pas faire `y = &c;` car `c` est un entier de type `char`.

Initialiser un pointeur systématiquement lors de sa déclaration est une bonne habitude de programmation. Dans le cadre de ce cours, c'est même rendu obligatoire. Si l'adresse pointée est connue à la déclaration, alors elle doit être utilisée. Sinon, si elle ne sera connue que plus tard, alors le pointeur doit être initialisé avec la constante `NULL` (définie dans la bibliothèque stdlib.h), qui représente l'absence d'adresse.

Par exemple, si on veut déclarer un pointeur z sans connaitre à l'avance l'adresse sur laquelle il pointe, on fera : `long *z = NULL;`

10.1.4 Accès

Une fois qu'un pointeur a été initialisé, l'opérateur `*` permet d'accéder à la valeur contenue à l'adresse pointée.

exemple : afficher le contenu de la variable `x` en passant par les pointeurs `y` ou `z` :

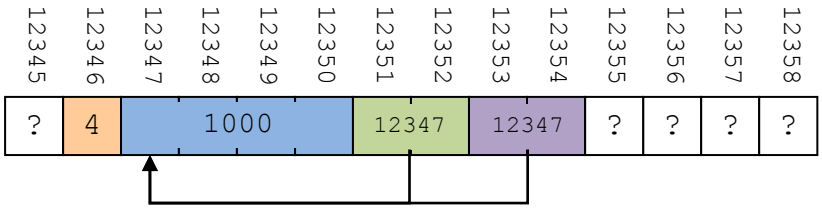
```
printf("contenu de x via x : %d",x);
printf("contenu de x via y : %d",*y);
printf("contenu de x via z : %d",*z);
```

Les trois instructions affichent la même valeur 1234.

De la même façon, si on modifie la valeur contenue à l'adresse pointée, on modifie la du même coup variable classique située à cette adresse.

exemple :

```
*y = 1000;
```



Si on exécute à nouveau les trois `printf` précédents, on obtiendra encore trois fois la même valeur (ici : 1000).

exemples : type et valeur d'expressions

- `x` : type `long` et valeur 1234
- `&x` : type `long*` et valeur 12347
- `y` : type `long*` et valeur 12347
- `z` : type `long*` et valeur 12347
- `*y` : type `long` et valeur 1234
- `*z` : type `long` et valeur 1234
- `&y` : type `long**` (adresse d'un pointeur d'entier, donc adresse de l'adresse d'un entier) et valeur 12351
- `&z` : type `long**` et valeur 12353
- `*(&x)` : type long et valeur 1234
- `&(*y)` : type long* et valeur 12347

10.1.5 Indirection

Le fait de manipuler une valeur via son adresse mémoire est appelé indirection :

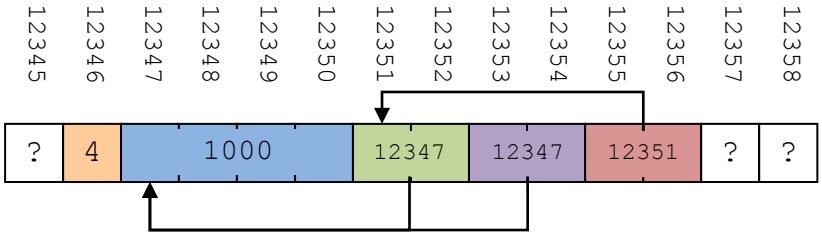
Indirection : capacité à accéder une valeur en utilisant son adresse.

Lorsqu'on manipule l'adresse de la valeur, on parle d'indirection simple. Mais il est possible de multiplier les niveaux d'indirection. Par exemple, si on manipule l'adresse d'un pointeur qui pointe lui-même sur la valeur qui nous intéresse, on parle d'indirection double, ou on dit qu'il y a deux niveaux d'indirection.

Il n'y a pas de limite au nombre de niveaux que l'on peut définir. En pratique, on n'a généralement pas besoin d'utiliser plus de 3 niveaux (et ce sera le cas en TP).

exemple : créons un pointeur de pointeur `w`, que l'on fait pointer sur `y` :

```
long **w = &x;
```



L'adresse stockée dans `w` est 12351 (adresse de `y`). Le type de l'expression `*w` est `short*`, et sa valeur est 12347, qui correspond bien au contenu de `y`, qui est l'adresse de l'entier `x`. Le type de l'expression `**w` est `short`, et sa valeur est 1000 qui est bien le `short` contenu dans `x`.

Remarque : il est important de ne pas confondre l'étoile `*` utilisée sur les types et celle utilisée sur les variables. Bien que le symbole utilisé soit le même, la sémantique est différente.

L'étoile utilisée sur les types indique que la variable est un pointeur. Chaque étoile rajoute un niveau d'indirection. Ainsi, `int*` représente un pointeur sur un entier, i.e. une variable contenant l'adresse d'un entier ; `int**` représente un pointeur sur un pointeur sur un entier, i.e. une variable qui contient l'adresse d'une variable qui contient l'adresse d'un entier, etc.

L'étoile utilisée sur une variable représente l'opérateur de déréférençage (aussi appelé opérateur d'indirection), qui est l'*opérateur inverse* de l'opérateur de référençage `&`. Là où `&` renvoie l'adresse d'une variable, `*` au contraire renvoie la valeur contenue à une adresse pointée.

Si on raisonne en termes de types, l'opérateur `&` rajoute une étoile (i.e. un niveau d'indirection) au type de la variable à laquelle il s'applique. Par exemple, pour une variable `int a`, l'expression `&a` a pour type `int*`. Pour un pointeur `int *b`, l'expression `&b` a pour type `int**`. L'opérateur `*`, au contraire, enlève une étoile au type du pointeur auquel il s'applique. Ainsi, pour un pointeur `int *b`, l'expression `*b` a pour type `int`. Pour un (double) pointeur `int **c`, l'expression `*c` a pour type `int*`, et `**c` a pour type `int`.

10.1.6 Exercice

Donnez le tableau de situation du programme suivant :

```
1   short x,y,*p1,*p2;
2   x = 4;
3   p1 = &x;
4   y = *p1;
5   y = 8;
6   p2 = &y;
7   (*p1)++;
8   y = p1;
9   y++;
10  p2 = 12;
11  p1 = y;
```

ligne	x	y	p1	*p1	p2	*p2
1	_	_	NULL	_	NULL	_
2	4	_	NULL	_	_	_
3	4	_	&x	4	NULL	_
4	4	4	&x	4	_	NULL
5	4	8	&x	4	_	NULL
6	4	8	&x	4	&y	8
7	5	8	&x	5	&y	8
8	5	&x	&x	5	&y	&x
9	5	&x+1	&x	5	&y	&x+1
10	5	&x+1	&x	5	12	?
11	5	&x+1	&x+1	5	12	?

En fait, les opérations des lignes 8, 10 et 11 ne sont pas valides :

- ligne 8 : on affecte une valeur `short*` à une variable de type `short`
- ligne 10 et 11 : on affecte une valeur `short` à un pointeur de type `short*`

10.2 Arithmétique des pointeurs

On peut réaliser deux types d'opérations sur les pointeurs :

- Travailler sur la *valeur* située à l'adresse pointée ;

- Travaillée sur l'*adresse* contenue dans le pointeur.

La modification de la valeur située à l'adresse pointée se fait comme pour une variable classique, avec en plus l'utilisation de l'opérateur `*`. Pour cette raison, on ne reviendra plus dessus.

Par contre, l'adresse pointée peut être modifiée de différentes façons, autres que l'affectation que nous avons déjà utilisée : par addition ou par soustraction. Le comportement de ces opérateurs varie en fonction du type des opérandes, et on parle d'*arithmétique des pointeurs* pour désigner ce comportement particulier.

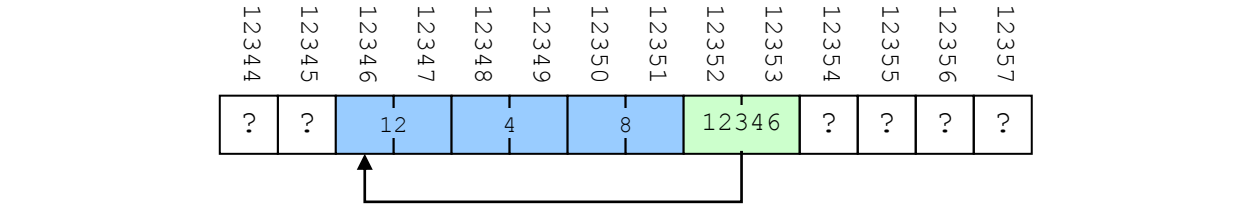
10.2.1 Addition/soustraction d'un entier

Soit `p` un pointeur sur une valeur de type `t` située à l'adresse a, et soit s le nombre d'octets occupés par une variable de type `t`. Il est possible de définir une expression dans laquelle on additionne ou soustrait un nombre entier k au pointeur `p`. La valeur de cette expression est une adresse de même type que `p`, augmentée ou diminuée de $k \times s$ octets. Formellement, l'expression `p+k` vaut $a + k \times s$ et l'expression `p-k` vaut $a - k \times s$.

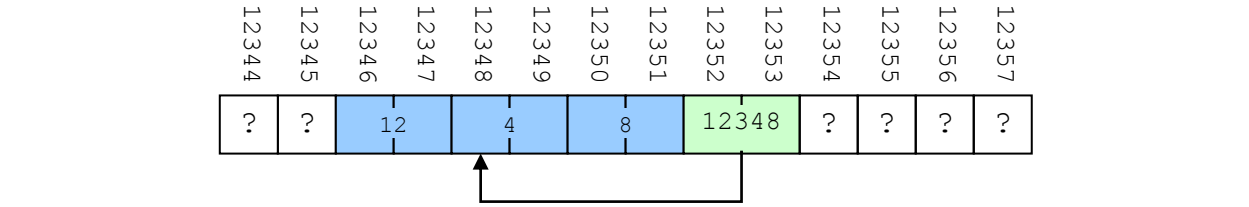
Ces deux opérations sont particulièrement adaptées au déplacement d'un pointeur dans un tableau. En supposant qu'on fait pointer `p` sur un tableau existant et de même type, le fait d'incrémenter `p` de k va permettre de se déplacer de k éléments vers la droite, alors que le fait de décrémenter `p` de k va le déplacer de k éléments vers la gauche.

exemple : on définit un tableau `tab`, puis un pointeur `p` pointant au début du tableau, et on incrémente `p` :

```
short tab[3]={12,4,8},*p;
p = &tab[0];
p++;
```

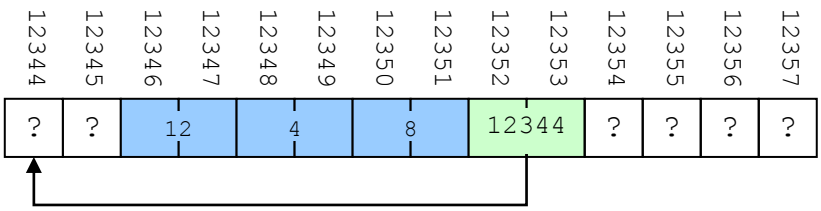



```
p++;
```



Dans cet exemple, on incrémente un pointeur de type `short*`, donc $s = 2$ et on augmente l'adresse qu'il contient de 1×2 octets. Cela revient, pour le pointeur, à se déplacer sur l'entier `short` situé à droite de celui pointé jusque là.

Si on avait effectué l'opération `p--;` à la place, on aurait eu :



Par conséquent, pour n'importe quel type de tableau, le fait d'additionner/soustraire un entier k à un pointeur pointant sur un des éléments du tableau permet de se déplacer dans le tableau d'un nombre d'éléments k.

exercices :

1) Affichez le contenu d'un tableau `short tab[10]` en utilisant un pointeur et une boucle `for`, et en modifiant l'adresse pointée à chaque itération.

```
short tab[10],*p,i;
...
p = &tab[0];
for(i=0;i<10;i++)
{  printf("element %d : %d",i,*p);
   p++;
}
```

2) Même chose, mais cette fois sans modifier le pointeur.

```
short tab[10],*p,i;
...
p = &tab[0];
for(i=0;i<10;i++)
   printf("element %d : %d",i,*(p+i));
```

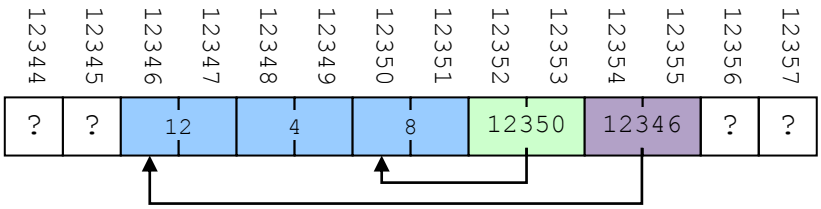
10.2.2 Soustraction de pointeurs

Nous avons vu que l'addition et la soustraction étaient toutes les deux définies si les opérandes sont un pointeur et un entier. Par contre, s'il s'agit de deux pointeurs, seule la soustraction est définie.

Il faut d'abord que les deux pointeurs soient du même type. Le résultat de l'opération est un entier représentant le nombre d'éléments séparant les éléments pointés dans la mémoire. Comme précédemment, on tient compte de la taille s d'un élément. C'est la raison pour laquelle on a besoin que les deux pointeurs soient du même type : ainsi, les éléments qu'ils permettent de manipuler sont tous de même taille.

exemple :

```
short tab[3]={12,4,8},*p1,*p2;
p1 = &tab[2];
p2 = &tab[0];
printf("p1-p2=%d",p1-p2);
```



Dans cet exemple, on initialise d'abord un tableau contenant 3 valeurs `short`, puis on fait pointer un pointeur `p1` sur son élément 0, et un pointeur `p2` sur son élément 2. On calcule ensuite l'expression `p1-p2`, qui correspond à un entier. En comparaison, le résultat de l'addition ou de la soustraction d'un pointeur et d'un entier était un pointeur. Ici, on calcule la différence entre les adresses pointées par `p1` et `p2`, ce qui donne un nombre d'octets, et on divise par la taille d'un short, pour obtenir un nombre d'élément : $(12350 - 12346)/2 = 4/2 = 2$.

10.3 Pointeurs et tableaux

Les pointeurs ne sont pas les premiers types relatifs à des adresses que nous rencontrons. En effet, on sait que l'identificateur d'un tableau est lui aussi interprété comme une adresse dans la plupart des expressions (cf. section 6). De nombreuses similarités existent effectivement

entre les deux types (pointeurs et tableaux), cependant il serait faux de croire qu'ils sont équivalents.

Dans cette sous-section, nous présentons d'abord les points communs et les différentes entre tableaux et pointeurs, puis nous nous intéressons aux tableaux de pointeurs.

10.3.1 Syntaxe similaire

Dans le début de cette section, nous avons vu qu'il était possible de parcourir un tableau `tab` en utilisant un pointeur `p` :

1. On initialise `p` avec l'adresse du premier élément du tableau `&tab[0]` (ou tout de tout autre élément du tableau convenant comme point de départ du parcours) ;
2. On additionne l'index `i` de l'élément voulu à `p` pour obtenir une expression `p+i` correspondant à l'adresse de l'élément voulu.
3. L'opérateur `*` appliqué à l'expression `p+i` donne `*(p+i)` et permet d'accéder au contenu de l'élément `i` tu tableau.

Mais nous savons aussi que l'identificateur d'un tableau utilisé dans une expression est généralement interprété comme l'adresse du premier élément du tableau. Pour être plus précis, lors de l'évaluation d'une expression de ce genre, un identificateur de type `xxxx[]` est remplacé implicitement (par le compilateur) par un pointeur de type `xxxx*`, pointant sur le premier élément du tableau.

L'adresse de l'élément `i` peut donc être calculée selon le même principe qu'avec le pointeur, en utilisant l'expression `tab+i`. Par conséquent, on peut utiliser de façon interchangeable les expressions `tab[i]`, `*(p+i)` et `*(tab+i)`. Il faut noter qu'elles ne sont pas *toujours* équivalentes cependant, car si `tab` désigne toujours l'adresse du *premier* élément du tableau, en revanche `p` peut désigner *n'importe quel élément*.

Il est possible d'utiliser la syntaxe des pointeurs avec les tableaux, mais le contraire est vrai aussi : on peut utiliser l'opérateur `[]` sur un pointeur. Ainsi, si on veut accéder à l'élément situé `i` éléments après `p`, on peut utiliser l'expression `p[i]`, qui est équivalente à `*(p+i)`.

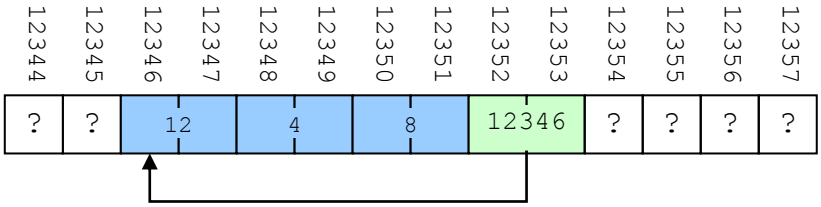
La différence avec les tableaux est que les pointeurs supportent l'utilisation d'index *négatifs*. Si $i < 0$, alors l'expression `p[i]` permet d'accéder à la valeur de l'élément situé `i` éléments *avant* `p`. Autrement dit, dans ce cas-là, cette expression `p[i]` est équivalente à `*(p-i)`. Un index négatif n'aurait pas de sens pour un tableau, car l'identificateur d'un tableau indiquant l'adresse du premier élément, on aurait alors forcément un débordement de tableau.

En conclusion, on peut donc compléter les différentes façons d'accéder à un élément dans un tableau : `tab[i]`, `*(p+i)`, `*(tab+i)` et `p[i]`.

10.3.2 Substitution des tableaux

Puisqu'on peut utiliser les tableaux et les pointeurs quasiment de la même façon, avec des contraintes supplémentaires pour les tableaux, il est tentant de supposer qu'un tableau est juste une sorte particulière de pointeur. Comme on sait que l'identificateur d'un tableau ne peut pas être une left-value (cf. section 6), et qu'on ne peut donc pas modifier l'adresse d'un tableau, les tableaux sont parfois qualifiés de *pointeurs constants*. Cependant, cette observation est fausse, et il suffit de dessiner l'occupation mémoire d'un tableau et d'un pointeur pour s'en apercevoir :

```
short tab[3]={12,4,8},*p;
p = &tab[0];
```



Le pointeur `p` est une variable qui occupe ici 2 octets en mémoire (rappelons que la taille d'un pointeur varie d'un système à l'autre, et que la valeur 2 n'est ici qu'un exemple). Son type est `short*` et sa valeur est l'adresse 12345.

Le tableau `tab` est une variable qui occupe 6 octets (soit 3 valeurs de type `short`). Son type est `short[3]` et il contient trois valeurs distinctes 12, 4 et 8. Notez que l'adresse 12346 n'apparait jamais explicitement, à la différence de ce qu'on avait observé pour `p`. La caractéristique principale d'un pointeur étant de contenir une adresse, on peut en conclure que `tab` n'est pas un pointeur, puisqu'il ne contient pas d'adresse.

Considérons les instructions suivantes :

```
tab[2] = 5;
p[2] = 5;
```

Leur résultat est le même : on place la valeur 5 dans l'élément numéro 2 du tableau. Mais leur évaluation est différente. Pour le tableau (1ère ligne), on obtient directement l'adresse de `tab` (comme pour n'importe quelle variable) et on se rend dans son élément numéro 2 que l'on modifie. Pour le pointeur (2ème ligne), on obtient l'adresse de `p`, va consulter sa valeur, qui est l'adresse de `tab`. On va ensuite à l'adresse de `tab`, puis on se déplace dans son élément numéro 2 pour finalement le modifier. On a donc une étape supplémentaire.

La confusion décrite plus haut vient du fait que, comme on l'a déjà mentionné, quand l'identificateur d'un tableau de type `xxxx[]` apparait dans une expression, il est remplacé par un pointeur de type `xxxx*` pointant sur son premier élément. Ce pointeur est géré automatiquement et de façon transparente par le compilateur. De plus, il est non-modifiable, et on a donc effectivement un pointeur *constant*.

Mais cette substitution n'a pas *toujours* lieu : il y a plusieurs exceptions, correspondant aux situations où l'identificateur du tableau est l'opérande de certains opérateurs particuliers. On peut citer entre autres `sizeof` et `&` (opérateur de référençage). Soit l'instruction suivante, appliquée au tableau de l'exemple précédent :

```
printf("Taille du tableau tab: %d\n", sizeof(tab));
```

Le résultat affiché sera 6, soit 2 × 3 octets, ce qui correspond bien à l'espace mémoire total occupé par le tableau. Si la substitution avait lieu pour `sizeof`, il renverrait la taille d'un pointeur, ce qui n'est pas le cas ici. Pour `&`, considérons l'expression suivante : `&tab`. Elle est de type `int (*)[10]`, c'est-à-dire : pointeur sur un tableau de 10 entiers. Là encore, il n'y a donc pas de substitution, sinon on aurait simplement `int**`.

Cette fameuse substitution a aussi lieu lorsqu'on passe un tableau en paramètre à une fonction : lors de l'appel, l'expression correspondant au nom du tableau est convertie implicitement en un pointeur, qui est lui-même passé à la fonction.

Cela signifie qu'une fonction reçoit *toujours* un pointeur, jamais un tableau. Lorsque le paramètre formel est déclaré en tant que tableau, il est lui aussi implicitement converti en un pointeur. Comme on a vu qu'il était possible de traiter un tableau à peu près comme un pointeur, cela ne pose pas de problème dans la fonction. Mais on pourrait tout aussi bien déclarer un pointeur en tant que paramètre formel, tout en passant un tableau en tant que paramètre effectif :

cela reviendrait au même, puisque le tableau (paramètre effectif) sera transformé en pointeur lors de l'appel.

exemple :

```
void ma_fonction(short tab[4])
{  ...
}

int main(int argc, char** argv)
{  short t[4];
   ...
   ma_fonction(t);
}
```

Ici, on déclare un tableau de 4 entiers `short` dans la fonction `main`. Lors de l'appel, on l'expression `t` qui doit être évaluée. Comme aucun des opérateurs mentionnés précédemment (`&` et `sizeof`) n'est utilisé, le tableau `t` est remplacé par un pointeur sur son premier élément. On passe donc du type `short[4]` au type `short*`. La fonction est supposée recevoir un tableau de 4 entiers `short`, mais en réalité le compilateur a changé cette déclaration en un paramètre de même nom mais de type `short*`. Comme la fonction reçoit un `short*` lors de l'appel de la fonction `main`, ces types sont consistants. Tout se passe comme si on avait directement déclaré dans notre code source une fonction d'en-tête `void ma_fonction(short* tab)`.

10.3.3 Tableaux multidimensionnels

Considérons maintenant des tableaux à plusieurs dimensions. Pour eux, la substitution ne concerne que la première dimension. Soit la matrice $M \times N$ suivante :

```
short m[M][N];
```

Le type de `m` est évidemment `short[M][N]`. Mais si `m` apparait dans une expression (sans `&` ni `sizeof`), alors elle sera remplacée par un pointeur de type `short (*)[N]`, autrement dit : pointeur sur un tableau de N entiers courts. L'expression `&m` aura pour type `short (*)[M][N]`, i.e. pointeur sur une matrice de $M \times N$ entiers courts.

exemple : on met à jour l'exemple précédent de façon à utiliser une matrice :

```
void ma_fonction(short tab[4][5])
{  ...
}

int main(int argc, char** argv)
{  short t[4][5];
   ...
   ma_fonction(t);
}
```

Alors l'en tête de la fonction pourrait tout aussi bien ne pas préciser la première dimension, qui est superflue :

```
void ma_fonction(short tab[][5])
```

Ou même directement déclarer un pointeur sur un tableau :

```
void ma_fonction(short (*tab)[5])
```

Remarque : notez bien les parenthèses autour de `*tab`. Sans elles, on obtient pas un pointeur sur un tableau de `short`, mais plutôt un tableau de pointeurs sur des `shorts` (cf. section 10.3.4).

Considérons les instructions suivantes :

```
printf("%p\n",m)
```

On sait que la variable `m` est remplacée par une adresse de type `short (*)[N]`. Cette adresse est celle de la matrice, donc de sa première ligne `m[0]`, donc du premier élément de cette ligne. Au final, on obtient l'adresse du premier élément du tableau.

```
printf("%p\n",*m)
```

L'application de l'opérateur `*` à `m` permet d'obtenir une expression de type `short[N]`. On sait qu'un type tableau est remplacé par un type pointeur sur les mêmes données, donc ici il sera remplacé par un pointeur de type `short*`. Donc on va afficher l'adresse de la première ligne du tableau, i.e. celle de son premier élément, i.e. la même que tout à l'heure. Par contre, le type est différent : `short*` au lieu de `short(*)[N]`. Ceci confirme que `short[][]` n'est pas équivalent à `short**`, comme on pourrait le croire *a priori* : si c'était le cas, les adresses obtenues avec ces deux expressions ne pourraient pas être les mêmes (sauf à pointer sur sa propre adresse).

```
printf("%p\n",**m)
```

La deuxième application de `*` transforme le `short*` en `short`, et on obtient finalement la valeur du tout premier élément du tableau. L'expression `**m` envoie la valeur de cet élément non pas parce qu'une matrice est un pointeur de pointeur, mais bien à cause du système de substitution transformant les tableaux en pointeurs lors de l'évaluation d'expressions.

10.3.4 Tableaux de pointeurs

Quand nous avons étudié les tableaux (section 6), nous avons vu qu'ils pouvaient contenir n'importe quel type de données, tant qu'elles étaient homogènes (toutes du même type). On peut en déduire qu'il est possible de créer des *tableaux de pointeurs*, i.e. des tableaux dont les éléments sont des pointeurs.

exemple : tableau de pointeurs sur des `char`

```
char *tab[4];
```

On obtient un tableau capable de contenir 4 adresses.

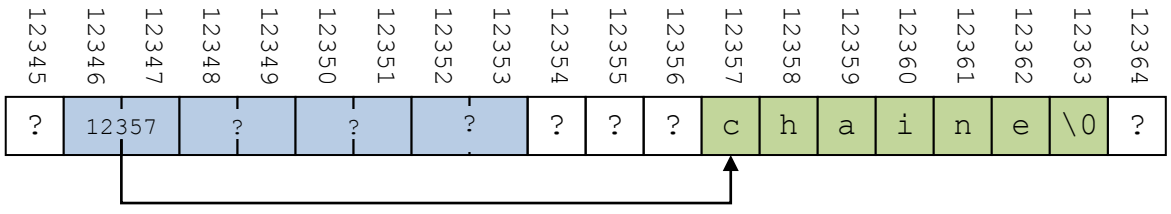
L'accès aux différents éléments du tableau se fait normalement, en respectant à la fois les principes d'accès des tableaux et des pointeurs.

exemple :

```
char c1[] = "chaine1";
*tab[0] = c1;
printf("la chaine 0 est : %s",tab[0]);
printf("entrez la chaine 1 : ");
scanf("%s",tab[1]);
```

Ici, on déclare une chaine de caractères initialisée à `"chaine1"`. Puis, on fait pointer le premier élément du tableau sur cette chaîne. Si on affiche cet élément `tab[0]`, on obtient bien `"chaine1"`. Dans la suite du programme, on initialise l'élément suivant du tableau (i.e. `tab[1]`) via une saisie de l'utilisateur.

En mémoire, on obtient une organisation de la forme suivante (toujours en supposant qu'une adresse prend deux octets sur le système considéré) :



Ce tableau permet de représenter une séquence de chaînes de caractères. On pourrait alternativement utiliser une matrice $N \times M$ de caractères, dans laquelle chaque ligne correspondrait à une chaine. Mais il y aurait quelques différences importantes :

- Les chaines seraient toutes placées en mémoire de façon *contiguë*, puisque c'est le principe du tableau (en particulier de la matrice).

- La longueur maximale serait la *même* pour toutes les chaînes (M).

Par comparaison, avec un tableau de pointeur on peut faire référence à des chaînes de caractères dont les longueurs peuvent être différentes. Ceci est dû au fait que les pointeurs évitent la contrainte de contiguïté propre aux tableaux. Cependant, en raison de l'utilisation d'un tableau pour stocker les pointeurs, on reste contraint sur le nombre maximal de chaînes que l'on peut référencer (ce qui est aussi le cas si on utilise une matrice) : N. Ce problème ne peut être résolu qu'en utilisant l'allocation dynamique de mémoire (section 11).

11 Allocation dynamique de mémoire

L'allocation dynamique permet de créer des tableaux dont la taille n'a pas besoin d'être connue au moment de la compilation. Les tableaux obtenus ne sont pas vraiment des variables, dans le sens où aucun identificateur ne leur est directement associé. On les manipule via des pointeurs, et la notion d'allocation dynamique est donc extrêmement liée à celle de pointeur.

Dans cette section, nous expliquons l'intérêt de l'allocation dynamique, et nous décrivons les principales fonctions du C relatives à cette technique.

11.1 Présentation

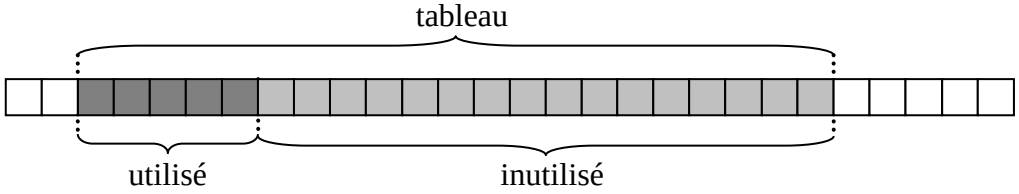
Quand on veut utiliser un tableau, nous devons décider de sa taille *a priori*, c'est-à-dire lorsque l'on écrit le programme. En effet, on en a obligatoirement besoin pour sa déclaration. On parle alors d'allocation *statique* de la mémoire, dans le sens où la taille du tableau est alors définitivement fixée : on ne peut plus la modifier. Comme on l'a vu précédemment, le tableau est stocké dans la pile ou dans le segment de données, en fonction de sa persistance.

Mais parfois, on ne connaît pas la taille du tableau à l'avance, car elle dépend d'informations extérieures telles que le comportement de l'utilisateur. Par exemple, supposons que l'on veuille stocker les notes de tous les étudiants d'une promotion dans un tableau d'entiers. Alors, il est nécessaire de connaitre le nombre d'étudiants à l'avance, car elle va directement déterminer la taille du tableau. On ne peut pas demander à l'utilisateur de saisir ce nombre, puis déclarer un tableau de la bonne taille, car la taille est alors déterminé à l'exécution, et non pas à la compilation (or elle doit être connue *avant* la compilation).

On doit malgré tout fixer une taille :

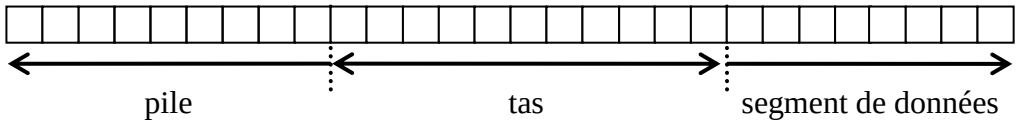
- Si elle est trop *petite*, on ne pourra pas stocker toutes les données.
- Si elle est trop grande, on va gaspiller de l'espace mémoire, car il sera réservé (et ne pourra donc pas être utilisé pour d'autres variables) mais inutilisé.

Par exemple, supposons qu'on a déclaré un tableau de taille 21, mais qu'on n'a finalement que 5 étudiants dans la promotion. Alors 16 éléments sont inutilisés :



L'allocation dynamique permet précisément de résoudre ce problème-là. Son rôle est de réserver une zone de mémoire dont la taille peut être spécifiée à l'exécution (et non pas à la compilation).

Les fonctions de gestion dynamique de la mémoire permettent d'allouer et de désallouée de la mémoire pendant de l'exécution. Cependant, la mémoire utilisée n'est prise ni dans la pile ni dans le segment de données, mais dans le tas.



La notion de *portée* des variables (variable globale ou locale) n'a pas cours avec la mémoire allouée dynamiquement, car celle-ci est manipulée uniquement grâce à des pointeurs. En d'autres termes, toutes les données du tas sont accessibles globalement, à condition bien sûr d'en connaître l'adresse (i.e. d'avoir accès à un pointeur sur cette mémoire).

La notion de *persistance* n'est pas pertinente non plus avec l'allocation dynamique, car c'est le programmeur qui décide manuellement quand faire l'allocation et quand faire la désallocation.

On peut relever les différences suivantes entre mémoire allouée dynamiquement et variables classiques :

- Emplacement : dans le tas, au lieu de la pile ou du segment de données ;
- Identificateur : aucun ;
- Type : aucun (initialement, mais une conversion explicite permet d'en donner un) ;
- Valeur : pas définie par défaut (comme pour une variable classique) ;
- Portée : équivalent à une portée globale
- Persistance : de l'allocation à la désallocation.

Un certain nombre de fonctions standard permettent de manipuler la mémoire allouée dynamiquement. La plus importante est `malloc`, qui effectue une allocation de base, les autres étant des variantes ou des fonctions secondaires.

11.2 Allocation simple

La fonction `malloc` (memory allocation) permet d'allouer dynamiquement une certaine quantité de mémoire. Son en-tête est la suivante :

```
void* malloc(size_t size)
```

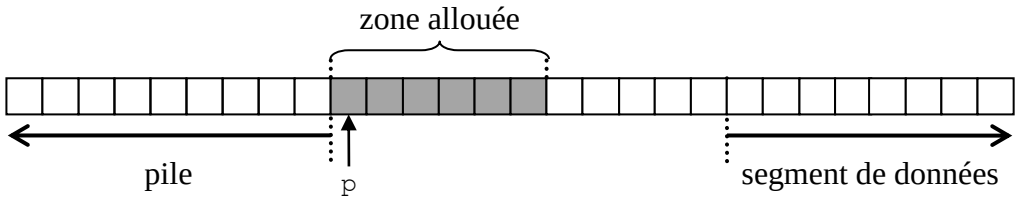
La fonction prend en paramètre un entier `size` indiquant la quantité de mémoire à allouer, exprimée en octets. Remarquez qu'on utilise le type `size_t`, déjà vu pour `sizeof` (cf. section 4.2.6).

Elle alloue une zone de mémoire de la taille spécifiée, et renvoie l'adresse du premier octet de cette zone. Remarquez que ce pointeur n'est pas différentié, dans le sens où son type est `void*`. Cela signifie qu'il est une adresse, mais qu'on ne sait pas sur quel type il pointe. En pratique, quand `malloc` est appelée, on convertira systématiquement ce `void*` en un type plus explicite, tel que `int*`, `float*`, etc., en fonction du besoin.

On utilise un pointeur pour récupérer l'adresse renvoyée par `malloc` et pouvoir utiliser par la suite la mémoire allouée.

exemple : on veut allouer une zone de 3 entiers `short`, i.e. 6 octets au total :

```
short *p;
p = (short*)malloc(3*2);
```



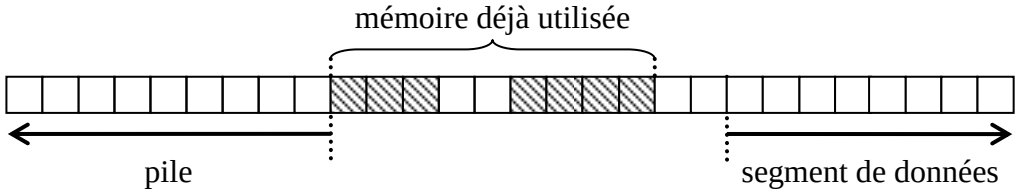
Pour éviter les erreurs lorsqu'on exprime la taille de la mémoire à allouer, on n'utilisera pas de constante littérale comme c'est le cas dans l'exemple ci-dessus avec le 2, qui représente la taille d'un `short`. On se servira plutôt de l'opérateur `sizeof`, qui renvoie la taille en octets d'une donnée du type spécifié en paramètre. L'exemple précédent devient donc :

```
p = (short*)malloc(3*sizeof(short));
```

Bien sûr, si on ne voulait qu'un seul short, on ferait à la place :

```
p = (short*)malloc(sizeof(short));
```

La fonction `malloc` tente d'allouer un espace *contigu*. Il est possible qu'elle échoue s'il ne reste pas assez de mémoire libre dans le tas. Par exemple, si on demande une zone de la même taille que dans l'exemple précédent, mais avec l'occupation du tas suivante :



En cas d'échec, il est nécessaire que le programme soit averti que l'allocation dynamique a échoué, sinon des erreurs pourraient apparaître. Pour cela, la fonction `malloc` renvoie la valeur `NULL` en cas d'erreur. Rappelons que cette constante représente une adresse invalide, i.e. en tant que valeur de pointeur, on peut l'interpréter comme un pointeur ne pointant sur rien du tout.

Par conséquent, il faut *toujours* tester la valeur renvoyée par `malloc` afin de pouvoir traiter une éventuelle erreur. On peut procéder de la façon suivante :

```
p = (short*)malloc(N*sizeof(short));
if(p == NULL)
   // traitement de l'erreur
else
   // traitement normal
```

Cependant, l'écriture suivante est plus compacte, et c'est celle que nous utiliserons. Elle consiste à déplacer l'affectation de `p` dans le test du `if` :

```
if((p = (short*)malloc(N*sizeof(short))) == NULL)
   // traitement de l'erreur
else
   // traitement normal
```

Rappel : la valeur et le type d'une expression d'affectation correspondent à la valeur et au type de la valeur affectée (éventuellement après conversion implicite). Donc ici, l'expression comparée à `NULL` est `p=(short*)malloc(N*sizeof(short))`, qui est elle-même égale à `(short*)malloc(N*sizeof(short))`, i.e. le pointeur renvoyé par `malloc`.

Remarque : la méthode consistant, dans une fonction, à renvoyer une valeur particulière indiquant une erreur est très courante en C. Nous l'utiliserons plus tard dans d'autres situations, notamment la manipulation de fichiers.

11.3 Autres fonctions

La fonction `calloc` est **identique** à `malloc`, à la différence des paramètres. Le `c` dans son nom correspond à count, i.e. nombre d'éléments.

```
void* calloc(size_t element_count, size_t element_size)
```

Tout comme `malloc`, elle prend un paramètre de type `size_t` correspondant à une taille en octets. Cependant, elle prend également un second paramètre entier, un facteur correspondant à un nombre d'élément. Il va multiplier la taille passée en premier paramètre, pour donner la taille totale de la zone à réserver.

exemple : réserver un tableau de *N* valeurs de type `short`

```
if((p = (short*)calloc(N,sizeof(short))) == NULL)
   // traitement de l'erreur
else
   // traitement normal
```

Remarque : on peut aussi considérer que `calloc` est qu'une généralisation de `malloc`, i.e. que `malloc` est un cas particulier de `calloc` dans lequel `element_count=1`.

La fonction `realloc` permet de réallouer une zone de mémoire qui avait été précédemment allouée par `malloc`, `calloc` ou `realloc`. Son en-tête est le suivant :

```
void* realloc(void *p, size_t size)
```

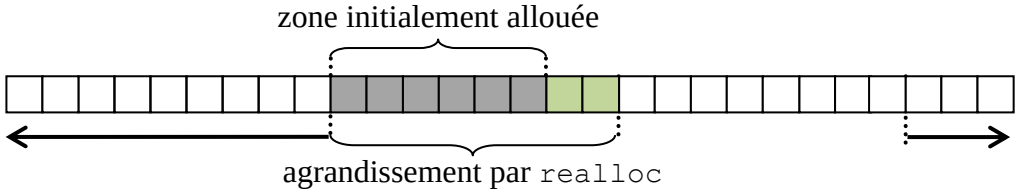
Le premier paramètre correspond à l'adresse de la zone mémoire à réallouer. Notez que son type est `void*`, ce qui signifie qu'on peut passer n'importe quel type de pointeur. Le second paramètre est le même que pour `malloc`, et il représente la nouvelle taille de la zone.

La fonction renvoie l'adresse de la nouvelle zone, qui *peut* être la même que celle passée en paramètre, ou bien `NULL` en cas d'erreur.

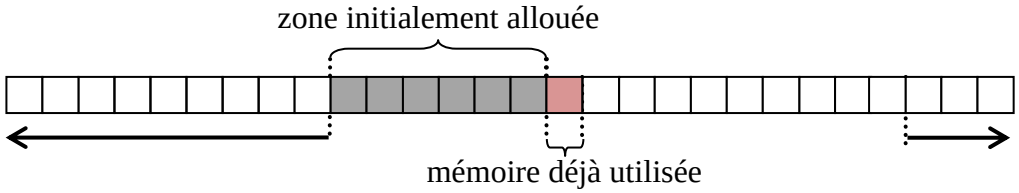
Si on décide d'agrandir de 1 élément la taille de notre tableau de 3 entiers `short` défini dans l'exemple précédent, il est nécessaire de faire une réallocation afin de disposer d'une zone de mémoire suffisamment grande (sinon, on risque le débordement de tableau).

```
if((p = (short*)realloc(p, 4*sizeof(short))) == NULL)
   // traitement de l'erreur
else
   // traitement normal
```

Dans la mesure du possible, `realloc` tente d'*agrandir* la zone *existante* :

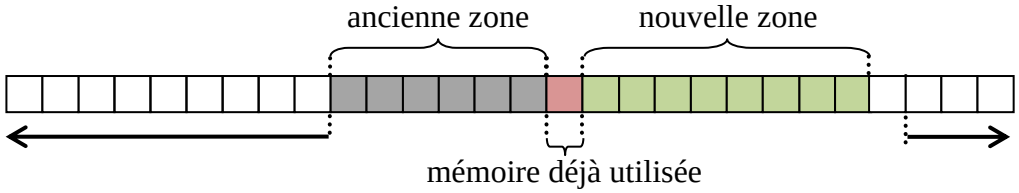


Mais parfois, il est impossible de simplement agrandir la zone existante, car les octets suivants ont déjà été réservés lors d'une allocation antérieure :



Dans ce cas là, `realloc` cherche une zone libre suffisamment grande dans le tas :

- Si elle n'en trouve pas, l'allocation échoue et la fonction renvoie `NULL` ;
- Sinon :
  - Les données contenues dans l'ancienne zone sont automatiquement recopiées dans la nouvelle.
  - L'ancienne zone est désallouée.
  - La fonction renvoie l'adresse de la nouvelle zone.



Remarque : si l'adresse passée en paramètre à la fonction vaut `NULL`, `realloc` se comporte exactement comme `malloc`, et alloue donc une nouvelle zone mémoire. De ce point de vue, on peut aussi considérer `realloc` comme une généralisation de `malloc`.

Enfin, la fonction `free` permet de libérer une zone de mémoire qui avait été allouée par `malloc`, `calloc` ou `realloc`.

```
void free(void *p)
```

Il suffit de passer en paramètre l'adresse de la zone à libérer. Ce paramètre est de type `void*`, donc il peut s'agit de n'importe quel type de pointeur.

exemple : pour libérer le tableau alloué dynamiquement dans l'exemple précédent :

```
free(p);
```

Comme `realloc`, `free` ne peut s'appliquer qu'à des pointeurs indiquant des zones mémoires allouées dynamiquement par `malloc`, `calloc` ou `realloc`. La fonction `free` ne peut pas être utilisée sur des variables allouée statiquement.

Quand écrit un programme, il faut faire particulièrement attention à bien libérer les zones allouées dynamiquement qui ne sont plus utilisées. C'est particulièrement vrai pour des programmes grands et/ou complexes. Ceci se fait automatiquement pour les variables locales, qui sont désallouées à la fin du bloc qui les contient, mais ce n'est pas le cas pour les zones allouées dynamiquement. La fonction free est donc très importante, car c'est elle qui permet de ne pas remplir le tas, ce qui provoquerait une erreur d'exécution appelée *fuite de mémoire*. Ceci se produit quand un programme occupe de plus en plus de mémoire, jusqu'à ce qu'il n'y en ait plus.

11.4 Exercices

Écrivez une fonction `saisis_entier` qui saisit et renvoie un entier. La zone mémoire où l'entier est stocké doit être réservée dans la fonction. Donnez également la fonction `main` permettant d'appeler la fonction. *

Faites une première version avec un résultat passé par valeur (`return`) et une seconde version avec un résultat passé par adresse.

- Par valeur :

```
int saisis_entier()
{  int *p,resultat;
   if((p = (int*)malloc(sizeof(int))) == NULL)
      printf("saisis_entier : erreur lors du malloc\n");
   else
   {  printf("entrez l'entier : ");
      scanf("%d",p);
      resultat = *p;
      free(p);
   }
   return resultat;
}
int main()
{  int x;
   x = saisit_entier();
   printf("%d\n",x);
}
```

- Par adresse :

```
void saisis_entier(int **x)
{  int *p;
   if((p = (int*)malloc(sizeof(int))) == NULL)
      printf("saisis_entier : erreur lors du malloc\n");
   else
   {  printf("entrez l'entier : ");
      scanf("%d",p);
      *x = p;
   }
}
int main()
{  int *x;
   saisit_entier(&x);
   printf("%d\n",*x);
}
```

12 Fichiers

Un fichier est une représentation de données stockées sur un support persistant :

Fichier : ensemble de données associées à un nom et un emplacement sur un support de stockage (disque dur, clé USB, disque optique, etc.).

En langage C, un fichier n'est pas structuré par nature, il est vu simplement comme une séquence d'octets. C'est au programmeur de donner une structure à ces données. De plus, le langage offre deux façons d'accéder aux fichiers :

- Fonctions de *bas niveau* :
  - Les fonctions correspondent à des routines du système d'exploitation (OS).
  - Ce type d'accès est donc très dépendant de l'OS utilisé, et par conséquent les programmes utilisant ces fonctions ne sont pas portables.
- Fonctions de **haut** niveau :
  - Fonctions indépendantes de l'OS, et donc programmes portables.

Dans ce cours nous n'utiliserons que les fonctions de haut niveau. Celles de bas niveau seront vues en cours de systèmes d'exploitation.

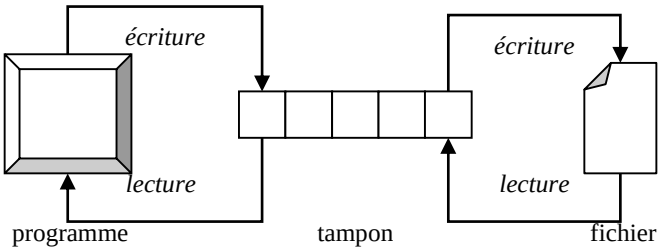
12.1 Structure `FILE`

L'accès aux fichiers est *bufférisé*, c'est-à-dire qu'on utilise un *tampon* (buffer) lors des écritures et des lectures.

Mémoire tampon : zone de mémoire utilisée pour y manipuler des données de façon temporaire, avant de les placer ailleurs.

Quand on veut écrire dans un fichier, on écrit d'abord dans une zone spécifique de la mémoire appelée tampon. Quand cette zone est pleine, le système écrit son contenu dans le fichier, puis on recommence à remplir le tampon. Cette méthode permet de limiter le nombre d'accès en écriture dans le fichier, car ces accès sont relativement coûteux en termes de ressources (surtout en temps).

De même, lorsqu'on demande une lecture, les données lues dans le fichier sont copiées dans un tampon. On ne lit donc pas directement dans le fichier, mais dans la mémoire tampon.



La manipulation des fichiers n'est pas directe, on utilise pour cela une représentation sous la forme d'une variable de type structure `FILE`, qui est défini dans `stdio.h`.

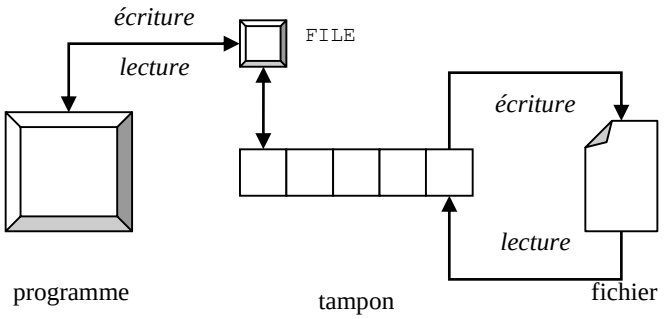
Remarque : il ne pas confondre le mot anglais *file*, qui signifie en français *fichier*, et le mot français *file* de données, qui se traduit en anglais par *queue* ou FIFO (cf. section 16).

La structure `FILE` contient différentes informations concernant le fichier, en particulier :

- Un pointeur vers l'adresse de début du tampon ;
- Un pointeur vers la position courante dans le tampon ;
- La taille du tampon.

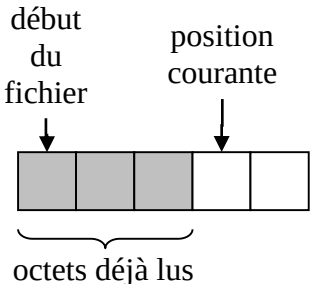
On n'a jamais besoin d'initialiser manuellement une variable de type `FILE` : ce sont les fonctions relatives à la gestion des fichiers qui s'en chargent. Lorsqu'on ouvre un fichier, on

récupère un *pointeur* vers une variable de type `FILE`, qui est utilisée pour lire ou écrire dans le fichier.



On accède de façon *séquentielle* au contenu du fichier (qui est une suite d'octets). Le pointeur de position courante indique quel sera le prochain octet accédé.

exemple : lecture



12.2 Ouverture/fermeture

Avant de pouvoir lire ou écrire dans un fichier, il est nécessaire de l'ouvrir. C'est cette *ouverture* qui permet d'initialiser une variable `FILE` pour représenter le fichier. L'ouverture est réalisée grâce à la fonction `fopen` dont l'en-tête est :

```
FILE* fopen(char *nom_fichier, char *mode);
```

La fonction renvoie un pointeur vers une structure `FILE`, ou bien `NULL` en cas d'erreur. Le paramètre `nom_fichier` est une chaîne de caractères contenant le *chemin* du fichier à ouvrir, i.e. son emplacement et son nom. Le paramètre `mode` est également une chaine de caractères, qui indique comment le fichier doit être ouvert. Il existe de nombreux modes, mais dans le cadre de ce cours, nous utiliserons les plus simples :

- `"r"` : lecture.
  - Renvoie `NULL` si le fichier n'existe pas.
- `"w"` : écriture.
  - Si le fichier existe : son contenu est réinitialisé.
  - Sinon : il est créé
- `"a"` : ajout de données à la fin du fichier.
  - Fichier créé s'il n'existe pas déjà.
  - Sinon, permet d'écrire de nouvelles données après les données existantes.

exemples : ouverture en lecture d'un fichier appelé `mon_fichier.txt` :

```
FILE *fp;
if((fp = fopen("mon_fichier.txt", "r")) == NULL)
{   //traitement en cas d'erreur
    ...
}
else
{   // traitement normal
    ...
}
```

La *fermeture* d'un fichier est l'opération réciproque de l'ouverture. Elle permet de notamment de libérer les ressources que le système avait réservé pour manipuler le fichier lors de l'ouverture, comme par exemple la variable de type `FILE` associée. En cas d'écriture, la fermeture permet aussi d'écrire dans le fichier les données encore présentes dans le buffer.

Elle est réalisée par la fonction `fclose`, dont l'en-tête est :

```
int fclose(FILE *fichier);
```

Le paramètre `fichier` est un pointeur vers une structure `FILE` représentant le fichier à fermer. Bien sûr, ce fichier doit avoir préalablement été ouvert par `fopen`. Si la fonction réussit, elle renvoie la valeur 0. Si elle échoue, elle renvoie la constante `EOF` (End of File). Cette constante est fréquemment utilisée dans les fonctions manipulant des fichiers, pour indiquer qu'un comportement anormal s'est produit.

exemple : fermeture du fichier précédemment ouvert

```
if((fclose(fp)) == EOF)
{   //traitement en cas d'erreur
    ...
}
else
{   // traitement normal
    ...
}
```

Il existe différentes façons d'accéder au contenu d'un fichier :

- Accès *non-formaté* : on lit/écrit uniquement des caractères ;
- Accès *formaté* : on lit/écrit des données typées ;
- Accès *par bloc* : on manipule des séquences d'octets, indépendamment de leur type.

Le reste de cette section est dédié à leur description.

12.3 Lecture/écriture non-formatées en mode caractère

On distingue deux sortes d'accès on-formaté :

- Mode *caractère* : on lit/écrit un seul caractère à la fois ;
- Mode *chaîne* : plusieurs caractères (une chaine de caractère) simultanément.

Commençons par le premier mode, qui est aussi le plus simple.

La lecture est réalisée grâce à la fonction `fgetc` :

```
int fgetc(FILE *fichier);
```

Le paramètre est un pointeur vers une structure `FILE` représentant un fichier déjà ouvert. La fonction renvoie le caractère lu, ou bien la constante `EOF` en cas d'erreur.

La valeur lue correspond à un `unsigned char`. On peut donc se demander pourquoi le type de retour de la fonction est `int`. Il se trouve que la constante `EOF` a généralement la valeur −1. Cette valeur négative ne peut pas être codée par un `unsigned char`, d'où l'utilisation d'un `int`.

exemple : on affiche le contenu d'un fichier déjà ouvert, et représenté par le pointeur `fp`

```
int c;
while((c=fgetc(fp)) != EOF)
    putchar(c);
```

Remarquez que l'on utilise dans la condition du `while` la même écriture compacte que précédemment dans le `if` du `malloc` (section 11.3). Si on voulait détailler le code source, on pourrait écrire de façon équivalente :

```
int c = fgetc(fp);
while(c!=EOF)
{   putchar(c);
```

```
    c = fgetc(fp);
}
```

Lorsqu'arrive à la fin du fichier, le programme tente de lire la valeur suivante, ce qui est impossible. Cela déclenche donc une erreur, et `fgetc` renvoie `EOF`. Le problème est que la valeur `EOF` peut aussi est renvoyée à cause d'une autre erreur, par exemple si l'accès au fichier n'est plus possible. Il est donc nécessaire de pouvoir distinguer ces deux cas.

Pour être sûr que l'erreur correspond bien à une fin de fichier, on utilisera la fonction `feof` :

```
int feof(FILE *fichier);
```

Elle renvoie une valeur non-nulle si la position courante correspond à la fin du fichier représenté par le pointeur. Sinon, elle renvoie la valeur 0.

exemple : révision de l'exemple précédent en utilisant `feof` :

```
int c;
while((c=fgetc(fp)) != EOF)
    putchar(c);
if(!feof(fp))
{   // traitement de l'erreur
    ...
}
```

L'écriture en mode caractère est réalisée grâce à la fonction `fputc` :

```
int fputc(int caractere, FILE *fichier);
```

La fonction prend un `int` en paramètre, mais il est convertit en `unsigned char`. Ce paramètre est de type `int` uniquement par souci de consistance avec `fgetc`. En cas de succès, la fonction renvoie la valeur qui a été écrite. Sinon, elle renvoie `EOF`.

exemple : on écrit dans un fichier

```
int c=5;
if((fputc(c,fp)) == EOF)
{   // traitement de l'erreur
    ...
}
```

Encore une fois, remarquez l'utilisation de l'écriture compacte dans la condition du `if`.

12.4 Lecture/écriture non-formatées en mode chaîne

Au lieu de lire un seul caractère à la fois, on peut manipuler des chaînes de caractères. La lecture d'une chaîne de caractères est effectuée grâce à la fonction `fgets` dont l'en-tête est :

```
char* fgets(char *chaine, int nombre, FILE *fichier);
```

Le pointeur `chaine` désigne le tampon, c'est-à-dire l'emplacement en mémoire où la chaîne lue doit être stockée (par exemple : un tableau de caractères). L'entier `nombre` indique le nombre maximal de caractères à lire. Le pointeur `fichier` désigne le fichier à lire (qui a été préalablement ouvert, bien entendu).

La fonction renvoie un pointeur vers le tampon, ou bien `NULL` en cas d'erreur. À noter que la fonction renvoie également `NULL` en cas de fin de fichier : comme pour `fgetc`, la fonction `feof` permet de distinguer une erreur d'une fin de fichier normale.

La fonction tente de lire `nombre-1` caractères dans le fichier. Elle rajoute automatiquement le caractère `'\0'` à la fin de la chaine. La lecture est interrompue si la fonction rencontre le caractère `'\n'` (le `'\n'` est quand même recopié dans le tampon).

exemple :

Soit un fichier contenant les lignes suivantes :

```
université
de
galatasaray
```

Supposons qu'on utilise l'instruction suivante :

```
char tampon[12];
while(fgets(tampon,12,fp) != NULL)
   printf("%s",tampon);
```

alors la première chaine obtenue sera :

```
{'u','n','i','v','e','r','s','i','t','é','\n','\0'}
```

la deuxième sera :

```
{'d','e','\n','\0'}
```

et la troisième sera :

```
{'g','a','l','a','t','a','s','a','r','a','y','\0'}
```

L'écriture d'une chaîne est effectuée grâce à la fonction `fputs` :

```
int fputs(char *chaine, FILE *fichier);
```

Le pointeur `chaine` désigne la chaîne de caractères à écrire dans le fichier. La fonction ne recopie pas le caractère `'\0'` terminal. La fonction renvoie une valeur non-négative en cas de succès, ou bien `EOF` en cas d'erreur.

exemple : pour créer un fichier correspondant à l'exemple de `fgets`

```
if(fputs("universite\nde\ngalatasaray\n", fp) == EOF)
   // traiter l'erreur
```

12.5 Lecture/écriture formatées

Considérons maintenant le cas où on veut formater les données écrites dans les fichiers textes, i.e. obtenir une écrire qui convienne à leur type. Ceci est réalisé au moyen des fonctions `fscanf` et `fprintf`, respectivement dédiées à la lecture et à l'écriture.

Elles se comportent comme les fonctions `scanf` et `printf`, à la différence du fait qu'un paramètre supplémentaire, placé en première position, désigne le fichier dans lequel lire ou écrire :

```
int fscanf(FILE *fichier, char *format,...);
int fprintf(FILE *fichier, char *format,...);
```

Ces fonctions renvoient le nombre de variables lues ou écrites, ou bien `EOF` en cas d'erreur (ou de fin de fichier).

exemple : lire un `float` dans un fichier

```
float x;
if(fscanf(fp, "%f", &x) == EOF)
   // traiter l'erreur
```

exemple : écrire un `float` dans un fichier

```
float x=12.34
if(fprintf(fp, "%f", x) == EOF)
   // traiter l'erreur
```

12.6 Lecture/écriture par bloc

Lorsqu'on veut manipuler des grandes quantités de données, sans s'occuper de leur type, on utilise les accès par bloc. Ceci est réalisé grâce aux fonctions `fread` et `fwrite` :

```
size_t fread(void* tampon,size_t taille,size_t nombre,FILE *fichier);
size_t fwrite(void* tampon,size_t taille,size_t nombre,FILE *fichier);
```

Ces fonctions permettent de manipuler un certain `nombre` de blocs contenant `taille` octets. Le type `size_t` est défini dans `stdio.h`, il s'agit d'un entier non-signé. Ce type a déjà été utilisé, notamment avec l'opérateur `sizeof` (section 4.2.6) et la fonction `calloc` (section 11.3).

La fonction `fread` lit `taille*nombre` octets dans le fichier et les recopie dans le tampon. La fonction `fwrite` lit `taille*nombre` octets dans le tampon et les recopie dans le fichier.

Les deux fonctions renvoient le nombre de blocs lus/écrits : une valeur inférieure à `nombre` indique donc qu'une erreur est survenue au cours du traitement.

12.7 Exercices

1) Écrivez un programme qui saisit une chaine de caractères et l'écrit dans un fichier avec `fputc`.

```
int main()
{
    // on saisit la chaine
    char chaine[10];
    printf("entrez la chaine : ");
    scanf("%s",chaine);

    // on ouvre le fichier
    FILE *fp;
    if((fp = fopen("fichier.txt", "w")) == NULL)
    {  //traitement en cas d'erreur
       printf("main : erreur dans fopen\n");
       exit(EXIT_FAILURE);
    }

    // on écrit la chaîne
    int i=0;
    while(chaine[i] !='\0')
    {  if((fputc(chaine[i],fp)) == EOF)
       {   // traitement de l'erreur
           printf("main : erreur dans fputc\n");
          exit(EXIT_FAILURE);
       }
       i++;
    }

    // on ferme le fichier
    if((fclose(fp)) == EOF)
    {  // traitement de l'erreur
       printf("main : erreur dans fclose\n");
       exit(EXIT_FAILURE);
    }
}
```

2) Écrivez un programme qui réalise une copie d'un fichier existant par lecture/écriture en mode caractère.

```
int main()
{
    printf("Lecture dans le fichier : \n");

    // on ouvre le fichier source en lecture
    FILE *source;
    if((source = fopen("fichier.txt", "r")) == NULL)
    {  //traitement en cas d'erreur
       printf("main : erreur dans fopen\n");
       exit(EXIT_FAILURE);
    }

    // on ouvre le fichier cible en écriture
    FILE *cible;
    if((cible = fopen("copie.txt", "w")) == NULL)
    {  //traitement en cas d'erreur
       printf("main : erreur dans fopen\n");
       exit(EXIT_FAILURE);
    }

    // on lit la source caractère par caractère
    // en écrivant chaque caractère dans la cible
```

```
        int c;
        while((c=fgetc(source)) != EOF)
        {  if((fputc(c,cible)) == EOF)
           {  // traitement de l'erreur
              printf("main : erreur dans fputc\n");
              exit(EXIT_FAILURE);
           }
           putchar(c);
        }
        if(!feof(source))
        {  // traitement de l'erreur
           printf("main : erreur dans fgetc\n");
           exit(EXIT_FAILURE);
        }

        // on ferme les fichiers
        if((fclose(source)) == EOF)
        {  // traitement de l'erreur
           printf("main : erreur dans fclose\n");
           exit(EXIT_FAILURE);
        }
        if((fclose(cible)) == EOF)
        {  // traitement de l'erreur
           printf("main : erreur dans fclose\n");
           exit(EXIT_FAILURE);
        }

        printf("\n");
    }
```

13 Fonctions récursives

Les fonctions que l'on a étudiées jusqu'à présent sont qualifiées d'itératives, dans le sens où le traitement qu'elles implémentent se base soit sur une unique exécution de leur code, soit sur une répétition basée sur des boucles.

Il est cependant possible de définir une autre sorte de fonctions, qui réalisent une répétition basée sur la notion de récursivité : une telle fonction s'appelle elle-même (que ce soit directement ou indirectement).

Cette section s'attache à la description de ces fonctions, en commençant par le principe mathématique de récurrence, avant de passer aux aspects plus programmatiques.

13.1 Présentation

En *mathématiques*, le *principe de récurrence* est défini de la façon suivante :

Principe de **récurrence** : une propriété P_n est vraie pour tout $n \in \mathbb{N}$ si et seulement si les deux conditions suivantes sont respectées : P_0 est vraie et $P_n \rightarrow P_{n+1}$.

Considérons maintenant une fonction *informatique* $f(\theta)$, où θ représente l'ensemble des paramètres reçus par f : il y en a au moins 1, éventuellement plus. On suppose que la fonction a un type de retour autre que `void`. On dira que cette fonction f est définie de façon récursive si :

- Elle renvoie le résultat d'un calcul non récursif pour une (ou plusieurs) valeurs de ses paramètres, que l'on note θ_0 et que l'on appelle *valeur initiale*. On note $g_0(\ \ )$ ce calcul non-récursif. On a alors : $f(\theta_0) = g_0(\theta_0)$.
- Pour toute *autre* valeur, la fonction renvoie le résultat d'un calcul lui aussi non-récursif, et noté $g(\ \ )$, appliqué au résultat d'un calcul récursif. Ce calcul récursif consiste à appliquer la fonction f à une version modifiée du paramètre, que l'on note θ'. On a donc : $f(\theta) = g\big(f(\theta')\big)$.

La récursivité apparait dans le fait que dans le second cas, on utilise la fonction pour calculer son propre résultat. On oppose l'approche récursive à l'approche *itérative*, reposant sur l'utilisation des boucles, et qui a été étudiée précédemment.

exemple : soit la suite arithmétique suivante :

$$u_n = \begin{cases} a \text{ si } n = 0 \\ u_{n-1} + b \text{ sinon} \end{cases}$$

- On peut l'implémenter de façon *itérative* :

```
int suite_u(int n)
{  int i,resultat=A;
   for(i=1;i<=n;i++)
      resultat=resultat+B;
   return resultat;
}
```

- Mais il est plus naturel de le faire de façon récursive :

```
1  int suite_u(int n)
2  {   int resultat;
3      if(n==0)
4         resultat = A;
5      else
6         resultat = suite_u(n-1) + B;
7      return resultat;
8  }
```

Les correspondances entre cette fonction et la définition précédente sont :

- f correspond à `suite_u` (ligne 1);

- θ_0 correspond à `n=0` (ligne 3) ;
- $g_0(\ \ )$ est définie par $g_0(x) = A$ (ligne 4);
- La modification du paramètre est : $\theta' = \theta - 1$ (ligne 6 : `n-1`) ;
- $g(\ \ )$ est définie par $g(x) = x + B$ (ligne 6 : `suite(…)+B`).

Si g est l'identité, alors on parle de récursivité *terminale* :

Récursivité terminale : le résultat issu de l'appel récursif est renvoyé tel quel, sans être transformé.

exemples :

- La suite u de l'exemple précédent n'est pas récursive terminale, car on ajoute B au résultat de l'appel récursif.
- La fonction suivante, qui calcule le PGCD d'un nombre, est récursive terminale :

```
int pgcd(int x, int y)
{  int temp,resultat;
   if(x<y)
      resultat = pgcd(y,x);
   else if(y == 0)
      resultat = x;
   else
      resultat = pgcd(y,x%y);
   return resultat;
}
```

La particularité des fonctions récursives *terminales* est qu'elles peuvent être traduites directement en programmes itératifs, grâce à l'utilisation de boucles.

```
int pgcd(int x, int y)
{  int temp;
   if(x<y)
   {  temp = x;
      x = y;
      y = temp;
   }
   while(y!=0)
   {  temp = x%y;
      x = y;
      y = temp;
   }
   return x;
}
```

Et inversement, tout programme itératif peut être transformé en une fonction récursive terminale.

13.2 Types de récursivité

On parle de récursivité *simple* quand la relation de récurrence ne fait référence qu'une seule fois à la fonction définie elle-même :

Récursivité simple : la fonction s'appelle elle-même une seule fois au maximum, sans appeler d'autre fonction récursive.

Ce cas correspond donc à celui traité dans la définition. La relation de récurrence avait été notée :

$$f(\theta) = g\big(f(\theta')\big)$$

On parle de récursivité *multiple* quand la relation de récurrence fait référence plusieurs fois à la fonction définie elle-même :

Récursivité multiple : la fonction s'appelle elle-même plus d'une fois au maximum, sans appeler d'autre fonction récursive.

L'*ordre* de la récursivité dépend du nombre d'occurrence de la fonction dans la relation de récurrence :

Ordre de la récursivité : nombre de fois que la fonction s'appelle elle-même au maximum.

La récursivité simple est d'ordre 1. Si une fonction s'appelle k fois, elle est d'ordre k. Dans le cas de la récursivité multiple, en général, chaque appel se fait en utilisant des valeurs de paramètres différents (mais ce n'est pas obligatoire). Notons $\theta'_1 \dots \theta'_k$ les différentes modifications apportées aux paramètres. Alors on peut noter la relation de récurrence de la façon suivante :

$$f(\theta) = g\big(f(\theta'_1), \dots, f(\theta'_k)\big)$$

Il faut bien noter que pour une récursivité multiple d'ordre k, le traitement représenté par g doit combiner le résultat provenant de k appels distincts.

exemple : soit la suite suivante :

$$u_n = \begin{cases} a \text{ si } n \in \{0,1\} \\ u_{n-1} + u_{n-2} + b \text{ sinon} \end{cases}$$

La fonction *récursive multiple* correspondante est :

```
1  int suite_u(int n)
2  {   int resultat;
3      if(n==0 || n==1)
4         resultat = A;
5      else
6         resultat = suite_u(n-1) + suite_u(n-2) + B;
7      return resultat;
8  }
```

Les correspondances entre cette fonction et la définition précédente sont :

- f correspond à `suite_u` (ligne 1);
- θ_0 appartient à $\{0,1\}$ (ligne 3) ;
- $g_0(\ \ )$ est définie par $g_0(x) = A$ (ligne 4) ;
- Les modifications du paramètre sont :
  - $\theta'_1 = \theta - 1$ (ligne 6 : `n-1`) ;
  - $\theta'_2 = \theta - 2$ (ligne 6 : `n-2`) ;
- $g(\ \ )$ est définie par $g(x_1, x_2) = x_1 + x_2 + B$ (ligne 6).

Il est aussi possible de définir des relations récursives indirectes. C'est le cas quand une fonction ne s'appelle pas elle-même directement, mais appelle une autre fonction qui elle-même appelle la première. On parle alors de récursivité *croisée* :

Récursivité croisée : la fonction s'appelle *indirectement*, via une ou plusieurs autres fonctions.

La récursivité croisée peut être simple ou multiple, en fonction de l'ordre de la relation de récurrence. Supposons qu'on a une récursivité croisée simple impliquant deux fonctions f et p. Notons g le traitement effectué par f dans le cas récursif et q celui effectué par p. On peut alors décrire formellement la relation de récurrence de la façon suivante :

$$f(\theta) = g\big(p(\theta')\big) \,;\, p(\theta') = q\big(f(\theta'')\big)$$

exemple : soit les deux suites suivantes :

$$u_n = \begin{cases} a_u \text{ si } n = 0 \\ 3u_{n-1} + 2v_{n-1} + b_u \text{ sinon} \end{cases} ; v_n = \begin{cases} a_v \text{ si } n = 0 \\ u_{n-1} + 4v_{n-1} + b_v \text{ sinon} \end{cases}$$

La fonction récursive correspondant à u est :

```
1  int suite_u(int n)
2  {   int resultat;
3      if(n==0)
4         resultat = A_U;
5      else
```

```
6         resultat = 3*suite_u(n-1) + 2*suite_v(n-1) + B_U;
7      return resultat;
8  }
```

La fonction récursive correspondant à v est :

```
1  int suite_v(int n)
2  {   int resultat;
3      if(n==0)
4         resultat = A_V;
5      else
6         resultat = suite_u(n-1) + 4*suite_v(n-1) + B_V;
7      return resultat;
8  }
```

On obtient une récursivité croisée et multiple (puisqu'on appelle deux fonctions récursives lignes 6).

Les correspondances entre ces fonctions et la définition précédente sont :

- f et p correspondent respectivement à `suite_u` et `suite_v` (lignes 1);
- Pour les deux, on a $\theta_0 = 0$ (lignes 3 : `n=0`) ;
- $g_0(\ )$ et $q_0(\ )$ sont respectivement définies par $g_0(x) = A_u$ et $q_0(x) = A_v$ (lignes 4) ;
- Les modifications du paramètre pour f sont :
  - $\theta_1' = \theta - 1$ (ligne 6 : `n-1`) ;
  - $\theta_2' = \theta - 1$ (ligne 6 : `n-2`) ;
- Les modifications du paramètre pour p sont :
  - $\theta_1'' = \theta' - 1$ (ligne 6 : `n-1`) ;
  - $\theta_2'' = \theta' - 1$ (ligne 6 : `n-2`) ;
- $g(\ )$ est définie par $g(x_1, x_2) = 3x_1 + 2x_2 + B_u$ (ligne 6) ;
- $q(\ )$ est définie par $q(x_1, x_2) = x_1 + 4x_2 + B_v$ (ligne 6).

13.3 Arbre des appels

Le concept d'*arbre des appels* permet de donner une représentation graphique de la récursivité, et ainsi de mieux la comprendre à travers sa visualisation :

Arbre des appels : représentation graphique des appels d'une ou plusieurs fonctions s'auto- ou s'inter-appelant, avec les valeurs des paramètres et les résultats renvoyés.

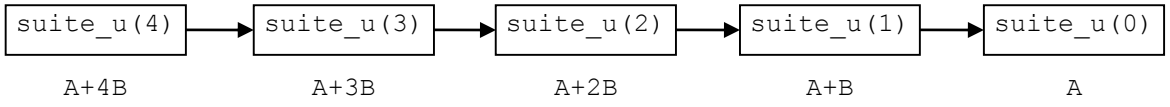
Par s'auto-appeler, on veut dire que la fonction s'appelle elle-même. Par s'inter-appeler, on veut dire que plusieurs fonctions s'appellent les unes les autres.

Si jamais la récursivité est simple, alors l'arbre est *linéaire*, c'est-à-dire qu'il ne contient qu'une seule branche. Si elle est multiple, tout dépend du nombre d'appels récursifs effectués.

exemples :

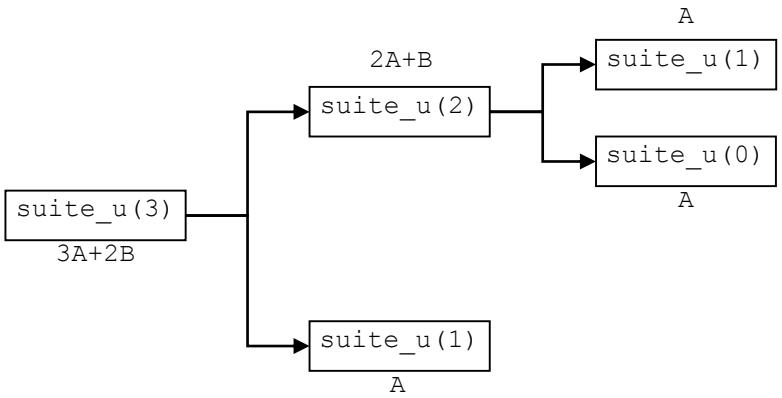
Pour la fonction de l'exemple de la sous-section 13.1, on obtient un arbre des appels *linéaire*. Ainsi, pour $n = 4$ on a :

$$u_n = \begin{cases} a \text{ si } n = 0 \\ u_{n-1} + b \text{ sinon} \end{cases}$$



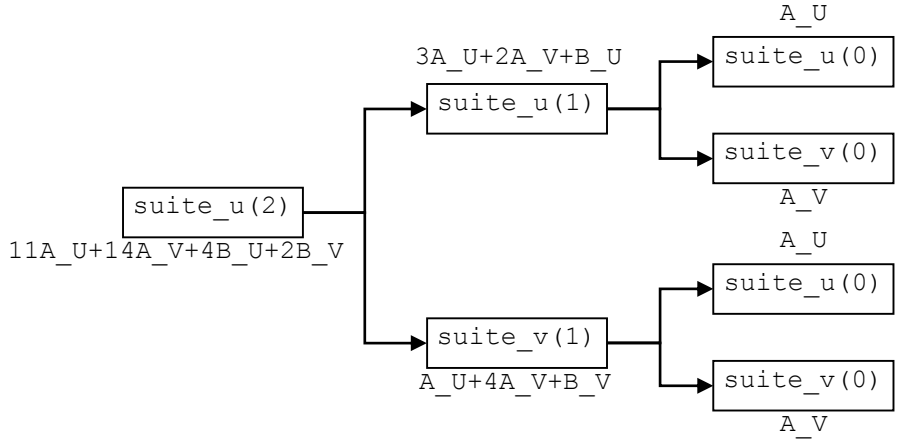
Pour le premier exemple de la sous-section 13.2, on a l'arbre suivant pour $n = 3$:

$$u_n = \begin{cases} a \text{ si } n \in \{0,1\} \\ u_{n-1} + u_{n-2} + b \text{ sinon} \end{cases}$$



Pour le second exemple de la section 13.2, avec $n = 2$, on obtient l'arbre suivant :

$$u_n = \begin{cases} a_u \text{ si } n = 0 \\ 3u_{n-1} + 2v_{n-1} + b_u \text{ sinon} \end{cases} ; v_n = \begin{cases} a_v \text{ si } n = 0 \\ u_{n-1} + 4v_{n-1} + b_v \text{ sinon} \end{cases}$$



13.4 Structure d'une fonction récursive

Pour être valide, une fonction récursive doit respecter certaines contraintes. Le risque, si ces contraintes ne sont pas respectées, est d'obtenir une fonction qui ne s'arrête jamais (similaire en cela à une boucle infinie) ou bien simplement une fonction qui n'est pas récursive (ce qui peut poser problème s'il s'agit d'un exercice demandant explicitement d'écrire une fonction récursive).

Étudions d'abord la structure d'une fonction récursive. Celle si doit nécessairement contenir :

- Au moins un *cas d'arrêt*, aussi appelé cas *de base* ou cas *terminal* ;
- Au moins un *cas général* ;

Et la fonction peut également contenir un ou plusieurs *cas d'erreur*. Ce que l'on appelle des cas correspondent ici à des valeurs particulières du (ou des) paramètre(s) de la fonction, qui entrainent des traitements spécifiques.

Cas d'arrêt : valeur(s) du (des) paramètre(s) entrainant la fin de la récursivité.

Ce cas correspond au $\theta = \theta_0$ dans la formulation donnée précédemment. Le traitement associé était noté $g_0(\theta_0)$. Si la fonction ne contient pas un test permettant d'identifier ce cas-là, elle ne s'arrêtera pas. La fonction g_0 ne doit surtout pas être récursive, sinon… ce n'est plus un cas d'arrêt, mais un cas général.

Cas général : valeur(s) du (des) paramètre(s) provoquant les appels récursifs.

Dans notre formalisation, ce cas correspond à $\theta \neq \theta_0$, et le traitement associé est $g(f(\theta'))$. C'est ce cas qui permet de définir la relation de récurrence, et il doit donc obligatoirement contenir un appel récursif (représenté ici par $f(\theta')$). Si votre fonction ne contient pas cet appel, alors elle n'est pas récursive.

Si on rappelle la fonction f avec exactement les mêmes paramètres, on va se retrouver en présence d'une fonction se répétant à l'infini. Il est donc important que les paramètres θ soient modifiés pour obtenir θ'. Cependant, il ne faut pas appliquer n'importe quelle modification : pour que la fonction finisse par se terminer, il faut que l'arbre d'appel aboutisse à un cas d'arrêt. Cela signifie que les paramètres doivent être modifiés de manière à *atteindre* θ_0 après un nombre *fini* d'appel.

Cas d'erreur : valeur(s) du (des) paramètres pour lequels la fonction n'est pas définie.

Un cas d'erreur termine lui aussi la chaîne d'appels récursifs. Cependant, à la différence du cas d'arrêt, la fonction est incapable de renvoyer une valeur relative au calcul qu'elle est supposée effectuée. Elle renverra généralement plutôt un code indiquant qu'une erreur s'est produite.

exemple : calcul de factorielle

Ici, on a les cas suivants :

- Cas d'erreur : $n < 0$
- Cas d'arrêt : $n \in \{0,1\}$
- Cas général: $n! = n \times (n-1)!$

```
int factorielle(int n)
{  int resultat=-1;
   if(n<0)
      printf("factorielle : erreur n<0");
   else if(n==0 || n==1)
      resultat=1;
   else
      resultat=n*factorielle(n-1);
   return resultat;
}
```

13.5 Comparaison itératif/récursif

Tous les problèmes peuvent être résolus à la fois de façon itérative et récursive. Par extension, tout programme récursif peut être transformé en un programme itératif équivalent, et vice-versa. En effet, en utilisant une structure de données appelée *pile de données* (section 15), il est possible de simuler, dans un programme itératif, l'empilement des appels récursifs. On a vu (sous-section 13.1) que tout programme itératif peut être exprimé sous la forme d'une fonction récursive terminale.

exemple : somme des entiers de 1 à *N*

- Version itérative :

```
int somme_it()
{  int i,resultat=0 ;
   for(i=0;i<=N;i++)
      resultat = resultat + i;
   return resultat;
}
```

- version récursive : (appel initial : `somme_rec(N)`)

```
int somme_rec(int i)
{  int resultat;
   if(i==0)
      resultat = 0;
   else
```

```
        resultat = i + somme_rec(i-1);
    return resultat;
}
```

On a vu précédemment (section 7) qu'à chaque appel d'une fonction, le système empile l'environnement de la fonction (les variables locales) dans une zone de la mémoire appelée *pile*. La grande différence entre une fonction effectuant un traitement itérativement et une fonction effectuant le même traitement récursivement est que :

- La fonction itérative est appelée une seule fois : un seul environnement est empilé
- La fonction récursive est appelée p fois : p environnements sont empilés.

L'approche récursive occupe donc plus de place en mémoire. Toutefois, sur du matériel récent, ce point n'est pertinent que si on traite une grande quantité de données, et/ou si le nombre d'appels récursifs est très important : on risque alors un dépassement de pile[21].

Mais cela ne veut pas dire que l'approche itérative est toujours la meilleure solution. D'autres critères que le performance entrent en jeu lorsqu'on écrit un programme, en particulier la difficulté à écrire le programme et la lisiblité du code source obtenu. Or, certains problèmes sont naturellement plus facile à résoudre de façon récursive. Dans ces cas-là, un programme récursif sera beaucoup plus compact, intelligible, et simple à concevoir que le programme itératif équivalent.

13.6 Exercices

1) Écrivez une fonction `puissance` qui reçoit deux entiers `x` et `y` en paramètres, et qui renvoie x^y par valeur, après l'avoir calculé récursivement. Le nombre `y` doit être positif ou nul, sinon la fonction le refuse.

```
int puissance(int x, int y)
{  int resultat=-1;
   if(y<0)
      printf("puissance : erreur y<0");
   else if(y==0)
      resultat=1;
   else if(y==1)
      resultat=x;
   else
      resultat=x*puissance(x,y-1);
   return resultat;
}
```

2) Pour tout $(n,p) \in \mathbb{N}^2$ avec $0 \leq p \leq n$, on définit l'entier C_n^p par :

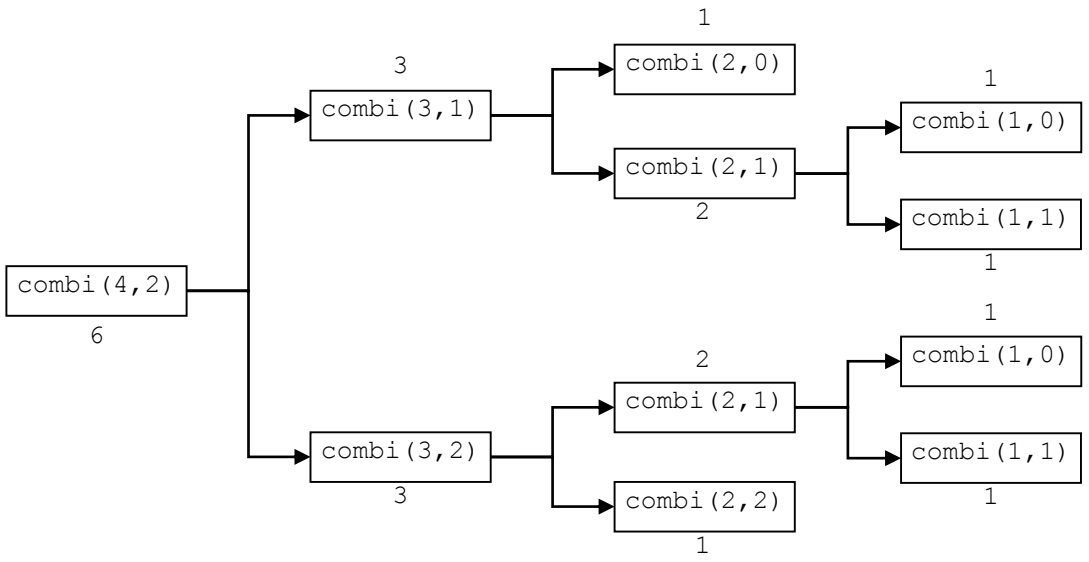
$$C_n^p = \begin{cases} 1 \text{ si } p = 0 \text{ ou } p = n \\ C_{n-1}^{p-1} + C_{n-1}^{p} \text{ sinon} \end{cases}$$

Écrivez une fonction récursive `combi` qui prend n et p en paramètres et retourne C_n^p par valeur. Vous donnerez l'arbre des appels de C_4^2.

```
int combi( int n, int p)
{  int resultat=-1;
   if(p<0)
      printf("combi : erreur p<0");
   else if(p>n)
      printf("combi : erreur p<n");
   else if (p==0 || n==p)
      resultat = 1;
   else
```

[21] *Stack overflow*, en anglais.

```
        resultat = combi(n-1,p-1) + combi(n-1,p);
    return resultat;
}
```



3) Écrivez une fonction `maximum` recevant un pointeur sur une chaîne de caractères `chaine` en paramètre, et renvoyant par valeur le caractère le plus grand du tableau, après l'avoir calculé récursivement.

```
char maximum(char *chaine)
{  char resultat=chaine[0];
   if(resultat != '\0')
   {  char max = maximum(chaine+1);
      if(max>resultat)
         resultat = max;
   }
   return resultat;
}
```

14 Listes chaînées

Les sections précédentes nous ont permis d'introduire les notions nécessaires à la description de structures de données plus avancées, que nous allons traiter à partir de maintenant. Nous commencerons par les *listes chaînées*, qui seront ensuite utilisées tout le long du semestre, à la fois en cours et en TP.

14.1 Présentation

Une liste correspond à un type de données *abstrait*, dans le sens où il peut être implémenté de différentes façons. Nous reviendrons bientôt sur cette notion de type abstrait.

Liste : séquence d'éléments uniformes.

Ici, *uniforme* signifie que tous les éléments constituant la liste sont du même type. On peut donc avoir une liste d'entiers, ou une liste de réel, etc. On distingue différentes parties dans une liste :

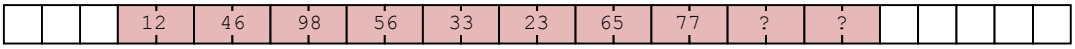
Tête de liste : le tout premier élément de la séquence.

Queue de liste : la sous-liste constituée de tous les éléments sauf le premier.

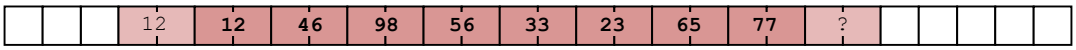
On peut considérer qu'un tableau respecte cette définition du concept de liste. Donc, un tableau est une implémentation possible d'une liste. Cependant, cette implémentation-là souffre de nombreuses limitations. En effet, une liste est le plus souvent utilisée de façon dynamique, c'est-à-dire que l'on veut y insérer de nouveaux éléments, ou bien supprimer des éléments existants. Or, on a vu (section 6) que ce genre d'utilisation n'est pas approprié pour les tableaux, en raison de leur taille *statique*.

On pourrait, à la place, utiliser à la place des tableaux alloués dynamiquement (section 11), mais cela cause aussi des désagréments. Premièrement, l'insertion et la suppression d'éléments dans le tableau nécessitent de décaler de nombreuses valeurs. Par exemple, si on veut insérer un élément en tête d'un tableau `tab`, il faut d'abord recopier chaque élément `tab[i]` dans l'élément `tab[i+1]`, afin de dégager de la place pour l'élément à insérer.

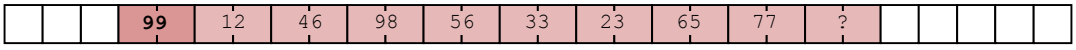
exemple : soit le tableau ci-dessous, pouvant contenir 10 entiers `short`, mais n'en contenant que 8 pour l'instant.



Si on veut rajouter la valeur 99 en tête, on doit d'abord décaler toutes les valeurs vers la droite :



Et ensuite seulement, on peut placer le 99 dans le premier élément :

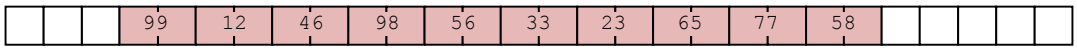


Il faut noter que dans le cas de la suppression d'un élément, c'est le contraire qui doit être réalisé : un décalage des valeurs du tableau vers la *gauche*.

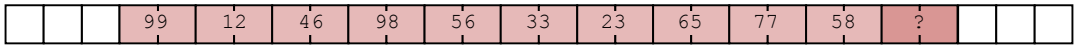
Deuxièmement, réaliser une insertion suppose qu'il reste assez de place dans le tableau. Et l'unique moyen de savoir ça est de connaître en permanence le nombre d'éléments stockés dans le tableau, ce qui impliquer par exemple d'utiliser une variable pour les compter. Dans l'exemple ci-dessus, on a besoin de savoir que notre tableau de 10 éléments ne contient initialement que 8 valeurs, puis 9 après l'insertion de 99.

Troisièmement, lorsqu'il n'y a plus assez de place dans le tableau, il est nécessaire d'effectuer une réallocation afin de l'agrandir.

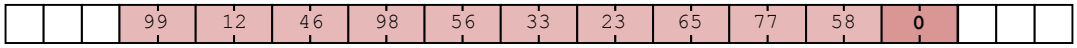
exemple : supposons qu'on ait le tableau suivant, et que l'on veut y insérer une nouvelle valeur 0.



Alors il va falloir d'abord l'agrandir d'un élément :



Puis enfin faire l'insertion :



Comme on l'a vu quand nous avons étudié l'allocation dynamique, s'il n'est pas possible d'agrandir la zone mémoire actuellement allouée, il est possible qu'une zone entièrement différente soit réservée, ce qui implique de recopier l'intégralité du tableau ailleurs en mémoire.

Ces différents inconvénients n'empêchent pas d'utiliser les tableaux pour représenter des listes, mais ils rendent cette approche inefficace. Ceci est dû au fait que les tableaux ont besoin de stocker leurs éléments de façon *contigüe* (cf. section 6).

Pour éviter ces problèmes, il faut donc adopter une implémentation dans laquelle la place d'un élément en mémoire n'est pas liée à sa place dans la liste : c'est le but des listes *chaînées* :

Liste chaînée : liste dont l'implémentation inclut un *lien* entre deux éléments consécutifs.

Dans un tableau, on n'a pas ce lien, car la position en mémoire de deux éléments consécutifs est contigüe (i.e. ils sont juste à côté). Le lien vient se substituer à cette contrainte spatiale, et permet de s'en affranchir. Il peut être de différentes natures, en fonction de l'implémentation adoptée. En langage C, on utilise le système de pointeurs.

Nous allons distinguer deux sortes de listes en fonction du nombre de lien par élément, et de leur direction : listes *simplement* ou *doublement* chaînées.

14.2 Listes simplement chaînées

Les listes simplement chaînées sont la version de base :

Liste simplement chaînée : chaque élément contient un lien vers un seul élément consécutif.

En pratique, nous utiliserons un pointeur pour indiquer l'adresse de l'élément *suivant* dans la liste. Donc, pour l'élément n de la liste, on va associer un pointeur sur l'élément $n+1$.

Notez que cette implémentation permet de parcourir la liste du début (premier élément) vers la fin (dernier élément). Par contre, on ne peut pas remonter la liste, i.e. passer d'un élément n à un élément $n-1$.

Dans cette sous-section, nous allons d'abord définir un type permettant de représenter ce genre de liste, puis des fonctions capables de les manipuler.

14.2.1 Structure de données

Pour un élément donné, nous avons deux informations à représenter :

- D'une part, la valeur de l'élément (par exemple : un entier s'il s'agit d'une liste d'entiers) ;
- D'autre part, le lien vers l'élément suivant, prenant la forme d'un pointeur.

Comme on l'a vu, seul une structure permet de représenter des données hétérogènes comme celles-ci. Nous définissons donc un type structuré `element` à cet effet :

```
typedef struct s_element
{  int valeur ;
   struct s_element *suivant;
} element;
```

Remarque : ici, on a supposé qu'on voulait représenter une liste d'entiers `int`, mais on pourrait substituer n'importe quel autre type, sans perte de généralité. On fera la même hypothèse tout le long de cette section.

Le premier champ est la valeur de l'élément, le second champ est le pointeur sur l'élément suivant. Il faut bien noter que le type de ce pointeur est `struct s_element`, et non pas `element`. En effet, le type `element` n'est pas encore connu du compilateur, puisqu'on est en train de le définir : on ne peut donc pas utiliser son nom. Par contre, on peut donner un nom à la structure elle-même (comme on l'a vu en section 8.2). C'est ce qu'on fait avec `s_element`, et c'est pourquoi on peut l'utiliser pour déclarer le pointeur. À noter que ce nom n'est utile qu'ici : on ne l'utilisera plus jamais ensuite, on lui préfèrera `element` (pour des raisons pratiques).

Remarque : la structure `s_element` peut être qualifiée de *récursive*, puisqu'elle est définie à partir d'elle-même.

Mais un élément n'est pas la liste entière, pour laquelle on a besoin d'un type spécifique. Nous déclarons un autre type structure, appelé `liste`, qui va nous servir à représenter la tête (i.e. le début) de la liste, sous la forme d'un champ `debut` pointant vers le premier `element` :

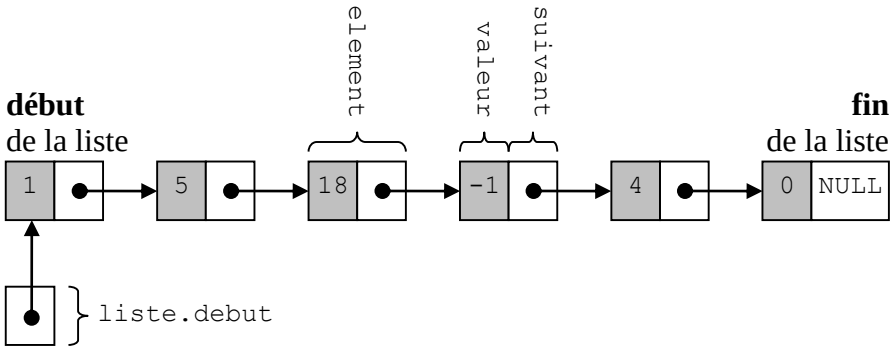
```
typedef struct
{  element *debut;
} liste;
```

Notez que maintenant, on a le droit d'utiliser `element` dans la déclaration de la structure, puisque le type `element` est déjà connu du compilateur.

On peut s'interroger sur la pertinence d'utiliser un type structure ne contenant qu'un seul champ. Nous avons fait ce choix afin de permettre à nos structures de données d'évoluer pour représenter des listes plus complexes, comme on le verra en section 14.3.

On a maintenant les structures de données nécessaires pour représenter une liste et les éléments qui la constituent. Pour être complet, il faut aussi pouvoir représenter la *fin* de la liste. Autrement dit, il faut décider sur quoi va pointer le tout dernier élément de la liste (puisqu'il n'y a pas d'élément après lui, il ne peut pas pointer dessus). Le choix le plus naturel est de ne pointer sur aucune adresse valide, et donc d'utiliser la constante `NULL`.

exemple : une liste simplement chaînée quelconque

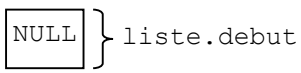


La conséquence de ce choix est que le champ `debut` d'une liste ne contenant aucun élément pointera sur `NULL`. On appelle ce genre de liste une liste vide :

Liste vide : liste qui existe en tant que variable, mais qui ne contient aucun élément.

exemple : une liste simplement chaînée vide

liste **vide**



Remarque : bien qu'on se soit concentré ici sur le cas le plus simple, on peut sans problème généraliser la notion de liste, de manière à ce que chaque élément contienne *plusieurs valeurs distinctes* au lieu d'une seule.

Cela peut être effectué de deux manières différentes :

- Soit en rajoutant des champs supplémentaires dans `element`, de manière à représenter ces données additionnelles ;
- Soit en définissant un type structure représentant toutes les données d'un élément, puis en utilisant ce type dans `element`, de manière à définir une liste de valeurs complexes (par opposition à une liste de valeurs simples, telle qu'une liste d'entiers).

14.2.2 Création

Pour créer une nouvelle liste, il suffit de déclarer une variable de type `liste` et d'initialiser sont champ `debut` à `NULL`.

Remarque : il est absolument essentiel de penser à initialiser ce pointeur, sinon votre programme provoquera des erreurs d'exécutions qui seront très difficiles à identifier.

Une fois qu'on a créé une liste, on veut la remplir. Pour cela, on a besoin de pouvoir créer de nouveaux éléments. Deux possibilités existent :

- Soit on déclare une variable de type `element` ;
- Soit on effectue une allocation dynamique.

La première méthode pose plusieurs problèmes. Le principal est que l'élément créé sera *local* à la fonction dans lequel il est déclaré. Cela risque de provoquer des erreurs si on veut utiliser la liste dans une fonction située plus haut dans l'arbre d'appel.

De plus, les variables allouées statiquement ne peuvent pas être désallouée avec `free`. Or, l'un des avantages de la liste par rapport aux tableaux est justement de permettre de changer l'occupation de la mémoire en fonction de la taille de la liste. Bien sûr, on ne veut pas perdre cet avantage.

Il faut donc utiliser la seconde, basée sur `malloc`. La fonction suivante prend une valeur en paramètre et renvoie un pointeur sur un nouvel `element`, ou bien `NULL` si l'allocation dynamique échoue :

```
element* cree_element(int valeur)
{  element *e;

   if((e = (element *) malloc(sizeof(element))) != NULL)
   {  e->valeur = valeur;
      e->suivant = NULL;
   }

   return e;
}
```

Remarquez bien qu'on initialise complètement l'élément créé : non seulement la valeur grâce au paramètre reçu, mais aussi le pointeur sur l'élément suivant, qui est inconnu (d'où l'utilisation de `NULL`). L'élément créé pourra être désalloué par `free` si nécessaire.

14.2.3 Parcours d'une liste

La plus grande partie des traitements effectués sur les listes consistent à les parcourir, c'est-à-dire à aller d'un bout à l'autre de la liste en traitant chaque élément consécutivement. Pour cela, l'approche standard consiste à utiliser pointeur temporaire, que nous noterons `temp`, et qui ne sera utilisé que pour cette tâche de parcours. Il est initialisé au début de la liste, et est modifié par une boucle (voire une fonction récursive).

exemple : la fonction `affiche_liste` permet d'afficher une liste d'entiers :

```
void affiche_liste(liste l)
{  element *temp = l.debut;

   printf("{");
   while(temp != NULL)
   {  printf(" %d", temp->valeur);
      temp = temp->suivant;
   }
   printf(" }\n");
}
```

Notez qu'on initialise `temp` avec `l.debut`, i.e. l'adresse du 1er élément de la liste. Remarquez aussi qu'on répète le même traitement jusqu'à ce que l'adresse pointée par `temp` soit `NULL`, i.e. tant qu'il reste des éléments dans la liste. Si celle-ci est vide, on ne rentre donc même pas dans la boucle `while`. Enfin, l'affectation `temp = temp->suivant` permet de faire pointer `temp` sur l'élément qui suit l'élément sur lequel il pointe à ce moment, c'est-à-dire : l'élément suivant.

Si on utilise cette fonction pour afficher les listes des deux figures précédentes, on obtient à l'écran les résultats suivants :

```
{ 1 5 18 -1 4 0 }
{  }
```

14.2.4 Accès à un élément

Le parcours d'une liste permet notamment d'accéder à un élément en particulier, tout simplement en parcourant la liste à partir du début et en s'arrêtant une fois atteint l'élément souhaité. Autrement dit, pour accéder à l'élément situé en position p de la liste, il faut parcourir les $p - 1$ éléments situés avant lui.

Soit `element* accede_element(liste l, int p)` la fonction qui renvoie l'élément (ou plutôt son adresse) situé à la position `p` dans la liste `l`. La valeur `p=0` désigne la tête de la liste (tout premier élément). La fonction renvoie `NULL` en cas d'erreur.

```
element* accede_element(liste l, int p)
{  element *resultat = l.debut;
   int i;

   i=0;
   while(i<p && resultat!=NULL)
   {  resultat = resultat->suivant;
      i++;
   }

   return resultat;
}
```

Le pointeur `temp` de la sous-section précédente est ici appelé `resultat`, car il s'agit directement du pointeur renvoyé par la fonction. Remarquez que la boucle `while` contient deux conditions : on parcourt la liste tant qu'on n'a pas atteint la position `p` (`i<p`), mais il faut aussi qu'on n'ait pas atteint prématurément la fin de la liste (`resultat !=NULL`). Autrement dit, si la liste a une longueur `n`, en théorie on doit avoir `p<n`. Si ce n'est pas le cas, alors la fonction va atteindre la fin de la liste avant d'atteindre la position `p`, et c'est pourquoi elle renverra `NULL`, ce qui indiquera donc une erreur.

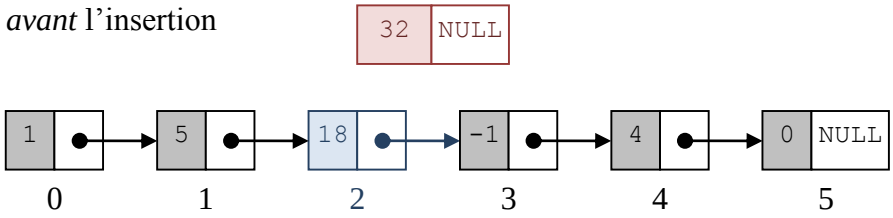
14.2.5 Insertion d'un élément

L'insertion d'un élément à une position quelconque dans une liste passe par la modification du pointeur `suivant` de l'élément à insérer, ainsi que celui de l'élément qui le précède dans la liste.

Quelle soit la position d'insertion, on n'aura jamais que ces deux modifications à effectuer. Ceci est à comparer avec le travail nécessaire avec un tableau : plus la position d'insertion est proche du début et plus il faut décaler d'éléments. Autrement dit, pour une liste on effectue un travail prenant un temps constant (2 opérations), alors que pour un tableau, ce temps est fonction du point d'insertion, et varie entre n (taille du tableau avant l'insertion) et 0 (insertion en fin de tableau).

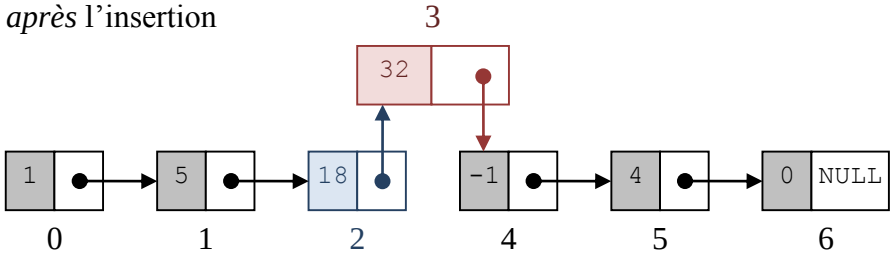
exemple : on veut insérer un nouvel élément de valeur 32 à la position 3 dans la liste de l'exemple précédent :

avant l'insertion



On doit modifier le pointeur `suivant` de l'élément précédent (position 2), et celui du nouvel élément.

après l'insertion



Soit `int insere_element(liste *l, element *e, int position)` la fonction qui insère l'élément pointé par `e` dans la liste pointée par `l`, à la position `p`. La fonction renvoie −1 si elle n'a pas pu insérer l'élément à la position demandée ou 0 si l'insertion s'est bien passée.

```
int insere_element(liste *l, element *e, int p)
{  element *temp = l->debut, *avant, *apres;
   int erreur = 0;

   // si on insère en position 0, on doit modifier l
   if(p==0)
   {  e->suivant = temp;
      l->debut = e;
   }
   // sinon, on doit modifier le champ 'suivant' de l'élément précédent
   else
   {  // on se place au bon endroit
      temp = accede_element(*l,p-1);
      // on insère l'élément
      if(temp==NULL)
         erreur = -1;
      else
      {  avant = temp;
         apres = temp->suivant;
         e->suivant = apres;
         avant->suivant = e;
      }
   }

   return erreur;
}
```

Notez qu'on est susceptible de modifier la variable `l` : si jamais `p=0`, il faut mettre à jour son champ `debut`. C'est pourquoi ici, à la différence des fonctions précédentes, la liste est passée par adresse. L'élément pointé par `e` sera *toujours* modifié, car on doit mettre à jour son champ suivant : il est donc lui aussi passé par adresse.

On utilise la fonction `accede_element` pour se rendre juste avant la position d'insertion (i.e. $p-1$). Ceci permet d'obtenir l'élément dont on doit modifier le champ suivant afin de le faire pointer sur le nouvel élément `e`.

Les pointeurs appelés `avant` et `apres` sont là pour rendre le code source plus clair : `avant` pointe sur l'élément situé avant le point d'insertion (position 2 dans l'exemple précédent), alors qu'`apres` pointe sur le point d'insertion lui-même (position 3 dans l'exemple précédent, dans la numérotation initiale).

On considère qu'une erreur se produit si la position `p` indiquée en paramètre est invalide, i.e. supérieure à la longueur de la liste. La fonction renvoie alors la valeur -1 pour bien montrer qu'une erreur s'est produite. C'est une pratique courante en langage C. Elle implique que quand on utilise cette fonction, on contrôle la valeur qu'elle renvoie, afin de s'assurer que tout s'est bien passé. Sinon, une erreur d'exécution risque de se produire plus tard dans le déroulement du programme, et sera difficile à corriger.

Remarque : l'élément reçu par la fonction `insere_element` est supposé avoir été préalablement initialisé grâce à la fonction `cree_element`.

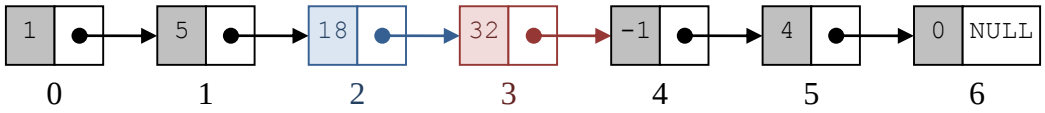
14.2.6 Suppression d'un élément

La suppression d'un élément dans une liste simplement chaînée se fait de façon à peu près symétrique à l'insertion. Il est nécessaire de modifier le pointeur `suivant` de l'élément qui précède l'élément à supprimer.

Comme pour l'insertion, quelle que soit la position concernée, le nombre d'opérations à réaliser sera constant, alors que pour un tableau on aurait toute une série de décalage (cette fois vers la *gauche*) à effectuer.

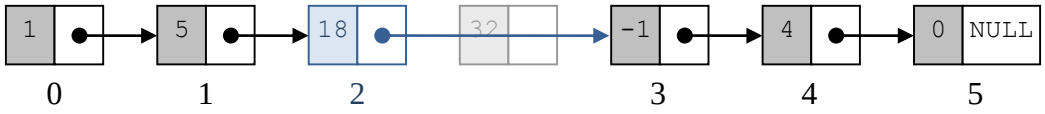
exemple : on veut supprimer l'élément de valeur 32 situé à la position 3 après l'exemple précédent :

avant la suppression



On doit modifier le pointeur `suivant` de l'élément précédent (position 2), de manière à le faire pointer vers l'élément suivant (position 4) de l'élément à supprimer (position 3).

après la suppression



Soit `int supprime_element(liste *l, int position)` la fonction qui supprime l'élément situé à la position `p` dans la liste `l`. La fonction renvoie -1 si elle n'a pas pu supprimer l'élément à la position demandée ou 0 si la suppression s'est bien passée.

```
int supprime_element(liste *l, int p)
{  element *temp = l->debut, *e, *avant, *apres;
   int erreur = 0;
```

```
    // si on supprime en position 0, on doit modifier l
    if(p==0)
    {  e = temp;
       if(e == NULL)
          erreur = -1;
       else
          l->debut = e->suivant;
    }
    // sinon, on doit modifier le champ 'suivant' de l'élément précédent
    else
    {  // on se place au bon endroit
       temp = accede_element(*l,p-1);
       // on supprime l'élément de la liste
       if(temp==NULL)
          erreur = -1;
       else
       {  avant = temp;
          e = temp->suivant;
          if(e == NULL)
             erreur = -1;
          else
          {  apres = e->suivant;
             avant->suivant = apres;
          }
       }
    }
    // on désalloue l'élément
    if(erreur != -1)
       free(e);

    // on termine la fonction
    return erreur;
}
```

Comme pour l'insertion, on est susceptible de modifier la liste, et elle doit donc être passée par adresse. Le principe de la fonction est le même que pour l'insertion : on utilise `accede_element` pour se placer au bon endroit dans la liste, puis on modifie le champ suivant approprié.

À noter que cette fois, on n'a pas besoin de modifier l'élément concerné, puisque celui-ci va de toute façon être désalloué avec `free`. Rappelons que pour pouvoir être désalloué par `free`, l'élément doit avoir été alloué avec `malloc` ou une autre fonction d'allocation (cf. section 11.3). Ici, c'est normalement le cas car les éléments sont supposés avoir été créés par `cree_element`, qui utilise elle-même `malloc`.

La fonction renvoie un code d'erreur égal à -1 en cas de problème : comme pour l'insertion, il s'agit de la situation où `p` est trop grand par rapport à la taille de la liste.

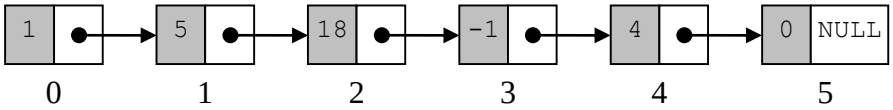
14.2.7 Exercices

1) (TP) Écrivez une fonction `int longueur_liste(liste l)` qui renvoie le nombre d'éléments présents dans la liste `l` passée en paramètre.

```
int longueur_liste(liste l)
{  element *temp = l.debut;
   int resultat = 0;

   while(temp != NULL)
   {  resultat++;
      temp = temp->suivant;
   }

   return resultat;
}
```

2) (TP) Écrivez une fonction `int recherche_valeur(liste l, int v)` qui recherche une valeur `v` dans une liste simplement chaînée `l` (qui n'est pas ordonnée). La fonction doit renvoyer la position de la première occurrence de la valeur dans la liste, ou `-1` si la liste ne contient pas d'éléments de valeur `v`.

exemple :



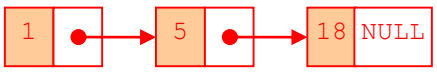
Si on recherche la valeur −1 dans cette liste, la fonction doit renvoyer 3. Si on recherche la valeur 99, elle doit renvoyer −1.

```
int recherche_valeur(liste l, int v)
{  int i = 0;
   int pos = -1;
   element *temp = l.debut;

   while(pos==-1 && temp!=NULL)
   {  if(temp->valeur == v)
         pos = i;
      else
      {  temp = temp->suivant;
         i++;
      }
   }

   return pos;
}
```

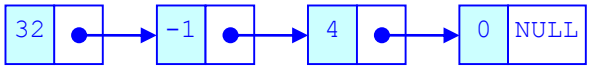
3) (TP) Écrivez une fonction `void fusionne_listes(liste *l1, liste l2)` qui prend en paramètres un pointeur sur une liste simplement chaînée `l1`, et une liste simplement chaînée `l2`. La fonction doit rajouter la liste `l2` à la fin de la liste `l1`.

exemple :

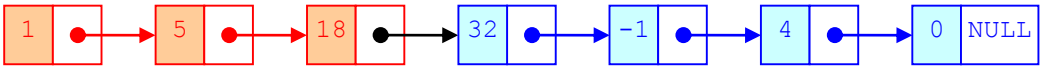
liste **1** avant la fusion



liste **2**



liste **1** après la fusion



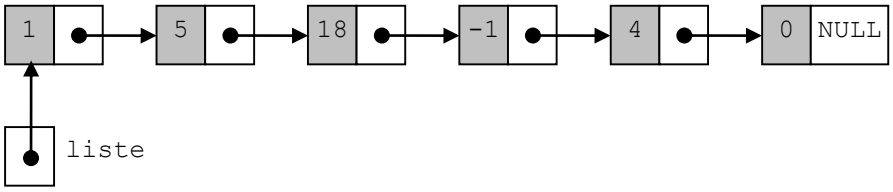

```
void fusionne_listes(liste *l1, liste l2)
{  element *temp = l1->debut;

   // si la liste 1 est vide, la fusion est la liste 2
   if(temp == NULL)
      l1->debut = l2.debut;
   else
   {  // sinon on va à la fin de la liste 1
      while(temp->suivant != NULL)
         temp = temp->suivant;
      //et on rajoute la liste 2
      temp->suivant = l2.debut;
   }
```

```
}
```

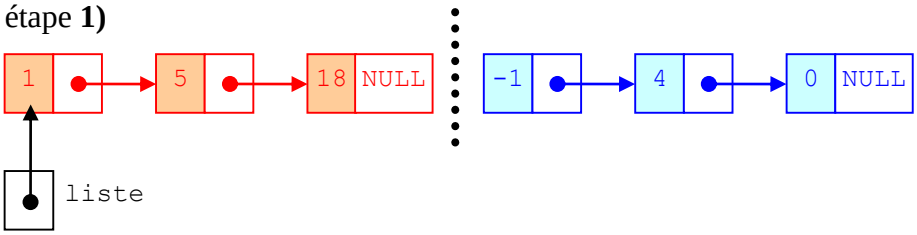
4) (TP) Écrivez une fonction `void melange_liste(liste *l)` qui prend un pointeur sur une liste simplement chaînée en paramètre, et dont le rôle est de :
   1) Couper la liste en deux, à la moitié (à un élément près si la taille est impaire) ;
   2) Intervertir les deux moitiés.

exemple :

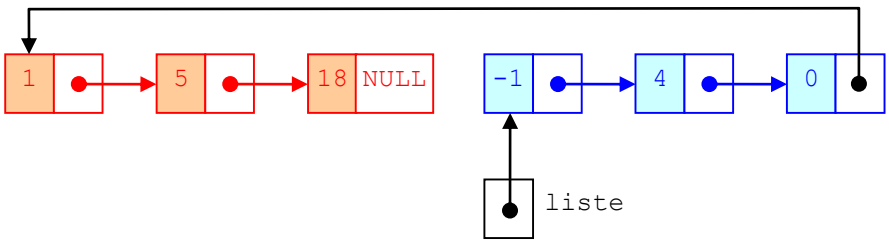
liste de *départ*



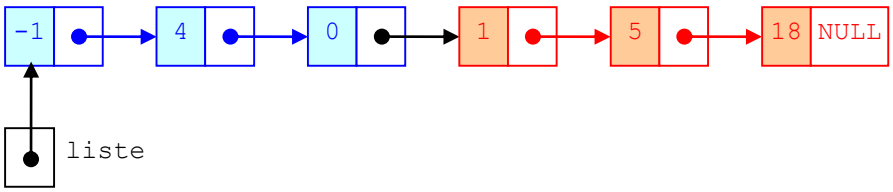
étape **1)**



étape **2)**



liste *résultat*



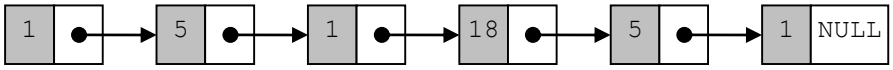

```
void melange_liste(liste *l)
{   int moitie = longueur_liste(*l)/2;
    element *temp, *debut;
    int i;

    // si la liste est de longueur 0 ou 1, il n'y a rien à faire
    if(moitie == 0)
        return ;

    // ..:: etape 1 ::..
    // on trouve le milieu de la liste
    temp = l->debut;
    for(i=1;i<moitie;i++)
        temp = temp->suivant;
    // on coupe la liste
    debut = temp->suivant;
    temp->suivant = NULL;
```

```
    // ..:: etape 2 ::..
    // on trouve la fin de la liste
    temp = debut;
    while(temp->suivant != NULL)
       temp = temp->suivant;
    // on relie l'ancienne fin à l'ancienne tête
    temp->suivant = l->debut;
    // on positionne la nouvelle tête
    l->debut = debut;
}
```

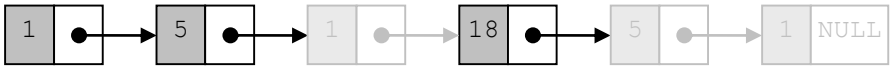
5) Des *doublons* sont des éléments d'une liste qui ont la même valeur. Écrivez une fonction `void supprime_doublons(liste l)` qui prend une liste simplement chaînée `l` en paramètre, et la traite de manière à ce que la même valeur n'apparaisse pas deux fois (i.e. il faut supprimer des doublons en trop).

exemple :

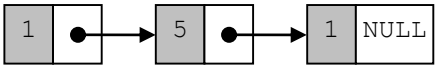
liste de *départ*



suppression des doublons



liste *résultat*




```
void supprime_doublons(liste l)
{  element *temp=l.debut,*temp2;
   element *avant,*apres;

   while (temp!=NULL)
   {  avant = temp;
      temp2 = avant->suivant;
      while (temp2!=NULL)
      {  apres = temp2->suivant;
         if (temp->valeur == temp2->valeur)
         {  avant->suivant = apres;
            free(temp2);
         }
         else
            avant = temp2;
         temp2 = apres;
      }
      temp = temp->suivant;
   }
}
```

6) Écrivez une fonction récursive `void affiche_liste_recursif(liste l)` qui affiche le contenu d'une liste simplement chaînée `l`.

```
void affiche_liste_recursif(liste l)
{  element *temp = l.debut;
   liste l_bis;

   if (temp != NULL)
   {  printf("%d ",temp->valeur);
      l_bis.debut = temp->suivant;
```

```
        affiche_liste_recursif(l_bis);
    }
}
```

7) (TP) On dit qu'une liste `l2` préfixe une liste `l1` si `l2` apparaît **entièrement** au début de `l1`.

exemples :

l1	**l2**	**préfixe ?**
{**1, 2, 4,** 8, 0, 3}	{**1, 2, 4**}	oui
{**1,** 2, 4, 8, 0, 3}	{**1**}	oui
{**1, 2, 4, 8**}	{**1, 2, 4, 8**}	oui
{}	{}	oui
{1, 2, 4, 8, 0, 3}	{}	oui
{1, 2, 4, 8, 0, 3}	{5, 6}	non
{1, 2, **4, 8, 0,** 3}	{**4, 8, 0**}	non
{**1, 2, 4, 8, 0, 3**}	{**1, 2, 4, 8, 0, 3,** 7}	non

Écrivez une fonction récursive `int est_prefixe(liste l1, liste l2)` qui prend deux listes `l1` et `l2` en paramètre, et renvoie un entier indiquant si `l2` est préfixe de `l1`. L'entier doit valoir 1 pour oui et 0 pour non.

```
int est_prefixe(liste l1, liste l2)
{  element *temp1=l1.debut, *temp2=l2.debut;
   liste l1_bis, l2_bis;
   int resultat;

   if (temp2==NULL)
      resultat = 1;
   else
      if (temp1==NULL)
         resultat = 0;
      else
         if (temp1->valeur == temp2->valeur)
         {  l1_bis.debut = temp1->suivant;
            l2_bis.debut = temp2->suivant;
            resultat = est_prefixe(l1_bis,l2_bis);
         }
         else
            resultat = 0;

   return resultat;
}
```

8) Écrivez une fonction `int copie_liste(liste l1, liste *l2)` qui réalise une copie de la liste `l1`. Cette copie doit être renvoyée via le paramètre `l2`, qui doit être initialement une liste vide. La fonction renvoie −1 en cas d'erreur et 0 en cas de succès.

```
int copie_liste(liste l1, liste *l2)
{  element *temp=l1.debut,*e,erreur=0;
   int i=0;

   while(temp!=NULL && erreur!=-1)
   {  if((e=cree_element(temp->valeur))==NULL)
         erreur = -1;
      else
      {  temp = temp->suivant;
         insere_element(l2, e, i);
         i++;
      }
   }
   return erreur;
```

```
}
```

14.3 Listes doublement chaînées

Il existe de nombreuses applications des listes chaînées dans lesquelles on n'a besoin de parcourir la liste que dans un seul sens (du début vers la fin). Mais il est également possible d'avoir besoin de la parcourir dans les *deux* sens.

On utilise alors une liste *doublement* chaînée :

Liste doublement chaînée : liste dans laquelle chaque élément est lié à ses deux éléments voisins (le précédent *et* le suivant).

C'est la présence de ces deux liens qui permet d'aller dans les deux directions. Bien sûr, cela implique de modifier à la fois les types définis pour les listes simples, et les fonctions qui les manipulent.

14.3.1 Structure de données

Le type structuré `element` va contenir un pointeur supplémentaire, vers l'élément *précédent*. Toujours pour une liste d'entiers de type `int`, on aura donc :

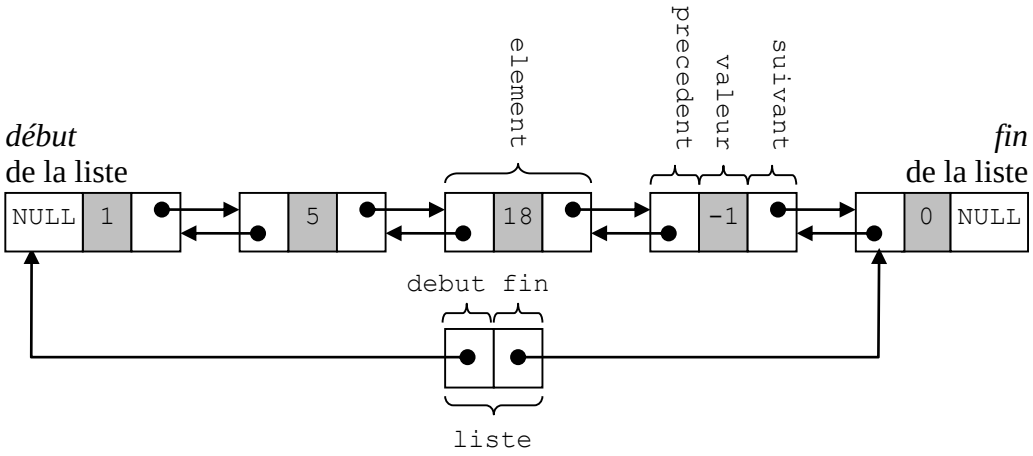
```
typedef struct s_element
{  int valeur ;
   struct s_element *precedent;
   struct s_element *suivant;
} element;
```

Exactement de la même façon, le type `liste` contiendra un pointeur supplémentaire sur le dernier élément de la liste :

```
typedef struct
{  element *debut;
   element *fin;
} liste;
```

Le pointeur `precedent` du *premier* élément d'une liste pointera vers `NULL`, exactement comme le pointeur `suivant` du *dernier* élément. Le pointeur `fin` d'une liste *vide* pointera lui aussi sur `NULL`, tout comme son pointeur `debut`.

exemple : représentation graphique d'une liste doublement chaînée :



14.3.2 Création et parcours

La création d'un élément de liste doublement chaînée sera presque identique à celle d'un élément de liste simple, la seule différence va concerner l'initialisation de l'élément : il faudra donner la valeur `NULL` au pointeur `fin` (en plus du pointeur `debut`, bien entendu).

Pour la création d'un élément, on utilisera une version modifiée de la fonction `cree_element` :

```
element* cree_element(int valeur)
{  element *e;

   if((e = (element *) malloc(sizeof(element))) != NULL)
   {  e->valeur = valeur;
      e->precedent = NULL;
      e->suivant = NULL;
   }

   return e;
}
```

La seule différence avec la version définie pour les listes simples est l'initialisation à `NULL` du champ `precedent`.

Le parcours en avant (i.e. du début vers la fin) est réalisé exactement comme pour une liste simple. Le parcours en arrière (de la fin vers le début) est réalisé en substituant `fin` à `debut` pour la liste, et `precedent` à `suivant` pour les éléments.

exemple : affichage inversé d'une liste

```
void affiche_liste_inv(liste l)
{  element *temp = l.fin;

   printf("{");
   while(temp != NULL)
   {  printf(" %d", temp->valeur);
      temp = temp->precedent;
   }
   printf(" }\n");
}
```

Pour la liste de la figure précédente, on aura l'affichage suivant :

```
{ 0 -1 18 5 1 }
```

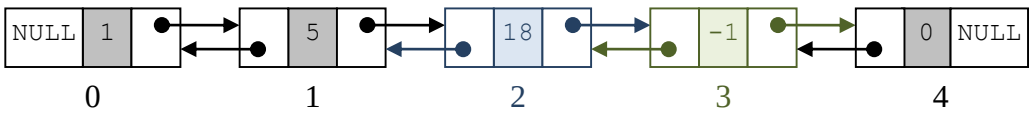
La fonction d'accès à l'élément situé en position p est exactement la même que pour les listes simples, on peut donc directement utiliser `accede_element` sans modification.

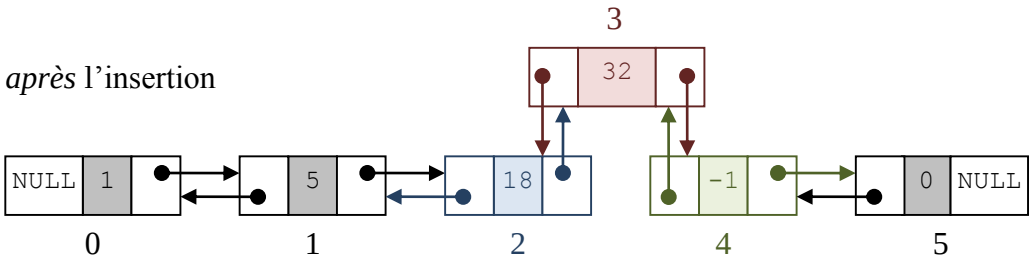
14.3.3 Insertion d'un élément

Par rapport à une liste simple, l'insertion d'un élément est quasiment similaire : on devra en plus modifier le pointeur `precedent` de l'élément à insérer et le pointeur `precedent` de l'élément situé *après* l'insertion.

exemple : on veut insérer un nouvel élément de valeur 32 à la position 3 dans la liste de l'exemple précédent :

avant l'insertion





On modifie la fonction `int insere_element(liste *l, element *e, int position)` (les modifications apparaissent en gras) :

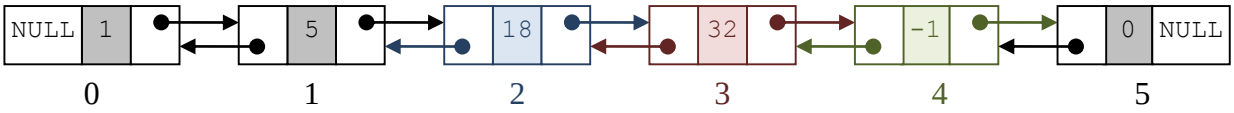
```
int insere_element(liste *l, element *e, int p)
{  element *temp = l->debut, *avant, *apres ;
   int erreur = 0;

   // si on insère en position 0, on doit modifier l
   if(p==0)
   {  if(temp != NULL)
         temp->precedent = e;
      else
         l->fin = e;
      e->suivant = temp;
      l->debut = e;
   }
   // sinon, on doit modifier le champ 'suivant' de l'élément précédent
   // et le champ 'précédent' de l'élément suivant
   else
   {  // on se place au bon endroit
      temp = accede_element(*l,p-1);
      // on insère l'élément
      if(temp==NULL)
         erreur = -1;
      else
      {  avant = temp;
         apres = temp->suivant;
         if(apres != NULL)
            apres->precedent = e;
         else
            l->fin = e;
         e->suivant = apres;
         e->precedent = avant;
         avant->suivant = e;
      }
   }

   return erreur;
}
```

Ces modifications concernent la gestion des deux pointeurs supplémentaires (`fin` dans `liste` et `precedent` dans `element`).

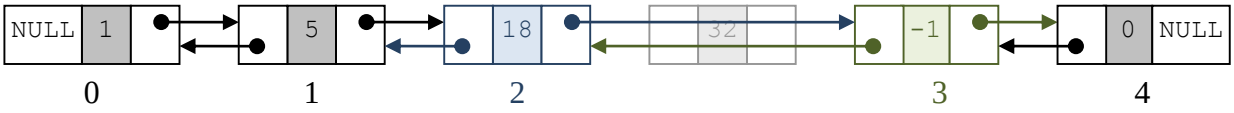
14.3.4 Suppression d'un élément

Comme pour l'insertion, la seule différence entre la suppression dans une liste simple et dans une liste double est la gestion des nouveaux pointeurs `fin` et `precedent`. Par exemple, si on veut supprimer l'élément de valeur 32 situé à la position 3 après l'exemple précédent, on a la situation suivante :

avant la suppression



après la suppression



On modifie la fonction `int supprime_element(liste *l, int position)` :

```
int supprime_element(liste *l, int p)
{  element *temp = l->debut, *e, *avant, *apres;
   int erreur = 0;

   // si on supprime en position 0, on doit modifier l
   if(p==0)
   {  if(temp==NULL)
         erreur = -1;
      else
      {  e = temp;
         l->debut = e->suivant;
         if(l->debut != NULL)
            l->debut->precedent = NULL;
         else
            l->fin = NULL;
      }
   }
   // sinon, on doit modifier le champ 'suivant' de l'élément précédent
   // et le champ 'precedent' de l'élément suivant
   else
   {  // on se place au bon endroit
      temp = accede_element(*l,p-1);
      // on supprime l'élément de la liste
      if(temp==NULL)
         erreur = -1;
      else
      {  avant = temp;
         e = temp->suivant;
         if(e == NULL)
            erreur = -1;
         else
         {  apres = e->suivant;
            avant->suivant = apres;
            if(apres != NULL)
               apres->precedent = avant;
            else
               l->fin = avant;
         }
      }
   }
   // on désalloue l'élément
   if(erreur != -1)
      free(e);

   return erreur;
}
```

14.3.5 Exercices

9) Écrivez une fonction `int echange_elements(liste *l, int p, int q)` qui intervertit les éléments `p` et `q` d'une liste `l` *doublement* chaînée. La fonction renvoie 0 en cas de succès ou −1 en cas d'erreur.

```
int echange_elements(liste *l, int p, int q)
{  element *temp_av,*temp_ap,*e_p,*e_q;
   int resultat=0;

   e_p = accede_element(*l, p);
   e_q = accede_element(*l, q);
   if(e_p==NULL || e_q==NULL)
      resultat = -1;
   else
   {  // pointeurs sur e_p
      if(e_p->precedent == NULL)
         l->debut = e_q;
      else
         e_p->precedent->suivant = e_q;
      if(e_p->suivant == NULL)
         l->fin = e_q;
      else
         e_p->suivant->precedent = e_q;
      //pointeurs sur e_q
      if(e_q->precedent == NULL)
         l->debut = e_p;
      else
         e_q->precedent->suivant = e_p;
      if(e_q->suivant == NULL)
         l->fin = e_p;
      else
         e_q->suivant->precedent = e_p;
      // pointeurs de e_p
      temp_av = e_p->precedent;
      temp_ap = e_p->suivant;
      e_p->precedent = e_q->precedent;
      e_p->suivant = e_q->suivant;
      // pointeurs de e_q
      e_q->precedent = temp_av;
      e_q->suivant = temp_ap;
   }
}
```

10) Écrivez une fonction `void inverse_liste(liste *l)` qui inverse la liste *doublement* chaînée `l` passée en paramètre.

```
void inverse_liste(liste *l)
{  element *temp = l->debut, *temp2;
   if(temp != NULL)
   {  l->debut = l->fin;
      l->fin = temp;
      while(temp != NULL)
      {  temp2 = temp->suivant;
         temp->suivant = temp->precedent;
         temp->precedent = temp2;
         temp = temp2;
      }
   }
```

```
    }
```

15 Piles de données

La pile de données fait partie, au même titre que la file de données que nous verrons en section 16, des structures de données avancées classiques de l'algorithmique. Nous allons d'abord la décrire de façon théorique en nous reposant sur le concept de *type abstrait*, puis nous introduiront deux implémentations différentes à titre d'illustration.

15.1 Présentation

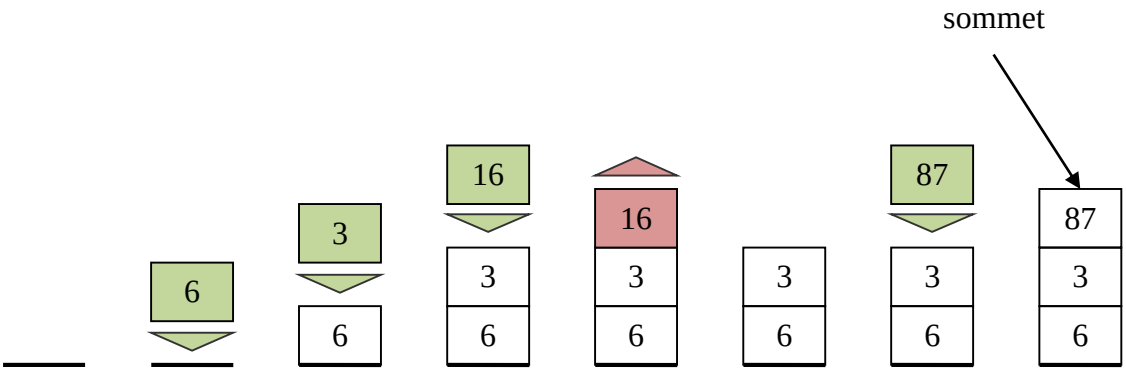
La *pile de données* (ou pile, pour faire plus court[22]) est une structure de données très simple mais aussi très répandue. En anglais, on parle de *stack*, ou *LIFO* (pour *Last In, First Out*). Tous ces noms viennent du mode de fonctionnement de la pile.

Une pile contient des informations *homogènes*, i.e. toutes de même type. Ce type peut être simple ou complexe. La particularité d'une pile est que seul le dernier élément inséré, qui est appelé le sommet, est accessible directement.

Sommet de la pile : dernier élément inséré dans la pile.

Quand un nouvel élément est inséré dans une pile, il est placé par-dessus l'ancien sommet, et en le recouvrant il devient lui-même le nouveau sommet. Cette opération est appelée l'*empilement*. L'opération inverse, consistant à supprimer un élément, est appelée le *dépilement*.

exemple : évolution d'une pile d'entiers dans laquelle on ajoute successivement les valeurs 6, 3 et 16, puis on supprime la valeur 16, puis on ajoute la valeur 87.



Remarque : comme dans la section 14 dédiée aux listes, on fera ici l'hypothèse qu'on veut manipuler des piles d'entiers. Cependant, on pourrait utiliser n'importe quel autre type, y compris complexe.

Comme pour les listes, on dira qu'une pile est *vide* si elle ne contient absolument aucun élément. Dans l'exemple précédent, la pile est initialement vide.

Le principe de pile est notamment utilisé :

- Dans les systèmes d'exploitation, pour gérer l'espace mémoire associé aux différents appels de fonction faits au sein d'un programme.
- Dans les compilateurs ou analyseurs syntaxiques, entre autres pour analyser des expressions bien parenthésées[23].

[22] Mais attention de ne pas confondre la structure de données et la partie de la mémoire qui porte le même nom et a été introduite en section 3.2.

[23] Cette utilisation fait l'objet de plusieurs TP inclus dans les deux autres volumes de support de cours.

- En intelligence artificielle, dans les algorithmes d'exploration d'arbres, pour effectuer par exemple un parcours en profondeur[24].

15.2 Type abstrait

15.2.1 Définition générale

Avant de donner une implémentation des piles de données, nous allons introduire la notion de type abstrait.

Type abstrait : définition axiomatique d'une structure de données.

On peut voir le type abstrait comme une spécification mathématique d'une structure de données, quelque chose de purement théorique. On se concentre seulement sur la description du comportement de la structure de données, et non pas sur la façon ont ce comportement va être réalisé. On peut dire qu'on se concentre sur la question Quoi ? en ignorant le Comment ?

L'intérêt du type abstrait est qu'il permet de donner une définition complètement indépendante de l'implémentation. La conséquence de cette propriété est qu'il est possible de proposer différentes implémentations différente pour la même structure de données. Toutes respectent le type abstrait, et produisent donc le même résultat. Mais chacune le fait avec ses propres avantages et inconvénients. Peut-être qu'une implémentation sera plus rapide dans certains cas, ou plus facile à implémenter, ou à maintenir, etc. De plus, on obtient une structure de donnée au comportement homogène, quel que soit le langage de programmation, la machine ou le système d'exploitation utilisé.

Type concret : résultat de l'implémentation d'un type abstrait.

Un type abstrait se compose d'un ensemble d'opérations et d'axiomes. Ils peuvent manipuler le type abstrait qu'on est en train de définir, ainsi que d'autres types prédéfinis.

En termes de langage C, ces opérations vont ensuite être implémentées sous la forme de fonctions applicables au type de données concerné. Les axiomes sont un ensemble de contraintes devant toujours être respectées, et qui précisent comment les fonctions devront se comporter.

15.2.2 Type abstrait *Piles de données*

Voici le type abstrait pour les piles de données. Chaque opération est listée avec son nom, sa description et son type (retour et paramètres).

Opérations

- `cree_pile` : créer une pile vide
  `_ → pile`
- `est_pile_vide` : déterminer si la pile est vide
  `pile → booleen`
- `sommet` : renvoyer le sommet de la pile
  `pile → element`
- `empile` : rajouter un élément dans une pile
  `pile × element → pile`
- `depile` : retirer un élément d'une pile
  `pile → pile`

L'opération `cree_pile` est capable de créer une nouvelle pile vide, sans prendre aucun paramètre d'entrée (d'où le signe _). Au contraire, `est_pile_vide` a besoin de recevoir une

[24] Ceci fait aussi l'objet d'un TP portant sur l'exploration de labyrinthe.

pile afin de tester si elle est vide ou pas, d'où la présence du type `pile`. Cette fonction renvoie soit vrai, soit faux, donc un booléen. On sait que le langage C ne contient pas de type booléen à proprement parler, et que ces valeurs seront représentées par des entiers. Cependant, cela relève de ce langage de programmation en particulier, et c'est donc un point qui est complètement ignoré lors de la spécification du type abstrait.

L'opération `sommet` reçoit une pile et renvoie son sommet, qui est un élément. L'opération `empile` a besoin de deux paramètres : l'élément à empiler, et la pile qui va le recevoir. Elle renvoie cette même pile augmentée de l'élément concerné. L'opération `depile` reçoit une pile, retire son sommet, et renvoie cette pile diminuée d'un élément.

Les axiomes sont les contraintes que les opérations doivent respecter :

Axiomes

∀ `e,p` ∈ `element` × `pile`, `p` étant non-vide :

- `est_pile_vide` :
  1. `est_pile_vide(cree())` = `vrai`
  2. `est_pile_vide(empile(p,e))` = `faux`
- `sommet` :
  3. `sommet(cree())` = indéfini (i.e. opération interdite)
  4. `sommet(empile(p,e))` = `e`
- `depile` :
  5. `depile(cree())` = indéfini
  6. `depile(empile(p,e))` = `p`

L'axiome 1 indique qu'une pile qui vient d'être créée est forcément vide. L'axiome 2, au contraire, dit qu'une pile dans laquelle on a empilé au moins un élément ne peut pas être vide.

L'axiome 3 stipule qu'on ne peut pas accéder au sommet d'une pile vide : cette opération n'est pas définie. Au contraire, l'axiome 4 dit que le sommet d'une pile est le dernier élément qui a été empilé.

L'axiome 5 indique qu'on ne peut pas dépiler une pile vide (opération indéfinie). Au contraire, l'axiome 6 dit que l'élément dépile d'une pile est le dernier élément à y avoir été empilé.

Toute implémentation délivrant ces opérations tout en respectant ces contraintes peut être considéré comme un type concret du type abstrait *pile de données*. Nous allons décrire deux implémentations possibles : l'une basée sur un tableau, l'autre sur une liste chaînée.

15.3 Implémentation par tableau

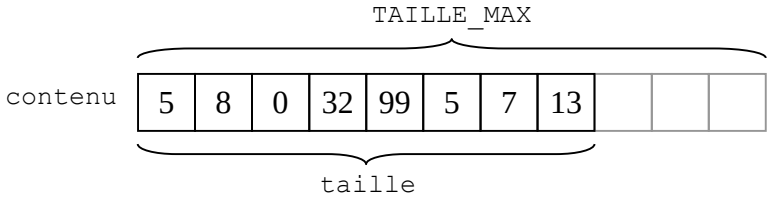
15.3.1 Structure de données

Pour implémenter une pile avec un tableau, on a besoin de connaître en permanence la taille de la pile, i.e. combien de cases du tableau sont occupées par des éléments de la pile. La meilleure solution est d'utiliser un type structuré incluant à la fois :

- Le tableau qui représente la pile ;
- L'entier qui représente sa taille.

exemple : une pile d'entiers

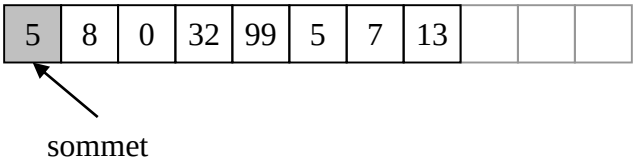
```
typedef struct
{  int taille;
   int contenu[TAILLE_MAX];
} pile;
```



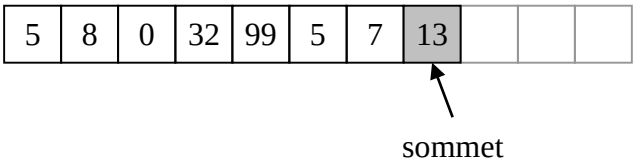
Remarque : `TAILLE_MAX` est bien sûr la taille maximale de la pile.

Dans notre implémentation, on a le choix entre deux possibilités pour manipuler la pile :

- Soit on considère que le sommet est au *début* du tableau (i.e. à gauche) :



- Soit on considère qu'il est à la *fin* de la partie du tableau occupée par la pile (i.e. à droite) :



Quand on insère ou supprime un élément dans une pile, on travaille toujours sur le sommet. La première solution implique donc de décaler l'intégralité des valeurs de la pile à chaque empilement ou dépilement. Au contraire, la deuxième solution évite ce travail, c'est pourquoi c'est celle que nous retiendrons.

À cause de la nature statique de l'implémentation par tableau (par opposition à la nature dynamique d'une implémentation par liste chaînée), on a une limitation de taille. On a donc une contrainte d'implémentation, qui vient se rajouter aux contraintes énoncées dans les axiomes du type abstrait : on ne doit pas empiler dans une pile qui est déjà pleine (i.e. qui a déjà atteint sa taille maximale). Pour cette raison, on rajoute une fonction `est_pile_pleine` dans cette implémentation du type abstrait `pile`.

Remarque : on pourrait utiliser une zone de mémoire allouée dynamiquement, au lieu d'un tableau statique. Cela constituerait une autre implémentation du type abstrait, qui n'aurait théoriquement pas de limite de taille (autre que la taille physique de la mémoire). On aurait aussi d'autres points d'implémentation à résoudre, par exemple : faut-il désallouer la mémoire à chaque fois qu'on dépile ? Nous laissons cette implémentation en exercice.

15.3.2 Création

Avec la structure de donnée à base de tableau, on crée une pile vide simplement en déclarant une variable et en initialisant son champ `taille` à 0. Il n'est pas nécessaire d'initialiser les valeurs du tableau, puisque celles-ci seront écrasées quand on empilera des éléments.

La fonction de création d'une pile est donc la suivante :

```
pile cree_pile()
{  pile p;
   p.taille = 0;
   return p;
}
```

Les deux fonctions de test sont également très simples. La pile sera *vide* si sa taille est nulle :

```
int est_pile_vide(pile p)
{  return (p.taille == 0);
}
```

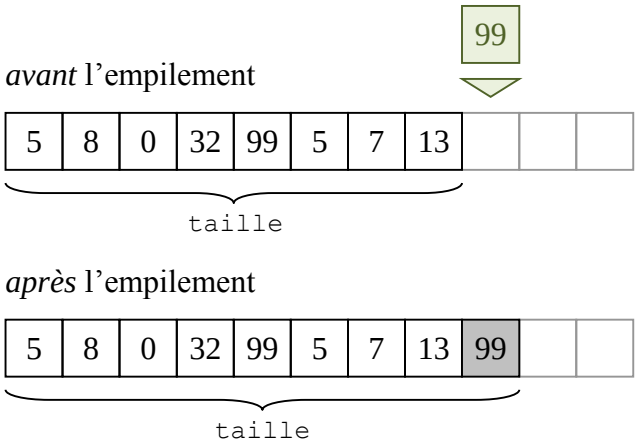
Et elle sera *pleine* si elle a atteint sa taille maximale, représentée par la constante `TAILLE_MAX` :

```
int est_pile_pleine(pile p)
{  return (p.taille == TAILLE_MAX);
}
```

15.3.3 Empilement

L'empilement se fait simplement en :

1. Rajoutant un élément à la fin de notre représentation de la pile ;
2. Mettant à jour la taille de la pile.



Soit `int empile (pile *p, int v)` la fonction qui empile la valeur `v` dans la pile pointée par `p`. La fonction renvoie -1 si elle n'a pas pu empiler la valeur ou 0 si l'empilement s'est bien passé.

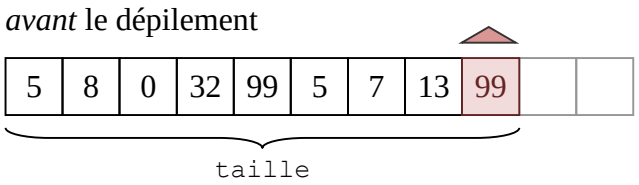
```
int empile (pile *p, int v)
{  int erreur = 0;

   if(est_pile_pleine(*p))
      erreur = -1;
   else
   {  p->contenu[p->taille] = v;
      p->taille ++ ;
   }

   return erreur;
}
```

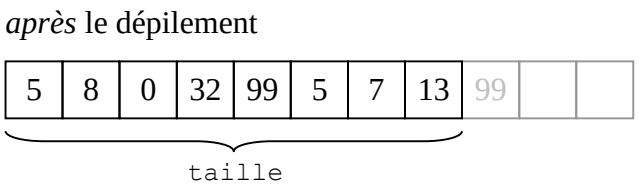
Remarque : la fonction définie ici renvoie un entier, ce qui n'est pas prévu par le type abstrait. Il s'agit d'un ajout dû à une contrainte d'implémentation, et aussi à une convention du C (renvoyer un code d'erreur en cas de problème). Il faut souligner que cette modification n'empêche pas la fonction de respecter les axiomes du type abstrait.

15.3.4 Dépilement

Le dépilement consiste à :

1. Supprimer le dernier élément de notre représentation de la pile ;
2. Mettre à jour la taille de la pile.





Soit `int depile (pile *p)` la fonction qui dépile la pile pointée par `p`. La fonction renvoie −1 si elle n'a pas pu dépiler ou 0 si le dépilement s'est bien passé.

```
int depile (pile *p)
{  int erreur = 0;

   if(est_pile_vide(*p))
      erreur = -1;
   else
      p->taille -- ;

   return erreur;
}
```

Là encore, l'entier renvoyé est un ajout par rapport au type abstrait, mais il n'interfère pas avec les axiomes.

15.3.5 Accès au sommet

Pour accéder au sommet, il suffit de renvoyer le dernier élément de notre représentation de la pile.

Soit `int sommet(pile p, int *v)` la fonction qui permet d'accéder au sommet de la `pile p`. La fonction transmet la valeur par adresse, grâce au paramètre `v`. La fonction renvoie −1 en cas d'erreur (si la pile est vide) ou 0 sinon.

```
int sommet(pile p, int *v)
{  int erreur = 0;

   if(est_pile_vide(p))
      erreur = -1;
   else
      *v = p.contenu[p.taille-1];

   return erreur;
}
```

15.4 Implémentation par liste chaînée

15.4.1 Structure de données

Les limitations d'une implémentation de pile par tableau sont les mêmes que celles que nous avions relevées quand nous avions comparé les tableaux et les listes en section 14.1 : taille limitée *a priori*, nombreux déplacements de données. L'intérêt de représenter une pile avec une liste chaînée est de résoudre tous ces problèmes.

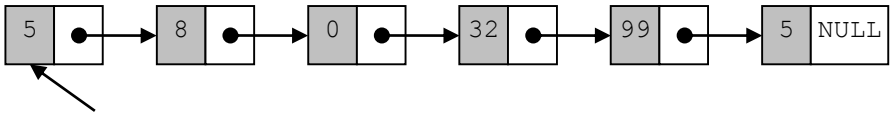
Avec une pile, on accède toujours à un seul bout de la liste. Donc, on peut utiliser une liste *simplement* chaînée, car on n'aura jamais besoin de parcourir la liste dans les deux sens. Le type `pile` se définit alors simplement en renommant le type `liste` :

```
typedef liste pile;
```

Autrement dit : notre pile est directement une liste. Par opposition, notre implémentation à base de tableau nécessitait non seulement le tableau lui-même, mais aussi la taille de la pile.

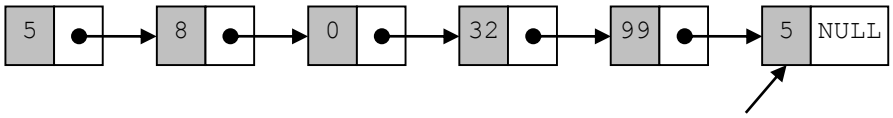
Comme pour l'implémentation par tableaux, la question se pose de savoir si on va considérer que le sommet de la pile est situé au *début* ou à la *fin* de la liste chaînée :

sommet au *début*



sommet

sommet à la *fin*



sommet

Si on choisit la seconde solution, on va devoir parcourir la liste en entier pour accéder au sommet, c'est-à-dire à chaque fois qu'on voudra effectuer un empilement ou un dépilement. Au contraire, si on considère que le sommet est situé au début de la liste, on n'aura qu'à faire des insertion/suppression de l'élément situé en tête de liste : on retiendra donc la première méthode.

15.4.2 Création

La création de pile revient tout simplement à créer une liste :

```
pile cree_pile()
{  liste l;
   l.debut = NULL;
   return l;
}
```

La pile sera donc vide ssi la liste est vide :

```
int est_pile_vide(pile p)
{  return (p.debut == NULL);
}
```

15.4.3 Empilement

L'empilement se fait simplement en rajoutant un élément au début de la liste. Soit `int empile (pile *p, int v)` la fonction qui empile la valeur `v` dans la pile pointée par `p`. La fonction renvoie -1 si elle n'a pas pu empiler la valeur ou 0 si l'empilement s'est bien passé.

```
int empile (pile *p, int v)
{  int erreur;
   element *e;

   e = cree_element(v);
   if(e == NULL)
      erreur = -1;
   else
      erreur = insere_element(p, e, 0);
```

```
    return erreur;
}
```

Remarquez que la fonction se charge d'abord de créer un nouvel élément, ce qui peut provoquer une erreur si l'allocation dynamique rencontre un problème (par exemple, s'il n'y a pas assez de mémoire libre). Autrement, on insère au début de la liste grâce à la fonction `insere_element` définie en section 14.2.5.

15.4.4 Dépilement

Le dépilement est réalisé en supprimant le sommet, c'est-à-dire le premier élément de la liste. Soit `int depile(pile *p)` la fonction qui dépile la pile pointée par `p`. La fonction renvoie −1 si elle n'a pas pu supprimer l'élément à la position demandée ou 0 si la suppression s'est bien passée.

```
int depile(pile *p)
{  int erreur = supprime_element(p, 0);
   return erreur;
}
```

On se contente d'appliquer `supprime_element` (section 14.2.6) pour sortir le premier élément de la liste. Si cette fonction rencontre une erreur (par exemple parce que la liste est vide), alors elle renverra −1 et ce code d'erreur sera propagé par `depile`.

15.4.5 Accès au sommet

Pour accéder au sommet, on renvoie la valeur du premier élément de notre représentation de la pile. Soit `int sommet(pile p, int *v)` la fonction qui permet d'accéder au sommet de la pile `p`. La fonction transmet la valeur par adresse grâce à la variable `v`. La fonction renvoie −1 en cas d'erreur ou 0 sinon.

```
int sommet(pile p, int *v)
{  int erreur = 0;
   element *e ;

   e = accede_element(p, 0);
   if(e == NULL)
      erreur = -1;
   else
      *v = e->valeur;

   return erreur;
}
```

16 Files de données

Comme pour la pile de données (section 15), nous allons décrire les files de donnés par le biais du type abstrait, avant d'en proposer différentes implémentations.

16.1 Présentation

À l'instar de la pile, la file de données est très répandue dans différents domaines de l'informatique. En anglais, on parle de *queue*, ou *FIFO* (pour *First In First Out*). Tous ces noms viennent du fonctionnement de la file, qui marche comme une file d'attente : lorsqu'on insère un élément, il va faire la queue à la fin de la file. Lorsqu'on on prend un élément, il s'agit du premier, car c'est celui qui attend depuis le plus longtemps.

Remarque : attention à ne pas confondre le mot français *file* avec le mot anglais *file*, qui désigne un fichier.

Comme la pile, la file et homogène : tous ses éléments contiennent une valeur du même type, qui peut être aussi bien simple qu'homogène. Comme la pile, on n'a accès qu'à un seul élément, mais cette fois il s'agit de celui qui a été inséré depuis le *plus* longtemps, alors qu'avec la pile il s'agissait de celui qui avait été inséré depuis le *moins* longtemps.

On peut voir une file comme un tuyau : d'un côté, on peut entrer des données et de l'autre on peut sortir des données. La seule donnée qui peut être enlevée de la file est la plus ancienne, on l'appelle la tête :

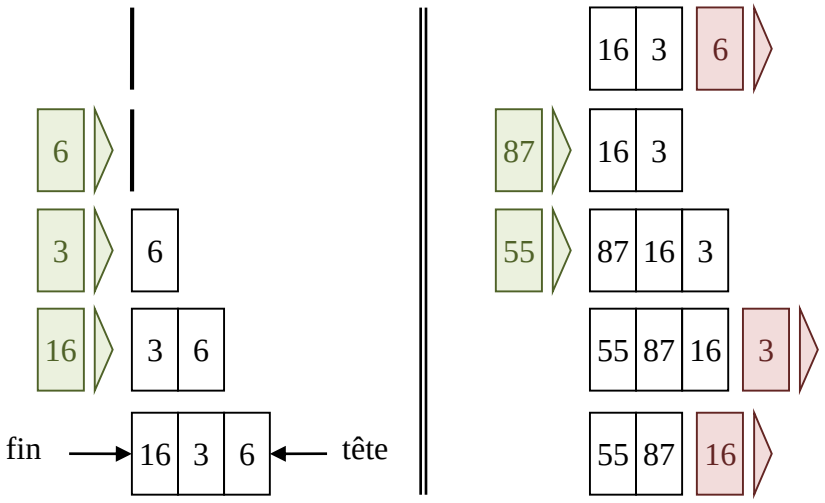
Tête de file : élément ayant été inséré depuis le *plus* longtemps.

Lors d'une insertion, le nouvel élément est placé après l'élément le plus récent, qui s'appelle la fin de la file :

Fin de file : élément ayant été inséré depuis le *moins* longtemps.

Dans une file, l'opération consistant à insérer un élément s'appelle l'*enfilement*, alors que l'opération inverse consistant à en retirer un est le *défilement*.

exemple : évolution d'une file d'entiers dans laquelle on insère 6, 3 et 16, puis on supprime 6, on insère 87 et 55, on supprime 3 et 16.



Le principe de file est notamment utilisé :

- Dans les systèmes d'exploitation, pour gérer les impressions (spouleurs d'impression), l'ordonnancement des processus, et plus généralement l'attente.
- En intelligence artificielle, dans les algorithmes d'exploration d'arbres de recherche, pour effectuer par exemple un parcours en largeur.

16.2 Type abstrait

Le type abstrait *file de données* se définit, comme pour la pile (cf. section 15.2.2), en spécifiant d'abord un ensemble d'opérations, puis les axiomes qu'elles doivent respecter.

Opérations

- `cree_file` : créer une file vide
  `_ → file`
- `est_vide` : déterminer si la file est vide
  `file → booleen`
- `tete` : renvoyer la tête de la file
  `file → element`
- `enfile` : rajouter un élément dans une file
  `file × element → file`
- `defile` : retirer un élément d'une file
  `file → file`

L'opération `cree_file` sert à créer une file vide, comme `cree_pile` pour les piles de données. L'opération `est_vide` teste si la file est vide, i.e. si elle ne contient aucun élément.

L'opération `tete` renvoie un pointeur sur le premier élément de la file, on peut la rapprocher de `sommet` pour la pile. L'opération `enfile` insère un élément à la fin de la file, tandis que `defile` supprime la tête de la file. On peut les comparer aux opérations `empile` et `depile` définies pour les piles de données.

Axiomes

∀ `e,f` ∈ `element` × `file`, `f` étant non-vide :

- `est_vide` :
  1. `est_vide(cree()) = vrai`
  2. `est_vide(enfile(f,e)) : faux`
- `tete` :
  3. `tete(cree())` = indéfini (i.e. opération interdite)
  4. `tete(enfile(f,e))` =
     - `e` si `est_vide(f)`
     - `tete(f)` si ¬`est_vide(f)`
- `defile` :
  5. `defile(cree())` = indéfini
  6. `defile(enfile(f,e))` =
     - `cree()` si `est_vide(f)`
     - `enfile(e,defile(f)` si ¬`est_vide(f)`

L'axiome 1 stipule qu'une file qui vient d'être créée est forcément vide. L'axiome 2 dit qu'au contraire, une file quelconque dans laquelle on a enfilé un élément ne peut pas être vide.

L'axiome 3 indique qu'il est indéfini de demander la tête d'une file qui est vide. L'axiome 4 dit que si la file n'est pas vide, on peut distinguer deux cas quand on en demande la tête. Si la file ne contient qu'un seul élément (1er cas listé), alors la tête correspond à cet élément. Sinon, la tête sera la même qu'on la demande avant ou après l'enfilement. Autrement dit, l'enfilement n'affecte pas la tête de la file (ce qui n'était pas le cas pour les piles).

L'axiome 5 interdit de défiler une file vide : en effet, si la file ne contient aucun élément, il est impossible d'en supprimer un. L'axiome 6 dit que si la file n'est pas vide, on doit distinguer deux cas. Soit la file ne contient qu'un seul élément, et alors le fait de la défiler produira une

liste vide. Soit la file contient plus d'un élément, et dans ce cas-là on obtient une file non-vide, sans qu'on puisse en dire plus à son sujet.

16.3 Implémentation simple par tableau

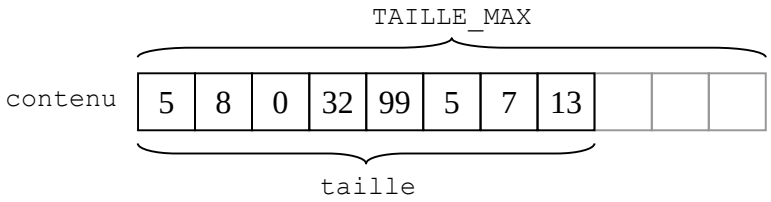
Comme pour les piles, on peut implémenter le type abstrait *file de données* en utilisant un tableau. Cependant, cette fois on peut distinguer deux implémentations différentes : l'une utilise le tableau comme on l'avait fait pour les piles, alors que l'autre adopte une approche *circulaire*.

16.3.1 Structure de données

Comme pour la pile, pour implémenter une file avec un tableau, on a besoin de connaître en permanence sa *taille*. Toujours comme pour la pile, on peut utiliser un type structuré incluant à la fois le tableau qui représente la file et l'entier qui représente sa taille.

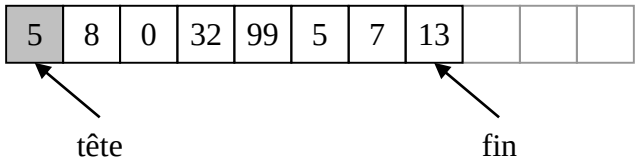
exemple : file d'entiers de taille maximale `TAILLE_MAX`

```
typedef struct
{  int taille;
   int contenu[TAILLE_MAX];
} file;
```

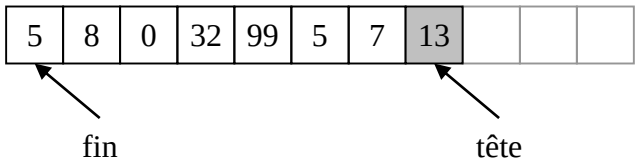


Comme avec les piles, dans notre implémentation, nous avons le choix entre deux possibilités pour manipuler la file :

- Soit on considère que la tête est au début du tableau :



- Soit on considère qu'elle est à la fin de la partie du tableau occupée par la file :



À la différence de la pile, ici on travaille sur les *deux* côtés du tableau, puisqu'on devra retirer les éléments du côté de la tête et rajouter les éléments de l'autre côté. Donc les deux implémentations sont comparables dans le sens où on devra gérer un décalage dans les deux cas : pour `defile` dans le premier cas, et pour `enfile` dans le second cas. À la différence de la pile, il n'y a pas ici de meilleur choix. Nous choisissons arbitrairement la première implémentation.

A l'instar de l'implémentation des piles, on a une contrainte supplémentaire liée à la nature statique des tableaux : on ne doit pas enfiler dans une file qui est déjà pleine (i.e. qui a déjà atteint sa taille maximale). Pour cette raison, on rajoute une fonction `est_file_pleine` dans cette implémentation du type abstrait *file de données*.

16.3.2 Création

Pour créer une file, on déclare simplement une variable de type file, et on n'oublie pas d'initialiser sa taille à 0. Comme pour la pile, il n'est pas nécessaire d'initialiser les valeurs du tableau, puisqu'elles seront écrasées quand on enfilera des valeurs.

```
file cree_file()
{  file f;
   f.taille = 0;
   return f;
}
```

La file sera *vide* si sa taille est nulle :

```
int est_file_vide(file f)
{  return (f.taille == 0);
}
```

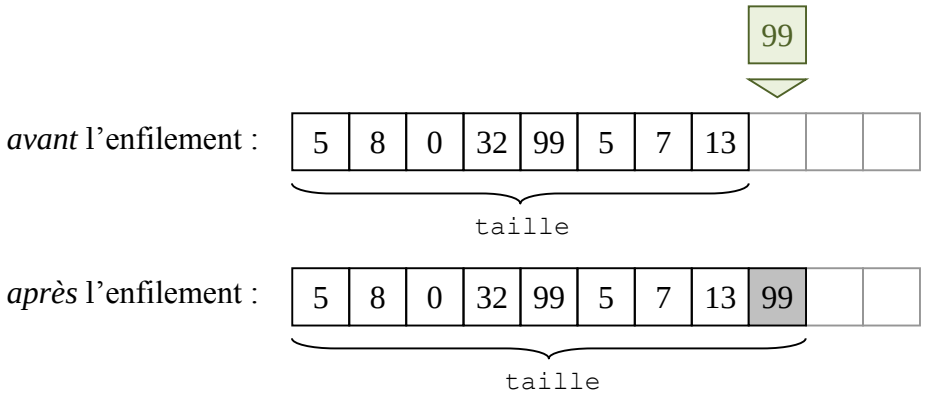
Elle sera *pleine* si elle a atteint sa taille maximale :

```
int est_file_pleine(file f)
{  return (f.taille == TAILLE_MAX);
}
```

16.3.3 Enfilement

L'enfilement consiste simplement à :

1. Copier l'entier à insérer à la fin de notre représentation de la file ;
2. Mettre à jour la taille de la file.



Soit `int enfile (file *f, int v)` la fonction qui enfile la valeur `v` dans la file pointée par `f`. La fonction renvoie -1 si elle n'a pas pu enfiler la valeur ou 0 si l'enfilement s'est bien passé.

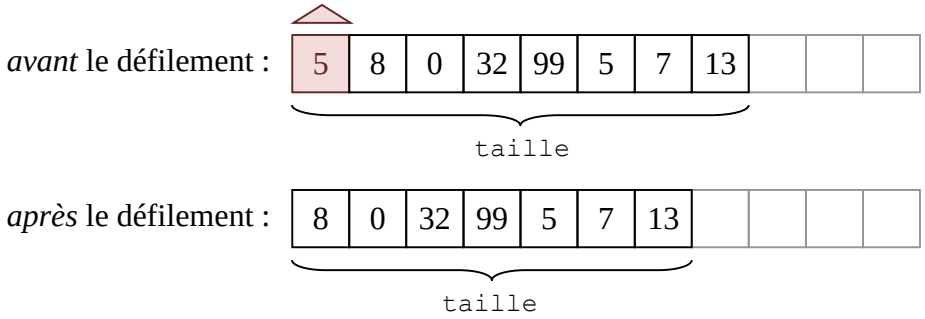
```
int enfile(file *f, int v)
{  int erreur = 0;

   if(est_file_pleine(*f))
      erreur = -1;
   else
   {  f->contenu[f->taille] = v;
      f->taille ++ ;
   }

   return erreur;
}
```

16.3.4 Défilement

Le défilement consiste à :

1. Décaler toutes les valeurs du tableau vers la gauche, supprimant ainsi la tête ;

2. Mettre à jour la taille de la file.



Soit `int defile (file *p)` la fonction qui défile la file pointée par `f`. La fonction renvoie −1 si elle n'a pas pu défiler ou 0 si le défilement s'est bien passé.

```
int defile(file *f)
{  int erreur = 0;

   if(est_file_vide(*f))
      erreur = -1;
   else
   {  for(i=0;i<f->taille-1;i++)
         f->contenu[i] = f->contenu[i+1];
      f->taille -- ;
   }

   return erreur;
}
```

16.3.5 Accès à la tête

Pour accéder à la tête, il suffit de renvoyer le premier élément du tableau.

Soit `int tete(file p, int *v)` la fonction qui permet d'accéder à la tête de la file `f`. La fonction transmet la valeur par adresse grâce à la variable `v`. La fonction renvoie −1 en cas d'erreur (si la file est vide) ou 0 sinon.

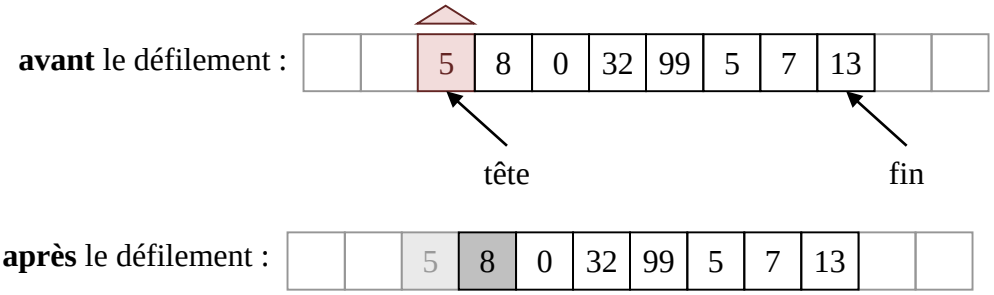
```
int tete(file f, int *v)
{  int erreur = 0;

   if(est_file_vide(f))
      erreur = -1;
   else
      *v = f.contenu[0];

   return erreur;
}
```

16.4 Implémentation circulaire par tableau

On sait que l'un des principaux problèmes de l'implémentation par tableau vient des nombreux déplacements de valeurs provoqués par chaque opération. Mais ici, il est possible d'éviter cela en considérant que le tableau est refermé sur lui-même. On parle alors d'implémentation *circulaire*.

16.4.1 Structure de données

Avec l'implémentation circulaire, le début de la file ne correspond plus forcément au début du tableau. Ceci entraine des complications dans l'implémentation des opérations. Mais l'intérêt est qu'en cas de défilement, au lieu de décaler toutes les valeurs de la file, on se contentera de modifier son début.



Ceci a aussi des conséquences sur la structure de données, car il va falloir représenter le début de la file. Pour cela, nous allons utiliser la structure de données modifiée suivante :

```
typedef struct
{  int debut;
   int taille;
   int contenu[TAILLE_MAX];
} file;
```

16.4.2 Création

Pour la création, on procède comme pour l'implémentation simple par tableau, en initialisant en plus le début de la file à 0.

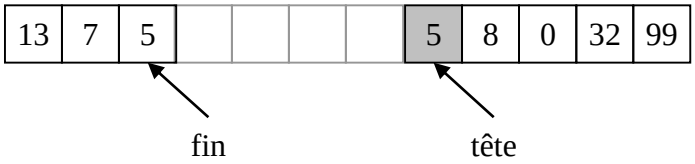
```
file cree_file()
{  file f;
   f.debut = 0;
   f.taille = 0;
   return f;
}
```

Les fonctions `est_file_vide` et `est_file_pleine` sont exactement les mêmes que dans l'implémentation simple par tableau, puisqu'elles reposent seulement sur l'utilisation du champ `taille`.

16.4.3 Enfilement

Pour insérer un nouvel élément, on ne peut plus procéder exactement comme dans l'implémentation simple par tableau, en copiant la nouvelle valeur dans la case numéro `taille` du tableau. En effet, le début de la file n'est pas forcément la case 0, donc le numéro de la case où on doit écrire la nouvelle valeur n'est pas forcément `0+taille`.

De plus, la file étant implémentée de manière circulaire, il est possible que le numéro de la case de fin soit *inférieur* au numéro de la case de début, comme dans l'exemple suivant :



On utilisera donc une formule plus générale pour calculer le numéro de la case où insérer la nouvelle valeur : `(debut+taille)%TAILLE_MAX`. L'opérateur modulo permet de se ramener à une valeur comprise entre 0 et `TAILLE_MAX-1`.

Soit `int enfile (file *f, int v)` la fonction qui enfile la valeur `v` dans la file pointée par `f`. La fonction renvoie -1 si elle n'a pas pu enfiler la valeur ou 0 si l'enfilement s'est bien passé.

```
int enfile(file *f, int v)
```

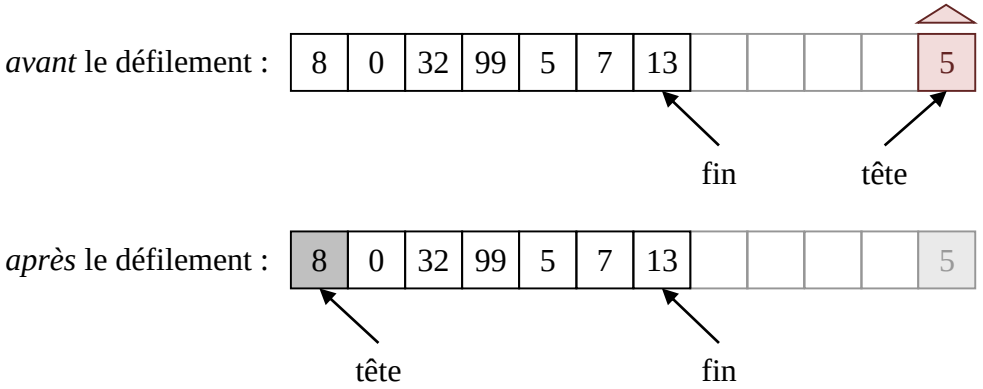
```
{  int erreur = 0;

   if(est_file_pleine(*f))
      erreur = -1;
   else
   {  f->contenu[(f->debut+f->taille)%TAILLE_MAX] = v;
      f->taille ++ ;
   }

   return erreur;
}
```

16.4.4 Défilement

Comme vu précédemment, le défilement va consister ici à :

1. Incrémenter le début de la file ;
2. Mettre à jour la taille de la file.

On peut avoir le cas particulier suivant :



Par conséquent, on utilisera une formule reposant sur le même principe que celle utilisée pour l'enfilement : le numéro de la nouvelle case de début de la file sera `(debut+1)%TAILLE_MAX`.

Soit `int defile (file *f)` la fonction qui défile la file pointée par `f`. La fonction renvoie -1 si elle n'a pas pu défiler ou 0 si le défilement s'est bien passé.

```
int defile(file *f)
{  int i, erreur = 0;

   if(est_file_vide(*f))
      erreur = -1;
   else
   {  f->debut=(f->debut+1)%TAILLE_MAX;
      f->taille -- ;
   }

   return erreur;
}
```

16.4.5 Accès à la tête

L'accès à la tête de la file se fait en utilisant le champ `debut`, et non plus la constante 0 comme dans l'implémentation simple.

Soit `int tete(file p, int *v)` la fonction qui permet d'accéder à la tête de la file `f`. La fonction transmet la valeur par adresse grâce à la variable `v`. La fonction renvoie -1 en cas d'erreur (si la file est vide) ou 0 sinon.

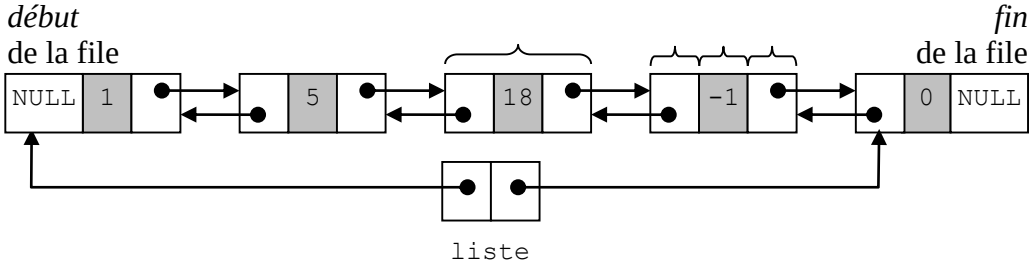
```
int tete(file p, int *v)
{  int erreur = 0;

   if(est_file_vide(f))
      erreur = -1;
   else
      *v = f.contenu[f.debut];

   return erreur;
}
```

16.5 Implémentation par liste chaînée

16.5.1 Structure de données

L'intérêt de représenter une file avec une liste chaînée est le même que pour les piles : la taille ne sera pas limitée *a priori*. De plus, on n'a pas besoin d'utiliser d'implémentation circulaire (à la différence d'une implémentation par tableau).

Cependant, on aura aussi besoin de manipuler à la fois le début (pour défiler) et la fin (pour enfiler) de la liste, donc nous utiliserons une liste *doublement* chaînée. Arbitrairement, on décide de placer la tête au début de la liste.



On utilise la structure de données classique déjà utilisée pour les piles :

```
typedef liste file;
```

16.5.2 Création

La création de file revient à créer une liste doublement chaînée et à initialiser ses deux pointeurs :

```
file cree_file()
{  liste l;
   l.debut = NULL;
   l.fin = NULL;
   return l;
}
```

Comme pour la pile, la file sera *vide* si la liste est vide :

```
int est_file_vide(file f)
{  return (f.debut == NULL);
}
```

16.5.3 Enfilement

L'enfilement se fait simplement en rajoutant un élément à la fin de la liste. On utilise pour cela le champ `fin` de `liste`.

Soit `int enfile (file *f, int v)` la fonction qui enfile la valeur `v` dans la file pointée par `f`. La fonction renvoie -1 si elle n'a pas pu enfiler l'élément ou 0 en cas de succès.

```
int enfile(file *f, int v)
```

```
{  int erreur = 0;
   element *e,*temp;

   e = cree_element(v);
   if(e == NULL)
      erreur = -1;
   else
   {  if(est_file_vide(*f))
         erreur = insere_element(f, e, 0);
      else
      {  temp = f->fin;
         temp->suivant = e;
         e->precedent = temp;
         f->fin = e;
      }
   }

   return erreur;
}
```

Remarque : dans le cas où on insère dans une liste non-vide (i.e. dernier cas traité dans la fonction ci-dessus), on pourrait aussi déterminer la position p du dernier élément contenu dans la liste, puis utiliser `insere_element` comme on l'a fait pour le cas où la liste est vide (juste au-dessus dans le code source). Cependant, cela impliquerait de parcourir deux fois la liste (une fois manuellement et une fois dans `insere_element`), ce qui ne serait pas très efficace. Le code source proposé permet d'éviter cet inconvénient.

16.5.4 Défilement

Le défilement est réalisé en supprimant le premier élément de la liste. Cette fois, on utilise le champ `debut` de `liste`.

Soit `int defile (file *f)` la fonction qui défile la file pointée par `f`. La fonction renvoie -1 si elle n'a pas pu défiler l'élément ou 0 en cas de succès.

```
int defile(file *f)
{  int erreur = supprime_element(f, 0);
   return erreur;
}
```

16.5.5 Accès à la tête

Pour accéder à la tête, on renvoie la valeur du premier élément de notre représentation de la file.

Soit `int tete(file f, int *v)` la fonction qui permet d'accéder à la tête de la file `f`. La fonction transmet la valeur par adresse grâce au paramètre `v`. La fonction renvoie -1 en cas d'erreur (si la file est vide) ou 0 sinon.

```
int tete(file f, int *v)
{  element *e;
   int erreur = 0;

   e = accede_element(f, 0);
   if(e == NULL)
      erreur = -1;
   else
      *v = e->valeur;
   return erreur;
}
```

17 Complexité algorithmique

Comme on l'a vu lors de l'implémentation des types abstraits, il est souvent possible d'implémenter plusieurs solutions à un problème donné. La question se pose alors de savoir quel algorithme est le plus efficace. Et pour effectuer cette comparaison, il est nécessaire de déterminer sur quels critères elle doit être effectuée. Le but de cette section est d'introduire le champ de la *théorie de la complexité*, dont l'un des buts est de répondre à ce type de question.

17.1 Introduction à la complexité

17.1.1 Motivation et principe

Théorie de la complexité : théorie développée dans le but de comparer efficacement différents algorithmes entre eux, Afin de savoir lequel était le meilleur pour un problème donné. Jusqu'aux années 70, on caractérisait les algorithmes de manière empirique, en considérant :

- Le temps d'exécution ;
- L'espace mémoire utilisé ;
- Les conditions d'exécution :
  - Taille des données
  - Type d'ordinateur
  - Système d'exploitation
  - Langage de programmation
  - Etc.

exemple :

Pour le problème consistant à calculer 12^{34}, sur un P4 à 3 GHz, avec un OS Windows XP et une implémentation en langage C, supposons que l'algorithme A met 1,2 secondes et utilise 13 Mo de mémoire, alors que l'algorithme B met 1,3 secondes et utilise 0,8 Mo. Alors on peut dire que A est plus rapide que B, mais que B est moins gourmand en mémoire.

Le problème de cette démarche est qu'elle ne permet pas de comparer les algorithmes de manière objective, car ils ne sont pas les seuls facteurs intervenant dans la performance de résolution du problème : d'autres facteurs extérieurs interviennent également (matériel, logiciel...).

La solution retenue est de se détacher de ses facteurs extérieurs, et donc de l'implémentation de l'algorithme. Pour cela, nous allons utiliser une approche plus théorique, reposant sur deux notions : les opérations élémentaires et les positions mémoire.

Opération élémentaire : opération atomique, correspondant à une instruction assembleur.

Rappelons qu'une instruction assembleur est atomique, dans le sens où elle correspond à un code directement interprétable par le processeur. Autrement dit, une instruction assembleur est associée à une action que le processeur peut réaliser, comme par exemple placer une certaine valeur dans à une certaine adresse en mémoire. Par comparaison, une instruction du langage C se décompose généralement en plusieurs instructions assembleur.

La notion d'opération élémentaire va être utilisée pour représenter la consommation que l'algorithme analysé effectue en termes de temps de calcul. On dira par exemple qu'il a besoin d'effectuer x opérations élémentaires pour résoudre le problème traité.

Position mémoire : unité de mémoire élémentaire, correspondant généralement à un octet.

La notion de position mémoire est une abstraction de l'occupation qu'un algorithme a de la mémoire. Elle va nous permettre d'évaluer cette consommation de façon indépendante de

certaines conditions d'exécution mentionnées précédemment, en considérant le nombre de positions mémoires qu'il utilise.

Ces deux notions nous permettent d'exprimer la *complexité* d'un algorithme en fonction de la *taille des données* qu'il traite.

Taille des données d'un problème : entier(s) représentant la grandeur des paramètres reçus par l'algorithme devant résoudre le problème.

Notez bien que la taille peut n'être représentée par un seul entier, mais aussi par plusieurs entiers distincts. Leur nombre et leur signification exacte dépendent fortement de la nature du problème étudié :

- Nombre d'éléments à traiter ;
- Grandeur des éléments à traiter ;
- etc.

exemples :

- Tri d'un tableau de taille N : la taille des donnés est N, le nombre d'éléments du tableau.
- Calcul de C_n^p : la taille des données est le couple (n, p).

L'idée est que si la taille est petite, le problème sera vraisemblablement plus facile à résoudre que si elle est énorme. On veut généralement savoir comment la *complexité* de l'algorithme évolue quand on fait grandir la taille du problème.

Complexité d'un algorithme : nombre d'opérations élémentaires ou de positions mémoire dont l'algorithme a besoin pour résoudre un problème d'une certaine taille.

Formellement, l'expression d'une complexité prend la forme d'une fonction mathématique de la taille des données.

exemples :

- Si le tri du tableau a une complexité $f(N) = an + b$, on dira que le tri possède une complexité *linéaire*.
- Si un autre algorithme de tri possède une complexité $f(N) = n^2$, alors on considèrera que ce tri a une complexité *quadratique*.
- La complexité du second tri est supérieure à celle du premier.

17.1.2 Complexités spatiale et temporelle

On distingue deux types de complexités : temporelle et spatiale.

Complexité temporelle : nombre total d'opérations élémentaires pour exécuter l'algorithme.

La complexité temporelle correspond donc au décompte de toutes les opérations élémentaires que l'algorithme a besoin d'effectuer pour résoudre le problème.

Complexité spatiale : nombre *maximal* de positions mémoire utilisées au cours de l'exécution.

À la différence des opérations élémentaire, qui s'enchaîne de façon séquentielle, l'occupation mémoire est une quantité qui évolue au cours du temps : en fonction de l'état de la pile (appels de fonction) et du segment de données (allocation dynamique) l'algorithme peut augmenter et diminuer le nombre de positions mémoire qu'il utilise. La complexité spatiale correspond à l'occupation *maximale* atteinte par l'algorithme *au cours de son exécution*.

Attention donc à ne pas considérer le total de toutes les positions mémoires occupées au cours de l'exécution, ou bien juste l'occupation mémoire observée juste avant que l'algorithme se termine.

exemple : 2 fonctions différentes calculant le produit des nombres d'un tableau d'entiers.

```
   int produit1(int tab[N])
1  {   int i;
2      int resultat=1;
3      for(i=0;i<N;i++)
4         resultat = resultat*tab[i];
5      return resultat;
   }

   int produit2(int tab[N])
1  {   int i=0;
2      int resultat=1;
3      while(resultat!=0 && i<N)
4      {  resultat = resultat*tab[i];
5         i++;
       }
6      return resultat;
   }
```

- Taille des données : N
- Complexités spatiales :
  - Supposons qu'un entier occupe m positions mémoire.
  - Alors pour les deux fonctions, on a :
    - Tableau de N entiers.
    - Variables `i` et `resultat`.
    - total : $S_1(N) = S_2(N) = m(2 + N)$
- Complexités temporelles :
  - On suppose que chaque opération/instruction correspond à un certain nombre d'opérations élémentaires :
    - multiplication : *a*.
    - addition : *b*.
    - comparaison : *c*.
    - affectation : *d*.
    - instruction `return` : *e*.
    - et logique : *f*.
  - Première fonction :
    - `2` : affectation de `resultat` (*d*).
    - `3` : affectation de `i` (*d*).
    - Dans le `for` :
      - `3` : comparaison `i<N` (*c*) et incrémentation de `i` (une addition (*b*) et une affectation (*d*)).
      - `4` : opération `*` (*a*) et affectation à `resultat` (*d*).
    - Nombre de répétitions du `for` : N
    - `3` : comparaison du `for` à la sortie de la boucle (*c*).
    - Retour de la valeur `resultat` (*e*).
    - On a : $d + d + N(c + b + d + a + d) + c + e = 2d + N(a + b + c + 2d) + c + e$
    - Total : $T_1(N) = Na + Nb + (N + 1)c + (N + 1)2d + e$
  - Seconde fonction :
    - `1` : affectation de `i` (*d*).
    - `2` : affectation de `resultat` (*d*).
    - Dans le `while` :
      - `3` : comparaisons `resultat!=0` (*c*) et `i<N` (*c*), opération `&&` (*f*).
      - `4` : opération `*` (*a*) et affectation à `resultat` (*d*).

- `5` : incrémentation de `i` (une addition (b) et une affectation (d)).

- Nombre de répétitions du `while` :
  - M si le tableau contient un zéro en $(M-1)^{\text{ème}}$ position.
  - N si le tableau ne contient pas de zéro.
- `3` : comparaisons du `while` à la sortie de la boucle :
  - 1 comparaison `resultat!=0` (c) si le tableau contient un zéro.
  - 2 comparaisons (c) et un `&&` (f) si le tableau ne contient pas de zéro.
- Retour de la valeur `resultat` (e).
- on a :
  - $d + d + M(c + f + c + a + d + b + d) + c + e = 2d + M(a + b + 2c + 2d + f) + c + e$
  - $d + d + N(c + f + c + a + d + b + d) + c + f + c + e = 2d + N(a + b + 2c + 2d + f) + 2c + f + e$
- Total :

$$T_2(N) = \begin{cases} (Ma + Mb + (2M+1)c + (M+1)2d + e + Mf)\ si\ \exists M: tab[M-1] = 0 \\ (Na + Nb + 2(N+1)c + (N+1)2d + e + (N+1)f)\ sinon \end{cases}$$

Remarque : il existe un lien entre les complexités spatiale et temporelle d'un algorithme. D'abord, sur une machine mono-processeur (i.e. ne possédant qu'un seul processeur, et donc capable d'exécuter une seule instruction à la fois), la complexité spatiale ne peut pas dépasser la complexité temporelle, puisqu'il faut effectuer *au moins* une opération pour initialiser une position mémoire.

De plus, il est souvent possible de faire diminuer la complexité temporelle en augmentant la complexité spatiale, et vice-versa. Par exemple, supposons qu'on réalise le calcul d'une quantité $x = f\big(h(x)\big) + g\big(h(x)\big)$, où. Supposons que h est une fonction difficile à calculer. On peut voir deux possibilités ici :

- Soit on calcule $h(x)$ deux fois, et cela risque de prendre longtemps ;
- Soit on définit une variable $y = h(x)$, ce qui augmente l'occupation mémoire, mais diminue le temps de calcul.

17.1.3 Opérations en temps constant

Même si on a progressé par rapport à la méthode empirique, la comparaison des complexités temporelles reste difficile, car on dépend encore trop des machines utilisées. En effet, le nombre d'opération élémentaires associé à une instruction/opération est dépendant de la machine utilisée. Par exemple, une multiplication peut correspondre à une seule opération élémentaire sur une machine, et à plusieurs sur une autre machine (par exemple une séquence d'additions).

De la même façon, pour la complexité spatiale, la représentation des données en mémoire varie suivant le langage et la machine. Par exemple, le nombre d'octets utilisés pour représenter un entier `int` en langage C n'est pas standard.

Déterminer certaines valeurs pose donc problème :

- Nombre d'opérations élémentaires associé à une instruction/opération ;
- Nombre d'emplacements mémoire associé à un type de donnée.

On n'a pas la possibilité de les connaître en général, i.e. de déterminer les valeurs des $a, \dots, f$ utilisés dans l'exemple précédent.

Une solution consiste à réaliser les simplifications, suivantes :

- Une instruction/opération correspond à un nombre constant c d'opérations élémentaires.
- Une variable de type simple (entier, booléen, réel, etc.) occupe un nombre constant c d'emplacements mémoire.
- Un ensemble de variables de type simple occupe un nombre de positions mémoire dépendant de la taille des données :
  - `tab[N]` occupe Nc positions.
  - `mat[N][M]` en occupe $(N \times M)c$.
  - Etc.

On parle alors d'*opérations en temps constant* et d'*emplacements en espace constant*. Comme nous le verrons plus tard, cette simplification ne porte pas à conséquence dans le cadre de la comparaison d'algorithmes.

exemple : supposons que toutes les instructions/opérations nécessitent c opérations élémentaires. Les complexités des fonctions de l'exemple deviennent alors respectivement :

- $T_1(N) = Nc + Nc + (N+1)c + (N+1)2c + c = (5N+4)c$.
- $T_2(N) = \begin{cases} Mc + Mc + (2M+1)c + (M+1)2c + c + Mc = (7M+4)c \\ Nc + Nc + 2(N+1)c + (N+1)2c + c + (N+1)c = (7N+6)c \end{cases}$

On peut alors facilement comparer les complexités spatiales des algorithmes :

- si $M > \frac{5}{7}N$, c'est-à-dire : s'il n'y a pas de zéro dans le tableau, ou bien s'il y a un zéro dans les deux derniers septièmes du tableau, alors `produit1` est plus efficace que `produit2`, car $T_1(N) < T_2(N)$.
- si $M < \frac{5}{7}N$, c'est-à-dire s'il y a un zéro dans les cinq premiers septièmes du tableau, alors `produit2` est plus efficace que `produit1`.

17.1.4 Cas favorable, moyen et défavorable

Dans l'exemple précédent, la fonction `produit2` possède une complexité temporelle qui varie en fonction des données d'entrée (i.e. le tableau à traiter). En effet, suivant la présence et la position du zéro, la fonction peut s'arrêter avant d'avoir atteint la fin du tableau.

En fonction des données traitées, on peut distinguer trois situations :

- *Meilleur* des cas : le zéro est situé au *début* du tableau → la fonction s'arrête tout de suite.
- *Pire* des cas : il n'y a *pas* de zéro dans le tableau, ou bien à la fin → la fonction parcourt le tableau jusqu'à la fin.
- Les *autres* cas : un zéro est présent dans le tableau (ailleurs qu'au début ou à la fin) → la fonction n'ira pas jusqu'au bout du tableau.

Sur ce principe, un algorithme pourra caractérisé en considérant sa complexité dans trois situations différentes, jugées particulièrement intéressantes : meilleur des cas, pire des cas, et en moyenne.

Complexité dans le meilleur des cas : complexité de l'algorithme quand les données traitées aboutissent à la complexité minimale. On parle aussi de cas *favorable*.

On rencontre rarement cette complexité, car le cas le plus favorable rend souvent trivial le problème traité, et n'est pas très discriminant (i.e. il ne permet pas de comparer les algorithmes de façon pertinente). De plus, ce cas est en général rare parmi l'ensemble des entrées possibles du domaine.

Complexité dans le pire des cas : complexité de l'algorithme quand les données traitées aboutissent à la complexité maximale. On parle aussi de cas *défavorable*.

Il s'agit de la complexité que l'on rencontre le plus souvent dans la littérature quand on veut décrire un algorithme. En effet, même si le cas défavorable est susceptible d'être rare, à l'instar du cas favorable, en revanche il est plus discriminant. Il est aussi plus pertinent d'obtenir une borne supérieure du temps d'exécution ou de l'occupation mémoire.

Complexité moyenne : complexité de l'algorithme moyennée sur l'ensemble de toutes les données possibles.

On peut considérer que c'est la complexité la plus représentative de l'algorithme. Cependant, elle est généralement difficile à calculer, et on ne la rencontre donc pas fréquemment.

exemple : on considère la fonction `produit2`

- Pire des cas : $T_2(N) = (7N + 6)c$
- Meilleur des cas ou cas favorable : $T_2(N) = (7 \times 1 + 4)c = 11c$
- Cas moyen :
  - $N + 1$ cas sont possibles :
    - Soit pas de zéro.
    - Soit zéro dans une position parmi $0; \ldots; (N - 1)$.
  - Si on suppose que ces $N + 1$ cas sont équiprobables, alors :

$$T_2(N) = \frac{(7N + 6)c + \sum_{i=1}^{N}(7 \times i + 4)c}{N + 1}$$

17.1.5 Notation asymptotique

La question que l'on se pose généralement quand on étudie un algorithme est : Que se passe-t-il si la taille des données est doublée ? Ceci amène tout une série de questions supplémentaires, comme : Est-ce que le temps de calcul est lui aussi doublé ? Ou plus ? Ou moins ? Même chose pour l'espace mémoire occupé ? Et que se passe-t-il si la taille est triplée ?

Pour résumer, on s'intéresse au comportement de l'algorithme quand la taille des données tend vers l'infini. Plus exactement, on s'intéresse à l'ordre de grandeur de sa complexité. Pour la représenter, on utilisera certaines des notations asymptotiques de Landau.

Rappelons d'abord formellement ces notations, avant de montrer comment elles peuvent être utilisées dans le cadre de la complexité algorithme. Soient f et g deux fonctions définies de $\mathbb{N}$ dans $\mathbb{N}$:

notation	définition	signification
$f \in O(g)$	$\exists a > 0, \exists x_0 > 0$ $\forall x \geq x_0 : f(x) \leq a \times g(x)$	f croît moins vite (ou aussi vite) que g (g est une borne supérieure asymptotique de f)
$f \in \Omega(g)$	$\exists a > 0, \exists x_0 > 0$ $\forall x \geq x_0 : f(x) \geq a \times g(x)$	f croît plus vite (ou aussi vite) que g (g est une borne inférieure asymptotique de f)
$f \in \Theta(g)$	$f \in O(g) \wedge f \in \Omega(g)$	f et g croissent à la même vitesse (g est une borne approchée asymptotique de f)

exemples : soient les fonctions $f(x) = 3x^2 + 1$, $g_1(x) = 4x$, $g_2(x) = x^2 + 4$, $g_3(x) = x^3$. On a alors :

- $f \in O(g_2)$: prenons $a = 3$ et $x_0 = 1$.
  - $f(x) = 3 \times 1^2 + 1 = 4$.
  - $3 \times g_2(x) = 3(1^2 + 4) = 7$.
- $f \in O(g_3)$: prenons $a = 1$ et $x_0 = 4$:
  - $f(x) = 3 \times 4^2 + 1 = 49$.

  - $g_3(x) = 4^3 = 64$.
- $f \in \Omega(g_1)$: prenons $a = 1$ et $x_0 = 1$:
  - $f(x) = 3 \times 1^2 + 1 = 4$.
  - $g_1(x) = 4 \times 1 = 4$.
- $f \in \Omega(g_2)$: prenons $a = 1$ et $x_0 = 2$:
  - $f(x) = 3 \times 2^2 + 1 = 13$.
  - $g_2(x) = 2^2 + 4 = 8$.
- $f \in \Theta(g_2)$ car $f \in \Omega(g_2)$ et $f \in O(g_2)$.

Dans ce cours, on utilisera essentiellement la notation *en grand* O, qui permet de majorer le comportement d'une fonction. Bien sûr, on veut obtenir la borne supérieure de degré le plus petit possible. La notation *en* Θ sera également utilisée, plus rarement, quand il est possible de borner la complexité de l'algorithme de manière plus précise.

On distingue habituellement différentes familles d'algorithmes en fonction de leur borne supérieure :

- Complexité constante : $O(1)$.
- Complexité logarithmique : $O(\log x)$.
- Complexité log-linéaire : $O(x \log x)$.
- Complexité linéaire : $O(n)$.
- Complexité quadratique : $O(x^2)$, cubique $O(n^3)$, ou plus généralement : polynômiale $O(x^k)$.
- Complexité exponentielle $O(k^x)$.

Remarque : en informatique, quand la base d'un logarithme n'est pas précisée, c'est qu'il s'agit d'un logarithme de base 2.

Pour simplifier le calcul de complexité, on utilise les abus de notation suivants :

- $f \in O(g)$ sera noté $f = O(g)$
- Dans une formule, $O(g)$ représente une fonction anonyme appartenant à $O(g)$
- $O(x^0)$ sera noté $O(1)$

exemple :
Dans $2x^2 + 3x + 1 = 2x^2 + O(x)$, $O(x)$ représente toute fonction f appartenant à $O(x)$ tout en vérifiant l'égalité. Ici, il s'agit de $f(x) = 3x + 1$.

Les propriétés suivantes seront utilisées lors des calculs de complexité :

- Soit $p(x)$ un polynôme de degré d, alors on a : $p(x) = \Theta(x^d)$
- $O(f) + O(g) = O(f + g) = O\big(\max(f + g)\big)$ (distributivité pour l'addition)
- $O(f)O(g) = O(fg)$ (distributivité pour le produit)
- $\forall a, b > 0 : O(\log_a n) = O(\log_b n)$

En pratique, pour obtenir la borne asymptotique supérieure une fois qu'on a exprimé f sous la forme d'un polynôme, il suffit de ne conserver que le terme de degré le plus élevé, et d'ignorer les coefficients constants.
exemple : $2x^2 + 3x + 1 = O(2x^2) + O(3x) + O(1) = O(2x^2) = O(x^2)$

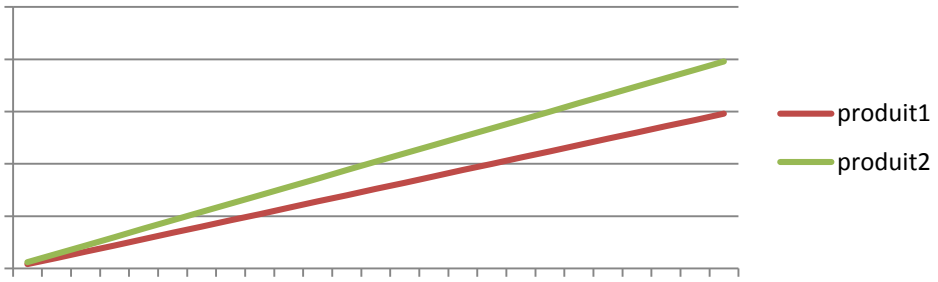
17.1.6 Complexité asymptotique

L'intérêt de la notation asymptotique est double. D'une part, les calculs de complexité sont simplifiés. D'autre part, l'information utile est préservée (on obtient l'ordre de grandeur).

exemple : on calcule les complexités asymptotiques des fonctions `produit1` et `produit2` dans le pire des cas. Toutes les opérations élémentaires ont une complexité asymptotique constante : $O(1)$.

- Complexités temporelles :
  - Première fonction :
    $$T_1(N) = O\big((5N+4)c\big) = O(5Nc) = O(N)$$
  - Seconde fonction :
    $$T_2(N) = O\big((7N+6)c\big) = O(7Nc) = O(N)$$

Si on compare les complexités temporelles dans le pire des cas, les deux fonctions ont la même complexité asymptotique. Pourtant, on sait (d'après nos calculs précédents) que les deux fonctions n'ont pas la même complexité dans le pire des cas : $T_1(N) = 4 + 5N$ et $T_2(N) = 6 + 7N$. Le fait que les deux fonctions aient la même complexité asymptotique signifie que lorsque N tend vers l'infini, la différence entre T_1 et T_2 est négligeable.

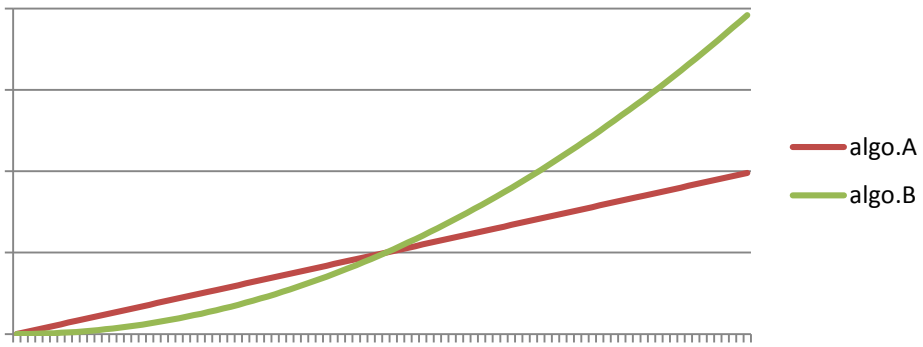


Remarque : quand on compare des algorithmes en termes de complexité asymptotique, il ne faut pas perdre de vue le fait que cette comparaison n'est valable que quand on parle de données de *grande* taille. En effet, sur de petites données, un algorithme en $O(N^2)$ peut être plus rapide qu'un algorithme en $O(N)$.

exemple : considérons les algorithmes A et B suivants :

- L'algorithme A possède une complexité temporelle de $T_A(N) = 100N$.
- L'algorithme B possède une complexité temporelle de $T_B(N) = 2N^2$.

N	Algorithme *A*	Algorithme *B*
10	1000	200
100	10000	20000
1000	100000	2000000
10000	1000000	200000000



Pour des valeurs de N inférieures à 50, on peut observer que B est plus rapide que A. Au-dessus de 50, A est largement plus performant, et la différence ne cesse de s'accroître.

Asymptotiquement, la complexité de A (qui est en $O(N)$) est plus faible que celle de B (qui est en $O(N^2)$).

L'ordre de grandeur de la complexité d'un algorithme est un concept indépendant des contraintes matérielles (machine, langage, implémentation…). Par conséquent, l'utilisation d'une machine plus puissante ne change rien en termes de comparaison asymptotique.

exemple : supposons qu'une opération élémentaire est effectuée en 1 unité de temps. Soit un problème traité par l'algorithme A en 5000 opérations élémentaires, c'est-à-dire en autant d'unités de temps. Ce problème a une taille $N = 50$. Pour l'algorithme B, un problème traitable en 5000 opérations élémentaires correspond également à un problème de taille $N = 50$.

Supposons qu'on dispose d'une machine 10 fois plus puissante : elle effectue 10 opérations élémentaires en 1 seule unité de temps. Alors pour la même durée, l'algorithme A peut traiter un problème de taille $N = 500$, c'est-à-dire 10 fois plus gros qu'avec la machine moins puissante. L'algorithme B peut traiter un problème de taille $N = \sqrt{5000/2} \approx 158$, c'est-à-dire seulement 3 fois plus gros qu'avec la machine moins puissante.

17.2 Calcul de complexité

Dans cette sous-section, nous nous intéressons plus concrètement au calcul de complexité pour des algorithmes itératifs, en considérant séparément les complexités temporelle et spatiale.

17.2.1 Complexité temporelle

Pour réaliser un calcul de complexité temporelle, on associe une certaine complexité à chaque élément composant l'algorithme. Nous nous concentrons ici sur la complexité dans le pire des cas.

- Opération élémentaire (on considère une instruction) : temps constant $O(1)$.
- Appel d'une fonction : soit f la complexité au pire de la fonction ; alors la complexité de l'appel est en $O(f)$.
- Instruction de test (`if` ou `switch`) : soient f la complexité de la condition, et g_i les complexités des différents blocs (`then`, `else`, `case`…) ; alors la complexité au pire est en $O\left(f + \max_i(g_i)\right)$.
- Instruction de répétition (`for`, `do`, `while`) : soient f la complexité du bloc et de la condition, et g la borne supérieure du nombre d'itérations au pire, alors la complexité au pire est en $O(f \cdot g)$
- Séquence d'instructions : soit f_i la complexité de la $i^{ème}$ instruction ; alors la complexité au pire de la séquence est en $O(\sum_i f_i)$.

exemple : recherche du plus petit élément dans un tableau non trié de taille N.

```
   int cherche_minimum (int tab[N])
   {  int i;
1     int indice=0;                    O(1)
2     for(i=1;i<N;i++)                 O(1)
3        if(tab[i] < tab[indice])      O(1)      } O(N)
4           indice = i;                O(1)      }
5     return tab[indice];              O(1)
   }
```

$O(N)$ (englobe les lignes 2 à 4) ; $O(N)$ (englobe les lignes 1 à 5)

- ligne `1` : affectation : $O(1)$.
- dans le `for` :

  - affectation : $O(1)$.
  - comparaison `i<N` : $O(1)$.
  - incrémentation `i++` : $O(1)$.
  - dans le `if` :
    - test : $O(1)$.
    - affectation : $O(1)$.
    - total : le `if` a une complexité de : $O(1) + O(1) = O(1)$.
  - total :
    - le `for` est répété $N - 1$ fois.
    - complexité : $O(1) + N\big(O(1) + O(1) + O(1)\big) = O(N)$.
- retour de valeur : $O(1)$.
- total : $O(1) + O(N) + O(1) = O(N)$.

17.2.2 Complexité spatiale

Comme on l'a vu précédemment, la complexité spatiale correspond à l'occupation mémoire maximale rencontrée au cours de l'application de l'algorithme. Comme pour la complexité temporelle, le calcul se fait en décomposant l'algorithme en différentes parties pour lesquelles on sait calculer la complexité. Là encore, on se concentre sur le pire des cas.

- Déclaration de variable : nombre de positions mémoire occupées par la variable.
- Appel d'une fonction : soit f la complexité au pire de la fonction ; alors la complexité de l'appel est en $O(f)$.
- Instruction de test (`if` ou `switch`) : soient f_i les complexités des différents blocs (`then`, `else`, `case`…) ; alors la complexité au pire est en $O\left(\max_i f_i\right)$.
- Instruction de répétition (`for`, `do`, `while`) : soit f la complexité du bloc ; alors la complexité au pire est en $O(f)$.
- Séquence d'instructions : soit f_i la complexité de la $i^{ème}$ instruction ; alors la complexité au pire de la séquence est en $O\left(\max_i f_i\right)$.

exemple : recherche du plus petit élément dans un tableau non trié de taille N.

- Tableau : $O(N)$.
- Variables simples `i` et `indice` : $O(1) + O(1) = O(1)$.

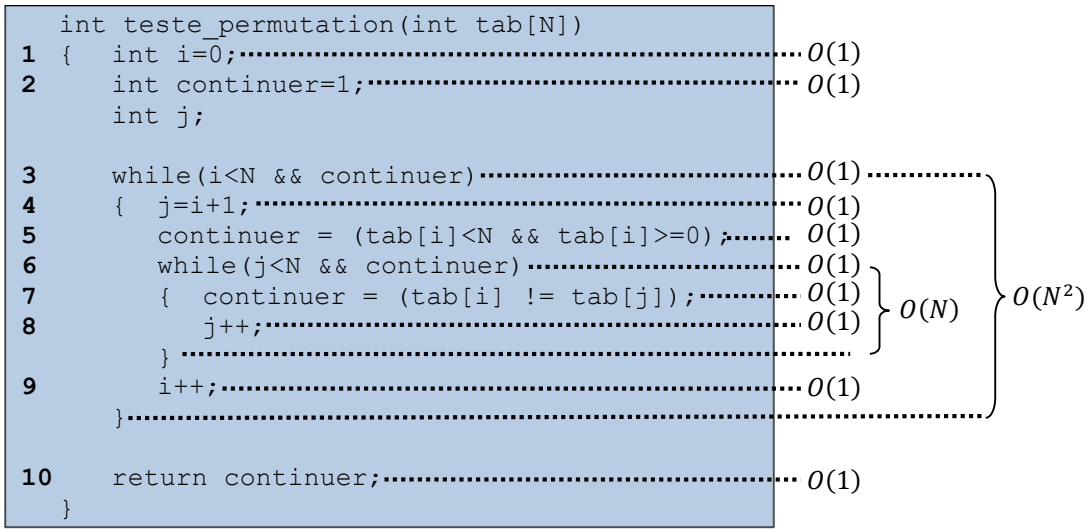
Remarque : si on effectuait une analyse de la complexité spatiale en tenant compte des spécificités du langage C, les tableaux/listes passés en paramètres devraient être considérés comme étant en $O(1)$ et non pas en $O(N)$. La raison est qu'on ne passe pas le tableau/liste lui-même, mais un simple pointeur. Le tableau/liste est créé en amont de la fonction étudiée (dans la fonction appelante, ou bien dans la fonction appelant la fonction appelante…). Mais, dans le cadre de ce cours, nous conviendrons qu'une liste/tableau a une complexité spatiale égale au nombre d'éléments contenus.

17.2.3 Exercices

17.2.3.1 Test de permutation v.1

La fonction `teste_permutation` teste si un tableau d'entiers de taille N correspond bien à une permutation de 0 à $N - 1$. La fonction renvoie 1 si la fonction est une permutation et 0 sinon.

Donnez les complexités spatiale et temporelle de l'algorithme.



Complexité temporelle :

- lignes `1` et `2` : affectations : $O(1)$
- ligne `3` : dans le `while` :
  - test : $O(1)$
  - lignes `4` et `5` : séquence d'opérations élémentaires $O(1)$
  - dans le `while` :
    - test : $O(1)$
    - lignes `7` et `8` : séquence d'opérations élémentaires $O(1)$
    - **total :**
      - nombre de répétitions : quel est le pire des cas ?
        - le tableau est une permutation
        - *j=1* à *N-1*, soit *N-1* répétitions
      - complexité : $(N-1)O(1) = O(N)$
  - ligne `9` : opérations élémentaires : $O(1)$
  - **total :**
    - nombre de répétitions : quel est le pire des cas ?
      - le tableau est une permutation.
      - $i = 0$ à $N-1$, soit N répétitions.
    - complexité : $N \cdot O(1+N) = O(N^2)$
- ligne `10` : opération élémentaire : $O(1)$
- **total :** $O(1) + O(N^2) + O(1) = O(N^2)$

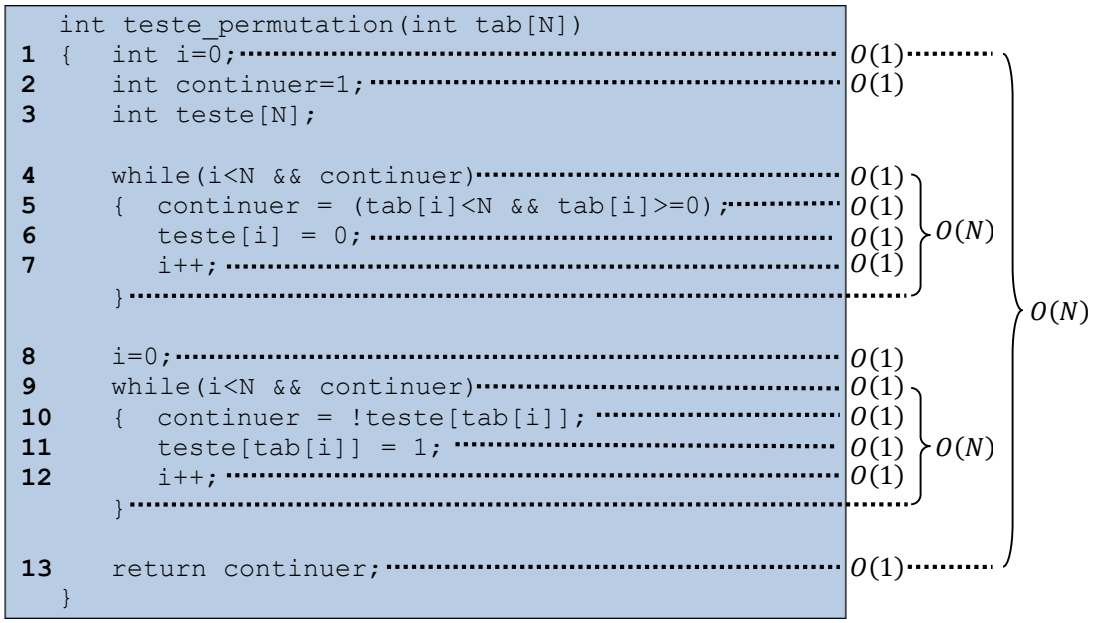
Complexité spatiale :

- on manipule exclusivement des entiers, donc on évaluera la complexité spatiale en termes de nombre de positions mémoire d'entiers.
- *1* tableau `tab` de taille *N* : $O(N)$
- *3* variables `i`, `j` et `continuer` : $3 \times O(1)$
- **total :** $O(N) + 3 \times O(1) = O(N)$

17.2.3.2 Test de permutation v.2

Écrivez une fonction qui réalise la même tâche que la précédente, mais avec une complexité temporelle en $O(N)$.

Calculez la complexité spatiale de cette fonction.



Complexité temporelle :

- lignes `1` et `2` : affectations : $O(1)$
- dans le `while` :
  - test : $O(1)$
  - lignes `5`, `6` et `7` : opérations élémentaires : $O(1)$
  - **total :**
    - nombre de répétitions au pire des cas : N
    - complexité : $N \times O(1) = O(N)$
- ligne `8` : affectation : $O(1)$
- dans le `while` :
  - test : $O(1)$
  - lignes `10`, `11` et `12` : opérations élémentaires : $O(1)$
  - **total :**
    - nombre de répétitions au pire des cas : N
    - complexité : $N \times O(1) = O(N)$
- ligne `13` : opération élémentaire : $O(1)$
- **total :** $O(1) + O(N) + O(N) + O(1) = O(N)$

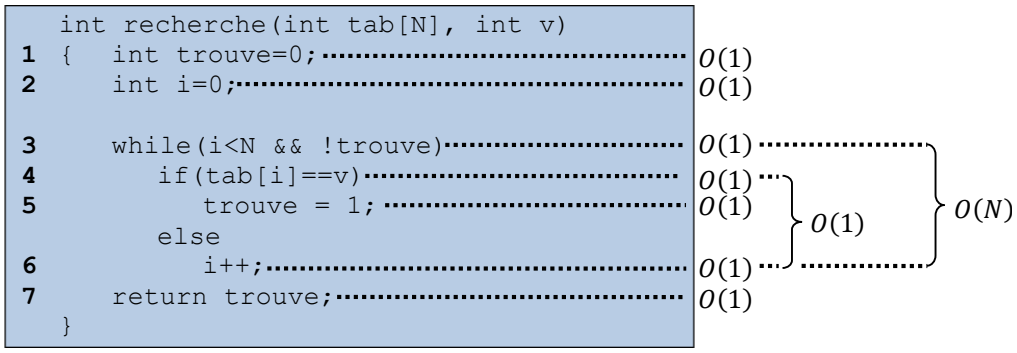
Complexité spatiale :

- 2 tableaux `tab` et `teste` de taille N : $O(N)$
- 2 variables simples `i` et `continuer` : $2 \times O(1)$
- **total :** $O(N) + 2 \times O(1) = O(N)$
- **Remarques :**
  - la complexité spatiale a-t-elle augmenté ? Oui : elle passe de $3 + N$ à $2 + 2N$.
  - Cela a-t-il une importance asymptotiquement parlant ? Non : la complexité asymptotique reste en $O(N)$.

17.2.3.3 Recherche linéaire

Écrivez une fonction `recherche` qui recherche une valeur `v` dans un tableau d'entiers *non-triés* `tab` de taille N. La fonction renvoie 1 si `v` apparaît dans `tab`, ou 0 sinon.

Donnez les complexités spatiale et temporelle de l'algorithme.



Complexité temporelle :

- lignes `1` et `2` : affectations : $O(1)$
- dans le `while` :
  - test : $O(1)$
  - dans le `if` :
    - test : $O(1)$
    - premier bloc :
      - ligne `5` : opération élémentaire : $O(1)$
    - `else` :
      - ligne `6` : opération élémentaire : $O(1)$
    - **total :** $O(1) + \max(O(1), O(1)) = O(1)$
  - **total :**
    - nombre de répétitions au pire des cas : N
    - complexité : $N \times O(1) = O(N)$.
- ligne `7` : opération élémentaire : $O(1)$.
- **total :** $O(1) + O(N) + O(1) = O(N)$.

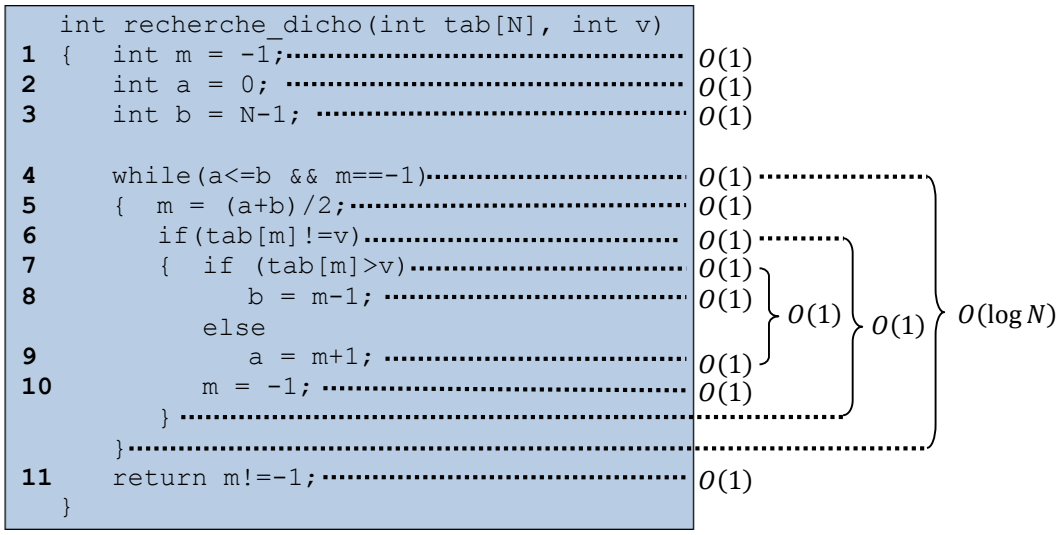
Complexité spatiale :

- 1 tableau `tab` de taille N : $O(N)$
- 3 variables simples `i`, `trouve` et `v` : $3 \times O(1)$
- **total :** $O(N) + 3 \times O(1) = O(N)$

17.2.3.4 Recherche dichotomique

Écrivez la fonction `recherche_dicho` qui recherche une valeur `v` dans un tableau d'entiers *triés* `tab` de taille N, en utilisant le principe de la dichotomie.

Donnez les complexités spatiale et temporelle de l'algorithme.



Complexité temporelle :

- lignes `1`, `2` et `3` : affectations : $O(1)$
- dans le `while` :
  - test : $O(1)$
  - dans le `if` :
    - test : $O(1)$
    - dans le `if` :
      - test : $O(1)$
      - premier bloc :
        - ligne `8` : opération élémentaire : $O(1)$
      - `else` :
        - ligne `9` : opération élémentaire : $O(1)$
      - **total :** $O(1) + \max\big(O(1), O(1)\big) = O(1)$
    - ligne `10` : opération élémentaire : $O(1)$
    - **total :** $O(1) + O(1) + O(1) = +O(1)$
  - **total :**
    - nombre de répétitions au pire des cas :
      - le `while` est répété tant qu'on n'a pas trouvé l'élément et que la recherche est possible.
      - le pire des cas est celui où on ne trouve pas l'élément : il faut poursuivre la recherche jusqu'à ce que ce ne soit plus possible.
      - à chaque itération, on coupe en deux l'intervalle exploré (ligne `5` : `m = (a+b)/2;`)
      - dans le pire des cas, on va donc couper l'intervalle en deux jusqu'à obtenir un intervalle ne pouvant plus être divisé (intervalle ne contenant qu'un seul élément).
      - la question est donc : combien de fois peut-on couper *N* en deux ?
      - supposons qu'on puisse couper *N* en deux *p* fois. On a alors : $N = 2^p + k$, avec $k \in \mathbb{N}$.
      - par définition du logarithme : $p \leq \log_2 N$.
      - On en déduit que l'intervalle de taille *N* peut être coupé au plus $\lfloor \log_2 N \rfloor + 1$ fois.
    - complexité : $(\lfloor \log_2 N \rfloor + 1) \times \big(O(1) + O(1)\big) = O(\log N)$.
- ligne `11` : opération élémentaire : $O(1)$.

- **total :** $O(1) + O(\log N) + O(1) = O(\log N)$.

Complexité spatiale :

- 1 tableau `tab` de taille N : $O(N)$
- 4 variables simples `a`, `b`, `m` et `v` : $4 \times O(1)$

17.3 Algorithmes récursifs

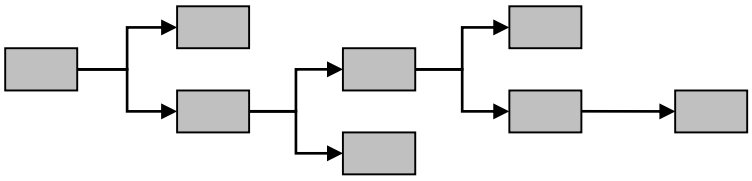
Le calcul de complexité pour des algorithmes récursifs est plus délicat qu'il ne l'était pour les algorithmes itératifs, c'est la raison pour laquelle nous les traitons en dernier. Dans cette sous-section, nous décrivons différentes méthodes permettant de résoudre la relation de récurrence, et un formulaire utilisable en examen est donné pour tirer parti des résultats présentés ici.

17.3.1 Généralités

On a vu en section 13.3 que le traitement récursif peut être décrit par un arbre d'appels, qui représente les relations entre les différents appels récursifs effectués au cours du traitement. Tout en haut de l'arbre, sa racine correspond au tout premier appel effectué. Le temps nécessaire à la résolution du problème traité correspond au temps nécessaire pour exécuter cet appel. Or, il y a deux types de traitements dans cet appel : une partie est locale, et l'autre est récursive, i.e. elle est effectuée au moyen d'appels supplémentaires. Le temps d'exécution de l'appel initial correspond donc à la somme du temps utilisé pour le traitement local, et du temps utilisé pour les appels récursifs. Le même principe s'applique pour chacun de ces appels récursifs. Au final, il vient que le temps total est égal à la somme des temps de traitement *local* pour tous les appels effectués.

En termes de complexité temporelle, cela signifie que la complexité temporelle de l'appel considéré est égale à la somme des complexités temporelles associées aux traitements locaux des appels effectués. Considérons par exemple un appel récursif simple, qui prend la forme d'un arbre linéaire du type suivant :

Alors la complexité temporelle de l'appel initial est calculée à partir des nœuds grisés (c'est-à-dire tous les nœuds). Si la récursivité multiple, on obtient un arbre non-linéaire du type suivant :



Là encore, on doit considérer tous les nœuds présents dans l'arbre, puisque ils doivent tous être traités avant que le calcul ne se termine (un appel initial ne peut pas s'achever avant que tous ses propres appels soient eux-mêmes terminés).

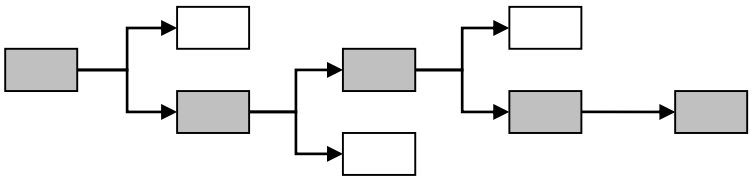
Pour la mémoire, la situation est un peu différente, car son occupation peut croître ou décroître avec l'évolution de l'exécution (alors que le temps de calcul ne peut que croître). On sait que la complexité spatiale correspond à l'occupation maximale atteinte lors de l'exécution de l'algorithme étudié. Pour reprendre l'image de l'arbre d'appel, cela signifie qu'on cherche le nœud tel que la somme des occupations spatiales de tous ses ancêtres est maximale. Autrement dit, on fait la somme des occupations spatiales pour la pire branche de l'arbre.

Pour le cas de la récursivité simple, cela signifie *généralement* qu'on fait la somme des occupations mémoires pour chaque appel.

Attention toutefois, ce n'est pas forcément toujours vrai. Par exemple, si la fonction utilise l'allocation dynamique, il est possible qu'elle *libère* de la mémoire *avant* l'appel ou qu'elle l'*alloue après* l'appel. Dans ce cas-là, on peut avoir une complexité spatiale ponctuellement élevée mais qui ne s'applique pas récursivement.

Par exemple, supposons que la fonction déclare 1 entier statique et un tableau de 1000 entiers dynamique. À ce moment-là, on a donc 1001 entiers en mémoire. Puis, la fonction libère ces 1000 entiers avant de faire l'appel récursif. Le deuxième appel va faire la même chose, ce qui signifie qu'à un moment donné, on occupera en mémoire l'espace nécessaire au stockage de 1002 entiers (et non pas 2002). À moins de répéter l'appel récursif un grand nombre de fois, on restera largement en dessous de la taille du tableau.

Pour la récursivité multiple, on a un arbre non-linéaire de la forme suivante :



Ici, la différence avec la complexité temporelle est encore plus nette. À un moment donné de l'exécution, on ne doit considérer qu'un seul nœud et tous ces ancêtres (i.e. les nœuds permettant de remonter jusqu'à la racine). Comme on s'intéresse à l'occupation maximale, on doit déterminer quelle est la pire des branches, comme ici celle représentée en gris. La complexité spatiale est alors la somme des complexités individuelles de chaque nœud présent sur cette branche.

En raison de l'aspect récursif des algorithmes étudiés, on a ici besoin d'une relation de récurrence pour exprimer ces complexités. Pour rappel, une récurrence est une équation ou une inégalité décrivant une fonction à partir de sa valeur pour des paramètres différents (généralement : plus petits).

Soit $T(n)$ la complexité temporelle de la fonction pour une taille n des données d'entrée. Si on considère un cas d'arrêt, $T(n)$ correspond à la complexité du traitement effectué pour ce cas. Sinon, dans le cas général, $T(n)$ dépend de :

- La complexité $f(n)$ du traitement effectué localement à la fonction ;
- La complexité de l'appel récursif $T\big(h(n)\big)$, avec $h(n) \neq n$.

Remarque : le plus souvent, $h(n)$ est de la forme $h(n) = n - k$ ou $h(n) = n/k$, k étant une constante.

exemple : calcul de factorielle

```
   int factorielle(int n)
   {  int resultat;
1     if(n<=1)    ..... O(1)  } O(1) .....
2        resultat = 1; ..... O(1)  }        .
      else                                  .
3        resultat = n*factorielle(n-1); ... O(1) + T(n − 1) .
4     return resultat; ..... O(1)          .
   }
```

Complexité temporelle :

- Complexité du cas d'arrêt $n = 1$:

    - ligne `1` : condition du `if` : $O(1)$
    - ligne `2` : affectation : $O(1)$
    - ligne `4` : opération élémentaire : $O(1)$
    - total : $T(n) = O(1) + O(1) + O(1) = O(1)$.
- Complexité du cas général $n > 1$:
    - ligne `1` : condition du `if` : $O(1)$
    - ligne `3` :
        - multiplication : $O(1)$
        - appel récursif : $T(n-1)$
    - ligne `4` : opération élémentaire : $O(1)$
    - total : $T(n) = O(1) + O(1) + T(n-1) + O(1) = O(1) + T(n-1)$.
    - remarque : ici $h(n) = n - 1$.

Complexité spatiale :

- complexité du cas d'arrêt $n = 1$:
    - deux variables entières `n` et `résultat`
    - total : $S(n) = O(1)$
- complexité du cas général $n > 1$:
    - deux variables entières `n` et `résultat`
    - les variables utilisées dans l'appel récursif : $S(n-1)$
    - total : $S(n) = O(1) + S(n-1)$

Une fois que la récurrence est établie, il faut la résoudre pour obtenir la complexité de l'algorithme. En d'autres termes, on doit exprimer la fonction $T(n)$ sans utiliser T. Il existe de nombreuses méthodes pour résoudre une récurrence. Dans le cadre de ce cours, nous limiterons aux suivantes :

- Méthode par *substitution* ;
- Méthode par *bornage* ;
- Méthode par *arbre*.

17.3.2 Méthode par substitution

Le principe de cette méthode est le suivant :

1. Dans l'équation définissant la complexité de la fonction, on remplace les $O(n)$ par des polynômes en $O(n)$, en utilisant des constantes arbitraires.
2. Dans la définition de $T(n)$ obtenue, on substitue récursivement les $T\big(h(n)\big)$ par leur définition, jusqu'à voir apparaître une relation entre $T(n)$ et $T\left(h^i(n)\right)$.
3. On prouve cette relation.
4. On en déduit une expression non-récursive de $T(n)$.

exemple : calcul de factorielle

1. On remplace dans $T(n)$:
    - Cas d'arrêt : $T(1) = O(1) = a$
    - Cas général : $T(n) = O(1) + T(n-1) = b + T(n-1)$
2. On tente de trouver une relation à démontrer :

$$T(n) = b + T(n-1) \quad (1)$$
$$T(n-1) = b + T(n-2) \quad (2)$$

On remplace (2) dans (1) :

$$T(n) = b + \big(\boldsymbol{b + T(n-2)}\big) = 2b + T(n-2) \quad (3)$$

On recommence :

$$T(n) = 2b + \left(\boldsymbol{b} + \boldsymbol{T(n-3)}\right) = 3b + T(n-3) \quad (4)$$
$$T(n) = 3b + \left(\boldsymbol{b} + \boldsymbol{T(n-4)}\right) = 4b + T(n-4) \quad (5)$$

etc.

On peut faire l'hypothèse qu'on s'intéresse à la propriété P_i telle que : $T(n) = ib + T(n-i)$ pour $1 < i < n$.

3. Prouvons P_i :
   - Cas de base : $i = 1$
     Le cas de base correspond à notre définition de la complexité :
     $$T(n) = 1b + T(n-1) \quad (P_1)$$
   - Cas de récurrence : $2 < i < n$
     On suppose que P_i est vraie, et on veut prouver P_{i+1} :
     $$T(n) = ib + T(n-i) \quad (P_i)$$
     Or on a par définition de T :
     $$T(n-i) = b + T(n-i-1)$$
     On remplace $T(n-i)$ dans P_i :
     $$T(n) = ib + \left(b + T(n-i-1)\right)$$
     $$T(n) = (i+1)b + T\left(n-(i+1)\right) \quad (P_{i+1})$$
   - On a prouvé P.
4. On peut donc maintenant se ramener à une expression de $T(n)$ en posant $i = n-1$:
   $$T(n) = (n-1)b + T\left(n-(n-1)\right)$$
   $$T(n) = (n-1)b + a$$
   - Au final, la complexité temporelle est donc $T(n) = O(n) + O(1) = O(n)$

Remarque : la complexité spatiale est identique, puisqu'elle repose sur la même définition récursive.

17.3.3 Méthode par bornage

La notation asymptotique peut être utilisée pour faciliter la démonstration : en effet, il est inutile de calculer la relation de récurrence *exacte*, puisqu'au final on ne veut qu'une borne supérieure.

Le principe de cette méthode est d'abord de déterminer intuitivement une hypothèse quant à la forme de la borne asymptotique supérieure (ou inférieure) de $T(n)$ (ou de $S(n)$), puis de prouver cette hypothèse par récurrence.

1. Soit une fonction g telle que $T = O(g)$. On suppose donc qu'il existe une constante k telle qu'à partir d'un certain $n \geq n_0$, on ait : $T(n) \leq k \times g(n)$.
2. Cas de base : on montre la propriété pour un n_0.
3. Cas de récurrence : on substitue l'expression de $T(n)$ dans $T\big(h(n)\big)$, et on tente de montrer que $T\big(h(n)\big) \leq k \times g\big(h(n)\big)$.

Remarque : On tente d'abord la démonstration avec une fonction g ne contenant qu'un seul terme, du degré le plus élevé. Si la démonstration bloque, on peut rajouter des termes de degré inférieur qui permettront éventuellement de simplifier une inégalité et d'aboutir.

exemple : calcul de factorielle

On reprend les constantes a et b posées précédemment :

$$T(1) = a \quad (1)$$
$$T(n) = T(n-1) + b \quad (2)$$

1. Supposons que T est en $O(n)$, avec $kg(n) = cn$, i.e. $T(n) \leq cn$ pour un certain $n \geq n_0$.
2. Cas de base : $n = 1$
   D'après (1), on a : $kg(1) = c$.
   On pose : $c \geq a$ de manière à avoir $T(1) \leq kg(1)$.
3. Cas de récurrence : $n > 1$
   Soit la propriété P_n :
   $$T(n) \leq cn \qquad (P_n)$$
   On suppose que P_n est vraie, et on veut montrer P_{n+1} :
   $$T(n+1) \leq c(n+1) \qquad (P_{n+1})$$
   On réécrit (2) sous la forme :
   $$T(n+1) = T(n) + b \qquad (3)$$
   On substitue P_n dans (3) :
   $$T(n+1) \leq cn + b \qquad (4)$$
   En posant $b = c$, l'équation (4) correspond bien à l'inégalité à prouver (P_{n+1}).
   On a donc prouvé que pour $a \leq b = c$ et $n \geq 1$, on a bien la propriété P_n.
   Donc $T(n)$ est bien en $O(n)$.

Remarque : avec cette méthode, on peut prouver que la complexité d'une fonction est en $O(g)$, mais on ne prouve pas qu'il n'existe pas de *meilleure* borne supérieure (i.e. une borne d'ordre inférieur). Cependant, on peut l'utiliser pour trouver une borne supérieure et une borne inférieure de même forme (on a donc une borne approchée).

exemple : recherche dichotomique récursive

```
   int recherche_dicho(int tab[], int debut, int fin, int v)
   {  int resultat,m;

1     if(debut<=fin) ····· O(1)
2     {  m = (debut+fin)/2; ····· O(1)
3        if(tab[m]==v) ····· O(1)
4           resultat = 1; ····· O(1)
         else
5        {  if(tab[m]>v) ····· O(1)
6              resultat = recherche_dicho(tab, debut, m-1, v);  T(N/2) + O(1)
            else
7              resultat = recherche_dicho(tab, m+1, fin, v); ··· T(N/2) + O(1)
         }
      }
      else
8        resultat = 0; ····· O(1)
9     return resultat; ····· O(1)
   }
```

Taille des données :

- partie du tableau `tab` comprise entre `debut` et `fin` : $n = fin - debut + 1$

Complexité temporelle :

- complexité du cas d'arrêt $n = 0$:
  - ligne `1` : condition du `if` : $O(1)$
  - ligne `8` : affectation : $O(1)$
  - ligne `9` : opération élémentaire : $O(1)$
  - total : $T(0) = O(1) + O(1) + O(1) = O(1)$.
- complexité du cas général $n > 0$:
  - ligne `1` : condition du `if` : $O(1)$

  - ligne `2` : opérations élémentaires : $O(1)$
  - ligne `3` : dans le `if` :
    - test : $O(1)$
    - premier bloc :
      - ligne `4` : opération élémentaire : $O(1)$
    - `else` :
      - ligne `5` : dans le `if` :
        - test : $O(1)$
        - premier bloc : $T(N/2) + O(1)$
        - `else` : $T(N/2) + O(1)$
        - total du `if` : $T(N/2) + O(1)$
      - total du `else` : $T(N/2) + O(1)$
    - total du `if` : $T(N/2) + O(1)$
  - ligne `9` : opération élémentaire : $O(1)$
  - total : $T(N) = T(N/2) + O(1)$
- résolution de la récurrence :
  - on pose :
    $$T(0) = a \quad (1)$$
    $$T(n) = T(n/2) + b \quad (2)$$
  - supposons que T est en $O(\log(n))$, avec $kg(n) = c\log(n) + d$.
  - i.e. : $T(n) \leq c\log(n) + d$.
- cas de base :
  - il y a un problème si on prend $n = 0$ car $\log(0)$ n'est pas défini.
  - mais le cas de base de la démonstration ne doit pas forcément être celui de la relation de récurrence, puisque dans le cadre de la notation asymptotique, on considère un n_0 suffisamment grand.
  - on peut donc prendre $n = 1$, et on a alors : $kg(1) = d$
  - on pose $a \leq d$ de manière à avoir $T(1) \leq kg(1)$.
- cas de récurrence : $n > 1$
  - Soit la propriété P_n telle que :
    $$T(n) \leq c\log(n) + d \quad (P_n)$$
  - on suppose que P_n est vraie, et on veut montrer P_{2n} :
    $$T(2n) \leq c(\log(2n)) + d \quad (P_{2n})$$
  - on récrit (2) sous la forme :
    $$T(2n) = T(n) + b \quad (3)$$
  - on substitue P_n dans (3) :
    $$T(2n) \leq (c\log(n) + d) + b \quad (4)$$
  - en posant $b = c$, on obtient :
    $$T(2n) \leq c\log(n) + d + c \quad (5)$$
    $$T(2n) \leq c(1 + \log(n)) + d \quad (6)$$
  - par définition de la fonction logarithme, on a $\log_2(2^1) = 1$, d'où :
    $$T(2n) \leq c(\log(2) + \log(n)) + d \quad (7)$$
  - par définition de la fonction logarithme, on a $\log(a) + \log(b) = \log(ab)$, d'où :
    $$T(2n) \leq c\log(2n) + d \quad (8)$$
  - (8) correspond bien à l'inégalité à prouver (P_{2n}).
- On a donc prouvé que pour $a \leq b = c$ et $n \geq 1$, on a bien la propriété P_n.
- Donc $T(n)$ est bien en $O(\log n)$.

17.3.4 Méthode par arbre

Il n'est pas forcément évident de deviner une forme de la borne supérieure, notamment dans le cas d'une récursivité multiple ou croisée. La méthode par arbre permet d'avoir une idée de la forme de la borne, ce qui permet d'en faire ensuite la preuve formelle en utilisant une méthode classique.

exemple : suite de Fibonacci

```
   int fibo(int n)
1  {   int resultat=1;          ..... O(1)
2      if(n>1)                  ..... O(1)
3         resultat = fibo(n-1)+fibo(n-2);  ..... O(1) + T(n-1) + T(n-2)
4      return resultat;         ..... O(1)
   }
```

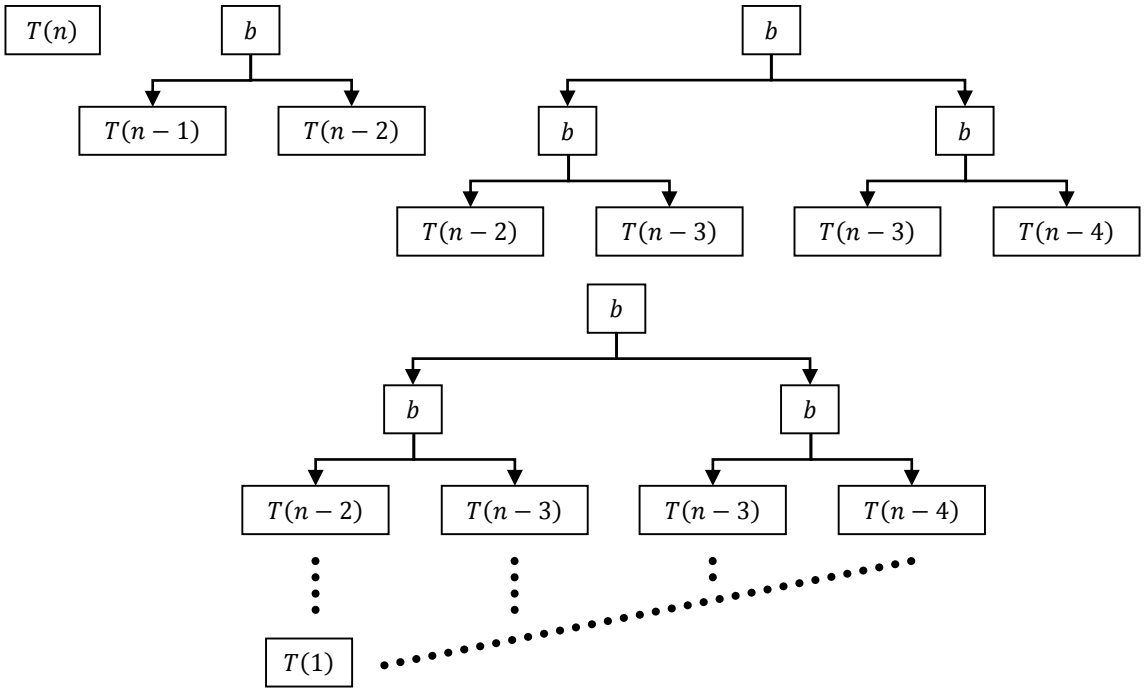
Complexité temporelle :

- Complexité du cas d'arrêt $n \leq 1$: $T(n) = a = O(1)$
- Complexité du cas général $n > 1$: $T(n) = T(n-1) + T(n-2) + b$

Complexité spatiale :

- Cas d'arrêt $n \leq 1$: $S(n) = a = O(1)$
- Cas général : $S(n) = \max\big(S(n-1), S(n-2)\big) + b$

Arbre d'appels :



- Hauteur de l'arbre : $p = n - 1$
- Borne supérieure du nombre de nœuds à la hauteur p : 2^p
- Borne supérieure du nombre d'appels total : $\sum_{p=0}^{n-1} 2^p = 2^n - 1$
- Hypothèses :
  - Pour la complexité temporelle : $T(n) = O(2^n - 1) = O(2^n)$.
  - Pour la complexité spatiale : $S(n) = O(n-1) = O(n)$.

Complexité temporelle :

- Résolution de la récurrence :

- on pose :

$$T(0) = T(1) = a \quad (1)$$
$$T(n) = T(n-1) + T(n-2) + b \quad (2)$$

- supposons que T est en $O(2^n)$, avec $kg(n) = c2^n + d$.
- i.e. : $T(n) \leq c2^n + d$.

- Cas de base : $n = 0$:
  - on a : $kg(0) = c + d$.
  - on pose : $a \leq c + d$.
- Cas de base : $n = 1$:
  - on a : $kg(1) = 2c + d$.
  - on pose : $a \leq 2c + d$.
- Cas de récurrence : $n > 2$:
  - soit les propriétés P_{n-1} et P_n telles que :

$$T(n-1) \leq c2^{n-1} + d \quad (P_{n-1})$$
$$T(n) \leq c2^n + d \quad (P_n)$$

  - on suppose que P_n et P_{n-1} sont vraies, et on veut prouver P_{n+1} :

$$T(n+1) \leq c2^{n+1} + d \quad (P_{n+1})$$

  - on réécrit (2) sous la forme :

$$T(n+1) = T(n) + T(n-1) + b \quad (3)$$

  - on substitue P_{n-1} et P_n dans (3) :

$$T(n+1) \leq (c2^n + d) + (c2^{n-1} + d) + b \quad (4)$$

  - on a $n > 2$, donc $2^n > 2^{n-1}$, d'où :

$$T(n+1) < c2^n + c2^n + 2d + b \quad (5)$$
$$T(n+1) < 2c2^n + 2d + b \quad (6)$$
$$T(n+1) < c2^{n+1} + 2d + b \quad (7)$$

  - on pose $b = -d$, de façon à ce que (7) corresponde à l'inégalité à prouver P_{n+1}.
- On a donc prouvé que pour $a \leq c + d$, $a \leq 2c + d$, $b = -d$ et $n \geq 1$, la propriété P_n est vérifiée.
- Donc $T(n)$ est bien en $O(2^n)$.

Complexité spatiale :

- On peut supposer que $S(n)$ est croissante (i.e. la mémoire occupée augmente avec la taille des données).
- Donc on suppose que $S(n-1) > S(n-2)$, d'où pour le cas général : $S(n) = S(n-1) + b$.
- On a déjà rencontré des récurrences de cette forme, et on sait que $S(n) = O(n)$.

17.3.5 Formulaire

Les mêmes méthodes pourraient s'appliquer à toutes les relations de récurrence. Cependant, les traiter en détail sort du cadre de ce cours. Comme nous allons quand même avoir besoin de calculer les complexités de certains algorithmes, on se propose plutôt d'utiliser le tableau suivant, qui associe une complexité asymptotique à différentes formes de récurrence.

Dans ce tableau, on suppose que le cas de base est $T(1) = O(1)$, et que $k \geq 0$.

Relation de récurrence	Complexité
$T(n) = T(n-1) + \Theta(n^k)$	$O(n^{k+1})$
$T(n) = cT(n-1) + \Theta(n^k)$ pour $c > 1$	$O(c^n)$

$T(n) = cT(n/d) + \Theta(n^k)$ pour $c > d^k$	$O\left(n^{\log_d c}\right)$
$T(n) = cT(n/d) + \Theta(n^k)$ pour $c < d^k$	$O(n^k)$
$T(n) = cT(n/d) + \Theta(n^k)$ pour $c = d^k$	$O(n^k \log n)$

exemples : utilisation du tableau sur différentes fonctions connues

- **Factorielle :**
  - $T(n) = O(1) + T(n-1)$.
  - 1ère ligne du tableau avec $k = 0$.
  - On a donc : $T(n) = O(n^{0+1}) = O(n)$.
- **Recherche dichotomique :**
  - $T(N) = T(N/2) + O(1)$.
  - Dernière ligne du tableau avec $d = 2$ et $k = 0$ d'où $d^k = 1$ et $c = d^k$.
  - On a donc : $T(n) = \mathrm{O}(n^0 \log n) = \mathrm{O}(\log n)$.
- **Suite de Fibonacci :**
  - $T(n) = T(n-1) + T(n-2) + O(1)$.
  - On considère que *T* est croissante, d'où : $T(n) \leq 2T(n-1) + O(1)$.
  - 2ème ligne du tableau avec $k = 0$ et $c = 2 > 1$.
  - On a donc : $T(n) = O(2^n)$.

18 Algorithmes de tri

Les algorithmes de tri constituent une classe étudiée depuis longtemps en algorithmique. Ils sont intéressants car, d'un point de vue pratique, la nécessité de trier des données est présente dans la plupart des domaines de l'informatique. De plus, d'un point de vue pédagogique, ils permettent d'illustrer de façon très parlante la notion de complexité algorithmique abordée en section 17.

Nous définissons d'abord la notion d'algorithme de tri de façon générale, ainsi que les différents aspects utilisés lorsqu'on veut les caractériser et les comparer. Dans un deuxième temps, nous présentons en détails les algorithmes de tri les plus connus.

18.1 Présentation

18.1.1 Définitions

On définit un algorithme de tri de la façon suivante :

Algorithme de tri : algorithme dont le but est d'organiser une collection d'éléments muni d'une relation d'ordre.

Notez bien qu'il s'agit d'une *collection*, et non pas d'un *ensemble*, ce qui signifie qu'il est possible que le groupe d'éléments que l'on veut trier contienne plusieurs fois la *même valeur*. Il n'est pas inutile de rappeler qu'une relation d'ordre est une relation *binaire* de la forme suivante :

Relation d'ordre : relation binaire, réflexive, transitive et antisymétrique .

Pour mémoire, le fait qu'une relation $\diamond$ est *binaire* signifie qu'elle relie deux ensembles d'objets X et Y. Elle est définie sur un sous-ensemble de $X \times Y$: pour tout couple $x \in X$ et $y \in Y$ de ce sous-ensemble, on dit que les deux objets sont *en relation*, et ceci est noté $x \diamond y$. Dans notre cas, les deux ensembles sont confondus, i.e. $X = Y$. La relation est dite *réflexive* si un objet est en relation avec lui-même, i.e. on a $\forall x \in X : x \diamond x$. La relation est *transitive* si, quand un 1er objet est en relation avec un 2ème, et que ce 2ème est lui-même en relation avec un 3ème, alors le 1er et le 3ème sont aussi en relation. Autrement dit : $\forall x, y, z \in X : [(x \diamond y) \wedge (y \diamond z)] \Longrightarrow x \diamond z$. Enfin, la relation est dite *antisymétrique* si quand deux objets sont en relation mutuelle, alors ils sont confondus, i.e. : $\forall x, y \in X : [(x \diamond y) \wedge (y \diamond x)] \Longrightarrow x = y$. Enfin, on dit que la relation est *totale* si elle met en relation toute paire d'objets de X, i.e. $\forall x, y \in X : (x \diamond y) \vee (y \diamond z)$. Le fait qu'une relation est totale signifie qu'il est possible de comparer n'importe quels éléments de l'ensemble considéré.

Par exemple, la relation $\leq$ définie sur les entiers est une relation d'ordre, car on a :

- Réflexivité : $\forall x \in \mathbb{N} : x \leq x$
- Transitivité : $\forall x, y, z \in \mathbb{N} : [(x \leq y) \wedge (y \leq z)] \Longrightarrow x \leq z$
- Antisymétrie : $\forall x, y \in \mathbb{N} : [(x \leq y) \wedge (y \leq x)] \Longrightarrow x = y$
- De plus, la relation est totale puisque $\forall x, y \in \mathbb{N} : (x \leq y) \vee (y \leq z)$

Mais on peut définir d'autres relations d'ordres, par exemple sur d'autres types de données. Considérons ainsi les chaînes de caractères : l'*ordre lexicographique*, qui est utilisé notamment dans les dictionnaires pour classer les mots, est lui aussi conforme à la définition d'une relation d'ordre.

Il est important de noter que le principe utilisé pour le tri est complètement indépendant de la nature de la relation d'ordre utilisée. On peut utiliser le même algorithme pour trier des entiers ou des chaînes de caractères. Tout ce qui va changer, c'est que dans un cas on utiliser l'opérateur C <=, alors que dans l'autre il faudra créer une fonction capable de comparer des chaînes de caractères.

Dans le cas le plus simple du tri, on va utiliser la relation d'ordre pour organiser les objets contenus dans l'ensemble *X* sur laquelle elle est définie. Mais il est aussi possible que les éléments que l'on veut trier soient de type complexe, en particulier de type structuré. La valeur de *X* associée à l'élément est alors appelée une *clé* :

Clé de tri : champ de l'élément à trier qui est utilisée pour effectuer la comparaison.

Autrement dit, la clé est la partie de l'élément sur laquelle la relation d'ordre s'applique. C'est cette clé qui permet de comparer deux éléments, et de décider lequel doit être placé avant l'autre dans la séquence triée.

exemples :

- éléments *simples* :
  - les éléments sont des entiers
  - la clé correspond à l'élément tout entier
- éléments *complexes* :
  - les éléments sont des étudiants caractérisés par un nom, un prénom et un âge.
  - si on effectue un tri par rapport à leur âge, alors l'âge est la clé et les autres données ne sont pas considérées lors du tri.

On peut également définir des clés *multiples*, c'est-à-dire utiliser une première clé pour ordonner deux éléments, puis une autre clé en cas d'égalité, et éventuellement encore d'autres clés supplémentaires.

exemple : dans l'exemple des étudiants, on pourrait ainsi décider de les trier en fonction de leur âge d'abord, puis de leur nom quand deux étudiants ont le même âge.

18.1.2 Caractéristiques

On classe les différents algorithmes de tris suivant plusieurs propriétés : *complexité* algorithmique, caractère *en place*, caractère *stable*. Pour la complexité, on procède comme on l'a fait précédemment en section 17 : meilleur/moyen/pire des cas, spatial/temporel. Les deux autres aspects sont spécifiques au problème de tri.

Caractère en place : se dit d'un tri qui trie la collection en la modifiant directement, i.e. sans passer par une structure de données secondaire.

En particulier, certains tris utilisent une copie de la collection, ce qui implique une plus grande consommation de temps (puisqu'il faut copier la collection) et de mémoire (puisqu'il faut stocker la copie). Le fait de travailler directement sur la structure originale est donc plus efficace de ce point de vue-là.

Caractère stable : se dit d'un tri qui ne modifie pas la position relative de deux éléments de même clé.

exemple : supposons qu'on a deux étudiants *A* et *B* de même âge, et que l'étudiant *A* est placé avant l'étudiant *B* dans la collection initiale :

Pour des raisons pédagogiques, les différents algorithmes de tri abordés dans ce cours seront appliqués à des tableaux d'entiers de taille `N`, mais ils peuvent bien sûr être appliqués à :

- N'importe quel type de données comparable autre que les entiers (réels, structures…) ;
- N'importe quelle structure séquentielle autre que les tableaux (listes…).

De plus, on considèrera dans ce cours que l'objectif est de trier les tableaux dans l'ordre *croissant* : le tri dans l'ordre *décroissant* peut être facilement obtenu en inversant les comparaisons effectuées dans les algorithmes.

La notion de tri sera approfondie en TP, puisque certains donneront l'occasion d'adapter aux listes des algorithmes vus en cours pour les tableaux ; d'implémenter d'autres algorithmes pas étudiés en cours, et enfin de trier des objets complexes (par opposition à de simples entiers).

18.2 Tri à bulles

18.2.1 Principe

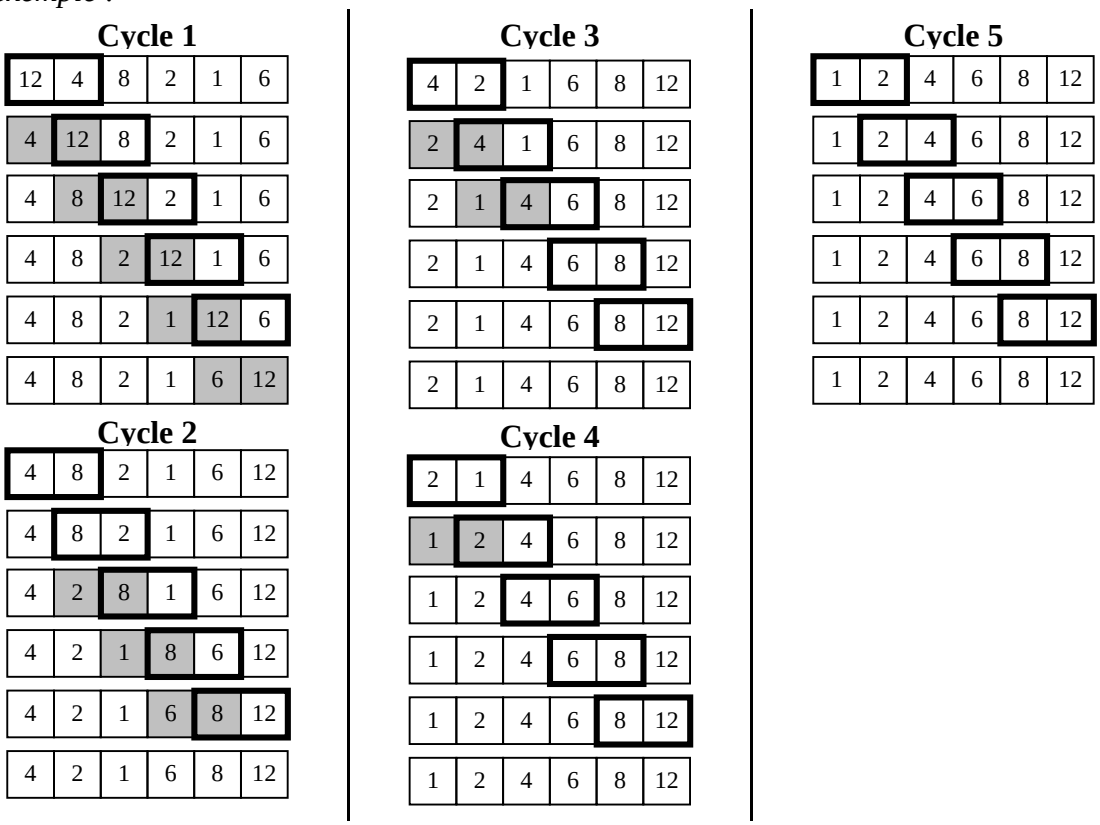
Le principe du tri à bulle[25] est de reproduire le déplacement d'une bulle d'air dans un liquide : elle remonte progressivement à la surface.

On appelle *cycle* la séquence d'actions suivante :

1. On part du *premier* élément du tableau.
2. On compare les éléments consécutifs deux à deux :
   - Si le premier est supérieur au deuxième : ils sont échangés ;
   - Sinon : on ne fait rien.
3. On passe aux deux éléments consécutifs suivants, jusqu'à arriver à la fin du tableau.

On répète ce cycle jusqu'à ce que plus aucun échange ne soit effectué.

exemple :



Dans la figure ci-dessus :

- Les valeurs encadrées sont les deux éléments consécutifs comparés.
- Les valeurs en gris sont celles que l'on vient d'échanger.

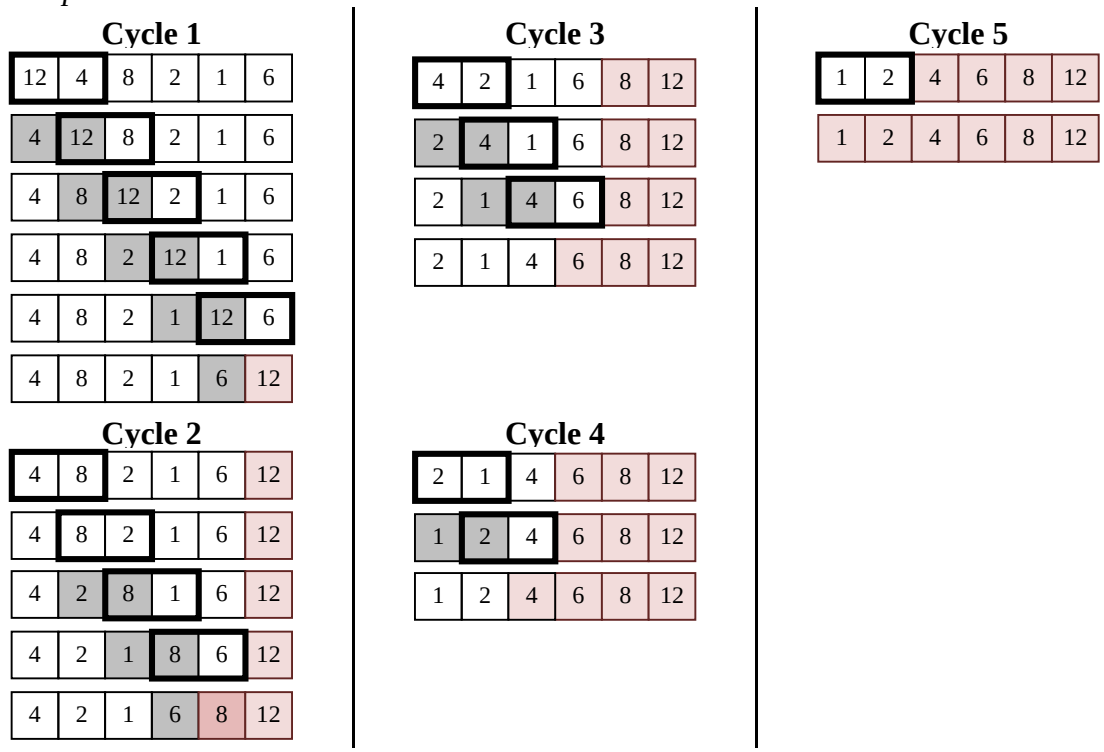
Il s'agit d'une version basique du tri à bulle, qui peut être améliorée, car certaines des opérations qu'elle effectue sont inutiles. En effet, on peut remarquer qu'à chaque cycle, le tri à bulle fait remonter une valeur jusqu'à ce qu'elle atteigne sa place définitive. Par exemple : à la

[25] En anglais : Bubble Sort

fin du 1^{er} cycle, la plus grande valeur est dans la $1^{ère}$ case en partant de la droite du tableau. À la fin du $2^{ème}$ cycle, la $2^{ème}$ plus grande valeur est dans la $2^{ème}$ case en partant de la droite du tableau, etc. Au $i^{ème}$ cycle, il est donc inutile d'effectuer les comparaisons après la case $N - i$, puisque ces valeurs sont forcément supérieures à toutes celles qui sont placées avant la case $N - i$.

Si on reprend l'exemple précédent, en représentant en rouge les cases ont on sait que la valeur ne changera plus jamais, et en supprimant les opérations inutiles :

exemple :



Le nombre de cycles ne change pas, mais leur durée diminue.

18.2.2 Implémentation

Soit `tri_bulle(int tab[])` la fonction qui trie le tableau d'entiers `tab` de taille N en utilisant le principe du tri à bulles.

```
   void tri_bulles(int tab[N])
1  {   int permut,i,temp,j=0;                          O(1)
2      do                                               O(N²)
3      {  permut = 0;                                   O(1)
4         for (i=1;i<N-j;i++)                           O(N)
5         {  if (tab[i-1]>tab[i])                       O(1)
6            {  temp = tab[i-1];                        O(1)
7               tab[i-1]=tab[i];                        O(1)
8               tab[i]=temp;                            O(1)
9               permut=1;                               O(1)
             }
          }
10        j++;                                          O(1)
       }
11     while(permut);                                   O(1)
   }
```

Complexité temporelle :

- ligne `1` : affectation : $O(1)$.

- ligne `2` : `do` :
  - ligne `3` : affectation : $O(1)$.
  - ligne `4` : `for` :
    - ligne `4` : opérations élémentaires : $O(1)$.
    - ligne `5` : `if` : opérations élémentaires : $O(1)$.
    - répétitions : N fois dans le pire des cas.
    - **total :** $N \times O(1) = O(N)$.
  - lignes `10` et `11` : opérations élémentaires $O(1)$.
  - répétitions : N fois dans le pire des cas.
  - **total :** $N \times O(N) = O(N^2)$.
- **total :** $T(N) = O(1) + O(N^2) = O(N^2)$.
- **remarque** :
  - complexité en moyenne : $O(N^2)$.

Complexité spatiale :

- 4 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- **total :** $S(N) = O(N)$.

Propriétés :

- En place : oui (on trie en modifiant directement le tableau).
- Stable : oui (on ne permute deux éléments qu'après une comparaison stricte).

18.3 Tri par sélection

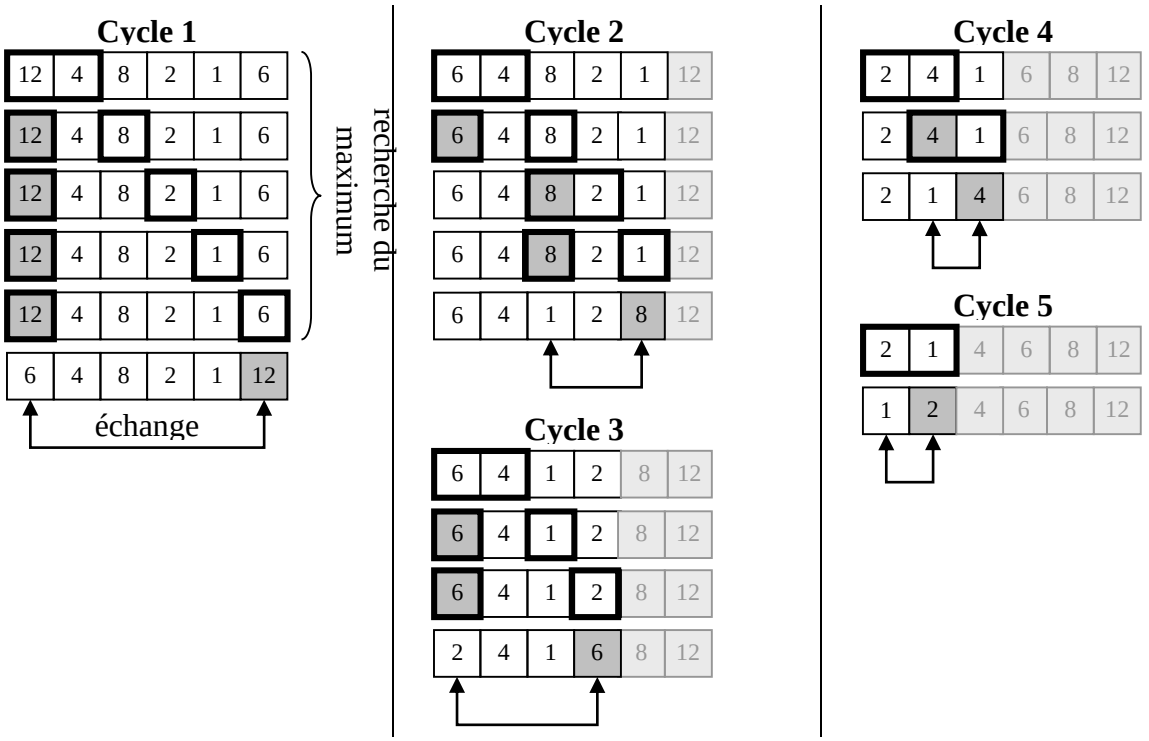
18.3.1 Principe

Le principe du tri par sélection[26] est de déterminer quel est le maximum du tableau, de le déplacer à l'endroit qu'il devrait occuper si le tableau était trié, puis de continuer avec les valeurs restantes.

1. On cherche la plus grande valeur du tableau.
2. On déplace cette valeur dans le tableau afin de la placer au bon endroit, c'est-à-dire à la dernière place (case $N-1$).
3. On cherche la 2ème plus grande valeur du tableau : on cherche donc la plus grande valeur du sous-tableau allant de la case 0 à la case $N-2$ incluse.
4. On déplace cette valeur dans le tableau afin de la placer au bon endroit, c'est-à-dire à l'avant-dernière place (case $N-2$), soit la 2ème case en partant de la fin du tableau.
5. On cherche la 3ème plus grande valeur du tableau : on cherche donc la plus grande valeur du sous-tableau allant de la case 0 à la case $N-3$ incluse.
6. On déplace cette valeur dans le tableau afin de la placer au bon endroit, c'est-à-dire à l'antépénultième place (case $N-3$), soit la 3ème case en partant de la fin du tableau.
7. On répète ce traitement jusqu'à ce que le sous-tableau à traiter ne contienne plus qu'une seule case (la case 0), qui contiendra obligatoirement le minimum de tout le tableau.

exemple :

[26] En anglais : Selection Sort.



Dans la figure ci-dessus :

- Les valeurs encadrées sont les éléments comparés.
- La valeur en gris est le maximum du sous-tableau.
- Les valeurs claires représentent la partie du tableau qui a été triée.

18.3.2 Implémentation

Soit `tri_selection(int tab[])` la fonction qui trie le tableau d'entiers `tab` de taille N en utilisant le principe du tri par sélection.

```
   void tri_selection(int tab[N])
   {   int m,i,j,temp;

1      for(i=N-1;i>0;i--)
2      {  m=0;                                   O(1)
3         for (j=0;j<=i;j++)
4         {  if(tab[j]>tab[m])                   O(1)   } O(1) } O(N)
5               m=j;                             O(1)   }
          }
6         temp=tab[i];                           O(1)
7         tab[i]=tab[m];                         O(1)
8         tab[m]=temp;                           O(1)
       }                                                          } O(N²)
   }
```

Complexité temporelle :

- ligne `1` : `for` :
  - lignes `1` et `2` : opérations élémentaires : $O(1)$.
  - ligne `3` : `for` :
    - ligne `3` : opérations élémentaires : $O(1)$.
    - ligne `4` : `if` : opérations élémentaires : $O(1)$.
    - répétitions : N fois au pire.
    - **total :** $N \times O(1) = O(N)$.
  - lignes `6`, `7` et `8` : opérations élémentaires : $O(1)$.

- répétitions : N fois au pire.
- **total :** $N \times O(N) = O(N^2)$.

- **total :** $T(N) = O(N^2)$.
- **remarques** :
  - complexité en **moyenne** : $O(N^2)$.

Complexité spatiale :

- 4 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- **total :** $S(N) = O(N)$.

Propriétés :

- En place : oui (on trie en modifiant directement le tableau).
- Stable : non (quand le maximum est placé à la fin, on modifie forcément sa position relativement à d'autres éléments dont la clé possède la même valeur).

18.4 Tri par insertion

18.4.1 Principe

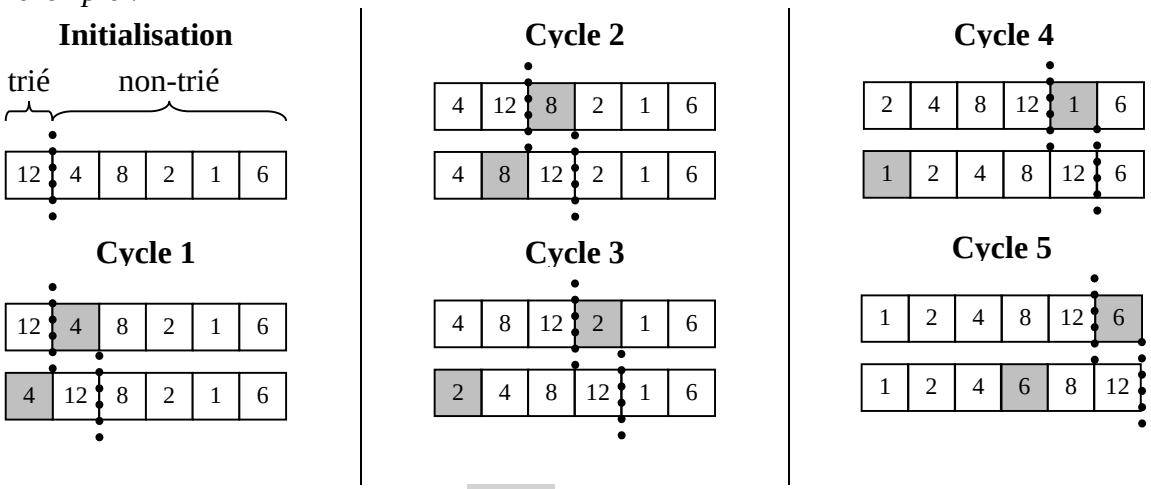
Le tri par insertion[27] effectue un découpage du tableau en deux parties :

- Une partie *pas encore* triée ;
- Une partie *déjà* triée.

Initialement, la partie triée est constituée d'un seul élément : le premier élément du tableau. Le principe de l'algorithme consiste alors à répéter le traitement suivant :

1. Sélectionner le premier élément de la partie non-triée ;
2. L'insérer à la bonne place dans la partie triée ;
3. Jusqu'à ce qu'il n'y ait plus aucun élément dans la partie non-triée.

exemple :

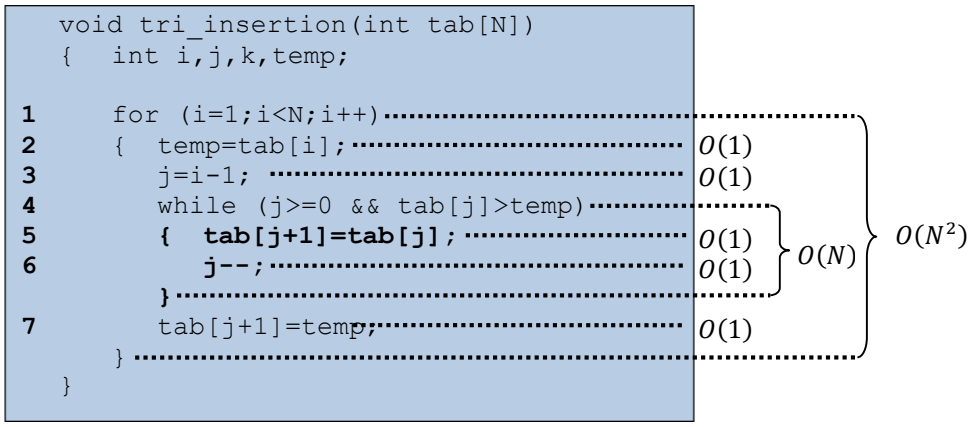


Dans la figure ci-dessus, la valeur en gris est l'élément de la partie non-triée qu'on insère dans la partie triée.

18.4.2 Implémentation

Soit `void tri_insertion(int tab[])` la fonction qui trie le tableau d'entiers `tab` de taille N en utilisant le principe du tri par insertion.

[27] En anglais : Insertion Sort.



Complexité temporelle :

- ligne `1` : `for` :
  - lignes `2` et `3` : opérations élémentaires : $O(1)$.
  - ligne `4` : `while` :
    - lignes `5` et `6` : opérations élémentaires : $O(1)$.
    - répétitions : $N-1$ au pire.
    - **total :** $N \times O(1) = O(N)$.
  - ligne `7` : affectation : $O(1)$.
  - répétitions : $N-1$ au pire.
  - **total :** $(N-1) \times O(N) = O(N^2)$.
- **total :** $T(N) = O(N^2)$.
- **remarques** :
  - complexité en **moyenne** : $O(N^2)$.

Complexité spatiale :

- 4 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- **total :** $S(N) = O(N)$.

Propriétés :

- En place : **oui** (on trie en modifiant directement le tableau).
- Stabilité : **oui** (on recherche la position d'insertion en partant de la fin du sous-tableau trié, donc la position relative d'éléments de même clé n'est pas modifiée).

18.5 Tri fusion

18.5.1 Principe

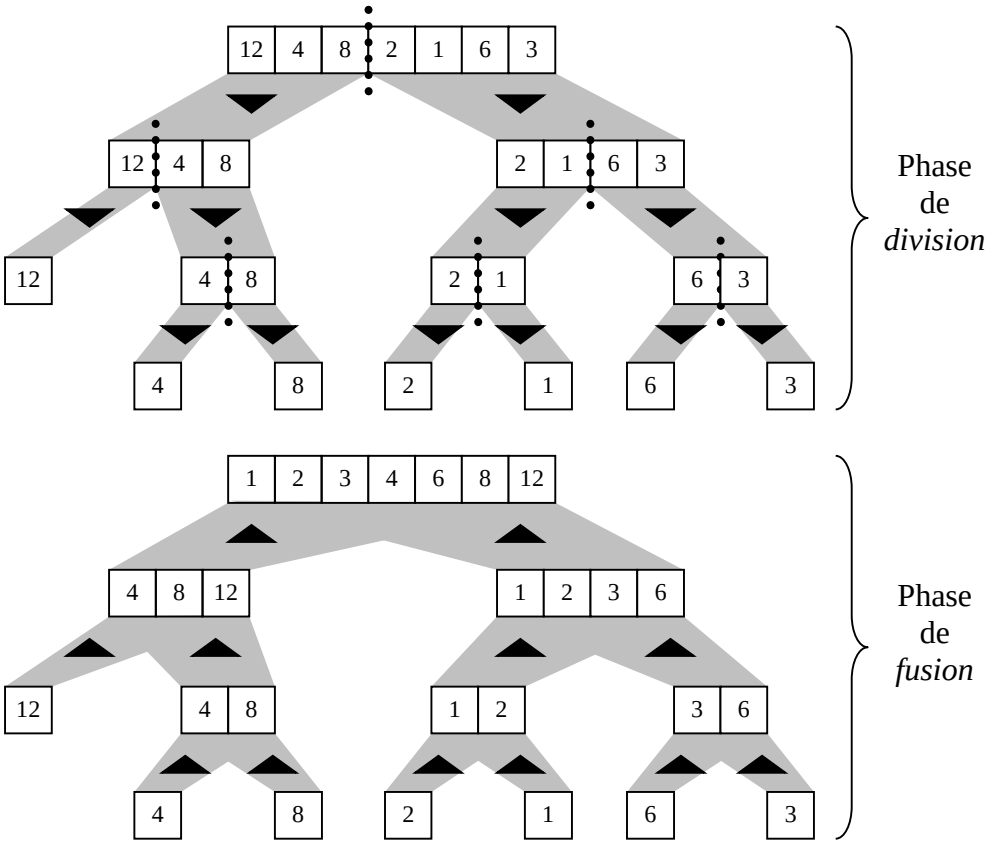
Le tri fusion[28] fonctionne sur le principe diviser-pour-régner : plusieurs petits problèmes sont plus faciles à résoudre qu'un seul gros problème.

Le tri se déroule en trois étapes :

1. *Division* : le tableau est coupé en deux (à peu près) en son milieu ;
2. *Tri* : chaque moitié est triée séparément récursivement ;
3. *Fusion* : les deux moitiés triées sont fusionnées pour obtenir une version triée du tableau initial.

exemple :

[28] En anglais : Merge Sort.



18.5.2 Implémentation

Chaque phase de l'algorithme est implémentée dans une fonction différente.

Soit `void division(int tab[], int tab1[], int taille1, int tab2[], int taille2)` la fonction qui divise le tableau d'entiers `tab` en deux parties stockées dans les tableaux `tab1` et `tab2`.

Les tailles de `tab1` et `tab2` sont respectivement `taille1` et `taille2`, et elles sont passées en paramètres (i.e. elles sont déjà calculées).

```
   void division(int tab[],  int tab1[],int taille1,
                             int tab2[],int taille2)
   {    int i;

1       for (i=0;i<taille1;i++)
2           tab1[i]=tab[i];
3       for (i=0;i<taille2;i++)
4           tab2[i]=tab[i+taille1];
   }
```

$O(N)$ (lignes 1–2) ; $O(1)$ (ligne 2) ; $O(N)$ (lignes 3–4) ; $O(1)$ (ligne 4) ; $O(N)$ (total)

Complexité temporelle :

- ligne `1` : `for` :
  - ligne `2` : affectation : $O(1)$.
  - répétitions : $N/2$ fois dans le pire des cas.
  - **total :** $N/2 \times O(1) = O(N)$.
- ligne `3` : `for` :
  - ligne `4` : affectation : $O(1)$.
  - répétitions : $N/2$ fois dans le pire des cas.

    - **total :** $N/2 \times O(1) = O(N)$.
- **total :** $T(N) = O(N) + O(N) = O(N)$.

Complexité spatiale :

- 3 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- 2 tableau de taille $N/2$: $O(N)$.
- **total :** $S(N) = O(N)$.

Soit `void fusion(int tab[], int tab1[], int taille1, int tab2[], int taille2)` la fonction qui fusionne deux tableaux d'entiers triés `tab1` et `tab2` en un seul tableau trié `tab`.

On connaît les tailles de `tab1` et `tab2`, qui sont respectivement `taille1` et `taille2`.

Complexité temporelle :

- ligne `1` : affectation : $O(1)$.
- ligne `2` : `while` :
    - ligne `3` : `if` :
        - lignes `4` et `5` : affectation : $O(1)$.
        - lignes `6` et `7` : affectation : $O(1)$.
        - **total :** $O(1)$.
    - ligne `8` : affectation : $O(1)$.
    - répétitions : $2N/2 - 1 = N - 1$ fois dans le pire des cas.
    - **total :** $(N-1) \times O(1) = O(N)$.
- ligne `9` : `if` :
    - ligne `10` : `for` :
        - lignes `11` et `12` : affectation : $O(1)$.
        - répétitions : $N/2$ fois dans le pire des cas.
        - **total :** $N/2 \times O(1) = O(N)$.
    - ligne `13` : `for` :
        - lignes `14` et `15` : affectation : $O(1)$.
        - répétitions : $N/2$ fois dans le pire des cas.
        - **total :** $N/2 \times O(1) = O(N)$.
    - **total :** $\max\big(O(N), O(N)\big) = O(N)$.
- **total :** $T(N) = O(1) + O(N) + O(N) = O(N)$.

Complexité spatiale :

- 6 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- 2 tableau de taille $N/2$: $O(N)$.
- **total :** $S(N) = O(N)$.

```
   void fusion(int tab[], int tab1[], int taille1,
                          int tab2[], int taille2)
1  {   int i=0,j=0,k=0,m;

2      while (i<taille1 && j<taille2)                    O(N)
3      {  if (tab1[i]<tab2[j])                           O(1)
4         {  tab[k]=tab1[i];                             O(1)
5            i++;                                        O(1)
          }
          else
6         {  tab[k]=tab2[j];                             O(1)
7            j++;                                        O(1)
          }
8         k++;                                           O(1)
       }
9      if (i!=taille1)                                   O(N)
10        for (m=i;m<taille1;m++)                        O(N)
11        {  tab[k]=tab1[m];                             O(1)
12           k++;                                        O(1)
          }
       else
13        for (m=j;m<taille2;m++)                        O(N)
14        {  tab[k]=tab2[m];                             O(1)
15           k++;                                        O(1)
          }
   }
```

Soit `void tri_fusion(int tab[], int taille)` la fonction qui trie récursivement un tableau d'entiers `tab` de taille `taille`, grâce aux fonctions `division` et `fusion`.

```
   void tri_fusion(int tab[], int taille)
1  {    int taille1=taille/2;                            O(1)
2       int taille2=taille-taille/2;                     O(1)
3       int tab1[taille1], tab2[taille2];                O(1)

4       if (taille>1)                                    2O(N) + 2T(N/2)
5       {  division(tab, tab1, taille1, tab2, taille2);  O(N)
6          tri_fusion(tab2,taille2);                     T(N/2)
7          tri_fusion(tab1,taille1);                     T(N/2)
8          fusion(tab, tab1, taille1, tab2, taille2);    O(N)
        }
   }
```

Complexité temporelle :

- complexité du **cas d'arrêt** $n = 1$:
  - lignes `1`, `2` et `3` : affectations : $O(1)$.
  - **total :** $T(1) = O(1)$.
- complexité du **cas général** $n > 0$:
  - lignes `1`, `2` et `3` : affectations : $O(1)$.
  - ligne `4` : `if` :
    - ligne `5` : `division` : $O(N)$.
    - lignes `6` et `7` : `tri_fusion` : $T(N/2)$.
    - ligne `8` : `fusion` : $O(N)$.
    - **total :** $2O(N) + 2T(N/2)$.
  - **total :** $T(N) = 2T(N/2) + 2O(N) + O(1) = 2T(N/2) + O(N)$.
- résolution de la **récurrence** :
  - on utilise le formulaire :

        - si $T(n) = cT(n/d) + \Theta(n^k)$ pour $c = d^k$
        - alors on a une complexité en $O(n^k \log n)$.
    - ici, on a :
        - $c = d = 2$ et $k = 1$.
        - d'où $T(N) = O(N \log N)$.
- **remarques** :
    - complexité en **moyenne** : $O(N \log N)$.

Complexité spatiale :

- 3 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- 2 tableaux de taille $N/2$: $O(N)$.
- profondeur de l'arbre d'appels : $\log N$.
- **total :** $S(N) = O(N \log N)$.

Propriétés :

- En place : **non**, car on utilise plusieurs tableaux lors des différentes phases. C'est essentiellement l'étape de fusion qui empêche de faire un tri en place.
- Stabilité : **oui** (lors de la fusion, en cas de clés identiques, on recopie d'abord la valeur du tableau gauche puis celle du tableau droit).

18.4 Tri rapide

18.4.1 Principe

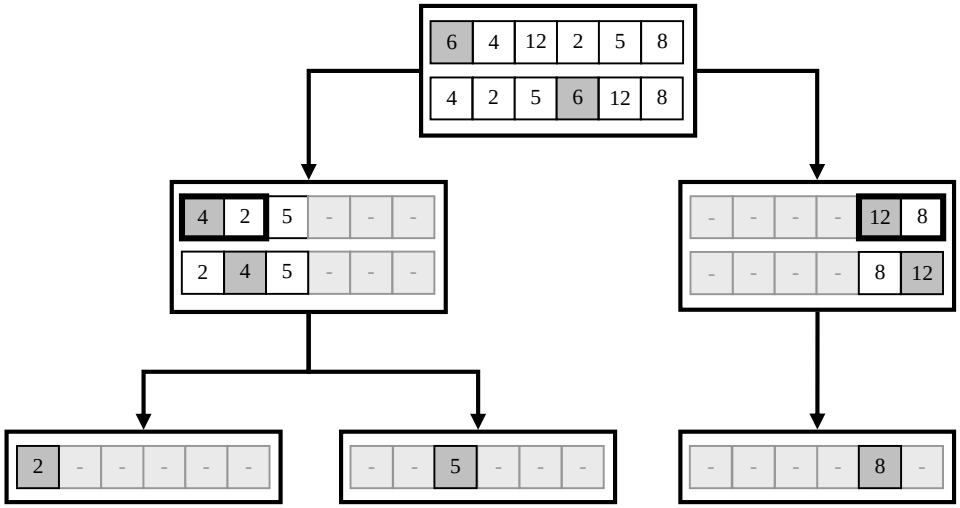
Le tri rapide[29] est également appelé : *tri de Hoare*, *tri par segmentation*, *tri des bijoutiers*… L'algorithme est le suivant :

1. On choisit un élément du tableau qui servira de *pivot*.
2. On effectue des échanges de valeurs dans le tableau, de manière à ce que :
    - Les valeurs *inférieures* au pivot soient placées à sa *gauche*.
    - Les valeurs *supérieures* au pivot soient placées à sa *droite*.
3. On applique récursivement ce traitement sur les deux parties du tableau (i.e. à *gauche* et à *droite* du pivot).

Remarque : on peut remarquer que le choix du pivot est *essentiel*. L'idéal est un pivot qui partage le tableau en deux parties de tailles égales. Mais déterminer un tel pivot coûte trop de temps, c'est pourquoi il est en général choisi de façon arbitraire. Ici par exemple, on prend la première case du tableau. Entre ces deux extrêmes, des méthodes d'approximation peu coûteuses existent pour faire un choix de compromis.

exemple :

[29] En anglais : Quick Sort.



Dans figure ci-dessus, la valeur indiquée en gris correspond au pivot sélectionné pour diviser le tableau en deux.

18.4.2 Implémentation

Soit `void tri_rapide(int tab[], int d, int f)` la fonction récursive qui trie le tableau d'entiers `tab` de taille *N* en utilisant le principe du tri rapide.

Les paramètres `d` et `f` marquent respectivement le début et la fin du sous-tableau en cours de traitement. Ces deux paramètres valent donc respectivement `0 et N-1` lors du premier appel.

```
   void tri_rapide(int tab[], int d, int f)
1  {  int taille=f-d+1;
      int i,j,temp;

2     if (taille>1)
3     {  j=d;                                   O(1)
4        for (i=d+1;i<=f;i++)
5        {  if (tab[i]<tab[d])
6           {  j++;                             O(1)
7              temp=tab[j];                     O(1)
8              tab[j]=tab[i];                   O(1)
9              tab[i]=temp;                     O(1)
           }                                     } O(1)  } O(N)
        }
10      temp=tab[d];                            O(1)
11      tab[d]=tab[j];                          O(1)
12      tab[j]=temp;                            O(1)
13      if (j>d)
14         tri_rapide(tab,d,j-1);               T(N/2)
15      if (j<f)
16         tri_rapide(tab,j+1,f);               T(N/2)
      }                                                  O(1) + O(N) + 2T(N/2)
   }
```

Complexité temporelle :

- complexité du **cas d'arrêt** $n = 1$:
  - ligne `1` : affectation : $O(1)$.
  - **total :** $T(1) = O(1)$.
- complexité du **cas général** $n > 0$:
  - ligne `1` : affectation : $O(1)$.

    - ligne `2` : `if` :
        - ligne `3` : affectations : $O(1)$.
        - ligne `4` : `for` :
            - ligne `5` : `if` :
                - lignes `6`, `7`, `8` et `9` : affectation : $O(1)$.
            - répétitions : $N-1$ fois si le pivot est le min et se trouve au début ou est le max et se trouve à la fin.
            - **total :** $(N-1)O(1) = O(N)$.
        - lignes `10`, `11`, et `12` : affectations : $O(1)$.
        - ligne `13` : `if` :
            - ligne `14` : `tri_rapide` : $T(N-1)$.
        - ligne `15` : `if` :
            - ligne `16` : `tri_rapide` : $T(1) = O(1)$.
        - **total :** $O(1) + O(N) + O(1) + T(N-1) + O(1)$.
    - **total :** $T(N) = O(N) + T(N-1)$.
- résolution de la **récurrence** :
    - on utilise le formulaire :
        - si $T(n) = T(n-1) + \Theta(n^k)$
        - alors on a une complexité en $O(n^{k+1})$.
    - ici, on a :
        - $k = 1$.
        - d'où $T(N) = O(N^2)$.
- **remarques :**
    - complexité en **moyenne** : $O(N \log N)$.
    - le temps dont l'algorithme a besoin pour trier le tableau dépend **fortement** du **pivot** choisi :
        - Dans le pire des cas, le pivot se retrouve complètement au début ou à la fin du tableau. On a donc un sous-tableau de taille $N-1$ et un autre de taille 0, et on obtient une complexité en $O(N^2)$.
        - Dans le meilleur des cas, le pivot sépare le tableau en deux sous-tableaux de taille égale $N/2$. On retrouve alors la même complexité que pour le tri fusion : $O(N \log N)$.

Complexité spatiale :

- 6 variables simples : $O(1)$.
- 1 tableau de taille N : $O(N)$.
- profondeur de l'arbre d'appels (pire des cas) : N.
- **total :** $S(N) = O(N^2)$.
- **remarque** : si on considère que le même tableau est passé à chaque fois, on a $S(N) = O(N)$.

Propriétés :

- En place : **oui** (on travaille directement dans le tableau initial).
- Stabilité : **non** (lors du déplacement des valeurs en fonction du pivot, des valeurs de même clé peuvent être échangées).

19 Arbres

Pour conclure ces notes de cours, nous allons étudier une structure de données un peu plus avancées que les piles et les files. Les arbres[30] sont un concept lui aussi très utilisé dans de nombreux contextes de l'informatique, non seulement en tant que structure de données, mais aussi plus généralement comme outil de modélisation. Il existe de nombreuses variantes de la structure d'arbre[31], et nous allons nous concentrer sur les plus simples.

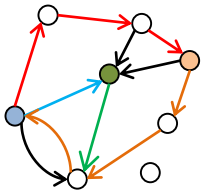
19.1 Définitions

19.1.1 Graphes

Le concept d'arbre est construit sur celui, plus général, de *graphe*. Comme les graphes sont utilisés dans de très nombreux domaines des sciences, notamment pour la modélisation, la terminologie qui s'y réfère n'est pas très normalisée. Cela signifie que les termes peuvent varier d'un auteur à l'autre, et que plusieurs termes différents peuvent désigner le même concept. Dans le cadre de ce cours, nous donnons les termes que nous pensons être les plus répandus.

Graphe orienté : un graphe $G = (V, E)$ est un couple formé de deux ensembles : un ensemble V de *nœuds* (ou sommets) et un ensemble E de *liens orientés* (ou arcs), qui relient les nœuds entre eux.

La figure ci-dessous (faites abstraction des couleurs, pour l'instant) donne un exemple de graphe, les cercles représentant les nœuds et les flèches les liens.



Un lien correspond lui-même formellement à un *couple* de nœud, et on a $E \subset V \times V$. On distingue le nœud de *départ* (ou source) et le nœud d'*arrivée* (ou cible). Par exemple, dans la figure ci-dessus, le nœud source du lien bleu est le nœud bleu, et son nœud cible est le nœud vert.

Parent d'un nœud : le parent d'un nœud N_1 est un autre nœud N_2 tel qu'il existe un lien allant de N_2 vers N_1, i.e. formellement : $(N_2, N_1) \in E$.

De façon symétrique, on définir la notion de fils :

Fils d'un nœud : le fils d'un nœud N_1 est un autre nœud N_2 tel qu'il existe un lien allant de N_1 vers N_2, i.e. formellement : $(N_1, N_2) \in E$.

Quand un nœud n'a pas de parent, on dit que c'est une *racine* :

Racine : nœud n'ayant aucun parent.

Les notions de *chemin* et de *cycle* découlent directement des concepts de nœud et lien.

Chemin entre deux nœuds : séquence de liens *consécutifs* permettant de relier (indirectement) les deux nœuds.

On dit que deux liens sont consécutifs quand le nœud cible de l'un est le nœud source de l'autre. Par exemple, dans la figure ci-dessus, les bleu et vert sont consécutifs.

[30] En anglais : Tree.

[31] Le concept mathématique d'*arbre*, ainsi que celui plus général de *graphe*, seront étudiés dans le cours de mathématiques discrètes.

La séquence de trois liens représentés en rouge dans la figure ci-dessus correspond à un chemin reliant le nœud bleu au nœud orange.

Cycle : chemin *fermé*, i.e. dont le premier et le dernier nœud sont confondus.

Dans la figure précédente, la séquence formée du chemin rouge et du chemin orange correspond à un cycle.

Longueur d'un chemin : nombre de liens composant le chemin.

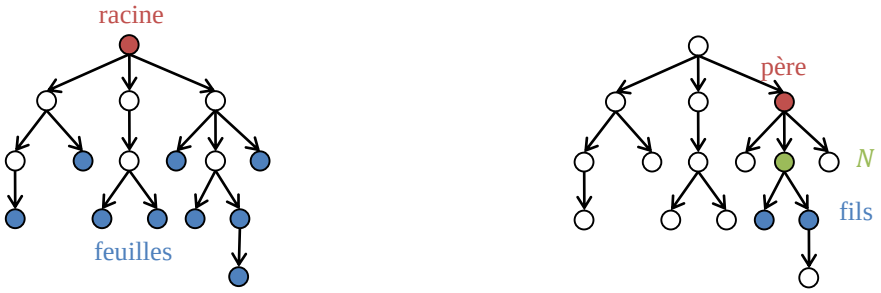
Dans la figure précédente, la longueur du chemin rouge est 3.

19.1.2 Arbres

Un *arbre* est une espèce particulière de graphe orienté :

Arbre : graphe *acyclique* orienté dans lequel chaque nœud n'a qu'un seul parent, sauf la racine unique qui n'en a pas du tout.

Le terme *acyclique* signifie que l'arbre ne contient aucun cycle. Un arbre ne contient qu'une seule racine (i.e. nœud sans parent) et tous les autres nœuds ont exactement un parent. Le résultat est une structure hiérarchique, telle que celles données en exemples dans la figure ci-dessous :



arbres d'arité 3

Chaque nœud contient une information appelée *label* (ou étiquette, ou clé). À cause des contraintes spécifiques aux arbres, un père peut avoir plusieurs fils, qui sont alors des *frères*.

Frères : ensemble de nœuds ayant le même parent.

Il existe aussi des nœuds qui n'ont aucun fils, et que l'on appelle les feuilles de l'arbre :

Feuille : nœud ne possédant aucun fils.

Un arbre est caractérisé par son *arité* :

Arité d'un arbre : nombre maximal de fils que peut avoir un nœud dans cet arbre.

En cas d'arité 2, on parle d'arbre binaire, puis d'arbre ternaire pour une arité 3, et plus généralement arbre n-aire pour une arité n.

19.1.3 Chemins

Par définition, il n'y a de pas de cycle (chemin fermé) dans un arbre.

arbres de hauteur 4

Les seuls chemins possibles sont *descendants*, i.e. ils partent d'un nœud et s'éloignent de la racine. Ceux reliant la racine à une feuille sont appelés branches :

Branche : chemin allant de la racine d'un arbre à une de ses feuilles.

En ce qui concerne les nœuds, on peut généraliser les concepts de parent et fils en définissant les ascendants et descendants :

Ascendants (ou ancêtres) d'un nœud : ensemble des nœuds constituant le chemin qui va de la racine au nœud considéré.

Par définition, la racine est donc un ascendant de tous les nœuds contenus dans l'arbre.

Descendants d'un nœud : ensemble des nœuds dont le nœud considéré est l'ascendant.

On peut également caractériser de façon numérique la position d'un nœud dans l'arbre :

Profondeur d'un nœud : longueur du chemin allant de la racine jusqu'au nœud considéré.

Enfin, on définir également des sous-structures, telles que la notion de *sous-arbre* :

Sous-arbre : un sous-arbre $A' = (V', E')$ d'un arbre $A = (V, E)$ est un arbre défini à partir d'un nœud $N \in V$ et tel que : 1) $N \in V'$ est la racine de A' ; 2) $V' \subset V$ est l'ensemble des descendants de N ; et 3) $E' \subset E$ est l'ensemble des liens entre les nœuds contenus dans V'.

Chaque nœud qui n'est pas une feuille est lui-même la racine d'un sous-arbre.

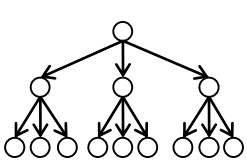
19.1.4 Propriétés

Un arbre peut lui aussi être caractérisé par différentes propriétés.

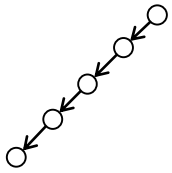
Hauteur d'un arbre : profondeur de son nœud le plus profond.

La hauteur décrit la taille de l'arbre. D'autres propriétés décrivent sa forme :

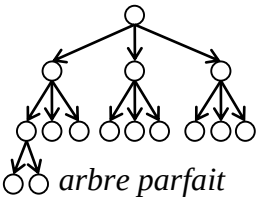
Arbre complet : arbre n-naire de hauteur h dans lequel tout nœud de profondeur strictement inférieure à h possède exactement n fils.



arbre complet



arbre dégénéré



arbre parfait

Un arbre complet est un arbre régulier, dans le sens où tous les nœuds possèdent le même nombre de liens (sauf les feuilles, évidemment). Un autre type d'arbre régulier est l'arbre *linéaire*, que l'on a déjà abordé lorsque l'on a étudié les arbres d'appel (section 13.3) :

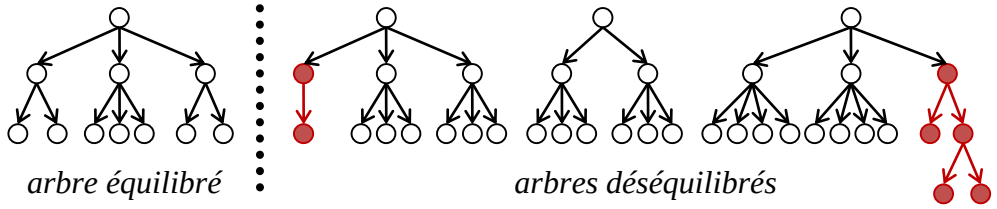
Arbre linéaire (ou dégénéré) : chaque nœud qui n'est pas une feuille ne possède qu'un seul fils.

On peut considérer qu'un arbre linéaire est un arbre complet unaire. Si on relâche partiellement la contrainte de régularité, on peut définir l'arbre *parfait* :

Arbre parfait : arbre n-naire de hauteur h dans lequel tout nœud de profondeur strictement inférieure à $h - 1$ possède exactement n fils, et dont toutes les feuilles sont regroupées à gauche de l'arbre.

La différence avec l'arbre complet est que le dernier niveau n'est pas forcément régulier. Enfin, terminons avec la notion d'arbre *équilibré* :

Arbre équilibré : arbre n-naire tel que ses n sous-graphes contiennent le même nombre de nœuds à un nœud près, et sont tous eux-mêmes équilibrés.



Remarque : dans la figure, l'arbre déséquilibré situé au centre a 2 sous-arbres contenant chacun 4 nœuds, ce qui est équilibré. Cependant, il s'agit visiblement d'un arbre ternaire, donc il manque toute un sous-arbre partant de la racine, ce qui explique qu'on le considère comme déséquilibré.

Notez que, parmi toutes les notions présentées ici, celle d'équilibre est particulièrement variable d'un auteur à l'autre. En effet, on peut définir de nombreux autres critères d'équilibre. Par exemple, au lieu de compter les nœuds contenus dans les sous-arbres, on pourrait utiliser leur hauteur.

19.2 Arbres binaires

Après avoir défini de façon théorique ce qu'est un arbre en général, nous allons nous intéresser plus particulièrement aux arbres binaires, qui sont une forme simple d'arbre possédant de nombreuses applications en informatique.

Arbre binaire : arbre dans lequel un nœud peut avoir 0, 1 ou 2 fils.

Un arbre binaire peut être décomposé en trois parties :

- Sa *racine* ;
- Son sous-arbre *gauche* ;
- Son sous-arbre *droit*.

De même, chacun des sous-arbres peut être lui aussi décomposé de la même manière. Le type abstrait *arbre binaire* est basé sur cette décomposition récursive.

19.2.1 Type abstrait

Comme pour les piles et les files de données, le type abstrait proposé ici se compose d'un ensemble d'opérations et d'un ensemble de contraintes qui leur sont appliquées.

Opérations

- `cree_arbre` : créer un arbre vide
  `_ → arbre`
- `est_vide` : déterminer si l'arbre est vide
  `arbre → booleen`
- `racine` : renvoyer la valeur située à la racine de l'arbre
  `arbre → valeur`
- `fils_gauche` : renvoyer le sous-arbre gauche d'un nœud
  `arbre → arbre`
- `fils_droit` : renvoyer le sous-arbre droit d'un nœud
  `arbre → arbre`
- `enracine` : créer un arbre à partir d'une valeur et de deux sous-arbres
  `valeur × arbre × arbre → arbre`

Les opérations permettant de créer un arbre et de tester s'il est vide sont similaire à celles déjà définies pour les piles et les files.

Les opérations `racine`, `fils_gauche` et `fils_droit` sont des opérations d'accès, elles permettent de récupérer les trois constituants de l'arbre et d'y naviguer. À noter que racine

renvoie la valeur associée à la racine de l'arbre (i.e. son label ou sa clé), alors que `fils_gauche` et `fils_droit` renvoient ses deux sous-arbres, qui sont eux-mêmes du type `arbre`. On voit déjà qu'on aura une structure de données récursive, comme c'était déjà le cas avec les listes chaînées.

L'opération enracine permet de construire un arbre : elle crée un nouveau nœud en associant une racine et deux sous-arbres. Elle renvoie le nouvel arbre obtenu.

Axiomes

∀ `v, a1, a2` ∈ `valeur × arbre × arbre` :

- `est_vide` :
  1. `est_vide(cree()) = vrai`
  2. `est_vide(enracine(v,a1,a2)) : faux`
- `racine` :
  3. `racine(cree())` = indéfini (i.e. opération interdite)
  4. `racine(enracine(v,a1,a2)) = n`
- `fils_droit` :
  5. `fils_droit(cree())` = indéfini
  6. `fils_droit(enracine(v,a1,a2)) = a2`
- `fils_gauche` :
  7. `fils_gauche(cree())` = indéfini
  8. `fils_gauche(enracine(v,a1,a2)) = a1`

D'après l'axiome 1, un arbre qui vient d'être créé est forcément vide. Au contraire, l'axiome 2 stipule que qu'un arbre auquel on vient d'associer un nœud et deux sous-arbres (ceux-ci pouvant être eux-mêmes vides) ne peut pas être vide.

L'axiome 3 indique qu'un arbre vide ne contient aucun nœud, et qu'on ne peut donc pas demander sa racine. Au contraire, l'axiome 4 dit que racine renverra le dernier nœud inséré dans l'arbre, qui se trouve donc être sa racine.

Les axiomes 5 et 7 indiquent qu'un arbre vide ne contient pas de sous-arbre droit ni gauche. Les axiomes 6 et 8, au contraire, stipulent qu'un arbre non-vide contient forcément des sous-arbres droit et gauche, ceux-ci pouvant éventuellement être eux-mêmes vides.

19.2.2 Implémentation par pointeur

Les listes, que l'on avait utilisées pour implémenter les types abstraits *piles* et *files*, ne sont pas adaptés type abstrait *arbre binaire*. Nous allons définir une structure de données spécifiquement dans ce but.

19.2.2.1 Structure de données

Dans notre structure de données, chaque nœud doit contenir les informations suivantes :

- Une *valeur* représentant son *étiquette* ;
- Un *pointeur* vers son *fils gauche* ;
- Un *pointeur* vers son *fils droit*.

Supposons, sans perte de généralité, que l'on veut manipuler un arbre dont les nœuds contiennent des entiers `int`. Alors, on utilisera la structure suivante :

```
typedef struct s_noeud
{  int valeur ;
   struct s_noeud *gauche;
   struct s_noeud *droit;
} noeud;

typedef noeud* arbre;
```

Le premier type est une structure récursive appelée `noeud`. Elle contient bien la valeur entière et deux pointeurs vers les sous-arbres gauche et droite. À noter l'utilisation d'un nom `s_noeud` pour la structure elle-même, comme on l'avait fait pour définir les éléments d'une liste chaînée en section 14.2.1.

Le second type est seulement un nouveau nom `arbre` associé à un pointeur sur un `noeud`. Concrètement, cela signifie que les types `struct s_noeud*`, `noeud*` et `arbre` sont synonymes.

Un arbre vide sera représenté par un pointeur `NULL`.

19.2.2.2 Création

Pour la création, on déclare une variable de type `arbre` et on l'initialise à `NULL` :

```
arbre cree_arbre()
{  return NULL;
}
```

Comme on l'a vu, l'arbre sera vide s'il vaut `NULL` :

```
int est_vide(arbre a)
{  return (a == NULL);
}
```

19.2.2.3 Accès

L'accès à la racine est direct : si l'arbre n'est pas vide, on peut directement renvoyer sa valeur. Sinon, l'opération n'est pas définie et on renvoie un code d'erreur.

Soit `int racine(arbre a, int *r)` la fonction qui permet d'accéder à la racine de l'arbre `a`. La fonction transmet la valeur par adresse grâce au paramètre `r`. La fonction renvoie -1 en cas d'erreur (si l'arbre est vide) ou 0 sinon.

```
int racine(arbre a, int *r)
{  int erreur = 0;

   if(est_vide(a))
      erreur = -1;
   else
      *r = a->valeur;

   return erreur;
}
```

L'accès aux fils est symétrique : soit `arbre fils_gauche(arbre a, arbre *f)` la fonction qui permet d'accéder au fils gauche de `a`. La fonction transmet le nœud par adresse grâce à la variable `f`. La fonction renvoie -1 en cas d'erreur (si l'arbre est vide) ou 0 sinon.

```
int fils_gauche(arbre a, arbre *f)
{  int erreur = 0;

   if(est_vide(a))
      erreur = -1;
   else
      *f = a->gauche;
   return erreur;
}
```

La fonction `arbre fils_droit(arbre a, arbre *f)` est obtenue simplement en remplaçant le mot `gauche` (indiqué en **gras**) par `droit` dans le code source ci-dessus.

19.2.2.4 Insertion

L'enracinement consiste à créer un nouvel arbre à partir de deux sous-arbres et d'un nœud racine.

Soit `arbre enracine(int v, arbre a1, arbre a2)` la fonction qui crée et renvoie un arbre dont la racine est `n` et dont les sous-arbres gauche et droit sont respectivement `a1` et `a2`.

```
arbre enracine(int v, arbre a1, arbre a2)
{  noeud *n;

   if((n = (noeud *) malloc(sizeof(noeud))) != NULL)
   {  n->valeur = v;
      n->gauche = a1;
      n->droit = a2;
   }
   return n;
}
```

19.2.3 Implémentation par tableau

19.2.3.1 Structure de données

à compléter (laissé en exercice !)

19.2.3.2 Création

à compléter (laissé en exercice !)

19.2.3.3 Accès

à compléter (laissé en exercice !)

19.2.3.4 Insertion

à compléter (laissé en exercice !)

19.2.4 Exercices

19.2.4.1 Types de parcours

1) Écrivez une fonction `void parcours_prefixe(arbre a)` qui affiche le parcours *préfixe* d'un arbre `a`.

```
void parcours_prefixe(arbre a)
{  int v;
   arbre g,d;

   if (!est_vide(a))
   {  racine(a,&v);
      printf("%d ",v);
      fils_gauche(a,&g);
      parcours_prefixe(g);
      fils_droit(a,&d);
      parcours_prefixe(d);
   }
}
```

2) Écrivez une fonction `void parcours_infixe(arbre a)` qui affiche le parcours *infixe* d'un arbre `a`.

```
void parcours_infixe(arbre a)
{  int v;
   arbre g,d;

   if (!est_vide(a))
   {  fils_gauche(a,&g);
      parcours_infixe(g);
      racine(a,&v);
```

```
        printf("%d ",v);
        fils_droit(a,&d);
        parcours_infixe(d);
    }
}
```

3) Écrivez une fonction `void parcours_postfixe(arbre a)` qui affiche le parcours *postfixe* d'un arbre `a`.

```
void parcours_postfixe(arbre a)
{  int v;
   arbre g,d;

   if (!est_vide(a))
   {  fils_gauche(a,&g);
      parcours_postfixe(g);
      fils_droit(a,&d);
      parcours_postfixe(d);
      racine(a,&v);
      printf("%d ",v);
   }
}
```

19.2.4.2 Hauteur

4) Écrivez une fonction `int hauteur(arbre a)` qui calcule récursivement la hauteur d'un arbre `a`.

```
int hauteur(arbre a)
{  int resultat = 0
   int pg,pd;
   arbre g,d;

   if (!est_vide(a))
   {  fils_gauche(a,&g);
      pg=hauteur(g);
      fils_droit(a,&d);
      pd=hauteur(d);
      if (pg>pd)
         resultat = 1+pg;
      else
         resultat = 1+pd;
   }
   return resultat;
}
```

19.2.4.3 Génération aléatoire

5) Écrivez une fonction `arbre genere_arbre(int h)` qui génère aléatoirement un arbre binaire parfait de hauteur `h`.

```
arbre genere_arbre(int h)
{  arbre resultat,g,d;
   int v;

   if(h==0)
      resultat = cree_arbre();
   else
   {  g=genere_arbre(h-1);
      d=genere_arbre(h-1);
      v=rand()*100/RAND_MAX;
      resultat = enracine(v, g, d);
   }
   return resultat;
}
```

19.2.4.1 Affichage vertical

6) Écrivez une fonction récursive `void affiche_arbre(arbre a, int niveau)` qui affiche verticalement l'arbre binaire `a`.

exemple :

```
      23
   45
      65
26
      78
   56
      89
```

```
void affiche_arbre(arbre a, int niveau)
{  int v,i;
   arbre g,d;

   if (!est_vide(a))
   {  fils_droit(a,&g);
      affiche_arbre(g,niveau+1);
      racine(a,&v);
      for(i=0;i<niveau;i++)
         printf("\t");
      printf("%d\n",v);
      fils_gauche(a,&d);
      affiche_arbre(d,niveau+1);
   }
}
```

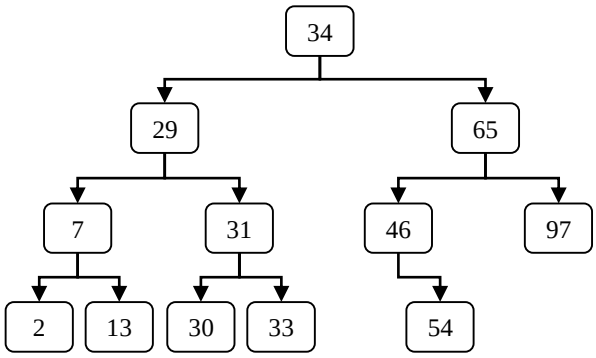
19.3 Arbres binaires de recherche

Un arbre binaire de recherche est un type spécifique d'arbre binaire, tel que pour tout nœud *N* qui n'est pas une feuille :

- Tout nœud appartenant au sous-arbre de gauche (s'il existe) possède une étiquette dont la valeur est *inférieure* à celle de *N*.
- Tout nœud appartenant au sous-arbre de droite (s'il existe) possède une étiquette dont la valeur est *supérieure* à celle de *N*.
- Les deux sous-arbres sont eux-mêmes des arbres binaires de recherche.

Donc, les valeurs contenues dans l'arbre ne sont pas réparties comme on veut : elles doivent suivre un certain nombre de règles qui permettent de les organiser. Ces règles rendent l'implémentation plus compliquée, mais en contrepartie permettent un accès plus rapide au contenu de l'arbre.

exemple :



Toutes les valeurs à gauche de la racine 34 lui sont strictement inférieures. Toutes les valeurs à droite lui sont strictement supérieures. De la même façon, le sous-arbre dont la racine

est 29 est lui-même un arbre de recherche : toutes les valeurs à sa gauche sont inférieures à 29, et toutes celles à sa droite lui sont supérieure.

Il faut noter que comme pour les tris (section 18), le concept d'arbre de recherche est défini indépendamment de la relation d'ordre utilisée pour comparer les clés des nœuds. Autrement dit, on peut utiliser une autre relation d'ordre que celle choisie dans l'exemple ci-dessus, sans pour autant devoir modifier le type abstrait *arbre de recherche*.

Remarque : un arbre binaire de recherche ne peut pas contenir plusieurs fois la même valeur.

19.3.1 Type abstrait

Le type abstrait arbre binaire de recherche reprend celui des arbres binaires, complété de nouvelles opérations et axiomes.

Opérations

- `cherche` : rechercher une valeur dans l'arbre.
  `arbre × valeur → booleen`
- `minimum/maximum` : obtenir le minimum/maximum de l'arbre.
  `arbre → arbre`
- `predecesseur/successeur` : obtenir le prédécesseur/successeur d'un nœud.
  `arbre → arbre`
- `insere` : insérer une valeur au niveau des feuilles, au bon endroit.
  `arbre × valeur → arbre`
- `supprime` : supprimer une valeur de l'arbre.
  `arbre × valeur → arbre`

L'opération `recherche` permet de déterminer si l'arbre contient une valeur en particulier. Les opérations `minimum` et `maximum` renvoie les valeurs respectivement les plus petite et grande.

Les opérations `predecesseur` et `successeur` renvoie non pas des valeurs, mais des nœuds. Il s'agit des nœuds précédant ou suivant directement un nœud donné. Cet ordre est relatif à la relation utilisée pour organiser les nœuds dans l'arbre.

L'opération `insere` permet de rajouter une nouvelle valeur dans l'arbre. Celle-ci sera insérée à l'endroit approprié, de manière à conserver les propriétés caractéristiques d'un arbre de recherche. Même chose pour `supprime`, qui est l'opération réciproque consistant à retirer une valeur déjà contenue dans l'arbre.

Remarque : l'insertion peut également se faire à la racine (plus compliqué).

Axiomes

∀ `v, x, a1, a2` ∈ `valeur × valeur × arbre × arbre` :

- `cherche` :
  1. `cherche(cree(),x) = faux`
  2. `cherche(enracine(v,a1,a2),x)`
     - si `x=v` : `vrai`
     - si `x<v` : `cherche(a1,x)`
     - si `x>v` : `cherche(a2,x)`
- `maximum/minimum` :

3. `maximum/minimum(cree())` : indéfini
4. `minimum(enracine(v,a1,a2))`
   - si `a1=cree()` : `enracine(v,a1,a2)`
   - sinon : `minimum(a1)`
5. `maximum(enracine(v,a1,a2))`
   - si `a2=cree()` : `enracine(v,a1,a2)`
   - sinon : `maximum(a2)`

- `predecesseur/successeur` :
  6. `predecesseur/successeur(cree())` : indéfini
  7. `predecesseur(enracine(v,a1,a2))`
     - si `a1=cree()` : `cree()`
     - sinon : `maximum(a1)`
  8. `successeur(enracine(v,a1,a2))` :
     - si `a2=cree()` : `cree()`
     - sinon : `minimum(a2)`
- `insere` :
  9. `insere(cree(),x)= enracine(x,cree(),cree())`
  10. `insere(enracine(v,a1,a2),x)`
      - si `x=v` : indéfini
      - si `x<v` : `enracine(v,insere(a1,x),a2)`
      - si `x>v` : `enracine(v,a1,insere(a2,x))`
- `supprime` :
  11. `supprime(cree(),x)` : indéfini
  12. `supprime(enracine(v,a1,a2),x)` :
      - si `x=v` :
        - si `a1=cree()` et `a2=cree()` : `cree()`
        - si `a1=cree()`: `a2`
        - si `a2=cree()`: `a1`
        - sinon : on pose `p` et `a3` tels que :
          - `p=racine(predecesseur(enracine(v,a1,a2)))`
          - `enracine(v,a3,a2)=supprime(enracine(v,a1,a2),p)`
        - résultat : `enracine(p, a3, a2)`
      - si `x<v` : `enracine(v,supprime(a1,x),a2)`
      - si `x>v` : `enracine(v,a1,supprime(a2,x))`

Comme on peut le voir, le nombre d'axiomes est augmenté de façon conséquente. L'axiome 1 indique qu'un arbre vide ne peut contenir aucune valeur : si on y cherche une valeur, on obtient le résultat *faux* quelle que soit la valeur. L'axiome 2 traite le cas d'un arbre non-vide. Si la racine correspond à la valeur recherchée, alors le résultat est *vrai*. Sinon, on cherche récursivement dans l'un des sous-arbres, en fonction de la comparaison de la valeur cherchée et de la racine.

L'axiome 3 dit qu'on ne peut pas obtenir les minimum et maximum d'un arbre vide. L'axiome 4 (resp. 5) stipule que le minimum (resp. maximum) d'un arbre est la valeur de sa racine dans le cas où son sous-arbre gauche (resp. droit) est vide, et que dans le cas contraire il s'agir du minimum (resp. maximum) de ce sous-arbre.

L'axiome 6 dit que les opérations `predecesseur` et `successeur` ne sont pas définir pour un arbre vide. L'axiome 7 (resp. 8) indique que le prédécesseur (resp. successeur) d'un nœud

est le minimum (resp. maximum) de son arbre gauche (resp. droit). Si l'arbre gauche (resp.droit) est vide, alors le prédécesseur (resp. successeur) est lui-même l'arbre vide.

L'axiome 9 indique que quand on insère une valeur dans un arbre vide, celle-ci devient sa racine. L'axiome 10 traite le cas d'un arbre non vide. Si la valeur est égale à la clé de la racine, l'opération `insere` n'est pas définie. Rappelons qu'un arbre de recherche ne peut pas contenir plusieurs occurrences de la même valeur. Sinon, si la valeur est inférieure (resp. supérieure) à la clé de la racine, alors on effectue l'insertion récursivement dans le sous-arbre gauche (resp. droit).

L'axiome 11 stipule qu'on ne peut pas supprimer une valeur d'un arbre vide, quelle que soit cette valeur. L'axiome 12 traite le cas d'un arbre non-vide. Si la valeur correspond à la clé de la racine, alors on considère les deux sous-arbres. Si les deux sont vides, alors cela signifie que l'arbre ne contenait qu'un seul élément, et le résultat de la suppression est alors l'arbre vide. Si un seul des deux sous-arbres est vide, alors il correspond au résultat de la suppression. Si aucun des deux sous-arbres n'est vide, on pose `p` la valeur du prédécesseur de la racine (rappelons que la valeur de la racine est celle que l'on veut supprimer) et `a3` le sous-arbre qu'on obtiendrait en supprimant `p` du sous-arbre gauche original. Notez la nature récursive de ce traitement. Le résultat est l'arbre contenant `p` en racine, `a3` en tant que sous-arbre gauche, et le sous-arbre droit original (qui n'est pas modifié). Toujours pour l'axiome 12, si la valeur à supprimer ne correspond pas à la racine, alors on répète le traitement récursivement sur le sous-arbre gauche (resp. droit) si la valeur est inférieure (resp. supérieure) à celle de la racine.

19.3.2 Implémentation par pointeur

On reprend exactement la même implémentation que pour les arbres binaires classiques, et on définit les nouvelles opérations sous la forme de fonctions supplémentaires.

19.3.2.1 Recherche

Soit `int recherche(arbre a, int v)` la fonction récursive qui recherche la valeur `v` dans l'arbre `a`. La fonction renvoie 1 si `v` apparaît dans l'arbre, et 0 sinon.

```
int recherche(arbre a, int v)
{  int resultat=0,valeur;
   arbre g,d;

   if(!est_vide(a))
   {  racine(a,&valeur);
      if(valeur==v)
         resultat = 1;
      else
      {  if(valeur>v)
         {  fils_gauche(a,&g);
            resultat = recherche(g, v);
         }
         else
         {  fils_droit(a,&d);
            resultat = recherche(d, v);
         }
      }
   }
   return resultat;
}
```

La complexité temporelle de cette fonction dépend de la hauteur de l'arbre :

- Pour un arbre *équilibré* contenant n nœuds, cette fonction a une complexité en $O(\log n)$, car on effectue une recherche *dichotomique* (cf. les TP pour plus de détails sur la recherche dichotomique).
- Par contre, si l'arbre est *dégénéré*, la complexité est en $O(n)$ (car on se ramène à une recherche dans une liste chaînée).

19.3.2.2 Minimum et maximum

Soit la fonction `void minimum(arbre a, arbre *m)` qui renvoie via le paramètre `m` le sous-arbre dont la racine a la valeur minimale dans l'arbre `a`. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

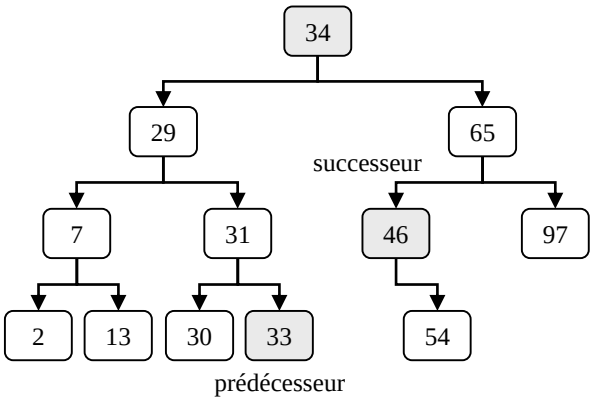
```
int minimum(arbre a, arbre *m)
{  int erreur=0;
    arbre g;

   if(est_vide(a))
      erreur=-1;
   else
   {  fils_gauche(a,&g);
      if (est_vide(g))
         *m=a;
      else
         minimum(g,m);
   }
   return erreur;
}
```

Soit la fonction `void maximum(arbre a, arbre *m)` qui renvoie via le paramètre `m` le sous-arbre dont la racine a la valeur maximale dans l'arbre `a`. La fonction renvoie -1 en cas d'erreur et 0 en cas de succès.

```
int maximum(arbre a, arbre *m)
{  int erreur=0;
    arbre d;

   if(est_vide(a))
      erreur=-1;
   else
   {  fils_droit(a,&d);
      if (est_vide(d))
         *m=a;
      else
         maximum(d,m);
   }
   return erreur;
}
```

19.3.2.3 Successeur et prédécesseur

Rappelons que dans un arbre binaire de recherche, on parle de *prédécesseur* d'un nœud N pour le nœud dont la valeur est *maximale* parmi les nœuds de valeur inférieure à celle de N. De même, on parle de *successeur* pour le nœud dont la valeur est *minimale* parmi les nœuds de valeur supérieure à celle de N.



Soit la fonction `int predecesseur(arbre a, arbre *p)` qui renvoie dans `p` le sous-arbre dont la racine contient la valeur précédant l'étiquette de `a` dans l'arbre. La fonction renvoie −1 en cas d'erreur et 0 en cas de succès.

```
int predecesseur(arbre a, arbre *p)
{  int erreur=0;
    arbre g;

   if(est_vide(a))
      erreur=-1;
   else
   {  fils_gauche(a,&g);
      if(est_vide(g))
         erreur=-1;
      else
         maximum(g,p);
   }
   return erreur;
}
```

Soit la fonction `int successeur(arbre a, arbre *p)` qui renvoie dans `p` le sous-arbre dont la racine contient la valeur succédant à l'étiquette de `a` dans l'arbre. La fonction renvoie −1 en cas d'erreur et 0 en cas de succès.

```
int successeur(arbre a, arbre *s)
{  int erreur=0;
    arbre d;

   if(est_vide(a))
      erreur=-1;
   else
   {  fils_droit(a,&d);
      if(est_vide(d))
         erreur=-1;
      else
         minimum(d,s);
   }
   return erreur;
}
```

19.3.2.4 Insertion

Soit `int insertion(arbre *a, int v)` la fonction qui insère au bon endroit la valeur `v` dans l'arbre `a`. La fonction renvoie 0 si l'insertion s'est bien passée, et −1 en cas d'échec (i.e. si la valeur est déjà dans l'arbre).

```
int insere(arbre *a, int v)
{  int valeur,erreur=0;
   arbre g,d;

   if(est_vide(*a))
      *a = enracine(v,cree_arbre(),cree_arbre());
   else
   {  racine(*a,&valeur);
      fils_gauche(*a,&g);
      fils_droit(*a,&d);
      if(v<valeur)
      {  insere(&g, v);
         *a = enracine(valeur,g,d);
      }
      else
      {  if(valeur<v)
         {  insere(&d, v);
            *a = enracine(valeur,g,d);
         }
         else
```

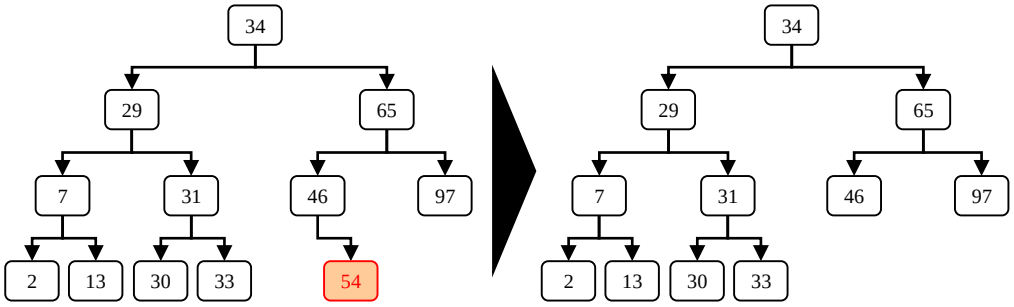
```
                erreur=-1;
            }
        }
        return erreur;
    }
```

La complexité temporelle de la fonction est la même que celle de recherche, puisque la méthode pour rechercher le bon endroit d'insertion est la même, et que l'insertion elle-même est effectuée en temps constant.

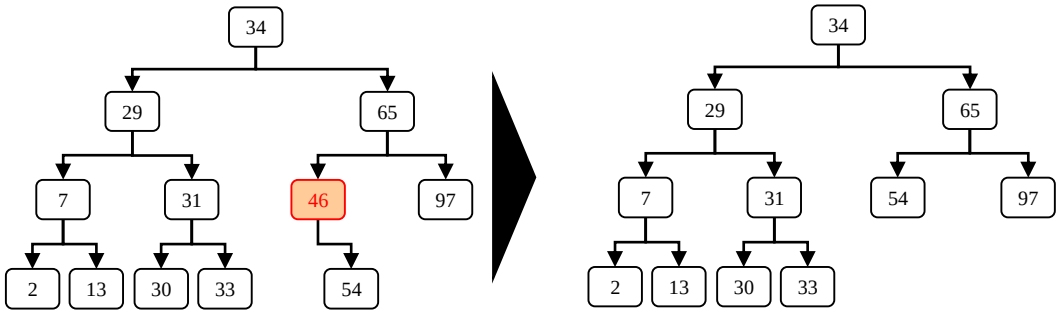
19.3.2.5 Suppression

La suppression dans un arbre binaire de recherche est plus complexe que l'insertion. Comme spécifié dans le type abstrait, plusieurs cas sont possibles :

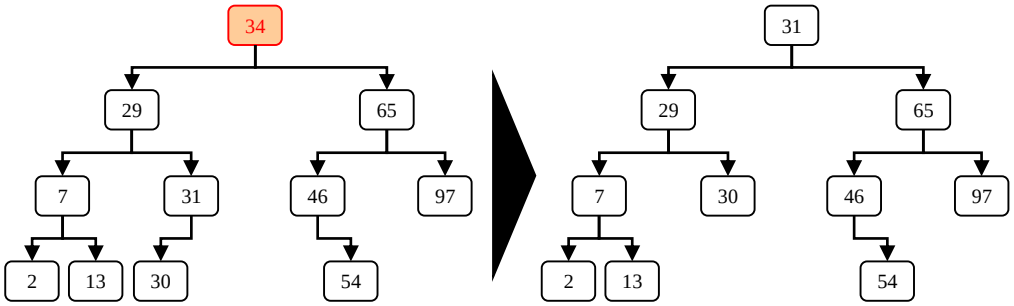
- Si le nœud que l'on veut supprimer est une *feuille* : on le supprime *directement*.



- Si le nœud possède *un seul fils* : le nœud est *remplacé* par son fils.



- Si le nœud possède *deux fils* : le nœud est remplacé par son *prédécesseur* ou son successeur.



Soit `int supression(arbre *a, int v)` la fonction qui supprime la valeur `v` dans l'arbre `a`. La fonction renvoie 0 si l'insertion s'est bien passée, et −1 en cas d'échec (si la valeur n'a pas été trouvée dans l'arbre).

```
int supprime(arbre *a, int v)
{   int erreur=0,valeur,valtemp;
    arbre g,d,temp;

    if(est_vide(*a))
       erreur=-1;
```

```
      else
      {  racine(*a,&valeur);
         fils_gauche(*a,&g);
         fils_droit(*a,&d);
         if(v<valeur)
         {  supprime(&g,v);
            *a = enracine(valeur,g,d);
         }
         else if(v>valeur)
         {  supprime(&d,v);
            *a = enracine(valeur,g,d);
         }
         else
         {  if(est_vide(g))
            {  if(est_vide(d))
               {  free(*a);
                  *a=cree_arbre();
               }
               else
               {  temp=*a;
                  *a = d;
                  free(temp);
               }
            }
            else
            {  if(est_vide(d))
               {  temp=*a;
                  *a = g;
                  free(temp);
               }
               else
               {  predecesseur(*a,&temp);
                  racine(temp,&valtemp);
                  supprime(&g,valtemp);
                  *a = enracine(valtemp,g,d);
               }
            }
         }
      }
      return erreur;
   }
```

La complexité de la fonction est la même que pour celle de recherche.

19.3.3 Implémentation par tableau

19.3.3.1 Recherche

à compléter (laissé en exercice !)

19.3.3.2 Minimum et maximum

à compléter (laissé en exercice !)

19.3.3.3 Successeur et prédécesseur

à compléter (laissé en exercice !)

19.3.3.4 Insertion

à compléter (laissé en exercice !)

19.3.3.5 Suppression

à compléter (laissé en exercice !)

19.3.4 Exercices

19.3.4.1 Génération aléatoire

Écrivez une fonction `arbre genere_arbre_rech(int n)` qui génère aléatoirement un arbre binaire de recherche contenant `n` nœuds.

```
arbre genere_arbre_rech(int n)
{  int v,i;
   arbre a;

   a = cree_arbre();
   for (i=0;i<n;i++)
   {  do
         v = rand()*100/RAND_MAX;
      while(insere(&a,v)==-1);
   }
   return a;
}
```

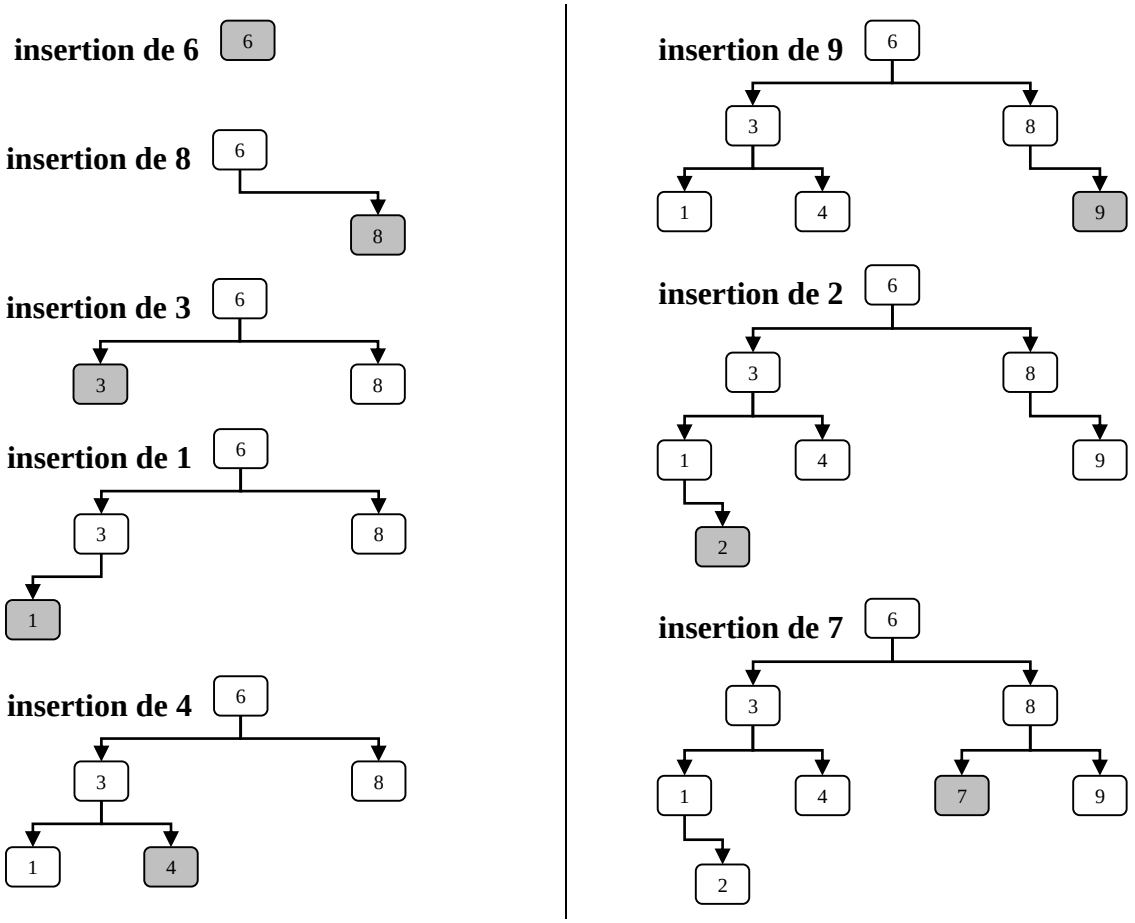
19.3.4.2 Tri par arbre binaire de recherche

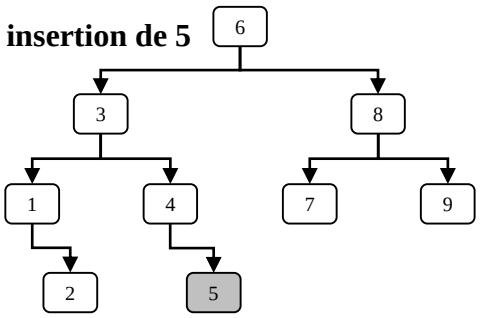
Le tri utilisant un arbre binaire de recherche se fait en deux étapes :

1) On construit l'arbre binaire de recherche en insérant un par un dans un arbre vide les éléments de la séquence à trier.
2) On construit la séquence triée en effectuant un parcours infixe de l'arbre.

exemple : On veut trier la séquence d'entiers $(6, 8, 3, 1, 4, 9, 2, 7, 5)$

- **construction** de l'**arbre** :





- **construction** de **l'ensemble trié** :
  - le parcours infixe donne la séquence *(1, 2, 3, 4, 5, 6, 7, 8, 9)*.

Écrivez une fonction `void parcours_infixe_tab(arbre a, int tab[], int *index)` qui recopie récursivement dans un tableau les valeurs contenues dans un arbre binaire de recherche. Le paramètre `a` désigne l'arbre à traiter, `tab` est le tableau à remplir, et `index` est le numéro de la prochaine case du tableau qu'il faudra remplir.

```
void parcours_infixe_tab(arbre a, int tab[], int *index)
{  int v;
   arbre g,d;

   if (!est_vide(a))
   {  fils_gauche(a,&g);
      parcours_infixe_tab(g,tab,index);
      racine(a,&v);
      tab[*index]=v;
      (*index)++;
      fils_droit(a,&d);
      parcours_infixe_tab(d,tab,index);
   }
}
```

En utilisant la fonction `parcours_infixe_tab`, écrivez une fonction `void tri_arbre_binaire(int tab[N])` qui trie un tableau `tab` de taille N grâce au principe du tri par arbre de recherche.

```
void tri_arbre_binaire(int tab[N])
{  arbre a=cree_arbre();
   int i,index=0;

   for(i=0;i<N;i++)
      insere(&a, tab[i]);
   parcours_infixe_tab(a, tab, &index);
}
```

Complexité temporelle :

- Pour une séquence de longueur N, dans le pire des cas, la complexité de la **première** étape est équivalente à celle de N insertions dans un arbre **dégénéré**, soit $O(N^2)$.
- La **deuxième** étape a une complexité en $O(N)$ quel que soit l'arbre, puisqu'il faut parcourir l'arbre **en entier**.
- Donc la complexité **totale** de la fonction de tri dans le pire des cas est en $O(N^2)$.
- Par contre, si l'arbre est **équilibré**, l'insertion est en $O(\log N)$ au lieu de $O(N)$, donc on a une complexité totale du tri en $O(N \log N)$.