

Introduction à l'informatique

Bernard Boigelot

E-mail : Bernard.Boigelot@uliege.be
WWW : <https://people.montefiore.uliege.be/boigelot/>
<https://people.montefiore.uliege.be/boigelot/cours/info/>

Chapitre 1

Les ordinateurs et les programmes

Les ordinateurs

Définition: Un **ordinateur** est une machine capable de

- résoudre des problèmes et de
- traiter des données

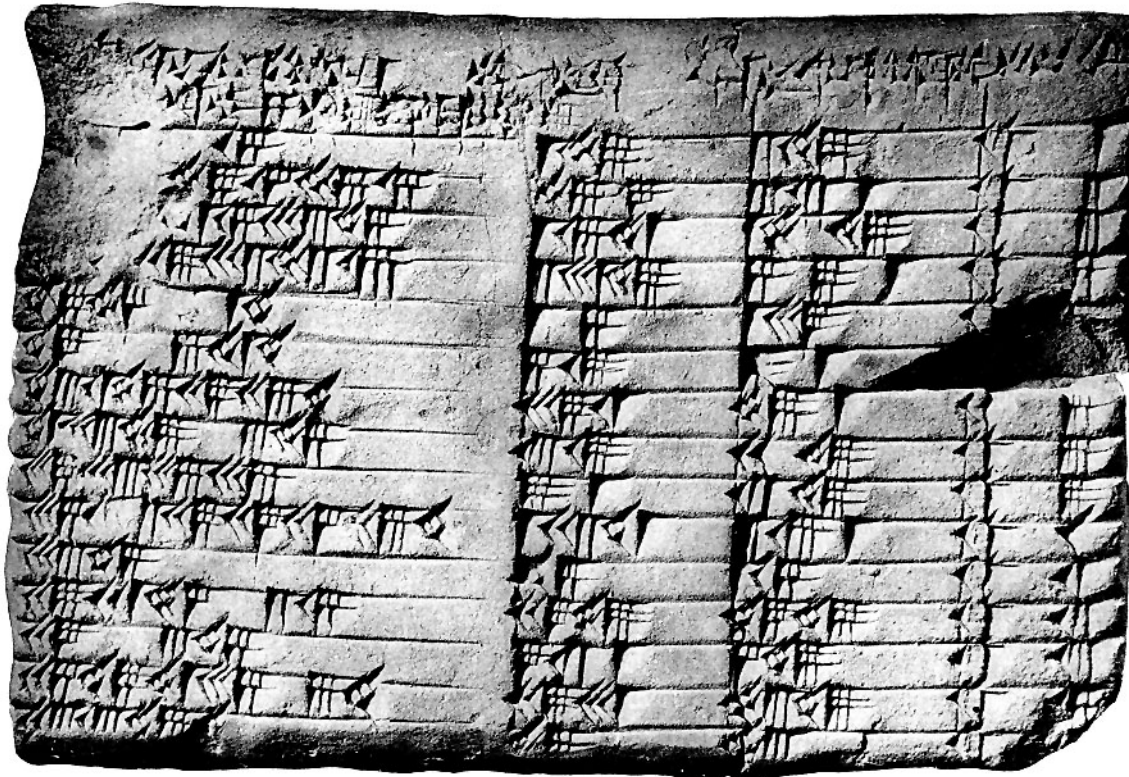
en effectuant des opérations préétablies.

Objectifs du cours:

- Apprendre à programmer un ordinateur.
- Être capable d'apporter une solution algorithmique à un problème.

Un peu d'histoire ...

≈ –2000: Notation positionnelle des nombres, **abaque** (Mésopotamie).



(Source: https://en.wikipedia.org/wiki/Plimpton_322)

≈ 825: Procédures de calcul symbolique (al-Khwārizmī, Perse).



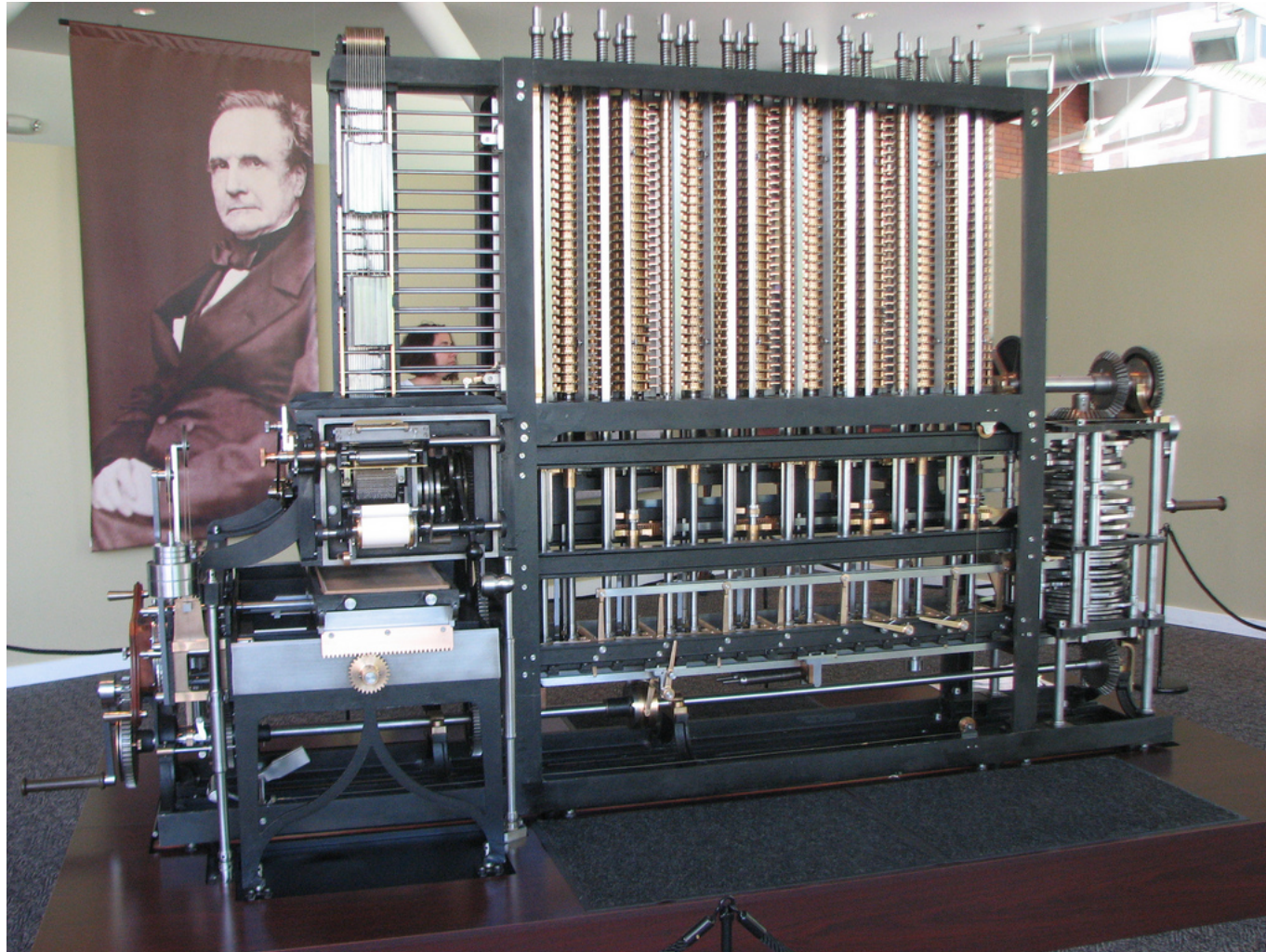
(Source: <https://en.wikipedia.org/wiki/Al-Khwarizmi>)

1625: Machine permettant d'**automatiser** le calcul d'opérations arithmétiques (Pascal).



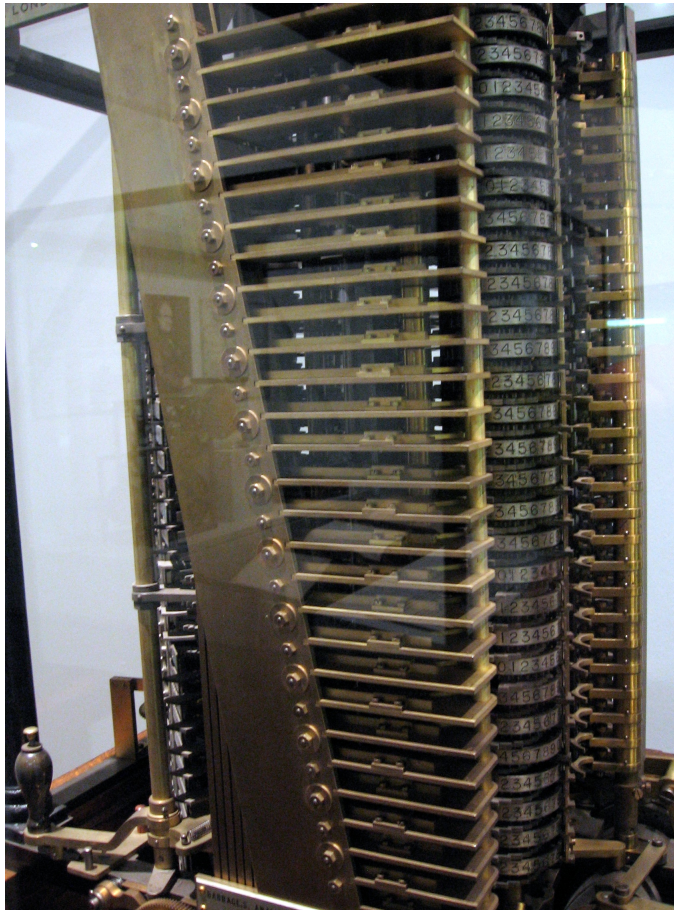
(Source: Marcin Wichary, <https://www.flickr.com/photos/mwichary/>)

1822: Machine capable d'effectuer automatiquement une combinaison particulière d'opérations arithmétiques (Babbage, *difference engine*).



(Source: Erik Pitti, <https://www.flickr.com/photos/epitti/>)

1837: Machine **programmable** (Babbage, *analytical engine*, travail inachevé).



(Source: Marcin Wichary, <https://www.flickr.com/photos/mwichary/>)

1937: (Turing) Étude théorique

- des opérations de calcul pouvant être effectuées par un **procédé automatique**, et
- de certaines **limitations fondamentales** de l'informatique.



(Source: https://en.wikipedia.org/wiki/Alan_Turing)

1941: Premier **ordinateur programmable** entièrement fonctionnel (Zuse Z3).



(Source: [https://en.wikipedia.org/wiki/Z3_\(computer\)](https://en.wikipedia.org/wiki/Z3_(computer)))

Caractéristiques:

- Programmes introduits à l'aide de **bandes perforées**.
- Mécanismes **électromécaniques** ($\approx$ 2000 relais).
- **Puissance de calcul** de 5 à 10 opérations par seconde.

1948 – 1951: Premiers ordinateurs capables de retenir et de gérer leurs propres programmes:

- Manchester *Small-Scale Experimental Machine*.
- *Electronic Delay Storage Automatic Calculator (EDSAC)*, Cambridge, Angleterre.
- *Electronic Discrete Variable Automatic Computer (EDVAC)*, USA.

Caractéristiques:

- Capable de retenir environ 1000 nombres de 15 chiffres.
- Puissance de calcul d'environ 1200 opérations par seconde.
- Construit à l'aide d'environ 18000 tubes à vide.
- 56 kW.
- 7,8 tonnes.
- Coût de fabrication de l'ordre de \$500000 (en 1951).

1947 Invention du **transistor**. (Premier ordinateur à transistors en **1953**.)

1954 Langage de programmation **Fortran**.

1958 Invention du **circuit intégré**.

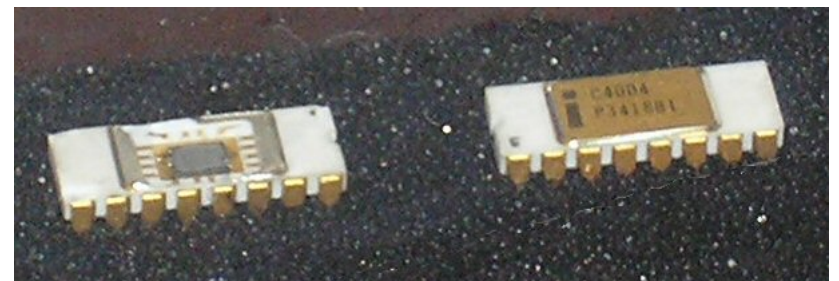
1959 Langage **COBOL**.

1969 **Système d'exploitation** UNIX.

1971 Premier **microprocesseur** (Intel 4004):

Caractéristiques:

- 2300 transistors.
- 92000 opérations par seconde.
- Prix initial de l'ordre de \$200.



(http://en.wikipedia.org/wiki/Intel_4004)

1972 Langage C.

1974 Premiers microordinateurs.

1978 Microprocesseur Intel 8086.

1981 IBM PC.

1985 Système d'exploitation Windows.

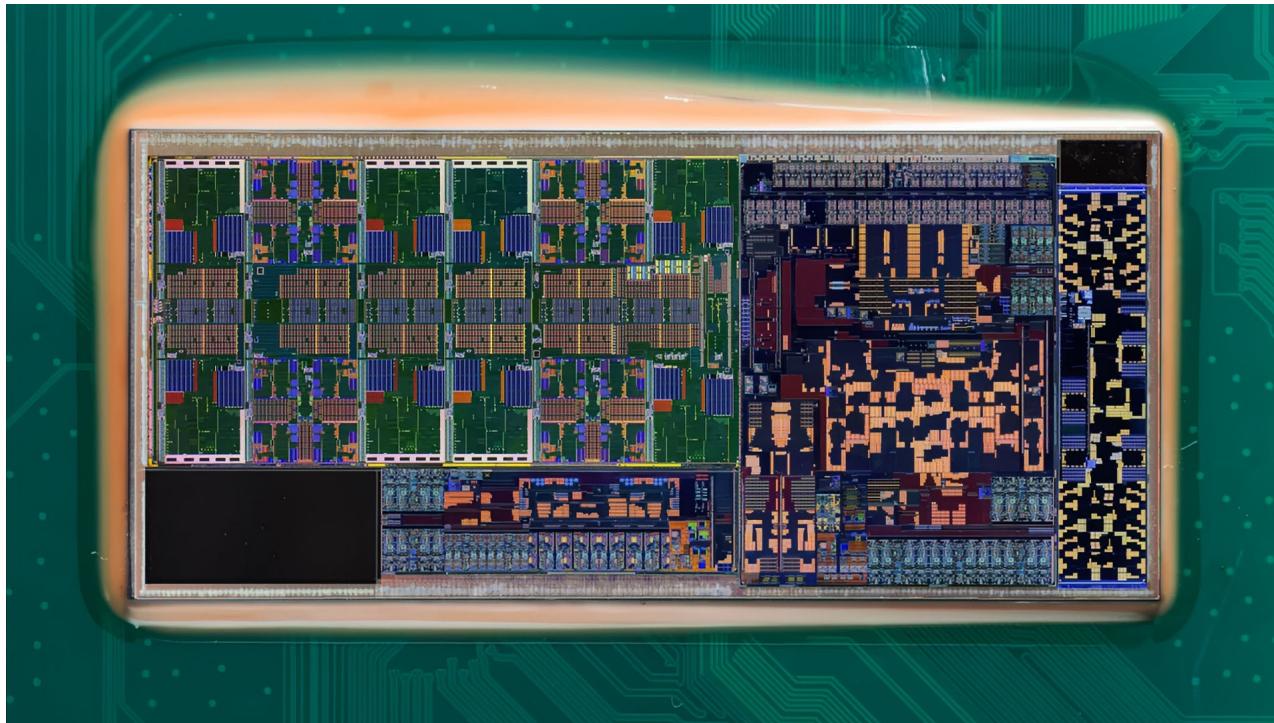
1989 *World-Wide Web*.

1995 Langage Java.

2003 Premier microprocesseur 64 bits destiné au grand public.

2007 iPhone.

2024 Microprocesseur Core Ultra 9:



(Source: Asus)

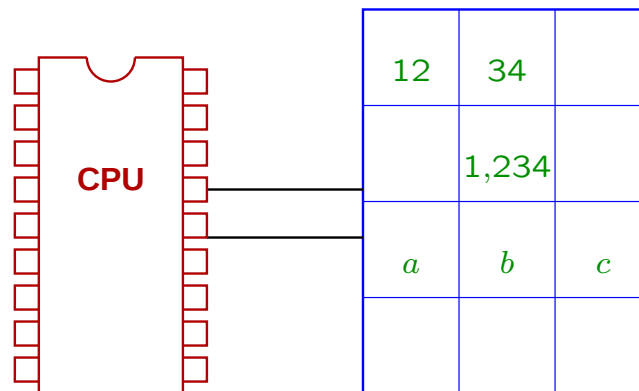
Caractéristiques:

- **Puissance de calcul** d'environ 37000 milliards d'opérations par seconde.
- 17,8 milliards de **transistors**, pour une taille de 243 mm².
- de 125 à 250 W.
- Environ \$600.

La forme abstraite d'un ordinateur

Du point de vue d'un programmeur, un ordinateur peut être vu comme une machine composée

- d'un **processeur** capable d'effectuer des opérations de traitement, et
- d'une ou plusieurs **mémoires** chargées de retenir
 - les **données** manipulées (données d'**entrée**, de **sortie**, de **travail**),
 - les **programmes** spécifiant les opérations à effectuer.



Les algorithmes

Définition: Un **algorithme** est une procédure permettant de **résoudre un problème** en décrivant précisément les **opérations à effectuer**.

Résoudre un problème implique généralement de transformer des **données d'entrée** en **données de sortie**.

- Les données d'entrée peuvent être soit **entièrement connues** avant d'exécuter l'algorithme, soit fournies **au cours de son exécution**.
- De même, les données de sortie sont soit générées en **fin d'exécution**, soit communiquées **en cours d'exécution** de l'algorithme.

Exemples de problèmes:

- Calculer la **racine carrée** d'un nombre positif, à une certaine précision près.

- Déterminer si un nombre naturel est **premier** ou non.
- Calculer le **plus court chemin** entre deux points dans un espace donné.
- Affronter un adversaire au **jeu d'échecs**.
- **Piloter un avion** en fonction de consignes spécifiées par un opérateur et de mesures transmises par des capteurs.
- Repérer la **position d'un visage** dans une image.
- **Traduire** une conversation du français vers l'anglais.
- **Simuler** l'évolution d'un phénomène physique.
- ...

Quelques problèmes algorithmiques

1. Fonction factorielle

Énoncé: Étant donné un nombre entier $n \geq 1$, calculer

$$n! = n(n-1)(n-2)\dots 1.$$

Solution: Dans ce cas, la **définition** du problème mène directement à une **solution algorithmique**: On peut obtenir la valeur de $n!$ à partir de celle de n en calculant explicitement le produit de **tous les entiers strictement positifs** qui sont **inférieurs ou égaux** à n .

2. Plus grand commun diviseur (PGCD)

Énoncé: Étant donnés deux nombres entiers $a, b \geq 1$, calculer le **plus grand entier** $\gcd(a, b)$ qui divise à la fois a et b .

Exemples:

- $\gcd(24, 18) = 6$.
- $\gcd(1000, 1000) = 1000$.
- $\gcd(1, 107) = 1$.
- $\gcd(5229827045084057754, 1570889565818035132) = 67956946$.

Première solution (recherche exhaustive, *brute-force search*):

- De l'énoncé, on peut déduire immédiatement que pour tous $a, b \geq 1$, on a:

$$1 \leq \gcd(a, b) \leq \min(a, b).$$

- Pour calculer $\gcd(a, b)$, il suffit donc d'énumérer tous les entiers dans l'intervalle $[1, \min(a, b)]$, de tester s'ils divisent à la fois a et b , et de garder le plus grand entier qui satisfait cette condition.

Inconvénient: Le nombre d'opérations à effectuer devient prohibitif pour de grandes valeurs de a et de b .

Meilleure solution (algorithme d'Euclide):

- On observe que l'on a pour tous $a, b \geq 1$:

$$\gcd(a, b) = \gcd(b, a).$$

- Si un nombre divise à la fois a et b , alors il divise également leur différence. On en déduit que pour tous $a, b \geq 1$ tels que $a \leq b$, on a:
 - $\gcd(a, b) = \gcd(a, b - a)$, et
 - $\gcd(a, b) = \gcd(a, b \bmod a)$,

en considérant que

- pour tout $n \geq 1$, $\gcd(n, 0) = n$, et
- $x \bmod y$ dénote le reste de la division de x par y .

- On obtient donc la procédure suivante:
 1. Si $a > b$, alors **permuter** a et b .
 2. Tant que $a \neq 0$, **répéter** l'opération suivante:
 - **Remplacer** (a, b) par $(b \bmod a, a)$.
 3. **Retourner** la valeur de b .

Illustration: Calcul de $\gcd(24, 18)$:

Étape	a	b
1	24	18
2	18	24
3	6	18
4	0	6

Efficacité: Il a été démontré que le **nombre d'étapes** nécessaires au calcul de $\gcd(a, b)$ par cette méthode, avec $a \leq b$, est borné par **cinq fois le nombre de chiffres** de l'écriture décimale de a [Lamé, 1844].

3. Racine carrée

Énoncé: Étant donnés un nombre réel $r \geq 0$ et une **précision souhaitée** $\varepsilon > 0$, calculer une valeur **sqrt(r)** telle que

$$|\text{sqrt}(r) - \sqrt{r}| \leq \varepsilon.$$

Remarque: Dans ce cas, l'énoncé du problème ne mène **pas directement** à une solution algorithmique . . .

Exemple de solution (algorithme de Héron):

1. Partir d'une **estimation initiale** quelconque x de $\sqrt{r}$.
2. Remplacer x par $x' = \frac{1}{2}(x + \frac{r}{x})$.
3. Tant que les **deux dernières estimations** x et x' sont telles que $|x - x'| \geq \varepsilon$, **répéter l'étape précédente**.
4. Sinon, retourner la valeur de la **dernière estimation** x' .

Illustration: Calcul de $\sqrt{2}$, $\varepsilon = 10^{-6}$:

Étape	Estimation x
1	2.000000000
2	1.500000000
3	1.416666667
4	1.41421569
5	1.41421356
6	1.41421356

L'implémentation d'un algorithme

Un algorithme peut être implémenté sous la forme d'un **programme**, qui spécifie comment il peut être **exécuté** par un processeur d'un ordinateur.

Notes:

- L'ensemble des **opérations élémentaires** composant les programmes et pouvant être exécutées par un processeur est limité.
- Cet ensemble dépend de l'**architecture matérielle** du processeur utilisé.
- Il a été établi qu'un **très petit ensemble d'opérations** suffit à implémenter **n'importe quel algorithme**.

Les langages de programmation

Afin de pouvoir s'abstraire des **détails matériels** de l'ordinateur utilisé, on a défini des **langages de programmation** facilitant l'implémentation des algorithmes.

Il existe différentes **catégories** de langages: impératifs ou non, de haut ou de bas niveau, génériques ou spécifiques à un domaine d'application, ...

Dans le cadre de ce cours, nous nous limiterons aux langages **impératifs**, qui précisent explicitement les **opérations à effectuer** à chaque étape de l'exécution des programmes.

Un programme écrit dans un langage de programmation n'est en général **pas directement exécutable** par un processeur, et doit être traduit en **code machine** afin de pouvoir être exécuté. Cette traduction peut être effectuée:

- soit **préalablement** à l'exécution du programme (**compilation**),
- soit **au cours de** cette exécution (**interprétation**).

La syntaxe et la sémantique d'un langage

Un langage de programmation est défini par:

- sa **syntaxe**, qui décrit précisément quels sont les programmes qu'il est permis d'écrire, et
- sa **sémantique**, qui donne un sens à ces programmes. On distingue:
 - la sémantique **statique** d'un programme, qui spécifie un ensemble de **contraintes** qu'un programme doit satisfaire afin d'être considéré valide, et
 - la sémantique **dynamique**, qui décrit les opérations effectuées lors de l'exécution d'un programme.

Exemple de programme: Calcul du PGCD en langage C

```
#include <stdio.h>

int main()
{
    int a, b, c;

    printf("Entrez deux entiers strictement positifs: ");
    scanf("%d %d", &a, &b);

    if (a > b)
    {
        c = a;
        a = b;
        b = c;
    }

    while (a != 0)
    {
        c = a;
        a = b % a;
        b = c;
    }

    printf("Le PGCD est égal à %d.\n", b);
}
```

Quelques explications

Remarque: Les notions évoquées ici seront étudiées en détail dans la suite du cours.

- Le programme est composé principalement d'instructions:
 - `int a, b, c`
 - `scanf("%d %d", &a, &b)`
 - `c = a`
 - ...
- La syntaxe du langage impose de terminer la plupart des instructions par un point-virgule `;`.

- Des instructions peuvent être assemblées en **séquences**:

```
c = a;  
a = b;  
b = c;
```

- Une séquence d'instructions peut être englobée dans un **bloc**, délimité par des accolades.

```
{  
    c = a;  
    a = b;  
    b = c;  
}
```

Un bloc joue le même **rôle syntaxique** qu'une **instruction individuelle** (langage **structuré**).

- Les **espaces**, tabulations et sauts de lignes sont ignorés, mais aident à rendre le programme plus lisible (**indentation** des instructions).

- Le bloc qui suit

```
int main()
```

est celui qui est exécuté en **premier lieu**.

- Les instructions d'une séquence sont exécutées **l'une après l'autre**, dans l'ordre où elles apparaissent.
- L'instruction

```
int a, b, c
```

est une **déclaration de variables**. Elle spécifie que les **identificateurs** `a`, `b`, `c` désignent, dans la suite du bloc, trois **variables** capables de retenir chacune une valeur entière.

- Une instruction de la forme

```
c = a
```

est une **affectation**. Son effet consiste à évaluer le membre de droite de l'**opérateur** `=`, et à en écrire la valeur dans la variable donnée par son membre de gauche. (Ici, la valeur de `a` est transférée dans `c`.)

- L'instruction

$a = b \% a$

est également une affectation, mais la valeur à transférer dans la variable a est ici obtenue en évaluant l'expression

$b \% a$,

qui calcule le reste de la division de b par a .

- `if` et `while` sont des mots-clés du langage, et désignent des instructions de contrôle:

- L'instruction

$\text{if } (\text{expression } E)$
 $\text{instruction } I$

évalue l'expression E , et n'exécute l'instruction I que si cette expression est vraie (choix conditionnel).

– L'instruction

```
while (expression E)  
  instruction I
```

évalue l'expression E . Ensuite,

- * si l'expression est vraie, alors l'instruction I est exécutée et le processus d'évaluation de l'expression E est répété,
- * si l'expression est fausse, alors l'exécution de l'instruction `while` se termine.

- Les expressions

$a > b$

et

$a \neq 0$

testent respectivement si a possède une valeur supérieure à b , et si la valeur de a est non nulle.

- Les instructions

`printf( ... )` et `scanf( ... )`

ne correspondent pas à des mots-clés du langage, mais constituent des **invocations de fonctions**, c'est-à-dire qu'elles conduisent à exécuter du code **externe au programme**, disponible dans une **bibliothèque** (*library*):

- `printf` permet d'**afficher** des valeurs sur la console,
- `scanf` permet de **lire des valeurs** entrées au clavier.

- La ligne

`#include <stdio.h>`

n'est pas une instruction du langage, mais une **directive de compilation** indiquant au compilateur de consulter le fichier `stdio.h`, dans lequel sont définies les modalités d'utilisation des fonctions `printf` et `scanf`.

La compilation et l'exécution du programme

Principes:

- Un **compilateur** transforme le **code source** en **code machine**.
- Ce dernier peut être **exécuté** par le processeur.

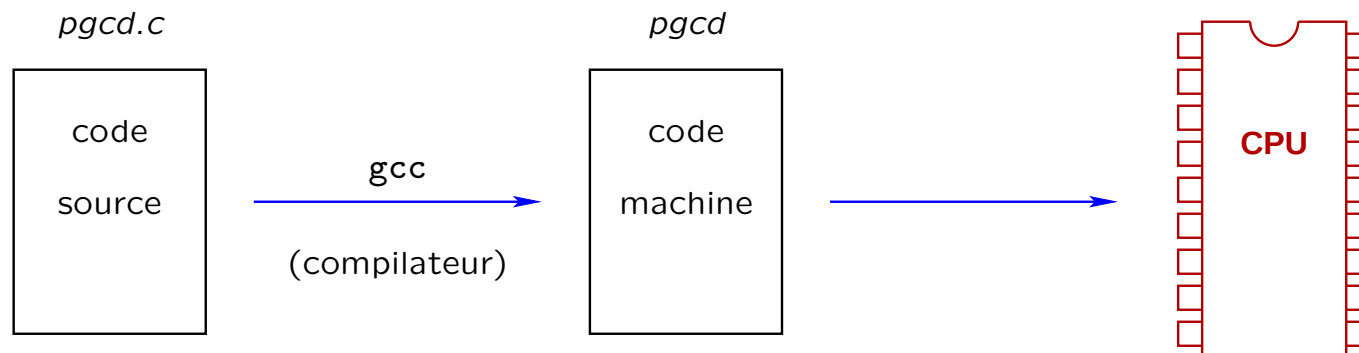
Illustration (UNIX):

```
% gcc -o pgcd pgcd.c
```

```
% ./pgcd
```

```
Entrez deux entiers strictement positifs: 19675050 8178000
```

```
Le PGCD est égal à 4350.
```



Les ressources consommées

Nous avons vu que le problème de calculer le PGCD de deux nombres possède **plusieurs solutions** algorithmiques, présentant des **coûts d'exécution** très différents.

Dans la suite du cours, nous nous intéresserons à la **complexité** des algorithmes, qui vise à estimer

- le **temps** nécessaire à l'exécution d'un algorithme, et
- l'**espace mémoire** consommé.

En général, les complexités en temps et en espace sont calculées dans le **pire des cas** (c'est-à-dire, pour les **instances les plus difficiles** du problème). Il existe cependant également des notions de **complexité moyenne**.

Les limitations de l'informatique

Il est possible de démontrer que certains problèmes

- ne possèdent **pas de solution algorithmique efficace**, quelle que soit la solution choisie.

Exemple: Déterminer si une formule de la **théorie arithmétique additive des entiers** est vraie ou non [Fischer & Rabin, 1974].

- n'admettent **aucune solution** algorithmique.

Exemples:

- Déterminer si une formule de la **théorie arithmétique additive et multiplicative des entiers** est vraie ou non [Gödel, 1931].
- Déterminer si un programme fourni en entrée **s'arrête ou non** [Turing, 1936].
- Déterminer si un **polynôme à plusieurs variables** possède ou non un **zéro entier** (dixième problème d'Hilbert) [Matiyasevich, Robinson, Davis & Putnam, 1970].

Chapitre 2

Les bases du langage C

Introduction

La **forme la plus simple** d'un programme C est la suivante:

```
int main()  
{  
    ...  
}
```

Rappels:

- Une partie de code délimitée par des accolades `{ ... }` forme un **bloc**.
- Un bloc contient des **instructions exécutables**.
- Dans la plupart des cas, un bloc peut se **substituer** à une instruction exécutable (C est un langage **structuré**).

Remarques: Nous étudierons plus tard

- la signification exacte de la déclaration `int main()`,
- l'organisation de programmes **plus complexes**.

Les variables

Dans un programme, une **variable** désigne une zone de mémoire capable de retenir une **valeur** et de lui faire subir des **opérations**.

Une variable est caractérisée par

- un **identificateur**, qui permet d'y faire référence.

En C, les identificateurs sont composés de **lettres** (a–z, A–Z, _) et de **chiffres**, et doivent commencer par une lettre. Les majuscules et les minuscules sont distinguées.

- un **type**, qui détermine
 - la **quantité de mémoire** associée à la variable,
 - la façon dont les valeurs de la variable sont **encodées** dans cette mémoire, et
 - les **opérations** permises sur cette variable.

Les types primitifs

Les **types de base** des variables sont les suivants:

<code>char</code>	Caractère, ou entier codé sur 8 bits
<code>int</code>	Entier
<code>float</code>	Nombre réel (IEEE 754 simple précision)
<code>double</code>	Nombre réel (IEEE 754 double précision)

Notes:

- La **taille** d'une variable de type `int` **dépend de l'architecture** utilisée.
Sur un PC moderne (architecture x86-64), il s'agit souvent de **32 bits**,
ce qui signifie que les valeurs possibles forment l'intervalle

$$[-2^{31}, 2^{31} - 1].$$

- Les modificateurs `signed` et `unsigned` peuvent être appliqués aux types `char` et `int` afin de les rendre respectivement **signés** ou **non signés**.

Exemples:

- `unsigned char`: Valeur comprise entre 0 et 255.
- `unsigned int`: Valeur comprise entre 0 et $2^{32} - 1$ (x86-64).
- `signed` et `unsigned` employés seuls sont équivalents à `signed int` et `unsigned int`.
- Le type `int` est signé par défaut.
- Les modificateurs `short`, `long` et `long long` peuvent être appliqués au type `int` afin d'en modifier la taille.
 - Leur signification exacte dépend de l'architecture.
 - Ils peuvent être combinés avec `signed` ou `unsigned`.
 - Lorsqu'ils sont employés, l'écriture de `int` est facultative.

Exemple: `unsigned long long`: Valeur comprise entre 0 et $2^{64} - 1$ (x86-64).

- Il existe également un type `long double`, correspondant à la précision étendue du standard IEEE 754.

La déclaration des variables

Une **déclaration** de variables est une instruction qui définit l'**identificateur** et le **type** d'une ou de plusieurs variables. En C, les variables doivent toujours être **déclarées** avant d'être **utilisées**.

Syntaxe:

```
[ const ] type identificateur [ = valeur ]  
      [ , identificateur [ = valeur ] ]  
      :  
      [ , identificateur [ = valeur ] ] ;
```

Notes:

- Dans cette description de la syntaxe, les crochets `[ ... ]` indiquent des éléments **optionnels**.
- Le mot-clé `const` permet de définir des variables dont la valeur **ne peut pas être modifiée**.

Exemples:

```
int          i = 0, j = -1, k;  
unsigned long code_barre;  
char         symbole = 'a', marqueur = '\n';  
const double avogadro = 6.02214179e23;
```

Remarque: Lors de l'écriture d'un programme, on veillera à choisir des **noms de variables** qui

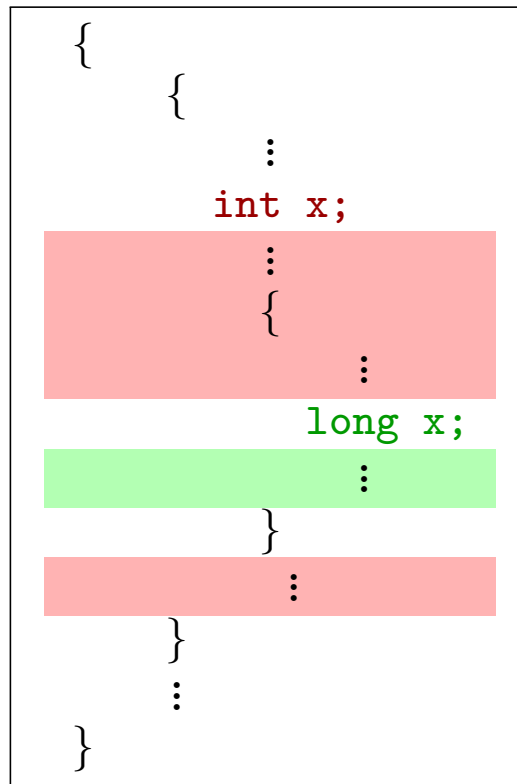
- décrivent clairement l'**usage** qui en est fait, et
- sont **cohérents**.

La portée d'une déclaration

À l'intérieur d'un bloc, une **déclaration de variable**

- est effective **immédiatement**, et jusqu'à la **fin du plus petit bloc B** contenant cette déclaration;
- **masque** les déclarations de variables de **même nom** situées dans des **blocs extérieurs** à B .

Illustration:



Note: Pour améliorer la lisibilité des programmes, il est conseillé de déclarer les variables au **début des blocs**.

(Cette contrainte n'est cependant **pas imposée** par le langage.)

Les expressions

Les **expressions** sont des formes particulières d'**instructions exécutables**, qui spécifient comment calculer une **valeur**. Une expression peut être

- une **variable**, dénotée par son identificateur. L'**évaluation** de l'expression retourne alors la valeur courante de cette variable.

Exemple: `avogadro`.

- une **constante**, ou *littéral*.

Exemples: `3.142592654`, `'a'`.

- obtenue par l'application d'**opérateurs** à d'autres expressions.

Exemple: `(4.0 / 3.0) * pi * r * r * r`.

Les opérateurs arithmétiques

Ces opérateurs sont les suivants:

+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Reste de la division (modulo)

Remarques:

- La nature de l'opération diffère selon le **type** des opérandes.

Exemple:

$$\begin{array}{lll} 2 / 3 & \longrightarrow & 0 \\ 2.0 / 3.0 & \longrightarrow & 0,666666 \dots \end{array}$$

- L'opérateur de reste de la division n'est applicable qu'à des **entiers**, et ne correspond à la notion mathématique de modulo que pour des opérandes **positifs**.

- La **priorité** des opérateurs suit les règles classiques de l'algèbre. Des **parenthèses** peuvent également être utilisées pour indiquer l'ordre d'évaluation.

Exemple: $a * a + b * b$ est équivalent à $(a * a) + (b * b)$.

- Les opérateurs $+$ et $-$ possèdent également une **forme unaire**.

Les opérateurs de comparaison

Ces opérateurs permettent de **comparer** deux valeurs:

<	Plus petit que
>	Plus grand que
<=	Plus petit ou égal à
>=	Plus grand ou égal à
==	Égal à
!=	Différent de

Notes:

Ces opérateurs retournent des valeurs **booléennes**, c'est-à-dire *vraies* ou *fausses*.

En langage C, *vrai* est représenté par une **valeur entière non nulle**, et *faux* par la **valeur entière 0**.

(Les versions récentes du langage incluent également un **type de base spécifique** pour les valeurs booléennes.)

Les opérateurs booléens

Ces opérateurs correspondent à des **opérations logiques**:

&&	Et logique
 	Ou logique
!	Négation (opérateur unaire)

Exemples:

- L'expression `c >= 'a' && c <= 'z'` est vraie si et seulement si la variable `c` contient une **lettre minuscule**.
- Les expressions `!(x >= 1000)` et `x < 1000` sont **équivalentes** (pour `x` entier).

Remarque importante: Les opérandes des opérateurs `&&` et `||` sont évalués **de gauche à droite** et en **circuit court**.

Cela signifie que l'évaluation des opérandes cesse dès que la valeur de l'expression **peut être déterminée**.

Exemple:

Lors de l'évaluation de l'expression

$$n \neq 0 \ \&\& \ m / n > 1 ,$$

le quotient m / n n'est calculé **que si n est non nulle**, ce qui évite une **division par zéro**.

Les opérateurs d'affectation

L'opérateur `=` permet d'affecter une valeur à une variable.

- Son membre de gauche contient l'identificateur de cette variable (*"valeur à gauche"*).
- L'évaluation de son membre de droite fournit la nouvelle valeur à attribuer à la variable.
- Après l'opération d'affectation, l'expression elle-même devient égale à la valeur qui a été affectée.

Exemples:

- `x = x + 2` incrémente la valeur de `x` de deux unités.
- `(a = b) < 10` recopie la valeur de `b` dans `a`, et teste ensuite si cette valeur est inférieure à 10.
- `i = j = k = 0` initialise à zéro les trois variables `i`, `j` et `k`.

Les opérateurs $\boxed{+=}$, $\boxed{-=}$, $\boxed{*=}$, $\boxed{/=}$ et $\boxed{\%=}$ permettent des raccourcis d'écriture: L'expression

$\boxed{\textit{var } \alpha = \textit{expr}}$,

avec $\alpha \in \{+, -, *, /, \%\}$, est équivalente à l'expression

$\boxed{\textit{var} = \textit{var } \alpha \textit{ expr}}$.

Exemple: Les expressions $\boxed{x += 2}$ et $\boxed{x = x + 2}$ sont équivalentes.

Les opérateurs d'incrément et de décrétement

Les opérateurs `++` et `--` sont des opérateurs **unaires**, dont l'opérande est une **valeur à gauche** (p. ex., un identificateur de variable). Ces opérateurs peuvent être placés **à droite ou à gauche** de leur opérande.

- *Si l'opérateur est placé à droite:* La variable est **incrémentée** (`++`) ou **décrémentée** (`--`) d'une unité. La valeur de l'expression correspond à celle de la variable **avant** l'opération.
- *Si l'opérateur est placé à gauche:* Idem, mais la valeur de l'expression correspond à celle de la variable **après** l'opération d'**incrément** ou de **décrément**.

Exemples: (Initialement, $x = y = 0$ dans chaque cas.)

- `x = y++` $\longrightarrow$ x garde la valeur 0 et y devient égal à 1.
- `x = --y` $\longrightarrow$ x et y deviennent égaux à -1 .

Remarques:

- À l'aide de ces opérateurs, il est possible d'écrire des expressions syntaxiquement correctes, mais dont la **sémantique n'est pas définie**.

Exemple: `x = --x + x++`

De telles constructions n'ont pas d'utilité et sont **à éviter!**

- Dans les expressions combinant ces opérateurs avec des **opérateurs booléens**, **l'évaluation en circuit court** de ces derniers influence le résultat.

Exemple: L'expression `x > 0 && --x`

- ne décrémente `x` que si cette variable est **strictement positive**.
- retourne une valeur vraie si et seulement si `x` est initialement **strictement supérieure à 1**.

L'opérateur virgule

Dans certains cas, on souhaite effectuer **plusieurs évaluations**, dans un ordre donné, au sein d'une **même expression**.

L'opérateur `,` est un opérateur **binaire** qui évalue ses opérandes de **gauche à droite**, et prend **la valeur et le type** de son opérande de droite.

Exemple: (Initialement, `x = y = 0.`)

`x = (y++, y++)` $\longrightarrow$ `x` prend la valeur 1 et `y` devient égal à 2.

Remarque: L'opérateur `,` possède une **priorité inférieure** à celle de l'opérateur `=`. Dans l'expression précédente, les **parenthèses** sont donc nécessaires.

La conversion de type

Il est parfois permis d'affecter à une variable d'un certain type t_1 une valeur v d'un **autre type** t_2 . C'est notamment le cas si t_1 et t_2 sont des types numériques.

Une **conversion** de la valeur v vers le type t_1 est alors effectuée.

Exemple:

```
double p = 3.1416;  
int x;  
  
x = p;    →    x prend la valeur 3.
```

Il existe également un opérateur permettant d'effectuer une **conversion explicite** de type (*casting*).

Syntaxe:

`(type) expr` ,

où *expr* fournit la valeur à convertir vers le type *type*.

Exemple:

```
double a = 2.0, b = 3.0, x, y;
```

```
x = a / b;    → x prend la valeur 0,666666...
```

```
y = (int) a / (int) b;    → y prend la valeur 0.
```

Les expressions conditionnelles

Une **expression** E de la forme

$cond \ ? \ expr1 \ : \ expr2$

s'évalue de la façon suivante:

- La **condition** $cond$ est d'abord évaluée en une valeur booléenne.
- **Si cette valeur est vraie**, alors $expr1$ est évaluée, et l'expression E en prend la valeur.
- **Si cette valeur est fausse**, alors $expr2$ est évaluée, et l'expression E en prend la valeur.

Exemples:

- $max = (a > b) \ ? \ a \ : \ b$
- $incr_x \ ? \ x++ \ : \ y++$

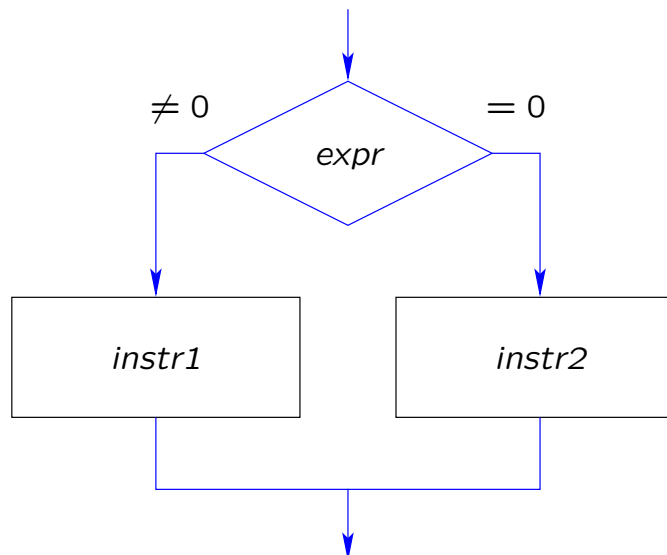
Le choix conditionnel binaire

L'instruction `if` est une instruction de **contrôle** qui permet d'orienter l'exécution du code selon l'évaluation d'une **condition booléenne**.

Syntaxe:

```
if (expr)
  instr1;
[ else
  instr2; ]
```

Sémantique:



- La *condition* *expr* est d'abord évaluée.
- Si cette condition est vraie, alors *l'instruction instr1* est exécutée.
- Sinon (et si la clause `else` est présente), *l'instruction instr2* est exécutée.

Remarques:

- Les instructions *instr1* et *instr2* peuvent prendre la forme *d'autres instructions de contrôle*, ou être remplacées par des *blocs*.
- Une clause `else` se rapporte à l'instruction `if` *la plus proche* appartenant au *même bloc*.

Il s'agit d'une source d'erreur fréquente!

Illustration:

```
if (expr1)
    if (expr2)
        {
            :
        }
else
    {
        :
    }
```

Dans ce cas, la clause `else` se rapporte à la deuxième instruction `if`, contrairement à ce que l'indentation laisse supposer ...

Version correcte:

```
if (expr1)
{
    if (expr2)
    {
        :
    }
}
else
    :
```

Exemple: Programme déterminant si une année est **bissextile** ou non:

```
#include <stdio.h>

int main()
{
    int annee, est_bissextile;

    printf("Entrez une année: ");
    scanf("%d", &annee);

    if (!(annee % 400))
        est_bissextile = 1;
    else
        if (!(annee % 100))
            est_bissextile = 0;
        else
            est_bissextile = !(annee % 4);

    printf("Cette année ");

    if (est_bissextile)
        printf("est");
    else
        printf("n'est pas");

    printf(" bissextile.\n");
}
```

Le choix conditionnel multiple

L'instruction `switch` permet d'effectuer un choix possédant un **nombre quelconque** de branches.

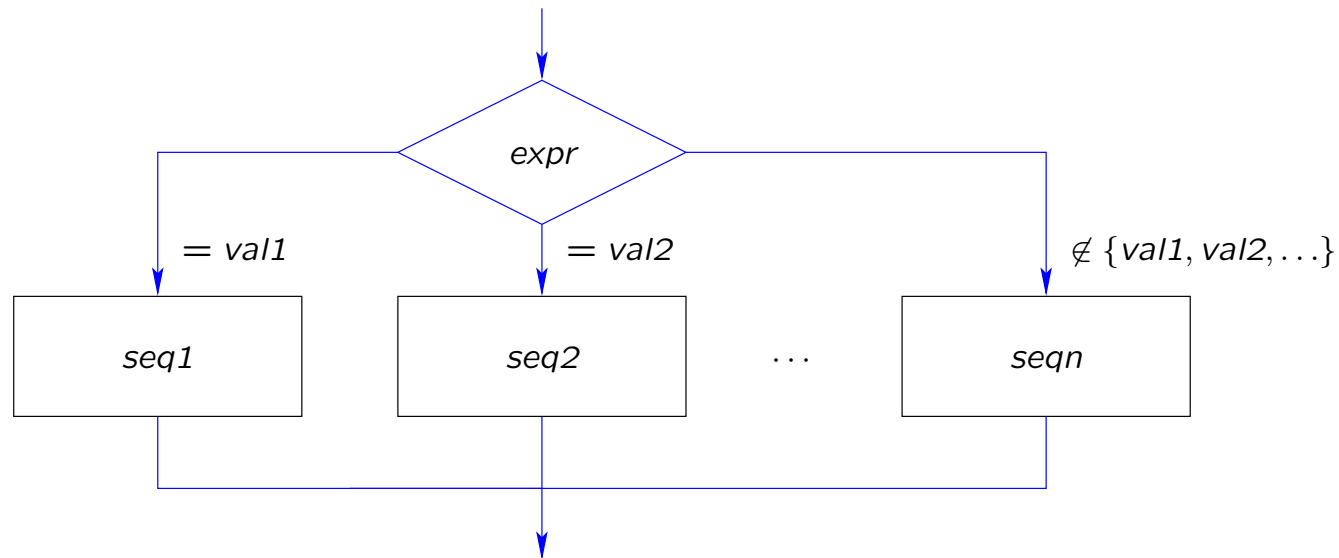
Syntaxe:

```
switch (expr)
{
    case val1:
        seq1;
        break;
    case val2:
        seq2;
        break;
    :
    [ default:
        seqn; ]
}
```

Notes:

- *expr* est une expression de type entier ou caractère.
- *val1*, *val2*, ... sont des constantes du même type que *expr*.
- *seq1*, *seq2*, ..., *seqn* sont des séquences d'instructions exécutables.

Sémantique:



- L'**expression *expr*** est d'abord évaluée.
- Si la valeur de cette expression est égale à **une des constantes** *val1*, *val2*, . . . , alors l'exécution se poursuit **après la clause** `case` correspondante.
- Sinon l'exécution continue **après la clause** `default`, si celle-ci est présente. (Dans le cas contraire, le choix multiple se termine.)
- L'exécution d'une instruction `break` **termine immédiatement** l'exécution de l'instruction de choix multiple.

Remarques:

- Les instructions `break` sont **facultatives**, mais souvent utiles.

Lorsqu'elles sont absentes, l'exécution se poursuit à la **clause suivante**, au lieu de terminer l'instruction `switch`.

- Une instruction `switch` est généralement **plus efficace** qu'un grand nombre d'instructions `if` imbriquées.

Exemple: Description d'une carte à jouer.

```
#include <stdio.h>

int main()
{
    int carte;

    printf("Entrez un nombre entre 1 et 13: ");
    scanf("%d", &carte);

    printf("Cette carte est ");

    switch (carte)
    {
        case 1:
            printf("un as");
            break;

        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
        case 10:
```

```
    printf("un %d", carte);  
    break;  
  
case 11:  
    printf("un valet");  
    break;  
  
case 12:  
    printf("une dame");  
    break;  
  
case 13:  
    printf("un roi");  
    break;  
  
default:  
    printf("invalide");  
}  
  
printf(".\n");  
}
```

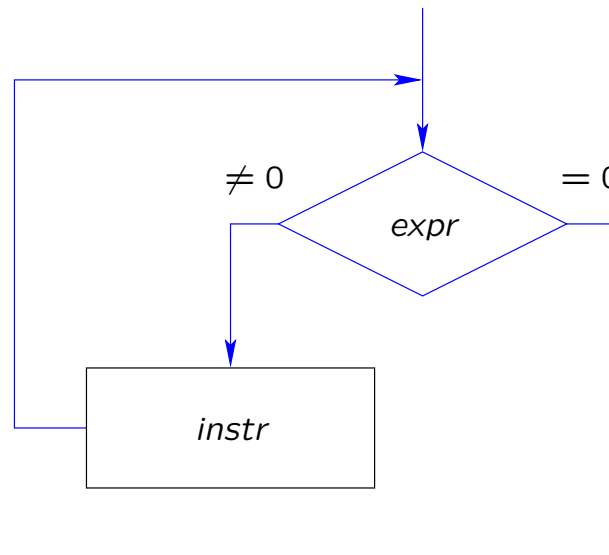
Les instructions de boucle

L'instruction `while` permet d'*itérer* une instruction tant qu'une *condition est satisfaite*. Cette condition est évaluée *avant* chaque itération.

Syntaxe:

```
while (expr)
    instr;
```

Sémantique:

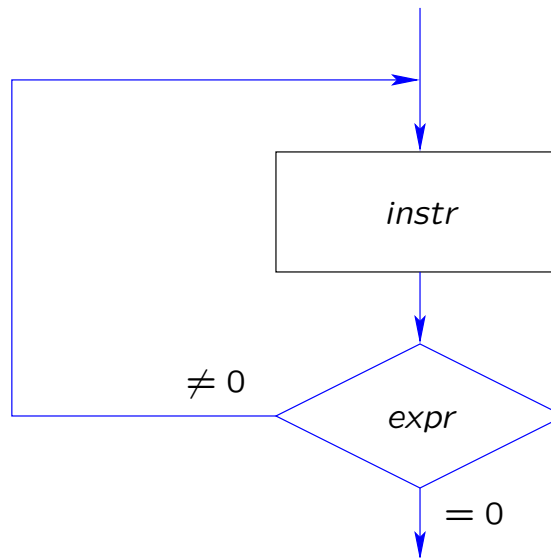


L'instruction `do ... while` est **similaire**, mais évalue la condition permettant de poursuivre l'itération **après celle-ci**.

Syntaxe:

```
do  
    instr;  
while (expr);
```

Sémantique:



L'instruction `for` permet de programmer une boucle en spécifiant explicitement les opérations à effectuer

- **avant** de rentrer dans cette boucle (*expr1*),
- afin de décider s'il faut ou non **continuer à itérer** la boucle (*expr2*),
- entre **deux itérations** (*expr3*), et
- dans le **corps** de la boucle (*instr*).

Syntaxe:

```
for (expr1; expr2; expr3)  
    instr;
```

Sémantique: Cette instruction est équivalente à

```
{  
    expr1;  
    while (expr2)  
    {  
        instr;  
        expr3;  
    }  
}
```

Notes:

- Les expressions *expr1* et *expr3* ainsi que l'instruction *instr* sont **optionnelles**.
- L'expression *expr2* peut **également être omise**, auquel cas elle est considérée comme étant **toujours vraie**.

L'instruction suivante implémente donc une **boucle infinie**:

```
for (;;) ;
```

- Dans les versions modernes du langage C, *expr1* peut être remplacée par une **déclaration de variable**.
- L'instruction *instr* peut bien sûr être remplacée par un **bloc**.

Exemple: Générateur de tables de multiplication.

```
#include <stdio.h>

int main()
{
    unsigned i, j;

    for (i = 1; i <= 9; i++)
    {
        printf("\nTable pour %u\n", i);
        printf("=====\n\n");

        for (j = 1; j <= 9; j++)
            printf("%u x %u = %u\n", j, i, i * j);
    }
}
```

Les instructions de rupture de séquence

- Nous avons vu que l'instruction `break` permet de **terminer immédiatement** l'exécution d'une instruction `switch`.

Cette instruction `break` peut **également être employée** dans le corps d'une instruction `while`, `do ... while` ou `for`, afin de **sortir anticipativement** de la boucle.

Exemple:

```
for (i = 0;; i++)  
    if (i > 100)  
        break;
```

- Dans le corps d'une instruction `while`, `do ... while` ou `for`, l'instruction `continue` permet quant à elle de **passer immédiatement** à l'itération suivante.

Remarque: Les instructions `break` et `continue` se rapportent toujours à l'instruction d'itération **la plus proche**.

Exemple:

```
for (i = 0; i < 100; i++)  
    for (j = 1; j < i; j++)  
    {  
        if (!(i % j))  
            continue;  
        :  
    }
```

Les commentaires

Un programme n'est pas uniquement destiné à être traité par un ordinateur, mais doit également être lisible pour les programmeurs, qui sont fréquemment amenés à modifier ou compléter du code existant. Il est donc souvent utile d'y inclure des commentaires.

En langage C, les commentaires sont délimités par les marqueurs `/*` et `*/`. Leur contenu est ignoré par le compilateur.

(Les versions récentes du langage permettent également d'inclure des commentaires commençant par `//` et se terminant par un saut de ligne.)

Exemple:

```
/* Position de l'objet */  
  
double x, y; // Coordonnées  
double h;    // Hauteur
```

Chapitre 3

Quelques notions d'algorithmique

Exemple de problème: Recherche de nombres parfaits

Définition: Un nombre entier positif est **parfait** s'il est égal à la **somme de ses diviseurs** (excepté lui-même).

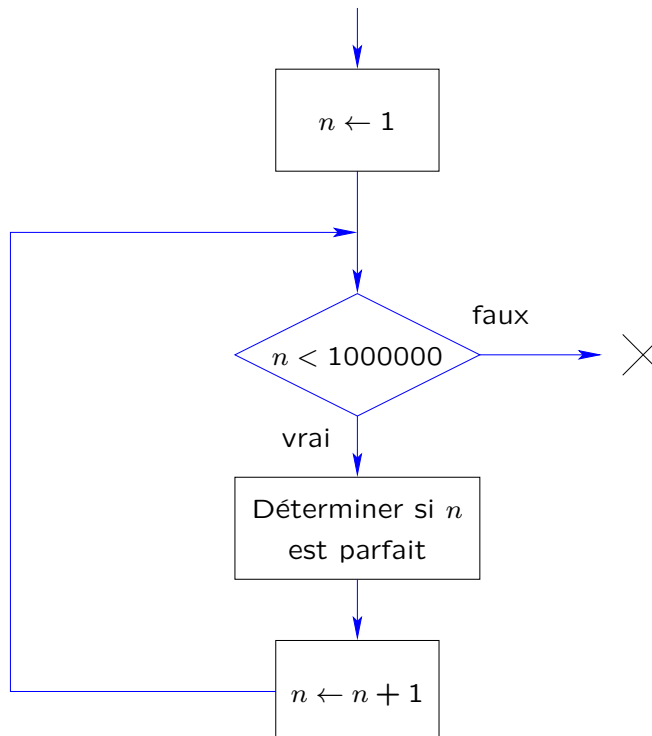
Exemple: $28 = 1 + 2 + 4 + 7 + 14$.

Énoncé du problème: Rechercher **tous les nombres parfaits** appartenant à un intervalle donné (p. ex., ceux inférieurs à 10^6).

Décomposition du problème

Approche: On peut distinguer **deux sous-problèmes** qui peuvent être résolus séparément:

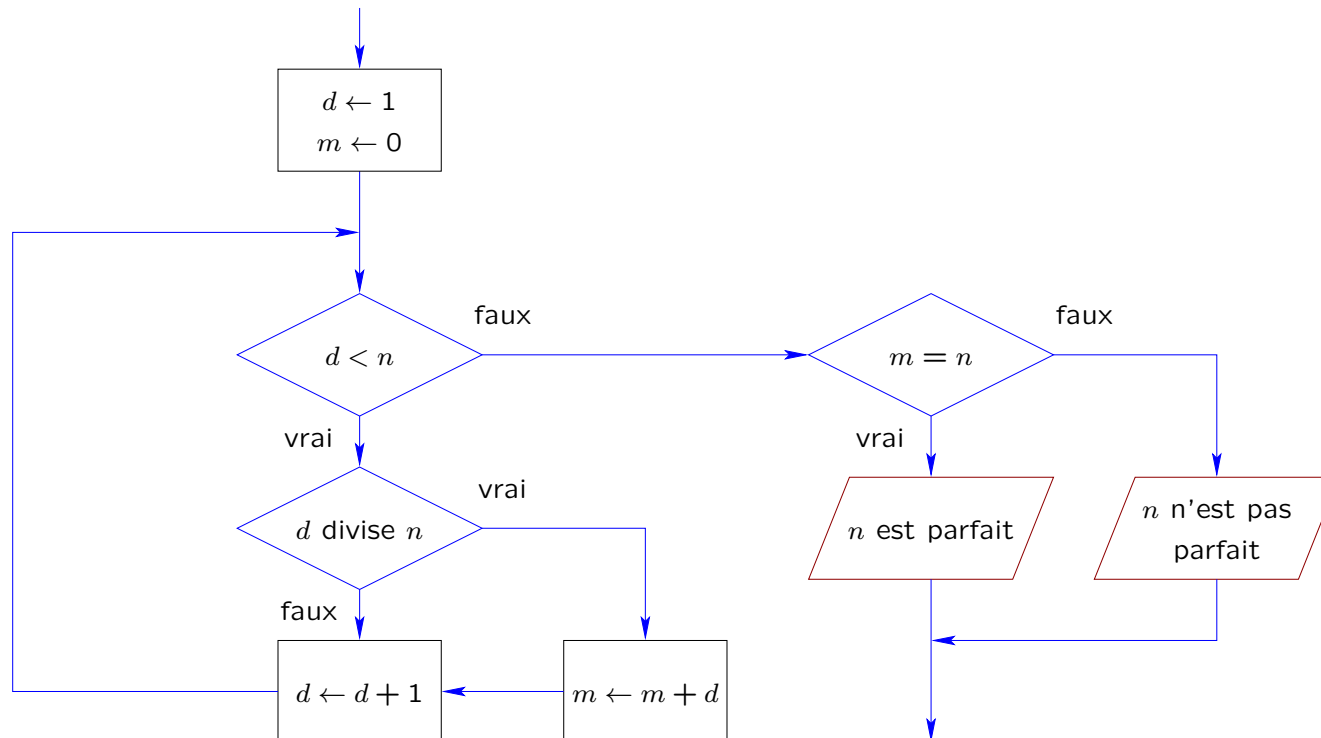
- **Énumérer** toutes les valeurs n appartenant à l'ensemble de recherche $[1, 10^6 - 1]$.
- Pour une **valeur n donnée**, déterminer si elle constitue ou non un nombre parfait.



Déterminer si un nombre n est parfait

Première solution: On peut explicitement:

- Énumérer les **diviseurs** d de n , en envisageant successivement **tous les candidats** possibles: $1, 2, \dots, n - 1$.
- Calculer la **somme** m de ces diviseurs.
- **Comparer** cette somme avec la valeur de n .



Traduction en C

```
#include <stdio.h>

int main()
{
    const unsigned n_max = 999999;

    unsigned n, m, d;

    for (n = 1; n <= n_max; n++)
    {
        for (d = 1, m = 0; d < n; d++)
            if (!(n % d))
                m += d;

        if (m == n)
            printf("%u\n", n);
    }
}
```

Améliorations possibles

- Tous les nombres entiers sont **divisibles par 1**.

On peut donc commencer **l'énumération des diviseurs** à $d = 2$, en **initialisant** m à 1.

Note: Il est alors nécessaire de **commencer la recherche** à $n = 2$, afin d'exclure le nombre 1 qui n'est pas parfait.

- Pour une valeur donnée de n , le **plus grand diviseur** à considérer est égal à $\lfloor n/2 \rfloor$.
- Lorsque la valeur de m **dépasse celle de n** , on peut arrêter l'énumération des diviseurs pour la valeur courante de n .

Traduction en C (version 2)

```
#include <stdio.h>

int main()
{
    const unsigned n_max = 999999;

    unsigned n, m, d;

    for (n = 2; n <= n_max; n++)
    {
        for (d = 2, m = 1; d <= n / 2 && m <= n; d++)
            if (!(n % d))
                m += d;

        if (m == n)
            printf("%u\n", n);
    }
}
```

Amélioration supplémentaire

Observation: Si d est un diviseur de n , alors $\frac{n}{d}$ l'est également.

On peut donc énumérer tous les diviseurs de n (exceptés 1 et n) de la façon suivante:

- Considérer toutes les valeurs de d dans l'intervalle $[2, \lfloor \sqrt{n} \rfloor]$ qui divisent n .
- Pour chacune de ces valeurs de d , considérer également le diviseur $d' = \frac{n}{d}$.

Remarque: Si n est un carré parfait, alors il faut veiller à ne pas considérer deux fois le diviseur $d = d' = \sqrt{n}$.

Traduction en C (version 3)

```
#include <stdio.h>

int main()
{
    const unsigned n_max = 999999;

    unsigned n, m, d, d2;

    for (n = 2; n <= n_max; n++)
    {
        for (d = 2, m = 1; d * d <= n && m <= n; d++)
            if (!(n % d))
            {
                m += d;
                d2 = n / d;
                if (d2 != d)
                    m += d2;
            }

        if (m == n)
            printf("%u\n", n);
    }
}
```

Comparaison des performances

Résultats expérimentaux: (Intel Core i7-9750H 2,60 GHz)

n_max	Temps d'exécution		
	version 1	version 2	version 3
999	0,95 ms	0,47 ms	58 μ s
9999	93 ms	45 ms	1,4 ms
99999	9,2 s	4,4 s	37 ms
999999	930 s	447 s	1,1 s
9999999	> 1 h	> 1 h	33 s
99999999	> 1 h	> 1 h	1061 s

Observations: On constate que les versions 1 et 2 présentent un comportement similaire, mais que la version 3 est considérablement meilleure.

La complexité en temps

Objectif: Évaluer le **temps de calcul** nécessaire à l'exécution d'un programme

- en fonction de la **taille des données d'entrée** (dans notre exemple, la valeur de `n_max`),
- pour de **grandes instances** du problème,
- si plusieurs exécutions sont possibles, dans le **cas le plus défavorable** (*worst case*),
- de façon indépendante de la **plate-forme d'exécution**.

Solution:

- On compte le **nombre d'instructions élémentaires** exécutées par le programme, le temps d'exécution individuel de chacune de ces instructions étant supposé **borné**.
- On exprime le résultat dans la **notation "grand-O"**.

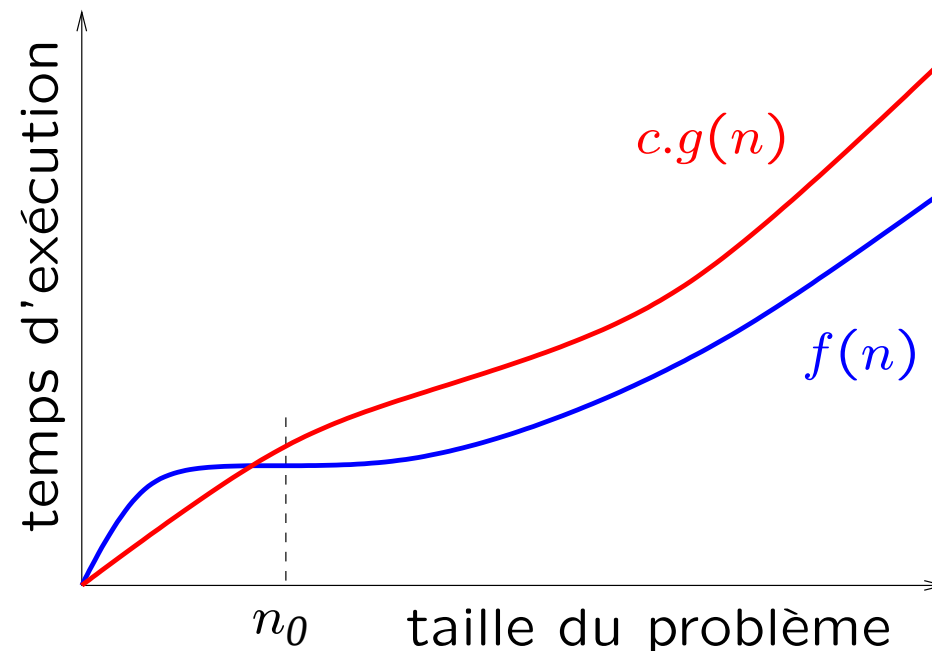
La notation asymptotique “grand-O”

Définition: Soient f et g deux fonctions $\mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. On a

$f \in O(g)$ (par abus d'écriture: $f(n) \in O(g(n))$)

ssi

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_{\geq 0} : \forall n > n_0 : f(n) \leq c.g(n).$$



Propriétés de la notation “grand-O”

- Si $f(n) \in O(g(n))$, alors pour tout $k \in \mathbb{R}_{\geq 0}$, on a $k.f(n) \in O(g(n))$.

En particulier:

$$\begin{aligned} a^{n+b} &\in O(a^n) \quad (\text{pour } a \geq 0) \\ \log_a(n) &\in O(\log_b(n)) \quad (\text{pour } a > 1 \text{ et } b > 1) \end{aligned}$$

- Si $e(n), f(n) \in O(g(n))$, alors $e(n) + f(n) \in O(g(n))$.

En particulier:

$$\sum_{i=1}^m a_i n^i \in O(n^m)$$

- Si $e(n) \in O(g(n))$ et $f(n) \in O(h(n))$, alors $e(n).f(n) \in O(g(n).h(n))$.

Ces propriétés permettent de **simplifier** les expressions de complexité.

Par exemple, la classe

$$O((3n^2 + 5n + 1)((n + 2) \log_2 n + 2n))$$

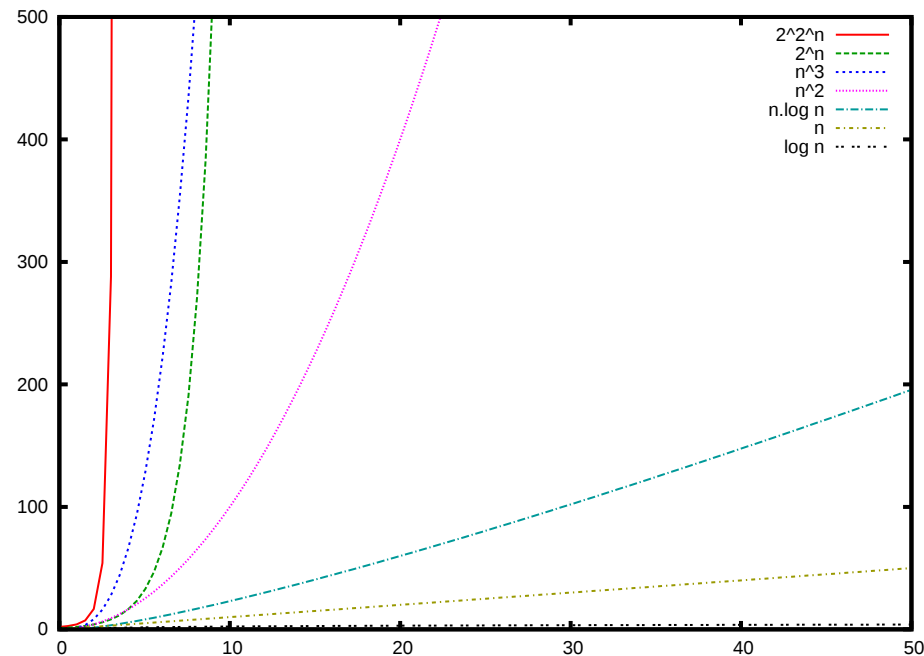
s'écrit plus simplement

$$O(n^3 \log n).$$

Une hiérarchie de classes de complexité

On a :

$$\begin{aligned} O(1) &\subset O(\log \log n) \subset O(\log n) \subset O(n) \subset O(n \log n) \\ &\subset O(n^2) \subset O(n^3) \subset O(n^4) \subset \dots \\ &\subset O(2^n) \subset O(2^{2^n}) \subset O(2^{2^{2^n}}) \subset \dots \\ &\subset O(2^{2^{2^{\dots \}} n}) \subset \dots \end{aligned}$$



Application

Question: Quelle est la **complexité en temps** du programme de recherche de nombres parfaits (version 1)?

Solution: Établissons un **inventaire** des instructions exécutées par le programme:

- Certaines instructions ne sont exécutées **qu'une seule fois**: déclarations de `n_max`, `n`, `m`, `d`, affectation `n = 1`:

$$O(1).$$

- D'autres instructions sont quant à elles exécutées **`n_max` ou `n_max + 1` fois**: `n <= n_max`, `d = 1`, `m = 0`, `n++`:

$$O(n_max).$$

- Les instructions restantes sont exécutées **au plus $1 + 2 + 3 + \dots + n_max$ fois**:

$$O\left(\frac{n_max(n_max + 1)}{2}\right) = O(n_max^2).$$

Réponse: La complexité en temps du programme vaut donc

$$O(1 + n_{\max} + n_{\max}^2) = O(n_{\max}^2).$$

En appliquant le même raisonnement aux versions améliorées du programme, on obtient:

- Version 2: $O(n_{\max}^2)$,
- Version 3: $O(n_{\max}\sqrt{n_{\max}}) = O(n_{\max}^{1,5})$.

L'analyse d'un programme

Question: Comment peut-on garantir qu'un programme est correct?

Réponse: On peut analyser certaines propriétés d'un programme en écrivant et en démontrant des triplets de la forme

$$\{P\} S \{Q\},$$

où

- S est un fragment de code exécutable,
- P est une précondition, et
- Q est une postcondition.

Principes:

- P et Q sont des **assertions** (c'est-à-dire des formules s'évaluant en une valeur booléenne), qui peuvent faire intervenir
 - la valeur courante des **variables** et des **constantes** du programme analysé,
 - les **données d'entrée et de sortie** de ce programme.
- Le triplet $\{P\} S \{Q\}$ est **valide** si et seulement si, dans tous les cas où P est vrai, alors après l'exécution de S , Q est également vrai.

Exemples:

- $\{10 \leq x \leq 20\} \quad x++; \quad \{11 \leq x \leq 21\}.$
- $\{x \in [-2^{31}, 2^{31} - 1]\} \quad \text{if } (x < 0) \ x = 0; \quad \{x \in [0, 2^{31} - 1]\}.$
- $\{V\} \quad (\text{n'importe quel code}) \quad \{V\}.$
- $\{V\} \quad \text{for } (;;); \quad \{F\}.$
- $\{F\} \quad (\text{n'importe quel code}) \quad \{ \text{n'importe quelle assertion} \}.$

- Il est souvent pratique d'utiliser des notations différentes pour la valeur d'une variable **avant** (x) et **après** (x') l'exécution de S .

Exemple:

$$\{ V \} \quad x++; \quad \left\{ \begin{array}{ll} x' = x + 1 & \text{si } x < 2^{31} - 1 \\ x' = -2^{31} & \text{si } x = 2^{31} - 1 \end{array} \right\}$$

Notes:

- Pour analyser un programme S , on cherchera à démontrer un triplet $\{P\} S \{Q\}$ dans lequel P décrit les **contraintes initiales** imposées aux données d'entrée, et Q permet de **déduire la propriété souhaitée** du programme.
- L'utilisation de triplets permet de prouver qu'un programme fournit des résultats corrects **à condition qu'il se termine** (*correction partielle*).

Une autre technique sera employée pour prouver la **terminaison** du programme.

- Pour démontrer un triplet $\{P\} S \{Q\}$ dans le cas où S **contient une boucle**, on s'aidera d'un **invariant de boucle**, qui est une assertion vraie **avant et après** chaque itération.

Les invariants de boucle

Plus précisément, pour le triplet

$$\begin{array}{c} \{P\} \\ \text{while } (C) \\ \quad S; \\ \{Q\}, \end{array}$$

un **invariant de boucle** I est une assertion pour laquelle on a

- $P \Rightarrow I$,
- $\{I \wedge C\} S \{I\}$ valide, et
- $(I \wedge \neg C) \Rightarrow Q$.

Un tel I est donc initialement vrai, reste vrai après chaque itération de la boucle, et garantit que la postcondition est satisfaite en sortie de boucle. Son existence prouve donc la **validité du triplet original**.

Illustration: Recherche de nombres parfaits (v2)

Commençons par analyser la partie du programme chargée de déterminer si un **nombre n donné** est parfait ou non.

En d'autres termes, il s'agit de démontrer le **triplet suivant**:

$$\{n \geq 2\}$$

```
for (d = 2, m = 1; d <= n / 2 && m <= n; d++)  
    if (!(n % d))  
        m += d;
```

$$\{m = n \Leftrightarrow n \text{ est un nombre parfait}\}$$

En exécutant l'**instruction d'initialisation** de la boucle for, le problème se réduit à établir

```

{ $n \geq 2, d = 2, m = 1$ }
for (;  $d \leq n / 2 \ \&\& \ m \leq n; d++$ )
    if ( $!(n \% d)$ )
         $m += d$ ;

{ $m = n \Leftrightarrow n$  est un nombre parfait},
```

que l'on peut **réécrire**

```

{ $n \geq 2, d = 2, m = 1$ }
while ( $d \leq n / 2 \ \&\& \ m \leq n$ )
{
    if ( $!(n \% d)$ )
         $m += d$ ;
     $d++$ ;
}

{ $m = n \Leftrightarrow n$  est un nombre parfait}.
```

Essayons maintenant de trouver un **invariant de boucle** I .

Cahier des charges:

- I doit être **impliqué** par la précondition

$$n \geq 2, d = 2, m = 1.$$

- I doit être vrai **avant et après chaque itération** de la boucle
(c'est-à-dire, au moment d'évaluer son *gardien* $d \leq n / 2 \ \&\& \ m \leq n$).
- À la **sortie de la boucle** `while`, la postcondition

$$\{m = n \Leftrightarrow n \text{ est un nombre parfait}\}$$

doit être une **conséquence** de I .

Solution:

$$I : n \geq 2, d \leq \left\lfloor \frac{n}{2} \right\rfloor + 1, \text{ et } m = \text{somme des diviseurs de } n \text{ inférieurs à } d.$$

Justifications:

- On a bien $n \geq 2, d = 2, m = 1 \Rightarrow I$, car quel que soit $n \geq 2$, le seul diviseur de n qui est **strictement inférieur à 2** est 1.
- I est **préservé par la boucle**, car on a

$$\{I, d \leq \left\lfloor \frac{n}{2} \right\rfloor, m \leq n\}$$

```
if (!(n % d))  
    m += d;  
d++;
```

$$\{I\}.$$

En effet, l'exécution de ce fragment de code remplace la valeur de d par $d' = d + 1$, qui satisfait bien

$$d' \leq \left\lfloor \frac{n}{2} \right\rfloor + 1.$$

Il y a deux cas d'exécution possibles:

- Si d divise n , alors

$$\{I, d \leq \left\lfloor \frac{n}{2} \right\rfloor, d \text{ divise } n\} \text{ m } += d; d++; \{I\},$$

car

$$(\text{somme des diviseurs de } n \text{ inf. à } d') = (\text{somme des diviseurs de } n \text{ inf. à } d) + d.$$

- Si d ne divise pas n , alors

$$\{I, d \leq \left\lfloor \frac{n}{2} \right\rfloor, d \text{ ne divise pas } n\} d++; \{I\},$$

car

$$(\text{somme des diviseurs de } n \text{ inf. à } d') = (\text{somme des diviseurs de } n \text{ inf. à } d).$$

- À la sortie de la boucle while, on a

$$I \text{ et } (d > \left\lfloor \frac{n}{2} \right\rfloor \text{ ou } m > n).$$

En d'autres termes, on a

- soit I et $d > \left\lfloor \frac{n}{2} \right\rfloor$. Dans ce cas, on a nécessairement

$$d = \left\lfloor \frac{n}{2} \right\rfloor + 1 \leq n,$$

et m est égal à la somme des diviseurs de n inférieurs à n . Donc, $m = n$ ssi n est un nombre parfait (postcondition).

- soit I et $m > n$. Alors, la somme des diviseurs de n (hormis n) est supérieure à n . Donc, $m \neq n$ et n n'est pas un nombre parfait (et la postcondition est satisfaite).

Résumé: On vient donc d'établir que le triplet suivant est valide:

$$\{n \geq 2\}$$

```
for (d = 2, m = 1; d <= n / 2 && m <= n; d++)  
    if (!(n % d))  
        m += d;  
    {m = n  $\Leftrightarrow$  n est un nombre parfait}
```

On en déduit que le code suivant détermine correctement si un nombre $n \geq 2$ est parfait ou non:

```
for (d = 2, m = 1; d <= n / 2 && m <= n; d++)  
    if (!(n % d))  
        m += d;  
  
if (m == n)  
    printf("%u\n", n);
```

Remarque: Le même procédé peut servir à démontrer que la **boucle principale**

```
for (n = 2; n <= n_max; n++)  
{  
    /* Traitement de n */  
}
```

explore correctement **toutes les valeurs de n** dans l'intervalle $[2, n_max]$. Il suffit en effet

- de considérer une **variable fictive** T représentant l'ensemble des valeurs **déjà traitées**,
- de poser la **précondition** $n_max \geq 2$ et $T = \{ \}$ et la **postcondition** $T = [2, n_max]$,
- d'utiliser comme **invariant de boucle** l'assertion

$$n \leq n_max + 1 \text{ et } T = [2, n - 1].$$

La terminaison d'un programme

La validité d'un triplet $\{P\} S \{Q\}$ n'implique pas que l'exécution du fragment de code S **se termine**.

Pour démontrer qu'un programme se termine, il faut prouver qu'il finit toujours par **sortir** de **toutes ses boucles**.

Un procédé simple consiste à munir chaque instruction de boucle d'un **variant de boucle** (ou fonction de terminaison), qui est une expression possédant les propriétés suivantes:

- Elle ne dépend que des variables et constantes du programme, et est évaluée **avant chaque itération** de la boucle.
- Sa valeur est un **entier naturel**, qui **décroît strictement** à chaque itération de la boucle.

Étant donné qu'il n'existe pas de **séquence infinie** d'entiers naturels strictement décroissants, la seconde propriété entraîne nécessairement la **terminaison** de la boucle considérée.

Remarques:

- Si une boucle possède un variant valide, alors elle se termine. Mais l'inverse n'est pas vrai: Il existe des boucles dont l'exécution se termine toujours et qui **n'admettent pas de variant** (sous la forme où nous l'avons défini).
- De façon plus générale, il n'existe **aucun algorithme** capable de déterminer si un programme donné s'arrête ou non [Turing, 1936].

Applications:

- Terminaison de la boucle

```
for (d = 2, m = 1; d <= n / 2 && m <= n; d++)  
    if (!(n % d))  
        m += d;
```

pour une valeur donnée de n : On peut employer le **variant de boucle**

$$v = \left\lfloor \frac{n}{2} \right\rfloor + 1 - d.$$

En effet:

- Nous avons précédemment établi que la condition $d \leq \left\lfloor \frac{n}{2} \right\rfloor + 1$ est un **invariant** de la boucle, ce qui entraîne $v \in \mathbb{N}$.
- Chaque itération de la boucle **incrémente** d et donc **décrémente** v d'une unité.

- Terminaison de la boucle:

```
for (n = 2; n <= n_max; n++)  
{  
    /* Traitement de n */  
}
```

Il suffit d'utiliser le variant de boucle

$$v = n_max + 1 - n.$$

Chapitre 4

Les fonctions et les procédures

Définition

En programmation, le terme **fonction** désigne un morceau de code qui peut être **invoqué**, ou **appelé** (c'est-à-dire que son **exécution** peut être déclenchée) à partir de plusieurs endroits d'un programme.

Principes:

- Pendant l'exécution d'une fonction, le code **appelant** (c'est-à-dire, celui qui a invoqué la fonction) est **suspendu**. Son exécution reprend dès que celle de la fonction invoquée est terminée.
- Une fonction peut posséder des **paramètres**, qui lui permettent de recevoir des **arguments** lors de son invocation.
- À la fin de son exécution, une fonction peut transmettre une **valeur de retour** au code qui l'a invoquée. Une fonction qui ne **retourne pas de valeur** est appelée une **procédure**.

La programmation des fonctions

En langage C, la **définition** d'une fonction prend une des formes suivantes:

```
type-retour id ([ type-par1 id1 [, type-par2 id2 [, ...]])  
{  
    :  
}
```

```
type-retour id (void)  
{  
    :  
}
```

Notes:

- La deuxième forme est celle d'une fonction qui ne possède **pas de paramètre**.

- *id* est un identificateur qui **nomme** la fonction, afin de pouvoir y faire référence dans d'autres parties du programme.
- *type-retour* est le type de la **valeur de retour** de la fonction. Dans le cas d'une **procédure**, on utilise le type fictif `void`.
- *type-par1*, *type-par2*, ... spécifient le type des **paramètres** successifs de la fonction.
- *id1*, *id2*, ... sont des identificateurs qui **nomment** les paramètres de la fonction, afin de pouvoir y faire référence dans le **corps** de celle-ci.
- Le bloc `{ ... }` constitue le **corps** de la fonction, et comprend les instructions qui seront exécutées lors de l'invocation de celle-ci.
- En C standard, il est interdit de définir une fonction dans le **corps d'une autre fonction**. Un programme C est donc essentiellement composé d'une **suite de définitions** de fonctions, qui peuvent être réparties dans **plusieurs fichiers source**.

- Dans le corps d'une fonction, l'instruction de contrôle

`return [ valeur ];`

termine immédiatement l'exécution de celle-ci, et retourne la valeur fournie au code appelant.

Remarque:

- Dans le cas d'une fonction dont le type de retour n'est pas void, l'instruction `return` doit être suivie d'une expression du type approprié, et une telle instruction doit toujours être exécutée avant de sortir de la fonction.
- Dans une procédure, l'instruction `return` n'est jamais accompagnée d'une valeur, et son exécution en fin de procédure est facultative.

Exemple

Fonction calculant le **plus grand commun diviseur** de deux entiers strictement positifs:

```
int pgcd(int a, int b)
{
    int c;

    if (a > b)
    {
        c = a;
        a = b;
        b = c;
    }

    while (a)
    {
        c = a;
        a = b % a;
        b = c;
    }

    return b;
}
```

Remarque: Les **paramètres** d'une fonction (dans cet exemple, a et b) peuvent être vus comme étant des **variables** dont

- la **portée** est limitée au corps de la fonction, et
- la **valeur** est initialisée à l'aide des **arguments** fournis lors de l'invocation de la fonction.

Dans le corps d'une fonction, il est permis de **modifier** la valeur des paramètres comme pour toute autre variable. Une telle modification des arguments n'a aucune influence sur le **code appelant** (*passage par valeur*).

La déclaration d'une fonction

Pour pouvoir **invoquer** une fonction à partir d'un programme, il est nécessaire que le type des **paramètres** et de la **valeur de retour** éventuelle de cette fonction soient connus.

Ces informations peuvent être fournies au compilateur:

- soit **implicitement**, en **définissant** la fonction. Les détails de celle-ci sont alors connus **à partir de** cette définition, jusqu'à la fin du fichier source courant.
- soit **explicitement**, en fournissant une **déclaration** de la fonction, aussi appelée **prototype**.

En C, un prototype prend une des deux formes suivantes, selon que la fonction admette ou non des **paramètres**:

```
type-retour id ( type-par1 [ id1 ] [, type-par2 [ id2 ] [, ...]] );
```

```
type-retour id (void);
```

Notes:

- Les identificateurs de paramètres *id1*, *id2*, ... sont **facultatifs** dans les prototypes. Ils ne doivent **pas nécessairement correspondre** à ceux figurant dans la définition de la fonction.
- L'ensemble des prototypes des fonctions définies dans un **fichier source** `source.c` sont habituellement regroupés dans un **fichier d'en-tête** (*header file*) `source.h`.

Un tel fichier peut être **incorporé** à un fichier source grâce à la **directive de pré-traitement** (*preprocessing directive*) `#include`.

Celle-ci possède deux formes:

- `#include <source.h>` pour un fichier d'en-tête `source.h` **standard**.
- `#include "source.h"` pour un fichier d'en-tête **rédigé par le programmeur**.

Remarques: Dans les programmes que nous avons étudiés jusqu'ici:

- `stdio.h` est un fichier d'en-tête standard, contenant les prototypes de plusieurs fonctions d'entrée et de sortie (*standard input and output*), dont `scanf` et `printf`.

La directive `#include <stdio.h>` permet donc au compilateur de connaître les modalités d'invocation de ces fonctions.

- La définition `int main() { ... }` est celle d'une fonction appelée `main`, qui constitue le point d'entrée d'un programme C. Cela signifie que cette fonction est invoquée dès le début de l'exécution du programme.
- Cette fonction `main` retourne une valeur entière, qui représente un code de diagnostic renvoyé par le programme à la fin de son exécution. Par convention, le code 0 correspond à une exécution sans erreur. À partir de maintenant, nous ajouterons donc par défaut une instruction `return 0;` à la fin de la fonction `main` de nos programmes.

L'invocation d'une fonction

Si une fonction f a été correctement **définie** et (si nécessaire) **déclarée**, alors elle peut être **invoquée** en évaluant une expression de la forme

$$f(expr1, expr2, \dots)$$

où $expr1, expr2, \dots$ sont des expressions dont le type correspond à celui des **paramètres** de f .

- Si le **type de retour** de f n'est **pas void**, alors cette expression devient égale à la **valeur retournée** par f à la fin de son exécution.
- Si f est déclarée **void**, alors cette expression ne **possède pas de valeur**.

Exemple

Calcul du **plus petit commun multiple** (PPCM) de deux nombres entiers strictement positifs:

```
int pgcd(int, int);

int ppcm(int a, int b)
{
    int g;

    g = pgcd(a, b);

    return (a / g) * b;
}
```

Remarque: Ce code est **compilable**. Pour qu'il puisse être **exécuté**, il est nécessaire que

- l'on compile également la **fonction pgcd**, et que
- l'on **invoque** la fonction `ppcm` à partir d'une autre partie du programme.

L'interface et l'implémentation d'une fonction

Dans un programme, une fonction possède:

- une **interface**, qui regroupe les informations permettant de l'**utiliser**:
 - son **nom**,
 - le nombre, le type, et la signification de ses **paramètres**,
 - le type et la signification de sa **valeur de retour** éventuelle,
 - une description du **travail** qu'elle effectue,
 - des **contraintes** d'utilisation éventuelles.
- une **implémentation**, formée par le détail des instructions qui composent son **corps**.

La présence de fonctions dans un programme y introduit donc de la **modularité**: En connaissant son **interface**, il est possible d'exploiter une fonction sans en connaître précisément les **détails d'implémentation**.

La récursivité

Il est permis d'écrire des fonctions qui **s'invoquent elles-mêmes**. Ces fonctions sont alors dites **récursives**.

Exemple 1:

```
int pgcd(int a, int b)
{
    int c;

    if (a > b)
        return pgcd(b, a);

    while (a)
    {
        c = a;
        a = b % a;
        b = c;
    }

    return b;
}
```

Exemple 2:

```
int pgcd(int a, int b)
{
    if (a > b)
        return pgcd(b, a);

    if (a)
        return pgcd(b % a, a);

    return b;
}
```

Note: Comme on le voit ici, le mécanisme d'invocation récursive permet parfois de **simplifier** l'écriture d'une fonction.

Cependant, chaque **invocation d'une fonction** nécessite de mémoriser:

- l'endroit où le **code appelant** devra continuer son exécution à la fin de cette invocation,
- la valeur des **arguments** fournis à la fonction,
- la valeur des variables **locales à la fonction**.

Dans le cas d'une fonction récursive, **chaque invocation** de la fonction au sein d'une invocation déjà existante nécessite d'allouer une **nouvelle zone de mémoire** pour retenir ces données. Les programmes récursifs peuvent donc être **gourmands en mémoire**.

Illustration: Fonction factorielle.

```
unsigned fact(unsigned n)
{
    if (n <= 1)
        return 1;

    return n * fact(n - 1);
}
```

On peut estimer la **complexité en espace** de cette fonction, correspondant à la **quantité de mémoire** nécessaire à son exécution. Tout comme la complexité en temps, on exprime cette complexité en espace à l'aide de la **notation "grand-O"**:

- Le calcul de $\text{fact}(n)$ pour $n \geq 1$ nécessite **n invocations imbriquées** de la fonction `fact`.

- Chacune de ces invocations nécessite l'allocation d'une **quantité identique** de mémoire.

La **complexité en espace** du calcul de $\text{fact}(n)$ est donc $O(n)$.

Remarques:

- Pour ce problème, il est facile d'écrire une **version itérative** de la fonction factorielle, possédant une complexité en espace **$O(1)$** (donc meilleure):

```
unsigned fact(unsigned n)
{
    unsigned r;

    for (r = 1; n > 1; n--)
        r *= n;

    return r;
}
```

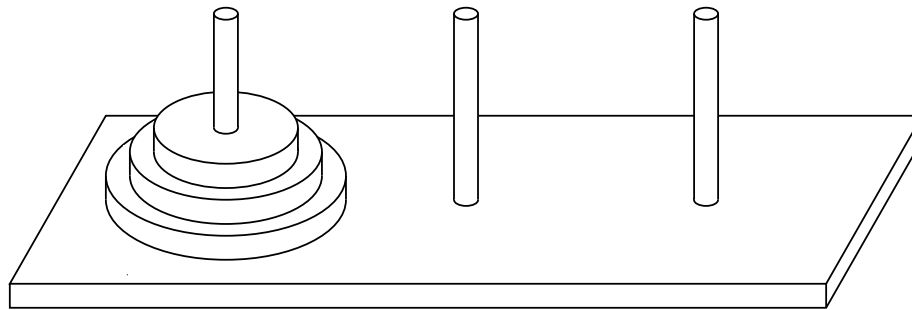
- Pour que l'exécution d'une fonction récursive se **termine**, il est nécessaire que celle-ci possède un ou plusieurs **cas de base**, libres d'appels récursifs, et qui finissent toujours par être atteints.

Pour **prouver** qu'une fonction récursive se termine, on peut employer un **variant**, tout comme pour les instructions de boucles étudiées au chapitre 3.

- Il existe des langages de programmation dits **fonctionnels**, pour lesquels
 - la récursivité est le mécanisme de base permettant de **répéter** l'exécution de certaines opérations, et
 - des mécanismes spécifiques permettent d'éliminer ou d'atténuer le **surcoût en mémoire** des appels récursifs.

Exercice: Les tours de Hanoi

On considère un jeu comprenant **trois tiges** sur lesquelles sont empilés un certain nombre de **disques** de diamètres différents.



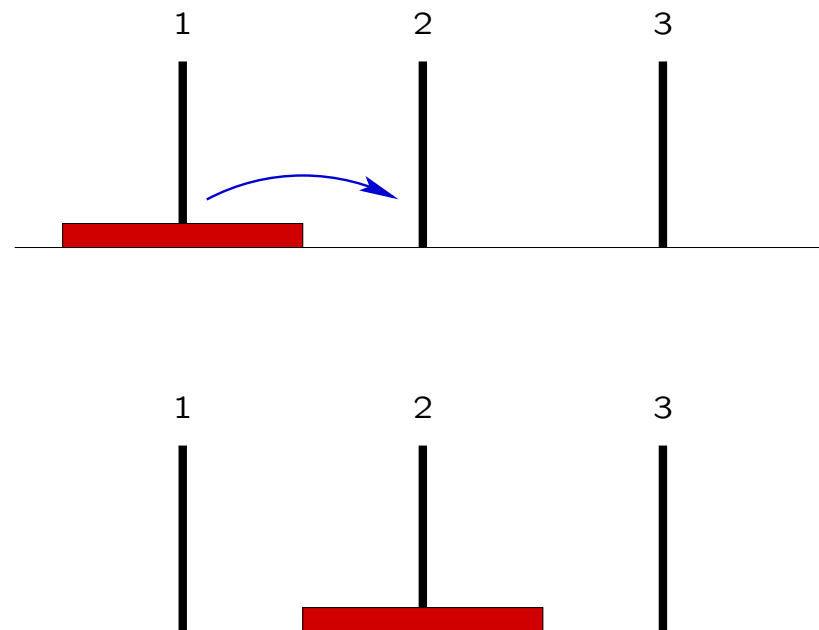
Règles du jeu:

- Initialement, tous les disques sont placés sur la **première tige**, par ordre **décroissant** de diamètre.
- L'objectif est d'amener tous les disques sur la **deuxième tige**.
- On ne peut déplacer qu'**un seul disque** à la fois.
- À chaque étape, on ne peut **poser un disque** que sur un disque de diamètre **supérieur** (ou sur le plancher).

Problème: Écrire un programme capable de **générer une séquence de mouvements** permettant d'atteindre l'objectif, pour un **nombre n** donné de disques.

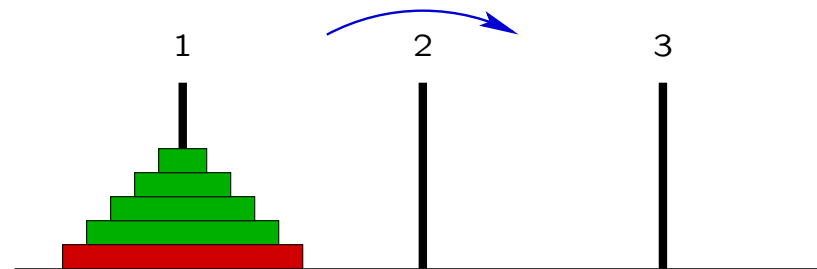
Observations:

- Si $n = 1$, la solution est **immédiate**:

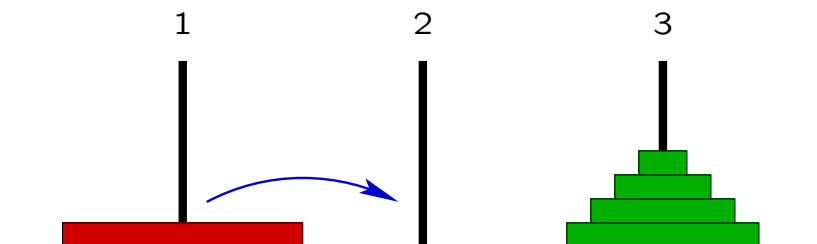


- Si $n > 1$, le problème peut se résoudre en **trois étapes**:

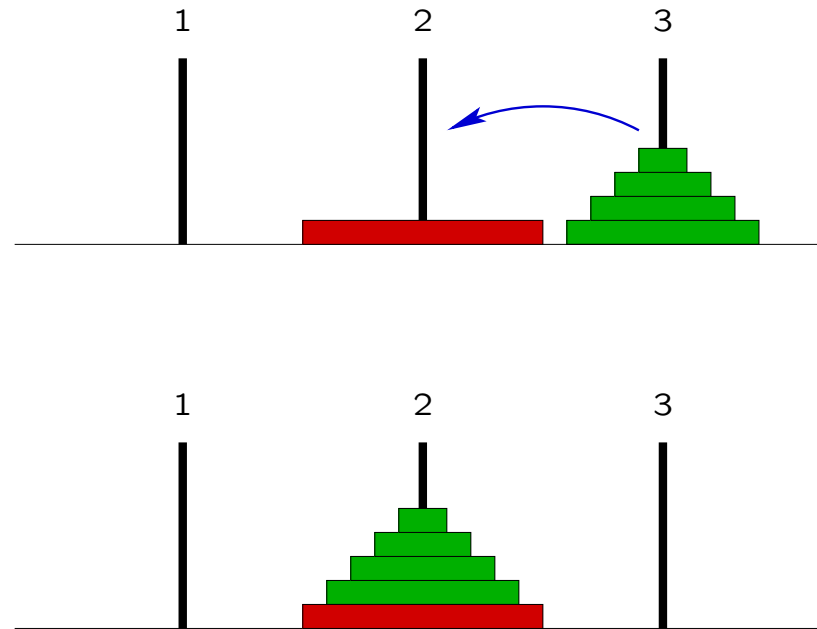
1. Déplacer $n - 1$ **disques** de la tige 1 vers la tige 3.



2. Déplacer le **plus grand disque** de la tige 1 vers la tige 2.



3. Déplacer $n - 1$ disques de la tige 3 vers la tige 2.



Remarque: Les étapes 1 et 3 peuvent se résoudre d'une façon similaire au problème global pour $n - 1$ disques. (La seule différence est qu'il faut explicitement préciser quelles sont les tiges de départ et d'arrivée.)

Cela mène donc à une solution récursive naturelle.

Solution en C

```
#include <stdio.h>

/* Déplace <n> disques de la tige <origine> vers la tige
   <destination>, en s'aidant de la tige <autre>. */

void hanoi(int n, int origine, int destination, int autre)
{
    if (n <= 1)
    {
        printf("Déplacer un disque de la tige %d vers la tige %d.\n",
               origine, destination);
        return;
    }

    hanoi(n - 1, origine, autre, destination);
    hanoi(1, origine, destination, autre);
    hanoi(n - 1, autre, destination, origine);
}
```

```
int main()
{
    int n;

    do
    {
        printf("Entrez un nombre de disques: ");
        scanf("%d", &n);
    }
    while (n < 1);

    hanoi(n, 1, 2, 3);

    return 0;
}
```

Question: Quelles sont les complexités en **temps** et en **espace** de ce programme?

Réponse:

Pour un nombre de disques $n > 1$ donné:

- le nombre d'appels de la fonction `hanoi` afin d'effectuer les étapes 1 et 3 vaut

$$2 + \dots + 2^{n-1} = 2^n - 2.$$

- le nombre d'appels de `hanoi` en vue d'effectuer l'étape 2 est égal à

$$1 + 2 + \dots + 2^{n-2} = 2^{n-1} - 1.$$

- Le nombre total d'appels de `hanoi` est donc égal à

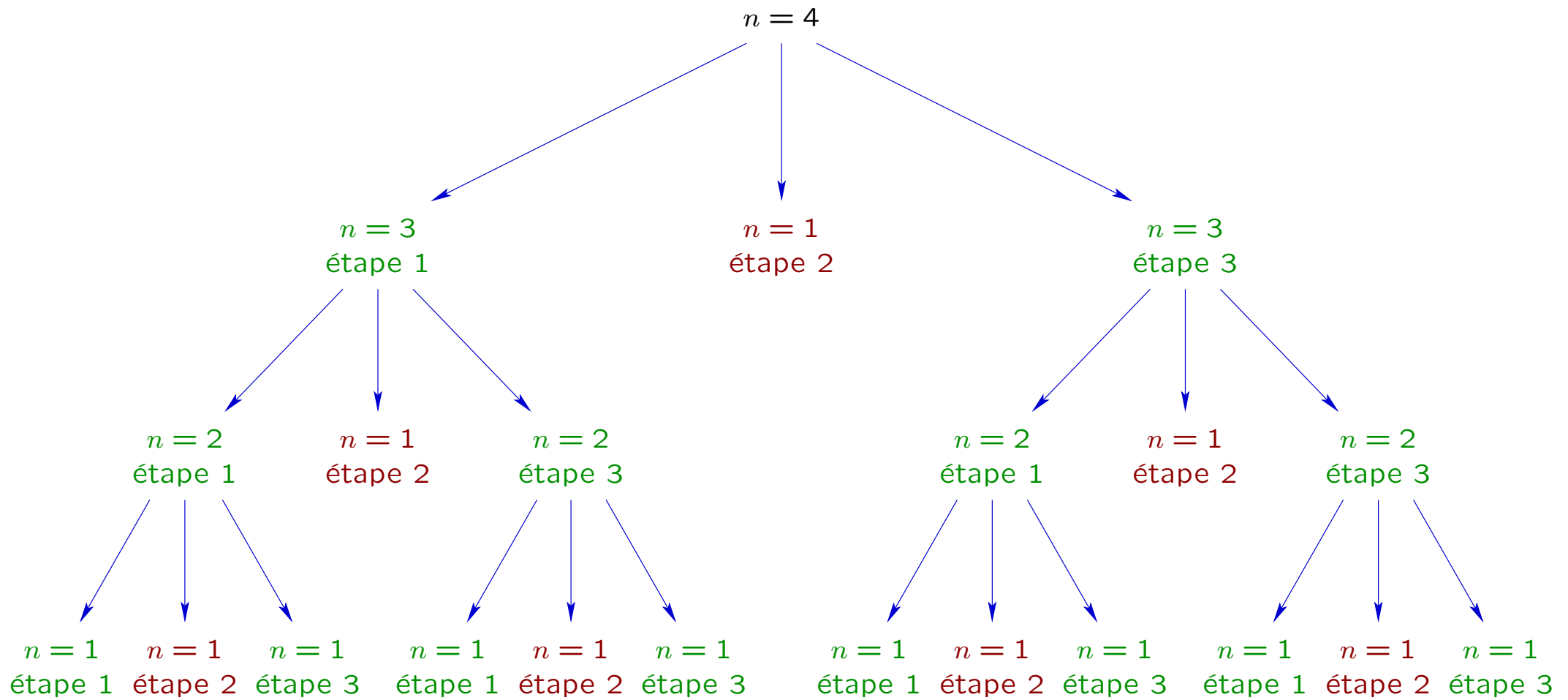
$$1 + (2^n - 2) + (2^{n-1} - 1),$$

dès lors la complexité en temps du programme est $O(2^n)$.

- La profondeur de récursion (c'est-à-dire, le nombre maximum d'invocations imbriquées) de la fonction `hanoi` est égal à n .

La complexité en espace du programme est donc $O(n)$.

Illustration ($n = 4$):



Les variables globales

Jusqu'à présent, nous n'avons rencontré que des variables déclarées à l'intérieur de blocs d'instructions exécutables (variables locales). La portée de ces variables a été étudiée au chapitre 2.

En C, il est également possible de définir des variables en dehors de toute fonction. Ces variables sont alors dites globales.

Syntaxe:

```
[ static ] [ const ] type identificateur [ = valeur ]  
                                [ , identificateur [ = valeur ] ]  
                                ⋮  
                                [ , identificateur [ = valeur ] ];
```

Notes:

- Les instructions de définition de variables globales se placent au même endroit que les définitions de fonctions.

- Si le mot-clé `static` est employé, alors la **portée** des variables est limitée au **fichier source** contenant leur définition.
- En l'absence du mot-clé `static`, la portée des variables définies s'étend à l'**ensemble du programme**, même si le code de celui-ci est réparti dans **plusieurs fichiers** source.

Remarque: Si une variable globale est définie dans un fichier source et **utilisée dans un autre**, il est nécessaire que ce dernier fichier contienne une **déclaration** de la variable, afin que le compilateur connaisse son nom et son type.

Une telle déclaration de variable(s) prend la forme suivante:

```
extern type identificateur [, identificateur [, ... ] ];
```

- L'utilisation de variables globales présente **plusieurs inconvénients**:
 - Les fonctions qui **manipulent ces variables** peuvent produire des **effets de bord**: Les opérations effectuées peuvent dépendre d'autres données que les **arguments** de la fonction, et peuvent conduire à **modifier** la valeur des variables globales.
 - Dans les programmes de grande taille, les définitions de variables globales **non marquées static** peuvent potentiellement conduire à des **conflits de noms**.

On visera donc à **limiter** dans la mesure du possible l'emploi de variables globales, afin de garder une bonne **modularité** des programmes.

- Il est également possible d'employer le mot-clé `static` pour déclarer des variables **locales à une fonction**. La valeur de telles variables est alors **préservée** d'un appel à l'autre de cette fonction (ce qui conduit également à des **effets de bord**).

- Enfin, il est aussi permis de déclarer des fonctions `static`. Comme pour les variables, cela signifie que leur **portée** est alors limitée au **fichier source** qui contient leur définition

Exemple 1: Compteur simple

Fichier compteur.c:

```
#include "compteur.h"

static unsigned compteur_valeur = 0;

void compteur_init(void)
{
    compteur_valeur = 0;
}

void compteur_plus(void)
{
    compteur_valeur++;
}

void compteur_moins(void)
{
    if (compteur_valeur)
        compteur_valeur--;
}

int compteur_est_zero(void)
{
    return !compteur_valeur;
}
```

Fichier `compteur.h`:

```
void compteur_init(void);
void compteur_plus(void);
void compteur_moins(void);
int  compteur_est_zero(void);
```

Fichier `compteur-test.c`:

```
#include <stdio.h>
#include "compteur.h"

static void affiche(void)
{
    printf(compteur_est_zero() ? "= 0\n" : "!= 0\n");
}

int main()
{
    compteur_init();
```

```
compteur_plus();  
compteur_plus();  
compteur_moins();  
  
affiche();  
  
compteur_moins();  
  
affiche();  
  
return 0;  
}
```

Exemple 2: Mesure du nombre d'appels à une fonction

Fichier nombre-appels.c:

```
#include "nombre-appels.h"

unsigned nombre_appels = 0;

void f(void)
{
    nombre_appels++;

    /* Autres opérations */
}
```

Fichier nombre-appels.h:

```
extern unsigned nombre_appels;

void f(void);
```

Fichier nombre-appels-test.c:

```
#include <stdio.h>
#include "nombre-appels.h"

int main()
{
    f();
    f();
    f();

    printf("nombre d'appels de f(): %u\n", nombre_appels);

    return 0;
}
```

Les macros

En plus des fonctions, le langage C possède un autre mécanisme permettant de **déployer un même fragment de code** à plusieurs endroits d'un programme.

La construction

```
#define nom([( id1 [, id2 [, ... ]])]) texte
```

définit une **macro**. Il s'agit d'une directive de pré-traitement réalisant une **substitution syntaxique**, appliquée **avant la compilation** proprement dite:

- Dans la suite du fichier source courant, **toutes les occurrences** de *nom* suivies du nombre approprié d'arguments (entre parenthèses), seront **remplacées** par *texte*.
- Dans le **texte de substitution** *texte*, toutes les occurrences des identificateurs *id1*, *id2*, ... (éventuels) sont remplacées par les **arguments** qui suivent *nom* dans le programme.

Exemples:

- `#define PI 3.14159265358979323846`
- `#define carre(x) ((x) * (x))`
- `#define max(a, b) ((a) > (b) ? (a) : (b))`

Notes:

- Les macros ne sont **pas des fonctions**! En particulier, leur comportement diffère lorsque l'évaluation de leurs arguments provoque un **effet de bord**.

Illustration: `carre(n++)` se substitue en `((n++) * (n++))`, dont l'évaluation incrémente `n` **deux fois**.

- La **priorité des opérateurs** peut conduire à des erreurs difficiles à déceler.

Illustration: Si l'on avait écrit

```
#define carre(x) x * x ,
```

alors `carre(a + b)` se transformerait en `a + b * a + b`, qui ne correspond pas à l'opération souhaitée.

L'emploi de **parenthèses** dans le texte des macros est donc vivement conseillé!

- Il est possible de définir des macros possédant un texte de substitution qui s'étend sur **plusieurs lignes** du programme, en terminant chaque ligne intermédiaire par le **marqueur de continuation** `\`.

Chapitre 5

Les tableaux et les chaînes de caractères

Les vecteurs

Principes: Un **vecteur**, ou **tableau unidimensionnel**, est une **collection**

$$[x_0 \ x_1 \ \cdots \ x_{n-1}]$$

de n variables x_i , avec $n > 0$, dont chacune

- possède le **même type**, et
- est accessible sur base de son **indice** $i \in [0, n - 1]$.

En langage C, une **définition de vecteur(s)** prend la forme suivante:

```
[ static ] [ const ] type id[taille] [ = { val1, val2, ... } ]  
[ , id[taille] [ = { val1, val2, ... } ] ]  
⋮  
[ , id[taille] [ = { val1, val2, ... } ] ];
```

Notes:

- Une telle définition se place au **même endroit** que les autres définitions de variables.
- Les règles de **portée** et la signification des mots-clés `static` et `const` sont inchangées.
- *type* désigne ici le type de **chaque élément** d'un vecteur.
- *taille* spécifie le **nombre d'éléments** alloués à un vecteur. Ces éléments sont toujours indicés consécutivement et **à partir de 0**.
- Les constructions $\boxed{= \{ val1, val2, \dots \}}$ sont facultatives, et permettent d'attribuer des **valeurs initiales** aux éléments d'un vecteur. Lorsqu'elles sont présentes, on peut **omettre la taille** du vecteur dans sa définition.
- On peut **accéder à un vecteur** via des expressions de la forme $\boxed{id[expr]}$, représentant **l'élément du vecteur *id*** dont l'**indice** est donné par la valeur de *expr*.

Attention: C'est au **programmeur** de garantir que la valeur de *expr* correspond toujours à un **indice valide**!

Application: *Crible d'Ératosthène*

Objectif: Déterminer l'ensemble des **nombre premiers** (c'est-à-dire, ceux possédant exactement **deux diviseurs**) jusqu'à une borne n donnée (par exemple, $n = 10^6$).

Algorithme:

1. On considère un ensemble P contenant initialement **tous les nombres** dans l'intervalle $[2, n]$.
2. Le **plus petit** nombre p contenu dans P (initialement, 2) est un nombre premier. On peut donc **afficher** p , et ensuite **enlever de** P ses multiples $p, 2p, 3p, 4p, \dots$.
3. S'il reste des nombres dans P , on recommence l'étape 2.

Amélioration: À l'étape 2, les multiples $2p, 3p, \dots, (p-1)p$ de p ont **déjà été éliminés** lors des itérations précédentes. On peut donc se contenter d'enlever de P , outre p , uniquement ses multiples $p^2, (p+1)p, (p+2)p, \dots$

Implémentation:

- On peut représenter l'ensemble P par un **vecteur de booléens** B : Pour tout $i \in [2, n]$, on aura $B[i] = 0$ si $i \notin P$, et $B[i] = 1$ si $i \in P$.
- Pour rechercher le **plus petit élément** de P , on parcourt B jusqu'à trouver une valeur égale à 1.

Remarque: Les valeurs de p considérées à l'étape 2 de l'algorithme sont **croissantes**. On peut donc à cette étape

- commencer le parcours de B **juste après** la valeur de p obtenue à l'itération précédente (ou en $p = 2$ pour la première itération), et
- se **dispenser de retirer** de B les nombres premiers p qui ont déjà été détectés.

Traduction en C:

```
#include <stdio.h>

#define N_MAX 999999

static unsigned char b[N_MAX + 1];

int main()
{
    unsigned i, p;

    for (i = 2; i <= N_MAX; i++)
        b[i] = 1;

    for (p = 2; p <= N_MAX; p++)
    {
        if (!b[p])
            continue;

        printf("%u\n", p);

        for (i = p; (unsigned long long) i * p <= N_MAX; i++)
            b[i * p] = 0;
    }

    return 0;
}
```

Remarques:

- Les deux premiers éléments du vecteur `b` sont inutilisés.

Il aurait été possible de représenter l'appartenance d'un indice i à l'ensemble P par le contenu de `B[i - 2]`, mais cela aurait légèrement nui à la lisibilité du code, tout en apportant un gain en espace peu significatif.

- Le type `unsigned char` choisi pour les éléments du vecteur est un bon compromis entre l'espace mémoire occupé et la facilité d'accès à ces éléments.
- Une conversion de type est nécessaire pour évaluer $i * p \leq N_MAX$ sans provoquer de dépassement arithmétique, si les entiers sont représentés sur 32 bits.

Les fonctions sur les tableaux

Un vecteur peut être passé **en argument** à une fonction. Pour **déclarer** le paramètre correspondant, on utilise alors la forme

`type id[]`

,

où *type* est le **type d'un élément du vecteur**, et *id* l'**identificateur représentant ce vecteur** dans le corps de la fonction.

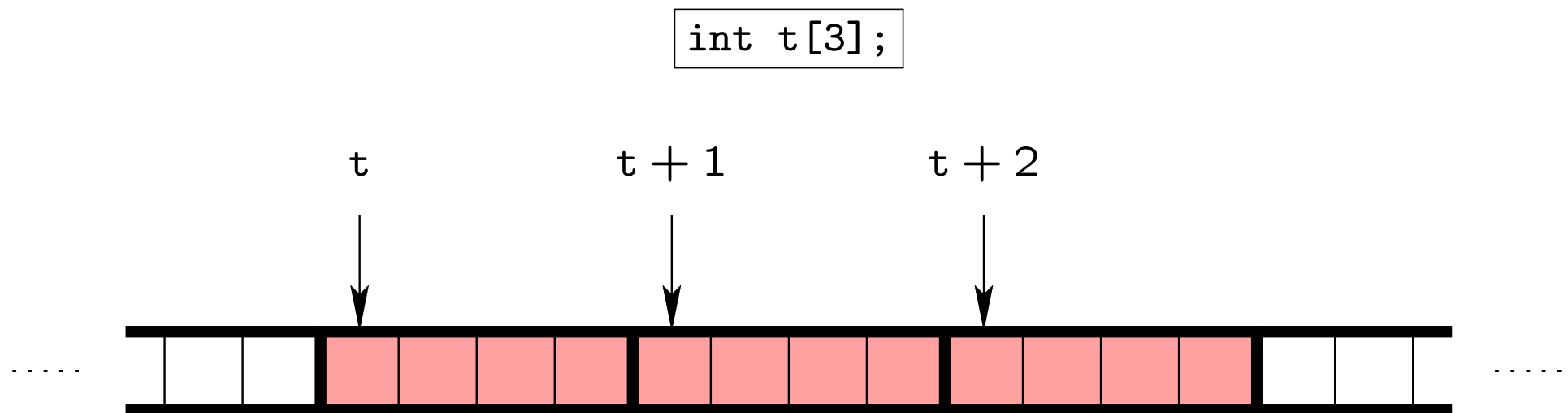
Notes:

- Lorsqu'un tableau est passé en argument à une fonction, la **seule donnée** effectivement transmise à celle-ci est un **pointeur** vers ce tableau (c'est-à-dire, l'**endroit** où se trouve ce tableau dans la mémoire de l'ordinateur).

Par conséquent, les **modifications** apportées par la fonction aux éléments du tableau ont un effet **global**.

- En C, les éléments d'un tableau sont toujours placés dans des **emplacements consécutifs** de la mémoire. Un pointeur vers un tableau pointe en fait vers son **premier élément**.
- Si p est un pointeur vers un tableau et i une valeur entière, l'expression $p + i$ fournit un pointeur vers le **$(i + 1)$ -ème** élément du tableau.

Illustration: (en supposant les entiers codés sur 32 bits)



Exemple:

Fonction **permutant deux éléments consécutifs** d'un vecteur d'entiers:

```
void permute_consecutifs(int t[])
{
    int aux;

    aux = t[0];
    t[0] = t[1];
    t[1] = aux;
}
```

Exemple d'utilisation:

```
int v[100];

:

permute_consecutifs(v + 10);    /* Permute les 11ème et 12ème éléments du vecteur. */
```

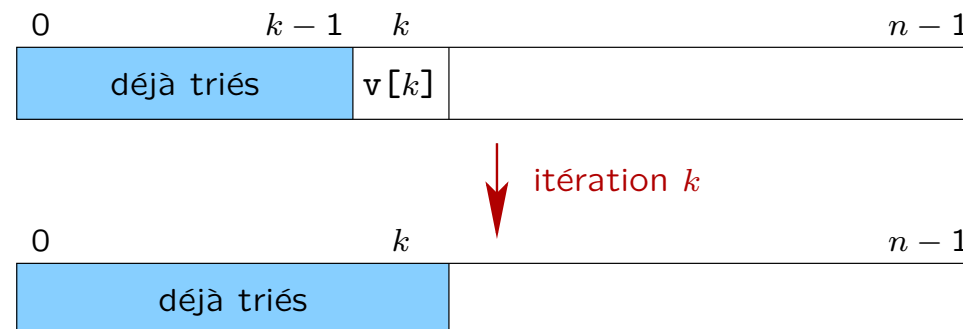
Le tri d'un vecteur

Problème: Comment peut-on **réordonner** les éléments d'un vecteur donné, de façon à les disposer par **ordre croissant** de valeur?

Première solution (tri par insertion, *insertion sort*):

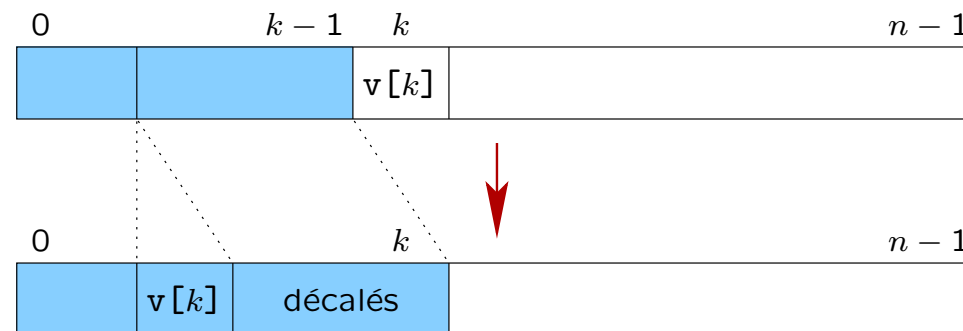
Pour trier un vecteur v de taille n :

- On effectue $n - 1$ **itérations** successivement chargées de trier les **préfixes** de taille 2, 3, 4, $\dots$, n du vecteur.
- Juste avant l'itération k , pour $k \in [1, n - 1]$, les k **premiers éléments du vecteur** sont déjà triés. Le but de cette itération est d'y **insérer** la valeur du $(k + 1)$ -ème élément.



Pour effectuer cette insertion:

1. On recopie la valeur de $v[k]$ dans une **variable auxiliaire** a .
2. On balaie **de la droite vers la gauche** les éléments du vecteur $v[0 : k - 1]$ qui sont **supérieurs à a** , et on les **décale** d'une position vers la droite.
3. On **insère** la valeur sauvegardée dans a immédiatement à gauche du dernier élément décalé.



Traduction en C:

```
/* Tri du vecteur v de taille n, par ordre croissant */  
  
void tri_insertion(int v[], int n)  
{  
    int k, i, a;  
  
    for (k = 1; k < n; k++)  
    {  
        a = v[k];  
        for (i = k - 1; i >= 0 && v[i] > a; i--)  
            v[i + 1] = v[i];  
  
        v[i + 1] = a;  
    }  
}
```

Complexité:

- En temps: $O(n^2)$.
- En espace: $O(n)$ en comptant les données d'entrée, $O(1)$ sinon.

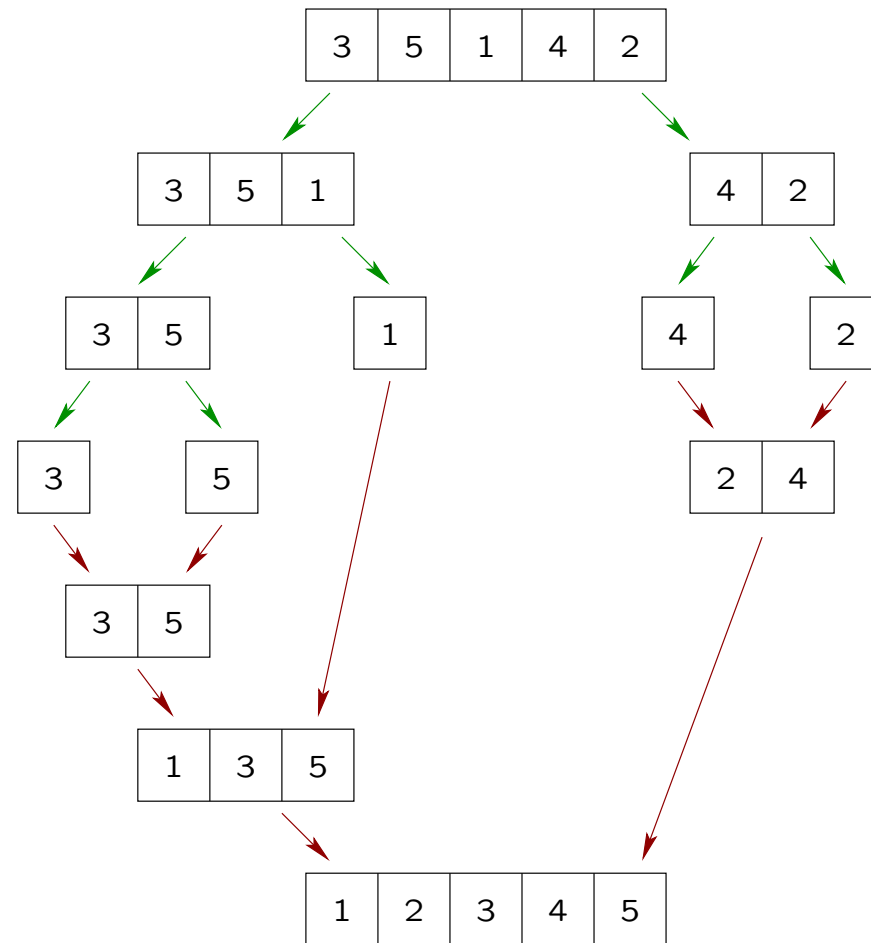
Meilleure solution (tri par fusion, *merge sort*)

Principe: On procède **récurivement**. Pour trier un vecteur de taille n :

- Si $n = 1$, il n'y a **rien à faire**.
- Si $n > 1$:
 1. On **divise** le vecteur en deux sous-vecteurs de tailles égales (ou **égales à une unité près** dans le cas d'un nombre impair d'éléments).
 2. On **trie récurivement** chaque sous-vecteur.
 3. On **fusionne** les deux sous-vecteurs triés.

L'opération de **fusion** de deux sous-vecteurs peut s'effectuer en retirant un par un leurs éléments à partir du **premier**, en les **comparant**, et en les plaçant dans un **nouveau vecteur** dans l'ordre approprié.

Illustration:



Implémentation

Problème: L'opération de **fusion** nécessite un **espace mémoire auxiliaire** pour y placer son résultat.

Solution simple: Avant de fusionner deux sous-vecteurs, on les **recopie** vers un espace auxiliaire préalloué. Ensuite, l'opération de fusion place son résultat à l'**emplacement initial** des deux sous-vecteurs.

Remarque: Il est possible de développer des solutions plus élaborées qui

- utilisent **alternativement** deux espaces mémoire distincts (en d'autres termes, la fusion de deux sous-vecteurs extraits d'**un de ces espaces** produit son résultat dans **l'autre espace**), ou bien qui
- ne nécessitent pas d'**espace de travail** supplémentaire.

Traduction en C:

```
/*
   Fusionne les vecteurs v1[a1:b1] et v2[a2:b2], supposés triés,
   et place le résultat dans le vecteur v3[], supposé distinct
   de v1[] et v2[] ainsi que de taille suffisante
*/

static void fusion(const int v1[], unsigned a1, unsigned b1,
                  const int v2[], unsigned a2, unsigned b2,
                  int v3[])
{
    unsigned i1, i2, i3;

    for (i1 = a1, i2 = a2, i3 = 0; i1 <= b1 || i2 <= b2;)
        if (i1 > b1)
            v3[i3++] = v2[i2++];
        else
            if (i2 > b2)
                v3[i3++] = v1[i1++];
            else
                v3[i3++] = v1[i1] < v2[i2] ? v1[i1++] : v2[i2++];
}
```

```

/*
    Trie le vecteur v[a:b], par ordre croissant, en utilisant
    le vecteur v_aux[a:b] comme espace de travail auxiliaire
*/

static void tri_sousvecteur(int v[], unsigned a, unsigned b, int v_aux[])
{
    unsigned n, c, i;

    n = b - a + 1;
    if (n < 2)
        return;

    c = a + (n + 1) / 2;
    tri_sousvecteur(v, a, c - 1, v_aux);
    tri_sousvecteur(v, c, b, v_aux);

    for (i = a; i <= b; i++)
        v_aux[i] = v[i];

    fusion(v_aux, a, c - 1,
          v_aux, c, b,
          v + a);
}

```

```
/*  
    Trie le vecteur v de taille n > 0, par ordre croissant, en utilisant  
    le vecteur v_aux, de même taille, comme espace de travail auxiliaire  
*/  
  
void tri_fusion(int v[], unsigned n, int v_aux[])  
{  
    tri_sousvecteur(v, 0, n - 1, v_aux);  
}
```

Complexité en temps:

- Le coût d'une opération de **copie** ou de **fusion** est $O(n)$, où n représente la taille du vecteur copié ou fusionné.
- Pour trier un vecteur de taille $n = 2^k$, avec $k \in \mathbb{N}$, on doit copier et/ou fusionner:
 - 2 vecteurs de taille $\frac{n}{2}$,
 - 4 vecteurs de taille $\frac{n}{4}$,
 - 8 vecteurs de taille $\frac{n}{8}$,
 - $\vdots$
 - n vecteurs de taille 1.
- Le coût total vaut donc $O(kn) = O(n \log n)$.

Complexité en espace: $O(n + \log n) = O(n)$.

Les chaînes de caractères

De nombreux programmes sont amenés à devoir manipuler des **données textuelles**. Celles-ci sont représentées par des **chaînes de caractères** (*strings*), qui sont des **séquences de symboles** extraits d'un alphabet fixé.

En langage C, les chaînes de caractères sont équivalentes à des **vecteurs** dont chaque élément est de type **char**.

Principes:

- La **fin d'une chaîne** est repérée par un caractère spécial '**\0**', dont la valeur numérique est **nulle**.

Il est important de tenir compte de la présence de ce **caractère de terminaison** lors de l'allocation d'une chaîne. Ainsi, la définition

```
char message[10];
```

ne permet de placer dans `message` que des chaînes de longueur inférieure ou égale à 9.

- Une chaîne de caractères peut être initialisée lors de sa définition grâce à une construction de la forme

```
char id[] = "texte";
```

Attention: Cette construction est une forme particulière de **définition de variable**. En dehors d'une telle définition, il n'est pas permis d'utiliser l'**opérateur d'affectation** `=` pour modifier le contenu d'une chaîne de caractères (ou, plus généralement, d'un **tableau**).

Pour effectuer une telle modification, il est nécessaire de **modifier individuellement** les éléments du tableau.

Exemple 1: Fonction **recopiant le contenu** d'une chaîne de caractères dans une autre:

```
/*
   Recopie la chaîne <src> dans la chaîne <dest>, en transférant
   au plus n symboles (incluant le terminateur éventuel).
*/

void strncpy(char dest[], const char src[], unsigned n)
{
    unsigned i;

    for (i = 0; i < n; i++)
        if (!((dest[i] = src[i])))
            break;
}
```

Remarque: Une fonction similaire, ainsi que de nombreuses autres fonctions effectuant des **traitements élémentaires** des chaînes de caractères, sont disponibles dans la **bibliothèque standard** du langage C. (Le **fichier d'en-tête** correspondant est `string.h`.)

Exemple 2: Fonction comptant le **nombre de chiffres** présents dans une chaîne de caractères:

```
unsigned nb_chiffres(const char s[])
{
    unsigned i, n;

    for (n = i = 0; s[i]; i++)
        if (s[i] >= '0' && s[i] <= '9')
            n++;

    return n;
}
```

Note: Cette fonction suppose que les symboles encodant les chiffres sont **consécutifs** dans l'alphabet (c'est le cas pour les procédés d'encodage courants).

Les tableaux multidimensionnels

Les mécanismes de définition et d'accès aux tableaux ne sont pas limités aux **vecteurs**. Des **tableaux multidimensionnels** peuvent être définis par des constructions de la forme suivante:

```
[ static ] [ const ] type id[taille1] [taille2] ... [taillen]  
    [ = { { ... { val, val, ... },  
            ⋮  
            { val, val, ... } ... } } ];
```

Principes:

- Cette définition crée un **tableau à n dimensions** appelé *id*, de taille $taille1.taille2 \dots taille_n$.
- Chaque **élément** de ce tableau est de type *type*, et est identifié par un **tuple** $(i_1, i_2, \dots, i_n)$, avec $\forall k \in [1, n]: 0 \leq i_k < taille_k$.

- Une **expression** de la forme

$$id[expr1][expr2] \dots [expr_n],$$

où l'évaluation de $expr1, expr2, \dots, expr_n$ produit respectivement les entiers $i_1, i_2, \dots, i_n$, permet d'**accéder à l'élément du tableau** identifié par le tuple $(i_1, i_2, \dots, i_n)$.

- Lorsqu'une fonction **reçoit en argument** un pointeur vers un tableau multidimensionnel, on utilise une **déclaration de paramètre** de la forme

$$type \ id[[taille1]][taille2] \dots [taille_n].$$

Remarque: La raison pour laquelle la taille de toutes les dimensions du tableau **sauf la première** doit être précisée est technique: Les éléments d'un tableau multidimensionnel sont placés **consécutivement** en mémoire. Pour calculer **l'emplacement** de l'élément $id[i_1][i_2] \dots [i_n]$ à partir des valeurs de $i_1, i_2, \dots, i_n$, il est nécessaire de connaître la valeur de $taille2, taille3, \dots, taille_n$.

Illustration: Considérons le **tableau bidimensionnel** défini par

```
int a[3][4];
```

qui comporte 3 lignes et 4 colonnes. Si l'on place ses 12 éléments dans des **emplacements consécutifs** numérotés 0, 1, 2, ... 11 de la mémoire, alors l'élément

```
a[i][j]
```

se trouvera dans l'emplacement possédant le numéro

$$4i + j.$$

On voit que, pour effectuer ce calcul, il n'est **pas nécessaire** de connaître la taille de la **première dimension** du tableau.

Application: Bibliothèque de transformations géométriques dans le plan

Principe: On représente une transformation linéaire

$$\vec{x} \mapsto \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \vec{x} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

où $\vec{x} \in \mathbb{R}^2$ contient les coordonnées d'un point du plan, par le tableau

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}.$$

Code C: Fichier geometrie-2d.c:

```
#include <math.h>
#include "geometrie-2d.h"

/*
   Place dans la transformation <t> une rotation d'angle <angle> autour
   de l'origine des axes.
*/

void geo2d_creer_rotation(double angle, double t[2][3])
{
    t[0][0] = t[1][1] = cos(angle);
    t[1][0] = sin(angle);
    t[0][1] = -t[1][0];
    t[0][2] = t[1][2] = 0.0;
}

/*
   Place dans la transformation <t> une mise à l'échelle de facteur
   <facteur> par rapport à l'origine des axes.
*/

void geo2d_creer_echelle(double facteur, double t[2][3])
{
    t[0][0] = t[1][1] = facteur;
    t[0][1] = t[1][0] = t[0][2] = t[1][2] = 0.0;
}
```

```

/*
    Place dans la transformation <t> une translation de vecteur (<x>, <y>).
*/

void geo2d_creer_translation(double x, double y, double t[2][3])
{
    t[0][0] = t[1][1] = 1.0;
    t[0][1] = t[1][0] = 0.0;
    t[0][2] = x;
    t[1][2] = y;
}

/*
    Compose deux transformations <t1> et <t2>, effectuées
    dans cet ordre, et place le résultat dans la transformation <t>.
*/

void geo2d_composer(const double t1[2][3], const double t2[2][3],
                    double t[2][3])
{
    unsigned i, j;

    for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
            t[i][j] = t2[i][0] * t1[0][j] + t2[i][1] * t1[1][j];

    t[0][2] += t2[0][2];
    t[1][2] += t2[1][2];
}

```

```

/*
    Applique la transformation <t> aux coordonnées <v> d'un point dans
    le plan.
*/

void geo2d_appliquer(const double t[2][3], double v[2])
{
    double tv[2][3], tr[2][3];

    geo2d_creer_translation(v[0], v[1], tv);
    geo2d_composer(tv, t, tr);

    v[0] = tr[0][2];
    v[1] = tr[1][2];
}

```

Fichier `geometrie-2d.h`:

```

void geo2d_creer_rotation(double, double[2][3]);
void geo2d_creer_echelle(double, double[2][3]);
void geo2d_creer_translation(double, double, double[2][3]);
void geo2d_composer(const double[2][3], const double[2][3], double[2][3]);
void geo2d_appliquer(const double[2][3], double[2]);

```

Exemple d'utilisation de cette bibliothèque:

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

#include "geometrie-2d.h"

int main()
{
    double p[2] = { 2.0, 3.0 };
    double tr[2][3];

    printf("Coordonnées initiales      : (%lf, %lf)\n", p[0], p[1]);

    geo2d_creer_echelle(0.5, tr);
    geo2d_appliquer(tr, p);
    printf("Après la mise à l'échelle: (%lf, %lf)\n", p[0], p[1]);

    geo2d_creer_rotation(-M_PI / 2.0, tr);
    geo2d_appliquer(tr, p);
    printf("Après la rotation          : (%lf, %lf)\n", p[0], p[1]);

    geo2d_creer_translation(-1.0, 2.0, tr);
    geo2d_appliquer(tr, p);
    printf("Après la translation        : (%lf, %lf)\n", p[0], p[1]);

    return 0;
}
```

Note: Pour compiler cet exemple, il est nécessaire de préciser que la **bibliothèque mathématique**, contenant notamment l'implémentation des fonctions `sin` et `cos`, doit être liée au programme:

```
% gcc -o test test-geometrie-2d.c geometrie-2d.c -lm
```

Chapitre 6

Les structures et les pointeurs

Les données structurées

Motivation: Il est souvent pratique de **regrouper** au sein d'une même structure des **données hétérogènes** destinées à être manipulées ensemble.

Exemples:

- Un **nombre complexe**, obtenu en associant une partie réelle et une partie imaginaire.
- Une **transformation géométrique** (telle que celles étudiées au chapitre 5), combinant une matrice de transformation et un vecteur de translation.
- La **fiche signalétique** d'un étudiant dans une base de données, regroupant nom, prénom, date de naissance, données d'inscription, photographie, ...
- La représentation d'une **image**, composée d'un pointeur vers une séquence de pixels organisés en lignes successives, associé aux nombres de lignes et de colonnes de l'image.

Les données structurées en langage C

En C, un **type structuré** se définit grâce à la construction suivante:

```
struct id
{
    type1 id1 [, id1b [, ... ]];
    type2 id2 [, id2b [, ... ]];
    :
};
```

Principes:

- Ce fragment de code définit un **nouveau type** portant le nom `struct id`.
- Les identificateurs *id1*, *id1b*, *id2*, *id2b*, ... désignent des **champs** de la structure. Chaque variable de type `struct id` possèdera une **instance** de chacun de ces champs.

- Dans une expression, l'opérateur `.` permet d'accéder aux **champs individuels** d'une structure: Si `v` est une **variable de type structuré** et `c` un **champ de cette structure**, alors `v.c` désigne le champ `c` de `v`.
- Les définitions de structures se placent habituellement en **dehors** des fonctions, et leur portée court alors jusqu'à la **fin du fichier source** incluant ces définitions. (Il est également permis d'en placer **à l'intérieur des fonctions**; les règles de portée sont alors identiques à celles des **variables locales**.)

Exemple:

```
struct complexe
{
    double re, im;
};

static struct complexe x, y, z;

:

z.re = x.re * y.re - x.im * y.im;
z.im = x.re * y.im + x.im * y.re;

:
```

Remarques:

- Il est possible de **fusionner** la définition d'un type structuré avec une définition de **variable(s)** possédant ce type. Dans ce cas, on peut **omettre le nom** du type.

Illustration:

```
struct
{
    double re, im;
} x, y, z;
```

- On peut **simplifier** l'utilisation des structures grâce à la construction

`typedef type id;`

 ,

qui permet dans la suite du programme de désigner le type *type* par le **nom abrégé** *id*.

Dans cette construction, si *type* est un type structuré, alors il est autorisé de **ne pas lui attribuer de nom** à la suite du mot-clé `struct`.

Illustration:

```
typedef struct
{
    double re, im;
} complexe;

static complexe x, y, z;
```

- Un type structuré peut définir des champs qui sont eux-mêmes structurés, ou constitués de tableaux.

Illustration:

```
#define LONGUEUR_MAX_NOM 50

struct etudiant
{
    char          nom[LONGUEUR_MAX_NOM + 1],
                prenom[LONGUEUR_MAX_NOM + 1];
    struct date   date_naissance;
};
```

L'utilisation des types structurés

- Le type des **paramètres** et de la **valeur de retour** d'une fonction peuvent être structurés.

Tout comme pour les **types primitifs**, le passage d'un argument structuré à une fonction s'effectue **par valeur**. En d'autres termes, les champs de la valeur structurée passée à la fonction sont **recopiés** dans les champs correspondants du paramètre, et une modification de la valeur de celui-ci dans la fonction **n'est pas répercutée** vers le code appelant.

- Les variables possédant un type structuré peuvent également faire l'objet d'**affectations**, mais **il n'est pas permis** de comparer leur valeur à l'aide de l'opérateur `==`.

Application: Bibliothèque de fonctions manipulant des nombres complexes

Fichier `complexe.c`:

```
#include <math.h>
#include "complexe.h"

complexe complexe_init(double re, double im)
{
    complexe c;

    c.re = re;
    c.im = im;

    return c;
}

double complexe_re(complexe c)
{
    return c.re;
}

double complexe_im(complexe c)
{
    return c.im;
}
```

```
complexe complexe_somme(complexe a, complexe b)
{
    complexe c;

    c.re = a.re + b.re;
    c.im = a.im + b.im;

    return c;
}
```

```
complexe complexe_difference(complexe a, complexe b)
{
    complexe c;

    c.re = a.re - b.re;
    c.im = a.im - b.im;

    return c;
}
```

```
complexe complexe_produit(complexe a, complexe b)
{
    complexe c;

    c.re = a.re * b.re - a.im * b.im;
    c.im = a.re * b.im + a.im * b.re;

    return c;
}
```

```
double complexe_module(complexe a)
{
    return hypot(a.re, a.im);
}

double complexe_distance(complexe a, complexe b)
{
    return complexe_module(complexe_difference(a, b));
}
```

Fichier complexe.h:

```
typedef struct
{
    double re, im;
} complexe;

complexe complexe_init(double, double);
double    complexe_re(complexe);
double    complexe_im(complexe);
complexe  complexe_somme(complexe, complexe);
complexe  complexe_difference(complexe, complexe);
complexe  complexe_produit(complexe, complexe);
double    complexe_module(complexe);
double    complexe_distance(complexe, complexe);
```

Le passage par variable

Dans certains cas, passer des arguments à une fonction **par valeur** présente des inconvénients:

- Si ces arguments sont de **grande taille**, recopier leur valeur consomme du **temps d'exécution** et nécessite de l'**espace** supplémentaire.
- La fonction ne peut pas **retourner des données** vers le code appelant par l'intermédiaire de ces arguments.

Une autre possibilité, que nous avons déjà mise en œuvre pour passer des **tableaux** à une fonction, est le passage d'arguments **par variable**. Cela consiste à transmettre à la fonction des **pointeurs** vers des données, plutôt que la valeur de celles-ci.

Les pointeurs

Rappel: Un **pointeur** vers une donnée représente l'**endroit** où cette donnée se trouve dans la mémoire de l'ordinateur.

Les pointeurs en langage C:

- Si T est un **type quelconque**, alors $T *$ désigne le type d'un pointeur vers une **valeur de type T** .

Note: Le type $void *$ désigne un pointeur vers des valeurs de **n'importe quel type**.

- Si v est une **variable**, alors $\&v$ s'évalue en un **pointeur vers v** .

Note: La variable v peut être **primitive**, un **élément** d'une variable de tableau, un **champ** d'une variable structurée, ou elle-même de type **pointeur**.

- Si $expr$ est une expression s'évaluant en un pointeur p , alors $*expr$ représente la **valeur pointée par p** .

Remarque importante: C'est au **programmeur** de s'assurer que *expr* pointe bien vers un **emplacement valide de la mémoire** lorsque `*expr` est évalué.

- Les pointeurs peuvent subir des **affectations** par le biais de l'opérateur `=`, et être **comparés** à l'aide de `==`.
- La **valeur spéciale** `NULL`, définie (entre autres) dans les fichiers d'en-tête `stdio.h` et `stdlib.h`, représente un **pointeur vide**, utile pour repérer des cas particuliers. **Convertie en un entier**, `NULL` vaut 0.

Exemple 1:

```
#include <stdio.h>

int main()
{
    int a;
    int *p; /* On aurait aussi pu écrire int a, *p; */

    p = &a; /* p pointe vers a */
    *p = 42; /* L'objet pointé par p (c'est-à-dire a) prend la valeur 42 */

    printf("%d\n", a); /* Affiche 42 */

    return 0;
}
```

Exemple 2: (passage d'arguments par variable)

```
#include <stdio.h>

/* Permute les entiers pointés par p1 et p2. */

void permuter(int *p1, int *p2)
{
    int aux;

    aux = *p1;
    *p1 = *p2;
    *p2 = aux;
}

int main()
{
    int a, b;

    a = 42;
    b = -7;

    printf("a = %d, b = %d\n", a, b); /* Affiche "a = 42, b = -7" */
    permuter(&a, &b);
    printf("a = %d, b = %d\n", a, b); /* Affiche "a = -7, b = 42" */

    return 0;
}
```

Exemple 3: La fonction `int scanf(const char *format, void *p1, void *p2, ...)` de la bibliothèque standard (fichier d'en-tête `stdio.h`) permet de lire depuis la console un certain nombre de valeurs spécifiées par la chaîne de format `format`. Ces valeurs sont placées par la fonction aux endroits désignés par les pointeurs `p1`, `p2`, ... La fonction retourne le nombre de valeurs effectivement lues, ou une valeur négative en cas d'erreur.

Les spécificateurs de type pouvant être utilisés dans la chaîne de format sont notamment `"%d"` (entier signé), `"%u"` (entier non signé) et `"%lf"` (réel en double précision).

Exemple:

```
int m, n;

printf("Entrez deux valeurs entières: ");
scanf("%d %d", &m, &n);
```

Les pointeurs vers des données structurées

Dans les programmes C, on est souvent amené à manipuler des **données structurées** par l'intermédiaire de pointeurs.

Si p est un **pointeur vers une donnée** possédant un champ c , alors la notation

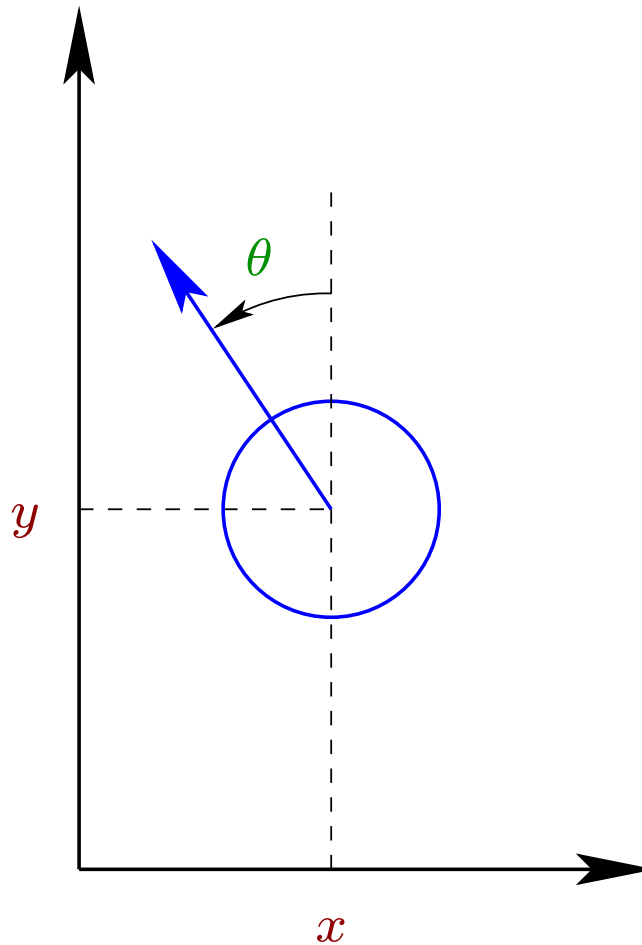
$$p \rightarrow c$$

est une **abréviation** pour

$$(*p).c ,$$

et permet donc d'**accéder** au champ c de la donnée pointée par p .

Illustration: Bibliothèque de fonctions permettant de gérer la **position** et l'**orientation** d'un objet mobile dans un espace à deux dimensions.



Fichier configuration-2d.c:

```
#include <math.h>
#include "configuration-2d.h"

/*
    Retourne un angle équivalent à <angle>, mais appartenant à
    l'intervalle  $[-\pi, \pi)$ .
*/

static double normaliser_angle(double angle)
{
    double r;

    r = fmod(angle, 2.0 * M_PI);
    if (r < -M_PI)
        r += 2.0 * M_PI;
    else
        if (r >= M_PI)
            r -= 2.0 * M_PI;

    return r;
}

/*
    Initialise la configuration *<c> avec la position (<x>, <y>) et
    l'orientation <theta>.
*/
```

```

void init_configuration(configuration *c, double x, double y, double theta)
{
    c -> x = x;
    c -> y = y;
    c -> theta = normaliser_angle(theta);
}

/*
    Met à jour la configuration *<c> après un déplacement de longueur <l>
    dans la direction courante.
*/

void deplacer_configuration(configuration *c, double l)
{
    c -> x -= l * sin(c -> theta);
    c -> y += l * cos(c -> theta);
}

/*
    Met à jour la configuration *<c> après une rotation d'angle <angle>.
*/

void tourner_configuration(configuration *c, double angle)
{
    c -> theta = normaliser_angle(c -> theta + angle);
}

```

Fichier configuration-2d.h:

```
typedef struct
{
    double x, y;    /* position */
    double theta;   /* orientation */
} configuration;

void init_configuration(configuration *, double, double, double);
void deplacer_configuration(configuration *, double);
void tourner_configuration(configuration *, double);
```

Note: La fonction `fmod( $x$ ,  $y$ )` de la **bibliothèque standard** permet de calculer le **reste de la division** de x par y , et retourne un résultat de **même signe** que x . Son prototype figure dans le fichier d'en-tête `math.h`.

Les pointeurs et les tableaux

Nous avons vu au chapitre 5 que, si v est une variable définie comme un **vecteur**, alors

- l'expression v fournit un pointeur vers le **premier élément** $v[0]$ de ce vecteur, et
- une expression de la forme $v + i$, où i est un entier, représente un pointeur vers le **$(i + 1)$ -ème élément** $v[i]$ du vecteur.

On en déduit que

- les expressions v et $\&v[0]$ sont **équivalentes**.
- les expressions $v + i$ et $\&v[i]$ sont elles aussi équivalentes.
- si T est un **type quelconque**, alors des déclarations d'un **paramètre a d'une fonction** de la forme $T *a$ et $T a[]$ sont **équivalentes**, et peuvent être utilisées indifféremment.

Illustration:

```
/*  
    Retourne la longueur de la chaîne de caractères <s>.  
*/  
  
unsigned strlen(const char *s)  
{  
    unsigned l = 0;  
  
    while (*s++)  
        l++;  
  
    return l;  
}
```

Remarques:

- Une **fonction similaire** est définie dans la bibliothèque standard. (Le fichier d'en-tête correspondant est `string.h`.)
- Dans l'expression `*s++`, l'opérateur `++` est **prioritaire** sur l'opérateur `*`.

L'allocation dynamique de mémoire

Beaucoup d'applications sont amenées à traiter des données dont la **taille n'est pas connue** au moment de la compilation du programme.

On utilise alors des fonctions de la bibliothèque standard, permettant au cours de l'exécution du programme d'**allouer** des blocs de mémoire de taille arbitraire, de les **redimensionner**, et de les **libérer** lorsqu'ils ne sont plus utiles.

Aperçu:

- La fonction `void *malloc(unsigned n)` tente d'allouer un **nouveau bloc de n octets**, et retourne un pointeur vers ce bloc (ou NULL en cas d'erreur).
- La fonction `void *realloc(void *p, unsigned n)` tente de **redimensionner** le bloc pointé par p (qui doit précédemment avoir été alloué), afin de lui donner la **nouvelle taille n**. Cette opération peut **déplacer** le bloc. Cette fonction retourne un pointeur vers le **nouvel emplacement** du bloc, ou NULL en cas d'erreur.

- La fonction `void free(void *p)` libère le bloc pointé par `p` (qui doit précédemment avoir été alloué).

Notes:

- Les prototypes de ces fonctions sont définis dans le fichier d'en-tête standard `stdlib.h`.
- Ces fonctions permettent de recopier le contenu de blocs de mémoire:
 - `void *memcpy(void *dest, const void *src, unsigned n)` recopie `n` octets du bloc pointé par `src` vers le bloc pointé vers `dest`, et retourne `dest`. Les blocs `src` et `dest` ne peuvent pas se chevaucher.
 - `void *memmove(void *dest, const void *src, unsigned n)` effectue la même opération, mais sur des blocs `src` et `dest` qui peuvent se chevaucher.

Leurs prototypes se trouvent dans le fichier d'en-tête `string.h`.

- L'opérateur `sizeof`, appliqué à une variable ou à un type, retourne le nombre d'octets nécessaires au stockage de la variable ou d'une instance du type.

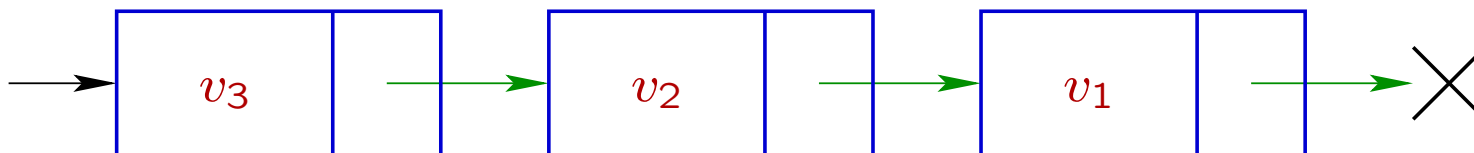
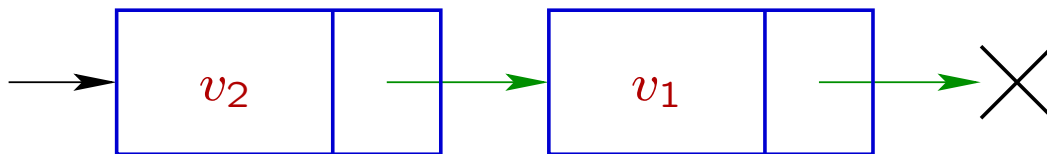
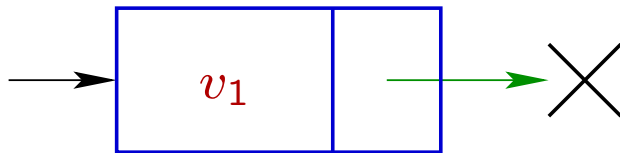
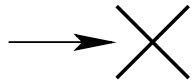
Exemple: Retournement d'une séquence de valeurs

Problème: Écrire un programme capable de lire au clavier un **nombre arbitraire** de valeurs, et de les restituer en **ordre inverse**.

Solution: On peut mémoriser les valeurs déjà lues sous la forme d'une **liste simplement liée**, c'est-à-dire une séquence d'éléments $e_1, e_2, \dots, e_p$ telle que:

- chaque élément e_i contient
 - une **valeur lue**, et
 - un **pointeur** vers l'élément suivant e_{i+1} (égal au **pointeur vide** dans le cas du dernier élément e_p).
- chaque **valeur lue** est représentée par un nouvel élément placé en **tête de la liste** (c'est-à-dire qu'il devient l'élément e_1).

Illustration: (Les valeurs $v_1, v_2, v_3, \dots$ sont lues dans cet ordre.)



Traduction en C:

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

struct element_liste
{
    int valeur;
    struct element_liste *suivant;
};

/*
    Retourne une liste simplement liée contenant des valeurs lues au clavier, par ordre
    inverse de lecture. En cas d'erreur, interrompt l'exécution du programme.
*/

static struct element_liste *creer_liste(void)
{
    struct element_liste *liste, *nouvel_element;
    int valeur;

    for (liste = NULL;;)
    {
        printf("Entrez une valeur positive (ou négative pour terminer): ");
        if (scanf("%d", &valeur) != 1 || valeur < 0)
            return liste;
    }
}
```

```

    nouvel_element = malloc(sizeof(struct element_liste));
    if (!nouvel_element)
        exit(-1);

    nouvel_element -> valeur = valeur;
    nouvel_element -> suivant = liste;
    liste = nouvel_element;
}
}

/*
    Affiche à l'écran les valeurs contenues dans la liste simplement liée <l> et libère
    cette liste.
*/

static void afficher_et_liberer_liste(struct element_liste *l)
{
    struct element_liste *e, *e_suivant;

    for (e = l; e; e = e_suivant)
    {
        printf(" %d", e -> valeur);
        e_suivant = e -> suivant;
        free(e);
    }
}

```

```
int main()
{
    struct element_liste *liste;

    liste = creer_liste();

    printf("Liste retournée:");
    afficher_et_liberer_liste(liste);
    printf("\n");

    return 0;
}
```

Remarques:

- La fonction `void exit(int status)` (fichier d'en-tête `stdlib.h`) permet de **terminer immédiatement** l'exécution du programme, en retournant le **code de diagnostic** `status` à l'environnement.
- À la **fin de l'exécution** d'un programme, les blocs de mémoire encore alloués sont **automatiquement libérés**.

L'allocation dynamique de tableaux

Les mécanismes d'allocation dynamique de mémoire peuvent être utilisés afin d'allouer des tableaux dont la taille n'est connue qu'à l'exécution d'un programme.

- **Tableaux unidimensionnels:** Il suffit d'un seul appel à `malloc`, en calculant soigneusement la quantité de mémoire nécessaire.

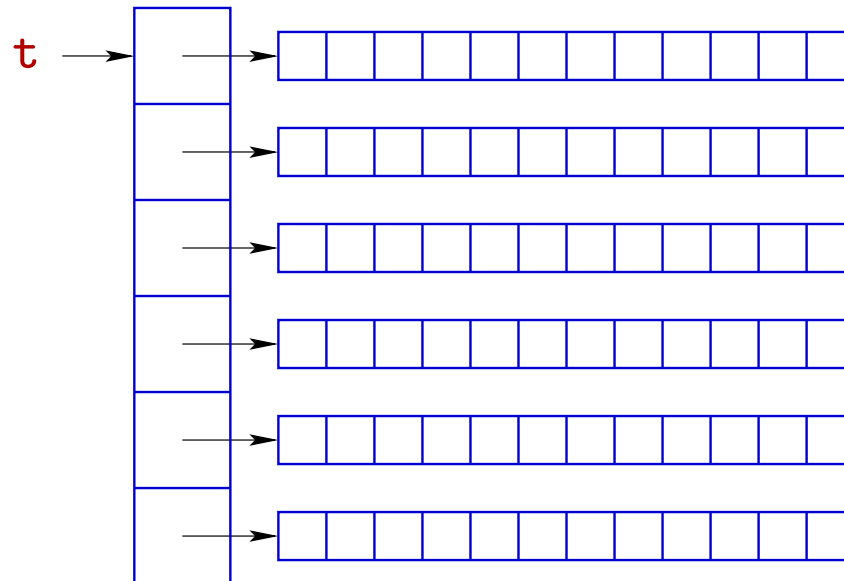
Exemple:

```
int n;  
double *v;  
  
:  
  
/* Calcul de la taille n à donner au vecteur v. */  
  
:  
  
v = malloc(n * sizeof(double));  
  
/* On peut maintenant placer des valeurs dans v[0], v[1], ..., v[n-1]. */
```

- **Tableaux multidimensionnels:** Si t est un tableau de dimension $d > 1$, alors $t[0]$, $t[1]$, $\dots$ contiennent des pointeurs vers des tableaux de dimension $d - 1$ représentant les lignes de t .

Pour allouer t , il faut donc

- allouer un vecteur destiné à contenir $t[0]$, $t[1]$, $\dots$,
- allouer récursivement des tableaux de dimension $d - 1$ correspondant aux lignes de t , et
- placer des pointeurs vers ces tableaux dans les éléments du vecteur.



Exemple 1: Allocation d'un tableau triangulaire.

```
#include <stdlib.h>

/* Alloue un tableau triangulaire t d'entiers, possédant un
 * élément t[i][j] pour tout i dans l'intervalle [0, n-1]
 * et j tel que 0 <= j <= i. Retourne un pointeur vers le
 * tableau créé, ou NULL en cas d'erreur. */

int **creer_tableau_triangulaire(unsigned n)
{
    int **t;
    unsigned i, k;

    if (!(t = malloc(n * sizeof(int *))))
        return NULL;

    for (i = 0; i < n; i++)
        if (!(t[i] = malloc((i + 1) * sizeof(int))))
        {
            for (k = 0; k < i; k++)
                free(t[k]);

            free(t);
            return NULL;
        }

    return t;
}
```

Exemple 2: Bibliothèque de fonctions manipulant des matrices à composantes réelles.

Fichier `matrice.c`:

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

/*
    Crée une nouvelle matrice de n lignes et m colonnes, dont les éléments
    sont initialement nuls.
    Retourne un pointeur vers la matrice créée, ou NULL en cas d'erreur.
*/

matrice *creer_matrice(unsigned n, unsigned m)
{
    matrice *r;
    unsigned i, j;

    r = malloc(sizeof(matrice));
    if (!r)
        return NULL;

    r -> el = malloc(n * sizeof(double *));
```

```

if (!(r -> el))
{
    free(r);
    return NULL;
}

for (i = 0; i < n; i++)
{
    r -> el[i] = malloc(m * sizeof(double));
    if (!(r -> el[i]))
    {
        for (j = 0; j < i; j++)
            free(r -> el[j]);
        free(r -> el);
        free(r);
        return NULL;
    }

    for (j = 0; j < m; j++)
        r -> el[i][j] = 0.0;
}

r -> n = n;
r -> m = m;

return r;
}

```

```

/*
    Libère l'espace mémoire occupé par la matrice m.
*/

void liberer_matrice(matrice *m)
{
    unsigned i;

    for (i = 0; i < m -> n; i++)
        free(m -> el[i]);

    free(m -> el);
    free(m);
}

/*
    Calcule la somme de deux matrices a et b. Retourne un pointeur vers
    une nouvelle matrice contenant cette somme, ou le pointeur vide en
    cas d'erreur.
*/

matrice *somme_matrices(const matrice *a, const matrice *b)
{
    matrice *s;
    unsigned i, j;

    if (a -> n != b -> n || a -> m != b -> m)
        return NULL;

```

```

s = creer_matrice(a -> n, a -> m);
if (!s)
    return NULL;

for (i = 0; i < a -> n; i++)
    for (j = 0; j < a -> m; j++)
        s -> el[i][j] = a -> el[i][j] + b -> el[i][j];

return s;
}

/*
    Calcule le produit de deux matrices a et b. Retourne un pointeur
    vers une nouvelle matrice contenant ce produit, ou le pointeur
    vide en cas d'erreur.
*/

matrice *produit_matrices(const matrice *a, const matrice *b)
{
    matrice *p;
    unsigned i, j, k;

    if (a -> m != b -> n)
        return NULL;

    p = creer_matrice(a -> n, b -> m);

```

```

    if (!p)
        return NULL;

    for (i = 0; i < a -> n; i++)
        for (j = 0; j < b -> m; j++)
            for (k = 0; k < a -> m; k++)
                p -> el[i][j] += a -> el[i][k] * b -> el[k][j];

    return p;
}

```

Fichier matrice.h:

```

/* Représente une matrice réelle de n lignes et m colonnes */

typedef struct
{
    double **el;
    unsigned n, m;
} matrice;

matrice *creer_matrice(unsigned, unsigned);
void      liberer_matrice(matrice *);
matrice *somme_matrices(const matrice *, const matrice *);
matrice *produit_matrices(const matrice *, const matrice *);

```

Les paramètres de `main`

Jusqu'à présent, nous avons défini la fonction `main`, qui représente le **point d'entrée** d'un programme C, sans lui attribuer de **paramètres**.

On peut cependant associer à cette fonction

- un paramètre **entier**, habituellement nommé `argc`, qui représente le **nombre d'arguments** fournis au programme par son environnement au moment de son exécution.
- un paramètre prenant la forme d'un **tableau de chaînes de caractères**, habituellement appelé `argv`, dont les éléments contiennent ces arguments.

Remarque: Dans la plupart des environnements d'exécution, le premier argument est **toujours présent** et contient le **nom du programme exécuté**.

Exemple: Programme affichant les arguments qui lui sont fournis.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    printf("Les arguments du programme sont");

    for (i = 0; i < argc; i++)
        printf(" [%s]", argv[i]);

    printf(".\n");

    return 0;
}
```

Exemple de compilation et d'exécution (UNIX):

```
% gcc -o arguments arguments.c
% ./arguments 123 456 abc
Les arguments du programme sont [./arguments] [123] [456] [abc].
```

Chapitre 7

Les piles et les files

La pile

Définition: Une **pile** (*stack*) est une structure de données capable de retenir un **ensemble de valeurs**, accessibles selon une discipline **LIFO** (*Last In First Out*).

Interface:

- **push**(s , v): **Empile** la valeur v sur la pile s .
- **pop**(s): **Dépile** une valeur depuis la pile s , et retourne cette valeur. Signale une erreur si la pile est vide.
- **top**(s): Retourne la valeur présente au **sommet** de la pile s , sans la dépiler. Signale une erreur si la pile est vide.
- **size**(s): Retourne le **nombre de valeurs** présentes dans la pile s .
- **is_empty**(s): Détermine si le contenu de la pile s est **vide**.

Implémentation 1: Vecteur

Principes:

- Les valeurs contenues dans la pile sont placées dans les composantes successives d'un **vecteur**.
- Le **nombre de valeurs** présentes dans la pile est retenu séparément.
- Afin de permettre de mémoriser des **valeurs de type quelconque**, on ne retient que des **pointeurs** vers ces valeurs.
- Le nombre de valeurs pouvant être placées dans une pile est **borné**. Une erreur est signalée lorsque l'on tente d'appliquer une opération push à une **pile saturée**.

Implémentation en C

Fichier stack.h:

```
#define MAX_STACK_SIZE 1000

typedef struct
{
    void    *content[MAX_STACK_SIZE];
    unsigned nb_values;
} stack;

stack    *stack_new(void);
void      stack_free(stack *);
int       stack_push(stack *, void *);
void      *stack_pop(stack *);
void      *stack_top(stack *);
unsigned  stack_size(stack *);
int       stack_is_empty(stack *);
```

Fichier stack.c:

```
#include <stdlib.h>
#include "stack.h"

/*  Retourne une nouvelle pile vide, ou NULL en cas d'erreur.  */

stack *stack_new(void)
{
    stack *s;

    s = malloc(sizeof(stack));
    if (!s)
        return NULL;

    s -> nb_values = 0;

    return s;
}

/*  Libère la pile <s>.  */

void stack_free(stack *s)
{
    free(s);
}
```

```
/* Empile le pointeur <p> sur la pile <s>.  Retourne 0 en cas de succès,  
   ou -1 si la pile est saturée.  */
```

```
int stack_push(stack *s, void *p)  
{  
    if (s -> nb_values >= MAX_STACK_SIZE)  
        return -1;  
  
    s -> content[s -> nb_values++] = p;  
    return 0;  
}
```

```
/* Dépile un pointeur depuis la pile <s> et retourne ce pointeur,  
   ou NULL si la pile est vide.  */
```

```
void *stack_pop(stack *s)  
{  
    if (!(s -> nb_values))  
        return NULL;  
  
    return s -> content[--(s -> nb_values)];  
}
```

```
/* Retourne le pointeur au sommet de la pile <s>, sans le dépiler,  
   ou NULL si la pile est vide.  */
```

```

void *stack_top(stack *s)
{
    if (!(s -> nb_values))
        return NULL;

    return s -> content[s -> nb_values - 1];
}

/* Retourne le nombre d'éléments présents dans la pile <s>. */

unsigned stack_size(stack *s)
{
    return s -> nb_values;
}

/* Détermine si la pile <s> est vide. */

int stack_is_empty(stack *s)
{
    return s -> nb_values == 0;
}

```

Exemple de programme de test

```
#include <stdio.h>
#include "stack.h"

int main()
{
    int    i = 10, i2;
    double d = 3.1416, d2;
    stack *s;

    s = stack_new();
    if (!s || stack_push(s, &i) || stack_push(s, &d))
        return -1;

    d2 = *((double *) stack_pop(s));
    i2 = *((int *) stack_pop(s));

    printf("valeurs dépilées: %lf, %d.\n", d2, i2);

    stack_free(s);

    return 0;
}
```

Exemple d'application

Problème: Étant donné un **fichier** contenant un texte composé de

- **parenthèses** de différentes sortes: (,), [,], {, }, et
- **d'autres caractères,**

déterminer si les parenthèses sont **correctement appariées** dans ce texte.

Exemples:

- $[- (b) + \text{sqrt}(4 * (a) * (c))] / (2 * a) \longrightarrow \text{oui}$
- $((x) + (y)) / 2 \longrightarrow \text{non}$

Solution: On exécute l'algorithme suivant:

1. On crée une **pile**, initialement vide.
2. On examine les **caractères du fichier**, dans l'ordre. Pour chaque caractère c :
 - Si c n'est **pas une parenthèse**, alors on l'ignore.
 - Si c est une **parenthèse gauche**, alors on l'empile.
 - Si c est une **parenthèse droite**, alors:
 - (a) On **dépile** un caractère c' .
 - (b) Si cette opération échoue (pile vide), ou si c' n'est pas le caractère symétrique de c , alors on **signale une erreur**.
3. S'il **reste des caractères** dans la pile après avoir examiné la totalité du fichier, alors on **signale une erreur**.

Traduction en C

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

static void verifie_parenthese(stack *s, char c, unsigned ligne, unsigned col)
{
    char *p;

    p = stack_pop(s);
    if (!p || *p != c)
        printf("erreur: Symbole incorrect ligne %u, colonne %u.\n", ligne, col);
    free(p);
}

static void verifie_fichier(FILE *f)
{
    stack    *s;
    unsigned  num_ligne, num_col;
    int       c;
    char      *p;

    s = stack_new();
    if (!s)
        exit(-1);
```

```

for (num_ligne = num_col = 1;; num_col++)
{
    c = fgetc(f); /* Lecture d'un caractère dans le fichier. */
    switch (c)
    {
        case EOF: /* Fin du fichier. */
            if (!stack_is_empty(s))
            {
                printf("erreur: %d symbole(s) encore ouvert(s) en fin de fichier.\n",
                    stack_size(s));
                while (stack_size(s))
                    free(stack_pop(s));
            }

            stack_free(s);
            return;

        case '\n':
            num_ligne++;
            num_col = 0;
            break;

        case '(':
        case '[':
        case '{':
            p = malloc(1);
            if (!p)
                exit(-1);
    }
}

```

```

        *p = c;

        if (stack_push(s, p))
            exit(-1);
        break;

    case ')':
        verifie_parenthese(s, '(', num_ligne, num_col);
        break;

    case ']':
        verifie_parenthese(s, '[', num_ligne, num_col);
        break;

    case '}':
        verifie_parenthese(s, '{', num_ligne, num_col);
        break;
    }
}

int main(int argc, char *argv[])
{
    FILE *f;

    if (argc != 2)
    {
        printf("usage: %s <nom-fichier>\n", argv[0]);
    }
}

```

```

    exit(-1);
}

f = fopen(argv[1], "r"); /* Ouverture d'un fichier en lecture. */
if (!f)
{
    printf("impossible d'ouvrir le fichier [%s]\n", argv[1]);
    exit(-1);
}

verifie_fichier(f);

fclose(f); /* Fermeture du fichier. */
exit(0);
}

```

Implémentation 2: Liste simplement liée

Principes:

- Les valeurs placées dans la pile sont retenues dans une **liste simplement liée** (*Linked List*, cf. chapitre 6).
- L'opération `push` place son nouvel élément en **tête de liste**.
- Les opérations `pop` et `top` travaillent également en **tête de liste**.
- On mémorise séparément le **nombre de valeurs** présentes dans la pile.

Avantage: Les éléments de la liste peuvent être **alloués à la demande**, ce qui permet d'adapter l'**espace mémoire** consommé par la structure à l'utilisation qui en est faite.

Implémentation en C

Fichier stack-ll.h:

```
typedef struct _stack_element
{
    void                *content;
    struct _stack_element *next;
} stack_element;

typedef struct
{
    stack_element *first;
    unsigned      nb_values;
} stack;

stack  *stack_new(void);
void    stack_free(stack *);
int     stack_push(stack *, void *);
void    *stack_pop(stack *);
void    *stack_top(stack *);
unsigned stack_size(stack *);
int     stack_is_empty(stack *);
```

Fichier stack-ll.c:

```
#include <stdlib.h>
#include <stdlib.h>
#include "stack-ll.h"

/*  Retourne une nouvelle pile vide, ou NULL en cas d'erreur.  */

stack *stack_new(void)
{
    stack *s;

    s = malloc(sizeof(stack));
    if (!s)
        return NULL;

    s -> first = NULL;
    s -> nb_values = 0;

    return s;
}

/*  Libère la pile <s>.  */

void stack_free(stack *s)
{
    stack_element *e, *e_next;
```

```

    for (e = s -> first; e; e = e_next)
    {
        e_next = e -> next;
        free(e);
    }

    free(s);
}

/* Empile le pointeur <p> sur la pile <s>.  Retourne 0 en cas de succès,
   ou -1 en cas d'insuffisance mémoire.  */

int stack_push(stack *s, void *p)
{
    stack_element *e;

    e = malloc(sizeof(stack_element));
    if (!e)
        return -1;

    e -> content = p;
    e -> next = s -> first;
    s -> first = e;
    s -> nb_values++;

    return 0;
}

```

```
/* Dépile un pointeur depuis la pile <s> et retourne ce pointeur,  
   ou NULL si la pile est vide.  */
```

```
void *stack_pop(stack *s)  
{  
    stack_element *e;  
    void          *p;  
  
    if (!(s -> first))  
        return NULL;  
  
    e = s -> first;  
    p = e -> content;  
  
    s -> first = e -> next;  
    s -> nb_values--;  
  
    free(e);  
  
    return p;  
}
```

```

/* Retourne le pointeur au sommet de la pile <s>, sans le dépiler,
   ou NULL si la pile est vide.  */

void *stack_top(stack *s)
{
    if (!(s -> first))
        return NULL;

    return s -> first -> content;
}

/* Retourne le nombre d'éléments présents dans la pile <s>.  */

unsigned stack_size(stack *s)
{
    return s -> nb_values;
}

/* Détermine si la pile <s> est vide.  */

int stack_is_empty(stack *s)
{
    return s -> nb_values == 0;
}

```

La file d'attente

Définition: Une file d'attente (*queue*, *FIFO buffer*) est une structure de données capable de retenir un **ensemble de valeurs**, accessibles selon une discipline **FIFO** (**F**irst **I**n **F**irst **O**ut).

Interface:

- `send( $q$ ,  $v$ )`: **Ajoute** la valeur v au contenu de la file q .
- `receive( $q$ )`: **Retire** une valeur de la file q , et retourne cette valeur. Signale une erreur si la file est vide.
- `front( $q$ )`: Retourne la valeur présente au **début de la file q** , sans la retirer. Signale une erreur si la file est vide.
- `size( $q$ )`: Retourne le **nombre de valeurs** présentes dans la file q .
- `is_empty( $q$ )`: Détermine si le contenu de la file q est **vide**.

Implémentation 1: Vecteur

Principes:

- Les valeurs contenues dans la file sont mémorisées dans un **vecteur**.
- Si N est la taille de ce vecteur, alors les valeurs ajoutées à la file sont placées à des indices **croissants modulo N** de ce vecteur.
- Des champs f et r situent respectivement les indices dans le vecteur:
 - de la **valeur la plus ancienne** (c'est-à-dire, la première qui quittera la file), et
 - du **premier emplacement libre** (c'est-à-dire, l'endroit où l'on placera la prochaine valeur ajoutée à la file).
- **Initialement**, on a $f = r = 0$.

Problème : Les situations où l'on a $f = r$ sont ambiguës.

Solution : On limite le **nombre d'objets** présents dans la file à $N - 1$.

Implémentation en C

Fichier queue.h:

```
#define MAX_QUEUE_SIZE 1000

typedef struct
{
    void    *contents[MAX_QUEUE_SIZE];
    unsigned front, rear;
} queue;

queue    *queue_new(void);
void      queue_free(queue *);
int       queue_send(queue *, void *);
void      *queue_receive(queue *);
void      *queue_front(queue *);
unsigned  queue_size(queue *);
int       queue_is_empty(queue *);
```

Fichier queue.c:

```
#include <stdlib.h>
#include "queue.h"

/* Retourne une nouvelle file vide, ou NULL en cas d'erreur. */

queue *queue_new(void)
{
    queue *q;

    q = malloc(sizeof(queue));
    if (!q)
        return NULL;

    q -> front = 0;
    q -> rear = 0;

    return q;
}

/* Libère la file <q>. */

void queue_free(queue *q)
{
    free(q);
}
```

```
/* Ajoute le pointeur <p> à la file <q>.  Retourne 0 en cas de succès,  
   ou -1 si la file est saturée.  */
```

```
int queue_send(queue *q, void *p)  
{  
    if (queue_size(q) >= MAX_QUEUE_SIZE - 1)  
        return -1;  
  
    q -> contents[q -> rear] = p;  
    q -> rear = (q -> rear + 1) % MAX_QUEUE_SIZE;  
  
    return 0;  
}
```

```
/* Retire un pointeur depuis la file <q> et retourne ce pointeur,  
   ou NULL si la file est vide.  */
```

```
void *queue_receive(queue *q)  
{  
    void *p;  
  
    if (q -> front == q -> rear)  
        return NULL;  
  
    p = q -> contents[q -> front];  
    q -> front = (q -> front + 1) % MAX_QUEUE_SIZE;  
  
    return p;  
}
```

```

/* Retourne le pointeur en tête de la file <q>, sans le retirer,
   ou NULL si la file est vide.  */

void *queue_front(queue *q)
{
    if (q -> front == q -> rear)
        return NULL;

    return q -> contents[q -> front];
}

/* Retourne le nombre d'éléments présents dans la file <q>.  */

unsigned queue_size(queue *q)
{
    return (MAX_QUEUE_SIZE + q -> rear - q -> front) % MAX_QUEUE_SIZE;
}

/* Détermine si la file <q> est vide.  */

int queue_is_empty(queue *q)
{
    return q -> front == q -> rear;
}

```

Implémentation 2: Liste simplement liée

Principes:

- Le contenu de la file est représenté par une **liste simplement liée** de pointeurs.
- Les valeurs ajoutées à la file sont placées en **fin de liste**.
- Des champs `head` et `tail` maintiennent des pointeurs vers les éléments situés respectivement au **début** et à la **fin** de la liste.
- On mémorise le **nombre de valeurs** présentes dans la file dans un champ séparé.

Implémentation en C

Fichier queue-ll.h:

```
typedef struct _queue_element
{
    void                *content;
    struct _queue_element *next;
} queue_element;

typedef struct
{
    queue_element *head, *tail;
    unsigned      nb_values;
} queue;

queue  *queue_new(void);
void    queue_free(queue *);
int     queue_send(queue *, void *);
void    *queue_receive(queue *);
void    *queue_front(queue *);
unsigned queue_size(queue *);
int     queue_is_empty(queue *);
```

Fichier queue-ll.c:

```
#include <stdlib.h>
#include "queue-ll.h"

/*  Retourne une nouvelle file vide, ou NULL en cas d'erreur.  */

queue *queue_new(void)
{
    queue *q;

    q = malloc(sizeof(queue));
    if (!q)
        return NULL;

    q -> head = NULL;
    q -> tail = NULL;
    q -> nb_values = 0;

    return q;
}

/*  Libère la file <q>.  */

void queue_free(queue *q)
{
    queue_element *e, *e_next;
```

```

    for (e = q -> head; e; e = e_next)
    {
        e_next = e -> next;
        free(e);
    }

    free(q);
}

/*  Ajoute le pointeur <p> à la file <q>.  Retourne 0 en cas de succès,
    ou -1 en cas d'insuffisance mémoire.  */

int queue_send(queue *q, void *p)
{
    queue_element *e;

    e = malloc(sizeof(queue_element));
    if (!e)
        return -1;

    e -> content = p;
    e -> next = NULL;

    if (q -> tail)
        q -> tail -> next = e;
    else
        q -> head = e;
}

```

```

    q -> tail = e;
    q -> nb_values++;

    return 0;
}

/*  Extrait un pointeur de la file <q> et retourne ce pointeur,
    ou NULL si la file est vide.  */

void *queue_receive(queue *q)
{
    queue_element *e;
    void          *p;

    if (!(q -> head))
        return NULL;

    e = q -> head;
    p = e -> content;

    q -> head = e -> next;
    if (--(q -> nb_values))
        q -> tail = NULL;

    free(e);

    return p;
}

```

```
/* Retourne le pointeur en tête de la file <q>, sans le retirer,  
   ou NULL si la file est vide.  */
```

```
void *queue_front(queue *q)  
{  
    if (!(q -> head))  
        return NULL;  
  
    return q -> head -> content;  
}
```

```
/* Retourne le nombre d'éléments présents dans la file <q>.  */
```

```
unsigned queue_size(queue *q)  
{  
    return q -> nb_values;  
}
```

```
/* Détermine si la file <q> est vide.  */
```

```
int queue_is_empty(queue *q)  
{  
    return q -> nb_values == 0;  
}
```