



1816
2016

ÉCOLE NATIONALE SUPÉRIEURE DES MINES

Algorithmique et Programmation

Programmation C

Nabil ABSI, Valeria BORODIN, Pierre UNY

{absi, valeria.borodin, uny}@emse.fr

2017-2018



Généralités

Objectifs :

- s'initier à la programmation en C
- acquérir les notions fondamentales de l'algorithmique
- écrire et implémenter des algorithmes en utilisant C
- manipuler de structures de données avancées

Thèmes :




Tri ■

Recursiveité ■

Traitement de chaînes de caractères ■

Simulation de files d'attente ■

Références

-  T.H. Cormen. *Introduction to Algorithms*. MIT Press, 2009. ISBN 9780262033848.
-  R. Malgouyres, R. Zrour, and F. Feschet. *Initiation à l'algorithmique et à la programmation en C- 3e éd. : Cours avec 129 exercices corrigés*. Informatique. Dunod, 2014. ISBN 9782100713899.
-  C. Delannoy. *Programmer en langage C : Cours et exercices corrigés*. Noire. Eyrolles, 2014. ISBN 9782212292244.
- **ANSI C Standard Library :** <http://www.csse.uwa.edu.au/programming/ansic-library.html>
- **Online reference for the C (standard) library :** <http://code-reference.com/c>

INTRODUCTION

- *Généralités : **algorithmique** et **programmation***
- *Algorithme vs. programme*
- *Un bref historique du C*
- *Pourquoi le C*
- *Du source à l'exécutable*
- ...

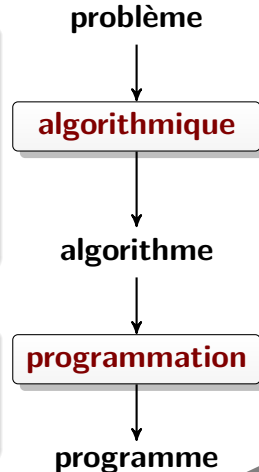
Algorithme vs. programme

Définition (Notion d'*algorithme*^a)

- *description formelle d'un procédé de traitement qui permet, à partir d'un ensemble d'informations initiales, d'obtenir des informations déduites;*
- *succession finie et non ambiguë d'opérations;*
- *se termine toujours.*

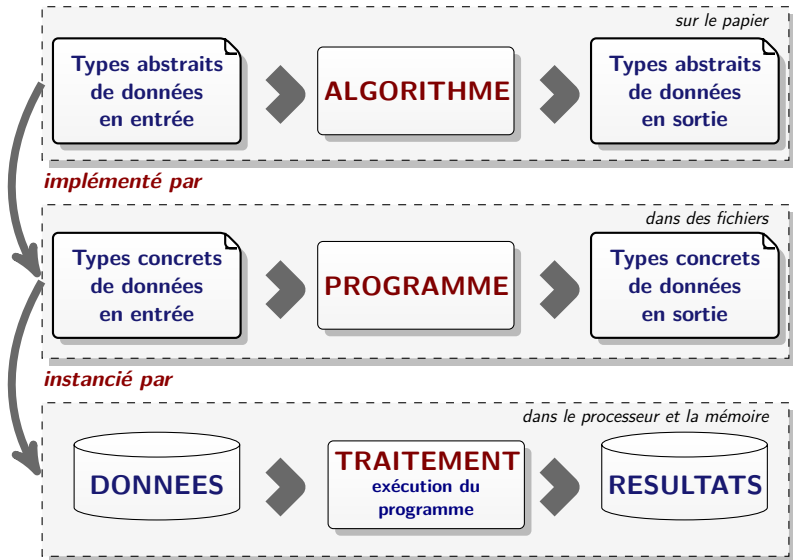
Définition (Notion de *programme*)

- *suite d'instructions définies dans un langage donné;*
- *décrit un algorithme.*



^amot dérivé du nom du mathématicien **Al-Khwârizmî** (≈ 780-850)

Algorithme vs. programme



Un bref historique

- **1967** langage BPCL (Richards, Bell Labs)
Basic Combined Programming Language
- **1970** langage B (Thompson, Bell Labs)
pas de types, manipulation de mots machine, ...
- **1972** **1er compilateur C** (Kernighan, Ritchie, Bell Labs)
- **1978** 1ère spécification publique C
- **1989** norme ANSI
- **1999** norme ISO/IEC9899: C99
- **2011** norme ISO/CEI 9899:2011: C11

Pourquoi C

- **modulaire** : peut être découpé en modules indépendants, qui peuvent être compilés séparément
- **universel** : n'est pas orienté vers un domaine d'application particulier
- **haut niveau** : détails hardware cachés au programmeur
- **souple** : très permissif, accès à la mémoire
- **portable** : un programme C conforme aux normes est utilisable sur une grande variété de machines
- **utile** au développement sur les **systèmes embarqués**

Classement TIOBE

Toujours autant utilisé

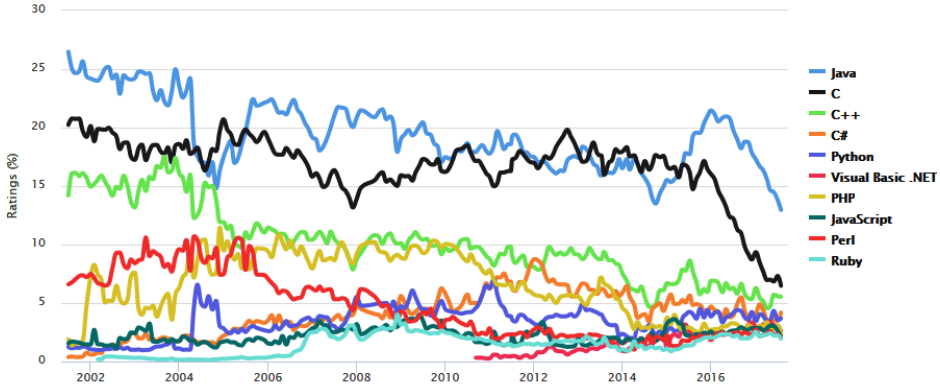


FIGURE – Index TIOBE (Source www.tiobe.com)

Langage C : *caractéristiques*

- langage ***impératif*** (par opposition aux langages *déclaratifs*)
- langage ***compilé*** (par opposition aux langages *interprétés*)
 - ▶ un programme C ne peut pas être exécuté tel quel
 - ▶ il est nécessaire de le traduire en langage machine
- **typage *statique*** : tout objet C a un type défini à la compilation



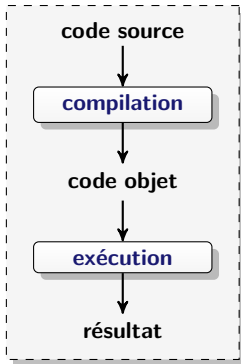
Outils nécessaires :

- **éditeur de texte** : écrire les codes sources, e.g. **gedit**, etc.
- **compilateur** : transformer les lignes de codes en programme exécutable

Du code source à l'exécution

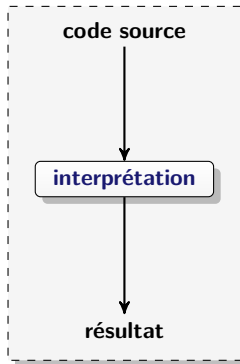
COMPILATION

ADA, C/C++, Fortran



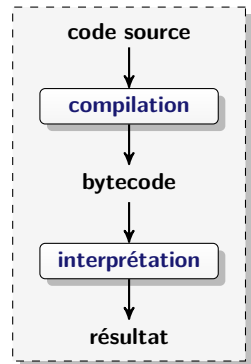
INTERPRETATION

Lisp, Prolog



COMPILATION/ INTERPRETATION

Python, Java



Définition (Compilateur)

Un **compilateur** est un programme informatique qui traduit un langage, le langage source (**programme texte**), en un autre, appelé le langage cible (**programme binaire exécutable**).

- **cc** : compilateur C sous UNIX
- **gcc** : compilateur C du projet GNU
 - ▶ installé d'office sur les systèmes Linux/Unix/macOSX



Compilation d'un programme :

`gcc [options] <fichiers à compiler> (-o <fichier de sortie>)`

- sans options, **gcc** compile et génère un exécutable de nom **a.out**
- **-o** permet de spécifier le nom du fichier de sortie

Quelques options : (<https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>)

- **-g** : mode debugger
- **-std=c99** : vérifier la conformité à la norme C99
- **-Wall** : imprimer tous les messages d'avertissement
- **-Werror** : produire une erreur à la place d'un avertissement
- **-w** : supprimer tous les avertissements



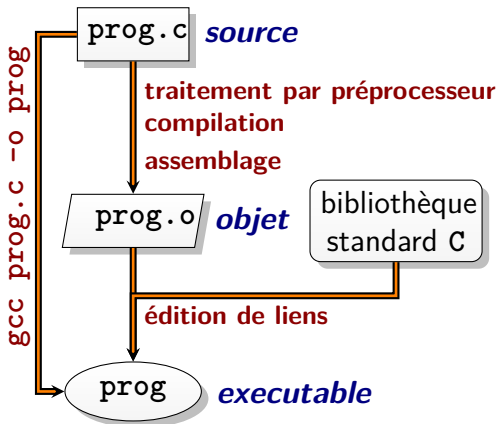
Exemple

`gcc -Wall -Werror -ansi -g prog1.c -o prog1.x`

Génération d'applications

Étapes d'une compilation

Pour passer d'un fichier **source** `prog.c` à un fichier **exécutable** `prog`, la compilation se divise schématiquement en quatre phases :



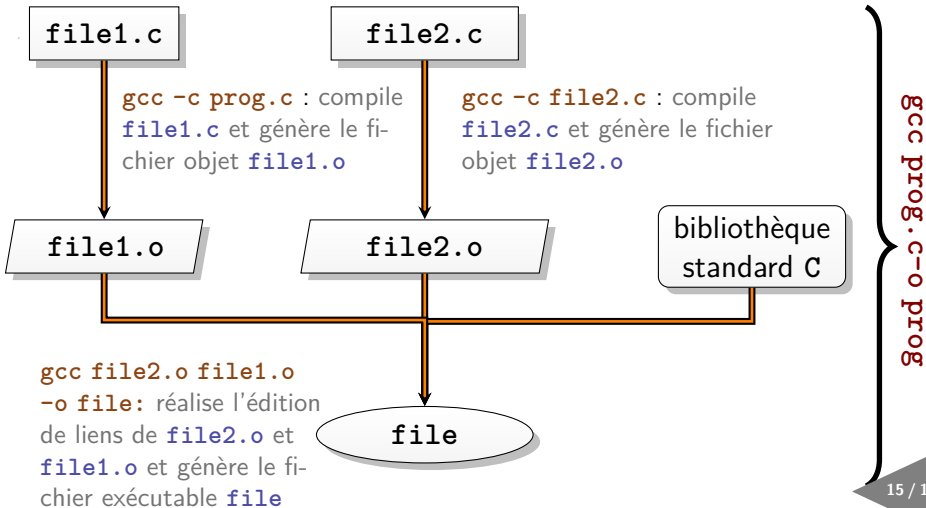
`gcc prog.c -o prog` \iff

`gcc -c prog.c` : compile `prog.c` et génère le fichier objet `prog.o`

`gcc prog.o -o prog` : fait l'édition de lien de `prog.o` et génère le fichier exécutable `prog`

Génération d'applications

Compilation séparée



Bonnes habitudes de programmation

- La compilation doit se faire **sans erreur ni avertissement (warning)**.
- Une mauvaise indentation n'est pas sanctionnée. Les programmes doivent cependant être **correctement indentés**.
- Les **noms** des fonctions, variables, etc. doivent être :
 - ▶ **parlants**
 - ▶ **pertinents**
 - ▶ **cohérents** (conventions **uniformes**).
- Les programmes doivent **être commentés** de façon **constructive**.

Composants élémentaires d'un programme en C

- **identificateurs** : associe un nom à une entité du programme
 - ▶ nom de variable ou de fonction
 - ▶ type défini par `typedef`, `struct`, `union` ou `enum`
 - ▶ étiquette
- **mots-clés**
- **constantes**
- **chaînes de caractères**
- **opérateurs** : arithmétiques, relationnels, d'affectation, sur les bits, etc.
- **signes de ponctuation**
- **commentaires** :

```
/* ceci est un commentaire bloc */  
// ceci est un commentaire ligne
```

Variables



Syntaxe :

```
<type> <nom_variable>;
```

Dans un programme, les données sont manipulées via des **variables** :

- une variable occupe une **case mémoire** ;
- une variable a un **type** qui définit quelles opérations sont valides (entier, booléen, réel, caractère, etc.) ;
- elle doit être déclarée **avant** d'être utilisée ;
- une variable est désignée par un **nom** (identificateur).

Une **déclaration de variable** est l'attribution d'un type et d'un nom.



Déclarer une variable d'un certain type interdit de l'utiliser pour stocker des informations d'un autre type.

Constantes en C



Syntaxe :

```
#define NOM_CONSTANTE texte_de_replacement
```



Exemple :

```
#define MAX 99  /* ATTENTION : pas de ; */  
#define PI 3.14  
  
int main() {  
    ...  
    float p = PI; /* déclaration et initialisation de variable */  
    char tab[MAX];  
    return 0;  
}
```

Mots réservés du langage C : (norme ANSI C99)

■ spécificateurs de stockage :

```
auto register static extern typedef
```

■ spécificateurs de type :

```
char    double enum    float    int        long  
short   signed struct union   unsigned void
```

■ qualificateurs de type :

```
const volatile
```

■ instructions de contrôle

```
break case continue default do        else  
for     goto if          switch while
```

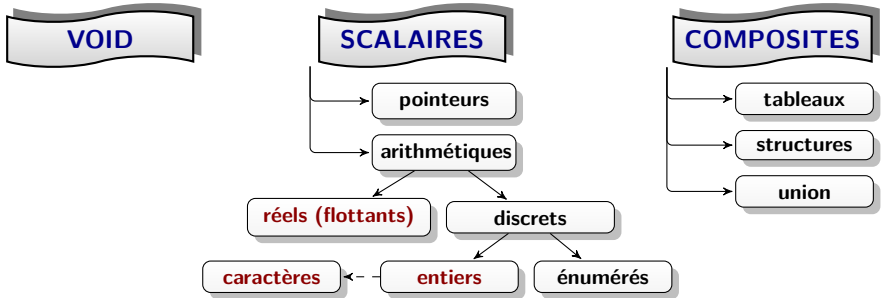
■ divers :

```
return sizeof
```

Opérateurs en C

- **opérateurs arithmétiques** : +, -, *, /, %
- **opérateurs d'affectation** : =, +=, -=, *=, /=
- **opérateurs logiques** : &&, ||, !
- **opérateurs de comparaison** : ==, !=, <, >, <=, >=
- **opérateurs d'incrément/décrémentation** : ++, --
- **opérateurs sur les bits** : <<, >>, &, |, ~, ^
- **autres opérateurs particuliers** : ? :, sizeof, cast

Types du langage C



■ Définit :

- ▶ un ensemble de *valeurs*
- ▶ un ensemble d'*opérations* applicables à ces valeurs

■ Détermine :

- ▶ la taille mémoire allouée aux variables de ce type
- ▶ le codage des valeurs

Type entier

Caractéristiques :

■ 5 types de variables entières :

1) <code>char</code>	2) <code>short int</code>	3) <code>int</code>	4) <code>long int</code>	5) <code>long long int</code>
----------------------	---------------------------	---------------------	--------------------------	-------------------------------

- peuvent prendre les modificateurs **signed** et **unsigned**, qui *ne changent pas la taille des types, mais la plage de valeurs*

Représentation des entiers signés :

- **Signe + Valeur Absolue (SVA)** : un bit contient le signe du nombre, les autres bits sont utilisés pour la valeur absolue
- **Complément à 1 (CPL1)** : les bits des nombres négatifs sont inversés
- **Complément à 2 (CPL2)** : les nombres négatifs sont représentés comme le complément à 1 et en ajoutant 1

Type entier

TABLE – Domaines \mathcal{D} de valeurs minimaux des types entiers (C90 et C99) quelle que soit sa représentation (SVA, CPL1 ou CPL2)

Type	Taille	Domaine minimal	Format
<code>signed char</code>	≥ 8 bits	$[-(2^7); 2^7 - 1]$	<code>%c (%hhi)</code>
<code>unsigned char</code>	≥ 8 bits	$[0; 2^8 - 1]$	<code>%c (%hhu)</code>
<code>short</code>	≥ 16 bits	$[-(2^{15} - 1); 2^{15} - 1]$	<code>%hi</code>
<code>unsigned short</code>	≥ 16 bits	$[0; 2^{16} - 1]$	<code>%hu</code>
<code>int</code>	≥ 16 bits	$[-(2^{15} - 1); 2^{15} - 1]$	<code>%i (%d)</code>
<code>unsigned int</code>	≥ 16 bits	$[0; 2^{16} - 1]$	<code>%u</code>
<code>long</code>	≥ 32 bits	$[-(2^{31} - 1); 2^{31} - 1]$	<code>%li</code>
<code>unsigned long</code>	≥ 32 bits	$[0; 2^{32} - 1]$	<code>%lu</code>
<code>long long</code>	≥ 64 bits	$[-(2^{63} - 1); 2^{63} - 1]$	<code>%lli</code>
<code>unsigned long long</code>	≥ 64 bits	$[0; 2^{64} - 1]$	<code>%llu</code>

$$\mathcal{D}(\text{char}) \leq \mathcal{D}(\text{short}) \leq \mathcal{D}(\text{int}) \leq \mathcal{D}(\text{long}) \leq \mathcal{D}(\text{long long})$$

Type caractère

- par définition, la taille du type `char` vaut toujours 1 octet (8 bits) :
 - ▶ 1 octet : 256 valeurs du ASCII étendu (www.table-ascii.com/)
- `char` en C peut être assimilé à un entier
- la plus petite donnée qui puisse être stockée dans une variable, i.e. sa taille définit l'unité de calcul pour les quantités de mémoire



- Un `char` au sens du C ne vaut pas toujours un octet. Il occupera au *minimum* 8 bits, mais il existe des architectures, ayant des char de 9 bits, de 16 bits, voire plus.
- Le `char` peut être **de base** (i.e. implicitement) **signed** ou **unsigned**, au choix du compilateur, ce qui peut s'avérer dangereux.



J. André, M. Goossens. *Codage des caractères et multi-linguisme : de l'ASCII à UNICODE - ISO/IEC-10646*. Cahiers GUTenberg (20),1985.

Types réels

- les nombres réels sont approximés par des *nombres à virgule flottante*
- les flottants sont stockés en mémoire sous la représentation de la *virgule flottante normalisée* : **mantisse** et un **exposant**,
e.g. $3.546E - 3$ est égal à 3.546×10^{-3} , i.e. 0,003546
- 3 types de variables à virgule flottante :

Type	Précision	Domaine minimal	Format
<code>float</code>	simple précision	$[10^{-37}; 10^{37}]$	<code>%f</code>
<code>double</code>	double précision	$[10^{-37}; 10^{37}]$	<code>%f (%lf)</code>
<code>long double</code>	suivant l'implémentation	$[10^{-37}; 10^{37}]$	<code>%Lf (%LF)</code>



- Le C90 était assez floue concernant les nombres à virgule flottante, leurs représentations, la précision des opérations, etc.
- Le C99 clarifie : **une implémentation C doit respecter la norme IEC 60559 :1989 Arithmétique binaire en virgule flottante pour systèmes à microprocesseur.**

Autres types

La norme C99 introduit :

■ le **type booléen** :

- ▶ prend uniquement la valeur vrai (**true**) ou faux (**false**)
- ▶ **stdbool.h** définit le type **bool**

■ les **types complexes** :

- ▶ sous les noms complex **float complex**, **double complex**, et **long double**
- ▶ la constante **I** correspond à la constante mathématique i (racine de -1)

Structure d'un programme C

```
[directives au préprocesseur]
[déclarations de variables externes]
[déclarations de fonctions]

int main() {
    déclarations de variables locales
    instructions
    return 0;
}
```

```
/* inclusion de bibliothèque */
#include <stdio.h>

/* programme principal */
int main () {

    printf("Hello world ! \n");
    return 0;

}
```

Un programme comprend :

- une liste **[optionnelle]** de déclarations (de variables globales, de types, de structures, etc.)
- une liste **[optionnelle]** de déclarations de fonctions
- une fonction **main**, **unique** et **obligatoire**, qui est le *point d'entrée* du programme

Les entrées-sorties conversationnelles :


Les fonctions `printf` et `scanf` font partie de la bibliothèque standard d'entrées/sorties `stdio.h` (**Standard Input/Output Header**) :

- `printf` : écriture formatée sur la sortie standard `stdout` (l'écran, par défaut)
- `scanf` : lecture formatée du flux standard d'entrée `stdin` (le clavier, par défaut)

Quelques codes de conversions :

- `%c` : `char`
- `%d` : `int`
- `%u` : `unsigned int`
- `%ld` : `long`
- `%f` : `float`, `double` écrit en notation décimale avec 6 chiffres après le point
- `%s` : chaîne de caractères

La fonction printf




Syntaxe :

```
#include <stdio.h>
printf (format, liste_d'expressions)
```

■ format :

- ▶ constante chaîne (entre " ")
- ▶ pointeur sur une chaîne de caractères (notion étudiée ultérieurement)

■ liste_d'expressions : suite d'expressions séparées par des virgules d'un type en accord avec le code format correspondant



Exemple :

```
int x = 10;
float y = 17.02;
printf("les valeurs sont: %d et %f \n", x, y);
```

La fonction scanf



Syntaxe :

```
#include <stdio.h>
int scanf ( const char *format [, arg [, arg]...]);
```

La fonction **scanf** lit les caractères en entrée, les interprète en concordance avec les spécifications de format décrites dans la chaîne **format**, et place les résultats dans les arguments **arg**.



Exemple :

```
printf("entier: ");
scanf("%d", &i);

printf("2 entiers et 1 double: ");
scanf("%d%d%lf", &i, &j, &d);

printf("saisir un caractère : ");
scanf( " %c", &c); // espace avant %c
```


Instruction if

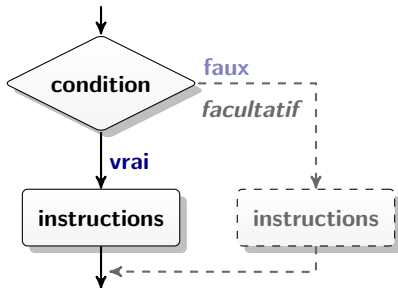
■ Notation C

```
if (expression)
    bloc_instructions_1
else
    bloc_instructions_2
```

■ Notation algorithmique

Algorithm 1 : instruction if

- 1: **if** (*expression*) **then**
- 2: instructions_1
- 3: **else**
- 4: instructions_2
- 5: **end if**



Exemple :

```
if (a > b)
    max = a;
else
    max = b;
```

else-if vs. switch

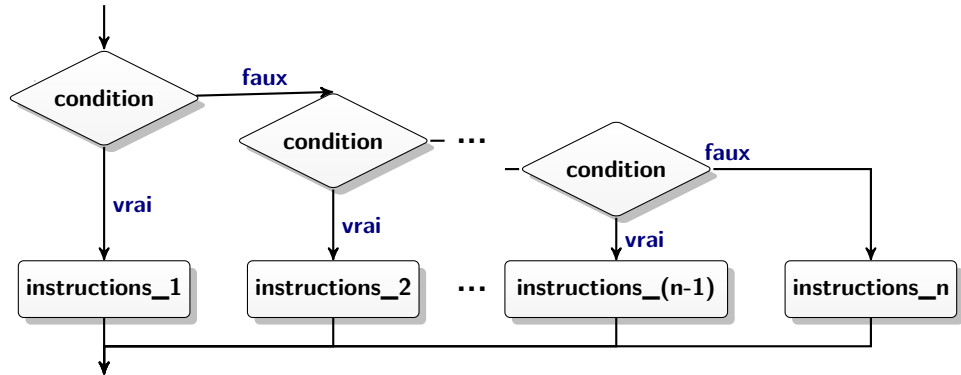


FIGURE – Choix en cascade

Instruction switch

■ Notation C

```
switch (expression) {  
    case constante_1 :  
        [bloc_instructions_1]  
    case constante_2 :  
        [bloc_instructions_2]  
    ...  
    case constante_n :  
        [bloc_instructions_n]  
    [default :  
        bloc_instructions]  
}
```



Exemple :

```
int n;  
printf ("n = ");  
scanf ("%d", &n);  
switch (n) {  
    case 0 :  
        printf ("nul\n");  
        break;  
    case 1:  
        printf ("petit\n");  
        break;  
    case 2:  
    case 3:  
        printf ("moyen\n");  
        break;  
    default:  
        printf ("grand\n");  
        break;  
}
```

Boucle *tant que*

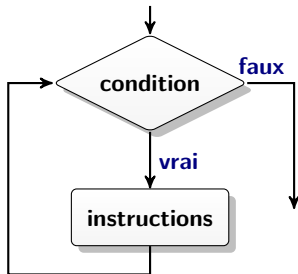
■ Notation C

```
while(condition)  
    bloc_instructions
```

■ Notation algorithmique

Algorithm 2 : instruction **while**

- 1: **while** (*condition*) **do**
- 2: bloc d'instructions
- 3: **end while**



Exemple :

```
int i = 0;  
while (i < MAX) {  
    printf("%d\n", i);  
    i++;  
}
```

Boucle *jusqu'à*

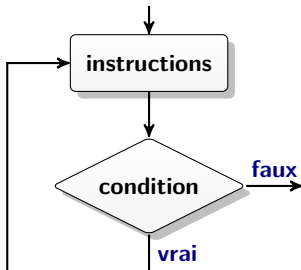
■ Notation C

```
do  
    bloc_instructions  
while(condition);
```

■ Notation algorithmique

Algorithm 3 : instruction **do-while**

- 1: **repeat**
- 2: bloc d'instructions
- 3: **until** (*condition*)



Exemple :

```
int i = 0;  
do {  
    printf("%d\n", i);  
    i++;  
}  
while (i < MAX);
```

Boucle for

■ Notation C

```
for(expression_1; expression_2; expression_3)  
    bloc_instructions
```

■ Notation algorithmique

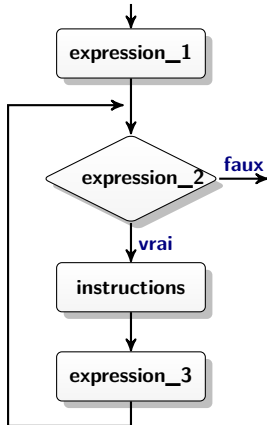
Algorithm 4 : instruction **for**

- 1: **for** k in \mathcal{D} by step of n **do**
- 2: bloc d'instructions
- 3: **end for**



Exemple :

```
for (int i = 0; i < MAX; i++)  
    printf("%d", i);
```



La fonction :

- constituant **de base** d'un programme **C**
- la seule sorte de **module** existant en **C** (idem en C++ et en Java)
- en **C** joue un rôle aussi **général** que la procédure (**Pascal**) ou le sous-programme (**Fortran**, **Basic**) des autres langages

Mise en œuvre :

- **déclaration**
- **appel** (utilisation)
- **définition**

Fonction : *définition*

Syntaxe :

```
type nom_fonction (type_1 arg_1,..., type_n arg_n) {  
  [déclarations de variables locales]  
  liste d'instructions  
}
```

■ en-tête :

- ▶ **type** : type de la fonction, i.e. valeur qu'elle retourne
- ▶ **nom_fonction**
- ▶ **type_1 arg_1,..., type_n arg_n** : liste d'arguments, appelés *paramètres formels*

■ corps :

- ▶ [déclarations de variables locales]
- ▶ liste d'instructions précédée d'une **instruction de retour à la fonction appelante** : **return** [expressions];

Fonctions



Exemple :

```
#include <stdio.h>

int puissance(int x, int n); //déclaration: prototype

int main() {

    int p;
    p = puissance(2,5); //appel avec 2 paramètres effectifs: 2 et 5
    printf("%f\n", p);
    return 0;
}

/* calcul x à la puissance n */
int puissance(int x, int n) { //définition avec 2 paramètres
    int i, p = 1;                //formels
    for (i = 1; i <= n; i++)
        p = p * x;
    return p;
}
```

Fonction : *anatomie*

type de la
valeur de retour

nom de la
fonction

liste de paramètres

```
int puissance (int x, int n) {  
  
    int i, p = 1;  ←-----  
  
    for (i = 1; i <= n; i++)  ←-----  
        p = p * x;  
  
    return p;  ←-----  
}
```

déclarations des
variables locales

instructions

instruction return

Fonction : *paramètres*

Les échanges entre les fonctions sont réalisées par les paramètres.

■ paramètres effectifs :

- ▶ paramètres *fournis lors de l'appel* de la fonction

■ paramètres formels (*muets*) :

- ▶ paramètres figurant dans l'en-tête de la fonction
- ▶ copies locales des valeurs *passées lors de l'appel*
- ▶ leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire

Fonction : *l'instruction* return

- peut mentionner n'importe quelle expression



Exemple :

```
float trinome (float x, int b, int c) {  
    return x * x + b * x + c;  
}
```

- peut apparaître à plusieurs reprises dans une fonction



Exemple :

```
double somme_valeur_absolue (double u, double v) {  
    double s;  
    s = u + v;  
    if (s > 0) return s;  
    else return -s;  
}
```

- si le type de l'expression dans **return** est \neq du celui déclaré dans l'en-tête, le compilateur mettra *automatiquement* en place des instructions de *conversion*

Fonctions

sans valeur de retour

Une fonction *peut* ne pas retourner de valeur :

Exemple :

```
#include <stdio.h>

/* prototype de affiche_carres */
void affiche_carres (int debut, int fin);

int main() {
    int debut = 5, fin = 10;
    affiche_carres (debut, fin);
    return 0;
}

void affiche_carres (int d, int f) {
    for (int i = d; i <= f; i++)
        printf ("%d a pour carré %d\n", i, i*i);
}
```

Tableaux : *définition*

Définition

Un **tableau** est une variable contenant une suite d'éléments, **tous de même type** désignés par un identificateur unique.

- Un **tableau** permet de mémoriser plusieurs données du même type.
- La déclaration du tableau **en une ligne** suffit pour déclarer toutes les données.
- Toutes les données du tableau doivent être **de même type** (même si le type des éléments du tableau peut être un type complexe comme une structure).

Tableaux :

déclaration statique



Syntaxe :

On déclare un tableau (statique) par :

```
typeElements nomTableau[NOMBRE_ELEMENTS];
```

- **typeElements** : type des éléments du tableau ;
- **nomTableau** : nom du tableau ;
- **NOMBRE_ELEMENTS** : constante indiquant le nombre d'éléments du tableau.




La taille d'un tableau statique doit être connue **lors de la compilation**, i.e. :

- soit une constante entière ;
- soit une constante symbolique.

NB : Un **objet statique** est un objet dont l'emplacement en mémoire est réservé **lors de la compilation** (et *pas à l'exécution*).

Parcourir un tableau

- les éléments d'un tableau sont numérotés par des *indices*
- les indices des éléments d'un tableau *commencent à 0* et non pas à 1
- dans un tableau de longueur n , on peut accéder aux cases d'indice 0 à $n - 1$

 **Exemple : déclaration d'un tableau de 100 entiers appelé *tab***

```
int tab[100];
```

Les éléments du tableau `tab`, qui comporte 100 éléments, ont des indices 0, 1, 2, ..., 99.

tab[0]	tab[1]	tab[2]	...	tab[98]	tab[99]
--------	--------	--------	-----	---------	---------

Parcourir un tableau



L'accès à un élément d'indice **supérieur ou égal à la taille d'un tableau** n'est pas valide et provoquera généralement une **erreur** mémoire (segmentation fault).

Tableaux : *exemples*

Exemple : *lecture et affichage d'un tableau*

```
#include <stdio.h>
#define NB_ELEMENTS 15 /* nombre d'elements du tableau */

/* fonction affichage d'un tableau */
void affichage_tab(float tableau[]) {
    for (int i=0; i<NB_ELEMENTS; i++)
        printf("l'element numéro %d vaut %f\n", i, tableau[i]);
}

/* fonction main : lit un tableau et le fait afficher */
int main() {
    float tableau[NB_ELEMENTS]; /* déclaration du tableau */
    for (int i=0; i<NB_ELEMENTS; i++) {
        printf("Entrez l'element %d : ", i);
        scanf("%f", &tableau[i]); /* lecture d'un element */
    }
    affichage_tab(tableau); /* affichage du tableau */
    return 0;
}
```

Tableaux : *exemples*

Exemple : *initialisation lors de la déclaration d'un tableau*

```
int main() {  
  
    int tab[5] = {3, 56, 21, 34, 6}; /* initialisation du tableau */  
    for (int i=0; i<5; i++) /* affichage des éléments */  
        printf("tab[%d] = %d\n", i, tab[i]);  
    return 0;  
}
```

Tableaux à double entrée

Un **tableau à double entrée** (*dimension 2* ou *matrice*) est un *tableau de tableaux*.

Exemple : *déclaration statique d'un tableau de dimension 2*

```
#define NB_LIGNES 100
#define NB_COLONNES 50
...
int tableau[NB_LIGNES][NB_COLONNES];
```

- l'élément `tableau[i]` est un tableau, et son élément `j` est donc noté `tableau[i][j]`
- **par convention** l'indice `i` représente le numéro d'une ligne, et l'indice `j` représente le numéro d'une colonne

Tableau transmis en argument

Exemple 1 : *fonction travaillant sur un tableau de taille fixe*

```
void init_tab (int tab[]) {  
    for (int i=0; i<10; i++)  
        tab[i] = 1;  
}
```

Exemple 2 : *fonction travaillant sur un tableau de taille variable*

```
int somme_tableau (int tab[], int taille_tab) {  
  
    int somme = 0, i;  
    for (i=0; i<taille_tab; i++)  
        somme += tab[i];  
  
    return somme;  
}
```

Algorithmique : *art de construire des algorithmes*

- **Validité** : aptitude à *réaliser exactement* la tâche pour laquelle il a été conçu
- **Robustesse** : aptitude à se protéger de *conditions anormales* d'utilisation
- **Réutilisabilité** : aptitude à *être réutilisé* pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu
- **Efficacité** : aptitude à *utiliser de manière optimale* les ressources du matériel qui l'exécute
- **Complexité** : le *nombre d'instructions élémentaires* à exécuter pour réaliser la tâche pour laquelle il a été conçu

Complexité : *définition*

Définition

La **complexité** d'un algorithme est une estimation du nombre d'opérations de base effectuées par l'algorithme en fonction de la taille des données en entrée de l'algorithme.

- La nature de l'algorithme sera différente selon que sa complexité sera plutôt de l'ordre de n , de n^2 , de n^3 , ou bien de 2^n . Le temps de calcul pris par l'algorithme ne sera pas le même.



Exemple

Si un algorithme écrit dans une fonction prend en entrée un **tableau** de n éléments, la complexité de l'algorithme sera une estimation du **nombre total d'opérations de base** nécessaires pour l'algorithme, en fonction de n . Plus n sera grand, plus il faudra d'opérations.

Quelques ordres de complexité classiques

TABLE – Temps d'exécution en fonction de n en effectuant 10^6 opérations par secondes

n	20	50	100	200	500	1000
$10^3 n$	0,02 s	0,05 s	0,1 s	0,2 s	0,5 s	1 s
$100n^2$	0,04 s	0,25 s	1 s	4 s	25 s	1 min 40 s
$10n^3$	0,08 s	1,25 s	10 s	1 min 20 s	20 min 50 s	2 h 46 min 40 s
2^n	1,05 s	36 ans 2 mois 11 jours	4,08E+16 ans	5,17E+46 ans	1,05E+137 ans	3,45E+287 ans
3^n	58 min 7 s	23080568019 ans	1,66E+34 ans	8,54E+81 ans	1,17E+225 ans	
$n!$	78218 ans	9,78E+50 ans	3,00E+144 ans			

Complexité : *la notion de O*

Définition

On appelle **opération de base** (ou opération élémentaire) :

- *toute affectation*
 - *test de comparaison : $=, <, >, \leq, \dots$*
 - *opération arithmétique : $+, -, *, /$*
 - *appel de fonctions comme `sqrt`, `incrément`, `décrément`.*
-
- Lorsqu'une fonction ou une procédure est appelée, le coût de cette fonction ou procédure est le nombre total d'opérations élémentaires engendrées par l'appel de cette fonction ou procédure.
 - Le temps de calcul pris par l'algorithme (sur une machine donnée) est directement lié à ce nombre d'opérations.

Complexité : la notion de O

Soit un algorithme dépendant d'une donnée d de taille n (e.g. un tableau de n éléments). Notons $\#(d, n)$ le nombre d'opérations engendrées par l'algorithme.

- On dit que l'algorithme est en **$O(f(n))$** si et seulement si on peut trouver un nombre K tel que (pour n assez grand) :

$$\#(d, n) \leq K * f(n)$$

- On dit que l'algorithme est en **$O(n)$** si et seulement si on peut trouver un nombre K tel que (pour n assez grand) :

$$\#(d, n) \leq K * n$$

Notation	Type de complexité
$O(1)$	complexité constante (indépendante de la taille de la donnée)
$O(\log(n))$	complexité logarithmique
$O(n)$	complexité linéaire
$O(n\log(n))$	complexité linéarithmique ou quasi-linéaire
$O(n^2)$	complexité quadratique
$O(n^3)$	complexité cubique
$O(n^p)$	complexité polynomiale
$O(e^n)$	complexité exponentielle
$O(n!)$	complexité factorielle

Complexité : *la notion de O*

Exemples : *calcul du maximum d'un tableau $T[n]$*

Algorithm 5 : *naturel*

```
1: max =  $T[0]$ 
2: for  $i = 1$  to  $n - 1$  do
3:   if  $T[i] > \text{max}$  then
4:     max =  $T[i]$ 
5:   end if
6: end for
```

■ Complexité : $O(n)$

Algorithm 6 : *pseudo-amélioré*

```
1: max_found = false,  $i = 0$ 
2: repeat
3:    $k = 1$ 
4:   for  $j = 0$  to  $n - 1$  do
5:     if  $T[i] \geq T[j]$  then
6:        $k = k + 1$ 
7:     end if
8:   end for
9:   if  $k == n$  then
10:    max_found = true
11:   else
12:     $i = i + 1$ 
13:   end if
14: until max_found is true
```

■ Complexité : $O(n^2)$

Complexité : *exemple*



Exemple : *matrice identité*

```
/* Génération de la matrice identité      */
/* Entrée : matrice M vide de taille nxn */

void Identite(double **M, int n) {
    int i,j;

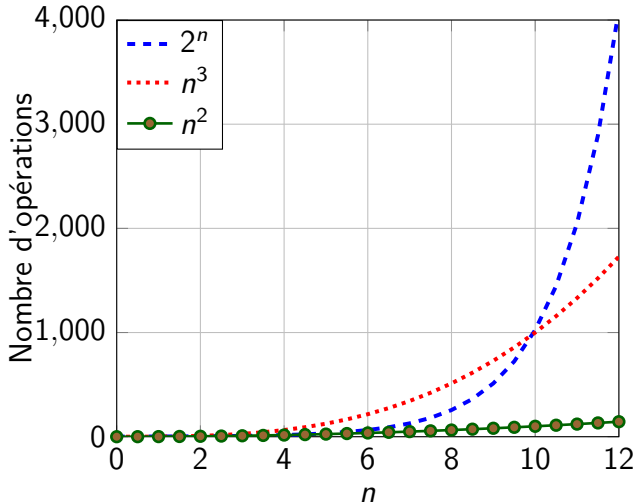
    for (i=0 ;i<n ;i++)
        for (j=0 ;j<n ;j++)
            M[i][j]=0;

    for (i=0 ;i<n ;i++)
        M[i][i]=1;
}
```

■ Complexité $O(n^2)$

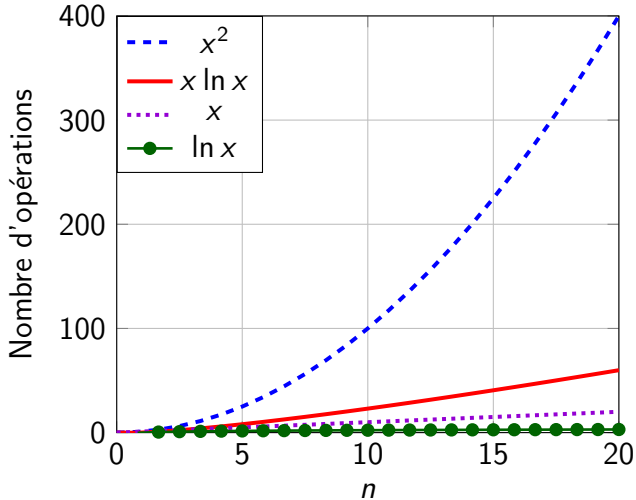


Complexité





Complexité



RÉCURSIVITÉ



Tour de Hanoï - source :
[https://www.hypershop.fr/
Marque-RoyaltyLane](https://www.hypershop.fr/Marque-RoyaltyLane)

Factorielle ■
Suite de Fibonacci ■
Algorithmes de tri ■
... ■

Conditions d'application :

description des
éléments de base

+

règles de construc-
tion réursive

définition récurrente d'un ensemble

- les **règles de construction récurives** permettent de caractériser les éléments d'un ensemble à partir d'éléments plus simples du même ensemble (en général obtenus ou construits à l'étape précédente)
- il y a toujours un ou plusieurs **éléments de base**.

Récurtivité

Récurtivité en informatique

- en informatique les traitements récurtifs doivent s'appliquer à partir d'un ***modèle récurtif cohérent***
- s'il s'agit des éléments d'un ensemble, ils doivent être construits à partir d'une ***définition récurtive*** et d'un ***élément de base*** (e.g. suite de Fibonacci, etc.)
- s'il s'agit d'une méthode de résolution, le problème à résoudre doit se ***décomposer en sous-problèmes de même nature***, et ce jusqu'à une ***formulation élémentaire*** (problème de base)

Récessivité

Exemple de fonction récessive : *Factorielle*

■ Récessive :

$$F(i) = \begin{cases} 1, & i = 0 \\ i \cdot F(i-1), & i > 0 \end{cases}$$

■ Fonction C associée :

```
long factrec (int n) {  
    if (n == 0)  
        return 1;  
    return n*factrec(n-1);  
}
```

■ Formule :

$$F(i) = \begin{cases} 1, & i = 0 \\ 1 \cdot 2 \cdot 3 \cdot \dots \cdot i, & i > 0 \end{cases}$$

■ Fonction C associée :

```
long factit (int n) {  
    int fact = 1;  
    while (n > 0) {  
        fact = fact*n;  
        n--;  
    }  
    return fact;  
}
```

Fonction factorielle récursive

- le **principe de résolution récursif** réclame que tout problème trouve une solution en résolvant un **sous problème de même nature**
- il est possible de calculer la factorielle d'un entier de manière récursive grâce à l'application de ce principe :

$$F(N) = N \cdot F(N - 1)$$

- l'appel récursif doit s'arrêter lorsque $N = 0$
(**c'est le test d'arrêt**)

Récurtivité

Les langages de programmation

- dans les langages de programmation la récursivité est la possibilité donnée à une fonction (ou une procédure) à s'appeler (se rappeler) elle-même
- les schémas de programmation qui en découlent sont en général plus simples et plus concis
- le prix à en payer est en général une consommation mémoire à l'exécution plus importante et un temps de traitement plus long
- souvent difficile à déboguer

Récurtivité

Récurtivité dite *directe*

Une fonction qui s'appelle elle-même.

- l'appel récursif est conditionné à un ***test d'arrêt***
- les instructions de ***pré-traitement*** se déroulent avant l'appel récursif
- les instructions de ***post-traitement*** après l'appel récursif

Modèle de fonction récursive directe

```
void recursive(void) {  
  
    //instructions de pré-traitement  
  
    if (!test_arret)  
        recursive();  
  
    //instructions de post-traitement  
}
```

Récurtivité

Récurtivité directe : *exemple*



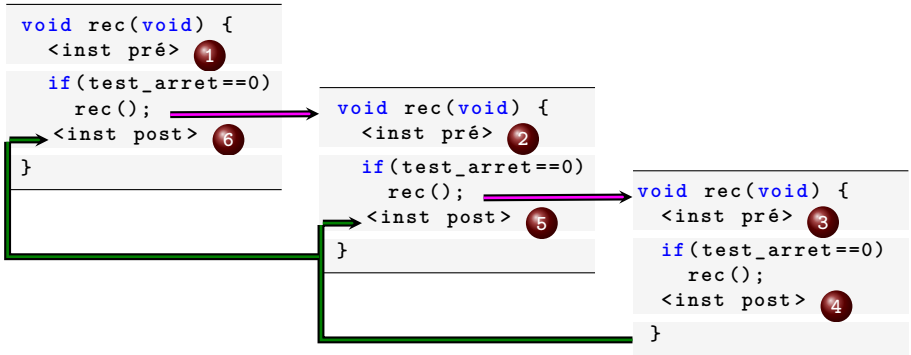
Récurtivité directe :

Considérons une fonction `rec()` qui s'appelle elle même 3 fois.

- chaque nouvel appel conserve l'adresse de retour ;
- les instructions de pré-traitement sont exécutées avant les appels récurtifs
- les instructions de post-traitement sont exécutées après les appels récurtifs, dans l'ordre de dépilement, i.e. inverse

Récurtivité

Réursive directe : exemple avec 3 appels récursifs



➡ **appel récursif** : branchement

← **retour récursif** : instruction suivante de la fonction appelante

Modèle de fonctions récessives *indirectes*



Récessivité indirecte :

Une fonction A appelle une fonction B. La fonction B appelle la fonction A.

```
void recursiveA(void) {  
    // instructions A1  
  
    if (test_continuation_B)  
        recursiveB();  
  
    // instruction A2  
}
```

```
void recursiveB(void) {  
    // instructions B1  
  
    if (test_continuation_A)  
        recursiveA();  
  
    // instruction B2  
}
```

Application de la récursivité

Certaines méthodes de résolution sont plus simples lorsqu'elles sont programmées récursivement :

- ▶ certains algorithmes de tri
- ▶ traitement des listes
- ▶ traitement des arbres

ALGORITHMES DE TRI

- *Algorithmes qui permettent d'organiser un ensemble d'objets selon une relation d'ordre déterminée*
- *Utiles à de nombreux algorithmes plus complexe*
- ...

Tris par comparaison :

Algorithmes lents $O(n^2)$: tri par sélection, tri à bulles, tri par insertion, ... ■

Algorithmes rapides $O(n \log n)$: tri fusion, tri de Shell, tri par tas, tri quicksort, ... ■

Tri par fusion

La récursivité *partout...*

- Le **tri par fusion** est basé sur la technique algorithmique ***divide et impera*** :
 - ▶ **diviser** le problème en un certain nombre de sous-problèmes ;
 - ▶ **régner** sur les sous-problèmes en les résolvant de manière récursive ;
 - ▶ **combiner** les solutions des sous-problèmes pour produire la solution du problème original.
- L'opération principale consiste à réunir 2 listes triées en une seule.
- L'efficacité de l'algorithme vient du fait que 2 listes triées peuvent être fusionnées en ***temps linéaire*** $O(n)$.

Tri par fusion

La récursivité *partout*...

- Division de $T[i, \dots, j]$ en 2 sous-tableaux, $n = j - i + 1$



- Appel récursif de $\text{TRI_FUSION}(T, i, i + \lfloor \frac{n}{2} \rfloor - 1)$
- Appel récursif de $\text{TRI_FUSION}(T, i + \lfloor \frac{n}{2} \rfloor, j)$
- Construction de la solution par un algorithme de fusion de $T[i, \dots, i + \lfloor \frac{n}{2} \rfloor - 1]$ et $T[i + \lfloor \frac{n}{2} \rfloor, \dots, j]$

Tri par fusion :

TRI_FUSION(T, i, j)

Diviser, régner, combiner

Algorithm 7 : TRI_FUSION(T, i, j)

```
1:  $n = j - i + 1$   
2: if  $n > 1$  then  
3:   TRI_FUSION( $T, i, i + \lfloor \frac{n}{2} \rfloor - 1$ )  
4:   TRI_FUSION( $T, i + \lfloor \frac{n}{2} \rfloor, j$ )  
5:   FUSIONNER( $T, i, i + \lfloor \frac{n}{2} \rfloor - 1, j$ )  
6: end if
```

Tri par fusion :

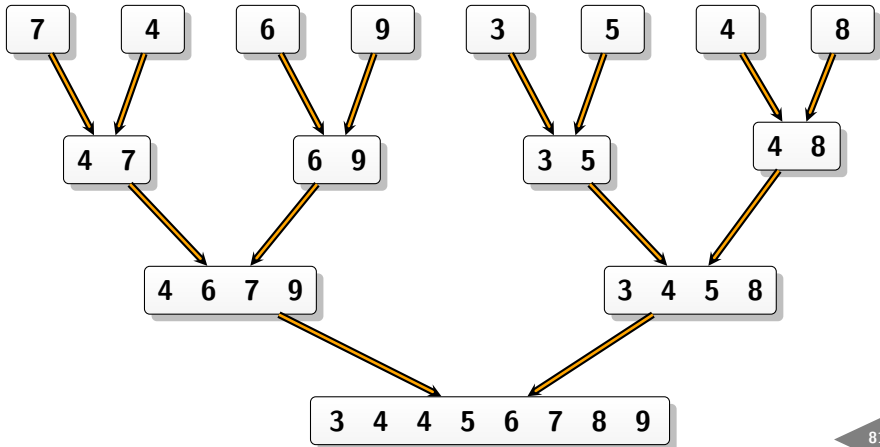
FUSIONNER(T, g, m, d)

Algorithm 8 : FUSIONNER(T, g, m, d)

```
1: for  $i$  de  $g$  à  $m$  do
2:    $R[i] = T[i]$ 
3: end for
4: for  $j$  de  $m + 1$  à  $d$  do
5:    $R[j] = T[d + m + 1 - j]$ 
6: end for
7:  $i = g, j = d$ 
8: for  $k$  de  $g$  à  $d$  do
9:   if  $R[i] < R[j]$  then
10:     $T[k] = R[i], i = i + 1$ 
11:  else
12:     $T[k] = R[j], j = j - 1$ 
13:  end if
14: end for
```


Tri par fusion

Fonctionnement du tri par fusion sur le tableau $\{7, 4, 6, 9, 3, 5, 4, 8\}$



Tri par fusion : *complexité*

Soit $\theta(n)$ la ***fonction complexité*** de TRI_FUSION

- FUSIONNER(T, i, k, j) exécute au plus $n = j - i + 1$ opérations élémentaires
- TRI_FUSION($T, 1, N$) engendre deux tris fusion et une fusion (complexité maximale $N - 1$)
 - ▶ $\theta(1) = 0$
 - ▶ $\theta(n) \leq \theta(\lceil \frac{n}{2} \rceil) + \theta(\lfloor \frac{n}{2} \rfloor) + n$

Tri par fusion : *complexité*

Considérons la suite récurrente $U(n)$ définie par :

- $U(1) = 0$
 - $U(n) = U(\lfloor \frac{n}{2} \rfloor) + U(\lceil \frac{n}{2} \rceil) + n$
-

- $U(n)$ majore toute solution de $\theta(n)$
- Exprimons $n = 2^k$: $U(n) = U(2^k)$
- Posons $V(k) = U(2^k)$. Combien vaut $V(k)$?
 - ▶ $V(0) = 0$
 - ▶ $V(k) = 2 \cdot V(k-1) + 2^k$

Tri par fusion : *complexité*

Raisonnement par récurrence :

$$2^0 V(k) = 2^1 V(k-1) + 2^k$$

$$2^1 V(k-1) = 2^2 V(k-2) + 2^k$$

$$2^2 V(k-2) = 2^3 V(k-3) + 2^k$$

.....

$$2^{k-1} V(1) = 2^k V(0) + 2^k$$

$$2^k V(0) = 0$$

En sommant, on obtient

$$V(k) = k2^k = n \log_2 n \iff U(n) = n \log_2 n$$

Donc, la **complexité** de l'algorithme tri par fusion : $O(n \log_2 n)$

POINTEURS ALLOCATION DYNAMIQUE

-
- *Notion d'adresse*
 - *Pointeurs*
 - *Pointeurs et fonctions*
 - ...

Adresse d'une variable

- une **variable** occupe un emplacement dans la **mémoire centrale** de l'ordinateur
- la mémoire est composée d'une **suite d'octets** :
 - ▶ numérotés de 0 à la taille de mémoire
 - ▶ identifiés **de manière univoque** par un *numéro*, appelé **adresse**
- l'**adresse** d'une variable est l'adresse mémoire de *son premier octet*
- en C, l'adresse d'une variable est exprimée par l'**opérateur &**



Exemple :

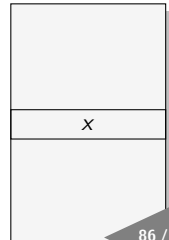
```
int x; /* occupe 4 octets en mémoire
        sur une machine 32 bit */
```

&x vaut 8052 : adresse mémoire de x, i.e. l'adresse mémoire à partir de laquelle l'objet est stocké

adresse

0
1
:
8052
:

mémoire



Pointeurs

Les variables dont les valeurs sont des adresses s'appellent des **pointeurs**.



Syntaxe :

```
<type_variable_pointée>* <pointeur>;
```

- un pointeur est **typé** : `<type_variable_pointée>`
- l'*opérateur unaire d'indirection* `*` permet d'accéder directement à la valeur de l'objet pointé `<pointeur>`

Mise en œuvre d'un pointeur se déroule en 3 étapes :

- déclaration
- affectation
- utilisation

Pointeurs :

déclaration et affectation



Exemple :

```
#include <stdio.h>

int main() {
    /* déclaration d'une variable p de
    type pointeur sur int */
    int *p;

    int x = 3;

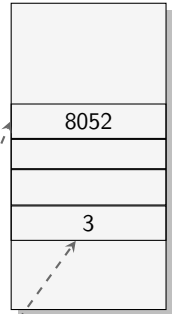
    /* chargement d'un pointeur */
    /* rangement dans p l'adresse de x*/
    p = &x;

    float z;
    float *q; // un pointeur est typé
    q = &z; // on dit que q pointe sur z
    ...
    return 0;
}
```

adresse

0
1
:
8040
:
8052
:

mémoire



p

x

Pointeurs : *manipulation*



Exemple :

```
/* si p est un pointeur, *p désigne la variable pointée par p */  
int  x, y;  
int  *p, *q;  
  
p = &x; // p pointe sur x  
*p = 5; // équivalent à x = 5  
y = *p; // équivalent à y = x  
  
/* ERREUR : pas d'initialisation de la variable pointeur */  
*q = 21;
```



- avant d'utiliser l'objet pointé par un pointeur p, il faut s'assurer que le pointeur p **contient l'adresse d'un emplacement mémoire correct**
- si l'on déclare un pointeur q sans le faire pointer sur un emplacement mémoire correct, tout accès à *q produira soit un résultat faux, soit une erreur mémoire (erreur de segmentation).

Pointeurs : *fonctions*

Passage de paramètre par valeur :

- dans un ***passage par valeur*** la fonction ne peut pas modifier la variable passée en paramètre
- la variable passée en paramètre est automatiquement *recopiée* et la fonction travaille sur une *copie* de la variable
- la modification de la copie *n'entraîne pas* une modification de la variable originale



Exemple :

```
void  afficher  (int a) {  
    printf("valeur = %d\n", a);  
}  
  
....  
int   x = 10;  
  
// appel de afficher :  
afficher(x); //le compilateur copie la valeur de x dans a  
afficher(20); //le compilateur copie la valeur 20 dans a
```

Pointeurs : *fonctions*



Exemple : passage de paramètre par valeur

```
void limiter_a_100 (int a) {  
    if (a > 100)  
        a = 100;  
}  
...  
int x = 150;  
limiter_a_100(x);
```

- x n'est pas modifié, la fonction a modifié sa copie locale a
- le paramètre a est une copie de la variable passée en paramètre
- la fonction peut modifier a, cela n'aura aucune incidence sur la variable x passée

Pour qu'une fonction puisse **modifier** la variable passée en paramètre, elle doit recevoir en paramètre non pas la **valeur**, mais l'**adresse de la variable**.

Pointeurs : *fonctions*

Passage de paramètre par adresse :

- pour **modifier** une variable par un appel de fonction, il faut passer en paramètre non pas la **valeur**, mais un **pointeur** qui pointe sur la variable
- il contient l'adresse de la variable à modifier



Exemple :

```
/* mise de 100 dans la variable pointée par p, donc dans la
variable dont l'adresse est passée en paramètre */
```

```
void limiter_a_100 (int *p) {
    if (*p > 100) *p = 100;
}
```

```
...
```

```
int x = 150, y = 120;
```

```
/* à l'appel le paramètre local p reçoit l'adresse de x =>
c'est x qui est modifié par la fonction */
```

```
limiter_a_100(&x);
limiter_a_100(&y); // idem pour y
```

Pointeurs : *fonctions*



- lors de l'exécution du programme, les variables locales d'une fonction sont *créées à l'entrée* dans la fonction puis *détruites en sortie*
- elles seront *recréées à l'appel suivant* de la fonction, mais *pas nécessairement au même emplacement mémoire*
- donc, si x est une variable locale & x change de valeur d'un appel de la fonction à un autre.

Cas particulier des tableaux :

- un tableau se passe nécessairement par adresse
- le *nom de tableau* est un *pointeur constant*
- l'adresse d'un tableau `tab` s'exprime par `tab` et non pas `&tab`
- le *nom du tableau* désigne l'*adresse de son premier élément*
- l'adresse d'un tableau est l'adresse de son premier élément, donc peut s'exprimer aussi par `&tab[0]`



Exemple :

```
float tab[3];  
float *ptab;  
ptab = tab;
```

- $\text{tab} \iff \&\text{tab}[0]$ et $\text{tab}+i \iff \&\text{tab}[i], \forall i = \overline{0,2}$
- $*\text{ptab} \iff \text{tab}[0]$ et $*(\text{ptab}+i) \iff \text{tab}[i], \forall i = \overline{0,2}$

Cas particulier des tableaux :



Exemple :

```
// équivalent à void afficher (int t[], int taille)
void afficher (int *t, int taille) {

    /*dans la fonction, que le paramètre soit déclaré en int *t
    ou en int t[], le tableau se manipule par t[i] */

    for (int i = 0; i < taille; i++)
        printf("%d ", t[i]);
}

...
afficher(tab, 20); // équivalent à afficher(&tab[0], 20)
```

- Une fonction peut donc modifier un tableau passé en paramètre.

Allocation dynamique

Généralités :

- la taille des données n'est pas toujours connue lors de la programmation
- la plupart du temps :

réserve de taille *maximale* \Rightarrow gaspillage de l'espace mémoire

Généralités :

Saisie d'un tableau d'entiers avec demande de la taille réelle utilisée :

```
int tab[100]; //réservation de la taille max (supposée)
int nb_int, i;

printf("nombre d'entiers: ");
scanf("%d", &nb_int);

for(i=0; i<nb_int; i++) //problème si nb_int >= 100
    scanf("%d", &tab[i]);
```

Allocation dynamique

Généralités :

Pour ajuster la taille au besoin exact :

- demande d'allocation dynamique de mémoire
- via l'utilisation de la fonction standard **malloc** de la bibliothèque `<stdlib.h>`

Fonction `malloc()`



Prototype

```
void *malloc (size_t taille); //dans <stdlib.h>
```

- permet d'allouer un bloc mémoire de `taille` octets, sans l'initialiser
- **valeur de retour :**
 - ▶ un pointeur vers l'adresse du bloc alloué, s'il y a suffisamment de mémoire disponible
 - ▶ le pointeur `NULL`, sinon (place disponible insuffisante)

Valeur de retour/Conversion de type

- `malloc()` alloue une suite d'octets sans présumer de ce que le programmeur va y mettre (des entiers, des structures, etc.)
- le pointeur retourné est de type `void*` : il pointe sur une zone de mémoire sans précision du type des données pointées
- pour manipuler les données, il faut convertir le pointeur en un pointeur du *type des données*

Allocation dynamique

Conversions : exemple

```
int *ptab;  
int nb_int, nb_oct, i;  
  
printf("nombre d'entiers: ");  
scanf("%d", &nb_int);  
  
nb_oct = nb_int * sizeof(int);  
ptab = (int*) malloc(nb_oct);  
  
for(i = 0; i < nb_int; i++)  
    scanf("%d", &ptab[i]);
```

pas de réservation de
tableau à la compilation

taille en octets d'une
variable du type `int`

taille allouée
(en octets)

compilation conversion
de pointeur (cast)

Libération de la mémoire

- lorsque l'on a fini d'utiliser une zone mémoire allouée dynamiquement, **il faut la libérer**
- utilisation de la fonction standard **free** de la bibliothèque **<stdlib.h>**

Allocation dynamique

Fonction `free()`



Prototype

```
void free(void *ptr); //dans <stdlib.h>
```

- libère la zone mémoire, pointée par le pointeur `ptr`, et allouée précédemment par `malloc`
- **valeur de retour :**
 - ▶ aucune valeur (un `void`)



Exemple

```
...  
ptab = (int*) malloc(nb_oct);  
...  
free(ptab);  
...
```

CHAÎNES DE CARACTERES

-
- *Notions, stockage, opérations, etc.*
 - *Fonctions prédéfinies : <stdio.h>, <string.h>*

Chaîne de caractères

Une *chaîne de caractères* est une séquence finie de caractères.

- en C, il *n'existe pas* de type chaîne de caractères prédéfini
- une chaîne de caractères est traitée comme un *tableau de caractères* se terminant par le **caractère nul** (code ASCII 0) exprimé par : `'\0'` ou simplement **0**
 - ▶ le caractère nul sert à repérer *la fin* de la chaîne
- une chaîne de n caractères occupe en mémoire un emplacement de $n + 1$ octets

tab[0] tab[1] tab[2] tab[3] tab[4] tab[5] tab[6] tab[7] tab[8] tab[9]

'I'	'S'	'M'	'I'	'N'	'2'	'0'	'1'	'7'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

Chaîne de caractères et *constantes*

- les chaînes de caractères **constantes** sont indiquées entre **guillemets**
 - ▶ par exemple : "ISMIN2017"
 - ▶ la chaîne de caractères **vide** est alors : ""



À ne pas confondre :

- 'x' : **caractère** constant (type **char**) *codé dans un octet*, qui a une valeur numérique (120 dans le code ASCII)
- "x" : **tableau de deux caractères**, le caractère 'x' et le caractère '\0', *codé dans deux octets*

Chaîne de caractères : *déclaration/initiaisation*

Tableau de char :

```
#include <stdio.h>

void main() {

    char promo[20] = "ISMIN2017";

    ...
}
```

Pointeur sur char :

```
#include <stdio.h>

void main() {

    char *promo = "ISMIN2017";

    ...
}
```

- `char promo[20] = "ISMIN2017";` /* initialisation d'un tableau de caractères, dont 10 éléments ne sont pas *explicitement* initialisés */
- `char promo[] = "ISMIN2017";` /* déclaration d'un tableau initialisé de 10 éléments */

Chaîne de caractères : *déclaration/initiaisation*

```
#include <stdio.h>

void main() {

    char *promo = "ISMIN2017";
    char nom_eleve[30];

    /* %s réclame l'adresse de la chaîne */

    printf("Ma promo est : %s", promo);

    scanf("%s", nom_eleve);
}
```

Chaîne de caractères : *déclaration/initiaisation*

Exemple : *tableaux de pointeurs sur des chaînes*

```
#include <stdio.h>

void main() {

/* déclaration d'un tableau de 7 pointeurs, chacun d'entre eux
désignant une chaîne de caractères constante */

    char *jour[7] = {"lundi", "mardi", "mercredi", "jeudi",
                     "vendredi", "samedi", "dimanche"};
    int i;

    printf("donnez un entier entre 1 et 7 : ");
    scanf("%d", &i);

    printf ("le jour numéro %d de la semaine est %s", i, jour[i-1]);
}
```

Chaîne de caractères : *fonctions prédéfinies*

- **<stdio.h>** : affichage et lecture de chaînes de caractères
- **<string.h>** : traitement de chaînes de caractères
 - ▶ copie de chaînes
 - ▶ concaténation
 - ▶ comparaison
 - ▶ recherche
 - ▶ conversions

Lien utile : www.gnu.org/software/libc/manual/html_mono/libc.html#String-and-Array-Utilities

Chaîne de caractères : *fonctions prédéfinies*

Fonctions de `<stdio.h>` (hormis `printf` et `scanf`) :

Impression et lecture de caractères

Prototype	Action
<code>int fgetc(FILE *fplot)</code>	lecture d'un char depuis un fichier
<code>int fputc(int c, FILE *fplot)</code>	écriture d'un char dans un fichier
<code>int getchar(void)</code>	lecture d'un char depuis l'entrée standard
<code>int putchar(int c)</code>	écriture d'un char sur la sortie standard
<code>char* fgets(char* s, int n, FILE *fplot)</code>	lecture d'un string depuis un fichier
<code>int* fputs(char* s, FILE *fplot)</code>	écriture d'un string dans un fichier
<code>char* gets(char* s)</code>	lecture d'un string depuis l'entrée standard
<code>int* puts(char* s)</code>	écriture d'un string sur la sortie standard



Toutes les fonctions de lecture/écriture de string dans des fichiers peuvent être utilisées pour lire/écrire au clavier. Il suffit d'utiliser comme pointeur de fichier (de type `FILE*`) le flot d'entrée standard `stdin` et de sortie standard `stdout`.

Chaîne de caractères : *fonctions prédéfinies*

```
#include <stdio.h>

/* Les fonctions printf et scanf permettent de lire ou d'afficher
simultanément plusieurs informations de type quelconque. En
revanche, gets et puts ne traitent qu'une chaîne à la fois. */

void main() {
    char nom[20], prenom[20], ville[25];

    printf("quelle est votre ville : ");
    fgets(ville, 25, stdin);
    printf("donnez votre nom et votre prénom : ");
    scanf("%s %s", nom, prenom);

    printf("bonjour cher %s %s qui habitez à ", prenom, nom);
    puts(ville);
}
```

- `scanf` : lecture de caractères ne contenant aucun caractère d'espacement (espace, tabulation, fin de ligne, etc); les espaces initiaux éventuels sont sautés
- `fgets` : lecture d'une ligne complète y compris la fin de ligne

Chaîne de caractères :

fonctions prédéfinies

Quelques fonctions de `<string.h>` :

Manipulation de chaînes de caractères

Prototype	Action
<code>int strlen(char* c)</code>	retourne la longueur de <code>c</code>
<code>char* strcpy(char* c1, char* c2)</code>	copie le string <code>c2</code> dans le string <code>c1</code> ; retourne <code>c1</code>
<code>int strcmp(char* c1, char* c2)</code>	compare <code>c1</code> et <code>c2</code> pour l'ordre lexicographique : retourne une valeur négative si <code>c1</code> est inférieure à <code>c2</code> , une valeur positive si <code>c1</code> est supérieure à <code>c2</code> , et 0 si ils sont identiques
<code>char* strcat(char* c1, char* c2)</code>	copie le string <code>c2</code> à la fin du string <code>c1</code> ; retourne <code>c1</code>
<code>char* strncpy(char* c1, char* c2, int n)</code>	copie <code>n</code> char du string <code>c2</code> dans le string <code>c1</code> ; retourne <code>c1</code>
<code>char* strchr(char* c1, char c)</code>	retourne un pointeur sur la dernière occurrence de <code>c</code> dans <code>c1</code> , et NULL si <code>c</code> n'y figure pas

Chaîne de caractères : *fontions prédéfinies*



Exemple : fonction strlen

```
char* promo = "ISMIN2017" ;  
  
strlen("bonjour"); // vaudra 7  
strlen(promo);      // vaudra 9
```



Exemple : fonction strcat

```
#include <stdio.h>  
#include <string.h>  
  
void main() {  
    char ch1[50] = "bonjour";  
    char *ch2 = " monsieur";  
    printf("avant : %s\n", ch1); // avant : bonjour  
  
    strcat(ch1, ch2) ;  
    printf("après : %s", ch1); // après : bonjour monsieur  
}
```

Chaîne de caractères : *fontions prédéfinies*



Exemple : fonction strcpy

```
#include <stdio.h>
#include <string.h>
void main() {

    char ch1[7] = "xxxxxxx";
    char ch2[20] = "ISMIN";

    strcpy(ch1, ch2);
    printf("%s", ch1); // affiche ISMINxx
}
```

Chaîne de caractères : *fontions prédéfinies*



Exemple : fonction strcmp

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

int main() {
    char mot1[50], mot2[50];
    printf("Veuillez saisir un mot (<= 49 lettres)");
    scanf("%s", mot1);
    printf("Veuillez saisir autre mot (<= 49 lettres)");
    scanf("%s", mot2);

    if (strcmp(mot1, mot2)==0)
        printf("les deux mots sont égaux");
    if (strcmp(mot1, mot2) < 0)
        printf("%s vient avant %s dans l'OA", mot1, mot2);
    if (strcmp(mot1, mot2) > 0)
        printf("%s vient après %s dans l'OA", mot1, mot2);

    return 0;
}
```

Les structures

Définition (Structure)

Une **structure** est un nouveau type de variable permettant de regrouper sous un seul nom, différents **champs** (ou **enregistrements**) de types différents ou pas.




Syntaxe :


```
struct nom_structure {  
    /* champ(s) composant(s) la structure */  
};
```

- une **structure** permet de traiter un ensemble d'informations hétérogènes comme un **tout**
- la **structure** est un **type**


Les structures : *exemples*

 Exemple 1 :

```
struct temps { // déclaration de la structure
    unsigned h; // 3 champs heures, minutes, secondes
    unsigned min; // les champs se déclarent comme des variables
    double sec; // on ne peut pas initialiser les valeurs
};
```

 Exemple 2 :

```
struct point3D {
    float x, y, z;
};
```

 Exemple 3 :

```
struct personne {
    char nom[100];
    char prenom[100];
    int annee_naissance;
};
```

Les structures



- La définition d'un *modèle de structure* ne réserve pas de variables correspondant à cette structure.
- Une fois un tel modèle défini, il est possible de déclarer des variables du type correspondant.

```
struct personne {  
    char nom[100];  
    char prenom[100];  
    int annee_naissance;  
};  
  
...  
  
struct personne p_1, p_2;
```

```
struct personne {  
    char nom[100];  
    char prenom[100];  
    int annee_naissance;  
} p_1, p_2;
```

Les structures : *initilisation*

Règles d'initialisation qui sont en vigueur pour tout type de variables

- il est possible d'initialiser explicitement une structure lors de sa déclaration
- en absence d'initialisation explicite, les structures de classe statique sont, par défaut, initialisées à 0

```
#include <stdio.h>

struct temps {
    unsigned heures;
    unsigned minutes;
    double    secondes;
};

int main() {

    struct temps t = { 7, 12, 0.16 };

    return 0;
}
```


Les structures : *utilisation*

- chaque champ d'une structure peut être manipulé comme une variable du type correspondant
- l'**accès à un champ** se réalise en faisant suivre le nom de la variable structure de l'**opérateur point** (.) suivi du nom de champ tel qu'il a été défini dans le modèle

```
#include <stdio.h>

struct temps {
    unsigned heures;
    unsigned minutes;
    double    secondes;
};

int main() {
    struct temps t;
    t.heures = 12; // affecte 12 au champ heures de la structure temps
    t.minutes = 5;
    t.secondes = 12.567;
    return 0;
}
```

Les structures : *utilisation*

En C, on peut utiliser une structure de deux manières :

- en travaillant *individuellement* sur chacun de ses champs
- en travaillant de *manière globale* sur l'ensemble de la structure

```
#include <stdio.h>

struct temps {
    unsigned heures;
    unsigned minutes;
    double    secondes;
};

int main() {
    struct temps t;
    t.heures = 10;
    t.minutes = 2;
    t.secondes = 7.5;

    return 0;
}
```

```
#include <stdio.h>

struct temps {
    unsigned heures;
    unsigned minutes;
    double    secondes;
};

int main() {
    struct temps t1, t2;
    t1.heures = 10;
    t1.minutes = 2;
    t1.secondes = 7.5;

    t2 = t1;
    return 0;
}
```

Les structures : *utilisation*



L'***affectation globale*** entre deux variables d'un même type structure *est la seule opération autorisée*, mais pas les autres (e.g. comparaisons, etc.).

L'instruction : typedef

- **typedef** permet de définir des *types synonymes*
- **typedef** ne crée pas de nouveau type à proprement parler



Exemple 1 :

```
typedef int entier ; // entier est synonyme de int
entier n, p ;        // équivalent à int n, p ;
```



Exemple 2 :

```
struct point3D {
    float x, y, z;
};
typedef struct point3D Point3D;
Point3D a, b, c;
```



Exemple 3 :

```
typedef struct {
    float x, y, z;
} Point3D;
Point3D p;
```

Imbrication de structures

■ Tableaux de structures

```
#include <stdio.h>
#define MAX_DATE 10
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;

int main () {
    Date liste_date[MAX_DATE];

    liste_date[0].jour = 15;
    liste_date[0].mois = 9;
    liste_date[0].annee = 1996;
    liste_date[4].jour = 28;
    liste_date[4].mois = 8;
    liste_date[4].annee = 1996;

    return 0;
}
```

■ Structures comportant d'autres structures

```
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;

typedef struct {
    char nom[50] ;
    char prenom[70] ;
    Date date_naissance;
} Personne;

Personne p;
p.date_naissance.annee = 1998;
```

Structures et *pointeurs*

- comme pour les autres types, il est possible de définir des pointeurs sur des structures
- l'accès aux champs à partir de son adresse se fait en utilisant l'*opérateur* -> (équivalent à (*p).)

```
typedef struct {  
    int jour;  
    int mois;  
    int annee;  
} Date;  
  
int main(){  
    Date date_actuelle;  
    Date *p;  
    p = &date_actuelle;  
  
    p->jour = 21;    // (*p).jour  
    p->mois = 10;   // (*p).mois  
    p->annee = 2017; // (*p).annee  
    return 0;  
}
```

Passage de paramètre par valeur :



Exemple :

```
#include <stdio.h>

typedef struct {
    int    x;
    float  y;
} Point2D;

void afficher_str (Point2D s) {
    printf("%d %f", s.x, s.y);
}

int main() {
    Point2D p; p.x = 10; p.y = 120.75;
    afficher_str (p);
    return 0;
}
```

Structures et *fonctions*

Passage de paramètre par adresse :



Exemple :

```
#include <stdio.h>
typedef struct {
    int x;
    int y;
} Point2D;

void initialiser_str (Point2D *s) {
    s->x = 0; s->y = 0;
}

int main() {

    Point2D p;
    p.x = 10; p.y = 120.75;
    initialiser_str(&p);

    return 0;
}
```


Structures et *fonctions*



Exemple :

```
#include <stdio.h>
typedef struct {
    int    x;
    float  y;
} Point2D;

/* la fonction suivante retourne la structure */
Point2D saisiePoint2D(){
    Point2D p; /* variable de type Point2D */
    printf("Entrez 2 coordonnées séparées par des espaces\n");
    scanf("%d %f", &p.x, &p.y); /* saisie des champs */
    return p;
}

int main() {
    Point2D p;
    p = saisiePoint2D();
    return 0;
}
```

Passage de paramètre *par adresse/ par valeur* :



- ***programmation performante*** : privilégier le passage des structures par adresse
- ***le passage par valeur peu performant*** : à cause des copies faites à chaque appel de fonction

Conseil : passer les structures par adresse, même si la fonction ne modifie pas la variable de type structure

Un programme a en général besoin de :

- **lire** des données : texte, nombres, images, sons, mesures, ...
- **sauvegarder** des résultats : texte, nombres, images, sons, signaux générés, ...

Cela se fait en lisant et en écrivant dans des **fichiers**.

Fichiers : *ouverture d'un fichier*



Syntaxe : ouverture d'un fichier

```
FILE* fopen(char* nomDuFichier, char* modeOuverture);
```

- renvoie un pointeur sur FILE



Exemple : ouverture en mode lecture seule

```
FILE *fichier; // déclaration d'un pointeur de type FILE  
fichier = fopen("nomfichier.txt", "r");
```

- un fichier peut être ouvert en différents *modes* : "r", "w", "a", "r+", "w+" "a+"

- **Où se trouve le fichier ouvert ?**
 - Dans le répertoire de travail (là où est le fichier exécutable)
- **Comment travailler sur un fichier situé ailleurs ?**
 - Fournir le chemin absolu d'accès.

Fichiers : *ouverture d'un fichier*

Les différents modes possibles pour un fichier sont :

- "r" : (*read*) mode **lecture seule**. Ouverture du fichier pour des lectures. Le fichier doit exister. Positionnement initial en début de fichier.
- "w" : (*write*) mode **écriture seule**. Ouverture du fichier pour des écritures. Le fichier est créé s'il n'existe pas, il est vidé s'il existe.
- "a" : (*append*) mode **ajout**. Ouverture du fichier pour des écritures en fin de fichier. Le fichier est créé s'il n'existe pas.
- "r+" : mode **lecture-écriture**. Ouverture pour des lectures et des écritures. Le fichier doit exister. Positionnement initial en début de fichier.
- "w+" : mode **lecture-écriture**. . Ouverture pour des lectures et des écritures. Le fichier est créé s'il n'existe pas, il est vidé s'il existe.
- "a+" : mode **lecture-écriture**. Ouverture pour des lectures et des écritures, avec toute écriture en fin de fichier. Le fichier est créé s'il n'existe pas. Pour les lectures, positionnement initial en début de fichier.

Fichiers : *fermeture d'un fichier*

Il faut toujours fermer un fichier après l'avoir utilisé afin de libérer la mémoire.



Syntaxe : fermeture d'un fichier

```
int fclose(FILE *fichier);
```

Cette fonction renvoie :

- 0 : si la fermeture a marché
- EOF : si la fermeture a échoué



Exemple :

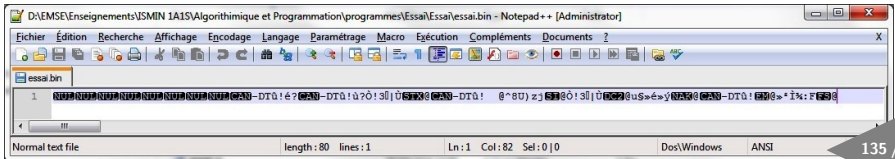
```
FILE* fichier;  
fichier = fopen("file.txt", "r+");  
if (fichier != NULL) {  
    ...  
    fclose(fichier); // on ferme le fichier qui a été ouvert  
}
```

Fichiers binaires

Un *fichier binaire* contient du code binaire

- on ne peut pas visualiser son contenu avec un éditeur de texte
- permet de stocker les données de façon *plus précise* et *plus compacte* pour coder des nombres
- les fonctions de lecture/écriture dans un fichier binaire sont :
 - ▶ `fread`
 - ▶ `fwrite`

qui lisent et écrivent des *blocs de données* sous forme binaire



Fichiers binaires : *ouverture*

On ouvre un fichier binaire avec les modes suivants :

- "rb" (*read*) : lecture
- "wb" (*write*) : écriture (le fichier est écrasé s'il existe)
- "ab" (*append*) : écriture à la fin d'un fichier existant




| Suffixe **b** : inutile avec les compilateurs actuels

Fichiers binaires : *lecture/écriture des tableaux*

Pour lire/écrire dans un fichier binaire, on lit/écrit en général les éléments d'un tableau :

- chaque élément du tableau est appelé un **bloc**
- chaque bloc possède une **taille en octets**, e.g.
 - ▶ un **char** correspond à 1 octet
 - ▶ un **float** correspond à 4 octets
 - ▶ etc.

Fichiers binaires : *écriture d'un bloc*



Syntaxe : écriture d'un bloc de données en binaire

```
int fwrite(void *source, int size_type, int nb, FILE *file);
```

- **écrit** tout un ensemble de blocs en un seul appel
- **retourne** le nombre d'éléments effectivement écrits



Exemple :

```
FILE* fichier;  
fichier = fopen("essai.bin", "w+");  
  
float tableau[3] = {1.23, 2.98, 3.45};  
fwrite(tableau, sizeof(float), 3, fichier);
```

Fichiers binaires : *lecture d'un bloc*



Syntaxe : lecture d'un bloc de données en binaire

```
int fread(void *destination, int size_type, int nb, FILE *file);
```

- **lit** tout un ensemble de bloc en un seul appel
- **retourne** le nombre d'éléments effectivement lus
- si le nombre d'éléments effectivement lus *est inférieur* à **nb**, il résulte que la fin du fichier est atteinte



Exemple :

```
FILE* fichier;  
fichier = fopen("essai.bin", "r+");  
  
float tableau[3];  
fread(tableau, sizeof(float), 3, fichier);
```

Fichiers binaires


Exemple : écriture/lecture dans un/d'un fichier binaire

```
#include <stdio.h>
#define NB_ELEMENTS 10 /* nombre d'elements des tableaux */
int main() {
    int tableau1[NB_ELEMENTS],tableau2[NB_ELEMENTS];
    for (int i=0; i<NB_ELEMENTS; i++)
        tableau1[i]=i*i;

    FILE *fichier;
    fichier = fopen("essai.bin","w+"); // ouverture en écriture
    if(fichier != NULL){
        fwrite(tableau1,sizeof(int),NB_ELEMENTS,fichier); // écriture
        fclose(fichier); // fermeture du fichier
    }
    fichier = fopen("essai.bin","r+"); // ouverture en lecture
    if(fichier != NULL){
        fread(tableau2,sizeof(int),NB_ELEMENTS,fichier); // lecture
        fclose(fichier); // fermeture du fichier
    }
    return 0;
}
```

Fichiers binaires : *stockage de structures*

On peut sauvegarder des tableaux de structures

 **Les structures ne doivent pas contenir des pointeurs.** Dans le cas contraire, c'est les *adresses* et non les *valeurs* qui seront stockées.



Exemple :

```
typedef struct {  
    int annee_naissance;  
    char nom[30];  
    char prenom[30];  
} Personne;  
...  
FILE* fichier;  
fichier = fopen("essai.bin","w+");  
  
Personne personne_Charpak = {1924,{"Charpak"},"Georges"};  
fwrite(&personne_Charpak, sizeof(Personne), 1, fichier );
```

Fichiers binaires : *se positionner dans un fichier*

- à chaque instant, un pointeur de fichier ouvert *se trouve à une position courante*
- chaque appel à `fread` ou `fwrite` *fait avancer la position courante* du nombre d'octets lus ou écrits

-
- la fonction `fseek` permet de *se positionner* dans un fichier en modifiant la position courante pour pouvoir lire ou écrire à l'endroit souhaité
 - lorsqu'on écrit sur un emplacement, la donnée qui existait éventuellement à cet emplacement *est effacée et remplacée* par la donnée écrite

Fichiers binaires : *se positionner dans un fichier*

Syntaxe :

```
int fseek(FILE *fichier, long offset, int origine);
```

La fonction **modifie la position** du pointeur **fichier** d'un nombre d'octets égal à **offset** à partir de l'**origine**. L'**origine** peut être :

- **SEEK_SET** : on se positionne par rapport au **début** du fichier
- **SEEK_END** : on se positionne par rapport à la **fin** du fichier
- **SEEK_CUR** : on se positionne par rapport à la position **courante** actuelle (position avant l'appel de **fseek**)



Exemple : Modification d'une valeur

```
/* **** */
// La fonction prend deux entiers "i" et "nouvelleValeur" et un
// pointeur "fichier" vers un fichier en paramètres, elle permet
// à l'utilisateur de modifier le (i)ème entier du fichier par
// "nouvelleValeur".
/* **** */

void ModifieNombre(int i, int nouvelleValeur, FILE *fichier) {
    int n;

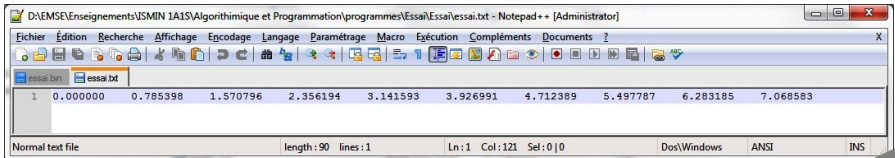
    fseek(fichier, (i-1)*sizeof(int), SEEK_SET); // positionnement
    fread(&n, sizeof(int), 1, fichier); // lecture
    printf("L'ancienne valeur vaut %d\n", n);

    fseek(fichier, -sizeof(int), SEEK_CUR); // recul d'une case
    fwrite(&nouvelleValeur, sizeof(int), 1, fichier); //écriture
}
```


Fichiers : *fichiers texte*

- un fichier texte contient du texte **ASCII**
- le contenu d'un fichier texte peut être visualisé avec un éditeur de texte
- les fonctions de lecture et écriture :
 - ▶ **fprintf**
 - ▶ **fscanf**

sont analogues aux fonctions de lecture et d'écriture de texte dans une console, **printf** et **scanf**.



Fichiers texte : *écriture et lecture*



Exemple : Lecture formatée

```
double a;  
printf("%lf",a);  
  
/* presque la même syntaxe que printf */  
fprintf(fichier,"%lf",a);
```



Exemple : Écriture formatée

```
double a;  
scanf("%lf",&a);  
  
/* presque la même syntaxe que scanf */  
fscanf(fichier,"%lf",&a);
```



Exemple : Écriture dans un fichier

```
#include <stdio.h>
#define NB_ELEMENTS 10 /* nombre d'elements du tableau */
int main() {
    int tableau[NB_ELEMENTS]; /* déclaration d'un tableau */
    for (int i=0; i<NB_ELEMENTS; i++)
        tableau[i]=i*i;

    FILE *fichier;
    /* ouverture du fichier en écriture */
    fichier = fopen("essai.txt","w");
    /* vérifier que le fichier a bien été ouvert */
    if(fichier != NULL){

        for (int i=0 ; i<NB_ELEMENTS; i++) // ecriture du tableau
            fprintf(fichier,"%d\t",tableau[i]);

        fclose(fichier); // fermeture du fichier
    }
    return 0;
}
```

Exemple : Lecture à partir d'un fichier

```
#include <stdio.h>
#define NB_ELEMENTS 10 /* nombre d'elements du tableau */
int main() {
    int tableau[NB_ELEMENTS]; /* déclaration d'un tableau */

    FILE *fichier;
    /* ouverture du fichier en écriture */
    fichier = fopen("essai.txt","w");

    /* vérifier que le fichier a bien été ouvert */
    if(fichier != NULL){

        for (int i=0; i<NB_ELEMENTS; i++)
            fscanf(fichier, "%d", &tableau[i]);

        fclose(fichier); // fermeture du fichier
    }
    return 0;
}
```

Fichiers : *binaires vs. texte*

	Fichiers textes	Fichiers binaires
Taille	peu compacte	compacte
Lisibilité utilisant un programme courant	oui	non
Lisibilité utilisant un programme spécifique	oui	oui
Lecture/Écriture par bloc	non	oui

STRUCTURES DE DONNEES

Une **structure de données** est une **organisation logique** des données permettant de *simplifier* ou d'*accélérer* leur traitement.

Listes chaînées ■
File, pile ■
Tas ■

Listes chaînées *définition*

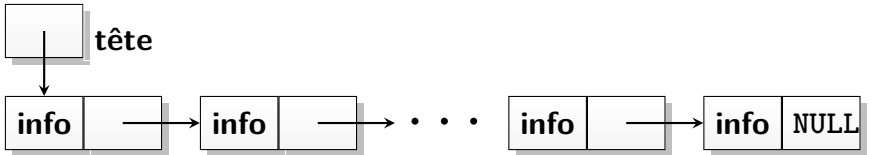
Une **liste chaînée** est une suite d'éléments, appelés *nœuds*, chacun composé :

- de l'*information* que l'on veut traiter
- d'un *lien de chaînage* qui est un pointeur vers l'élément suivant de la liste



Listes chaînées *mise en œuvre*

- chaque nœud pointe vers son successeur, sauf le dernier, qui contient le pointeur **NULL**
- l'exploitation d'une liste nécessite un pointeur *spécial*, appelé **tête**, qui pointe vers le premier élément de la liste



Listes chaînées

définition d'une liste en C

```
/* exemple de définition d'un type noeud */  
typedef struct noeud {  
    int info;  
    struct noeud *suiv;  
} T_noeud;  
  
...  
/* exemple de déclaration et initialisation d'un  
pointeur qui pointe vers un noeud */  
T_noeud *tete = NULL;  
...
```



| tete doit être initialisée à la valeur NULL

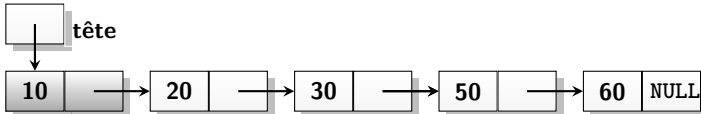
Listes chaînées

Insertion d'un nœud au début d'une liste

- Liste vide : tête

- Liste non-vide \mathcal{L} :

- Liste \mathcal{L} **après** l'insertion de :

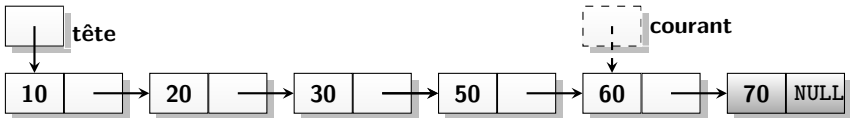
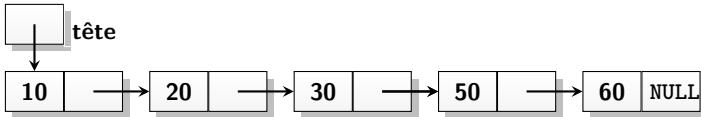


Insertion d'un nœud au début d'une liste

```
...  
T_noeud *tete = NULL;  
...  
T_noeud *nouveau;  
nouveau = (T_noeud *) malloc(sizeof(T_noeud));  
nouveau->info = 10;  
if (tete == NULL) {  
    nouveau->suiv = NULL;  
    tete = nouveau;  
}  
else {  
    nouveau->suiv = tete;  
    tete = nouveau;  
}  
...
```

Listes chaînées

Insertion d'un nœud en fin de liste



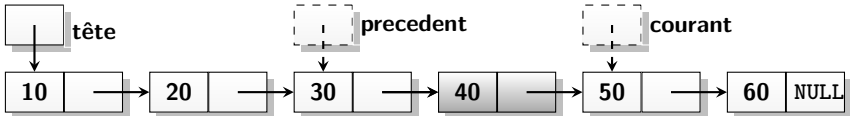
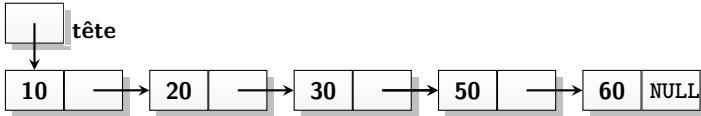
- l'insertion d'un nœud en fin de liste nécessite de repérer la fin de la liste
- parcours des éléments jusqu'à trouver le pointeur NULL

Insertion d'un nœud en fin de liste

```
...  
T_noeud *nouveau, *courant;  
nouveau = (T_noeud *) malloc(sizeof(T_noeud));  
nouveau->info = 70;  
courant = tete;  
  
while(courant->suiv != NULL)  
    courant = courant->suiv;  
  
nouveau->suiv = NULL;  
courant->suiv = nouveau;  
...
```

Listes chaînées

Insertion d'un noeud au milieu d'une liste



- l'insertion d'un nœud au milieu d'une liste nécessite de repérer l'endroit d'insertion
- parcours des éléments jusqu'à trouver cet endroit

Listes chaînées

Insertion d'un noeud au milieu d'une liste

```
...  
T_noeud *nouveau, *precedent, *courant;  
nouveau = (T_noeud *) malloc(sizeof(T_noeud));  
nouveau->info = 40;  
precedent = NULL;  
courant = tete;  
  
while(courant->info < nouveau->info) {  
    precedent = courant;  
    courant = courant->suiv;  
}  
  
nouveau->suiv = courant;  
precedent->suiv = nouveau;  
...
```

LES TAS

-
- *Définitions*
 - *Propriétés*
 - *Algorithme de tri par tas*

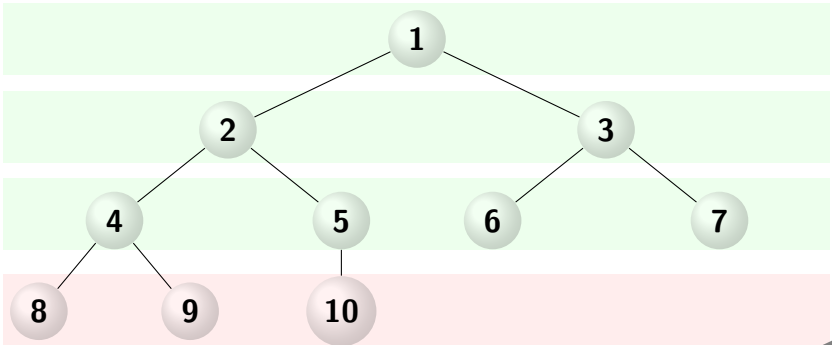
- le terme **tas** fut, au départ, inventé dans le contexte du ***tri par tas***
- il a depuis pris le sens de *mémoire récupérable* (*garbage-collected storage*)

Dans ce qui suit, le tas n'est pas une portion de mémoire récupérable, mais une structure de données.

Les tas : *notions préliminaires*

Définition

Un **arbre binaire parfait** est un arbre dont tous les niveaux sauf éventuellement le dernier sont remplis, et dans ce cas les feuilles du dernier niveau sont regroupées à gauche.



Exemple d'un arbre parfait à 10 sommets

Arbre binaire parfait A de hauteur h :

- les niveaux $0, 1, \dots, h - 1$ de A sont complets, i.e :
le niveau j contient 2^j sommets
- les j sommets du niveau h sont constitués :
 - ▶ si $j = 2q$: des 2 fils des q premiers sommets du niveau $h - 1$
 - ▶ si $j = 2q + 1$: des 2 fils des q premiers sommets du niveau $h - 1$ et du fils gauche du $(q + 1)$ ième sommet du niveau $h - 1$



- 1 Il n'existe qu'un **seul** arbre binaire parfait à n sommets (P_n)
- 2 La hauteur de (P_n) est $\lfloor \log_2 n \rfloor$

Les tas

Soit E un *ensemble* où chaque élément e est affecté d'une **priorité** $\text{priority}(e)$. L'ensemble des priorités est muni d'un ordre total \leq (\geq).

Définition

Un **tournoi** T pour $(E, \text{priority})$ est un arbre binaire sur E tel que :

$$\text{priority}(\text{père}(x)) \geq \text{priority}(x),$$

pour tout sommet x distinct de la racine.

Définition

Un **tas** est un **arbre tournoi parfait**.

Les tas : *propriétés*

Soit T un tournoi de n nœuds et N une numérotation ordonnée par une relation \leq (\geq) :

- les sous-arbres de T eux-même sont des tas
- la racine de T est un sommet de priorité minimum (maximum)
- **propriétés liées à la numérotation N** , pour chaque nœud x de T :
 - ▶ $N(fg(x)) = 2N(x)$, $N(fd(x)) = 2N(x) + 1$, où :
 - $fg(x)$: fils gauche de x
 - $fd(x)$: fils droit de x
 - ▶ $N(\text{père}(x)) = N(x) \div 2$, avec $x \neq \text{racine}(T)$
 - ▶ $2N(x) > n \Rightarrow x$ est une feuille
 - ▶ $N(\text{DERNIERE_FEUILLE}(T)) = n$
- en utilisant la numérotation N , ***un tas peut être représenté par un tableau***

Les tas : *tableau*

Les tas sont généralement représentés et manipulés sous la forme d'un tableau :

- tout *tableau* peut être considérée comme *arbre binaire parfait*
- les *tas* sont des cas particuliers de tableaux $T[n]$, i.e. ceux qui respectent la *propriété des tas max (min)*

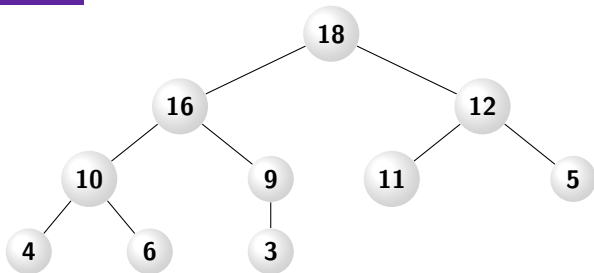
$$T[\text{père}(i)] \geq T[i] \quad \left(T[\text{père}(i)] \leq T[i] \right), \forall i = \overline{1, n}$$

ou autrement dit,

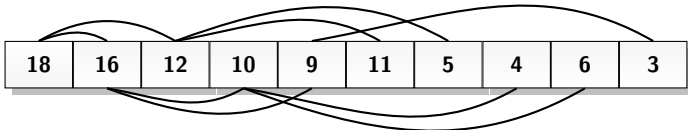
un père est toujours plus grand (petit) que ses deux fils.

Les tas : *tableau*

Plusieurs représentation d'un tas :



Tas vu comme un arbre binaire parfait



Tas vu comme un tableau

Les tas : *tableau*

Filiation et paternité : soit $T = \{18, 16, 12, 10, 9, 11, 5, 4, 6, 3\}$ avec $n = 10$

racine : $T[1]$

18	16	12	10	9	11	5	4	6	3
----	----	----	----	---	----	---	---	---	---

fil gauche du nœud $T[i] : T[2i]$

18	16	12	10	9	11	5	4	6	3
----	----	----	----	---	----	---	---	---	---

fil droit du nœud $T[i] : T[2i + 1]$

18	16	12	10	9	11	5	4	6	3
----	----	----	----	---	----	---	---	---	---

feuilles : $T[i]$ avec $2i > n$

18	16	12	10	9	11	5	4	6	3
----	----	----	----	---	----	---	---	---	---

père du nœud $T[i] : T[i \div 2]$

18	16	12	10	9	11	5	4	6	3
----	----	----	----	---	----	---	---	---	---

Les tas

Quelques opérations de base :

- **CréerTas(E)** : crée et retourne un nouveau tas E sans élément
- **Insérer(e , p , E)** : insère l'élément e de priorité p dans le tas E
- **SupprimerMin(E)** : supprime dans le tas E un élément de priorité minimale
- **Min(E)** : renvoie un élément de priorité minimum de E
- etc.

Tri par tas (*Heap sort*)

- algorithme proposé par **J. W. J. Williams** en 1964
- tri non récursif
- **étapes de base** :
 - ▶ convertir un tableau en tas
 - ▶ trier le tas en entassant l'élément *max* dans un temps $(n - 1)$ fois



Williams (J. W. J.). Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7 : 347–348, 1964.

? Exercice (Tri par tas) 1/3 :

Insertion d'un élément e de dans un tableau T :

Algorithm 9 : Insérer(e, T)

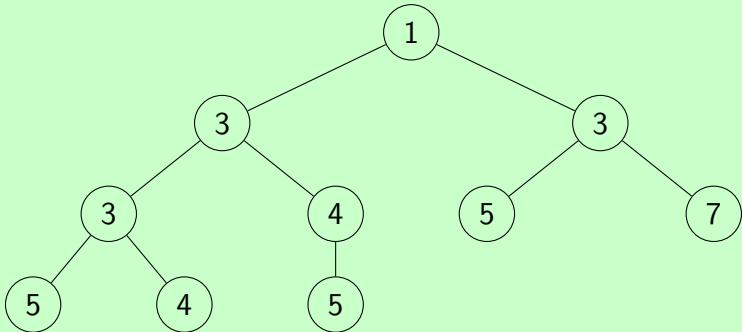
```
1:  $x = \text{Créer\_Dernière\_Feuille}(e, T)$ 
2: while ( $x \neq \text{racine}(T)$  et  $\text{priority}(x) < \text{priority}(\text{père}(x))$ )
   do
3:    $\text{Echanger}(x, \text{père}(x))$ 
4:    $x = \text{père}(x)$ 
5: end while
```

1 Quelle est la complexité de cette procédure ?

Les tas : *exercices*

? Exercice (Tri par tas) 2/3 :

- 2 Considérons le cas d'un tas min. Appliquez la procédure d'insertion à l'arbre suivant en insérant l'élément 2 :



? Exercice (Tri par tas) 3/3 :

- 3 Proposez une procédure pour la suppression d'un élément de priorité minimale. Quelle est sa complexité ?
- 4 Appliquez la procédure à l'arbre obtenu à la question 2.

Supposons maintenant qu'un seul élément du tas est mal placé. Les deux sous-arbres de cet éléments sont des tas. La procédure qui consiste à corriger cette violation s'appelle le **tamissage**.

- 6 Proposez une procédure pour le tamissage. Quelle est sa complexité ?
- 7 Proposez un algorithme pour la création d'un tas. Quelle est sa complexité ?
- 8 Proposez un algorithme de tri en se basant sur les tas.

MODULARITÉ

-
- *Généralités*
 - *Organisation d'un module*
 - *Utilisation d'un module*

Modularité : *généralités*

Un programme de taille ou de complexité importantes peut être développé sur *plusieurs fichiers* ou ***modules***.

Intérêt du découpage modulaire :

- **réutilisation** : un même module peut être utilisé dans plusieurs programmes différents (e.g. `stdio`, etc.)
- **compilation séparée**
- **meilleure lisibilité**
- **dissimulation des implémentations**
- **favorisation du travail en équipe**
- etc.

Modularité : *modules prédéfinis*

- le langage C offre un certain nombre de *modules prédéfinis*
 - ▶ `stdio` : entrées/sorties
 - ▶ `string` : chaînes de caractères
 - ▶ etc.
- la bonne utilisation de ces modules nécessite une directive `#include` appropriée :



Exemple

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


Modularité : *exemple*

programme mon_appli.c

```
//directives, prototypes, ...
int main() {
    // ...
}
float perimetre(...) {
    // ...
}
float aire(float b, float h){
    // ...
}
int combinaison(int p, int q){
    // ...
}
int factorielle(int n){
    // ...
}
```

module combinatoire

module combinatoire

```
//directives, prototypes, ...
int combinaison(int p, int q){
    // ...
}
int factorielle(int n){
    // ...
}
```

programme principale

module mon_appli

```
//directives, prototypes, ...
int main() {
    // ...
}
float perimetre(...) {
    // ...
}
float aire(float b, float h) {
    // ...
}
```

Modularité : *organisation d'un module*

Tout module est constitué de deux fichiers :

- une **interface** ou *spécification* des fonctionnalités offertes :
nom_fichier.h (appelé aussi *fichier d'en-tête* ou *header file*)
- une **implémentation** des fonctionnalités spécifiées dans
l'interface : **nom_fichier.c**

Modularité :

*contenu d'un fichier *.h*

Définitions *externes*, i.e. celles utilisables par d'autres modules clients :

- directives `#define`
- définition des types (`typedef`, etc.)
- prototypes des fonctions



Exemple

```
/* contenu du fichier combi.h */  
int combinaison(int p, int q);  
int factorielle(int n);
```

`combi.h` doit être inclus dans tous les modules utilisant les fonctions `combinaison` ou `factorielle`

Modularité :

*contenu d'un fichier *.c*

- inclusion de fichiers d'en-tête
- définition de variables globales
- définitions *propres* au module, i.e. non-accessibles aux autres modules :
 - ▶ constantes **internes** au module
 - ▶ types **internes** au module
 - ▶ prototypes des fonctions **internes** au module
- définition des fonctions

Modularité : *exemple complet*

mon_appli.c

```
#include <stdio.h>
#include "combi.h"
float perimetre(float a, float b,
               float c);
float aire(float b, float h);

int main() {
    printf("Perimetre du triangle =
           %d", triangle (3,4,5));
    printf("Aire du triangle =
           %d", triangle (3,4));
    int c = combinaison (5,9);
    return 0;
}

float perimetre(float a, float b,
               float c) {
    return a + b + c;
}

float aire(float b, float h) {
    return b*h/2;
}
```

combi.h

```
int combinaison(int p, int q);
int factorielle(int n);
```

combi.c

```
#include "combi.h"

int combinaison(int p, int q){
    return Fact(p)/((Fact(q)*Fact(p-q)));
}

int factorielle(int n) {
    int i, fa;
    fa = 1;
    for (i = 2; i <= n; i++)
        fa = fa * i;
    return(fa);
}
```

Modularité : *utilisation d'un module*

-
- Tout fichier utilisant les fonctionnalités d'un module doit inclure le fichier d'en-tête de ce module.
 - Ceci permet au compilateur de vérifier la bonne utilisation de ces fonctionnalités.

Génération d'applications

- le compilateur **C** a pour rôle de traduire un programme composé d'un ou de plusieurs fichiers sources **C** en un programme binaire exécutable
- l'utilitaire **make**
 - ▶ permet de gérer des projets constitués de fichiers sources multiples
 - ▶ facilite la compilation séparée de modules et va jusqu'à produire l'exécutable

L'utilitaire make : *tâches*

- lecture d'un fichier spécial, appelé **makefile**, créé auparavant
- comparaison date/heure de modification de chaque fichier **objet** avec date/heure de modification des fichiers **sources** et des fichiers d'en-tête (en fonction des listes de dépendances indiquées)
- ré-compilation si nécessaire (en cas de modification)
- comparaison date/heure de modification des fichiers **objet** avec date/heure de modification du fichier **exécutable** (en fonction de la liste de dépendances indiquée)
- édition de liens si nécessaire

Modularité : *exemple complet*

mon_appli.c

```
#include <stdio.h>
#include "combi.h"
float perimetre(float a, float b,
float c);
float aire(float b, float h);

int main() {
    printf("Perimetre du triangle =
    %d", triangle (3,4,5));
    printf("Aire du triangle =
    %d", triangle (3,4));
    int c = combinaison (5,9);
    return 0;
}

float perimetre(float a, float b,
float c) {
    return a + b + c;
}

float aire(float b, float h) {
    return b*h/2;
}
```

combi.h

```
int combinaison(int p, int q);
int factorielle(int n);
```

combi.c

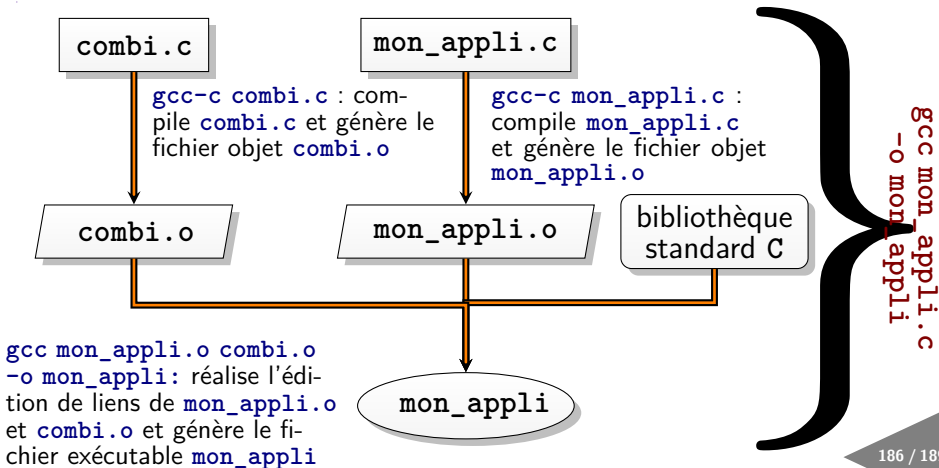
```
#include "combi.h"

int combinaison(int p, int q){
    return Fact(p)/((Fact(q)*Fact(p-q)));
}

int factorielle(int n) {
    int i, fa;
    fa = 1;
    for (i = 2; i <= n; i++)
        fa = fa * i;
    return(fa);
}
```

Génération d'applications

Compilation séparée :



Exemple de fichier : makefile

l'édition de liens n'est lancée que si `mon_appli.o` ou `combi.o` ont été modifiés

tabulation obligatoire et non des espaces

`mon_appli` dépend de `mon_appli.o` et `combi.o`

```
# mon premier makefile

mon_appli.o: mon_appli.c combi.h
    gcc -c mon_appli.c

combi.o: combi.c combi.h
    gcc -c combi.c

mon_appli: mon_appli.o combi.o
    gcc mon_appli.o combi.o -o mon_appli
```

■ `make mon_appli` génère le fichier exécutable `mon_appli`

Bonnes habitudes de programmation

- La compilation doit se faire **sans erreur ni avertissement (warning)**.
- Une mauvaise indentation n'est pas sanctionnée. Les programmes doivent cependant être **correctement indentés**.
- Les **noms** des fonctions, variables, etc. doivent être :
 - ▶ **parlants**
 - ▶ **pertinents**
 - ▶ **cohérents** (conventions **uniformes**).
- Les programmes doivent **être commentés** de façon **constructive**.

<https://www.kernel.org/doc/Documentation/CodingStyle>


MINES
Saint-Étienne

1816
2016



 **INSPIRING
INNOVATION**
SINCE 1816