Algorithmes et programmation
Langage C

Tatiana Aubonnet

Organisation

- 19 séances de 2h
- A travers les concepts de langage C on introduira:
  - **les structures de données séquentielles et récursives** telles que les tables, listes, fichier, piles, …
  - **l'implantation de ces structures** sous forme statique, puis dynamique.
  - **les algorithmes** (ex. algorithme de tri tas)
- Composées alternativement de:
  - Cours
  - Travaux Dirigés et Travaux Pratiques (Linux)
- Contrôle continu: les TPs sont individuels et notés.

Bibliographie

- **B. Kernighan, D. Ritchie**
  Le langage C (2 ème édition) , Masson, 1994
- **A. Aho, J. Hopcrotf, J. Ullmann**
  Structures de données et algorithmes, InterEditions, 1987
- **R. Sedgewick**
  Algorithmes en langage C, Inter Editions, 1991
- **P. Dax**
  Langage C (7 ème édition), Eyrolles, 1992
- **P. Aitken et Brandley L. Jones**
  Le langage C – Nouvelle Edition, Campus Press (Le programmeur), 2000

Introduction

- Pourquoi le langage C ?
- Etude de langage
  - La syntaxe
  - La sémantique
- Cycle de vie d'un logiciel
- La chaîne de production d'un programme
- Ecrire un programme en C
- Les instructions de base
  - L'instruction conditionnelle ***if***
  - Boucle ***for***
  - Boucle ***while***
- Un exemple

Pourquoi le langage C ? (1)

- **Avantages :**

  Le langage C mis au point au début des années 70 par **D.Ritchie** et **B.W.Kernighan** est:

  - Un langage simple qui manipule des objets élémentaires (caractères, nombres, adresses)
  - Portable (écrit à la norme ANSI) sur les systèmes disposant d'un compilateur C : Windows, UNIX, VMS (système des VAX) ou l'OS des mainframes IBM.
  - Introduit la notion de **prototype** ou **déclaration des fonctions**
  - Comporte des instructions et des structures de haut niveau (contrairement à l'assembleur par exemple)

Pourquoi le langage C ? (2)

- **Inconvénients :**
  - Gestion des entrées/sorties
  - Messages d'erreurs incompréhensibles
  - Laisse passer les erreurs syntaxique et sémantique avec sa propre interprétation

Etude de langage: la syntaxe

- **Syntaxe**: n.f du gr. *sun,* **avec** et *taxis,* **ordre**

- Partie de la grammaire qui traite de la fonction et de la disposition des mots dans la phrase. *

- Ex: « J'étudie le langage C » est syntaxiquement *correct.*

  «J'étudie langage C » est syntaxiquement *incorrect.*

---

* D'après le *Larousse Elémentaire*

Etude de langage: la sémantique

- **Sémantique** : n.f du préfixe gr. *sêma*, **signe, symbole** et du siffixe *-ique*, **qui a rapport à**
- Etude des mots et des phrases considérés dans leurs significations. *
- Ex: «La terre est ronde comme une orange» sémantiquement *correct*.

  «La terre est bleue comme une orange» sémantiquement *incorrect*.

---

* D'après le *Nouveau Larousse Universel*

Cycle de vie d'un logiciel (1)

- 7 étapes:
  1. Spécification d'un cahier de charges
  2. Conception d'un algorithme de résolution
  3. Analyse du programme sous forme de spécifications
     - Spécifications externes
     - Spécifications internes
  4. Codage dans le langage choisi

Cycle de vie d'un logiciel (2)

5. Test en mise au point
6. Recette
   - Logiciel
   - Manuel d'utilisation
   - Jeu de données utilisées
7. L'évolution ultérieure du programme

La chaîne de production (1)

- On écrit un programme en C dans un fichier reconnu par le nom de la norme: **nom_du fichier .c**
- **xEmacs**: pour écrire le programme
  - Ouvrir sous Linux (UNIX) : **xemacs &**
- Compilation (traduire en langage plus proche de la machine pour obtenir un exécutable)

  A la fin de la compilation : fichier « objet » .o
- Sous Unix la compilation du fichier qui s'appelle nom.c s'obtient à l'aide de l'instruction **gcc**

gcc -o nom.c

option

Cette instruction produit un fichier objet: nom.o

- Quand il y a des erreurs le fichier nom.o n'aboutit pas.

La chaîne de production (2)

- En C on a des bibliothèques pour gérer les entrées/sorties
- 3 phases:
  - **Compilation**
  - **Edition des liens**
  - **Exécution**
- La commande gcc nom.c effectue la compilation puis l'édition de liens.
  - produit un fichier exécutable nommé par défaut **a.out**
- Pour ranger l'exécutable dans un fichier autre que a.out, on écrit:

  gcc –o nom nom.c
- Pour exécuter le programme on écrit: **nom_du_fichier**

La chaîne de production (3)

- Tout ceci constitue une chaîne de production en C:
  - Ecrire le programme dans un fichier
  - Compilation
  - L'édition des liens
  - Exécution

Ecrire un programme en C (1)

- /* commentaire ignoré par un compilateur */

  ou // généralement réservé pour une ligne de code que l'on veut désactiver temporairement.
- Un programme commence par: **# include <stdio.h>** - une directive de pré-compilation
- Tout programme écrit en C contient exactement une fonction «main».
- Un programme est constitué de fonctions.
- La syntaxe pour une fonction:
  - type de résultat
  - l'identificateur d'une fonction
  - (liste des types de paramètres)

```
int main (void)  // fonction principal
{
}
```

délimitent la fonction

il n'y a pas de paramètres

Ecrire un programme en C (2)

- Un exemple: écrire bonjour à l'écran

L'en-tête →

Le corps de la fonction →

```
int main (void)  // fonction principal
{
   printf (« bonjour \ n »)  ;
   return 0 ; // provoque la fin du programme
}
```

Ecrire un programme en C (3)

- D'une façon générale, un programme en C se présente de la façon suivante:
  - #..... déclaration de pré-compilation
  - déclaration ou définition d'objets globaux
  - (variables globales)
  - types, prototypes de fonctions
- Des fonctions se présentent de la manière suivante:
  - L'en-tête
  - Corps: déclarations, définitions, objets locaux
- On ne trouve pas de fonctions dans les fonctions!

L'instruction conditionnelle

- Représenter la partie de programme suivante:

  Si une certaine condition est vérifiée,

  alors exécution de l'instruction 1 et 1 bis

  sinon - l'instruction 2

- Structure de l'aiguillage:

```
if (condition booléenne)
{
      instr1 ;
      instr 1 bis ;
}
 else
      instr 2 ;
```

L'instruction: for (boucle pour)

- Représenter la partie de programme suivante:

Pour i variant de 1 à n par pas de 1,

faire instruction 1 et instruction 2 sinon instruction 3.

```
int i ;   // entier sur (2 ou 4 octets), valeur à calculer
int n ;
for (i=1 ;  i<=n ;  i=i+1)
```

attribution condition évolution de compteur

```
{
   instr 1 ;
   instr 2 ;
}
   instr 3 ;
```

Instruction : while (boucle tant que)

- Représenter la partie de programme suivante:

  tant que (condition) faire instruction 1.

```
while (condition)
{
  instr 1 ;
  instr 1 bis ;
}
```

Boucle « do .. while »

Faire

instruction 1 ;

instruction 2 ;

…

tant que (la condition soit faite)

```
do
{
        inst 1 ;
        inst 2 ;
}
        while (condition);
```

Exercice 1 : Conversion Fahrenheit en Celsius (1)

```
# include <stdio.h>
int main(void)
{
   int debut, fin, intervalle ;
   int celsius, farh ;
   debut = 0 ;              // borne inférieure supérieure
   fin = 120 ;              // borne supérieure
   intervalle = 5 ;         // par pas de 5
```

Exercice 1 : Conversion Fahrenheit en Celsius (2)

```
    farh = debut ;
    while (farh <= fin)
    {
      celsius = 5 * (farh - 32)/9 ;
      printf (« % d », celsius) ;
      farh = farh + intervalle ;
    }
      return 0 ;
}
```

Types de données, les tableaux statiques, les fonctions

Sommaire

- Identificateurs
- Types de données
- Opérateurs
- Tableaux statiques
- Tableaux statiques à plusieurs dimensions
- Fonctions

Identificateurs

- Permettent de nommer des entités qui seront manipulées par l'ordinateur
  - Variables
  - Fonctions
  - Types
  - Constantes …
- Les identificateurs noms des entités sont écrit à l'aide de caractères suivants:
  - A, b, …z, A, B, … Z
  - 10 chiffres : 0, 1, …. 9
- L'identificateur doit commencer par une lettre
- On ne peut pas utiliser un mot clé de langage comme identificateur (ex: if, main…).

Types de base

- Toute variable possède un type
- Les types de base sont les suivants:
  - Types « entiers »
    - Char
    - Short int ou short
    - Int
    - Long int ou long
  - Types « réels »
    - Float
    - Double

Types de données

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32768 à 32767
unsigned short int	Entier court non signé	2	0 à 65535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32768 à 32767 -2147483647 à 2147483647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65535 0 à 4294967295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	2	0 à 4 294 967 295
float	flottant (réel)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
double	flottant double	8	$1.7*10^{-308}$ à $1.7*10^{308}$
long double	flottant double long	10	$3.4*10^{-4932}$ à $3.4*10^{4932}$

Les opérateurs arithmétiques

- Addition: +
- Soustraction: -
- Multiplication: *
- Division: /
- Reste modulo: %
  - La division entre entier donne un coefficient entier

    Ex: int n ;

    n = 7/2 ;

    le résultat est entier: 3
  - Pour avoir 3.5, faire la conversion

    float x ;

    x = (float) 7/2 ;

    la division est une division réelle.

Liste de comparaisons

- <,
- >,
- <= pour ≤
- >= pour ≥
- = = pour l'égalité
- ! pour la négation
- ! = pour la différence

Les caractères

Pour considérer un caractère en tant que tel on met les apostrophes avant le caractère et après.

- Ex1:

```
char c ;
….
c = 'm' ; ou  c = 109 ;  // ou caractère m est attribué à 109
… printf («exemple %d» , c) ;  // exemple 109 à l'écran
```

imprime la valeur

```
printf(«exemple %c» , c) ;    // exemple m à l'écran
```

imprime le caractère

- Ex 2:

```
int n, m ;
 n=2 ; m=3 ;
 printf(«bla%d bli %d blu» , n ,m) ;    // bla 2 bli 3 blu
```

Expressions logiques

- ou : II
- et : &&
- non: ! (placé avant l'expression à nier)
  - Exemple:

```
….
int a ;
if ((a<3) && (a>1))
```

- Toute expression possède une valeur.
- Toute **valeur non nulle** est logiquement **assimilée à « vrai »**
- La **valeur nulle** (0) est **assimilée a « faux »**
  - Exemple: l'affectation n=3, possède la valeur 3

Exercices

- Quelle est la valeur de l'expression n==4 dans le programme suivant:

A. int n=3 ;

```
….
if (n==4) … réponse?
if (n) …
n=3;
if (n!=0) …
if (1) …
```

B. int n=3 ;

```
….
if (n=4) …  n=2 ;
printf(« n=%d », n) ;
```

Type de données complexes

- Les variables, telles que nous les avons vues, ne permettent de stocker qu'une seule donnée à la fois.
- Le langage C propose deux types de structures:
  - **les tableaux**: permettant de stocker plusieurs données de même type.
  - **les structures**: pouvant contenir des données hétérogènes.

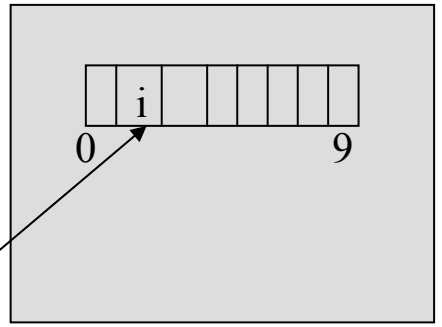
Tableaux statiques (1)

- Un tableau est une collection de **variables de même type** rangées consécutivement (pour nous) dans l'espace mémoire.

- Pour réserver la place mémoire nécessaire pour un tableau (statique) de 10 caractères:

```
int tableau [10] ;
```

type de case — nombre de cases souhaitées



accès directe à la place mémoire

Tableaux statiques (2)

- Un tableau statique est une entité constante: il ne peut pas apparaître (seul) à gauche d'un signe d'affectation.

  - Ex: int tableau 1[10] ;
    int tableau 2 [10] ;
    On ne peut pas écrire: tableau 1 = tableau 2, car tableau est une entité constante.

- Les cases d'un tableau de 10 cases sont numérotés de 0 à 9.

- On accède à la case d'indice i (i≥0) par tableau [i].

Exercice: tableaux statiques

- Ecrire un programme (sans entrées/sorties) qui range les 10 premiers entiers non nuls dans un tableau, puis faire la somme de cases de ce tableau.

```
int main(void)
{
    int table[10] ;  // définir un tableau
    int  compteur ;
    int somme ;
    for (compteur = 0 ; compteur <10 ; compteur ++) // initialiser le tableau
    table[compteur] = compteur + 1 ;
    somme = 0 ;
    for (compteur = 0 ; compteur <10 ; compteur ++)  // calculer la somme
    somme+ = table[compteur] ;
    return 0 ;
}
```

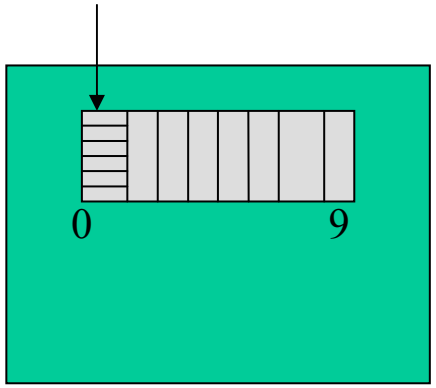
- On peut initialiser le tableau au moment de sa définition
  - Ex: int table[ ] = {1, 2, 3, 4} ; // 4 cases avec des valeurs 1, 2, 3, 4

Un tableau non dimensionné

Tableaux statiques multidimensionnels

- Un tableau (de réels) à 2 dimensions est en fait un tableau à 1 dimension dont les cases contient des tableaux à 1 dimension.

  - Ex: float matrice [10] [10] ;



- Accéder à la case ligne i colonne j:

```
matrice [ i ] [ j ] ;     // donne élément de ligne i et colonne j
```

Fonctions: concepts (1)

- Une fonction permet de regrouper des instructions qui, logiquement, forment un bloc qu'on peut séparer du reste.
  - Ex: supposons que on veille écrire un programme de tri insertion
- On peut décomposer le programme en 3 parties:
  - Lecture de données
  - Traitement de données (ici le tri)
  - Affichage du résultat

Fonctions: concepts (2)

- Une fonction est constituée d'une en-tête et d'un corps:
  - L'en-tête précise le type de résultat renvoyé
  - L'identificateur: le nom de la fonction
  - Une liste de paramètres entre les parenthèses
    - Ex: int main (void) ou int main( )
  - Le corps de la fonction est une liste d'instructions entourées d'accolades.

Rq: on ne peut pas définir une fonction dans le corps d'une autre fonction.

Fonctions: variables locales

- Les variables définies à l'intérieur d'une fonction s'appellent des **variables locales** (à cette fonction)

  Elle ne sont connues qu'a l'intérieur de cette fonction.

  - Ex 1: void premiere_fonction (void)

```
{
    int i ;
    int n ;
     … (instructions)
     seconde_fonction ( ) ; // une fonction peut appeler une autre fonction
                                (récursivité croisé), l'exécution est suspendue
 }
```

  - Ex 2: void seconde_fonction (int n)

```
{
    int i ;
     … (instructions)
 }
```

Fonctions: variables globales, prototypes (1)

- Il est souhaitable de définir les variables globales.
- Des **variables globales** peuvent être définies en programme C: il faut pour cela les déclarer en dehors des fonctions (au début de programme). Elles sont alors connues et manipulables en début de programme:
- **Toute entité (fonction, variable …) doit être déclarée (on lui a attribué un identificateur) avant d'être manipulée** ⟶ **prototype de fonction**

Fonctions: variables globales, prototypes (2)

➢Structure d'un programme :

```
#include …
#define …  // directives de pré-compilation
variables globales
prototypes des fonctions (en-tête suivi d'un ;)
fonction 1 … en-tête;
        {
        corps
        }
fonction 2 …
 ….
dont la fonction main (au début ou à la fin du programme)
```

Adresses, le tri par insertion, identificateurs de formats

Sommaire

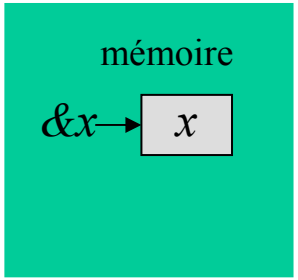
- La notion de pointeur
- Adresses
- Passage de paramètres à une fonction
- Tri insertion spécification externe
- Tri insertion spécification interne
  - Ecrire la fonction lecture
  - Ecrire la fonction affichage
  - Ecrire la fonction tri
- Identificateurs de format

La notion de pointeur

- Un pointeur est une variable contenant **l'adresse** d'une autre variable d'un type donné.
- Les pointeurs ont un grand nombre d'intérêts, ils permettent de:
  - de manipuler de façon simple des données pouvant être importantes (au lieu de passer à une fonction un élément très grand on pourra par exemple lui fournir un pointeur vers cet élément...)
  - de définir des **structures dynamiques**, c'est-à-dire qui évoluent au cours du temps (par opposition aux tableaux de données statiques, dont la taille est figée)
  - de **créer des structures chaînées**, c'est-à-dire comportant des maillons

Adresses (1)

- Une variable *x* est associée à l'emplacement en mémoire.
- Cette variable *x* est placée à l'adresse *x ;* toute variable possède une adresse que l'*x* peut manipuler explicitement.



- Déclaration des adresses:

On peut manipuler des variables de type adresse

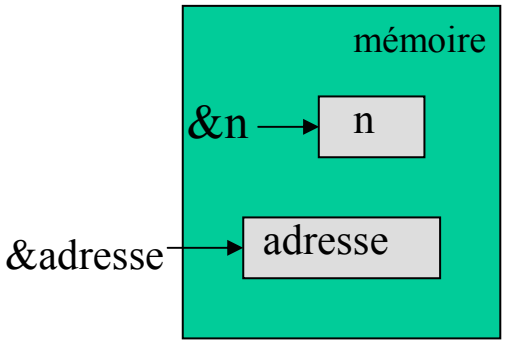
à l'aide de la syntaxe suivante:

Type_de_l'objet_dont_on manipule_adresse * identificateur_de_l'adresse ;

Ex: **int * adresse ;** // ceci définit une variable de nom « adresse » qui donne accès à un entier

int n ;

adresse = &n ;



Adresses (2)

On peut accéder au contenu d'une case en mémoire grâce au pointeur « adresse » par ***adresse**.

- Ex:

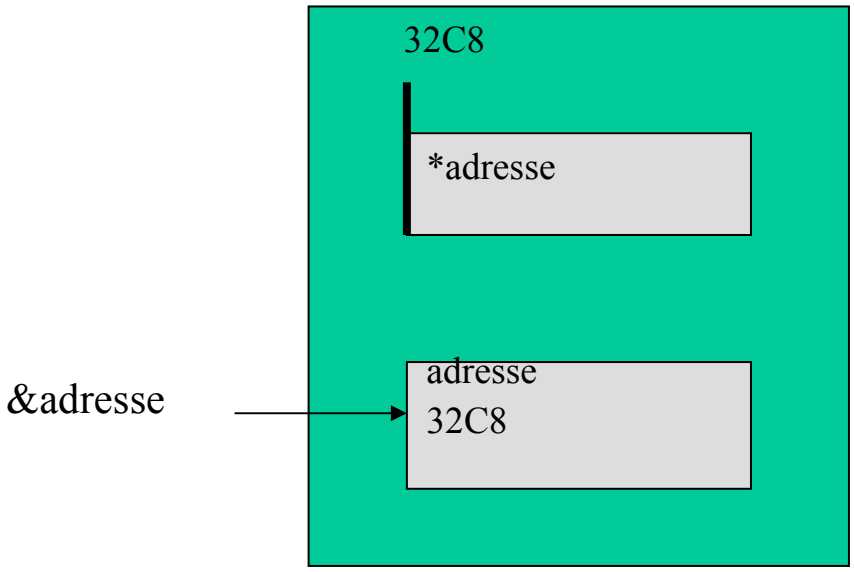
  int n ;

  *n – incorrecte

  &n - l'adresse de n

  *&n - c'est n

  &*n – incorrecte



Passage de paramètres à une fonction

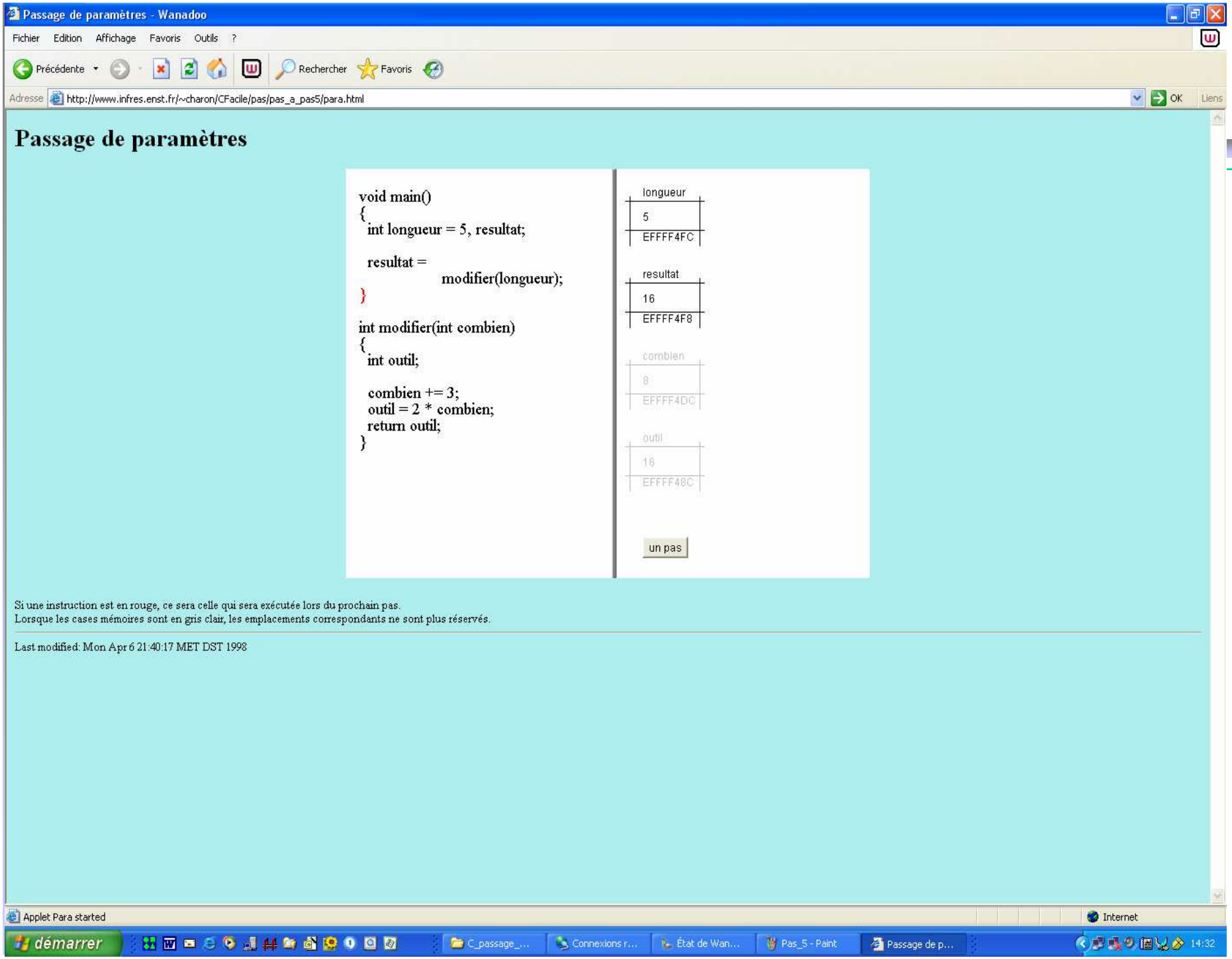
Une fonction en C ne peut jamais modifier la valeur de ces paramètres.

Si on veut modifier une variable *var* à l'aide d'une fonction *f*, on doit passer l'adresse de var *(var)* comme paramètre de *f*.

- Ex: la fonction scanf doit pouvoir changer la valeur de var (d'un certain type), on l'utilise sous la forme:
- Scanf (« % d ,&var»)

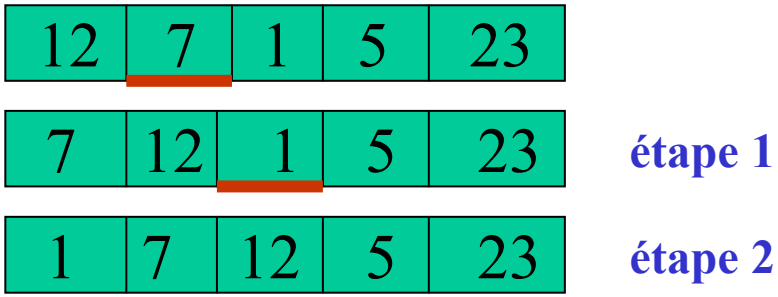
Lettre associé au type de var

- fflush(stdin) ; ⟶ vider la mémoire tampon d'entrée ;
- fflush(stdout) ;⟶ écrire le contenu de la mémoire tampon de sortie

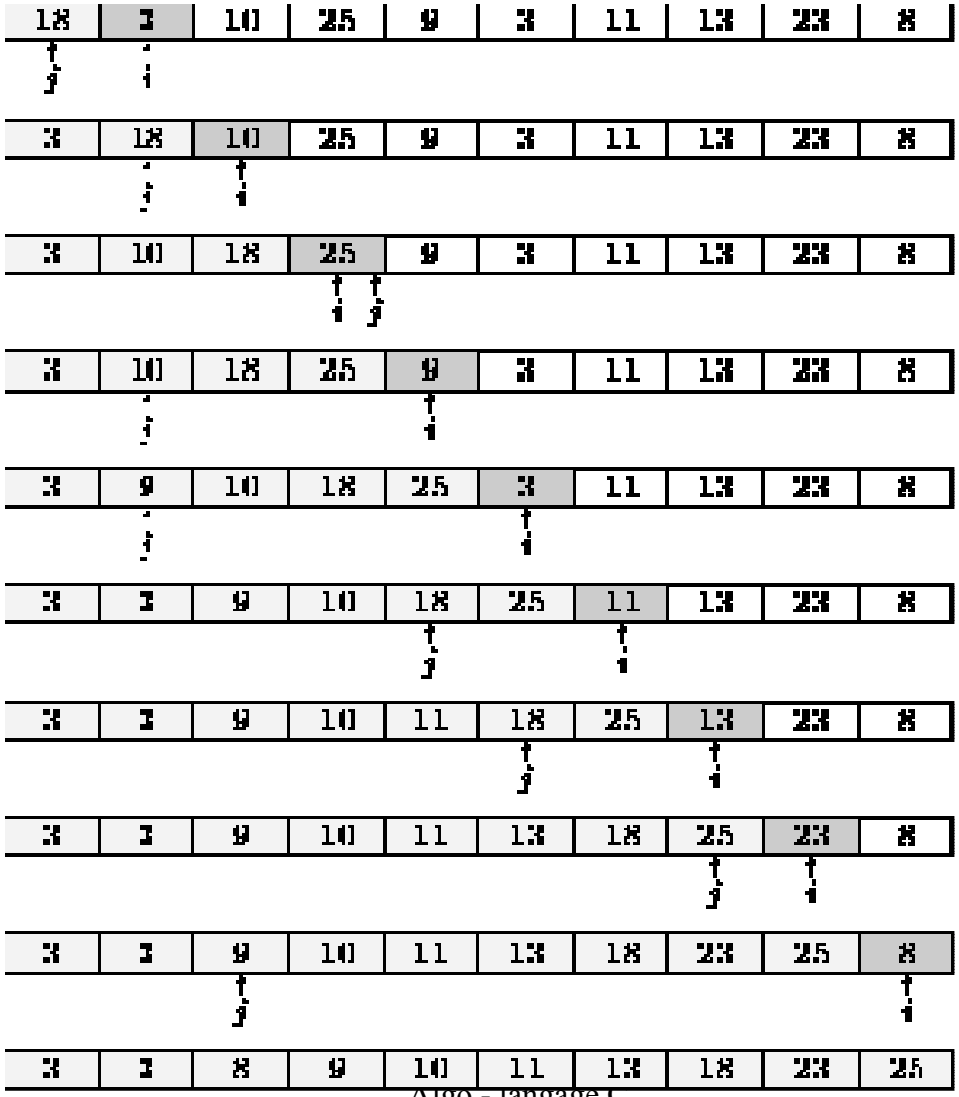

Passage de paramètres - Wanadoo
Fichier Edition Affichage Favoris Outils ?
Précédente
Rechercher
Favoris
Adresse
http://www.infres.enst.fr/~charon/CFacile/pas/pas_a_pas5/para.html
OK
Liens
Passage de paramètres
void main()
{
int longueur = 5, resultat;
resultat =
modifier(longueur);
}
int modifier(int combien)
{
int outil;
combien += 3;
outil = 2 * combien;
return outil;
}
longueur
5
EFFFF4FC
resultat
16
EFFFF4F8
combien
8
EFFFF4DC
outil
16
EFFFF48C
un pas
Si une instruction est en rouge, ce sera celle qui sera exécutée lors du prochain pas.
Lorsque les cases mémoires sont en gris clair, les emplacements correspondants ne sont plus réservés.
Last modified: Mon Apr 6 21:40:17 MET DST 1998
Applet Para started
Internet
démarrer
C_passage_...
Connexions r...
État de Wan...
Pas_5 - Paint
Passage de p...
14:32

Tri insertion : principe

- On considère que la gauche du tableau est déjà triée.
- On considère **l'élément suivant la partie triée** et on le fait "descendre" à sa place en le comparant à l'élément qui le précède.
- On commence par "sauver" l'élément à mettre en place dans une variable "cle".
- On pousse ensuite d'une position vers la droite les éléments qui le précèdent et qui sont plus grands que lui.
- On le range dans le tableau lorsqu'on a trouvé sa place.
- Attention: le premier indice d'un tableau est 0 !



Tri insertion : exemple



Tri insertion : analyse

- Analyse :
  - Lecture – fonction lecture ;
  - Traitement – fonction tri ;
  - Affichage – fonction affichage

  Ces fonctions ne reçoivent aucun résultat et ne manipulent aucun paramètres

Tri insertion: spécification externe

- Préciser les variables globales:
  - Un entier N: nombre d'entiers à trier
  - Un tableau de 100 entiers appelé T
- Préciser les fonctions:
  - La fonction lecture doit lire N et les N entiers à trier, elle les place dans T.
  - La fonction tri doit trier par ordre croissant les éléments de T à l'aide de tri insertion.
  - La fonction affichage doit écrire le contenu de tableau T trié.

Tri insertion: spécification interne

De la fonction lecture

- Lire N ;
- Tant que N ≤ 1 ou N > 100, relire N
- Pour i variant de 1 à N, lire T(i).

De la fonction affichage

- Pour i variant de 1 à N écrire T(i) à l'écran.

De la fonction tri:

- Pour i variant de 2 à N, faire : aux ⟵ T(i)

  aux reçoit T(i) : mémoriser ce que doit être écrasé

  j ⟵ i-1 *initialiser J*

  tant que (j ≥ 1er indice) et (T(j) > aux) faire

  T(j+1) ⟵ T(j)

  j ⟵ j-1

  T(j+1) ⟵ aux *Placer la valeur aux dans T(j+1)*

Ecrire la fonction lecture

```
// Fonction lecture
void lecture (void)
    {
        int i ;
        printf(« Introduisez le nombre d'entier N \ n") ;
        scanf("%d",&N) ;
        while((N <= 1) II (N>100))
    {
        printf(« N doit etre entre 2 et 100 inclut \ n ») ;
        scanf("%d",&N) ;
    }
        printf(« Donner les %d valeurs » , N) ;
        for(i=0 ; i <= N ; i++)
        scanf("%d",&T[ i ]) ;
        }
```

Ecrire la fonction tri

```
// Fonction tri
void ecrire (void)
 {
    int i, j, aux ; // variables locales
    for (i=1 ; i<N +1 ; i++)
 {
    aux = T[i] ;
    j= i-1 ;
    while((j>=0) && T[j] > aux))
     { T[j+1] = T[j] ;
      j = j-1 ; // ou j - -
       }
    T[j+1] = aux ;
 }
 }
```

Ecrire la fonction affichage

```
// Fonction affichage
void affichage(void)
{
 int i ;
 printf («Voici les données triées:\n ») ;
 for (i=0 ; i<N ; i++)
 printf(« %5d»,T[ i ]) ;
 printf(« \n ») ;
 }
```

Fichier complet

```
// Fichier complet
# include <stdio.h>
int N ;  // variables globales
int T[100] ; // variables globales

void lecture(void) ;          // prototype de fonction
void trier(void) ;            // prototype de fonction
void affichage(void) ;        //prototype de fonction

void main(void)
{
    printf (« Bonjour ce programme… ») ;
    lecture ( ) ;
    trier ( ) ;
    affichage ( ) ;
    return 0 ;
}
```

Puis les 3 fonctions

Exercice

Changer le programme supposant que on ne déclare pas N dans les variables globales.

Ecrire la fonction lecture

```
// Fonction lecture
 int lecture (void)
      {
         int N;
         int i ;
         printf(« Introduisez le nombre d'entier N \ n") ;
         scanf("%d",&N) ;
         while((N <= 1) II (N>100))
      {
         printf(« N doit etre entre 2 et 100 inclut \ n ») ;
         scanf("%d",&N) ;
      }
         printf(« Donner les %d valeurs » , N) ;
         for(i=0 ; i <= N ; i++)
         scanf("%d",&T[ i ]) ;
         return N;
      }
```

Ecrire la fonction tri

```
// Fonction tri
void ecrire (int N)
 {
    int i, j, aux ; // variables locales
    for (i=1 ; i<N +1 ; i++)
 {
    aux = T[i] ;
    j= i-1 ;
    while((j>=0) && T[j] > aux))
     { T[j+1] = T[j] ;
      j = j-1 ; // ou j - -
       }
    T[j+1] = aux ;
 }
 }
```

Ecrire la fonction affichage

```
// Fonction affichage
void affichage(int N)
{
 int i ;
 printf («Voici les données triées:\n ») ;
 for (i=0 ; i<N ; i++)
 printf(« %5d»,T[ i ]) ;
 printf(« \n ») ;
 }
```

Fichier complet

```
// Fichier complet
# include <stdio.h>
int T[100] ; // variables globales

int lecture(void) ;  // prototype de fonction
void trier(int N) ;              // prototype de fonction
void affichage(int N) ;        //prototype de fonction

void main(void)
{
    int N;
    printf (« Bonjour ce programme… ») ;
    N= lecture ( ) ;
    trier ( N ) ;
    affichage (N ) ;
    return 0 ;
}
```

Puis les 3 fonctions

Identificateurs de format

Pour printf et scanf

- entier:
  - %d: valeur entière
  - %5d: écrit la variable entière sur au moins 5 caractères calés à droit
  - %-8d: idem calé à gauche
  - %ld: pour les « long int ».
- réel:
  - %f:
  - %7.4f: pour écrire une float sur au moins 7 caractères dont 4 après la virgule
- Caractère
  - %c
- Chaîne de caractères
  - %s
- Double
  - %lf

Calculer n!

Cahier des charges
calculer n! pour une valeur de n fournie par l'utilisateur et comprise entre 1 et 7. Si la valeur entière fournie par l'utilisateur n'est pas dans cet intervalle, il lui sera demandé une nouvelle valeur.

```
#include <stdio.h>

int main()
{
  int n;
  int i, fact=1;

   printf("entrez une valeur entiere positive "
     "inferieure ou egale a 7 : ");
  scanf("%d",&n);
  while((n<=0)||(n>7))
    {
      if (n<=0)
    printf("j'ai demande une valeur positive, "
           "redonnez la valeur : ");
      else
    printf("j'ai demande une valeur inferieure a 8, "
           "redonnez la valeur : ");
      scanf("%d",&n);
    }
  for(i=2;i<=n;++i) fact*=i;
  printf("la valeur de %d! est %d\n",n,fact);
  return 0;
}
```

Allocation dynamique de la place mémoire, structures de données

Sommaire

- Tableaux dynamiques
- Allocation dynamique de la place mémoire
- Comment libérer la place mémoire ?
- Structures de données
- Types énumérés
- Constantes
- Dichotomie

Allocation dynamique de la place mémoire - Tableaux dynamiques (1)

- Un tableau dynamique est un tableau dont le nombre de cases est fixe pendant l'exécution du programme. Formellement, un tableau dynamique est une variable de type adresse.

  - Ex: int * table ;

    Ceci déclare un tableau sans le définir, c'est-à-dire sans lui attribuer la place mémoire

Allocation dynamique de la place mémoire - Tableaux dynamiques (2)

- L'attribution de place mémoire de manière dynamique se fait à l'aide du: **malloc**

  - Ex1: Pour créer (définir) un tableau de 10 entier auquel on accède par table:

```
table = (int *) malloc(10*sizeof(int)) ;
```

↑

conversion de type

Une matrice d'entier bidimensionnelle ?

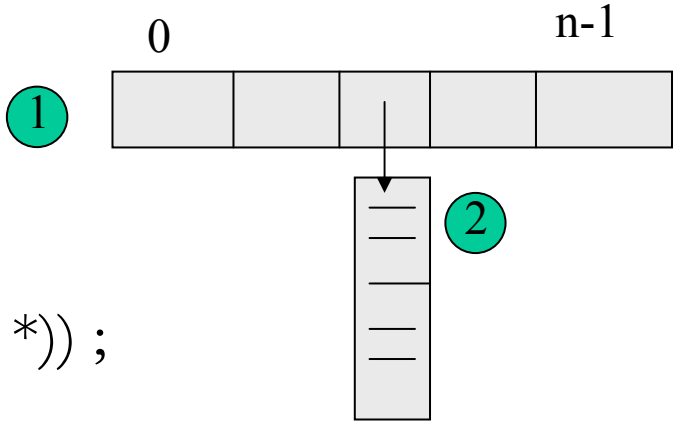
Tableaux dynamiques

- Ex 2: Créer tableau T $n \times m$ (n et m sont des variables entiers)

  en deux temps:

1- tableau horizontal

```
int ** T ;
T= (int **) malloc(10 * sizeof(int *)) ;
int i ;
for(i=0 ; i<n ; i++)
T[i] = (int *) malloc(n * sizeof(int)) ;
```



2- tableau vertical

Allocation dynamique de mémoire (1)

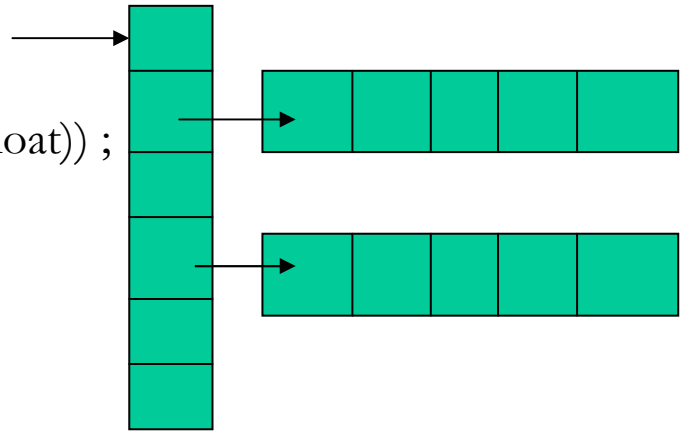
- **Ex: Matrice rectangulaire dynamique**
  - **Déclarer la matrice:**

```
float ** matrice ;      // déclarer la matrice
int n, m ;              // dimension de la matrice
int i, j ;              // indice de la matrice
```

  - **Définir la matrice (*attribuer la place mémoire*)**

```
matrice = (float **) malloc (n * sizeof (float*)) ;
if(matrice = = NULL) printf (« Plus de place!\n ») ;
else
{
for(i=0 ; i<n ; i++)
{
matrice [i] = (float *) malloc (m * sizeof (float)) ;
if(matrice[i] ==NULL) …
else
}
…
}
```

ou i+1



Allocation dynamique de mémoire (2)

- Pour utiliser malloc, on doit inclure le fichier d'en-tête **stdlib.h** au début du fichier:

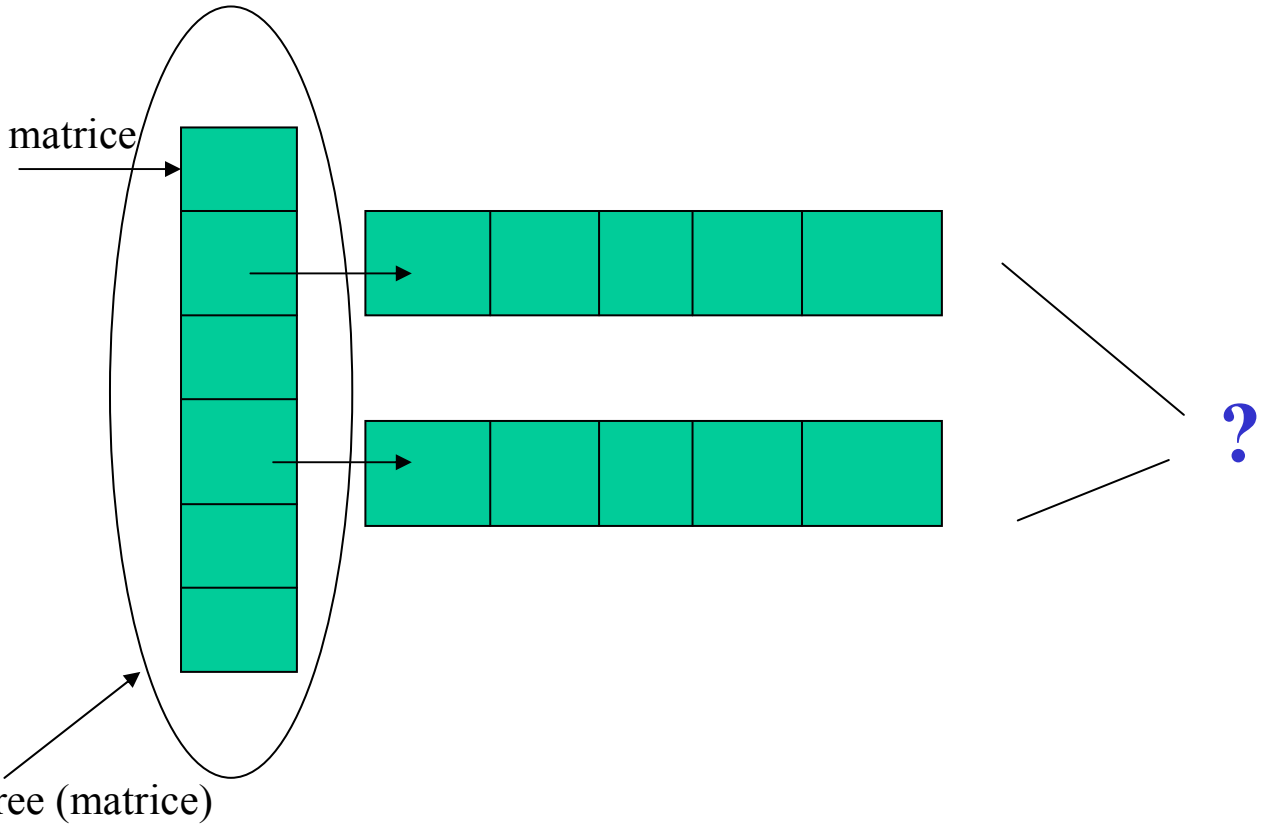
  # include <stdlib.h>

- Pour libérer la place mémoire on utilise la fonction ***free***
  - Exemple

```
int * tableau ;
int n ;
……
scanf (« %d », n) ; // lire n
…..
tableau  = (int*) malloc (n * sizeof (int)) ;
if (tableau = = NULL) …
free (tableau) ;  // libère la place à laquelle on accède grâce au tableau
```

Libérer la place mémoire (1)

matrice = (int *) malloc ((n *m) * sizeof (int)) ;



Libérer la place mémoire

- Pour libérer la place mémoire attribuée à un tableau bidimensionnel matrice:

```
for (i=0 ; i<n ; i ++)
free (matrice [ i ]) ;
free (matrice) ;
```

- Règle: Si on veut libérer toute la place mémoire on doit **appeler free autant de fois qu'on a appelé malloc**.

Fonctions

- **malloc( )** "Memory ALLOCation", pour réserver une zone.
- **free( )** pour libérer une zone préalablement allouée.
- **calloc( )** pour réserver une zone et l'initialiser à 0.
- **realloc( )** pour réallouer une zone avec une taille différente.

Réallocation dynamique

- La fonction **realloc( )** :

  void * realloc (void * base , size_t t)
- La fonction **realloc( )** redimensionne un bloc mémoire donné à la fonction via son adresse **base** avec une nouvelle taille **t.**
- La fonction réalloue le bloc mémoire tout en gardant le contenu de ce qui se trouvait dans le bloc précédent.
- La fonction ne fait qu'un changement de taille. Ceci est utile pour un tableau dynamique : en effet, on peut ajouter ou enlever une case à la fin du tableau sans le modifier.

Réallocation dynamique

- Exemple pour illustrer comment fonctionne la fonction **realloc ( ):**

```
int * tabentier;
/* Création d'un tableau de 3 entiers */

tabentier = calloc ( 3 , sizeof(int) );

tabentier[0] = 1;
tabentier[1] = 2;
tabentier[2] = 3;

/* Ajout d'un element au tableau */

tabentier = realloc (tabentier, 4 * sizeof(int) );

tabentier[3] = 4;

for ( i = 0 ; i < 3 ; i++ )
{
     printf(" tabentier[%d] = %d \n", i , tabentier[i] );
}
```

Structures (1)

- Les structures (en C: **struct**) permettent de définir des structures de données « complexes », c'est-à-dire de constituer plusieurs sous données non nécessairement homogènes.
  - Ex: Définir un type représentant des élèves et caractérise par:
    - nom (chaîne de caractères)
    - prénom (chaîne de caractères)
    - note (entier)

  On définit un type élève par:

```
struct eleve
{
  char nom [20] ;
  char prenom [20] ;
  int note ;
} ;
```

- On a ainsi définit un nouveau type: le type « struct eleve ».

Structures (2)

- On peut utiliser ce type comme tout autre type :
  - Ex:

```
struct matiere   // pour définir les notes attribuées aux élèves
{
  char nom_matrice [20] ;
  struct eleve * tableau_de_notes ;
} ;
```

- Pour définir une variable de type struct élève :
  - Ex:

```
struct eleve cet_eleve ;
```

Structures (3)

- Pour accéder à un membre de la variable:

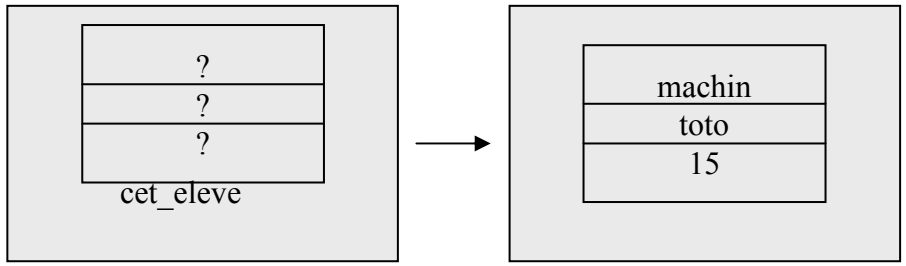
  nom_variable.identificateur_membre

  - Ex:

    cet_eleve.nom = « machin » ;

    cet_eleve.prenom = « toto » ;

    cet_eleve.note = « 15 » ;



attribution de la place mémoire

Structures (4)

- Opérations sur les structures :
  - **&** appliqué à une variable de type structure
    - Ex: & cet_eleve ;
  - **sizeof** « » à un type structure ou à une variable de type structure
  - **affectation** (=) entre entités de même type structure
    - Ex: struct eleve cet_eleve ;

      struct eleve autre_eleve ; autre_eleve = cet_eleve ;
  - mais on ne peut pas faire directement de comparaison (égalité, différence) avec des variables de type structure
    - Ex. if(autre_eleve = = cet_eleve) ;

Structures (5)

- Initialisation :
  - Ex:

```
struct date
{
  int jour ;      // décrit le type
  int mois ;
  int annee ;
}
  aujourd_hui = {22,10,2007} ;    // définit les valeurs
```

Variable de type struct date

Structures: renommage de type

- On peut définir des noms de type synonyme à l'aide de ***typedef***

  Ex: typedef struct data Date ;

  le type à renommer — ***nouveau nom de type***

- Strictement identique:

  Date aujourd_hui ;

  équivalent struct date aujourd_hui ;

  Ex: typedef char booléen ; // pour définir les booléens

Types énumérés (1)

- Un type énuméré est un type que l'on crée en définissant de manière exhaustive les valeurs qu'une variable de ce type peut prendre.
- Ceci se fait à l'aide du mot ***enum***

Syntaxe:

enum *identificateur* {liste des constantes symboliques};

Ex: enum booleen {faux, vrai};

nouveau type *valeurs possibles*

- Toute variable de tout type énuméré est en faite de type entier (int).
- Par défaut, la première valeur de la liste énumérée vaut 0, la suivante 1, etc.
- On peut changer ces valeur par défaut :
  - Ex: enum jours {lundi = 1, mardi, mercredi = 5} ;

incrémenté par 1

Types énumérés (2)

- Si un élément de la liste est initialisé explicitement par un signe d'affectation =, les valeurs des éléments suivants progressent à partir de cette valeur

  Ex: enum mois {janvier = 1, fevrier, mars, avril}

  incrémenté par 1

  donne: fevrier = 2, …. avril = 4

- Les énumérations sont un moyen pratique d'associer des noms à des valeurs constantes comme *#define*, mais avec l'avantage de la génération automatique des valeurs.

Constantes (1)

- Une constante est une variable dont la valeur est inchangeable lors de l'exécution d'un programme.
- Exemple: comment changer la précision de pi ?

```
….
float pi ;
float x ;
pi =3.14 ;
….
if (x = =3.14)
…
```

Constantes (2)

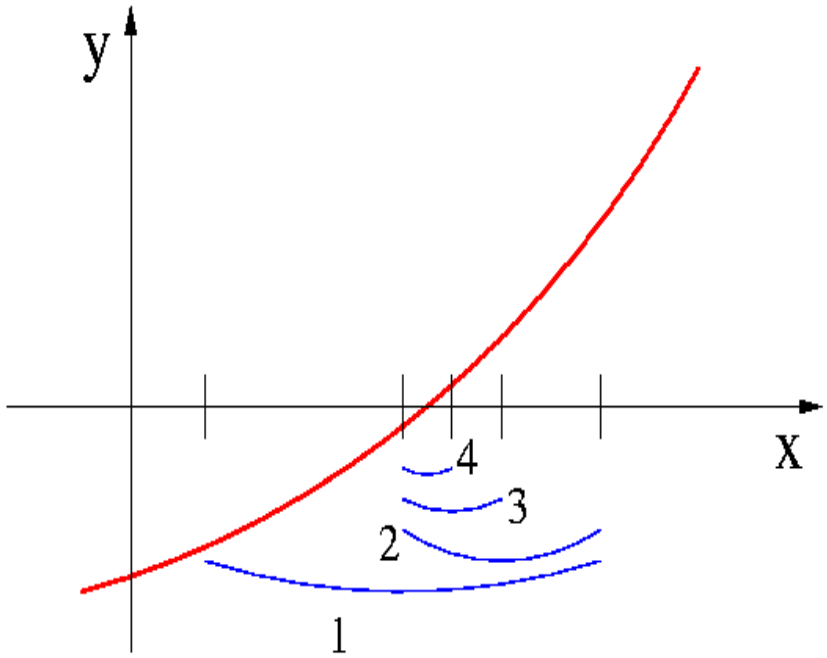
- Définir une constante: utiliser la déclaration de pré-compilation (**# define**), qui permet de remplacer toutes les occurrences du mot qui le suit par la valeur immédiatement derrière elle.

```
# define pi 3.1415927
....
if (x = = pi) …
```

- Constantes entières : type énuméré à 1 valeur
  - Ex: enum constante = { 2 } ;

Méthode de la dichotomie

« couper en deux » en grec



- Il s'agit, à partir d'un encadrement dans un intervalle [*a*, *b*] d'une racine de l'équation $f(x)=0$:
  - de réduire la taille de cet intervalle
  - jusqu'à la précision voulue, tout en gardant la racine à l'intérieur.
- Cela suppose que la fonction change de signe au passage de la racine.
- La réduction de l'intervalle consiste à le diviser par deux, en remplaçant une des deux extrémités de l'intervalle par le point milieu.

Spécifications externes (1)

- Principe: quand la fonction est continue

  (ie a <= b implique f(a)<=f(b)), alors on peut résoudre par approximations successives du résultat (sur des nombres flottants), sous la forme d'une dichotomie (ie "séparation en 2").
- La solution peut être :
  - récursive
  - mais on peut la traduire facilement en itération (le compilateur le fait d'ailleurs automatiquement).

Spécifications externes (2)

- Données:
  - **l'équation**
  - **2 floats a** et **b** tels que la solution x de l'équation f(x)=0 est tel que c est compris dans [a,b].
  - **la précision** "epsilon" (positive) voulue pour le résultat.

- Résultat: une approximation c du nombre x tel que f(x)=0 (f(c)=0 +/- epsilon).

Spécifications internes

- Approx(a,b:float)
  - On considère l'intervalle [a,b] dont on prend le milieu: c:=(a+b)/2.
  - Si f(c) est compris dans [0-epsilon, 0+epsilon], alors retourner c.
  - Sinon:

    si f(c) > 0+espilon alors retourner la valeur de Approx(a,c)

    sinon (on a f(c) < 0), retourner la valeur de Approx(c,b)

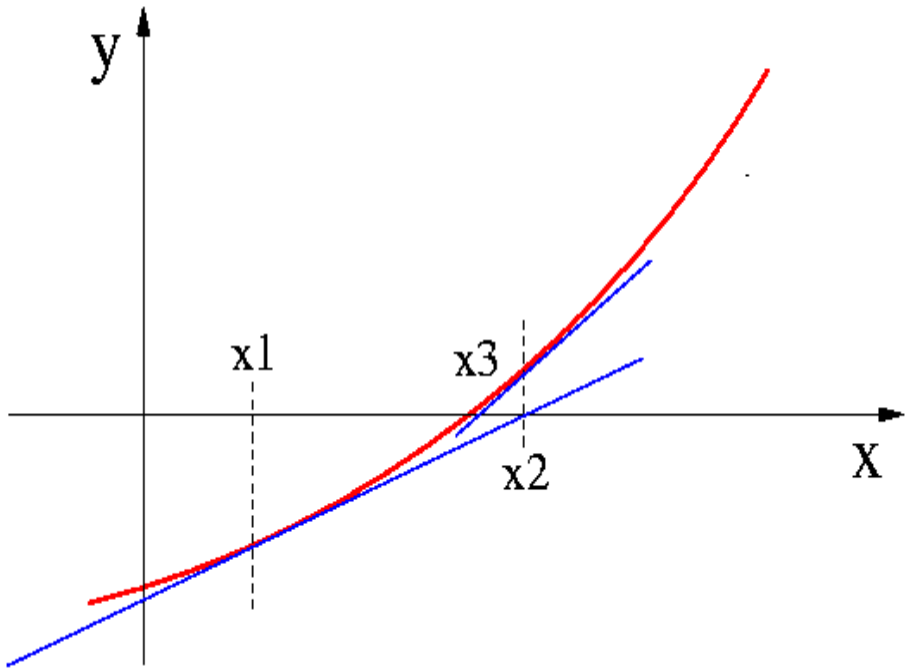
- On sait que x est dans [a,b], et on a divise l'intervalle [a,b] en 2 pour rechercher la valeur dans un intervalle plus petit [a,c] ou [c,b], jusqu'a ce que l'écart entre c et x soit suffisamment petit.

Avantages /Limites

- **Avantage :**
  - de cette méthode est qu'elle converge toujours.
- **Limites**
  - Pour un encadrement du zéro recherché il faut avoir une idée de la valeur approximative du zéro cherché.
  - La convergence est lente. En effet, pour la division de l'intervalle en deux, la précision de l'encadrement est multipliée par deux à chaque itération.
  - Il existe des méthodes bien plus performantes.

Méthode de Newton-Raphson

- Elle utilise les variations de la fonction dont on cherche le zéro.
- En partant d'un point, on trace la tangente à la courbe que l'on suit jusqu'à intercepter l'axe des abscisses.
- Le nouveau point obtenu est ainsi plus proche du zéro de la fonction, et on recommence l'opération jusqu'à la précision souhaitée.



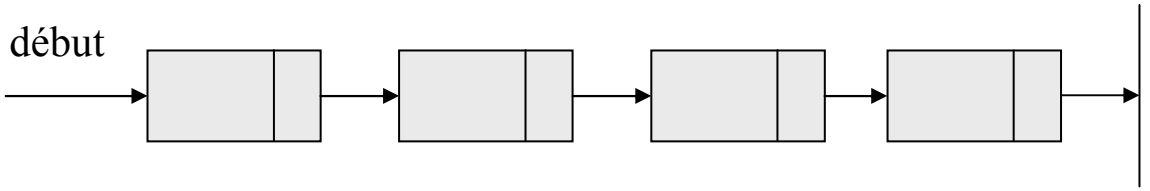
Listes chaînées, fichiers

Sommaire

- Listes chaînées
  - Ajouter un premier élément
  - Ajouter un élément en fin de liste
  - Rechercher un élément dans la liste chaînée
  - Exercices
- Fichiers

Listes chaînées (1)

- **Une liste chaînée** est une collection de maillons (en C des structures) tels que chaque maillon contient l'adresse du maillon suivant
  - Représentation:



- Il est nécessaire de conserver une "trace" du premier enregistrement afin de pouvoir accéder aux autres, c'est pourquoi un pointeur vers le premier élément de la liste est indispensable. Ce pointeur est appelé *pointeur de debut*.

- D'autre part, étant donné que le dernier enregistrement ne pointe vers rien, il est nécessaire de donner à son pointeur la valeur *NULL*

Listes chaînées (2)

- Ex: Que fait ce morceau du programme?

```
…
int n, m;
int * pointeur;
n=5;
pointeur = &n;
m = * pointeur;
m ++;
pointeur = &m;
*pointeur =3;
…
```

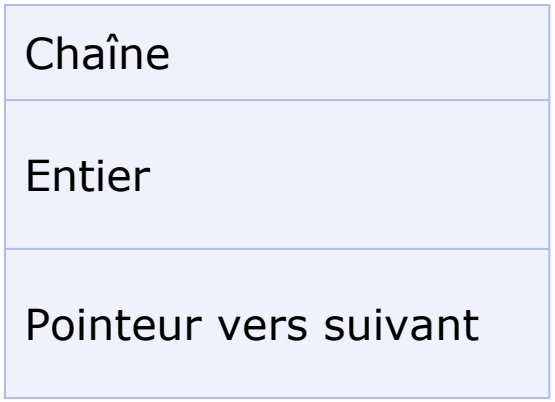
Listes chaînées (3)

- La définition du type permettant de manipuler les maillons d'une liste chaînée s'obtient à l'aide des structures:
  - Ex:

```
struct maillon
{
char chaine [16]
int donnee;
struct maillon * suivant;
};
typedef struct maillon * adr_maillon;
```

Listes chaînées (4)

- On représente généralement cette structure de la manière suivante :



- Il faut un pointeur pour définir le début.

  On peut définir les variables de type adresse de structure maillon par :

```
struct maillon * debut;
```

Ajout d'un premier élément

- Une fois la structure et les pointeurs définis, il est possible d'ajouter un premier maillon à la liste chaînée, puis de l'affecter au pointeur **Debut**. Pour cela il est nécessaire:

  - d'allouer la mémoire nécessaire au nouveau maillon grâce à la fonction *malloc*, selon la syntaxe suivante:

    Nouveau = (struct maillon*)malloc(sizeof(struct maillon));

  - d'assigner au champ "pointeur" du nouveau maillon, la valeur du pointeur vers le maillon de début

    Nouveau->pSuivant = Debut;

  - définir le nouveau maillon comme maillon de début

    Debut = Nouveau;

Le symbole "->"

- **Le symbole "->" ("tiret" suivi de "supérieur") est un raccourci.**
- **Si debut est l'adresse d'une structure contenant un champ entier valeur, l'expression :**
  - **debut->valeur est synonyme de (*debut).valeur**
  - ***debut est la structure sur laquelle pointe debut (ce qui signifie "dont debut est l'adresse") : l'expression (*debut).valeur représente donc le champ valeur de cette structure .**
- **Attention :**
- **Ecrire *debut.valeur**
  n'aurait pas de sens car le "." est prioritaire sur le symbole "*".

Ajout d'un élément en fin de liste (1)

- L'ajout d'un élément à la fin de la liste chaînée est similaire, à la différence près qu'il faut définir un pointeur (appelé généralement *pointeur courant*) afin de parcourir la liste jusqu'à atteindre le dernier maillon (celui dont le pointeur possède la valeur *NULL).*
- Les étapes à suivre sont :
  - la définition d'un pointeur ***courant*** :

    struct maillon * pCourant;
  - le parcours de la liste chaînée jusqu'au dernier noeud:

```
if (Debut != NULL)
{
pCourant = Debut;
while (pCourant->pSuivant != NULL) pCourant = pCourant->pSuivant;
}
```

Ajout d'un élément en fin de liste (2)

- Les étapes à suivre (suite):
  - l'allocation de mémoire pour le nouvel élément:

    Nouveau = (struct maillon*)malloc(sizeof(struct maillon));

  - faire pointer le pointeur courant vers le nouveau noeud, et le nouveau noeud vers NULL:

    Courant->pSuivant = Nouveau;

    Nouveau->pSuivant = NULL;

Exemple

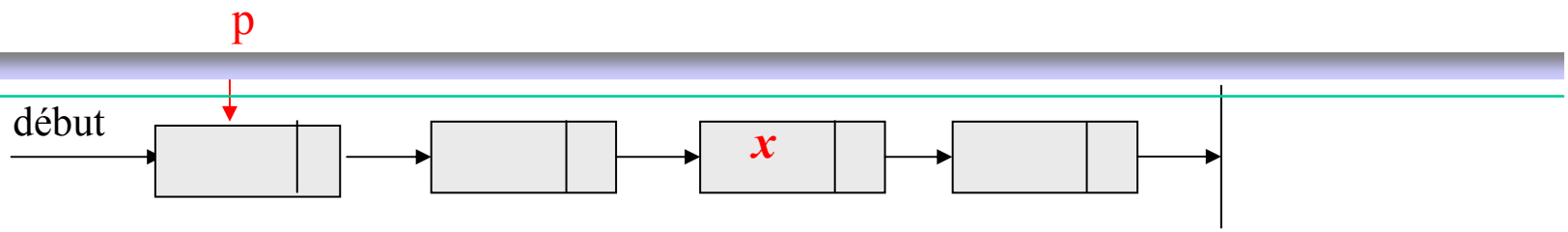
- Un exemple de manipulation d'une liste chaînée pour acquérir des données, en ajoutant les données les unes après les autres selon une **structure de pile**.

```
…
struct maillon
{
    struct maillon donnee;
    struct maillon * suivant;
};
struct maillon * debut ;
char reponse;
struct maillon * pointeur;
printf (« … »);
do
{
    printf(« donnee? »);
pointeur = (struct maillon *) malloc (sezeof(struct maillon));
```

Exemple (suite)

```
//on teste la réservation de la place mémoire
if (pointeur ! = NULL)
{
   scanf( « %d », &(* pointeur).donnee));
   (* pointeur).suivant = debut; // ou pointeur -> suivant = debut;
   debut = pointeur;
}
else …..;
prinf ( « encore ? »)
ffluch (stdin); // vider la mémoire tampon
scanf (« %c », &reponse);
}
   while (reponse = = '0' || reponse = = '0'));
```

Rechercher un élément (x) dans la liste chaînée



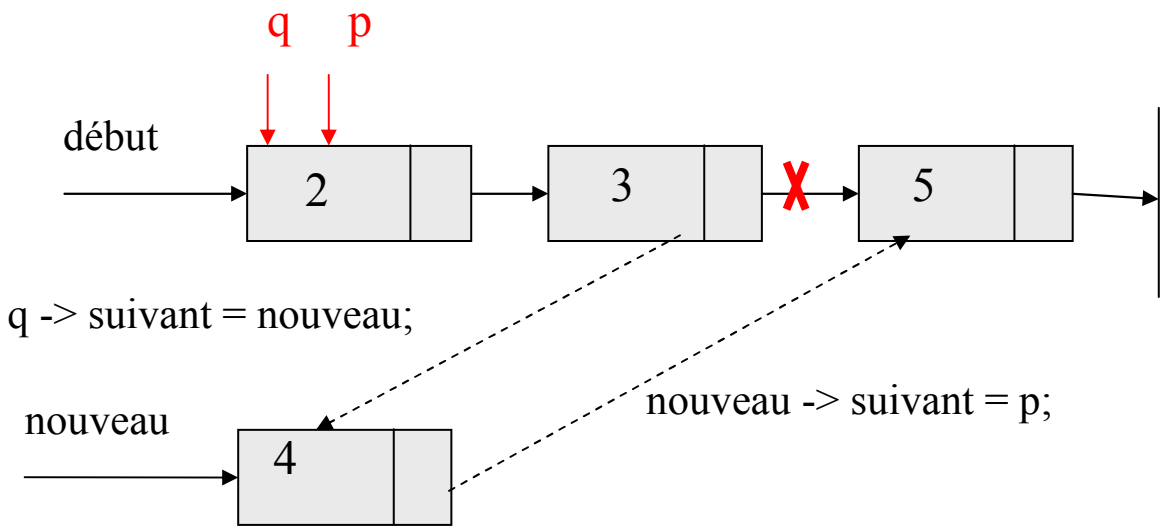
- Si l'x figure dans la liste chaînée :

```
struct maillon * debut;
struct maillon * p;
int x;
// Parcourir la liste et trouver x?
p = debut;
while ((p! =NULL) && (p ->donnee != x)) p = p -> suivant;
if (p == NULL) printf(« pas de %d\n » , x);
else printf («figure dans la liste %d\n » , x);
```

Listes chaînées triées

Exercice

- Comment insérer la donnée 4 ?



Cahier des charges

Lire des données entières et de les insérer au fur et à mesure dans une liste chaînée de façon à ce que cette liste soit ordonnée par ordre croissant des données.

Listes chaînées triées (1)

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

// Le type struct maillon est destiné à chaîner des "int"

struct maillon
{
  int valeur;
  struct maillon * suivant; //un pointeur vers une variable de type struct maillon
};
```

Listes chaînées triées (2)

// Dans ce programme, nous avons des variables de type "struct maillon *".
Pour alléger l'écriture, on utilise l'instruction typedef pour de renommer le type "**struct maillon ***" en le type "**VersMaillon**".

```
typedef struct maillon * VersMaillon;

VersMaillon inserer(int, VersMaillon);
void main()
{
  VersMaillon debut, p; //   On déclare deux variables de type VersMaillon ;
                            cette déclaration est équivalente à :  struct maillon * debut, * p ;
 int donnee, nb_donnee=0;
debut=NULL;
printf("Donnez vos données positives, tapez -1 pour terminer : \n");
```

Listes chaînées triées (3)

```
// On saisit les données
  scanf("%d",&donnee);
  while (donnee!=-1)&&(nb_donnee<4)
    {
     nb_donnee++;
     debut=inserer(donnee,debut);
    if(nb_donnee<4) scanf("%d",&donnee);
    }
```

Listes chaînées triées (4)

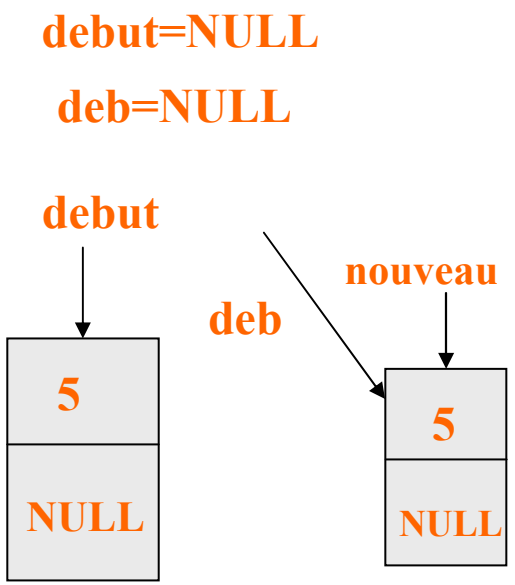
```
/*On affiche les données triées et on libère la mémoire allouée*/

while (debut!=NULL)
    {
     printf("%d ",debut->valeur); // debut est l'adresse d'une structure
                                  contenant un champ entier valeur ;
                                  debut->valeur est synonyme de (*debut).valeur.

     p=debut;
     debut=debut->suivant;
     free(p);
    }
  printf("\n");
}
```

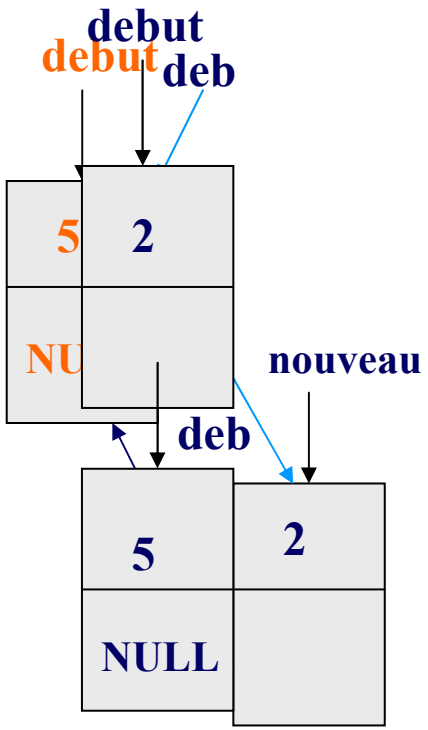
Listes chaînées triées (5) : fonction qui insère une donnée selon l'ordre croissant

```
VersMaillon inserer(int donnee, VersMaillon deb)
{
 VersMaillon nouveau, p, q;
 nouveau=(VersMaillon)malloc(sizeof(struct maillon));
 nouveau->valeur=donnee;
 if (deb==NULL)
   { nouveau->suivant=NULL;
    deb=nouveau;
   }
 else if (donnee<= deb->valeur)
   { nouveau->suivant=deb;
    deb=nouveau;
   }
 else
   {p=deb;
    q=deb->suivant;
    while((q!=NULL) && (donnee>q->valeur))
                {
                 p=q;
                 q=q->suivant;
                }
    p->suivant=nouveau;
    nouveau->suivant=q;
   }
 return deb;
}
```



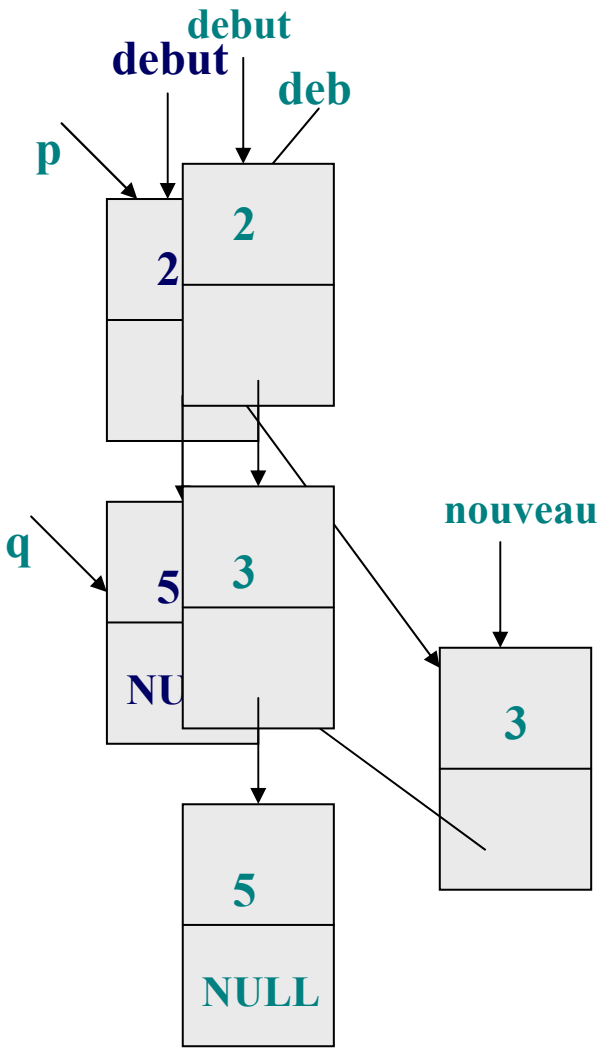
```
// On saisit les données
  scanf("%d",&donnee);
  while (donnee!=-1)&&(nb_donnee<4)
    {
     nb_donnee++;
     debut=inserer(donnee,debut);
    if(nb_donnee<4) scanf("%d",&donnee);
    }
```

```
VersMaillon inserer(int donnee, VersMaillon deb)
{
 VersMaillon nouveau, p, q;
 nouveau=(VersMaillon)malloc(sizeof(struct maillon));
 nouveau->valeur=donnee;
 if (deb==NULL)
   { nouveau->suivant=NULL;
     deb=nouveau;
   }
 else if (donnee<= deb->valeur)
   { nouveau->suivant=deb;
     deb=nouveau;
   }
 else
   {p=deb;
    q=deb->suivant;
    while((q!=NULL) && (donnee>q->valeur))
                {
                 p=q;
                 q=q->suivant;
                }
    p->suivant=nouveau;
    nouveau->suivant=q;
   }
 return deb;
}
```

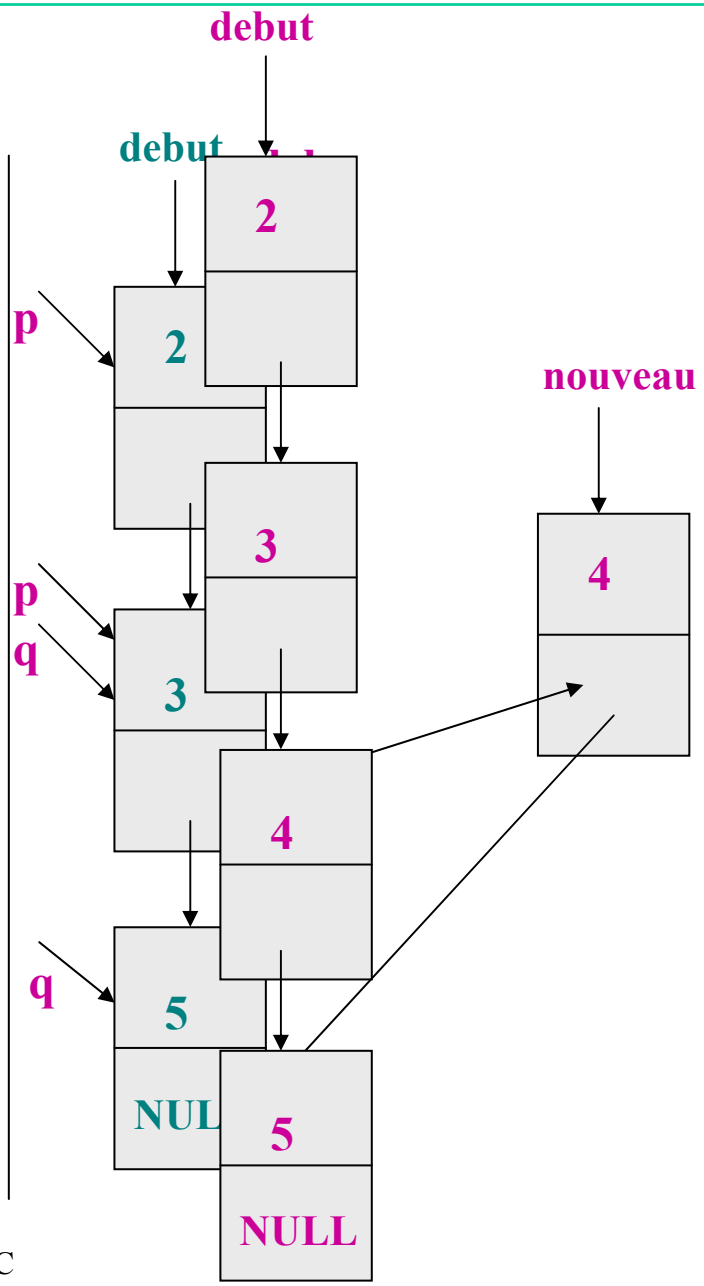


```
// On saisit les données
  scanf("%d",&donnee);
  while (donnee!=-1)&&(nb_donnee<4)
    {
     nb_donnee++;
     debut=inserer(donnee,debut);
    if(nb_donnee<4) scanf("%d",&donnee);
    }
```

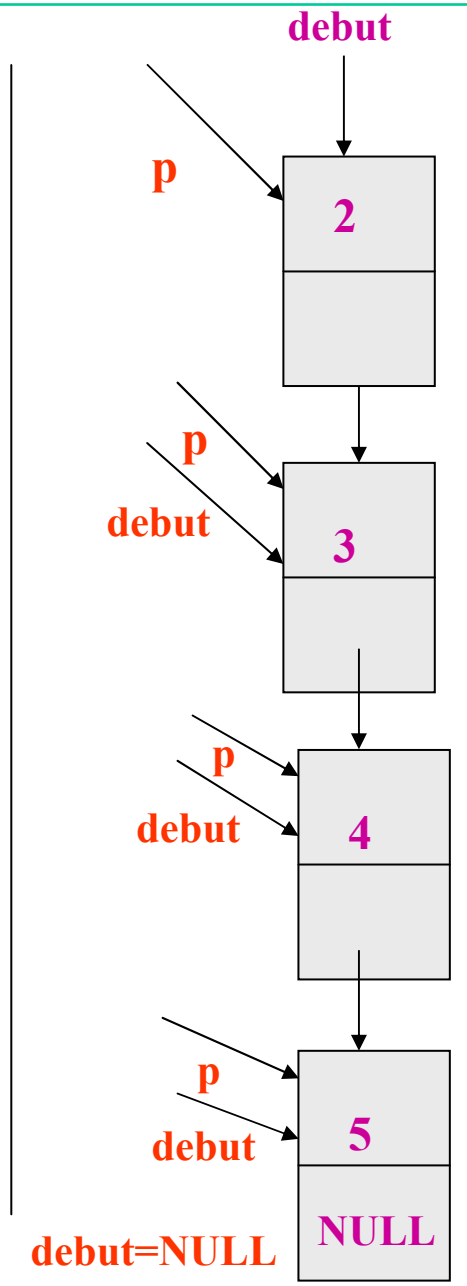
```
VersMaillon inserer(int donnee, VersMaillon deb)
{
 VersMaillon nouveau, p, q;
 nouveau=(VersMaillon)malloc(sizeof(struct maillon));
 nouveau->valeur=donnee;
 if (deb==NULL)
   { nouveau->suivant=NULL;
    deb=nouveau;
   }
 else if (donnee<= deb->valeur)
   { nouveau->suivant=deb;
    deb=nouveau;
   }
 else
   {p=deb;
    q=deb->suivant;
    while((q!=NULL) && (donnee>q->valeur))
                {
                 p=q;
                 q=q->suivant;
                }
    p->suivant=nouveau;
    nouveau->suivant=q;
   }
 return deb;
}
```



```
VersMaillon inserer(int donnee, VersMaillon deb)
{
 VersMaillon nouveau, p, q;
 nouveau=(VersMaillon)malloc(sizeof(struct maillon));
 nouveau->valeur=donnee;
 if (deb==NULL)
   { nouveau->suivant=NULL;
     deb=nouveau;
   }
 else if (donnee<= deb->valeur)
   { nouveau->suivant=deb;
     deb=nouveau;
   }
 else
   {p=deb;
    q=deb->suivant;
    while((q!=NULL) && (donnee>q->valeur))
                {
                  p=q;
                  q=q->suivant;
                }
    p->suivant=nouveau;
    nouveau->suivant=q;
   }
 return deb;
}
```



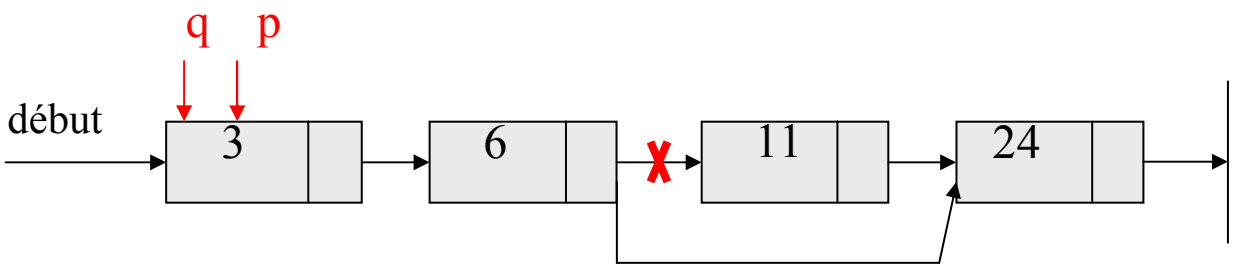
```
/*on libère la mémoire allouée*/
  while (debut!=NULL)
    {
      printf("%d ",debut->valeur);
      p=debut;
      debut=debut->suivant;
      free(p);
    }
  printf("\n");
}
```



Exercice

- Supprimer un élément

  ex: 11 à supprimer



q -> suivant = p -> suivant;

Les fichiers (1)

Pour conserver des données ou des résultats

A l'intérieur d'un programme, un fichier est représenté grâce à une variable de type **FILE**

- Ex: FILE * nom_fichier //Ce fichier est identifié par un nom.

On ouvre un fichier à l'aide de l'instruction ***fopen***

- Ex: mon_fichier = fopen (« donnees.don », « r »)

Les fichiers (2)

- Modes d'ouverture:
  - r : lecture
  - w : écriture
  - a : écriture à la fin du fichier
- A la fin d'utilisation d'un fichier il convient de le fermer à l'aide de ***fclose***
  - Ex: fclose (« donnes.don);

Les fichiers (3)

- Pour les entrées-sorties formatées sur fichier on peut employer les fonctions **fscanf, fprintf** où le premier argument est un pointeur sur le fichier à lire ou à écrire :

Ex:

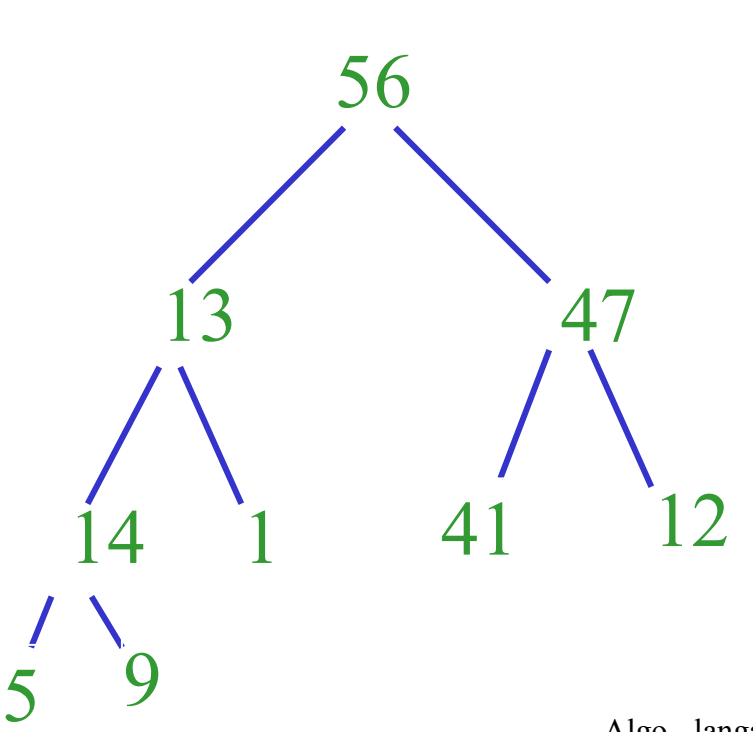
- fscanf (FILE * nom_fichier , « %d», &*var*);
  // où *var* est une variable de type int
- fprintf (FILE * nom_fichier, « message »);

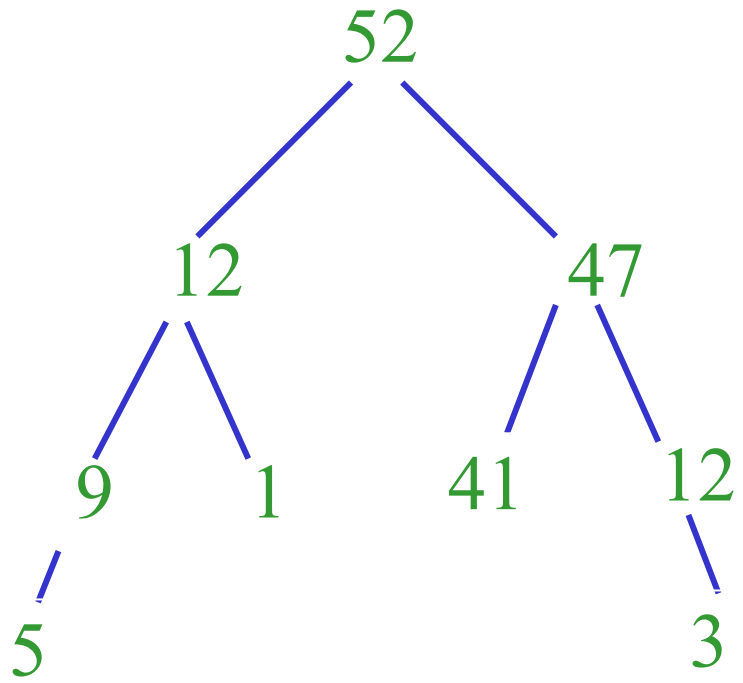
Algorithme de Tri Tas

Définition d'un tas (1)

Un tas est un **arbre binaire**

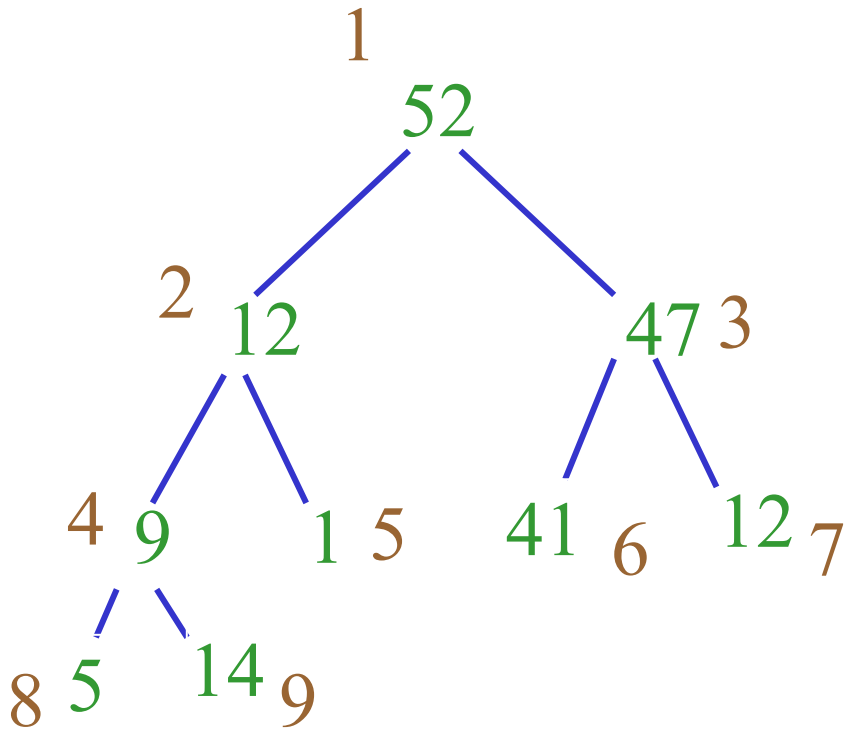
- "Parfait" :
  - toutes les rangées de l'arbre sont pleines sauf éventuellement la dernière , la dernière rangée est remplie de gauche à droite.





Définition d'un tas (2)

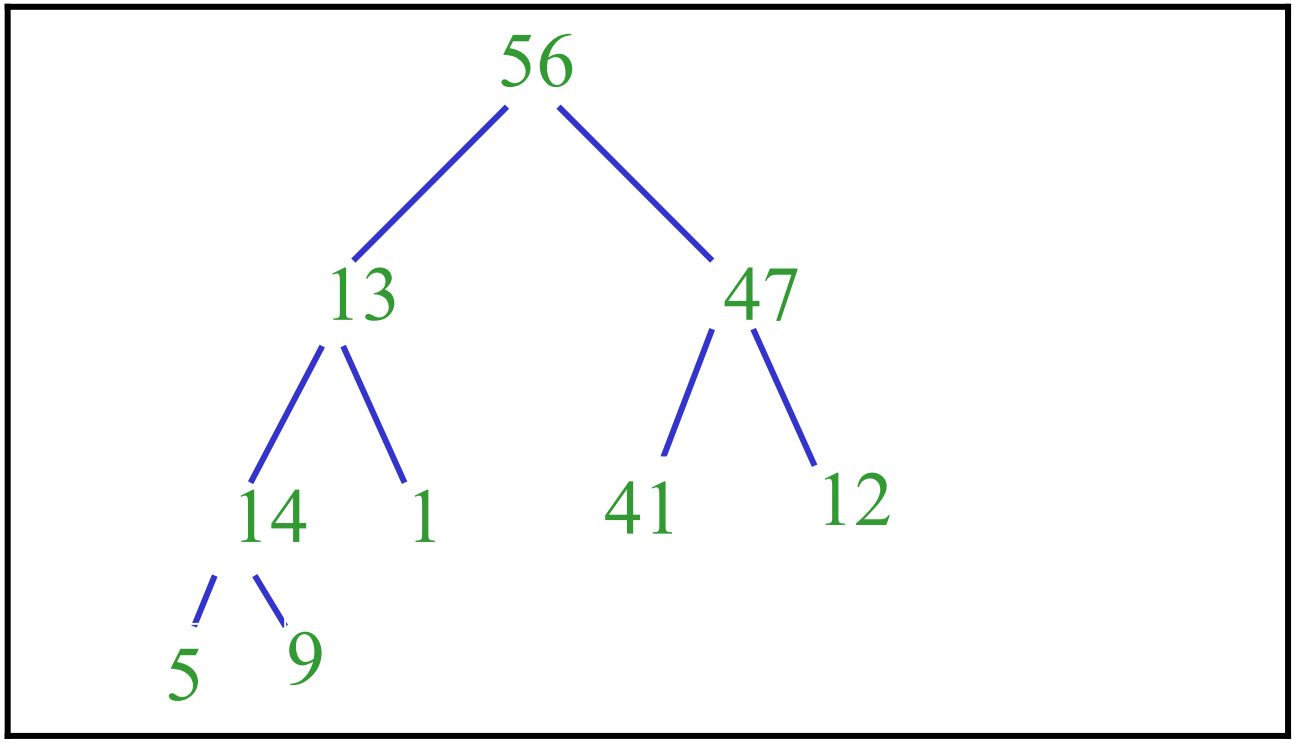
- Si on numérote les sommets de cet arbre de gauche à droite, dans chaque rangée, et de haut en bas, la racine ayant le numéro 1, on voit que les fils du sommet numéroté i sont numérotés 2i et 2i+1.



Définition d'un tas (3)

- Pour que l'arbre binaire parfait soit un tas, il faut qu'en chaque sommet l'élément qui s'y trouve soit plus grand que ceux situés en ses deux fils.
- Exemples:

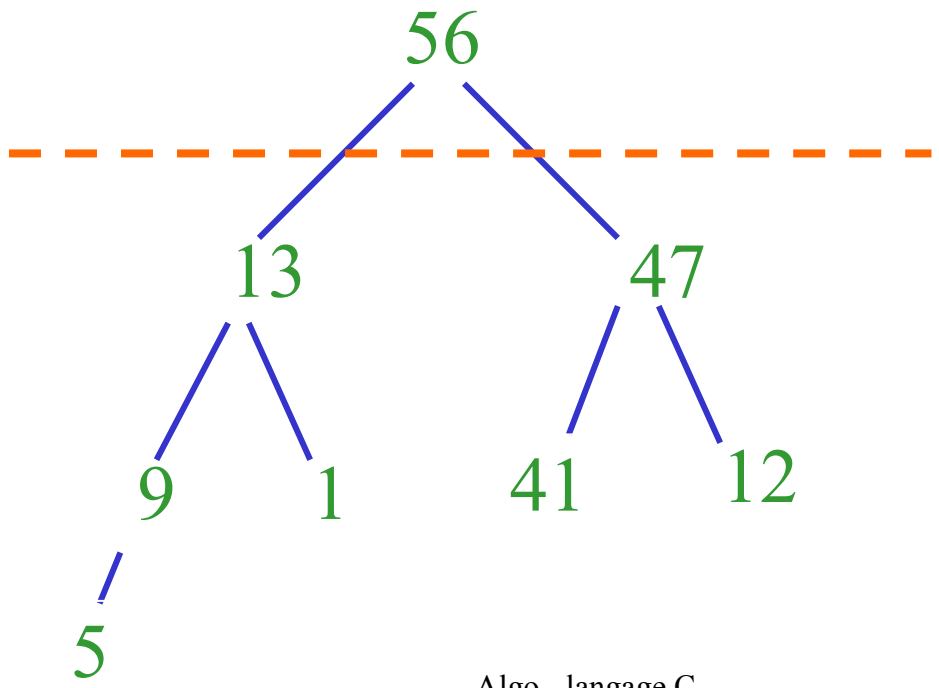
Tableau représentant un tas



56	13	47	14	1	41	12	5	9

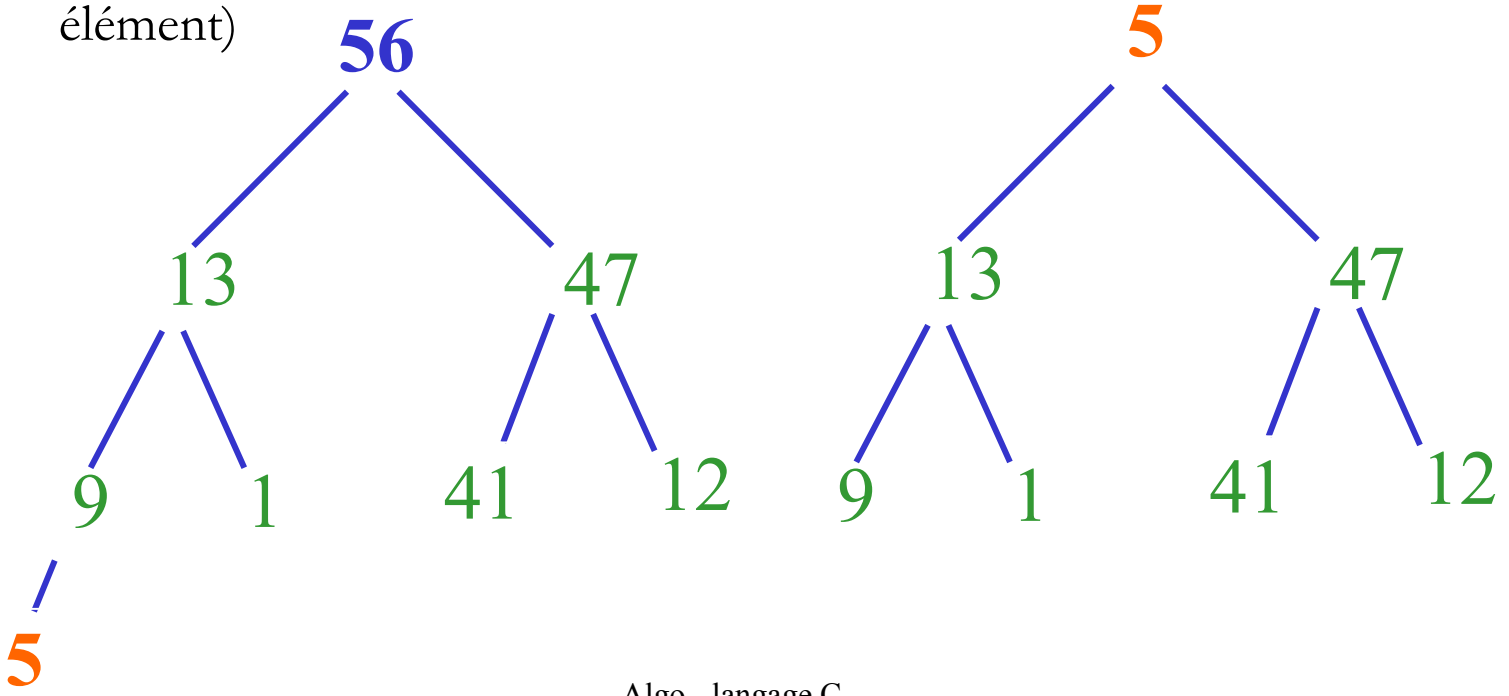
Algorithme tri tas (heapsort) (1)

- Une fois le tas construit avec tous les éléments à trier, le maximum de ceux-ci se trouve à la racine. On peut donc utiliser le tas pour trier les éléments :
  - on retire la racine, on restaure la structure de tas sur les éléments restants et on recommence.



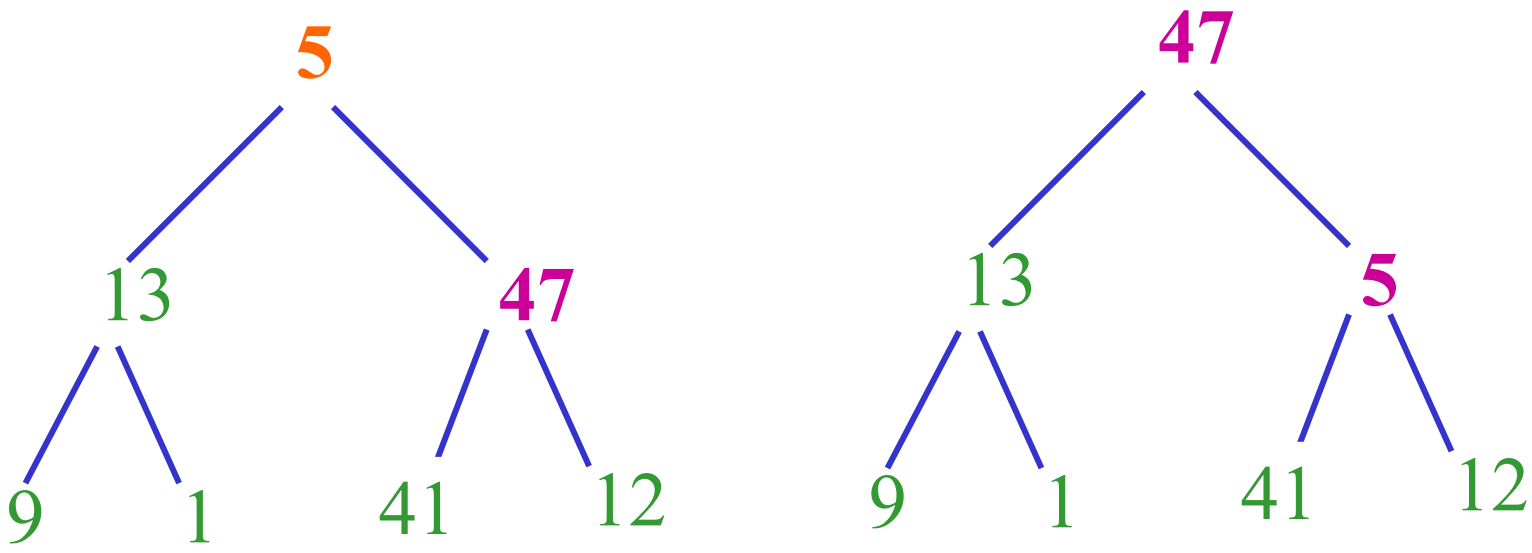
Algorithme (2)

- Pour restaurer la structure de tas :
  - on place le dernier élément du tas à la racine (à la place de l'élément qu'on vient de retirer pour construire la liste triée , en fait, pour ne pas consommer de place supplémentaire, **on échange la racine du tas courant** et **le dernier élément du tas courant**, et on considère que le tas courant est maintenant l'ancien tas courant privé de son dernier élément)

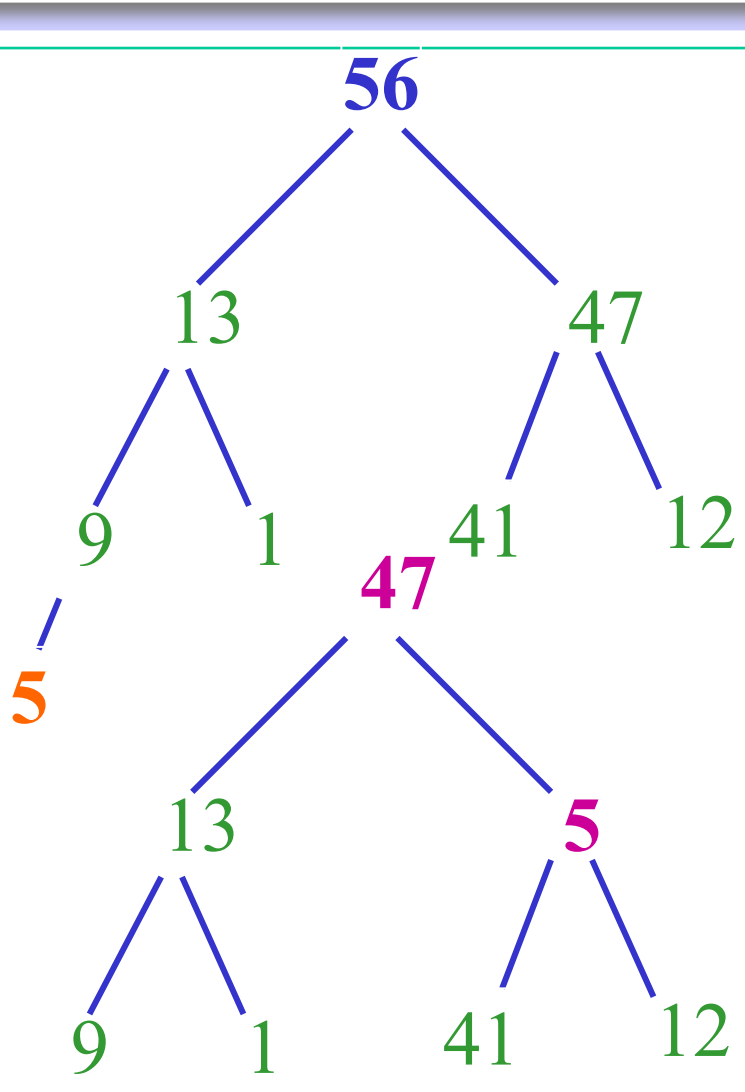


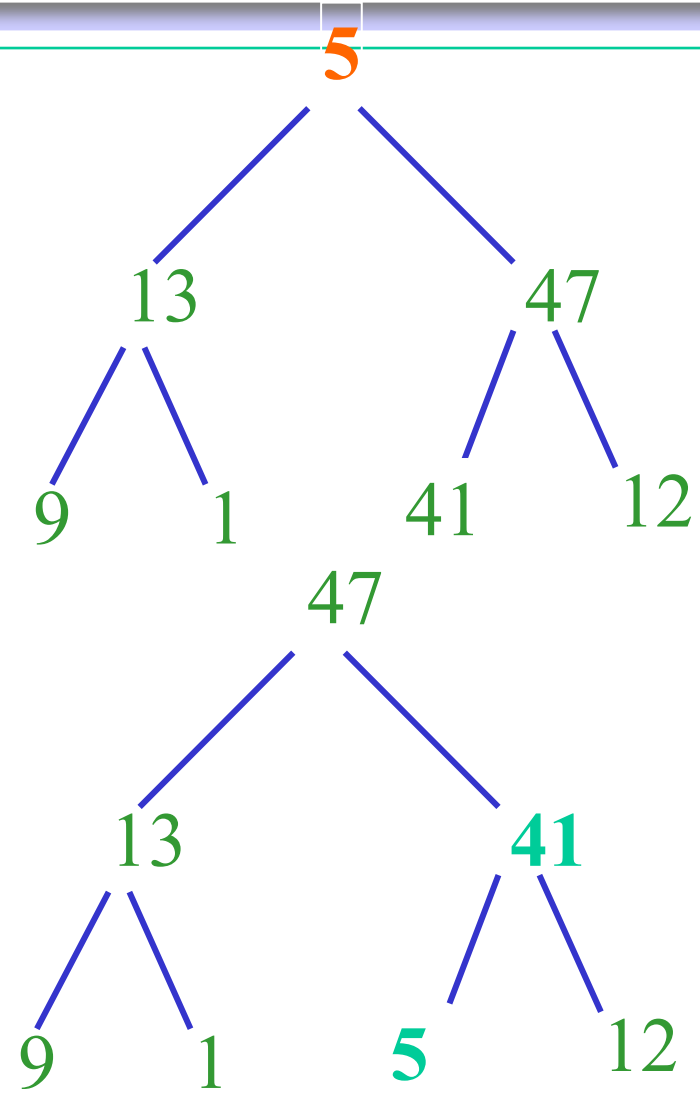
Algorithme (3)

- puis on fait "descendre" **le dernier élément** dans le tas en l'échangeant **avec le plus grand de ses fils**, si le plus grand de ses fils est plus grand que lui.



Algorithme (3)





- Quand on a fini, le tableau contient les éléments dans l'ordre croissant.

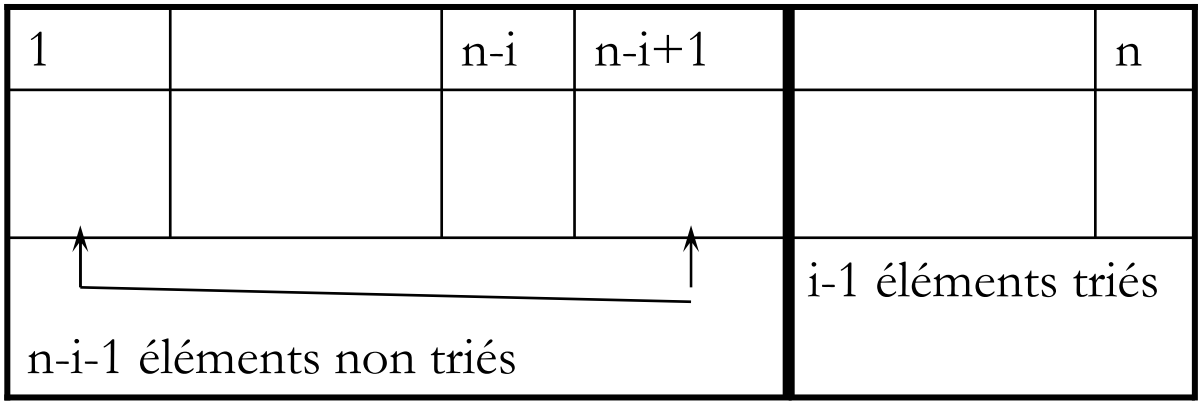
Construire le tas

- On part des tas préexistant (d'abord les feuilles)
- Et on construit ensuite l'arbre dont la racine est le père de ces feuilles… etc…
- Principe
  - On part du tableau à trier que l'on considère comme un arbre parfait
  - On l'ordonne en faisant « descendre » les pères qui sont plus petits que leur fils
    - Parcours du tableau du milieu jusqu'au début

    (i.e. en partant à droite sur l'avant dernier niveau

    Si le nœud courant est plus petit que son fils on le permute

Tri par tas



- 1) Construire le tas
- 2) trier le tableau
  - Parcourir le tableau de 1 à n-1
    - Commencer par permuter l'élément en pos 1 (la racine du tas) avec le dernier élément ; le plus grand élément est alors bien placé
    - Réorganiser le tas en faisant descendre l'élément en positon 1 si besoin
    - Et recommencer
  - Au début de l'étape i
    - De 1 à n-i-1 : le tableau est un tas
    - De i-1 à n le tableau est trié
      - A) permuter T[1] et T[n-i +1]
      - B) faire descendre la nouvelle racine

Principe du tri par tas

- Tri par sélection
  - On commence par placer l'élément le plus grand à sa place définitive au fond du tableau
  - Puis on recommence sur le tableau restant qui est un tas
  - Et ainsi de suite
- Tri sur place :
  - Le tableau à trier est d'abord réorganisé en tas
  - Puis le fond du tableau stocke les éléments les plus grands
- Deux Étapes
  1. Construire le tas
  2. Retirer la racine et la mettre dans le tableau trié et réorganiser le tas

  Et ce jusqu'à arbre vide

Exercice 6 : Algorithme de Tri Tas

Cahier des charges

- Trier un certain nombre d'entier à l'aide de l'algorithme de tri tas.
- Ces entiers figureront *dans un fichier* dont on demandera le nom à l'utilisateur *ou dans un tableau*.
- On écrira le résultat (entier triés) *dans un fichier* dont on demandera le nom à l'utilisateur *ou dans un tableau*.

Exercice 6 : Algorithme de Tri Tas (2)

Explications :

- Le tri tas est un tri "en place", c'est-à-dire que les éléments sont insérés et triés dans un unique tableau. Après leur insertion dans le tableau, les éléments sont tout d'abord rangés "en tas", le tableau simulant ce tas.
- Un tas est un arbre binaire "parfait" : toutes les rangées de l'arbre sont pleines sauf éventuellement la dernière, la dernière rangée est remplie de gauche à droite.
- Si on numérote les sommets de cet arbre de gauche à droite, dans chaque rangée, et de haut en bas, la racine ayant le numéro 1, on voit que les fils du sommet numéroté i sont numérotés 2i et 2i+1. Il y a donc une bijection naturelle entre les nœuds d'un tel arbre et les positions d'indice i, pour i variant entre 1 et n dans le tableau. A chaque sommet de l'arbre on associe un des éléments à trier.
- Pour que l'arbre binaire parfait soit un tas, il faut qu'en chaque sommet l'élément qui s'y trouve soit plus grand que ceux situés en ses deux fils.

Exercice 6 : Algorithme de Tri Tas (3)

Un algorithme possible de construction d'un tas est le suivant :

- On considère que l'on introduit les uns après les autres les éléments dans le tas, dont la structure est reconstituée après chaque insertion.
- Pour mettre à sa place l'élément en cours d'insertion, on l'introduit à la première place disponible, puis on le compare à son père : s'il est plus grand que son père, on l'échange avec son celui-ci, on recommence ces échanges jusqu'à ce que l'élément à insérer soit plus petit que son père ou qu'il ait atteint la racine.

Exercice 6 : Algorithme de Tri Tas (3)

Un algorithme possible de construction d'un tas est le suivant :

- On considère que l'on introduit les uns après les autres les éléments dans le tas, dont la structure est reconstituée après chaque insertion.
- Pour mettre à sa place l'élément en cours d'insertion, on l'introduit à la première place disponible, puis on le compare à son père : s'il est plus grand que son père, on l'échange avec son celui-ci, on recommence ces échanges jusqu'à ce que l'élément à insérer soit plus petit que son père ou qu'il ait atteint la racine.

Exercice 6 : Algorithme de Tri Tas (4)

- Une fois le tas construit avec tous les éléments à trier, le maximum de ceux-ci se trouve à la racine. On peut donc utiliser le tas pour trier les éléments : on retire la racine, on restaure la structure de tas sur les éléments restants et on recommence.
- Pour restaurer la structure de tas, on place le dernier élément du tas à la racine (à la place de l'élément qu'on vient de retirer pour construire la liste triée , en fait, pour ne pas consommer de place supplémentaire, on échange la racine du tas courant et le dernier élément du tas courant, et on considère que le tas courant est maintenant l'ancien tas courant privé de son dernier élément), puis on le fait "descendre" dans le tas en l'échangeant avec le plus grand de ses fils, si le plus grand de ses fils est plus grand que lui.
- Quand on a fini, le tableau contient les éléments dans l'ordre croissant.

Analyse du tri tas

- Le tas sera représenté par un tableau (global ou local) : T.

- On connaît pas le nombre d'éléments → tableau dynamique

Spécifications externes

- Variables globales :
  - Tableau contenant les entiers à trier : tas (sa taille n'est pas fixée) – tableau dynamique
  - Le nombre d'entier à trier
- Fonctions :
  - Construction (entasser, construire un tas) et lecture de donnes
  - Tri par Tas
  - Affichage de résultat
  - Fonction principale

Spécifications internes (1)

Spécifications internes (fonctions : Entasser, ConstruireUnTas, TriParTas)

i-indice de nœud

sous-fonction **FilsG(**i)

renvoyer (2i),

sous-fonction **FilsD**(i)

renvoyer (2i + 1)

Spécifications internes (2)

fonction Entasser (i, T , n)
//FilsG(i) et FilsD(i) sont des tas
début
iMax := i,
si (FilsG(i) ≤n) et (T [FilsG(i)] > T [iMax]) **alors**
iMax := FilsG(i),
fin si
si (FilsD(i) ≤ n) et (T [FilsD(i)] > T [iMax]) **alors**
    iMax := FilsD(i),
fin si
//iMax est donc l'indice du plus grand parmi T [i], T [F G(i)] et T [F D(i)]}
    si (iMax ≠i) **alors**
    Echanger(T [i], T [iMax]),
    Entasser(iMax, T , n),
fin si
// T est un tas
fin fonction

Spécifications internes (3)

// Construction d'un tas

fonction ConstruireUnTas(T , n)

//**In** : T un tableau de n éléments

//**Out** : T un tas

début

pour i := n/2 jusqu'à 1 faire

Entasser(i, T , n),

fin faire

//T est un tas

fin fonction

Spécifications internes (4)

//Tri par Tas : enlever le premier élément consiste simplement à l'échanger avec le dernier du tas et à décrémenter la taille du tas. On recouvre la propriété de tas en appliquant l'opération Entasser sur ce premier élément.

fonction TriParTas(T , n)

//**In** : T un tableau de n éléments

//**Out** : T trié

début

ConstruireUnTas(T , n),

pour i := n **jusqu'à** 2 **faire**

Echanger(T [1], T [i]),

Entasser(1, T , i 1),

fin faire

//T est trié

fin fonction

Coût du tri tas (1)

- Complexité :
  - en moyenne et dans le pire des cas en **O(n log n)** pour le nombre de comparaisons et d'échanges
  - tri sur place : pas de mémoire

Coût du tri tas (2)

- en moyenne cet algorithme est deux fois plus lent que le tri rapide et aussi plus lent que le tri fusion

- utilisé pour gérer des files de priorités

Chaîne de caractères, rupture de séquence, instruction switch, ligne de commande

Sommaire

- Chaîne de caractères
  - Déclaration
  - Opérations sur les chaînes de caractères
- Rupture de séquence (instructions return, break, continue)
- Instruction switch
- Ligne de commande

Chaîne de caractères (1)

- En langage C, une chaîne de caractères est un tableau, comportant plusieurs données de type **char**, dont le dernier élément est le caractère nul **'\0'**

- Ce caractère est un caractère de contrôle (donc non affichable) qui permet d'indiquer une fin de chaîne de caractères. Ainsi une chaîne composée de **n** éléments sera en fait un tableau **de n+1 éléments** de type char

- On peut par exemple représenter la chaîne "Licence" de la manière suivante:

Déclaration de chaîne de caractères

- Pour déclarer une chaîne de caractères en langage C, il suffit de définir un tableau de caractère:

  char chaine [51];
  ou char * chaine;

- Pour écrire le contenue d'une chaine de caractères utiliser :
  printf ("ma chaine est: % s", chaine);

- **scanf ("% s", chaine);** ajoute automatiquement \0 à la fin

Les fonctions de manipulation de chaînes de caractères

De nombreuses fonctions de manipulation de chaîne sont directement fournies. Ces fonctions se trouvent dans le fichier d'en-tête <string.h>, c'est la raison pour laquelle il faut ajouter la ligne suivante en début de programme:

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

Opérations sur les chaînes de caractères (1)

- char * s; //une chaîne dynamique

  s = (char * malloc (50 * sizeof (char)); // 49 caractères utiles

- Copies de chaînes:

  La fonction **strcpy** copie une chaîne existante

  sur une chaîne destinatrice:

  ex: strcpy (s, "bonjour"); //on recopie "bonjour" dans s, chaîne à laquelle on a attribué de la place mémoire

  - Si la taille N de transfert est plus longue que la chaîne émettrice, la chaîne destinatrice est complétée par des caractères nuls.
  - Inversement, si n est inférieur à la taille de la chaîne émettrice, la chaîne réceptrice sera incomplète.

Opérations sur les chaînes de caractères (2)

- Comparaison de chaînes:
  - Ex: strcmp (chaine1, chaine2);
  - La fonction **strcmp** retourne un entier:
    - <0 si la première chaîne se classe avant la seconde
    - =0 si les deux chaînes sont identiques
    - >0 si la première chaîne se classe après la seconde
- Longueur d'une chaîne :
  - strlen (chaine); //donne le nombre de caractères de chaîne

Opérations sur les chaînes de caractères (3)

- Concaténation de chaînes:
  - La seconde chaîne est recopiée à la fin de la première.
  - Le caractère nul de la fin de la première chaîne est écrasé par le premier caractère de la seconde chaîne

Exemples :

char *strcat(char *destination,char *source)

- recopie la source à la suite de la destination, rend un pointeur sur la destination

char *strncat (char *destination,char *source size_t n)

- **size_t** est un type d'entier non signé défini dans **string.h**
- la fonction **strncat** copie au plus n caractères
- les deux fonctions retournent un pointeur sur le dernier caractère des deux chaînes concaténés, c'est-à-dire le caractère \0.

Conversion de chaînes

- La bibliothèque standard offre des fonctions :
  - atoi - conversion d'une chaîne à un entier
  - atol - conversion d'une chaîne à un entier long
  - atof - conversion d'une chaîne à un flottant

Ruptures de séquence (1)

- Dans le cas où une boucle commande l'exécution d'un bloc d'instructions, on peut sortir de cette boucle alors que la condition de passage est encore valide. Ce type d'opération est appelé une **rupture de séquence**

- Les ruptures de séquence sont réalisées par les instructions suivantes qui correspondent à leur niveau de travail :
  - **return**
  - **break**
  - **continue**

Ruptures de séquence: l'instruction return

- Figure dans une fonction
- L'instruction return provoque la terminaison de l'exécution de la fonction dans laquelle elle se trouve et le retour à la fonction appelante.
- Les formes possibles du return pour la fonction void sont :
  - return; // sortir de la fonction
  - return expression;
- Le return permet de calculer une valeur correspondante au type de sa fonction. Ce calcul se fait en évaluant l'expression qui suit le return. L'expression est évaluée et la valeur calculée est retournée à la fonction appelante.

Ruptures de séquence: instruction break (1)

- L'instruction **break**
  - figure par exemple dans une boucle.
  - il a pour objectif de faire sortir de la boucle *for, while* ou *do while* dans des cas prédéfinis.
  - par exemple si l'on ne savait pas à quel moment le dénominateur (x-7) s'annule (pour des équations plus compliquées par exemple) il serait possible de faire arrêter la boucle en cas d'annulation du dénominateur, pour éviter une division par zéro!

Ruptures de séquence: instruction break (2)

- **Exemple**

```
for (x=1; x<=10; x++)
    {
     a = x-7; if (a == 0)
     { printf("division par 0");
     break;
     }
      printf("%f", 1/a);
    }
```

Ruptures de séquence: instruction continue (1)

- L'instruction **continue**
  - Figure dans une boucle.
  - inhibe les instructions de la boucle situées après le continue

  Ex:

```
while (i<n)
{
…
if (j ==n) continue;
   instr 1;
   instr 2;
   …
}
```

Ruptures de séquence: instruction continue (2)

- Exemple d'utilisation:

  - ce fragment de programme ne traite que les éléments positifs ou nuls du tableau T, il saute les valeurs négatives:

```
for(i=0; i<10; i++) // sauter les valeurs négatives
{
   if (T[ i ] <0)
   continue;
   …
```

L'instruction switch : les aiguillages multiples (1)

- Permet de réaliser les aiguillages multiples
- Sa syntaxe est la suivante:

```
int n;
scanf ("%d", &n);
switch (n)
{
    case 0:
                instruction 01;
                instruction 02;
    break;
    case 1:
                instruction 11;
                instruction 12;
    break;
    default: instruction 21;
    .....
}
```

L'instruction switch (2)

- Les parenthèses qui suivent le mot clé **switch** indiquent une expression dont la valeur est testée successivement par chacun des *case*.
- Lorsque l'expression testée est égale à une des valeurs suivant un *case*, la liste d'instruction qui suit celui-ci est exécuté.
- Le mot clé **break** indique la sortie de la structure conditionnelle.
- Le mot clé ***default*** précède la liste d'instructions qui sera exécutée si l'expression n'est jamais égale à une des valeurs.

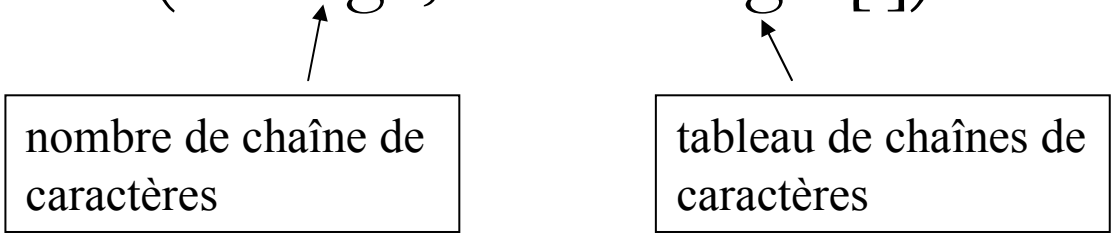
Ligne de commande (1)

La ligne de commande est l'endroit où l'on précise le programme à exécuter

Ex: > a.out

On peut préciser des paramètres sur la ligne de commande. Pour cela on change l'en-tête de la fonction main:

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



Ligne de commande (2)

La structure des arguments de la fonction main( ) reflète la liaison entre le langage C et le système d'exploitation, en particulier le système UNIX un entier et deux tableaux de pointeurs sur des caractères :

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

argument count — *argument values*

- **argc** contient le nombre d'arguments qui ont été passés lors de l'appel du binaire exécutable (nombre de mots dans la ligne de commande) ;
- **argv** contient les arguments de la ligne de commande. Ces arguments sont découpés en mots, chaque mot est référencé par un pointeur dans le tableau.

Ligne de commande (3)

Exemple, la commande:

> mon_programme nom_de_fichier 10

- alors argc vaut **2**
- argv est un tableau de chaîne de caractères

argv[0] vaut

« mon_programme »

argv[1] vaut

«nom_de_fichier»

argv [2] vaut

« 10 »

(qu'on peux convertir à l'aide de la fonction ***atoi***)

Récursivité, bibliothèque C standard

Sommaire

- **La récursivité**
  - La notion de récursivité
  - Récursivité et itération
  - Comment concevoir un sous-programme récursif ?
  - L'exemple: Tour d'Hanoi
- **La bibliothèque standard**
  - Objectifs
  - Organisation générale
  - Parties non normalisées

La notion de récursivité (1)

- Une fonction est dite récursive si son exécution ***fait appel à elle-même.***
- L'exemple le plus connu en matière de récursivité : la factorielle.
  - Ex: calcul de la factorielle d'une valeur entière positive ***n*** (n! = 1 * 2 * .... * n)

    Mais, n! = (1 * 2 * .... * (n-1)) * n = (n-1)! * n.
  - Donc, il suffit de savoir calculer (n-1)! pour pouvoir calculer la valeur de n! par multiplication avec n.
  - ***Le sous-problème*** du calcul de (n-1)! est le même que le problème initial, mais pour un cas "plus simple", car n-1 < n.

$$n! = \begin{cases} 1 \text{ si } n=0 \\ n * (n-1)! \text{ si } n \geq 1 \end{cases}$$

La notion de récursivité (2)

- ***Remarque*** : un appel récursif va produire lui-même un autre appel récursif, etc, ce qui produit une suite infinie d'appels. Il faut arrêter la suite d'appels au moment où le sous-problème peut être résolu directement.
- Dans la fonction précédente, il faut s'arrêter (ne pas faire d'appel récursif) si n = 0, car dans ce cas on connaît le résultat (1). Cette fonction factorielle peut s'écrire de la manière suivante :

```
int fact(int n)
{
if (n == 0) return 1 ;
else return n*fact(n-1) ;
}
```

- **Appel récursive doit *être conditionné*,** afin d'éviter des appels récursives infinis

Récursivité et itération

- Par l'appel répété d'un même sous-programme, la récursivité permet de réaliser des traitements répétitifs.
- Suivant le type de problème, la solution s'exprime plus naturellement par récursivité ou par itération.
  - En principe tout programme récursif peut être écrit à l'aide de boucles (par itération), sans récursivité.
  - Inversement, chaque type de boucle (while, for) peut être simulé par récursivité. Il s'agit en fait de deux manières de programmer différentes.

Utilisation de l'itération

- L'avantage essentiel de l'itération est *l'efficacité*. Un programme qui utilise des boucles décrit précisément chaque action à réaliser – ce style de programmation est appelé *impératif* ou *procédural*.
- Le code compilé d'un tel programme est une image assez fidèle des actions décrites par le programme, traduites en code machine.
- *Conclusion* : si on recherche l'efficacité (une exécution rapide) et le programme peut être écrit sans trop de difficulté en style itératif, on préférera l'itération.

Utilisation de l'itération: exemple

- Autre possibilité pour la factorielle:

```
int autre_fact (int n);
{
   int i;
   int resultat = 1;          //initialiser à 1
   for (i=2; i <= n; i++)
   resultat * = i;  // resultat = resultat * i
   return resultat;
}
```

Utilisation de la récursivité

- L'avantage de la récursivité est qu'elle se situe à un *niveau d'abstraction supérieur* par rapport à l'itération.

- Une solution récursive décrit comment calculer la solution à partir d'un cas plus simple – ce style de programmation est appelé *déclaratif*.

  - Au lieu de préciser chaque action à réaliser, on décrit ce qu'on veut obtenir - c'est ensuite au système de réaliser les actions nécessaires pour obtenir le résultat demandé.
  - La récursivité est géré par l'ordinateur

Comment concevoir un sous-programme récursif ?

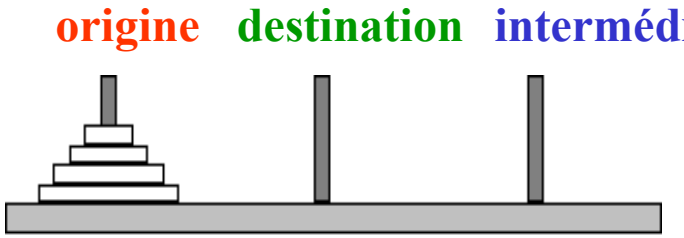
Dans l'écriture des programmes récursifs on retrouve généralement les étapes suivantes :

- **Trouver une décomposition récursive du problème**
  - Trouver l'élément de récursivité qui permet de définir les cas plus simples (*ex.* une valeur numérique qui décroît, une taille de données qui diminue).
  - Exprimer la solution dans le cas général en fonction de la solution du cas plus simple.
- **Trouver la condition d'arrêt de récursivité et la solution dans ce cas**
- **Réunir les deux étapes précédentes dans un seul programme**

Exemple: Tour de Hanoi

Cahier des charges

- Le but de ce programme est de résoudre le problème des tours de Hanoï.
- Ce problème est le suivant:
  - Le joueur dispose de ***n*** anneaux et de trois tours.
    Ces anneaux sont de tailles toutes différentes et à tout moment, au-dessus de tout anneau, il n'y a que des anneaux plus petits que lui.
    Initialement tous les anneaux sont disposés sur la première tour .



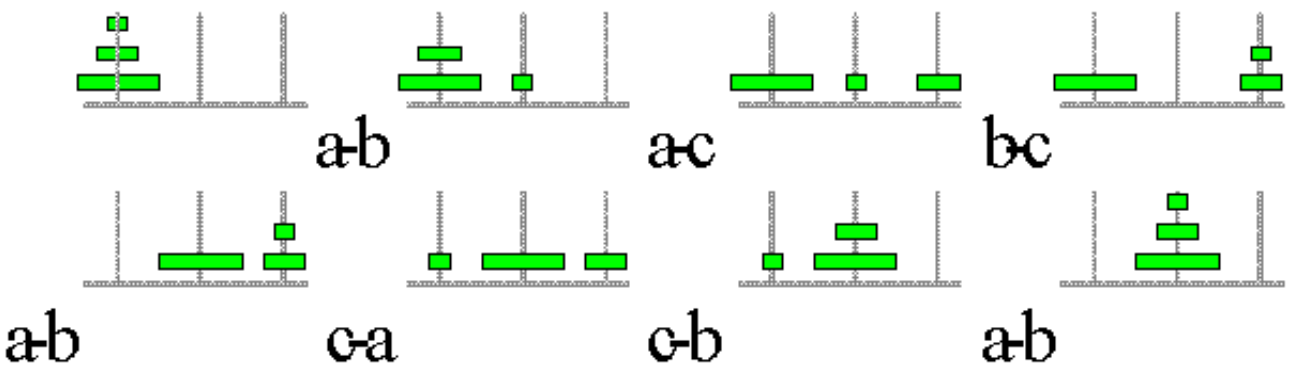
  - **Objectifs:** déplacer les anneaux une par une, en respectant la règle de l'empilement du piquet 1 vers le piquet 2.

  - La première tour est rouge, la seconde verte et la troisième bleue.

Tour de Hanoi

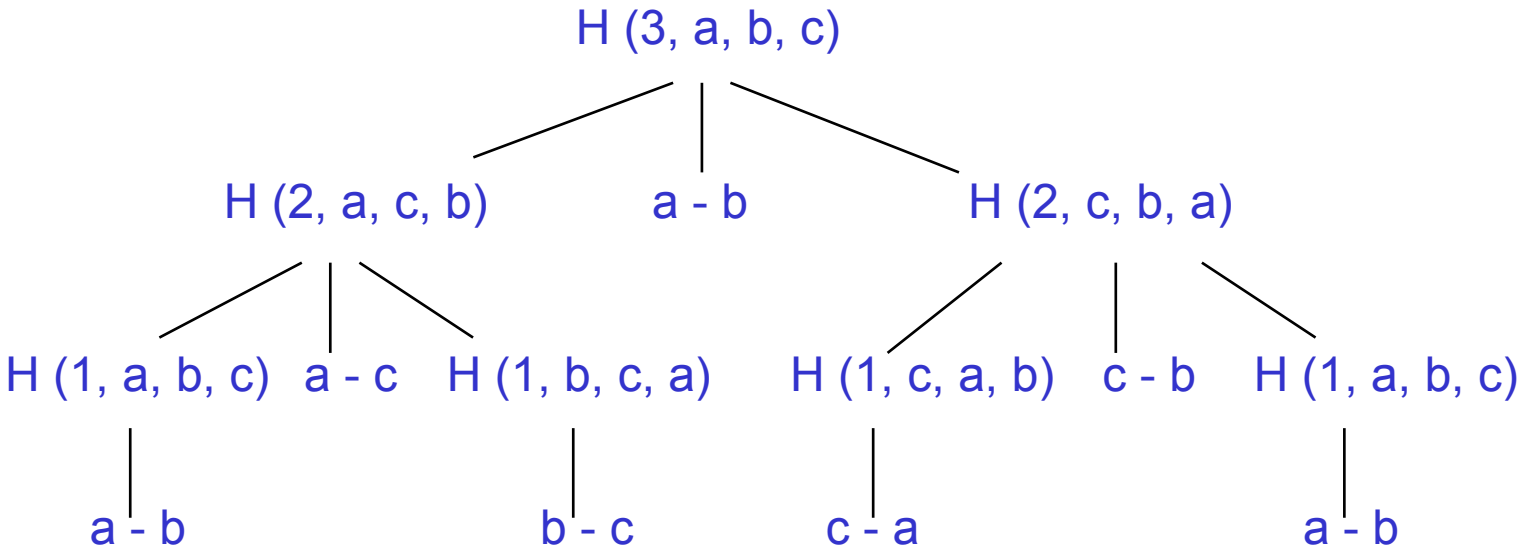
Règles de déplacement :

- On ne peut déplacer que un seul disque à la fois
- Un disque ne peut pas se trouver sur un disque de diamètre inférieur



Tour de Hanoi : exécution

La pile et la trace d'exécution à la suite de l'appel Hanoi (3, a, b, c)



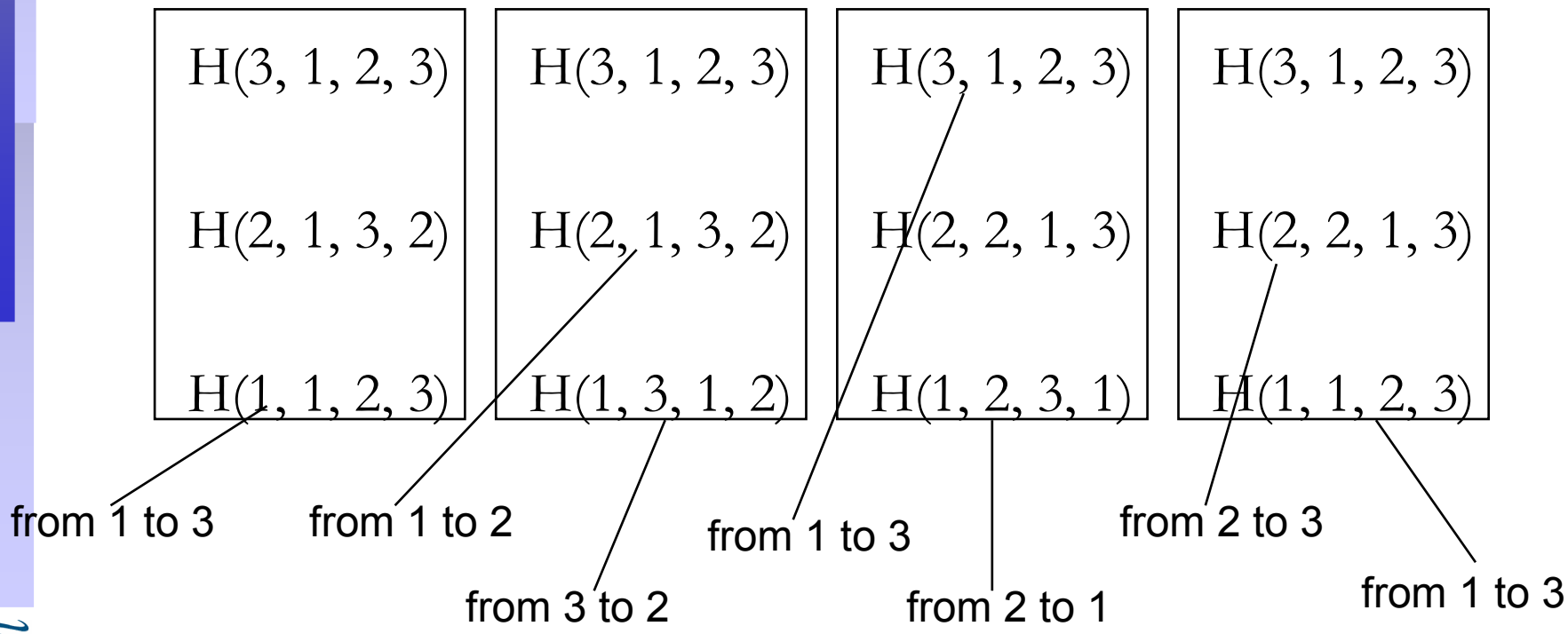
Temps d 'exécution

$T(H, n \text{ disques}) = O(2^n)$

$\text{Longueur}(H(n, a, b, c)) = 2^n - 1$

Exemple: Tour de Hanoi

la pile et la trace d'exécution à la suite de l'appel Hanoi(3, 1, 2, 3)



Spécifications externes (1)

- Nous allons traiter ce problème de manière récursive. Supposons que nous sachions déplacer un nombre donné d'anneaux situés en haut de la pile, d'une tour de départ donnée vers une tour d'arrivée donnée.
- Pour effectuer le déplacement nous allons construire de manière "récursive" **une fonction "hanoi",** à quatre paramètres:
  - **un entier** représentant le nombre de tours à déplacer,
  - puis trois paramètres de type "Tour": **type énuméré** contenant les couleurs des trois tours considérées, et qui représentent
    - l'origine,
    - la destination
    - et la tour intermédiaire.

Spécifications externes (2)

- La fonction hanoi est récursive, ce qui signifie qu'elle s'appelle elle-même:

  void hanoi (int n, Tour origine, Tour destination, Tour intermediaire)

- Pour transporter n disques d'une tour "origine" à une tour "destination" en utilisant une tour "intermédiaire", il suffit de :
  - transporter (n - 1) disques de la tour "origine" à la tour "intermédiaire" en utilisant la tour "destination"
  - transporter le dernier disque de la tour "origine" à la tour "destination"
  - transporter (n - 1) disques de la tour "intermédiaire" à la tour "destination" en utilisant la tour "origine".

Spécifications externes (3)

- La condition d'arrêt de récursivité:

  if (n>0)

- Cette condition est le "test d'arrêt" sur les appels récursifs; celui-ci évite que la fonction s'appelle elle-même éternellement.
- Le test d'arrêt peut se situer au début de la fonction récursive (comme ici) ou bien juste avant les appels récursifs à l'intérieur de la fonction.

```
void hanoi (int n,Tour origine,Tour destination,Tour intermediaire)
{
        if(n>0)
        {....
        }
}
```

Programme (1)

```
// ma fonction récursive
void hanoi (int n,Tour origine,Tour destination,Tour
intermediaire)
{
      if(n>0)
      { hanoi(n-1,origine,intermediaire,destination);
        printf("transportez l'anneau qui est en haut de la tour %s "
      "vers la tour %s\n",traduit(origine),traduit(destination));
        hanoi(n-1,intermediaire,destination,origine);
      }
}
```

Programme (2)

```
    void main( void)
{
    int n;
    printf("Avec combien d'anneaux voulez-vous jouer ? \n" );
    scanf("%d",&n);
    hanoi (n, rouge, verte, bleue);
}
```

Bibliothèque C standard (1)

Objectifs

- Fournir une interface générique avec le système:
  - portabilité
  - abstraction (fonctions de plus haut niveau)
  - concerne principalement les entrées-sorties
- Fournir des fonctions "d'utilité générale":
  - manipulation des chaînes et de caractères
  - fonctions mathématiques
- En augmentation: chiffrement, …

Bibliothèque C standard (2)

Organisation générale:

- Entrée/Sortie: <stdio.h>
  - manipulation de fichiers (*fopen, fclose, ffluch, rename…*), *page 156*
  - entrées/sorties de caractères (*fgetc, getc, fputc, putc,…*), *page 157*
  - entrées/sorties formatées (*fscanf, fprintf, …*)
  - entrées/sorties binaires (*fread, fwrite*)
  - positionnement dans les fichiers (*fseek, ftell, rewind, …*)
- Traitement des chaînes: <string.h>
  - copie (*strcpy, strncpy*)
  - comparaison (*strcmp, strncmp*)
  - recherche d'un caractère (*strchr, strnchr*)

Bibliothèque C standard (3)

Organisation générale (suite):

- Fonctions mathématiques: <math.h> - *page 167*
  - réels
  - entiers
- Fonctions d'ordre général: <stdlib.h>
  - d'allocation mémoire (*malloc, free,* ..)
  - d'échappement (*abort, exit,* …)
  - de conversion de chaînes (*atoi*, …)
  - fonctions système (*getenv, system*)

Bibliothèque C standard (4)

Organisation générale (suite):

- Gestion du temps: <time.h>
  - donner l'heure (*clock*, *time*)
  - convertir l'heure (*mktime*)
- Gestion des signaux:<signal.h>
  - définition de l'action à réaliser à l'instant de la délivrance d'un signal (*signal*)
  - envoi d'un signal (*raise*)
- Fonctions d'erreur: <assert.h>
  - diagnostic à l'exécution (*assert*)

Parties non normalisées

Parties non normalisées ou en cours de normalisation :

- Tris: <search.h>
- Threads: <pthreads.h>
- Chiffrement: <crypt.h>
- Gestion de paramètres: <getopt.h>
- Expressions régulières: <regexp.h >