Algorithmique et programmation (suite)

L1, S2

Février 2018

Vincent Rodin
Université de Bretagne Occidentale
vincent.rodin@univ-brest.fr
http://labsticc.univ-brest.fr/~rodin/FTP/Enseignements/L1/AlgoEtProg

Algorithmique et programmation
Plan

Licence 1, deuxième semestre

- ▷ Conventions d'écriture
- ▷ Procédures et fonctions
- ▷ Structures de données et tableaux
- ▷ Algorithmes de tri et complexité
- ▷ File et pile ...

... vers la programmation modulaire

Conventions d'écriture de programmes C

Expliciter l'implicite
{ – }
Indentation
Déclaration de variable, durée de vie, visibilité
Expressions et (–)
Commentaires

Expliciter l'implicite

Tout ce qui est explicite peut aider à :

- ▷ obtenir un code plus robuste
- ▷ faire moins d'erreurs de syntaxe
- ▷ avoir un code plus lisible (pour soi et pour les autres)

Et donc...

"Expliciter l'implicite"

...même si c'est plus long à écrire!

{ – } (1)

Afin de réduire le risque d'erreurs de syntaxe et d'exécution, mettre toujours des { – } même si ce n'est pas obligatoire !

▷ **Dans l'instruction conditionnelle `if`**

```
#include <stdio.h>

int main(void)
{
 int a = 0;

 if (a == 0)
 {
   printf("C'est zero\n");
 }
 else
 {
  printf("Ce n'est pas zero\n");
 }
 return 0;
}
```

Permet de bien délimiter les blocs d'instructions

{ – } (2)

▷ **Dans l'instruction `while`**

```
#include <stdio.h>

int main(void)
{
 int i;

 i=0;
 while (i<=9)
 {
  printf("%d\n",i); /* 0 => 9 */
  i = i + 1;
 }

 printf("%d\n",i); /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 i=9;
 while (i>=0)
 {
  printf("%d\n",i); /* 9 => 0 */
  i = i - 1;
 }

 printf("%d\n",i); /* -1 */

 return 0;
}
```


{ – } (3)

▷ **Dans l'instruction `do ... while`**

```
#include <stdio.h>

int main(void)
{
 int i;

 i=0;
 do
 {
  printf("%d\n",i); /* 0 => 9 */
  i = i + 1;
 } while (i<=9);

 printf("%d\n",i); /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 i=9;
 do
 {
  printf("%d\n",i); /* 9 => 0 */
  i = i - 1;
 } while (i>=0);

 printf("%d\n",i); /* -1 */

 return 0;
}
```


{ – } (4)

▷ **Dans l'instruction `for`**

```
#include <stdio.h>

int main(void)
{
 int i;

 for(i=0; i<=9; i = i + 1)
 {
  printf("%d\n",i); /* 0 => 9 */
 }

 printf("%d\n",i);  /* 10 */

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 int i;

 for(i=9; i>=0; i = i - 1)
 {
  printf("%d\n",i);  /* 9 => 0 */
 }

 printf("%d\n",i);   /* -1 */

 return 0;
}
```

{ – } (5)

▷ **Dans l'instruction `switch`**

```
#include <stdio.h>

int main(void)
{
 int n;
 printf("Donnez moi une valeur (soit 1, soit 2):\n");
 scanf("%d",&n);
 switch(n)
 {
  case 1 : { printf("Un\n");
             break;
           }
  case 2 : { printf("Deux\n");
             break;
           }
  default: { printf("Ni Un, ni deux\n");
             break;
           }
 }
 return 0;
}
```


Indentation (1)

L'indentation permet de rendre un programme plus lisible...

```
/*   A compiler avec:
    cc page010_sansIndentation.c -o page010_sansIndentation -lm
*/
#include <stdio.h>
#include <math.h>    /* Pour sqrt */
int main(void)
{
double a, b, c, delta, x1, x2;
printf("Calcul des racines de ax2 + bx + c = 0\n");
printf("Donnez moi les valeurs de a, b et c :\n");
scanf("%lg",&a); scanf("%lg",&b); scanf("%lg",&c);
if (a==0.0) { printf("Erreur a egal a 0 !\n"); }
else { delta = b*b - 4*a*c; if (delta<0)
{ printf("Pas de solution\n"); } else { if (delta>0.0)
{ x1 = (-b-sqrt(delta))/(2*a); x2 = (-b+sqrt(delta))/(2*a);
printf("Deux solutions: %g et %g\n",x1,x2); } else
{ x1 = x2 = -b/(2*a); printf("Une solution: %g\n",x1); } } } return 0; }
```

... pas lisible sans une bonne indentation !

Indentation (2)

```
/*  A compiler avec:
    cc page011_avecIndentation.c -o page011_avecIndentation -lm
*/
#include <stdio.h>
#include <math.h>    /* Pour sqrt */
int main(void)
{
 double a, b, c, delta, x1, x2;
 printf("Calcul des racines de ax2 + bx + c = 0\n");
 printf("Donnez moi les valeurs de a, b et c :\n");
 scanf("%lg",&a); scanf("%lg",&b); scanf("%lg",&c);
 if (a==0.0) { printf("Erreur a egal a 0 !\n"); }
 else {
   delta = b*b - 4*a*c;
   if (delta<0) { printf("Pas de solution\n"); }
   else {
     if (delta>0.0) {
       x1 = (-b-sqrt(delta))/(2*a);
       x2 = (-b+sqrt(delta))/(2*a);
       printf("Deux solutions: %g et %g\n",x1,x2);
     }
     else {
       x1 = x2 = -b/(2*a);
       printf("Une solution: %g\n",x1);
     }
   }
 }
 return 0;
}
```

C'est plus lisible ?

Déclaration de variable, durée de vie, visibilité

La déclaration d'une variable se fait en début de bloc { – }.

Attention :

- ▷ **La durée de vie d'une variable est celle du bloc.**
- ▷ **Une variable peut en cacher une autre.**

```
#include <stdio.h>

int main(void)
{
 int a=10, i;

 i=0;
 while (i<=9)
 {
  int a=100;             /* Creation a chaque entree dans le bloc... a chaque tour de boucle */

  printf("%d\n", i + a); /* 100 => 109 */
  i = i + 1;
 }

 printf("%d\n", i + a); /* 20 */

 return 0;
}
```

Expression et (–) (1)

Les couples de (et) permettent de changer l'ordre d'évalution des opérateurs...

```
#include <stdio.h>

int main(void)
{
 printf("2+3*4=%d\n",2+3*4);     /* 14 */
 printf("(2+3)*4=%d\n",(2+3)*4); /* 20 */

 return 0;
}
```

Analyse : le + est moins prioritaire que le ∗

⇒ En fonction de l'opération voulue, il faut ajouter des (et)

Expression et (–) (2)

Priorité : ◇ Un opérateur "attire" plus ou moins ses opérandes.
◇ En C, il existe 46 opérateurs ... et 17 niveaux de priorité !

Remarque:

△ Lorsque **deux** (ou plus) **opérateurs** interviennent dans une **expression**
⇒ la priorité des opérateurs s'applique...

△ Lorsque **deux opérateurs** ont **même priorité**
⇒ l'associativité des opérateurs s'applique...

Expression et (–) (3)

Priorité des opérateurs :

◇ Priorités décroissantes
◇ Entre deux lignes, les opérateurs ont la même priorité

Opérateurs	Symbole	Exemple	Associativité
Négation logique (unaire)	!	!(a==b)	←
multiplication	*	i * j	→
division	/	i / j	→
modulo	%	i % j	→
addition	+	i + j	→
soustraction	-	i - j	→
comparaisons (relations)	< <= > >=	i < j	→
égalité, inégalité	== !=	i == j i != j	→
ET (conjonction, logique)	&&	a && b	→
OU (disjonction, logique)	||	a || b	→

Mettre des commentaires utiles

```
/* Commentaire utile */
```

Essentiel pour la vie du programme...

pour le reprendre, l'améliorer, le corriger

...par vous ou quelqu'un d'autre

▷ **Exemple de commentaire inutile :**

```
i = i + 1;   /* j'augmente la valeur de i de 1 */
```

▷ **Exemples de commentaires utiles :**

```
 /* Description de la fonction, de l'algorithme, ... */
int sommeArithmetique(int n)
{
   ...    /* Description des particularites de l'algorithme */
   ...    /* des cas particuliers traites, ...              */
}
```

Procédures et fonctions

Spécification et implémentation
Concepteur versus utilisateur
Structuration des algorithmes: Diviser pour régner
Réutilisabilité des algorithmes: Paramétrer pour réutiliser
Procédures ; Fonctions
C: déclaration - définition
Spécification
Implémentation
Algorithme : Spécification + Implémentation
Passage de paramètre : par valeur, par adresse
Procédures et fonctions: comment choisir ?
Exemples de types de fonctions
Procédures et fonctions: écriture algorithmique "en français"
Fonctions récursives

Spécification et implémentation

▷ **Spécification d'un algorithme** **Quoi?**

La spécification décrit la fonction et l'utilisation d'un algorithme

ce que fait l'algorithme.

L'algorithme est vu comme une boîte noire donc on ne connaît pas le fonctionnement interne.

▷ **Implémentation d'un algorithme** **Comment?**

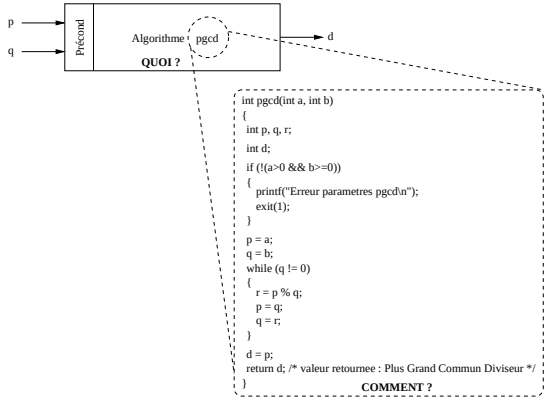
L'implémentation décrit le fonctionnement interne de l'algorithme

comment fait l'algorithme.

L'implémentation précise l'enchaînement des instructions nécessaires à la résolution du problème considéré.

Spécification et implémentation

La spécification décrit la fonction et l'utilisation d'un algorithme



L'implémentation décrit le fonctionnement interne de l'algorithme

Concepteur versus Utilisateur

▷ **Concepteur**

Le concepteur d'un algorithme définit :
- l'**interface** et
- l'**implémentation**

de l'algorithme.

▷ **Utilisateur**

L'utilisateur d'un algorithme n'a pas a connaître son implémentation ; seule l'**interface** de l'algorithme le concerne.

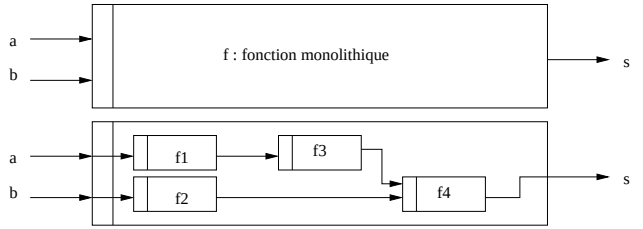
Selon la spécification de l'algorithme, l'utilisateur **appelle** (utilise) l'algorithme :
- sous forme de **fonction** ou
- sous forme de **procédure**. (i.e. procédure : fonction sans valeur de retour)

Structuration des algorithmes (1)

"Diviser pour régner"

▷ **Structuration**

Les procédures et les fonctions permettent de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite.



Structuration des algorithmes (2)

▷ **Problème**

Comment structurer un algorithme pour le rendre compréhensible?

```
#include <stdio.h>                                  |   /* suite */     /* Calcul de b en binaire  */
                                                    |   number=b;       /* meme chose que pour a ! */
/* Calcul de l'operation binaire a ET b en          |   ...
calculant les representations de a et de b */       |   /* Calcul de a ET b en binaire */
                                                    |
int main(void)                                      |   for(i=0;i<=nb_Bits-1;i=i+1)
{                                                   |   {
 int a,b,m;                                         |    if (a_Bits[i]+b_Bits[i]==2) { and_Bits[i]=1; }
 int i,nb_Bits=8;                                   |    else { and_Bits[i]=0; }
 int a_Bits[8], b_Bits[8], and_Bits[8];             |   }
 int number,remaining;                              |
                                                    |   /* Calcul en decimal de a ET b */
 printf("Donnez moi les valeurs de a et b\n");      |
 scanf("%d",&a); scanf("%d",&b);                    |   number=0; m=1;
                                                    |   for(i=nb_Bits-1;i>=0;i=i-1)
 number=a;       /* Calcul de a en binaire */       |   {
 for(i=nb_Bits-1;i>=0;i=i-1)                        |    number=number+(m*and_Bits[i]);
 {                                                  |    m=m*2;
  remaining=number%2;                               |   }
     number=number/2;                               |   printf("a Et b: %d\n",number);
  a_Bits[i]=remaining;                              |   return 0;
 }                                                  |  }
```

Un peu compliqué?

Structuration des algorithmes (3)

▷ **Elément de réponse**

Utiliser des fonctions et des procédures!

```
#include <stdio.h>     /* Calcul de l'operation binaire a ET b en calculant */
                       /* les representations de a et de b                  */

/* ... Il manque ici les definitions des fonctions: convBinary, andBinary et convDecimal */

int main(void)
{
 int a, b, number;
 int a_Bits[8], b_Bits[8], and_Bits[8];

 printf("Donnez moi les valeurs de a et b\n");
 scanf("%d",&a); scanf("%d",&b);

 convBinary(a,a_Bits);              /* Appel de fonction: conversion binaire de a */
 convBinary(b,b_Bits);              /* Appel de fonction: conversion binaire de b */
 andBinary(a_Bits,b_Bits,and_Bits); /* Appel de fonction: calcul ET binaire       */
 number = convDecimal(and_Bits);    /* Appel de fonction: calcul valeur decimale  */

 printf("a Et b: %d\n",number);
 return 0;
}
```

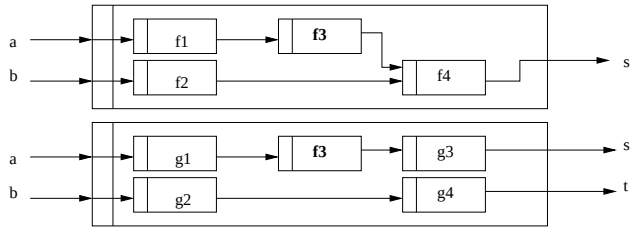
C'est bien plus simple...

Réutilisabilité des algorithmes (1)

"Paramétrer pour réutiliser"

▷ **Réutilisation**

Les fonctions et procédures sont réutilisables : si nous disposons d'une fonction capable d'effectuer un certain algorithme, nous pouvons la réutiliser un peu partout dans nos programmes sans avoir à la ré-écrire à chaque fois que l'on doit exécuter l'algorithme qu'elle encapsule.



Réutilisabilité des algorithmes (2)

▷ **Problème**

Comment réutiliser un algorithme existant sans avoir à le ré-écrire?

```
#include <stdio.h>
                                  |
int main(void)                    |
{                                 |
 int n,r,i;                       |
                                  |
 n = 3;                           |
 r = 1;                           |
 for(i=1;i<=n;i=i+1)              |
 {                                |
  r = r*i;                        |
 }                                |
                                  |
 printf("%d!=%d\n",n,r);          |
                                  |
```

```
/* suite */

n = 5;
r = 1;
for(i=1;i<=n;i=i+1)
{
 r = r*i;
}

printf("%d!=%d\n",n,r);

return 0;
}
```


Réutilisabilité des algorithmes (3)

▷ **Elément de réponse**

Encapsuler le code dans des fonctions ou des procédures.

```
#include <stdio.h>
                                  |
                                  |   int main(void)   /* suite */
int factorielle(int n)            |   {
{                                 |    int n, r;
 int r, i;                        |
                                  |    n = 3;
 r=1;                             |    r = factorielle(n);
 for(i=1;i<=n;i=i+1)              |    printf("%d!=%d\n",n,r);
 {                                |
  r = r*i;                        |    n = 5;
 }                                |    r = factorielle(n);
                                  |    printf("%d!=%d\n",n,r);
 return r;                        |
}                                 |    return 0;
                                  |   }
```


Procédure

▷ **Définition:** Une procédure est un bloc d'instructions nommé et paramétré qui **ne retourne pas** de valeur.

▷ **Exemple**

Type: `void printInt(int val); /*` attend un `int` et ne retourne rien (`void`) `*/`

Code:
```
void printInt(int val)
{
 printf("%d",val);
}
```

Appel:
```
#include <stdio.h>                  /* Pour l'utilisation de printf...  */
 void printInt(int val)
 {
  printf("%d",val);
 }
 int main(void)
 {
  int i = 17;
  printInt(i);
  return 0;
 }
```


Fonctions

▷ **Définition:** Une fonction est un bloc d'instructions nommé et paramétré qui **retourne** une valeur.

▷ **Exemple**

Type: `double sin(double angle); /*` attend un angle (réel) et en retourne le sinus (réel) `*/`

Code: présent dans la librairie mathématique $\Longrightarrow$ `-lm` (pour l'édition de liens)

Appel:
```
#include <stdio.h>              /* Pour l'utilisation de printf...  */
#include <math.h> /* Pour l'utilisation de double sin(double angle); */
int main(void)
{
 double angle = 3.14, s;
 s = sin(angle);            /* la valeur peut être affectée,affichée,utilisée...*/
 printf("sin(%g)=%g\n",angle,s);
 return 0;
}
```

C : déclaration – définition

En C, on ne parle que de fonction ...
... procédure ↔ fonction ne retournant rien: `void`

Eléments de vocabulaire en C:

▷ **La déclaration d'une fonction : son Type !**
⟹ caractéristiques en vue de son utilisation
⟹ Exemple : `void printInt(int val);`
Utilisation : `printInt(i);` avec `i` un `int`

▷ **La définition d'une fonction : son Code !**
⟹ Exemple :
```
void printInt(int val)
{
 printf("%d",val);
}
```


Spécification d'un algorithme

▷ **Nom :** un identificateur suffisamment explicite

▷ **Description :** une phrase qui dit ce que fait l'algorithme

▷ **Paramètres :** la liste des paramètres d'entrée-sortie de l'algorithme

▷ **Préconditions :** une liste d'expressions booléennes qui précisent les conditions d'application de l'algorithme

▷ **Appel :** des exemples d'utilisation de l'algorithme avec les résultats attendus

Exemple de spécification d'un algorithme

▷ **Nom :** sommeArithmetique

▷ **Description :** calcule la somme s des n premiers nombres entiers positifs :

$$s = 1 + 2 + 3 + ... + n = \sum_{i=1}^{i=n} i$$

▷ **Paramètres d'entrée-sortie :**

▷ **Précondition :** $n \in N^*$

Exemple de spécification d'un algorithme (suite)

▷ **Appel :** `s = sommeArithmetique(n)`

Exemples :
`s = sommeArithmetique(1)` ⟶ 1
`s = sommeArithmetique(5)` ⟶ 15
`s = sommeArithmetique(57)` ⟶ 1653
...
`s = sommeArithmetique(n)` ⟶ $\frac{n.(n+1)}{2}$

Spécification d'un algorithme : les étapes

Les 5 étapes de la spécification d'un algorithme

◇ donner un nom explicite à l'algorithme ;

◇ décrire par une phrase ce que fait l'algorithme ;

◇ définir les paramètres d'entrée-sortie de l'algorithme ;

◇ préciser les préconditions sur les paramètres d'entrée ;

◇ donner des exemples d'utilisation et les résultats attendus.

Implémentation d'un algorithme

▷ **Ecriture de l'en-tête et des paramètres**

▷ **Indication d'une valeur de retour (si fonction)**

▷ **Ecriture des préconditions**

▷ **Définition d'éventuelles variables auxiliaires**

▷ **Ecriture de l'algorithme**

▷ **Tests**

Sans oublier : commentaires et indentations !...
...et pas plus de 80 caractères par lignes.

Implémentation et lisibilité

Les variables auxiliaires

Exemples : ◇ `double delta = b*b - 4*a*c;`
◇ `int s = 0;`

▷ **Lisibilité d'un algorithme**

⟹ Définir et utiliser des variables auxiliaires...
...pour augmenter la lisibilité d'un algorithme !

▷ **Stockage de calculs intermédiaires**

⟹ Définir et utiliser des variables auxiliaires...
...pour éviter de faire plusieurs fois le même calcul !

Exemple d'implémentation : en-tête de la somme arithmétique

```
int sommeArithmetique(int n)
{









 /* ... */

 /* Fonction : il faut une valeur de retour */
}
```

Exemple d'implémentation : valeur de retour de la FONCTION somme arithmétique

```
int sommeArithmetique(int n)
{
 int s;









 /* ... */

 return s; /* Valeur de retour de la fonction */
}
```


Exemple d'implémentation : précondition de la somme arithmétique

```
int sommeArithmetique(int n)
{
 int s;

 if (n<0)
 {
  printf("Erreur parametre sommeArithmetique\n");
  exit(1); /* Arret du programme */
 }

 /* ... */

 return s; /* Valeur de retour de la fonction */
}
```


Exemple d'implémentation : variables auxiliaires (si besoin) de la somme arithmétique

```
int sommeArithmetique(int n)
{
 int s;
   /* Ici, eventuellement, declaration d'autres variables... */
 if (n<0)
 {
  printf("Erreur parametre sommeArithmetique\n");
  exit(1); /* Arret du programme */
 }

 /* ... */

 return s; /* Valeur de retour de la fonction */
}
```


Exemple d'implémentation : algorithme de la somme arithmétique

▷ **Version formule**

```
s = n*(n+1)/ 2;
```

▷ **Version itérative**

```
s = 0;                    /* Tout comme s, l'entier i doit etre */
for(i=0;i<=n;i=i+1)       /* defini au debut de la fonction     */
{
  s = s + i;
}
```

▷ **Version récursive**

```
if (n==0) { s = 0; }
else { s = n + sommeArithmetique (n-1); }
```

Plusieurs implémentations peuvent correspondre à la même spécification !

Exemple d'implémentation : algorithme de la somme arithmétique

```
int sommeArithmetique(int n)
{
 int s;

 if (n<0)
 {
  printf("Erreur parametre sommeArithmetique\n");
  exit(1); /* Arret du programme */
 }

 s = n*(n+1)/2;

 return s; /* Valeur de retour de la fonction */
}
```


Implémentation et validation (jeu de tests) : algorithme de la somme arithmétique

▷ **Tester les préconditions !**

```
#include <stdio.h>   /* Pour printf */
#include <stdlib.h>  /* Pour exit   */

/* Le code de la fonction sommeArithmetique */
/* ... */

int main(void)
{
 int r;

 r = sommeArithmetique(-1);  /* => Erreur parametre sommeArithmetique */

 printf("%d\n",r);
 return 0;
}
```


Implémentation et validation (jeu de tests) : algorithme de la somme arithmétique

▷ **Tester l'algorithme !**

```
#include <stdio.h>   /* Pour printf */
#include <stdlib.h>  /* Pour exit   */

/* Le code de la fonction sommeArithmetique */
/* ... */

int main(void)
{
 int i,r;
 for(i=0;i<=10;i=i+1)
 {
   r = sommeArithmetique(i); /* => 0 1 3 6 10 15 21 28 36 45 55 */
   printf("%d ",r);
 }
 printf("\n");
 return 0;
}
```


Implémentation et validation (jeu de tests) : algorithme de la somme arithmétique

▷ **Validité :** un algorithme doit être valide, conforme à sa spécification...

Implémentation d'un algorithme : les étapes

Les 6 principales étapes de l'implémentation d'un algorithme

◇ donner l'en-tête de la fonction ou de la procédure ;

◇ indiquer l'éventuelle valeur de retour (si fonction) ;

◇ écrire les préconditions ;

◇ définir les éventuelles variables auxiliaires ;

◇ écrire l'algorithme ;

◇ tester l'algorithme.

Algorithme : Spécification + Implémentation (1)

▷ **Spécification**

◇ donner un nom explicite à l'algorithme ;

◇ décrire par une phrase ce que fait l'algorithme ;

◇ définir les paramètres d'entrée-sortie de l'algorithme ;

◇ préciser les préconditions sur les paramètres d'entrée ;

◇ donner des exemples d'utilisation et les résultats attendus.

▷ **Implémentation** ... en lien avec la spécification !

◇ donner l'en-tête de la fonction ou de la procédure ;

◇ indiquer l'éventuelle valeur de retour (si fonction) ;

◇ écrire les préconditions ;

◇ définir les éventuelles variables auxiliaires ;

◇ écrire l'algorithme ;

◇ tester l'algorithme.

Algorithme : Spécification + Implémentation (2)

▷ **Propriétés d'un algorithme**

◇ validité : être conforme aux jeux de tests

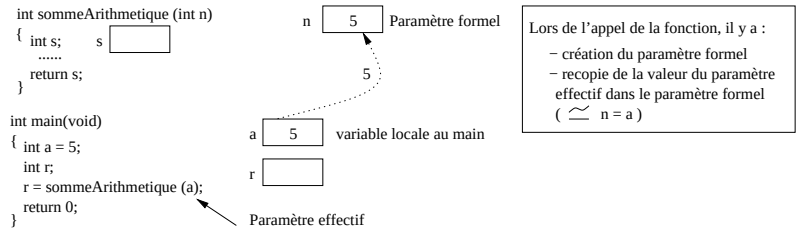
◇ robustesse : vérifier les préconditions

◇ réutilisable : être correctement paramétré

Passage de paramètre : par valeur (1)

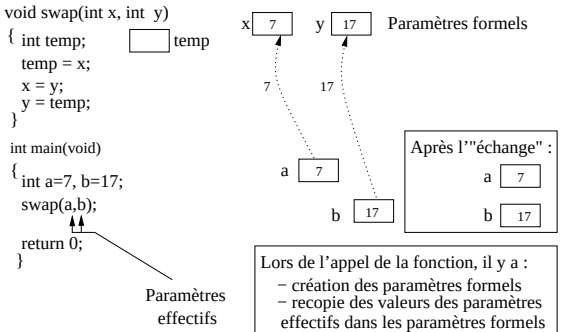
▷ Lors de la spécification d'un algorithme, il est possible d'indiquer des paramètres servant uniquement en **entrée**.

▷ Lors de l'appel de la fonction, les paramètres sont alors transmis **par valeur**.



Passage de paramètre : par valeur (2)

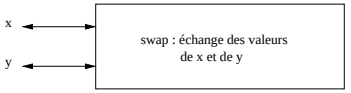
Autre exemple : fonction d'échange des valeurs de 2 variables **(ERREUR)**



MODIFICATION DE VARIABLE IMPOSSIBLE LORS D'UN PASSAGE PAR VALEUR

Passage de paramètre : par adresse (1)

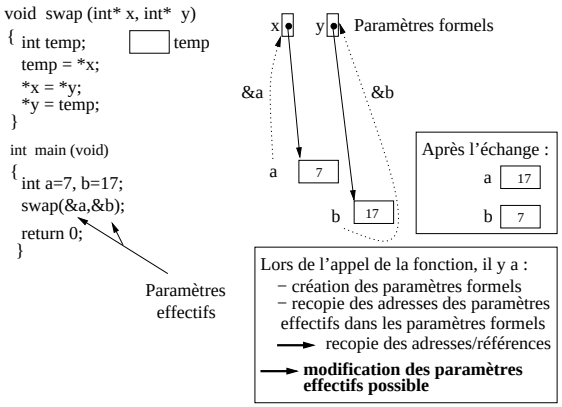
▷ Lors de la spécification d'un algorithme, il est possible d'indiquer des paramètres servant à la fois en **entrée** et en **sortie**.



▷ Lors de l'appel de la fonction, les paramètres sont alors transmis **par adresse**.

▷ Remarque : dans certains langages, il existe également le passage de paramètres par **référence** (assez similaire au passage par adresse).

Passage de paramètre : par adresse (2)



MODIFICATION DE VARIABLE POSSIBLE LORS D'UN PASSAGE PAR ADRESSE

Passage de paramètre : par adresse (3)

Résumé

En C, pour faire un passage de paramètre par adresse, il faut :

▷ Mettre lors de l'appel de la fonction:
un `&` suivi du nom du paramètre effectif.
$\Longrightarrow$ Permet d'indiquer que le paramètre effectif est modifiable
Exemple : `swap(&a,&b);`

▷ `&a` signifie "*l'endroit où l'on peut trouver la variable* `a`"

▷ `&b` signifie "*l'endroit où l'on peut trouver la variable* `b`"

Passage de paramètre : par adresse (4)

Résumé

En C, pour faire un passage de paramètre par adresse, il faut (suite) :

- ▷ Mettre dans l'en–tête de la fonction:
  un **type** suivi d'une `*` et le nom du paramètre formel.
  Exemple : `void swap(int* x, int* y)`
- ▷ Mettre dans le corps de la fonction:
  une `*` suivie du nom du paramètre formel.
  $\Longrightarrow$ Permet d'obtenir la valeur du paramètre effectif ou de modifier sa valeur.
  Exemple : `*x = *y;`
- ▷ `x` signifie "*l'endroit où l'on peut trouver la VRAIE variable*"
- ▷ `*x` signifie "*aller à l'endroit où l'on peut trouver la VRAIE variable*"
- ▷ idem pour `y` et `*y`

Passage de paramètre : par adresse (5)

Résumé

```
#include <stdio.h>

void swap(int* x,int* y)
{
 int temp;
 temp = *x;
 *x   = *y;
 *y   = temp;
}

int main(void)
{
 int a=7,b=17;
 printf("a=%d b=%d\n",a,b); /* a=7 b=17 */
 swap(&a,&b);
 printf("a=%d b=%d\n",a,b); /* a=17 b=7 */

 return 0;
}
```


Procédures et fonctions: comment choisir ?

Choix entre l'écriture de fonctions ou de procédures

Rappel: procédure $\leftrightarrow$ fonction ne retournant rien... `void`

- ▷ **Dépend du nombre de valeurs attendues par la fonction appelante :**
  - ◇ 0 : procédure ! ex: `void printInt(int val);`
  - ◇ 1 : fonction ! ex: `int sommeArithmetique(int n);`
  - ◇ >1 : en général procédure... ex: `void swap(int* x,int* y);`

  Sauf pour indiquer un code de retour afin de savoir si le traitement a été mené à bien sans erreur...
  ... ex: `int divise(int a, int b, int* aDivb);`

C : exemples de type de fonctions (1)

```
#include <stdio.h>
#include <stdlib.h>
void printInt(int val)          /* Fonction avec parametre et ne retournant rien... void */
{
 printf("%d",val);
}
int sommeArithmetique(int n)    /* Fonction avec parametre et retournant quelque chose   */
{
 int s;
 if (n<0) { printf("Erreur parametre sommeArithmetique\n"); exit(1);  /* Arret programme */
 }
 s = n*(n+1)/2;
 return s;
}
int main(void)
{
 int resultat;
 resultat=sommeArithmetique(5);
 printInt(resultat);
 return 0;
}
```

C : exemples de type de fonctions (2)

```
#include <stdio.h>
/* Fonction divise: - parametres d'entree: 2 entiers a et b
   ==============  - parametre de sortie: 1 entier connu par son adresse aDivb
                   - calcule la division de a par b... si c'est possible !
   Code de retour : 1 (vrai), si c'est possible ou 0 (faux), dans le cas contraire
*/
int divise(int a, int b, int* aDivb)                /* Fonction avec un code de retour */
{
 int ok = 0;
 if (b!=0) { ok=1; *aDivb = a/b; }
 return ok;
}
int main(void)
{
 int x=17, y=0, resultat;
 if (divise(x,y,&resultat)==1) {
        printf("Resultat de la division: %d\n",resultat);
 }
 else { printf("Attention, division par 0 !\n");
 }
 return 0;
}
```


Procédure/Fonction : écriture algorithmique en "français" (1)

- ▷ **Procédure/Fonction** : `procedure` ou `fonction`
- ▷ **Nom de l'algorithme** : `nom` de la fonction ou de la procédure
- ▷ **(**
- ▷ **Liste de paramètres séparés par des ,**
  - ◇ **paramètre en entrée uniquement :** `In type nom`
  - ◇ **paramètre en entrée/sortie :** `InOut type nom`
- ▷ **)**
- ▷ **Si fonction indiquer** : `type` de la valeur de retour
- ▷ **debut**
- ▷ **...**
- ▷ **fin**

Procédure/Fonction : écriture algorithmique en "français" (2)

Exemples:

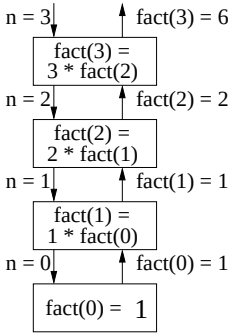
```
▷ fonction sommeArithmetique (In int n) : int
  debut
   soit int s                          (* variable auxiliaire *)
     si (n < 0) alors Erreur de Parametre    (* precondition *)
     s <- n*(n+1) / 2
     retourner s;
  fin
▷ procedure swap (InOut int x, IntOut int y)
  debut
   soit int temp                       (* variable auxiliaire *)
     temp <- x
     x    <- y
     y    <- temp
  fin
```


Fonctions récursives (1)

Fonction récursive : fonction qui, au cours de son exécution fait appel à elle-même (directement ou non).

Exemple : la factorielle, n! = 1∗2∗...∗n Rappel : 0! = 1

```
int fact(int n)
{
 int resultat;

 if (n<0) { printf("fact: erreur parametre\n");
            exit(1); /* Arret du programme */
 }

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



Fonctions récursives (2)

Avantage : programmation "facile" de certains algorithmes (tri rapide,...)
Inconvénient : pas forcément très efficace sur tous les problèmes.
Il existe des méthodes de "dé-récursivation".

```
int fact(int n)
{
 int indice, resultat;

 if (n<0) { printf("fact: erreur parametre\n");
            exit(1); /* Arret du programme */
 }

 resultat = 1;
 indice   = 1;
 while (indice <=n)
 {
  resultat = indice * resultat;
  indice   = indice + 1;
 }

 return resultat;
}
```

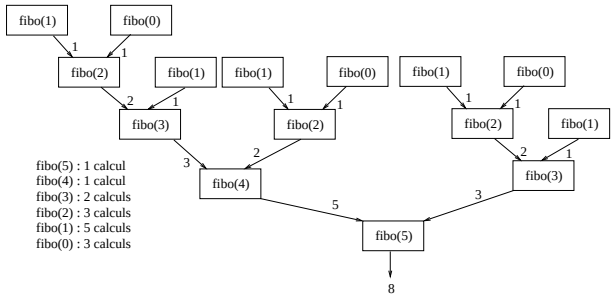

Fonctions récursives (3)

Autre exemple : la suite de Fibonacci

$$\begin{cases} \text{Fibonacci}(0) = 1 \text{ et Fibonacci}(1) = 1 \\ \text{Fibonacci}(n) = \text{Fibonacci}(n\text{-}2) + \text{Fibonacci}(n\text{-}1) \end{cases}$$

```
int fibo(int n)
{
 int resultat;

 if (n<0) { printf("fibo: erreur parametre\n");
            exit(1); /* Arret du programme */
 }

 if (n==0 || n==1)
 {
   resultat = 1;
 }
 else
 {
  resultat = fibo(n-1) + fibo(n-2);
 }

 return resultat;
}
```


Fonctions récursives (4)



Fonctions récursives (5)

Version non récursive :

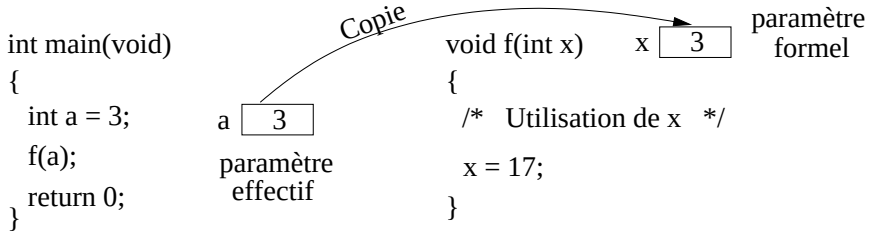
```
int fibo(int n)
{
 int resultat;

 if (n<0) { printf("fibo: erreur parametre\n");
            exit(1); /* Arret du programme */
 }

 if (n==0 || n==1) { resultat = 1; }
 else
 {
  int indice,un,un_1,un_2;
  un_1   = 1;
  un_2   = 1;
  indice = 2;
  while (indice <= n)
  {
   un   = un_1 + un_2;
   un_2 = un_1;
   un_1 = un;
   indice = indice + 1;
  }
  resultat = un;
 }

 return resultat;
}
```

Structures de données de base

Un petit retour sur les paramètres modifiables des fonctions en C

Lors de l'appel d'une fonction,
les arguments sont transmis **par valeur**.

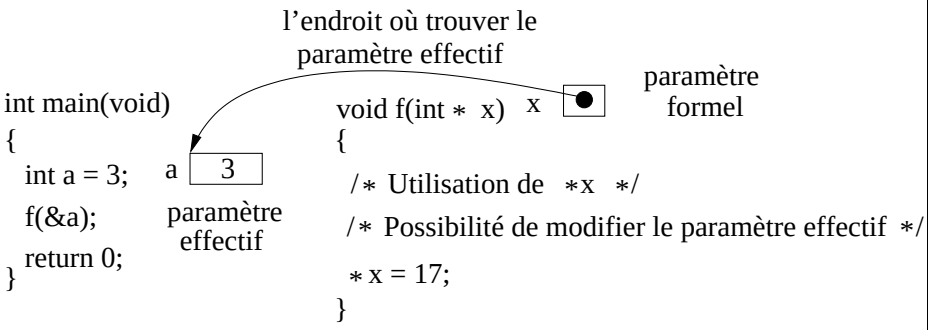
$\Longrightarrow$ elle ne reçoit qu'une **copie provisoire** de chaque argument.



$\Longrightarrow$ la fonction ne peut modifier ses arguments ! (In)

Un petit retour sur les paramètres modifiables des fonctions en C

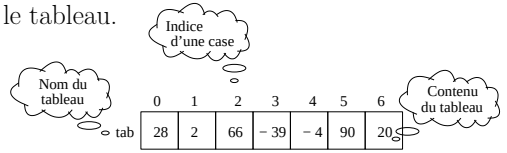
$\Longrightarrow$ SOLUTION: transmettre l'endroit où se trouve la variable et donc permettre la modification de cette variable dans la fonction (InOut)



Les tableaux

Définition :

Un tableau *tab* est une **suite de n éléments**,
accessibles par leur **rang** i dans le tableau.



Intérêts :

▷ Stockage d'un ensemble de **données de même type**.

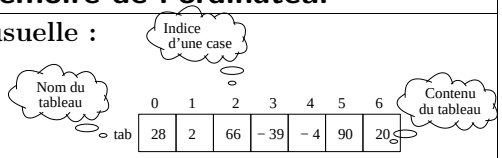
▷ **Traitements globaux** sur ces données.

Exemple d'utilité des tableaux :

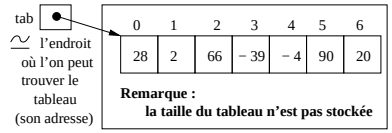
Demander à l'utilisateur de saisir une suite de nombres. Afficher cette suite après avoir divisé tous les nombres par la valeur maximale de la suite.
$\Longrightarrow$ Il faut conserver les nombres en mémoire en vue d'un traitement global.

Les tableaux en mémoire de l'ordinateur

Représentation simplifiée et usuelle :



Représentation en mémoire (en langage C)




```
int tab[7]={28,2,66,-39,-4,90,20};
int taille=7;
```

Et donc :

- ▷ La longueur du tableau : nombre de cases
  - ◇ Nombre défini lors de la création du tableau, non modifiable après.
  - ◇ Non stockée avec le tableau ⇒ doit être connue du programmeur
- ▷ Les indices valides : entiers de 0 à "longueur du tableau - 1"
- ▷ En algorithmique : la $(i+1)$ème case du tableau *tab* est notée $tab[i]$

Opérations usuelles sur les tableaux

Principales opérations sur les tableaux :

- ▷ **Création** d'un tableau d'une certaine taille qui doit être connue
- ▷ **Stockage** (écriture) d'une valeur dans une case d'un tableau
- ▷ **Consultation** (lecture) du contenu d'une case d'un tableau
- ▷ **Recherche** si une valeur est présente dans un tableau
- ▷ **Modification** de la façon dont les valeurs sont rangées dans le tableau (par exemple : tri du tableau, ...)
- ▷ **Opérations** entre tableaux (test d'égalité, affectation, ...)
- ▷ **Opérations** sur tous les éléments d'un tableau (par exemple : + 1 sur toutes les cases, ...)
- ▷ **...**

Et maintenant en C !

Les tableaux : création, taille, stockage, consultation

- ▷ **Création d'un tableau de taille n d'objets de type T :**
  - ◇ T `tab[`n`];` ...la valeur initiale de tous les éléments est indéterminée
  - ◇ T `tab[`n`] =` $\{e_0$ `,` e_1`, ... ,` $e_{n-1}\}$`;` ...les éléments doivent être connus
  - ◇ **Remarque :**
    - ○ la taille n doit être connue à la création du tableau
    - ○ utilisation de $\{e_0$ `,` e_1`, ... ,` $e_{n-1}\}$ **uniquement à la création**
- ▷ **Obtention de la taille d'un tableau : Pas possible !**
  - ◇ Rappel : la taille d'un tableau doit être connue par le programmeur
- ▷ **Stockage d'une valeur dans une case d'un tableau :**
  - ◇ Modification de la (i+1)ème case d'un tableau
  - ◇ `tab[i] = e` $0 \leq$ i $\leq$ La taille du tableau **- 1**
- ▷ **Consultation d'une case d'un tableau :**
  - ◇ Obtention de la valeur de la (i+1)ème case d'un tableau
  - ◇ `tab[i]` $0 \leq$ i $\leq$ La taille du tableau **- 1**

Les tableaux : création, taille, stockage, consultation

Exemple 1 : créations, consultations et modifications de tableaux

```
int main(void)
{
 int tab1[3] = {10, 20, 30};   /* 10, 20, 30 */
 int tab2[3];                  /*  ?,  ?,  ? */
 int i, taille = 3;

 for(i=0;i<=taille-1;i=i+1)    /* de 0 a taille-1 */
 {
  tab2[i]=tab1[i]/10;

  /* trace */
 }

 return 0;
}
```

Les tableaux : création, taille, stockage, consultation

Exemple 1 : créations, consultations et modifications de tableaux (suite)

Traces du programme :

```
tab1          taille   tab2       i

{10 20 30}    3        {1 ? ?}    0
{10 20 30}    3        {1 2 ?}    1
{10 20 30}    3        {1 2 3}    2
```


Les tableaux : création, taille, stockage, consultation

Exemple 2 : affichage d'un tableau

```
#include <stdio.h>

void printTab(int tab[], int size)              /* Pour les tableaux a 1 dimension, */
{                                               /*  dans l'en-tete d'une fonction,  */
 int i;                                         /*   il n'est pas obligatoire de    */
 printf("Les elements du tableau sont :\n");    /*  mettre la taille entre les []   */
 for(i=0;i<=size-1;i=i+1)
 {
  printf("%d ",tab[i]);
  /* trace */
 }
}

int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int taille = 5;

 printTab(t,taille);  printf("\n");
 return 0;
}
```


Les tableaux : création, taille, stockage, consultation

Exemple 2 : affichage d'un tableau (suite)

Traces du programme :

```
tab               size      i   Affichage

{90 67 2 50 23}   5         0   90
{90 67 2 50 23}   5         1   90 67
{90 67 2 50 23}   5         2   90 67 2
{90 67 2 50 23}   5         3   90 67 2 50
{90 67 2 50 23}   5         4   90 67 2 50 23
```


Les tableaux : création, taille, stockage, consultation

Exemple 3 : recherche du plus petit élément d'un tableau

```
#include <stdio.h>
#include <stdlib.h>    /* exit */

int getMinTab(int tab[], int size)
{
 int i, min;
 if (size==0) { printf("getMinTab: tab est vide\n"); exit(1); }      /* precondition */
 min = tab[0];
 for(i=1;i<=size-1;i=i+1)
 {
  if (tab[i] < min) { min = tab[i]; }
  /* trace */
 }
 return min;
}

int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int taille = 5, m;

 m = getMinTab(t,taille);  printf("Min du tableau:%d\n", m);
 return 0;
}
```

Les tableaux : création, taille, stockage, consultation

Exemple 3 : recherche du plus petit élément d'un tableau (suite)

Traces du programme :

```
                        tab                 min

Avant la boucle on a : {90 67 2 50 23}      90

Pendant la boucle :

tab                 size    min    i

{90 67 2 50 23}     5        67    1
{90 67 2 50 23}     5         2    2
{90 67 2 50 23}     5         2    3
{90 67 2 50 23}     5         2    4
```


Les tableaux : création, taille, stockage, consultation

Exemple 4 : recherche d'une valeur dans un tableau

```
#include <stdio.h>

/* parametres d'entree :  - tab un tableau d'int
                          - size la taille du tableau
                          - valeur, la valeur a chercher
   parametres de sortie : - indice, l'endroit ou trouver un int pour
                            stocker l'indice de la valeur dans tab

   Retourne 1 (vrai) si la valeur est presente dans le tableau, sinon 0 (faux)

   Remarque : le tableau tab ne doit pas etre forcemment trie                  */

int rechercheValeur(int tab[], int size, int valeur, int* indice)
{
 int i, trouve;

 trouve = 0;  /* faux */
 i      = 0;
 while (trouve!=1 && i<=size-1)  /* Identique a : while (!trouve && i<=size-1) */
 {
  if (tab[i] == valeur) { trouve  = 1; /* vrai */
                          *indice = i;
                        }
  /* trace */
  i = i + 1;
 }
 return trouve;
}
```


Les tableaux : création, taille, stockage, consultation

Exemple 4 : recherche d'une valeur dans un tableau (suite)

```
int main(void)
{
 int t[5] = {90, 67, 2, 50, 23};
 int taille = 5, val, ind;

 printf("Valeur a rechercher dans le tableau ? "); scanf("%d",&val);

 if (rechercheValeur(t,taille,val,&ind)==1)    /* if (rechercheValeur(t,taille,val,&ind)) */
 {
  printf("%d se trouve a l'indice %d\n",val,ind);
 }
 else
 {
  printf("%d ne se trouve pas dans le tableau\n",val);
 }

 return 0;
}
```


Les tableaux : création, taille, stockage, consultation

Exemple 4 : recherche d'une valeur dans un tableau (suite)

Traces du programme si l'utilisateur entre la valeur 50 :

```
tab               size   i   valeur  trouve  *indice

{90 67 2 50 23}   5      0   50      0       ?
{90 67 2 50 23}   5      1   50      0       ?
{90 67 2 50 23}   5      2   50      0       ?
{90 67 2 50 23}   5      3   50      1       3

Remarque: on ne rentre pas dans la boucle pour i = 4...
```

Passage de tableaux en paramètres

Remarque : dans les exemples précédents (2 à 4) les tableaux étaient considérés comme des paramètres d'entrée des fonctions.

Tableau en paramètre d'entrée/sortie de fonction :

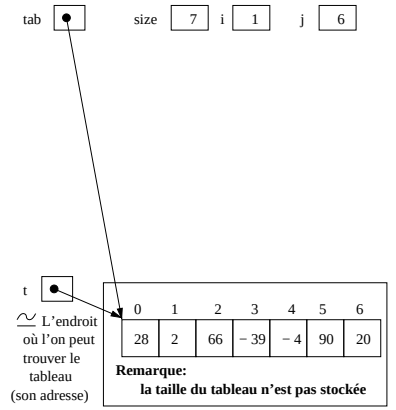
▷ le cas de la fonction d'échange de 2 valeurs d'un tableau ...un premier pas vers le tri !

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

void echange(int tab[], int size, int i, int j)
{
 int temp;

 if (!(i>=0 && i<=size-1)) { printf("Parametre i invalide\n"); exit(1); }
 if (!(j>=0 && j<=size-1)) { printf("Parametre j invalide\n"); exit(1); }

 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}
```

Pourquoi ça marche ????

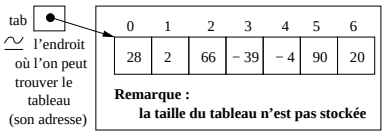
Passage de tableaux en paramètres

```
void echange(int tab[], int size, int i, int j)
{
 int temp;
                        /* Les preconditions ... */
 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}
int main(void)
{
 int t[7] = {28, 2, 66, -39, -4, 90, 20};
 int taille = 7;

 printTab(t,taille); /* 28 2 66 -39 -4 90 20 */

 echange(t,taille,1,6);

 printTab(t,taille); /* 28 20 66 -39 -4 90 2 */

 return 0;
}
```



C'est l'endroit où se trouve le tableau qui est transmis!

Passage de tableaux en paramètres



Et donc :

▷ Lors d'un appel de fonction avec un paramètre de type tableau, les valeurs du tableau peuvent être changées.

▷ Le tableau est alors un paramètre d'entrée et un paramètre de sortie.

Opérations entre tableaux : affectation, test d'égalité

▷ **Affectation entre tableaux :** **Pas d'affectation directe!**

```
void copieTab(int tabDes[], int tabSrc[], int size)
{
 int i;
 for(i=0; i<=size-1; i=i+1)
 {
  tabDes[i]=tabSrc[i];      /* tabDes et tabSrc doivent avoir la meme taille */
 }
}
```

▷ **Test d'égalité entre tableaux :** **Pas de test direct!**

```
/* 1 (vrai) : si tab1 = tab2, 0 (faux): si tab1!=tab2 */
int testTab(int tab1[], int tab2[], int size)
{
 int i=0, test = 1;  /* vrai */

 while (i<=size-1 && test==1)       /* Identique a : while (i<=size-1 && test) */
 {
  if (tab1[i]!=tab2[i]) { test = 0; /* faux */ }
  i = i + 1;
 }
 return test;
}
```

Opérations entre tableaux : affectation, test d'égalité

Et donc...

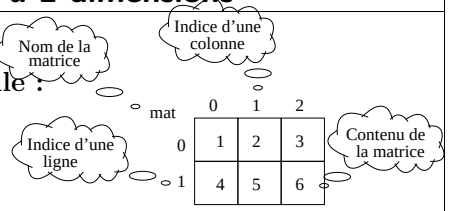
```
int main(void)
{
 int t1[3] = {30, 40, 50};
 int t2[3];

 t2 = t1              /* ____ INTERDIT ____ ( pas d'affectation entre tableaux )       */

 t2 = {30, 40, 50};   /* ____ INTERDIT ____ ( {.. , .., ..} uniquement a la creation ) */

 if (t1==t2)          /* ____ INTERDIT ____ ( pas de test direct de tableaux )         */
 {
  ...
 }
 else
 {
  ...
 }
 return 0;
}
```


Opérations entre tableaux : affectation, test d'égalité

```
#include <stdio.h>
void printTab(int tab[], int size)
{ /* le code de printTab ... */ }
void copieTab(int tabDes[], int tabSrc[], int size)
{ /* le code de copieTab ... */ }
int testTab(int tab1[], int tab2[], int size)
{ /* le code de testTab ... */ }
int main(void)
{
 int t1[7] = {28, 2, 66, -39, -4, 90, 20};
 int t2[7];
 int taille = 7;

 printTab(t1,taille);   /* 28 2 66 -39 -4 90 20 */

 copieTab(t2,t1,taille);

 printTab(t2,taille);   /* 28 2 66 -39 -4 90 20 */
                                                  /* Identique a : if (testTab(t1,t2,taille)) */
 if (testTab(t1,t2,taille)==1) { printf("t1 est egal a t2\n"); }
 else                          { printf("t1 n'est pas egal a t2\n"); }

 return 0;
}
```

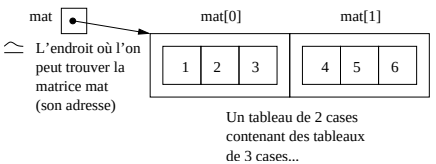

Matrices: tableaux à 2 dimensions

Représentation simplifiée et usuelle :



Représentation en mémoire (en langage C)

```
int mat[2][3]={ {1, 2, 3}, {4, 5, 6} };
int nbLignes=2;
int nbColonnes=3;
```



$\Longrightarrow$ **Matrice : tableau de tableaux**

Matrices: création et accès

▷ **Création d'une matrice de taille nXm d'objets de type T :**

◇ T `mat[n][m];` ...la valeur initiale de tous les éléments est indéterminée

◇ Mais on peut aussi indiquer l'ensemble des tableaux (les lignes) constitutant la matrice : T `mat[n][m] =` $\{ \{e_{00}, e_{01}, \ldots, e_{0m-1}\}, \{e_{10}, e_{11}, \ldots, e_{1m-1}\}, \ldots , \{e_{n-10}, e_{n-11}, \ldots, e_{n-1m-1}\} \};$

◇ **Remarque :**

- la taille nXm doit être connue à la création de la matrice
- utilisation de $\{ \{\ldots\}, \ldots, \{\ldots\} \}$ **uniquement à la création**

▷ `mat` **étant un tableau de tableaux :**

◇ **Accès aux différentes lignes :** `mat[l]` `mat[l]` est un tableau...

▷ `mat[l]` **étant un tableau :**

◇ **Accès aux différentes colonnes (et donc cases) :** `mat[l][c]`

Matrices: création et accès

Exemple 1 : création, consultations et modifications de matrices

```
int main(void)
{
 int mat1[2][3] = {{10, 20, 30} , {40, 50, 60}};   /* tableau de tableaux */
 int mat2[2][3]; /*  ?   ?   ?      ?   ?   ?   */
 int nbLignes = 2, nbCols = 3;
 int l,c;

 for(l=0;l<=nbLignes-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   mat2[l][c] = mat1[l][c]/10;
   /* trace */
  }
 }

 return 0;
}
```


Matrices: création et accès

Exemple 1 : création, consultations et modifications de matrices (suite)

Traces du programme :

```
l   c   mat1                      mat2

0   0   {{10 20 30} {40 50 60}}   {{1 ? ?} {? ? ?}}
0   1   {{10 20 30} {40 50 60}}   {{1 2 ?} {? ? ?}}
0   2   {{10 20 30} {40 50 60}}   {{1 2 3} {? ? ?}}
1   0   {{10 20 30} {40 50 60}}   {{1 2 3} {4 ? ?}}
1   1   {{10 20 30} {40 50 60}}   {{1 2 3} {4 5 ?}}
1   2   {{10 20 30} {40 50 60}}   {{1 2 3} {4 5 6}}
```


Matrices: création et accès

Exemple 2 : affichage d'une matrice

```
#include <stdio.h>

void printMat(int mat[2][3], int nbLignes, int nbCols) /* Tableau a plusieurs    */
{                                                      /* dimensions :           */
 int l,c;                                              /* obligatoire de mettre  */
                                                       /* la taille entre les [] */
 for(l=0;l<=nbLignes-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   printf("%d ",mat[l][c]);
   /* trace */
  }
  printf("\n");
 }
}
```


Matrices: création et accès

Exemple 2 : affichage d'une matrice (suite)

```
int main(void)
{
 int m[2][3] = { {1, 2, 3} , {4, 5, 6} };
 int nbLignes = 2, nbCols = 3;

 printMat(m,nbLignes,nbCols);

 return 0;
}
```

Rappel : c'est l'endroit où se trouve la matrice `m` qui est transmis à la fonction.

$\Longrightarrow$ Les éléments de la matrice sont potentiellement modifiables dans `printMat`.

Traces du programme :

```
l   c   mat                 Affichage

0   0   {{1 2 3} {4 5 6}}   1
0   1   {{1 2 3} {4 5 6}}   1 2
0   2   {{1 2 3} {4 5 6}}   1 2 3
1   0   {{1 2 3} {4 5 6}}   1 2 3 \n4
1   1   {{1 2 3} {4 5 6}}   1 2 3 \n4 5
1   2   {{1 2 3} {4 5 6}}   1 2 3 \n4 5 6
```

Matrices: création et accès

Exemple 3 : recherche du plus petit élément d'une matrice

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

int getMinMat(int mat[2][3], int nbLignes, int nbCols)
{
 int l,c, min;

 if (nbLignes==0||nbCols==0) { printf("Mat est vide !\n"); exit(1); }

 min=mat[0][0];
 for(l=0;l<=nbLignes-1;l=l+1)
 {
  for(c=0;c<=nbCols-1;c=c+1)
  {
   if (mat[l][c] < min) { min = mat[l][c]; }
   /* trace */
  }
 }

 return min;
}
```


Matrices: création et accès

Exemple 3 : recherche du plus petit élément d'une matrice (suite)

```
int main(void)
{
 int m[2][3] = { {90, 23, 5} , {4, 50, 67} };
 int nbLignes = 2, nbCols = 3;

 printf("Le min de la matrice est %d\n",getMinMat(m,nbLignes,nbCols));

 return 0;
}
```

Traces du programme :

```
l   c   mat                    min

0   0   {{90 23 5} {4 50 67}}   90
0   1   {{90 23 5} {4 50 67}}   23
0   2   {{90 23 5} {4 50 67}}   5
1   0   {{90 23 5} {4 50 67}}   4
1   1   {{90 23 5} {4 50 67}}   4
1   2   {{90 23 5} {4 50 67}}   4
```


Cas particuliers de tableaux : les chaînes de caractères

```
#include <stdio.h>
#include <string.h>    /* Pour strcpy, strcmp, ... */

int main(void)
{
 char str1[256]="salut";
 char str2[256];

 strcpy(str2,str1);                    /* str2=str1 INTERDIT */

 str1[0]='S';

 printf("str1=%s str2=%s\n",str1,str2);        /* str1=Salut str2=salut */

 if (strcmp(str1,str2)==0)             /* if (str1==str2) ... INTERDIT */
 {
  printf("str1 et str2 identiques\n");
 }
 else
 {
  printf("str1 et str2 differentes\n");
 }
 return 0;
}
```


```
/* '\0' : fin */
/* de chaine  */
```


Cas particuliers de tableaux : les chaînes de caractères

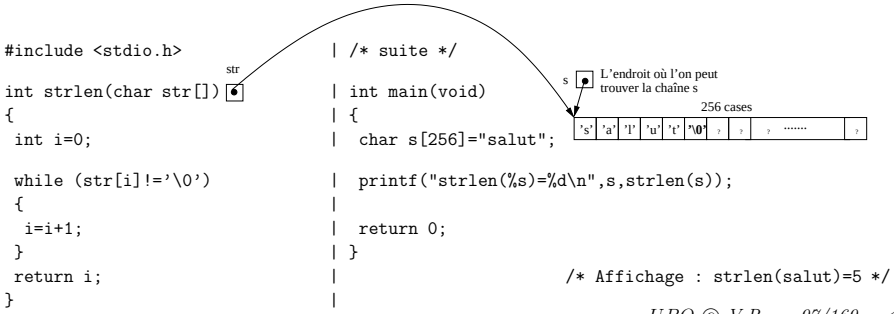
Et donc, comme une chaîne de caractères est un tableau de `char` :

- ▷ Accès au ième caractère d'une chaîne **`str`** :
  ... comme pour un tableau : **`str[i]`**
- ▷ Pour copier une chaîne **`str1`** dans une autre chaîne **`str2`**
  ... comme pour un tableau : **`str2=str1;`** est **INTERDIT**
  $\Longrightarrow$ **`strcpy(str2,str1)`**
- ▷ Pour tester l'égalité de 2 chaînes
  ... comme pour des tableaux : **`if (str1==str2) ...`** est **INTERDIT**
  $\Longrightarrow$ **`strcmp(str1,str2)`**
  `strcmp(str1,str2)` retourne :
  - ◇ 0 si **`str1`** et **`str2`** identiques
  - ◇ < 0 si **`str1`** est plus petit que **`str2`**
  - ◇ > 0 si **`str1`** est plus grand que **`str2`**

Cas particuliers de tableaux : les chaînes de caractères

Fonctions sur les chaînes de caractères

- ▷ Il existe un certains nombre de fonctions permettant de manipuler des chaînes. Exemples: `strcpy`, `strcmp`, ...
- ▷ Il existe également une fonction permettant de calculer la longueur d'une chaînes: `strlen`




```
#include <stdio.h>                | /* suite */
int strlen(char str[])            | int main(void)
{                                 | {
 int i=0;                         |  char s[256]="salut";
 while (str[i]!='\0')             |  printf("strlen(%s)=%d\n",s,strlen(s));
 {                                |
  i=i+1;                          |  return 0;
 }                                | }
 return i;                        |                    /* Affichage : strlen(salut)=5 */
}                                 |
```


Les enregistrements

Définitions :

- ▷ **Enregistrement**
  Un enregistrement est une structure de données composée d'éléments de types différents ;
- ▷ **Champ**
  Les éléments d'un enregistrement sont appelés champs ou membres.

Exemple :

- ▷ Personne : – nom
  – prénom
  – âge

Les enregistrements

Intérêts :

A partir de types déjà définis,

les enregistrements...
permettent de définir...
de nouveaux types de données.

$\Longrightarrow$ Structuration des données

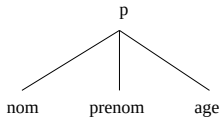
$\Longrightarrow$ Structuration des programmes

$\Longrightarrow$ Programmation et maintenance plus aisées, lisibilité accrue ...

Accès aux champs

Accès aux champs : `.`

Exemple : soit `p` un enregistrement représentant une personne particulière.



L'accès aux champs de `p` se fait ainsi: `p.nom`, `p.prenom`, `p.age`

A noter sur cet exemple:
- ▷ dans la pratique, l'âge ne serait pas stocké...
- ▷ on stockerait plutôt la date de naissance $\Rightarrow$ permet d'en déduire l'âge !

Les enregistrements en C

Exemple 1 : le type `struct personne`

```
#include <stdio.h>    /* Pour l'utilisation de printf... */
#include <string.h>   /* Pour l'utilisation de strcpy... */

struct personne       /* Definition d'un nouveau type : struct personne */
{
 char nom[256];
 char prenom[256];
 int  age;
};
```


Les enregistrements en C

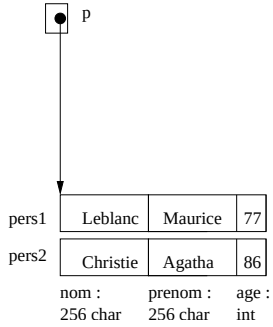
Exemple 1 : le type `struct personne (suite)`

```
struct personne createPersonne(char nom[], char prenom[], int age)
{
 struct personne p;

 strcpy(p.nom,nom);                      /* 1) p.nom    <= nom    */
 strcpy(p.prenom,prenom);                /* 2) p.prenom <= prenom */
 p.age = age;                            /* 3) p.age    <= age    */

 return p;
}

void printPersonne(struct personne p)
{
 printf("nom:    %s\n",p.nom);           /* 1) */
 printf("prenom: %s\n",p.prenom);        /* 2) */
 printf("age:    %d ans\n",p.age);       /* 3) */
}
```


Les enregistrements en C

Exemple 1 : le type `struct personne (suite)`

```
void anniversaire(struct personne* p)
{
 (*p).age = (*p).age + 1;

 printf("Bon anniversaire %s !\n",(*p).prenom);
}

int main(void)
{
 struct personne pers1 = {"Leblanc", "Maurice", 77};
 struct personne pers2;

 pers2 = createPersonne("Christie","Agatha",86);

 printf("Personne 1:\n"); printPersonne(pers1);
 printf("Personne 2:\n"); printPersonne(pers2);

 anniversaire(&pers1);        /* &pers1 : l'endroit ou l'on peut trouver pers1 */

 return 0;
}
```



Les enregistrements en C: test d'égalité, affectation (1)

▷ **Test d'égalité entre enregistrements**

→ Pas de test direct ! ←

$\Longrightarrow$ Il faut faire le **test des champs un par un**...

$\Longrightarrow$ et donc, avec le type `struct personne` :

```
/* Retourne 1 (vrai) : si p1=p2, 0 (faux): si p1!=p2 */
int testPersonne(struct personne p1, struct personne p2)
{
 int test = 1;  /* vrai */
 if (strcmp(p1.nom,p2.nom)!=0) { test = 0; /* faux */}
 else
 if (strcmp(p1.prenom,p2.prenom)!=0) { test = 0; /* faux */}
 else
 if (p1.age!=p2.age) { test = 0; /* faux */}
 return test;
}
```

Les enregistrements en C: test d'égalité, affectation (2)

▷ **Affectation entre enregistrements**

→ Affectation directe possible ! ←

$\Longrightarrow$ Et donc, avec le type `struct personne` : `pers2 = pers1;` est possible.

$\Longrightarrow$ cela correspond à une **affectation champs à champs**.

```
int main(void)
{
 struct personne pers1 = {"Leblanc", "Maurice", 77};
 struct personne pers2;

 pers2 = {"Leblanc", "Maurice", 77}; /* INTERDIT !!! */
                                     /* i.e : {.., .., ..} uniquement a la creation */

 pers2 = pers1;                      /* Affectation directe ok */

 if (testPersonne(pers1,pers2)==1)   /* Rappel : if (pers1==pers2) est INTERDIT !!! */
 {
  printf("C'est la meme personne !\n");
 }
 return 0;
}
```


Les enregistrements en C

Exemple 2 : le type `struct personne` + le type `struct ePersonne`

```
#include <stdio.h>    /* Pour l'utilisation de printf... */
#include <string.h>   /* Pour l'utilisation de strcpy... */

struct personne
{ ...
};
struct personne createPersonne(char nom[], char prenom[], int age)
{ ...
}
void printPersonne(struct personne p)
{ ...
}
void anniversaire(struct personne* p)
{ ...
}

struct ePersonne      /* Definition d'un nouveau type : struct ePersonne */
{
 struct personne p;
 char   email[256];
};
```


Les enregistrements en C

Exemple 2: le type `struct personne` + le type `struct ePersonne` (suite)

```
struct ePersonne createEPersonne(char nom[], char prenom[], int age, char email[])
{
 struct ePersonne ep;

 strcpy(ep.p.nom,nom);                  /* 1) ep.p.nom    <= nom    */
 strcpy(ep.p.prenom,prenom);            /* 2) ep.p.prenom <= prenom */
 ep.p.age = age;                        /* 3) ep.p.age    <= age    */
 strcpy(ep.email,email);                /* 4) ep.mail     <= mail   */

 /* Remarque: 1) + 2) + 3) ==> ep.p = createPersonne(nom,prenom,age); */

 return ep;
}

void printEPersonne(struct ePersonne ep)
{
 printf("nom:    %s\n",ep.p.nom);       /* 1) */
 printf("prenom: %s\n",ep.p.prenom);    /* 2) */
 printf("age:    %d ans\n",ep.p.age);   /* 3) */
 printf("email:  %s\n",ep.email);       /* 4) */

 /* Remarque: 1) + 2) + 3) ==> printPersonne(ep.p); */
}
```


Les enregistrements en C

Exemple 2: le type `struct personne` + le type `struct ePersonne` (suite)

```
int main(void)
{
 struct ePersonne ePers1={{"Leblanc", "Maurice", 77},"leblanc@univ-brest.fr"};
 struct ePersonne ePers2;

 ePers2=createEPersonne("Christie","Agatha",86,"christie@univ-brest.fr");

 printf("ePersonne 1:\n"); printEPersonne(ePers1);
 printf("ePersonne 2:\n"); printEPersonne(ePers2);

 anniversaire(&ePers1.p);  /* Identique a : anniversaire(&(ePers1.p)); */

 return 0;
}
```

Complexité des algorithmes

Définition et but
Exemples de calcul de coût
Mesure de complexité et comparaison d'algorithmes
Algorithmes exacts vs Heuristiques

Définition et but

Définition

La complexité est une mesure de la "difficulté" du calcul d'un algorithme en fonction de la taille `n` des données.

▷ On considère ici la **complexité en temps**
(= nombre d'opérations élémentaires nécessaires).

▷ Il existe également la **complexité en espace**
(= nombre de cases mémoire nécessaires).

But

Pour un problème donné, on cherche le meilleur algorithme :
▷ quels que soient le langage de programmation et la machine utilisée,

▷ avec des données de grande taille (exemple : des tableaux de grande taille)

Remarque : on ne cherche pas des "ruses" de programmation

Exemples de calcul de coût

Recherche séquentielle d'un élément `e` dans un tableau `tab` de `n` éléments :

```
int recherche(int tab[], int n, int e)      /* n : taille du tableau tab */
{                                           /* e : valeur a chercher     */
 int i, trouve;

 trouve = 0;  /* faux */
 i      = 0;
 while (!trouve && i<=n-1)  /* Identique a : while (trouve!=1 && i<=n-1) */
 {
  if (tab[i] == e) { trouve  = 1; /* vrai */ }

  i = i + 1;
 }

 return trouve;
}
```

Cet algorithme retourne : – `1` (vrai) si `e` est présent dans le tableau `tab`
– `0` (faux) sinon.

Exemples de calcul de coût

Recherche séquentielle d'un élément `e` dans un tableau `tab` de `n` éléments (suite) :

▷ Quel est le coût de cet algorithme?

▷ Si `e` est dans `tab` alors toutes les places dans `tab` sont possibles pour `e`

▷ La complexité en nombre de comparaisons de `e` avec un élément de `tab` :
◇ Meilleur cas : `1` ; pire cas : `n`
◇ En moyenne : $\frac{n+1}{2}$

▷ Ordre de grandeur de la complexité :
◇ Si on considère une machine 2 fois plus rapide alors le `/2` n'a pas d'importance $\Rightarrow$ `n+1`
◇ Si `n` est grand alors le `+1` n'a pas d'importance $\Rightarrow$ `n`

$\Longrightarrow O(n)$

Exemples de calcul de coût

Recherche dichotomique d'un élément `e` dans un tableau trié `tab` de `n` éléments :

Principe (voir TD-TP) :

- ▷ Le tableau doit être trié
- ▷ On compare l'élément cherché `e` par rapport à l'élément se trouvant au milieu `p` :
  - ◇ Si `e` = `p` alors on a trouvé !
  - ◇ Si `e` > `p` alors il faut chercher dans le sous–tableau à droite
  - ◇ Si `e` < `p` alors il faut chercher dans le sous–tableau à gauche
- ▷ On s'arrête si on a trouvé ou si l'espace de recherche est devenu vide.

Idée principale : à chaque étape, on divise l'espace de recherche par 2
$\Longrightarrow O(log_2(\mathtt{n}))$

Exemples de calcul de coût

Calcul du produit `m` de deux matrices `m1` et `m2` de taille `n`X`n` :

```
for(l=0; l<=n-1; l=l+1)
{
 for(c=0; c<=n-1; c=c+1)
 {
  m[l][c] = 0;
  for(i=0; i<=n-1; i=i+1)
  {
   m[l][c] = m[l][c] + m1[l][i]*m2[i][c];
  }
 }
}
```

- ▷ Quel est le coût de cet algorithme?
- ▷ Complexité en nombre de multiplications :
  - ◇ boucle interne : `n` multiplications
  - ◇ boucle du milieu : $\mathtt{n}^2$ multiplications
  - ◇ boucle externe : $\mathtt{n}^3$ multiplications $\Longrightarrow O(\mathtt{n}^3)$

Mesure de complexité et comparaison d'algorithmes

La notion de complexité permet de comparer des algorithmes

Exemple :
- ▷ Soit un algorithme A_1 de complexité en $O(\mathtt{n}^2)$
- ▷ Soit un algorithme A_2 de complexité en $O(\mathtt{2n})$

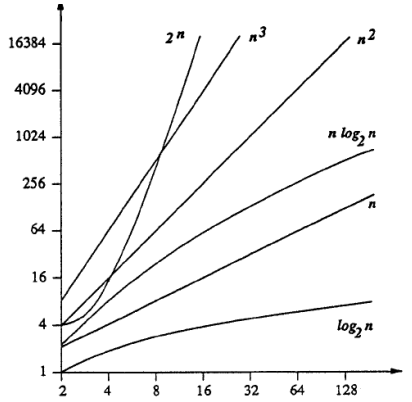
$\Longrightarrow$ A_2 est meilleur que A_1 dès que $\mathtt{n} > 2$

Pour comparer deux algorithmes, on cherchera l'ordre de grandeur asymptotique (limite quand `n` devient ∞) de ces algorithmes :

$$1 < log_2\,\mathtt{n} < \mathtt{n} < \mathtt{n}.log_2\,\mathtt{n} < \mathtt{n}^2 < \mathtt{n}^3 < ... < 2^{\mathtt{n}} < ...$$

Mesure de complexité et comparaison d'algorithmes

Histoire de comparer :



Mesure de complexité et comparaison d'algorithmes

Comment évolue le temps de calcul en fonction de la taille des données?

... en exécutant 10^6 opérations par seconde

$\mathtt{n}$ \ complexité	1	$log_2(\mathtt{n})$	$\mathtt{n}$	$\mathtt{n}.log_2(\mathtt{n})$	$\mathtt{n}^2$	$\mathtt{n}^3$	$2^{\mathtt{n}}$
10^2	$\approx 1\ \mu$s	6,64 μs	0,1 ms	0,66 ms	10 ms	1 s	4.10^{16} a
10^3	$\approx 1\ \mu$s	9,97 μs	1 ms	9,97 ms	1 s	16,66 m	∞
10^4	$\approx 1\ \mu$s	13,29 μs	10 ms	0,13 s	1,66 m	11,55 j	∞
10^5	$\approx 1\ \mu$s	16,61 μs	0,1 s	1,66 s	2,78 h	31,7 a	∞
10^6	$\approx 1\ \mu$s	19,93 μs	1 s	19,93 s	11,57 j	3.10^7 a	∞

Mesure de complexité et comparaison d'algorithmes

Quelle taille de données peut-on traiter en un temps de calcul donné?

... en exécutant toujours 10^6 opérations par seconde

temps \ complexité	1	$log_2(\mathtt{n})$	$\mathtt{n}$	$\mathtt{n}.log_2(\mathtt{n})$	$\mathtt{n}^2$	$\mathtt{n}^3$	$2^{\mathtt{n}}$
1 s	∞	∞	10^6	$6{,}3.10^4$	10^3	10^2	19
1 m	∞	∞	6.10^7	$2{,}8.10^6$	7.10^3	4.10^2	25
1 h	∞	∞	38.10^8	$1{,}3.10^8$	6.10^4	15.10^2	31
1 j	∞	∞	$8{,}6.10^{10}$	$2{,}7.10^9$	$2{,}9.10^5$	44.10^2	36

$\infty : > 10^{100}$

Algorithmes exacts vs Heuristiques

Algorithmes exacts

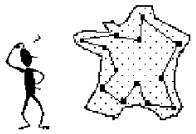
Les algorithmes résolvent les problèmes de manière exacte... mais parfois ils sont trop lents!

Exemple : le problème du voyageur de commerce

- ▷ Passer par une ville exactement 1 fois
- ▷ Revenir au départ
- ▷ Minimiser le chemin

$\Longrightarrow O(\mathtt{n}!)$

Et donc, si évaluation d'un chemin en $1\mu s$:



Nb villes	Nb possibilités	Temps de calcul
5	12	$12\mu s$
10	181440	0, 18 ms
15	43 milliards	12 heures
20	60.10^{15}	1928 ans
25	310.10^{21}	9, 8 milliards d'années

Heuristiques

Les heuristiques offrent une résolution rapide mais approchée.

Exemples : algos génétiques, recuit simulé, systèmes immunitaires artificiels,...

Algorithmes de tri

Définitions

Spécification d'un algorithme de tri

▷ **Entrée** : 1 tableau

▷ **Sortie** : 1 tableau contenant les mêmes éléments que le tableau d'entrée mais trié, c'est–à–dire : $\forall i, j \quad i < j \Longrightarrow t[i] \leq t[j]$ (ordre croissant)
$\forall i, j \quad i < j \Longrightarrow t[i] \geq t[j]$ (ordre décroissant)

▷ Remarques : – le même tableau peut servir en entrée et en sortie
– on considère ici le tri par ordre croissant

Plusieurs algorithmes de tri :

▷ Tri à bulle

▷ Tri par sélection

▷ Tri fusion

▷ Tri rapide

▷ ...

Une fonction bien utile : `echange`

Procédure d'échange de deux valeurs contenues dans un tableau

Beaucoup d'algorithmes de tri utilisent une procédure qui permet de permuter les valeurs contenues dans 2 cases d'un tableau.

```
void echange(int tab[], int size, int i, int j)
{
 int temp;

 if (!(i>=0 && i<=size-1)) { printf("Parametre i invalide\n"); exit(1); }
 if (!(j>=0 && j<=size-1)) { printf("Parametre j invalide\n"); exit(1); }

 temp   = tab[i];
 tab[i] = tab[j];
 tab[j] = temp;
}
```


Tri à bulle : définition

Principe

On fait "remonter" les éléments les plus grands vers la fin du tableau.
Pour les `n` premiers éléments (puis les `n-1`, ...), on exécute :

Pour chaque indice, en partant de 0 :

▷ On compare successivement un élément et l'élément qui suit dans le tableau.

▷ Si il est plus grand, on échange les valeurs, sinon on ne fait rien.

Exemple

tableau d'entrée : | 3 | 4 | 2 | 3 |

Première passe :
▷ 3 comparé à 4 (↮) | 3 | 4 | 2 | 3 |
▷ 4 comparé à 2 (↔) | 3 | 2 | 4 | 3 |
▷ 4 comparé à 3 (↔) | 3 | 2 | 3 | 4 |

Deuxième passe :
▷ 3 comparé à 2 (↔) | 2 | 3 | 3 | 4 |
▷ 3 comparé à 3 (↮) | 2 | 3 | 3 | 4 |

Troisième passe :
▷ 2 comparé à 3 (↮) | 2 | 3 | 3 | 4 |

Quatrième passe :

Tri à bulle : code

```
void triBulle(int tab[], int size)
{
 int i,j;
 int n = size;

 for(i=0; i<=n-1 ; i=i+1)
 {
  for(j=0; j<=n-1-i-1 ; j=j+1) /* n-1: comme d'habitude                    */
  {                            /* -i: car les i derniers sont deja tries    */
   if (tab[j]>tab[j+1])        /* -1: pour le j+1 (afin de ne pas deborder) */
   {
    echange(tab,size,j,j+1);
   }
   /* trace */
  }
 }
}
```

Tri à bulle : appel

```
#include <stdio.h>
#include <stdlib.h>  /* exit */
void printTab(int tab[], int size)
{
 /* le code de printTab */
}
void echange(int tab[], int size, int i, int j)
{
 /* le code de echange */
}
void triBulle(int tab[], int size)
{
 /* le code de triBulle */
}
int main(void)
{
 int tab[5]= {90,67,2,50,23};
 int taille = 5;
 printTab(tab,taille);      /* {90,67,2,50,23} */
 triBulle(tab,taille);
 printTab(tab,taille);      /* {2,23,50,67,90} */
 return 0;
}
```


Tri à bulle : traces

Traces du programme : avec le tableau {90, 67, 2, 50, 23}

i	j	tab	size
?	?	{90 67 2 50 23}	5
0	0	{67 90 2 50 23}	5
0	1	{67 2 90 50 23}	5
0	2	{67 2 50 90 23}	5
0	3	{67 2 50 23 90}	5
1	0	{ 2 67 50 23 90}	5
1	1	{ 2 50 67 23 90}	5
1	2	{ 2 50 23 67 90}	5
2	0	{ 2 50 23 67 90}	5
2	1	{ 2 23 50 67 90}	5
3	0	{ 2 23 50 67 90}	5
?	?	{ 2 23 50 67 90}	5

Tri à bulle : complexité

Nombre d'opérations nécessaires :

▷ Pour la boucle interne (`j`) : comparaison : 1 ;
échange éventuel : 3 affectations ;
gestion du `j` : 3 (addition, affectation, test)

$\Longrightarrow$ 7 opérations

▷ Pour la boucle externe (`i`) : gestion du `i` : 3 (addition, affectation, test) ;
la boucle interne (`j`)

$\Longrightarrow 3 * n + 7 * ((n-1) + (n-2) + ... + 1)$

$\Longrightarrow 3 * n + 7 * \frac{(n-1)*n}{2} = \frac{7}{2} n^2 - \frac{n}{2}$ opérations

Et donc :

Tri à bulle est en $O(n^2)$ ...

C'est un mauvais tri! En effet, il existe des tris en $0(n.log_2(n))$

Tri à bulle : amélioration

```
void triBulleRuse(int tab[], int size)
{
 int inversion = 1; /* vrai au depart */
 int i,j;
 int n = size;

 i=0;
 while (i<=n-1 && inversion) /* inversion permet de s'arreter si le tableau est deja trie */
 {
  inversion = 0;    /* faux */

  for(j=0; j<=n-1-i-1 ; j=j+1) /* n-1: comme d'habitude                    */
  {                            /* -i: car les i derniers sont deja tries   */
   if (tab[j]>tab[j+1])        /* -1: pour le j+1 (afin de ne pas deborder) */
   {
    echange(tab,size,j,j+1);
    inversion = 1;  /* vrai */
   }
  }
  i = i + 1;
 }
}
```

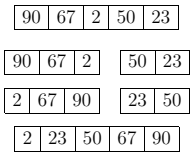
Complexité toujours en $O(n^2)$

Tri fusion : définition

Principe : "diviser pour régner"

Méthode :

- ▷ On divise le tableau en deux
- ▷ On trie les sous–tableaux
- ▷ On fusionne les sous–tableaux triés



Inconvénient du tri fusion :
Recopie dans des tableaux intermédiaires ...

Tri fusion : code (0)

Vérification si un tableau est trié ou non

```
/* Retourne 1 (vrai) si le tableau est trie       */
/* Retourne 0 (faux) si le tableau n'est pas trie */

int estTrie(int tab[], int size)
{
 int trie = 1; /* vrai par defaut */
 int i;

 i = 1;
 while (trie && i<=size-1)  /* Identique a : while (trie==1 && i<=size-1) */
 {
  if (tab[i-1]>tab[i])
  {
   trie = 0; /* faux */
  }
  i = i + 1;
 }

 return trie;
}
```


Tri fusion : code (0')

Copie dans `sousTab` d'un sous tableau de `tab` pour les indices allant de `inf` à `sup`...

```
void copieSousTab(int tab[], int sousTab[], int inf, int sup)
{
 int i,j;
 j=0;
 for(i=inf; i<=sup; i=i+1)
 {
  sousTab[j]=tab[i];
  j=j+1;
 }
}
```


Tri fusion : code (1)

Fusion de deux sous-tableaux triés

```
void fusion(int tab[], int g, int m, int d)
{
  int i;                        /* indice pour parcourir tab  */

  int nbElemsTab1 = m - g + 1;
  int tab1[nbElemsTab1];
  int i1;                       /* indice pour parcourir tab1 */

  int nbElemsTab2 = d - m;
  int tab2[nbElemsTab2];
  int i2;                       /* indice pour parcourir tab2 */

  copieSousTab(tab,tab1,g,m);   /* Copie, dans le tableau tab1, des elements */
                                /* de tab des indices g  m                   */

  copieSousTab(tab,tab2,m+1,d); /* Copie, dans le tableau tab2, des elements */
                                /* de tab des indices m+1  d                 */

  if (!estTrie(tab1,nbElemsTab1)) { printf("tab1 n'est pas trie\n"); exit(1); }
  if (!estTrie(tab2,nbElemsTab2)) { printf("tab2 n'est pas trie\n"); exit(1); }
```

Tri fusion : code (2)

Fusion de deux sous-tableaux triés (suite)

```
  i1=0; i2=0;
  for(i=g; i<=d; i=i+1)
  {
   if (i1<=nbElemsTab1-1 && i2<=nbElemsTab2-1)
   {
    if (tab1[i1]<tab2[i2]) { tab[i]=tab1[i1]; i1=i1+1; }
    else                   { tab[i]=tab2[i2]; i2=i2+1; }
   }
   else
   {   /* si l'un des tableaux est vide avant l'autre */
    if (i1<=nbElemsTab1-1)
    {
     tab[i]=tab1[i1]; i1=i1+1; /* tab2 est vide */
    }
    else
    {
     tab[i]=tab2[i2]; i2=i2+1; /* tab1 est vide */
    }
   }
  }
}
```


Tri fusion : code (3)

Division du tableau en 2, tri des 2 sous-tableaux, puis fusion

```
void triRecFusion(int tab[], int size, int depart, int arrivee)
{
 int n = size;
 if (depart<0   || depart >= size ||
     arrivee <0 || arrivee >= size)  { printf("Erreur depart ou arrivee\n");
                                       exit(1);
 }
 if (depart < arrivee)
 {
  int m = (depart+arrivee)/2;
  triRecFusion(tab,size,depart,m);
  triRecFusion(tab,size,m+1,arrivee);
  fusion(tab,depart,m,arrivee);
  /* trace */
 }
}
void triFusion(int tab[], int size)
{
 triRecFusion(tab,size,0,size-1);
}
```


Tri fusion : appel

```
#include <stdio.h>
#include <stdlib.h>  /* exit */

void printTab(int tab[], int size)
{ /* le code de printTab */ }
int estTrie(int tab[], int size)
{ /* le code de estTrie */ }
void copieSousTab(int tab[], int sousTab[], int inf, int sup)
{ /* le code de copieSousTab */ }
void fusion(int tab[], int g, int m, int d)
{ /* le code de fusion */ }
void triRecFusion(int tab[], int size, int depart, int arrivee)
{ /* le code de triRecFusion */ }
void triFusion(int tab[], int size)
{
 triRecFusion(tab,size,0,size-1);
}

int main(void)
{
 int tab[5]= {90,67,2,50,23};
 int taille = 5;
 printTab(tab,taille);      /* {90,67,2,50,23} */
 triFusion(tab,taille);
 printTab(tab,taille);      /* {2,23,50,67,90} */
 return 0;
}
```


Tri fusion : trace

Traces du programme : avec le tableau {90, 67, 2, 50, 23}

```
m   tab
-------------------------
?   {90  67   2  50  23}
-------------------------
0   {67  90   2  50  23}
1   { 2  67  90  50  23}
3   { 2  67  90  23  50}
2   { 2  23  50  67  90}
-------------------------
?   { 2  23  50  67  90}
-------------------------
```

Tri fusion : complexité

Pour simplifier,
on ne considère que les comparaisons entre éléments du tableau.

Evaluation de la complexité $C(\mathtt{n})$:

- ▷ Lorsque l'on exécute `fusion` sur le tableau complet de taille `n`, on exécute `(n - 1)` comparaisons.
- ▷ Si `n` est grand, on peut simplifier et considérer que l'on a `n` comparaisons.
- ▷ Comme on divise le tableau en 2 et que l'on recommence sur les sous tableaux : $C(\mathtt{n}) = \mathtt{n} + 2{*}C(\mathtt{n}/2)$

$$\Longrightarrow C(\mathtt{n}) = \mathtt{n} + \mathtt{n} * \mathtt{log_2(n)}$$ ... voir la démonstration à suivre

$$\Longrightarrow C(\mathtt{n}) = O(\mathtt{n} * \mathtt{log_2(n)})$$

Inconvénient du tri fusion :
Recopie dans des tableaux intermédiaires ...

Tri fusion : complexité

Démonstration :

Pour simplifier, on se place dans le cas où le tableau est de taille $\mathtt{n} = 2^p$.

$C(\mathtt{n}) = \mathtt{n} + 2{*}C(\mathtt{n}/2)$

$\Rightarrow C(2^p) = 2^p + 2{*}C(2^{p-1})$

$\Rightarrow C(2^p) = 2^p + 2^p{*}p$ ....A démontrer par récurrence (cf transparent suivant).

$\Rightarrow C(\mathtt{n}) = \mathtt{n} + \mathtt{n}{*}log_2(\mathtt{n})$ ... CQFD

Rappel : $\mathtt{n} = 2^p \Longleftrightarrow p = log_2(\mathtt{n})$

Tri fusion : complexité

Démontrons par récurrence que : $C(2^p) = 2^p + 2{*}C(2^{p-1})$
$\Rightarrow C(2^p) = 2^p + 2^p{*}p$

- ▷ **Initialisation :**

  $C(0) = 0$ (pas de donnée...)
  $C(2^0) = C(1) = 1 + 2{*}C(0) = 1 = 2^0 + 2^0{*}0$
  $C(2^1) = C(2) = 2 + 2{*}C(1) = 4 = 2^1 + 2^1{*}1$
  $C(2^2) = C(4) = 4 + 2{*}C(2) = 12 = 2^2 + 2^2{*}2$
  ...

- ▷ **Récurrence :**

$$\begin{aligned} C(2^{p+1}) &= 2^{p+1} + 2{*}C(2^p) \\ &= 2^{p+1} + 2{*}(2^p + 2^p{*}p) \\ &= 2^{p+1} + 2^{p+1} + 2^{p+1}{*}p \\ &= 2^{p+1} + 2^{p+1}{*}(1 + p) \\ &= 2^{p+1} + 2^{p+1}{*}(p + 1) \end{aligned}$$ ... CQFD

Tri rapide : définition

Un tri :

- ▷ sans tableau intermédiaire et
- ▷ de complexité moyenne en $O(\mathtt{n} * \mathtt{log_2(n)})$

Approche "diviser pour régner" :

- ▷ Diviser le tableau en deux sous–tableaux
- ▷ Trier chacun des sous–tableaux

Tri rapide : définition

Principe :

- ▷ On prend le premier élément du tableau, **T[0]**, appelé **pivot**.
- ▷ On met tous les éléments $>$**pivot** à droite
- ▷ On met tous les éléments $\leq$**pivot** à gauche
- ▷ On applique ce principe récursivement aux deux sous–tableaux obtenus

Remarque : pas besoin de fusionner les deux sous–tableaux triés...

File et pile

Notion de type abstrait
File (FIFO: First In First Out) ... *Queue* en anglais
Pile (LIFO: Last In First Out) ... *Stack* en anglais
Un pas de plus vers l'abstraction : la programmation modulaire

Notion de type abstrait

Objectif : description "formelle" d'un type de données.

Type abstrait :

- ▷ **Type :** t
- ▷ **Utilise :** le nom des types utilisés par t
- ▷ **Opérations :** opérations possibles sur un objet de type t
- ▷ **Préconditions :** préconditions sur ces opérations
- ▷ **Axiomes :** ... qui, pour un objet de type t, doivent être toujours vrais !

Remarque : description indépendante de toute implémentation, de tout langage de programmation.

Notion de type abstrait : exemple

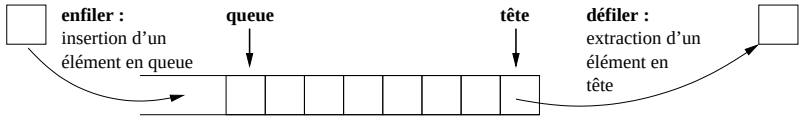
- ▷ **Type :** booléen
- ▷ **Utilise :** –
- ▷ **Opérations :**
  - ◇ vrai : $\longrightarrow$ booléen
  - ◇ faux : $\longrightarrow$ booléen
  - ◇ non ($\neg$) : booléen $\longrightarrow$ booléen
  - ◇ et ($\wedge$) : booléen X booléen $\longrightarrow$ booléen
  - ◇ ou ($\vee$) : booléen X booléen $\longrightarrow$ booléen
- ▷ **Préconditions :** –
- ▷ **Axiomes :**
  - ◇ $\neg$(vrai) = faux
  - ◇ $\neg$(faux) = vrai
  - ◇ a $\wedge$ vrai = a
  - ◇ a $\wedge$ faux = faux — Rappel : $\wedge$ est commutatif
  - ◇ a $\vee$ vrai = vrai
  - ◇ a $\vee$ faux = a — Rappel : $\vee$ est commutatif

File: introduction et intérêt

Introduction :

▷ Les files sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur.

▷ Dans une file les éléments sont systématiquement: **- ajoutés en queue** et **- extraits en tête**



Une file (First In Fisrt Out: FIFO)

Intérêt : les files sont des structures de données très utilisées car elles permettent des gérer les accès à des ressources informatiques (imprimante, processeur, ...).

Dans ce cours on se place dans le cas de files d'entiers, entiers qui peuvent représenter un numéro de ticket...

File : type abstrait (1)

▷ **Type :** file d'objets de type élément

▷ **Utilise :** booléen, élément

▷ **Opérations :** En anglais...

- ◇ créer : → file ... create
- ◇ estVide : file → booléen ... isEmpty
- ◇ estPleine : file → booléen ... isFull
- ◇ premier : file → élément ... first
- ◇ enfiler : file X élément → file ... enqueue
- ◇ défiler : file → file ... dequeue

▷ **Préconditions :**

- ◇ premier(f) ssi estVide(f) = faux
- ◇ enfiler(f,e) ssi estPleine(f) = faux
- ◇ défiler(f) ssi estVide(f) = faux

Avec f (une file) et e (un élément).

File : type abstrait (2)

▷ **Axiomes :**

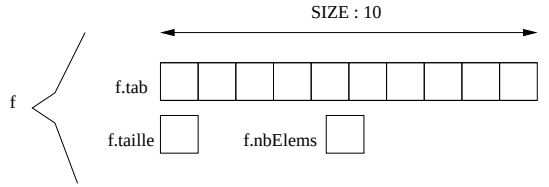
- ◇ estVide(créer()) = vrai
- ◇ estVide(enfiler(f,e)) = faux
- ◇ estVide(f) = vrai ⇒ premier(enfiler(f,e)) = e
- ◇ estVide(f) = faux ⇒ premier(enfiler(f,e)) = premier(f)
- ◇ estVide(f) = vrai ⇒ estVide(défiler(enfiler(f,e))) = vrai
- ◇ estVide(f) = faux ⇒ défiler(enfiler(f,e)) = enfiler(défiler(f) ,e)

Avec f (une file) et e (un élément).

File : implémentation C naïve (1)

```
#define SIZE 10

struct stfile { int tab[SIZE];
                int taille;    /* taille du tableau tab                    */
                int nbElems;   /* nombre de valeurs stockees dans le tableau */
              };

typedef struct stfile file;
```

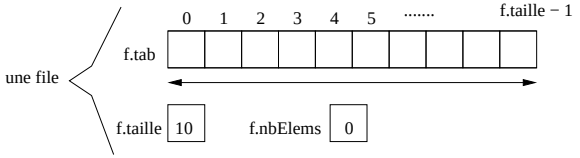
Si f est une `file` :



File : implémentation C naïve (2)

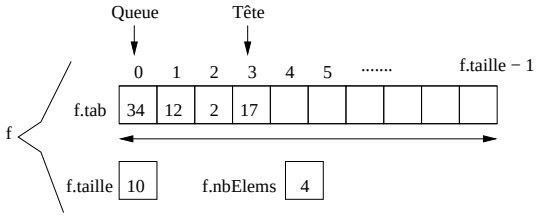
creerFile :

```
file creerFile(void)
{
 file f;
 f.taille  = SIZE;
 f.nbElems = 0;
 return f;
}
```



afficherFile :

```
void afficherFile(file f)
{
 int i;
 printf("[TETE] ");
 for(i=f.nbElems-1 ; i>=0; i=i-1)
 {
  printf("%d ",f.tab[i]);
 }
 printf("[QUEUE]");
}
```



File : implémentation C naïve (3)

estVideFile :

```
int estVideFile(file f)
{
 int returnValue = 0;
 if (f.nbElems == 0) returnValue=1;
 return returnValue;
}
```

estPleineFile

```
int estPleineFile(file f)
{
 int returnValue = 0;
 if (f.nbElems == f.taille) returnValue=1;
 return returnValue;
}
```

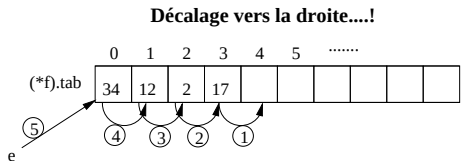
premierFile

```
int premierFile(file f)
{
 if (estVideFile(f)) { printf("File vide!!!...\n"); exit(1); }
 return f.tab[f.nbElems-1];
}
```


File : implémentation C naïve (4)

enfilerFile :

```
void enfilerFile(file* f, int e)
{
 int i;

 if (estPleineFile(*f)) { printf("File pleine!!!...\n"); exit(1); }

 for(i=(*f).nbElems-1 ; i>=0; i=i-1)
 {
  (*f).tab[i+1] = (*f).tab[i];
 }

 (*f).tab[0]  = e;
 (*f).nbElems = (*f).nbElems + 1;
}
```



File : implémentation C naïve (5)

defilerFile :

```
void defilerFile(file* f)
{
 if (estVideFile(*f)) { printf("File vide!!!...\n"); exit(1); }

 (*f).nbElems = (*f).nbElems - 1;
}
```

File : implémentation C naïve (6)

un menu pour tester les fonctions :

```
int menu (void)
{
 int rep, minRep = 0 , maxRep = 4;

 do
 {
   printf("\n");
   printf("0. Sortie\n");
   printf("1. Afficher la file\n");
   printf("2. Afficher la tete\n");
   printf("3. Enfiler un element\n");
   printf("4. Defiler un element\n");

   printf("Votre choix:");  scanf("%d",&rep);

   if (rep < minRep || rep > maxRep) { printf("Erreur de saisie\n"); }

  } while (rep < minRep || rep > maxRep);

 return rep;
}
```


File : implémentation C naïve (7)

un menu pour tester les fonctions (suite) :

```
int main(void)
{
 file f = creerFile();
 int rep;

 rep = menu();

 while (rep!=0)
 {                                /* On peut aussi faire des if..else */
  switch (rep)
  {
    case 1: { afficherFile(f);
              printf("\n");
              break;
            }
    case 2: { if (estVideFile(f))
              {
               printf("Attention, file vide!\n");
              }
              else { int premier = premierFile(f);
                     printf("La tete de la file est : %d\n",premier);
                   }
              break;
            }
                                                        /* main ... a suivre */
```


File : implémentation C naïve (8)

un menu pour tester les fonctions (suite) :

```
    case 3: { if (estPleineFile(f))                     /* main ... la suite */
              {
               printf("Attention, file pleine!\n");
              }
              else { int elem;
                     printf("Element a enfiler?\n");
                     scanf("%d",&elem);
                     enfilerFile(&f,elem);
                   }
              break;
            }
   default: { if (estVideFile(f))
              {
               printf("Attention, file vide!\n");
              }
              else { defilerFile(&f); }
              break;
            }
  }

  rep = menu();
 }

 printf("Sortie du programme\n");
 return 0;
}
```


File : implémentation C, naïve (9) ⇒ circulaire (1)

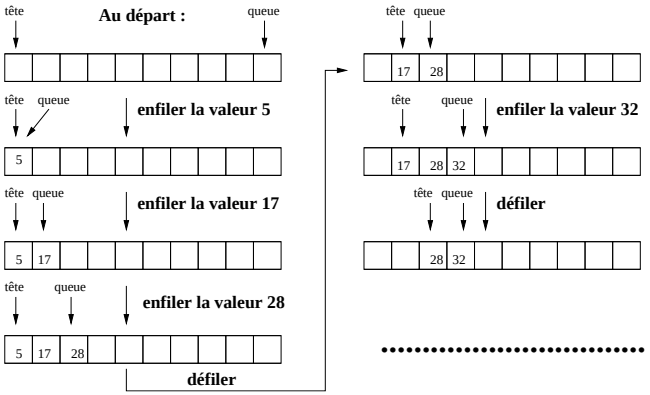
Attention : cette implémentation est un peu naïve !

- ▷ **Avantage :** très simple à mettre en œuvre
- ▷ **Inconvénient :** le décalage lors de `enfilerFile`

Solution : implémentation d'une file circulaire

```
#define SIZE 10

struct stfile { int tab[SIZE];
                int taille;    /* taille du tableau tab...                              */
                int nbElems;   /* pratique mais pas obligatoire : nb valeurs stockees... */
                int tete;      /* indice element de tete                                */
                int queue;     /* indice element de queue                               */
              };

typedef struct stfile file;
```

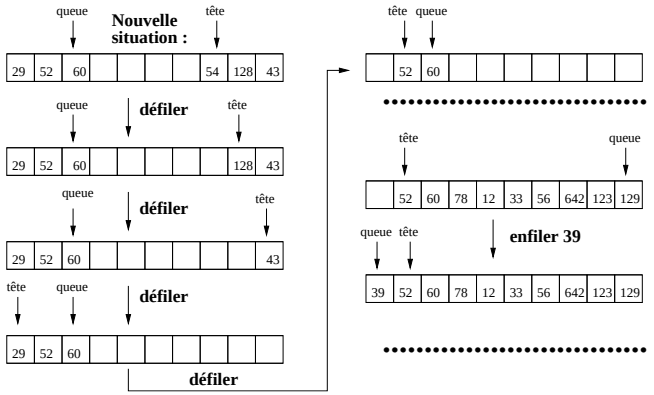
File : implémentation C circulaire (2)



`enfilerFile` $\Longrightarrow$ Evolution de la queue : `queue = (queue + 1) % taille` et, on met l'élément en position `queue`

`defilerFile` $\Longrightarrow$ Evolution de la tête : `tete = (tete + 1) % taille`

File : implémentation C circulaire (3)



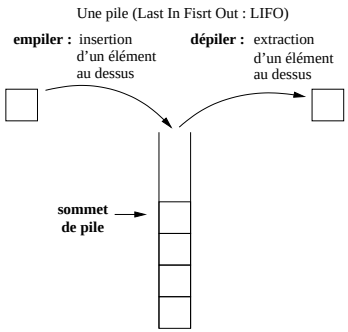
`enfilerFile` $\Longrightarrow$ Evolution de la queue : `queue = (queue + 1) % taille` et, on met l'élément en position `queue`

`defilerFile` $\Longrightarrow$ Evolution de la tête : `tete = (tete + 1) % taille`

Pile: introduction et intérêt

Introduction :

- ▷ Les piles sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur, traitement dans l'**ordre inverse de l'ordre d'arrivée**.
- ▷ Dans une pile les éléments sont systématiquement : **ajoutés au dessus** et **extraits au dessus**



Intérêt : les piles sont des structures de données très utilisées en informatique car elle permettent, notamment, d'implémenter les appels de fonctions.

Dans ce cours on se place dans le cas de piles d'entiers ...

Pile : type abstrait (1)

- ▷ **Type :** pile d'objets de type élément
- ▷ **Utilise :** booléen, élément
- ▷ **Opérations :** En anglais...
  - ◇ créer : $\rightarrow$ pile ... create
  - ◇ estVide : pile $\rightarrow$ booléen ... isEmpty
  - ◇ estPleine : pile $\rightarrow$ booléen ... isFull
  - ◇ sommet : pile $\rightarrow$ élément ... top
  - ◇ empiler : pile X élément $\rightarrow$ pile ... push
  - ◇ dépiler : pile $\rightarrow$ pile ... pop
- ▷ **Préconditions :**
  - ◇ sommet(p) ssi estVide(p) = faux
  - ◇ empiler(p,e) ssi estPleine(p) = faux
  - ◇ dépiler(p) ssi estVide(p) = faux

Avec p (une pile) et e (un élément).

Pile : type abstrait (2)

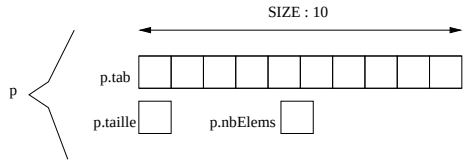
▷ **Axiomes :**
- ◇ estVide(créer()) = vrai
- ◇ estVide(empiler(p,e)) = faux
- ◇ sommet(empiler(p,e)) = e
- ◇ dépiler(empiler(p,e)) = p

Avec p (une pile) et e (un élément).

Pile : implémentation C (1)

```
#define SIZE 10

struct stpile { int tab[SIZE];
                int taille;    /* taille du tableau tab      */
                int nbElems;   /* nombre de valeurs stockees */
              };

typedef struct stpile pile;
```

Si p est une `pile` :



Pile : implémentation C (2)

Il reste à faire les fonctions :

```
▷ pile creerPile(void);
▷ int  estVidePile(pile p);
▷ int  estPleinePile(pile p);
▷ int  sommetPile(pile p);
▷ void empilerPile(pile* p, int e);
▷ void depilerPile(pile* p);
```

... et aussi `void afficherPile(pile p);`

Un pas de plus vers l'abstraction : la programmation modulaire

Un programmeur utilise un ensemble de fonctions sur un type donné.

Question :

Peut-on lui cacher la façon dont ce type et les fonctions associées sont implémentés ?

Réponse :

Oui, à travers la **programmation modulaire** et les **modules** !

Intérêts :
- ◇ Encapsulation des données et des traitements (encapsuler : cacher, enfermer,...).
- ◇ Réutilisation de modules existants (sans les ré–écrire !)

Un pas de plus vers l'abstraction : la programmation modulaire en C (1)

Exemple en C :

▷ On ne montre au programmeur que le fichier d'interface.
Ainsi, dans le cas des piles,
le programmeur a accès à un fichier : `modulePile.h`
`modulePile.h` ≃ type + en–tête des fonctions

▷ On écrit le fichier d'implémentation séparément du fichier d'interface...
Le programmeur n'a pas à connaître le contenu de ce fichier d'implémentation.
Ainsi, dans le cas des piles,
le code des fonctions peut être écrit dans un fichier : `modulePile.c`
`modulePile.c` ≃ code des fonctions

Un pas de plus vers l'abstraction : la programmation modulaire en C (2)

Ainsi, dans le fichier d'interface `modulePile.h` :

```
#ifndef _PILE_H_                                    /* modulePile.h */
#define _PILE_H_
#define SIZE 10

struct stpile { int tab[SIZE];
                int taille;    /* taille du tableau tab      */
                int nbElems;   /* nombre de valeurs stockees */
              };

typedef struct stpile pile;

pile creerPile(void);
void afficherPile(pile p);
int  estVidePile(pile p);
int  estPleinePile(pile p);
int  sommetPile(pile p);
void empilerPile(pile* p, int e);
void depilerPile(pile* p);
#endif
```


Un pas de plus vers l'abstraction : la programmation modulaire en C (3)

Ainsi, dans le fichier d'implémentation `modulePile.c` :

```
#include <stdio.h>                          /* modulePile.c */
#include <stdlib.h>   /* Pour exit */

#include "modulePile.h"

/*************************************************************************/

pile creerPile(void)
{
 pile p;

 p.taille  = SIZE;
 p.nbElems = 0;

 return p;
}

/*************************************************************************/

/* ... Et toutes les autres fonctions ... */
```


Un pas de plus vers l'abstraction : la programmation modulaire en C (4)

Compilation du module de pile : `cc -c modulePile.c`

$\Longrightarrow$ `modulePile.o`

... code objet non exécutable

Un pas de plus vers l'abstraction : la programmation modulaire en C (5)

Exemple d'utilisation du module de pile :

```
#include <stdio.h>                  /* useModulePile.c */

#include "modulePile.h"

int main(void)
{
 pile p = creerPile();
 int  n;

 printf("Une valeur a empiler ? ");
 scanf("%d",&n);

 if (!estPleinePile(p)) { empilerPile(&p,n); }

 afficherPile(p);

 /* .... */

 return 0;
}
```

Pour obtenir un exécutable :

```
cc modulePile.o useModulePile.c -o useModulePile
```