Algorithmique II

Le retour

FLORIAN LETOMBE
letombe@cril.univ-artois.fr
Bureau 105F

Le Cours

- Unité d'enseignement 2.2
- Module 2.23 : Outils et méthodes informatiques pour le multimédia
- Volume horaire : 30 h (6 h Cours, 12 h TD, 12 h TP)
- Objectifs :
  - aborder des techniques algorithmiques avancées
  - appréhender les notions de complexité et de structures de données
  - mettre en œuvre des algorithmes
  - apprentissage d'un langage de programmation (C++)
- Pré-requis : ce qu'on a vu au premier semestre !!!
- Contenu :
  - notions de complexité
  - récursivité (fonctions et procédures)
  - structures de données (tableaux, listes, files, piles, etc)
  - algorithmes associés (parcours, mise à jour, tri, ...)

Précisions

- Intervenant Cours & TDs :
  - F. Letombe
- Intervenants TPs :
  - J. Hondermarck
  - F. Letombe

- Modalités de contrôle de connaissances :
  - Théorique :

$$max(\frac{DS + Examen}{2}; Examen)$$

  - Pratique : 2 Examens TP
- Calcul de la moyenne :

$$Moyenne = \frac{2}{3}Théorique + \frac{1}{3}Pratique$$

- Rappels
  - Votre dernier examen
  - Types, variables
  - Structure conditionnelle/itérative
- Procédures et fonctions
  - Notion de bloc
  - Fonctions mathématiques
  - Déclaration d'une fonction
  - Appel d'une fonction
  - Passage de paramètres
  - Variables locales/globales et effets de bord
  - Récursivité
- Structures de données
  - Déclaration de types
  - Tableaux, listes, files, piles, fichiers
- Algorithmes de tri évolués
  - Le tri à bulle
  - Le tri par création
  - Le tri par sélection
  - Le tri par insertion
  - Le tri fusion

Citation

« D'une joie même, le souvenir a son amertume, et le rappel d'un plaisir n'est jamais sans douleur. »

Oscar Wilde

Extrait de **Le portrait de Dorian Gray**

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les raisons : vous vous foutez de tout, le monde vous emmerde, de toutes façons, vous avez eu une bonne note au DS, etc.

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les causes : vous vous foutez de tout, le coefficient est insignifiant, de toutes façons, vous aviez eu une bonne note au DS, etc
   - Les raisons : vous n'avez rien compris à la vie mes enfants !!!

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les causes : vous vous foutez de tout, le coefficient est insignifiant, de toutes façons, vous aviez eu une bonne note au DS, etc
   - Les raisons : vous n'avez rien compris à la vie mes enfants !!!
   - Les conséquences : pas d'avenir pour vous à court terme, pas de perspectives pour vos successeurs à long terme

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les causes : vous vous foutez de tout, le coefficient est insignifiant, de toutes façons, vous aviez eu une bonne note au DS, etc
   - Les raisons : vous n'avez rien compris à la vie mes enfants !!!
   - Les conséquences : pas d'avenir pour vous à court terme, pas de perspectives pour vos successeurs à long terme

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les causes : vous vous foutez de tout, le coefficient est insignifiant, de toutes façons, vous aviez eu une bonne note au DS, etc
   - Les raisons : vous n'avez rien compris à la vie mes enfants !!!
   - Les conséquences : pas d'avenir pour vous à court terme, pas de perspectives pour vos successeurs à long terme

Deux possibilités

1. Vous n'avez rien compris mais avez tout mis en œuvre pour pallier vos difficultés
   - Je vous fais peur ?!
   - Je suis injoignable ?
   - Vous êtes hermétiques à l'algorithmique
   - Vous n'avez pas lu la deuxième partie de la possibilité
2. Vous vous moquez complètement de cette matière
   - Les causes : vous vous foutez de tout, le coefficient est insignifiant, de toutes façons, vous aviez eu une bonne note au DS, etc
   - Les raisons : vous n'avez rien compris à la vie mes enfants !!!
   - Les conséquences : pas d'avenir pour vous à court terme, pas de perspectives pour vos successeurs à long terme

Changements

- Devoirs toutes les (2) semaines
- Ramassage éventuel de ces devoirs
- Interrogations orales notées (éventuellement sur ces devoirs)
- Attention : vous pouvez vous entraider, pas faire le travail pour un autre
- Cours disponibles sur Internet

`http://www.cril.univ-artois.fr/~letombe/enseignement.html`

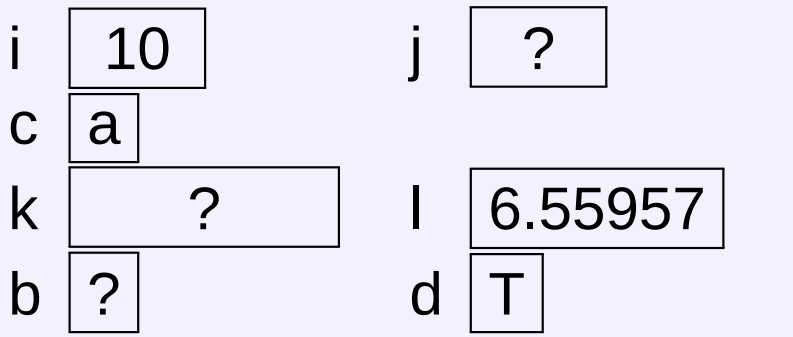
Palindromes

- En français :
  - À l'autel elle alla, elle le tua là.
  - Elu par cette crapule.
  - Esope reste ici et se repose.
  - Ta belle porte s'use trop, elle bat.
  - ...
  - Le palindrome de Saint-Gilles composé de 2119 mots a été construit par Pol Kools en 2004
- En anglais :
  - Madam, in Eden I'm Adam. *(Madame, dans l'Eden je suis Adam.)*
  - A man, a plan, a canal: Panama. *(Un homme, un projet, un canal: Panama.)*
- Même en mathématiques :
  - 1234+8765=9999=5678+4321

Déclaration

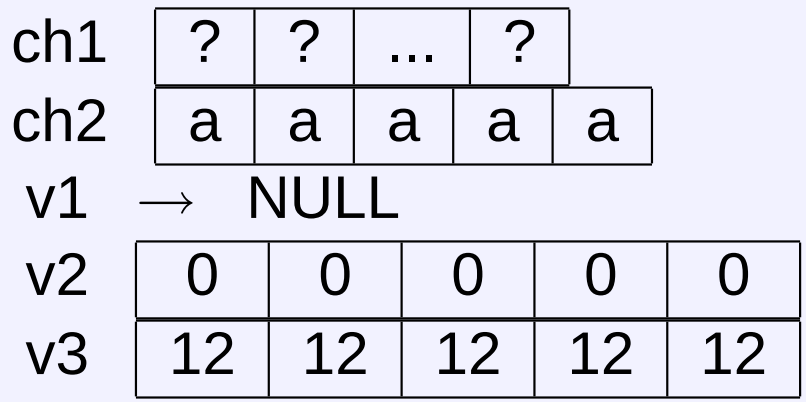
- Types simples :

```
int i = 10, j;
char c = 'a';
double k, l = 6.55957;
bool b, d = true;
```



- Types complexes :

```
string ch1,
       ch2(5, 'a');
vector<int> v1,
            v2(5),
            v3(5, 12);
```



Utilisation

```
string ch(5, 'a');
vector<int> v1(5),
            v2(5, 12);
int i = 10, j;
char c = 'a';
```

ch	a	a	a	a	a	
v1	0	0	0	0	0	
v2	12	12	12	12	12	
i	10			j	?	
c	a					

```
c = 'x';
ch[1] = c;
j = i++;
v1[0] = i;
v1[1] = j - 1;
v2[4] = v1[0];
```

c	x				
ch	a	x	a	a	a
i	11			j	10
v1	11	0	0	0	0
v1	11	9	0	0	0
v2	12	12	12	12	11

Conditionnelle

if (<expression logique>)
    <séquence d'instructions 1>
else <séquence d'instructions 2>

switch (<expression>) {
    case <expr const 1> : <séquence d'instructions 1>
    case <expr const 2> : <séquence d'instructions 2>
    ...
    case <expr const n> : <séquence d'instructions n>
    default : <séquence d'instructions n+1>
}

Itérative

while (<expression logique>)
    <séquence d'instructions>

for (<expression 1> ; <expr logique> ; <expression 2>)
    <séquence d'instructions>

do
        <séquence d'instructions>
while(<expression logique>)

Sur un exemple ...

```
#include <iostream>
#include <vector>

int main () {
    const int MAX = 10;
    vector<int> suite(MAX);
    int elem, i = 0;

    cout << "Suite de nombres positifs, terminée par 0 ?" << endl;
    do {
        cin >> elem;
        if (elem >= 0)
            suite[i++] = elem;
        else cout << "On a dit des nombres positifs" << endl;
    } while (i<MAX && (!i || suite[i-1]));
}
```

Citation

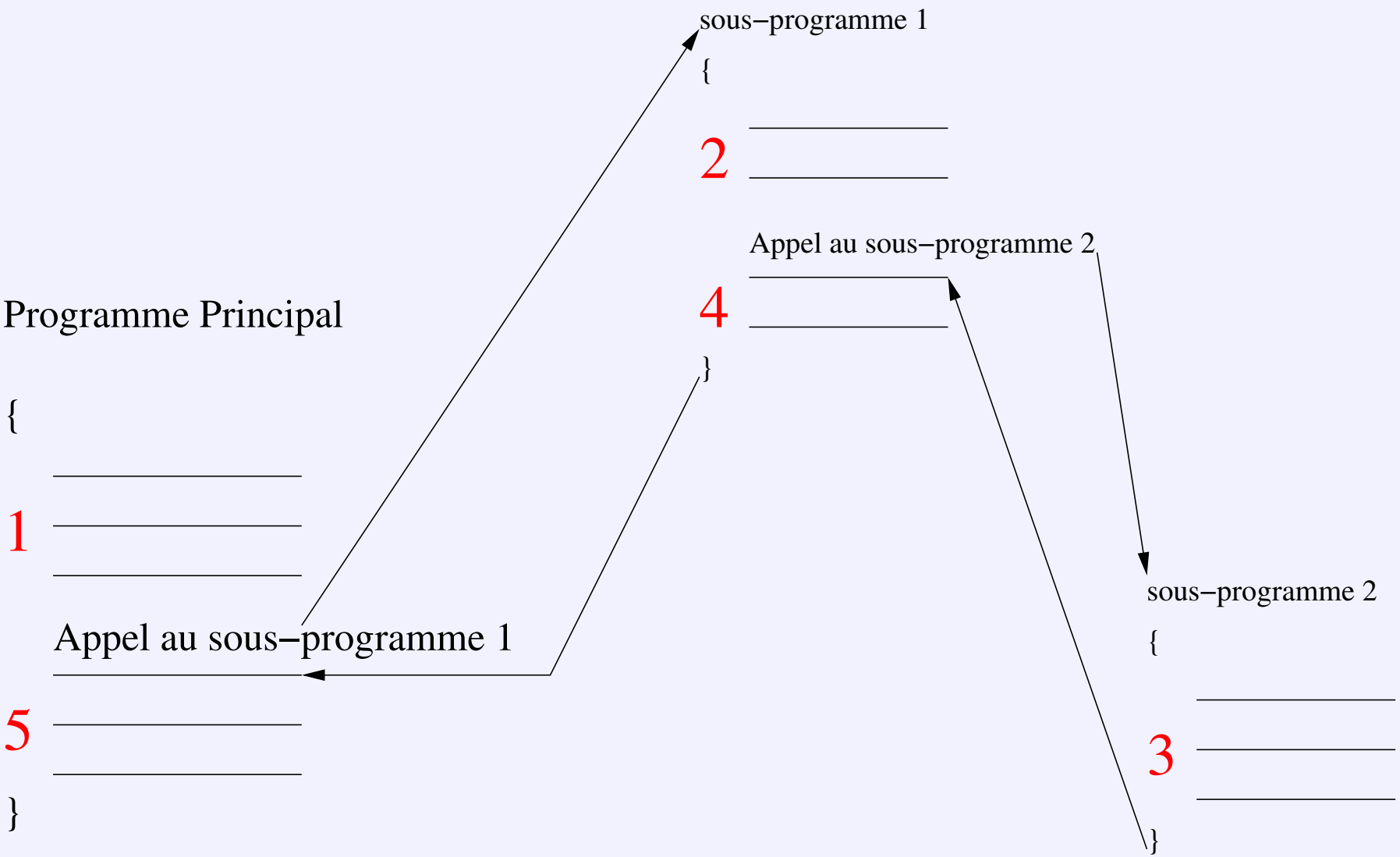
« *Procédure. Machine dans laquelle vous entrez tel un cochon et dont vous ressortez comme une saucisse.* »

Ambrose Bierce

Extrait de **Le dictionnaire du Diable**

- Lorsqu'une séquence d'instructions se répète plusieurs fois, il est intéressant de faire un sous-programme correspondant à ce bloc d'instructions et de l'utiliser autant de fois que nécessaire
- Cette séquence d'instructions sera définie dans un sous-programme qui peut prendre la forme d'une procédure ou d'une fonction

- De plus, un programme est presque toujours décomposable en modules qui peuvent alors être définis de manière indépendante. Cela permet de modifier éventuellement un module sans pour autant changer le corps du programme et de rendre le programme plus compréhensible (lisibilité)
- Un programme est alors un ensemble de procédures / fonctions


sous−programme 1
{
2
Appel au sous−programme 2
4
}
Programme Principal
{
1
Appel au sous−programme 1
5
}
sous−programme 2
{
3
}

Bases

- Les fonctions sont des outils
- Elles ont de nombreuses propriétés
- Règles scrupuleuses :
  - Avoir un ensemble de départ contenant l'ensemble de définition de la fonction et un ensemble d'arrivée
  - À chaque élément de cet ensemble de définition faire correspondre un de ceux de l'ensemble d'arrivée

Exemple

Transformer un nombre réel en un autre

- Ensemble de départ : $\mathbb{R}$
- Domaine de définition : tous les réels différents de 2 et -2
- Ensemble d'arrivée : $\mathbb{R}$
- Correspondance : à x, on associe $\frac{1}{x^2-4}$

Image/Antécédent

Si à un élément a, on fait correspondre un élément b,

- l'élement b est appelé l'image de a
- l'élément a est appelé un antécédent de b

Exemple

$f(x) = \frac{1}{x^2-4}$

- l'image de 3 est $\frac{1}{5}$
- les antécédents de $\frac{1}{5}$ sont 3 et -3

Notation

- Les fonction souvent utilisées finissent par porter des noms spécifiques (sin, cos ...)
- Les autres s'appellent f, g, etc
- L'image d'un élément a pour la fonction f est alors noté f(a) (ou g(a), etc)
- Pour résumer toutes ces informations, on utilise l'écriture suivante (spécification)

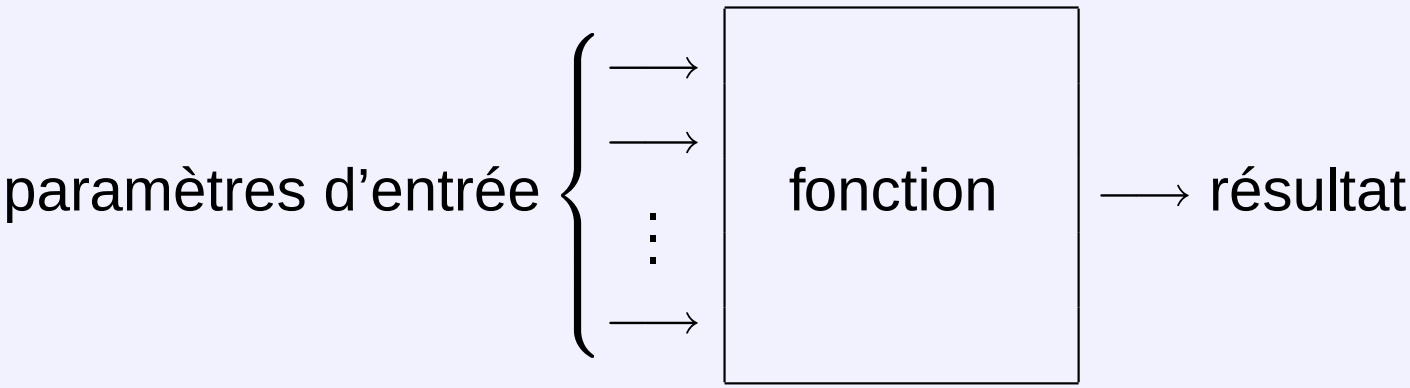
$$\begin{array}{rccl} f: & D & \rightarrow & A \\ & x & \mapsto & y \end{array}$$

avec $y = f(x)$ où D représente l'ensemble de départ et A l'ensemble d'arrivée

- La recherche du domaine de définition, si celui-ci est plus petit que l'ensemble de départ, reste à faire

Lien avec l'informatique

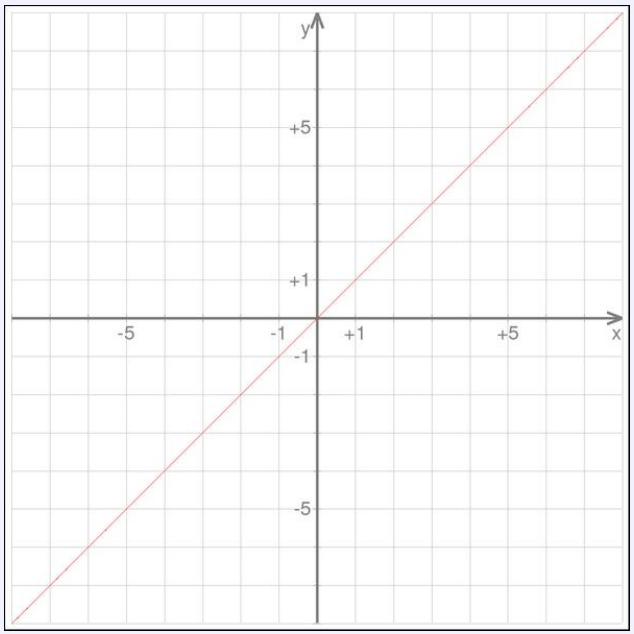
- Une fonction est un sous-programme qui retourne une valeur calculée en fonction des valeurs passées en entrée



- Une fonction prend zéro ou plusieurs paramètres et renvoie éventuellement un résultat
- Une fonction qui ne retourne pas de résultat est une procédure

Un exemple : la fonction identité

- L'identité ou application identique d'un ensemble est l'application de cet ensemble dans lui-même qui à chaque élément associe cet élément et lui seul
- Son graphe est donc la diagonale suivante :



- La fonction identité s'écrit $f(x) = x$

Syntaxe

<type de retour> identificateur(<liste de paramètres>) {
    <corps de la fonction>
}

où

- **type de retour** est le type du résultat renvoyé par la fonction (on utilise **void** lorsqu'il s'agit d'une procédure)
- **identificateur** est le nom que l'on a donné à la fonction
- **liste de paramètres** est la liste des paramètres formels donnés en entrée avec leurs types de la forme $type_1 p_1, \ldots, type_n p_n$, avec $p_i, (1 \leq i \leq n)$ un paramètre de type $type_i$
- **corps de la fonction** est un bloc d'instructions, pouvant comprendre la déclaration de « variables locales »

Précisions

- La valeur retournée par une fonction est d'un type donné donc il faut indiquer le type de cette valeur
- Dans la fonction, on a une suite d'instructions qui constitue le corps de la fonction
- Si le type de retour n'est pas *void*, le corps de la fonction doit comporter une instruction de la forme :

  return(*expression*);

  où *expresssion* est du type du résultat de la fonction
- Une telle expression met fin à l'exécution de la fonction et retourne *expression* comme résultat
- Il est possible d'utiliser l'instruction *return*; dans une procédure pour mettre fin à son exécution
- On utilise les valeurs données en entrée à l'intérieur du corps de la fonction grâce aux paramètres formels les identifiant

Exemple : $f(x) = \frac{1}{x^2-4}$

- Spécification :

$$\begin{array}{rccc} f\_x2\_4 : & \mathbb{R} & \longrightarrow & \mathbb{R} \\ & x & \longmapsto & \frac{1}{x^2-4} \end{array}$$

- Code C++ :

```
double f_x2_4 (double x) {
    double resultat;
    if ((x != 2) && (x != -2))
        resultat = 1/(x*x-4);
    else resultat = 0;
    return resultat;
}
```

Syntaxe

identificateur(<liste de paramètres>);

- L'identificateur de fonction doit être le nom de la fonction
- **liste de paramètres** est la liste des paramètres effectifs de la forme $q_1, \ldots, q_n$, avec $q_i, (1 \leq i \leq n)$ un paramètre de type $type_i$
- L'appel de la fonction retourne une valeur calculée en fonction des paramètres effectifs
- La liste des paramètres effectifs doit être compatible avec la liste des paramètres formels de la déclaration de la fonction
- Lors de l'appel, chacun des paramètres formels est remplacé par le paramètre effectif

Exemple : $f(x) = \frac{1}{x^2-4}$

- Soit la fonction déclarée ainsi :

```
double f_x2_4 (double x) {
    double resultat;
    if ((x != 2) && (x != -2))
        resultat = 1/(x*x-4);
    else resultat = 0;
    return resultat;
}
```

- L'appel à f_x2_4 pour un x=3 se fait comme suit :

```
f_x2_4(3);
```

Passage des paramètres par valeur

- C'est le cas standard
- Les paramètres sont initialisés par une copie des valeurs des paramètres effectifs
- Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres effectifs

Exemple

L'exécution du bout de programme (à gauche) avec la définition de la procédure *proc* (à droite)

```
int k=10;
proc(k);
cout << "k vaut "<< k << endl;
```

```
void proc(int a) {
    a++;
    cout << "a vaut "<< a << endl;
}
```

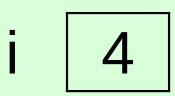
entrainera l'affichage de « a vaut 11 », puis de « k vaut 10 »

Notion de référence

- Il est parfois nécessaire d'autoriser une fonction à modifier la valeur d'un paramètre effectif
- Pour cela, il est possible d'utiliser des *références*
- La notion de référence n'est pas limité au passage de paramètres
- Une référence est un synonyme d'une variable, une autre manière de désigner le même emplacement mémoire
- On utilise le symbole & pour la déclaration d'une référence

Exemple

int i = 4;

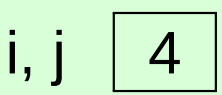


Notion de référence

- Il est parfois nécessaire d'autoriser une fonction à modifier la valeur d'un paramètre effectif
- Pour cela, il est possible d'utiliser des *références*
- La notion de référence n'est pas limité au passage de paramètres
- Une référence est un synonyme d'une variable, une autre manière de désigner le même emplacement mémoire
- On utilise le symbole & pour la déclaration d'une référence

Exemple

int i = 4;



int & j = i;

Passage des paramètres par référence

- Dans la liste de paramètres formels d'une définition de fonction, $type_i$ & p_i déclare le paramètre p_i comme une référence sur le $i^{\text{ème}}$ paramètre effectif q_i
- p_i est donc un synonyme de q_i
- Modifier p_i revient à modifier q_i

Exemple

L'exécution du bout de programme (à gauche) avec la définition de la procédure *proc* (à droite)

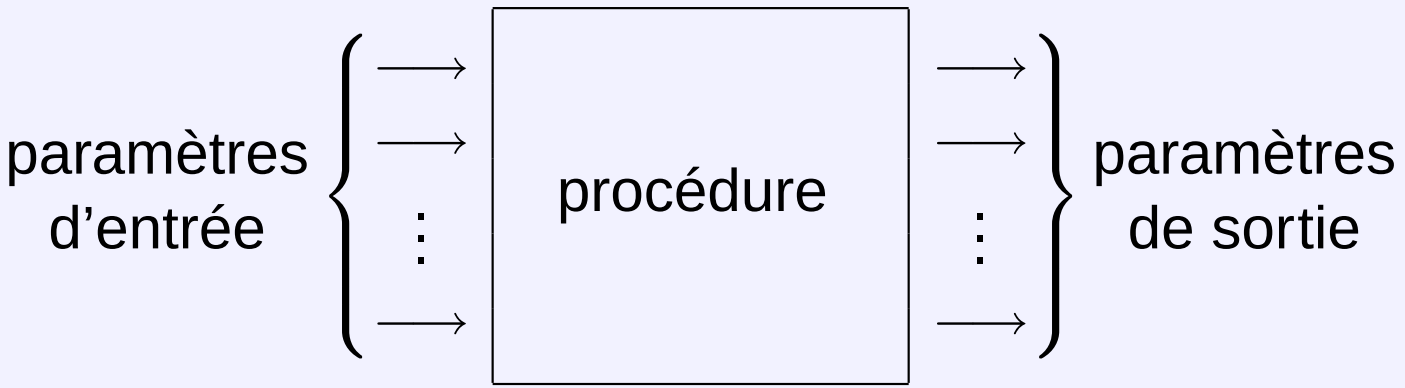
```
int k=10;
proc(k);
cout << "k vaut "<< k << endl;
```

```
void proc(int & a) {
    a++;
    cout << "a vaut "<< a << endl;
}
```

entrainera l'affichage de « a vaut 11 », puis de « k vaut 11 »

Précisions

- Une procédure pour laquelle le passage de paramètres par référence est employé peut être schématisée comme suit :



- Un passage par référence est plus efficace (surtout pour les types complexes tels des tableaux): il évite la recopie des arguments (seule l'adresse mémoire est transmise)
- Risque : modifier les données
- On utilise parfois le passage de paramètres en tant que référence sur une constante (**const** $type_i$ & p_i)

Valeur par défaut

- On peut attribuer une valeur par défaut aux paramètres d'une fonction

Exemple

La fonction *max* qui retourne le maximum entre deux nombres peut se définir ainsi :

```
int max(int a, int b=0) {
    return (a<b) ? a : b;
}
```

On peut alors appeler *max(i)* au lieu de *max(i, 0)*

- Seule restriction : si un paramètre a une valeur par défaut, tous les paramètres suivants en ont une aussi

Portée d'un identificateur

- La *portée* d'un identificateur (nom de variable, de fonction, ...) correspond aux parties du programme où cet identificateur peut être utilisé sans provoquer d'erreur à la compilation
- La portée d'une variable globale ou d'une fonction globale est égale au programme : elles peuvent être utilisées n'importe où
- Un identificateur correspondant à une variable locale peut être utilisé à partir de sa définition jusqu'à la fin de la première instruction composée ou bloc ($\{\ldots\}$) qui contient sa définition
- Deux variables peuvent avoir le même nom mais dans ce cas avoir deux portées différentes

Portée d'un identificateur (suite et fin)

- Par exemple, on peut utiliser le même nom de variable locale i dans deux fonctions différentes : les portées des deux variables sont alors disjointes
- On peut aussi avoir la situation suivante :

```
{
    int i=3;                          i  3
        {
            int i=5;                  i  3        i  5
            cout << i; // affiche 5
        }
    cout << i; // affiche 3           i  3
}
```

- Les deux variables i ont deux portées différentes : la portée de la seconde est incluse dans la portée de la première

Effet de bord

- Modification indirecte (volontaire ou non) de la valeur d'une variable
- Étymologie : traduction mot à mot de l'expression anglaise « side effect » qui signifie « effet secondaire » (comme les effets secondaires d'un médicament)

Exemple

- Si j'écris

  y=x+1;

  l'expression à droite du = est évaluée pour obtenir une valeur. La valeur de x n'est pas modifiée
- Par contre, dans

  y=x++;

  x++ renvoie une valeur, comme une expression classique, mais modifie également la valeur de x

Définition

Définition *(Récursivité)*

- Une fonction f est définie récursivement lorsque sa définition utilise f elle-même
- Une fonction récursive est une fonction qui peut s'appeler elle-même au cours de son exécution

- Le plus souvent, une définition récursive est
  - plus élégante
  - plus simple
  - plus lisible
- La forme est toujours une définition par cas
  - un cas général, dans lequel l'objet défini intervient
  - un (ou plusieurs) cas particulier(s), dans le(s)quel(s) on a une valeur immédiate

Observations

- Une fonction est définie récursivement lorsque la valeur de la fonction en un point x est définie par rapport à sa valeur en un point strictement « plus petit »
- De ce fait, le calcul se fait de proche en proche, jusqu'à atteindre le plus petit élément pour lequel il faut une valeur immédiate (c'est-à-dire non récursive)
- Le corps d'une fonction récursive doit toujours exprimer un choix, soit par une expression conditionnelle, soit par une définition par cas
- Toutes les applications de la fonction définie doivent l'être sur des valeurs plus petites que celle de l'argument
- Au moins un cas terminal rend une valeur qui n'utilise pas la fonction définie

Exemple

- La factorielle d'un nombre x ($x!$) peut s'écrire ainsi :

$$\begin{array}{rccc} fact: & \mathbb{N}^* & \longrightarrow & \mathbb{N}^* \\ & x & \longmapsto & x \times (x-1) \times (x-2) \times \cdots \times 2 \times 1 \end{array}$$

- Le code en C++ donne :

Version récursive

```
int factRec (int x) {
    if (x == 1)
        return 1;
    return x*fact(x−1);
}
```

Version itérative

```
int factIter (int x) {
    int f = 1;
    while (x > 1)
        f = f * n−−;
    return f;
}
```

- Remarque : en toute rigueur, on devrait utiliser des **unsigned int** afin de prendre en compte la spécification dans son ensemble ($\mathbb{N}^*$)

Citation

« *Et ma pile !* »
John Steed

Les fondamentaux

- Il est parfois nécessaire de définir nos propres types
- Ces nouveaux types peuvent :
  - redéfinir un type existant, par exemple pour lui donner un nom plus explicite (e.g. : **typedef char** caractere;)
  - composer des types existants par collection de valeurs de même type

Exemple

Pour définir un tableau t d'entiers à deux dimensions, on peut utiliser l'instruction **typedef vector<vector<int> >** t;
ou encore les expressions :

typedef vector<int> tabInt;
vector<tabInt**>** t;

  - Restreindre des types existants à certaines valeurs (**enum**)
  - composer des types existants par collection de valeurs de types différents (**struct**)

L'instruction *typedef*

- Le mot clé **typedef** permet de définir un nouveau type de données
- Syntaxe :

  typedef <caractéristiques du type> <nom du type>

  où
  - **caractéristiques du type** représente un type de données existant (e.g. double, int, ...)
  - **nom du type** définit le nom que vous donnez au nouveau type de donnée

Exemple

```
typedef vector<char> VectCarac;
int main () {
    VectCarac c(5);
    c[3] = 'a';
}
```

?	?	?	?	?
?	?	?	a	?

Les types énumérés

- Pour rendre plus lisible et compréhensible un programme, on peut utiliser un type énuméré qui permet d'associer un nom aux valeurs
- Syntaxe :

  enum identificateur { $enum_1, enum_2, \ldots, enum_n$ };

  où
  - **identificateur** est le nom (éventuellement vide) donné au type
  - **enum$_i$**, $1 \leq i \leq n$ sont les valeurs possibles pour une variable de ce type appelées constantes énumératives

Exemple

enum JourDeLaSemaine
{ Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche };

JourDeLaSemaine j;
j = Mardi;

Valeur des constantes énumératives

- Si aucune valeur pour les constantes énumératives n'est spécifiée, leurs valeurs débutent par 0 et augmentent de 1 à chaque descente dans la liste

Exemple

```
enum Feux { Vert, Orange, Rouge };
```

Dans cet exemple, Vert vaut 0, Orange vaut 1 et Rouge vaut 2

- Il est possible de donner des valeurs explicites aux constantes énumératives

Exemple

```
enum Feux { Vert = 1, Orange = 0, Rouge = -1 };
```

- Attention : il est conseillé, si on opte pour cette possibilité, de donner des valeurs à l'ensemble des constantes énumératives (risques de conflits de valeurs)

Règles de conversion de type énuméré

- Il existe une conversion implicite d'un type énuméré vers les entiers

Exemple

Considérons le type suivant Feux déclaré précédemment
Alors les lignes suivantes sont légales :

```
int i = Orange;       // Alors i vaut 1
int j = Rouge + 3;    // Alors j vaut 5
```

- En revanche, il n'existe pas de conversion implicite d'un entier vers un type énuméré

Exemple

L'instruction suivante ne peut être acceptée par le compilateur :

```
Feux f = 2;
```

Mais ceci est accepté :

```
Feux f = (Feux)2;
```

Opérations sur les types énumérés

- Valeur : position dans la déclaration
  (**int**) lundi $\rightsquigarrow$ 0
- Comparaison (e.g. Mercredi < Vendredi)
- Utilisation dans les boucles
  for (JourDeLaSemaine int i=0; i<=Vendredi; i++) . . .
- Utilisation dans les sélections de cas
  switch (JourDeLaSemaine j) {
  case Lundi : . . . }
- Écriture (e.g. **cout** << Mardi; $\rightsquigarrow$ 1)
- Remarques
  - les entiers, les booléens, les caractères sont des types énumérés prédéfinis ; par contre, les réels ne le sont pas
  - les opérateurs habituels d'écriture ne peuvent être utilisés pour les types énumérés (sauf spécification du programmeur)

Les structures

- Dans un tableau, toutes les composantes doivent être du même type
- Lorsque l'on souhaite regrouper dans un même type des valeurs ayant des types différents, on utilise la notion de structure
- Une structure contiendra un nombre fixe de composants appelés champs de types quelconques
- Syntaxe :

  struct identificateur {
  $champ_1$: $type_1$;
  ...
  $champ_n$: $type_n$;
  };

- où
  - **identificateur** est le nom de la structure
  - **champ$_i$**, $1 \leq i \leq n$ les champs de la structure

Précisions sur les structures

- Chaque champs est désigné par un sélecteur (identifiant) qui doit être différent des autres :

$$champ_1 \neq \cdots \neq champ_n$$

Exemple

```
const int NBEQUIP = 20;

struct match {
    int gagne, nul, perdu;
    int marques, encaisses;
};
```

```
struct club {
    string nom;
    match joues;
    int places;
};

typedef vector<club> division;
division d(NBEQUIP);
```

Opérations sur les structures

- Affectation (e.g. d[i] = d[j];)
- Utilisation d'une composante
  - Pour désigner un champ d'une variable de type enregistrement, on indique le nom de cette variable suivi d'un point puis du sélecteur de ce champ

Exemple

d[i].nom (de type **string**)
d[i].joues.gagne (de type **int**)

  - on peut alors effectuer toutes les opérations définies sur le type de la composante
- Fonctions membres : il est possible de déclarer des fonctions propres à une structure

Les tableaux à deux dimensions

- Nous avons vu (et revu) comment déclarer et utiliser un tableau à une dimension
- Il est parfois nécessaire de manipuler de tableaux à deux dimensions appelés matrices

Exemple

L'instruction **vector**<**vector**<**int**> > t;

définit un tableau t à deux dimensions
Pour l'initialiser, on peut utiliser l'instruction

vector<**vector**<**int**> > t(100, **vector**<**int**>(50,1));

i.e. on initialise chacune des 100 cases de *t* avec un tableau de taille 50 rempli de 1
On accède à une case du tableau par une instruction du type t[i][j], avec i et j des entiers

Les listes

- Il existe différents types de listes :
  - les files : le premier élément entré est également le premier sorti (e.g. les documents à imprimer)
  - les piles : le premier élément entré est le dernier sorti (e.g. une pile d'assiettes)
- Selon le type de la liste (file ou pile), les données ne seront pas gérées dans le même ordre
- Les files sont qualifiées de FIFO pour *First In, First Out*

$\rightarrow$	1	2	...	n	$\rightarrow$

- Les piles sont qualifiées de FILO pour *First In, Last Out*

1	2	...	n	$\leftrightarrows$

Les files : primitives

Les primitives qui suivent sont les fonctions de base qui permettent de manipuler les files

- « Enfiler » : ajoute un élément dans la file (« Push »)
- « Défiler » : renvoie le prochain élément de la file, et le retire de la file (« Pop »)
- « La file est-elle vide ? » : renvoie « vrai » si la file est vide, « faux » sinon
- « Nombre d'éléments dans la file » : renvoie le nombre d'éléments dans la file

Les piles : primitives

Les primitives qui suivent sont les fonctions de base qui permettent de manipuler les piles

- « Empiler » : ajoute un élément sur la pile (« Push »)
- « Dépiler » : enlève un élément de la pile et le renvoie (« Pop »)
- « La pile est-elle vide ? » : renvoie « vrai » si la file est vide, « faux » sinon
- « Nombre d'éléments dans la pile » : renvoie le nombre d'éléments dans la pile

Entrées/Sorties

- Nous savons déjà écrire sur la sortie standard (e.g. **cout** $<<$ n;)
- Utilisation de l'opérateur $<<$ à deux opérandes :
  - le « flot de sortie » concerné (ici **cout**)
  - l'expression dont on souhaite écrire la valeur (ici n)
- Nous savons lire sur l'entrée standard (e.g. **cin** $>>$ x;)
- Utilisation de l'opérateur $>>$ à deux opérandes :
  - le « flot d'entrée » concerné (ici **cin**)
  - la variable où on souhaite lire une information (ici x)
- Un flot peut être connecté à un pariphérique ou à un fichier
- Par convention, le flot **cout** (resp. **cin**) est connecté à la « sortie standard » (resp. l'« entrée standard »)
- Généralement, l'entrée standard (resp. la sortie standard) correspond par défaut au clavier (resp. à l'écran)
- Il est possible de rediriger ces flots (cf. cours de système)

Les flots

- En dehors des flots prédéfinis (il en existe d'autre que ceux standard), l'utilisateur peut définir lui même d'autres flots qu'il pourra connecter à un fichier de son choix
- Un flot est un objet d'un type particulier :
  - **ostream** pour le flot de sortie
  - **istream** pour le flot d'entrée
- Il est possible d'utiliser l'opérateur $<<$ (resp. $>>$) sur les flots de sortie (resp. d'entrée)
- Il est nécessaire d'incorporer le fichier en-tête **iostream** comme pour les flots prédéfinis

#include <iostream>

Connexion d'un flot de sortie à un fichier

- Utiliser un objet de type **ofstream** (dérivé de ostream)
- Incorporer le fichier en-tête **fstream**, en plus du fichier iostream
- Tout objet de type ofstream nécessite à sa construction deux arguments :
  - le nom du fichier concerné (sous forme de chaîne de caractères)
  - un mode d'ouverture défini par une constante entière (cf. suite)

Exemple

```
#include <iostream>
#include <fstream>
...
ofstream sortie("toto.dat", ios::out);
```

Connexion d'un flot de sortie à un fichier (suite ...)

- L'objet *sortie* est donc associé au fichier nommé "toto.dat", ouvert en écriture
- Une fois construit un objet de type ofstream, l'écriture dans le fichier qui lui est associé peut se faire comme pour n'importe quel flot en faisant appel à toutes les facilités du type ostream
- Lorsque l'on a fini d'écrire dans un fichier, il est nécessaire de fermer le flot à l'aide de l'instruction **close()**

Exemple

Après la déclaration précédente de *sortie*, nous pouvons employer les instructions telles que

```
sortie << 10 << "blabla" << 20 << endl;
sortie.close();
```

Connexion d'un flot d'entrée à un fichier

- Utiliser un objet de type **ifstream** (dérivé de istream)
- Incorporer le fichier en-tête **fstream**, en plus du fichier iostream
- Tout objet de type ifstream nécessite à sa construction deux arguments :
  - le nom du fichier concerné (sous forme de chaîne de caractères)
  - un mode d'ouverture défini par une constante entière (cf. suite)

Exemple

```
#include <iostream>
#include <fstream>
...
ifstream entree("titi.dat", ios::in);
```

Connexion d'un flot d'entrée à un fichier (suite ...)

- L'objet *entree* est donc associé au fichier nommé "titi.dat", ouvert en lecture
- Une fois construit un objet de type ifstream, la lecture dans le fichier qui lui est associé peut se faire comme pour n'importe quel flot en faisant appel à toutes les facilités du type istream
- Lorsque l'on a fini de lire dans un fichier, il est nécessaire de fermer le flot à l'aide de l'instruction **close()**

Exemple

Après la déclaration précédente de *entree*, nous pouvons employer les instructions telles que

```
entree >> element;
entree.close();
```

Les différents modes d'ouverture d'un fichier

- Le mode d'ouverture est défini par un mot d'état dans lequel chaque bit correspond à une signification particulière
- Pour activer plusieurs modes d'ouverture, il suffit de faire appel à l'opérateur | (e.g. ios::out|ios::trunc)

Bit	Action
ios::in	Ouverture en lecture (obligatoire pour ifstream)
ios::out	Ouverture en écriture (obligatoire pour ofstream)
ios::app	Ouverture en ajout de données (écriture en fin de fichier)
ios::trunc	Si le fichier existe, son contenu est perdu
ios::ate	Ouverture en lecture et écriture en fin de fichier
ios::binary	Utilisé seulement dans les systèmes qui distinguent fichiers texte des autres

Citation

« *Dieu a créé l'homme à son image. Ensuite, l'homme a évolué. Dieu, lui, on ne sait pas ...* »

Philippe Geluck

Le Chat – Le chat à Malibu

Avant propos

- Les algorithmes de tri sont très étudiés depuis longtemps en algorithmique
- Ils révèlent les difficultés de la « théorie de la complexité » et l'importance d'écrire des algorithmes « efficaces »
- Dans les exemples donnés ici, nous trions des tableaux d'entiers, mais ces algorithmes peuvent être adaptés selon les besoins
- Il est nécessaire, lorsque l'on trie des données, de savoir permuter deux éléments
- La procédure **echanger** :

```
void echanger(vector<int> & tab, int i, int j) {
    int memoire = tab[i];
    tab[i]=tab[j];
    tab[j]=memoire;
}
```

Présentation du « bubble sort »

- L'algorithme consiste à regarder les différentes valeurs adjacentes d'un tableau et à les permuter si le premier des deux éléments est supérieur au second
- Déroulement de l'algorithme
  - comparer les 2 premiers éléments
  - si le premier élément est supérieur au second, les permuter
  - faire de même avec les éléments 2 et 3, 3 et 4, ... (n-1) et n
  - une fois une étape achevée, le dernier élément du tableau est le plus grand
  - l'algorithme reprend pour les (n-1) éléments qui précèdent
  - l'algorithme termine lorsqu'il n'y a plus de permutations possibles
- Il faut donc au pire effectuer l'algorithme n fois pour trier n valeurs

Sur un exemple

Évolution du tableau au fil de l'algorithme (en rouge, les éléments qui sont comparés, et éventuellement permutés, pour passer à la ligne suivante)

5	3	1	2	6	4
3	5	1	2	6	4
3	1	5	2	6	4
3	1	2	5	6	4
3	1	2	5	6	4
3	1	2	5	4	6
1	3	2	5	4	6
1	2	3	5	4	6
1	2	3	5	4	6
1	2	3	4	5	6

Code source

```
void triBulle (vector<int> & tableau, int longueur) {
    int i; bool inversion;
    do {
        inversion=false;
        for(i=0; i<longueur-1; i++) {
            if (tableau[i]>tableau[i+1]) {
                echanger(tableau,i,i+1);
                inversion=true;
            }
        }
    } while (inversion);
}
```

Code source optimisé

```
void triBulle2 (vector<int> & tableau, int longueur) {
    int i; bool inversion;
    do {
        inversion=false;
        for(i=0; i<longueur-1; i++) {
            if (tableau[i]>tableau[i+1]) {
                echanger(tableau,i,i+1);
                inversion=true;
            }
        }
        longueur--;
    } while (inversion);
}
```

Présentation du tri par création

- Algorithme assez complexe pour des résultats d'efficacité restreints
- Permet de conserver une copie du tableau original (on peut aussi le faire avec les autres algorithmes)
- Déroulement de l'algorithme
  - créer un tableau vide de même taille que le tableau à trier
  - chercher dans le tableau à trier le plus petit élément afin de le placer en première position du tableau nouvellement créé
  - recherche successive les éléments suivants afin de les placer, à la suite, dans le nouveau tableau
  - l'algorithme termine lorsque tous les éléments du tableau ont été ajoutés

Sur un exemple

Soit le tableau à trier suivant :

5	3	1	2	6	4

Le nouveau tableau créé va évoluer ainsi au fil de l'algorithme :

1	?	?	?	?	?
1	2	?	?	?	?
1	2	3	?	?	?
1	2	3	4	?	?
1	2	3	4	5	?
1	2	3	4	5	6

Code source

```
int suivant (vector<int> tableau, int longueur,
                int positionDer, int maxi) {
    int i, valeurDer=tableau[positionDer],
        positionSuiv=-1, valeurSuiv=maxi;
    for (i=0; i<longueur; i++) {
        if (tableau[i]==valeurDer && i > positionDer)
            return (i);
        if (tableau[i]>valeurDer && tableau[i]<valeurSuiv) {
            valeurSuiv=tableau[i];
            positionSuiv=i;
        }
        if (tableau[i]==maxi && positionSuiv==-1)
            positionSuiv=i;
    }
    return (positionSuiv);
}
```

Code source (suite ...)

```
void triCreation (vector<int> source, vector<int> & destination,
                  int longueur) {
    int i, positionDer=0, mini=source[0], maxi=source[0];
    for (i=1; i<longueur; i++)
        if (source[i]<mini) {
            positionDer=i;
            mini=source[i];
        } else if (source[i]>maxi)
            maxi=source[i];
    destination[0]=mini;
    for (i=1; i<longueur; i++) {
        positionDer=suivant (source, longueur, positionDer, maxi);
        if (positionDer!=-1)
            destination[i]=source[positionDer];
        else cout << "Erreur dans le triCreation : élément perdu !";
    }
}
```

Présentation du tri par sélection

- Un des tris les plus instinctifs
- Principe : classer n valeurs, rechercher la plus grande et la placer en fin de liste, puis la plus grande valeur dans les valeurs restante et la placer en avant dernière position, etc
- Déroulement de l'algorithme
  - rechercher la position de plus grand élément d'un tableau à n valeurs
  - échanger cet élément avec le dernier élément du tableau
  - l'algorithme reprend pour les n-p premiers éléments, avec p le nombre d'itérations de l'algorithme
  - l'algorithme termine quand p=(n-1)

Sur un exemple

Évolution du tableau au fil de l'algorithme (en rouge, les valeurs déjà traitées)

5	3	1	2	6	4
5	3	1	2	4	6
4	3	1	2	5	6
2	3	1	4	5	6
2	1	3	4	5	6
1	2	3	4	5	6

Code source

```
void triSelection (vector<int> & tableau, int longueur) {
    int maxi, i;
    while (longueur>0) {
        maxi=0;
        for (i=1; i<longueur; i++)
            if (tableau[i]>tableau[maxi])
                maxi=i;
        echanger (tableau, maxi, (longueur-1));
        longueur--;
    }
}
```

Présentation du tri par insertion

- Algorithme que l'on peut qualifier de naïf
- Principe : piocher une à une les valeurs du tableau et les insérer, au bon endroit, dans le tableau trié constitué des valeurs précédemment piochées et triées
- Déroulement de l'algorithme
  - piocher une valeur parmi celles non encore triées (la seconde case pour la première itération de l'algorithme)
  - soit p l'indice de la valeur piochée, les (p-1) premières valeurs du tableau constituent le tableau trié dans lequel va être inséré la $p^{ième}$ valeur
  - l'algorithme reprend pour p=2, p=3, . . . , p=n, avec n la taille du tableau

Sur un exemple

Évolution du tableau au fil de l'algorithme (en rouge, les valeurs déjà traitées, en bleu la valeur de la mémoire qui contient la valeur à insérer)

Tableau						Mémoire
5	3	1	2	6	4	3
3	5	1	2	6	4	1
1	3	5	2	6	4	2
1	2	3	5	6	4	6
1	2	3	5	6	4	4
1	2	3	4	5	6	

Code source

```
void triInsertion (vector<int> & tableau, int longueur) {
    int i, memory, compt, marqueur;
    for (i=1; i<longueur; i++) {
        memory=tableau[i];
        compt=i-1;
        do {
            marqueur=false;
            if (tableau[compt]>memory) {
                tableau[compt+1]=tableau[compt--];
                marqueur=true;
            }
            if (compt<0)
                marqueur=false;
        } while (marqueur);
        tableau[compt+1]=memory;
    }
}
```

Présentation du tri fusion

- Construit suivant la stratégie « diviser pour régner »
- Principe : pour résoudre un gros problème, il est souvent plus facile de le diviser en petits problèmes élémentaires
- Une fois chaque petit problème résolu, il n'y a plus qu'à combiner les différentes solutions pour résoudre le problème global
- Déroulement de l'algorithme
  - division de l'ensemble de valeurs en deux parties
  - tri de chacun des deux ensembles
  - fusion des deux ensembles

Sur un exemple

Évolution du tableau au fil de l'algorithme (les commentaires expliquent ce qui est fait pour passer d'une ligne à l'autre)

Tableau										Commentaire
6	3	0	9	1	7	8	2	5	4	Division du tableau
6	3	0	9	1	7	8	2	5	4	Division de chaque sous-tableau
6	3	0	9	1	7	8	2	5	4	Division de chaque sous-tableau
6	3	0	9	1	7	8	2	5	4	Division des tableaux bleu-rouge, fusion des tableaux vert-rose
6	3	0	1	9	7	8	2	4	5	Fusion des tableaux bleu-rose et rouge-rose
3	6	0	1	9	7	8	2	4	5	Fusion des tableaux bleu-jaune et rouge-jaune
0	3	6	1	9	2	7	8	4	5	Fusion des tableaux bleu-vert et rouge-vert
0	1	3	6	9	2	4	5	7	8	Fusion des deux tableaux
0	1	2	3	4	5	6	7	8	9	Le tableau est trié, l'algorithme est terminé

Code source

```
void fusion (vector<int> & tableau, int deb1, int fin1, int fin2) {
    vector<int> table1(fin1–deb1+1);
    int deb2=fin1+1, compt1=deb1, compt2=deb2, i;
    for (i=deb1; i<=fin1; i++)
        table1[i–deb1]=tableau[i];
    for (i=deb1; i<=fin2; i++)
        if (compt1==deb2)
            break;
        else if (compt2==(fin2+1))
            tableau[i]=table1[compt1++–deb1];
        else if (table1[compt1–deb1]<tableau[compt2])
            tableau[i]=table1[compt1++–deb1];
        else tableau[i]=tableau[compt2++];
}
```

Code source (suite ...)

```
void triFusionBis (vector<int> & tableau, int deb, int fin) {
    if (deb!=fin) {
        int milieu=(fin+deb)/2;
        triFusionBis (tableau, deb, milieu);
        triFusionBis (tableau, milieu+1, fin);
        fusion (tableau, deb, milieu, fin);
    }
}

void triFusion (vector<int> & tableau, int longueur) {
    if (longueur>0)
        triFusionBis (tableau, 0, longueur–1);
}
```

Questions ?