# ALGORITHME ET PROGRAMMATION II

## Algorithmique/Programmation C/C++


**Pape Abdoulaye BARRO, Ph.D,**

**Enseignant-chercheur**
UFR des Sciences et technologies
Département Informatique

**E-LabTP**, Laboratoire des TP à Distance, UFR-SET,
**Marconi-Lab**, Laboratoire de Télécommunications, ICTP, Italie

**Email**: pape.abdoulaye.barro@gmail.com

# PRÉSENTATION ET OBJECTIF DU COURS

- **Organisation du travail (36h)**
  - Cours magistral
  - TD/TP
- **Evaluation**
  - Contrôles TP
  - Examen écrit
- **Outils de travail**
  - Visual C++
  - Dev-C ++
- **Prérequis**
  - Aucun

# GÉNÉRALITÉS

.....0100111010100.....

# LES BASES

## DÉFINITION 1:

L'algorithme est une suite d'instructions élémentaires, qui une fois exécutée correctement, conduit à un résultat donné.

- Il vient du mathématicien et astronome perse Muhammad ibn al-Khawarizmi, le père de l'algèbre, qui formalisa au 9e siècle la notion d'algorithme ;
- L'algorithme le plus célèbre est l'algorithme d'Euclide (permettant de calculer le PGCD de deux nombres dont on ne connait pas la factorisation).

# LES BASES

Les ***instructions*** et les ***données*** sont codées sous forme de nombres binaires qu'on appelle des ***mots***.

- Un ordinateur ne manipule que deux valeurs : 0 ou 1. En effet, nos ordinateurs sont constitués de circuits intégrés qui sont composés de nombreuses pistes dans lesquelles passe un courant électrique. Or, dans ces circuits il n'y a que deux possibilités : soit le courant passe et dans ce cas cela équivaut à une valeur de un (1), soit le courant ne passe pas, et dans ce cas c'est la valeur zéro (0) qui est retenue. C'est du binaire. Une unité binaire s'appelle un bit (binary digit), un mot inventé par **Claude Shannon** en 1948.

# LES BASES

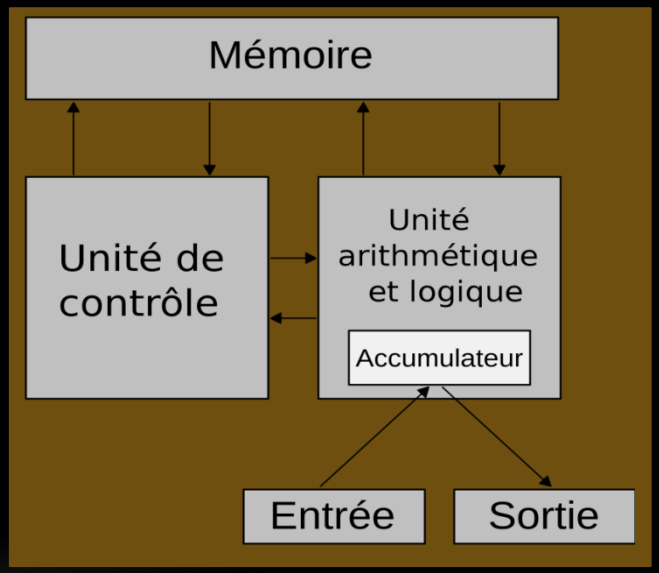
Les bits ne sont pas stockés individuellement dans une case mémoire. Ils sont regroupés, généralement par multiples de huit (8) c'est-à-dire en octet (qui représente les valeurs de 0 à 255).

- Avec l'augmentation des espaces de stockages, de la quantité de mémoire, du besoin de représentation de nombres de plus en plus grands, d'un accès plus rapide à la mémoire ou encore de plus d'instructions, il a fallu augmenter la taille des valeurs à manipuler. De 8, puis 16, puis 32, certains microprocesseurs peuvent manipuler des valeurs de 64 voire 128 bits, parfois plus... Ces valeurs deviennent difficiles à décrire et à représenter. Pour ces valeurs, on parle de mot mémoire.

# LES BASES

Un ordinateur est un système de traitement de l'information programmable qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations arithmétiques et logiques.

Les ordinateurs actuels sont tous basés sur des versions améliorées de l'architecture de **Von Neumann** (1944).



# LES BASES

L'architecture de Von Neumann est composée de 4 parties distinctes.

- l'unité arithmétique et logique (UAL ou ALU en anglais) ou unité de traitement qui a pour rôle d'effectuer les opérations de base. Certaines documentations lui rajoute des registres (quelques cases mémoires intégrés) et lui confère le nom de de processeur (CPU) ;
- l'unité de contrôle ou de commande (control unit) qui est chargée du séquençage des opérations ou le déroulement du programme. Elle récupère les instructions en mémoire et donne des ordres à l'ALU ;
- la mémoire (une suite de petites cases numérotées appelées registre) contient à la fois les données et le programme indiquant à l'unité de contrôle les calculs à faire. Pour pouvoir accéder à la mémoire, il suffit de connaitre son adresse;
- les dispositifs d'entrée-sortie permettent de communiquer avec le monde extérieur. Il peut s'agir d'un clavier pour entrer les données et d'un écran pour visualiser les résultats.

# LES BASES

**DÉFINITION 2:**

La programmation peut être vue comme l'art de déterminer un algorithme (une démarche) pour résoudre un problème et d'exprimer cet algorithme au moyen d'un langage de programmation (exemple C, C++, PYTHON, PHP, JAVA, etc.).

- La programmation est donc une activité fondamentale en informatique.

# LES BASES

L'efficacité d'un algorithme est fondamentale pour résoudre effectivement des problèmes. L'efficacité d'un algorithme est mesurée par son coût (complexité) en temps et en mémoire.

- La complexité d'un algorithme est en temps, le nombre d'opérations élémentaires effectuées pour traiter une donnée de taille n, en mémoire, l'espace mémoire nécessaire pour traiter une donnée de taille n.
  - **Exemple**: lancé un dé est un algorithme très simple, court, concis et rapide. Ce n'est pas toujours le cas pour d'autres algorithmes. Certains pourront nécessiter beaucoup de temps et de ressources.

# LE FORMALISME

# LE FORMALISME

Prenons l'exemple du jeu dé pour mieux représenter les différentes étapes.

- 1ère étape : lancer le dé;
- 2ème étape : saisir une valeur;
- 3ème étape : si la valeur saisie est différente de la valeur du dé, retourner à la *première étape*, sinon continuer;
- 4ème étape : afficher "bravo".

# LE FORMALISME

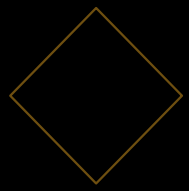
## LA REPRÉSENTATION GRAPHIQUE

Un **organigramme** est constitué de symboles dont les formes sont normalisées. Ces symboles sont reliés entre eux par des lignes fléchées qui indiquent le chemin. Ainsi, nous avons:

Représente un début ou une fin

Utilisé pour la lecture ou l'écriture

Utilisé pour les calculs

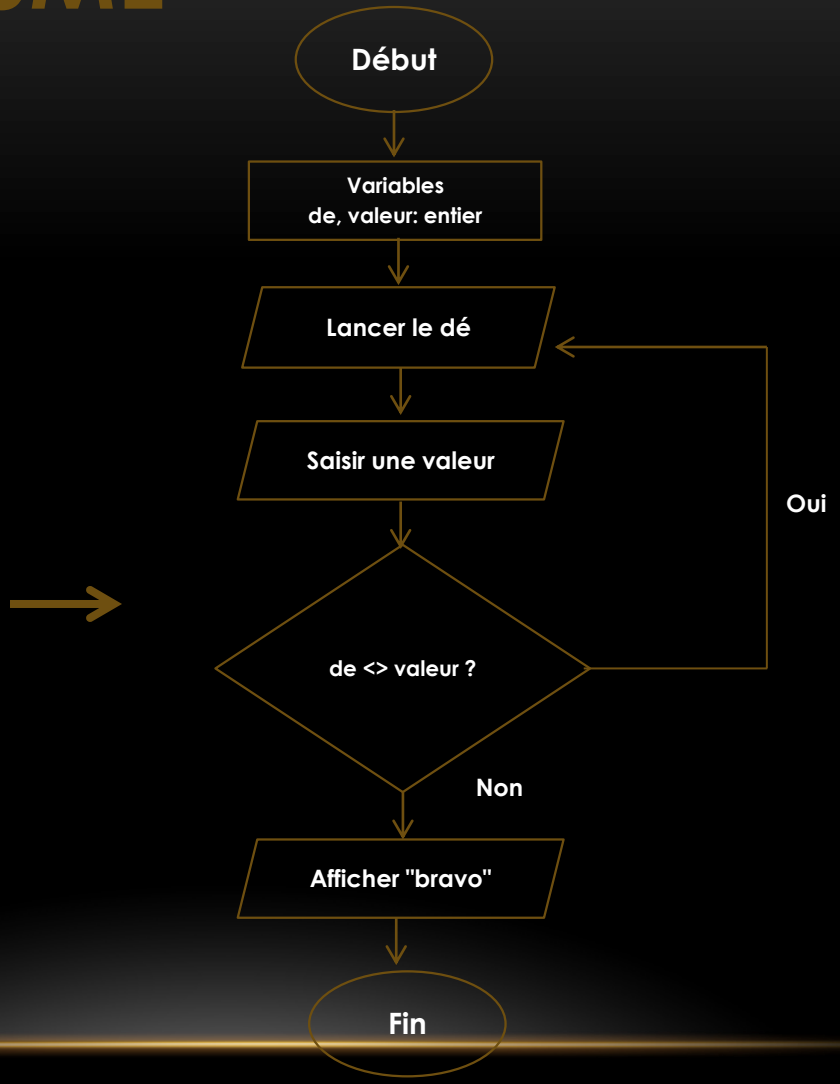


Représente les Tests

# LE FORMALISME

## L'ORGANIGRAMME

La représentation graphique du jeu de dé.



# LE FORMALISME

## L'ALGORITHME SOUS FORME DE TEXTE

Ecrire un algorithme sous forme de texte est sans doute la manière la plus simple de faire comprendre à un non informaticien ce que votre code est censé faire.

La syntaxe utilisée est simple, précise et concise.

Reprenons notre exemple sur le jeu de dé.

```
/* Commentaires : jeu de dé */
/* Nom du programme  */
ALGORITHME jeu_de_de
/* Déclarations des variables, constantes, types, etc */
VARIABLES
        de:entier;
        valeur:entier;
/* Déclarations de fonctions*/
/* Début du programme */
DEBUT
        de←aléatoire(6);
        valeur←0;
        Tantque valeur<>de Faire
                Ecrire ("saisir une valeur ");
                Lire (valeur);
        FinTantQue
        Ecrire ("Bravo");
FIN
```

# LE FORMALISME

## L'ALGORITHME SOUS FORME DE TEXTE

Ce pseudocode algorithmique, est décomposé en plusieurs parties.

- Le nom de l'algorithme situé après le mot "ALGORITHME",
- Une zone de déclaration des données utilisées par le programme. Cette zone commence par le mot "VARIABLES",
- Les instructions du programme sont encadrées par les mots "DEBUT" et "FIN",
- Chaque instructions est terminée par un point-virgule " ; ",
- Les commentaires peuvent être encadrés par séquences de caractères "/*" et "*/" s'il s'agit de plusieurs lignes ou par "//" s'il s'agit d'une seule ligne.
- L'affectation permet de donner une valeur à une variable : valeur <- 0 « reçoit ». Si valeur avait une valeur auparavant, cette valeur disparaît. Généralement, le formalisme est la suivante :

<id_variable> <- <expression>;

- Les opérations entrées/sorties permettent de récupérer une valeur venant de l'extérieur (Lire) ou de transmettre une valeur à l'extérieur (Ecrire).

# LE FORMALISME

## LE PROGRAMME EN C

Les lignes en haut qui commence par # sont appelées directives de préprocesseur (un programme qui se lance au début de la compilation).

Elles ajoutent avec le mot clé include des fichiers qui existe déjà pour la compilation. Ces fichiers sont appelés des bibliothèques, librairies en anglais).

Exemple, stdio.h (pour standard input output) est une bibliothèque qui vous permet entre autres d'interagir avec l'utilisateur.

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

int main(){
        // déclaration
        int de, valeur ;
        srand( time( NULL ) );
        de = rand() % 6 + 1 ;
        valeur = 0 ;
        while(valeur!=de){

                printf("veuillez saisir une
                valeur \n");
                scanf("%d", &valeur);
        }
        printf("Bravo !");

        return 0;

}
```

# LE FORMALISME

## LE PROGRAMME EN C

- **main** est la fonction principale. Le programme, commence toujours avec la fonction main. Elle débute avec l'accolade ouvrante « { » et se termine avec l'accolade fermante « } ». Chaque ligne à l'intérieur de main est appelée instruction.
- printf affiche un message à l'écran. Exemple : printf("veuillez saisir une valeur ").
- antislash (\) puis une seconde lettre indique qu'on veut aller à la ligne (\n) ou faire une tabulation (\t), etc…
- scanf récupère ce que l'utilisateur entre dans la console sous demande de la fonction printf. Il prend en entrée, le format de la donnée (int⇔%d, float⇔%f, etc.) et le nom de la variable en question précédé du symbole « & » .
- return 0 indique qu'on est arrivé à la fin de la fonction main en renvoyant la valeur 0. Cela n'empêchera pas le programme de fonctionner mais c'est plus sérieux de le mettre.

# LE FORMALISME

## LE PROGRAMME EN C++

- Comme dans C, nous avons les directives de préprocesseur. Ici, la standard est iostream « Input Output Stream », ce qui veut dire « Flux d'entrée-sortie » et donc est diffèrent de stdio de C.
- using namespace est un espace de noms. Son rôle est d'éviter les problèmes d'appel de fonction de même noms se situant dans deux bibliothèque diffèrent. std correspond à la bibliothèque standard, livrée par défaut avec le langage C++ et dont iotream fait partie.
- cout (comme printf en C) permet d'afficher un message à l'écran. Les morceaux de texte sont séparés par les chevrons ouvrants (<<) et le mot clé endl est utilisé pour réaliser des retours à la ligne.
- cin (comme scanf en C) permet de faire entrer des informations dans le programme. Les chevrons fermants (>>) sont utilisés pour cela.

```
# include <iostream>
#include <cstdlib>
#include <time.h>

using namespace std ;
int main(){
        // déclaration
        int de, valeur ;
        srand( time( NULL ) );
        de = rand() % 6 + 1 ;
        valeur = 0 ;
        while(valeur!=de){
                cout << "veuillez sairir une
                valeur " << endl;
                cin >> valeur ;
        }
        cout << "Bravo !" << endl;

        return 0;
}
```

# LES VARIABLES, LES OPÉRATEURS ET LES OPÉRATIONS

.....< + - * / ^ >.....

# VARIABLE

## DÉFINITION 3:

Une variable est un espace mémoire nomme, de taille fixée, prenant au cours du déroulement de l'algorithme un nombre indéfini de valeurs différentes.

- Un algorithme tourne généralement autours des variables;
- Le changement de valeur se fait par l'opération d'affectation.
- La variable diffère de la notion de constante qui, comme son nom l'indique, ne prend qu'une unique valeur au cours de l'exécution de l'algorithme.

# VARIABLE

## TYPE DE VARIABLE :

Dans la plupart des langages de programmation, avant de manipuler une variable, il faut préalablement déclarer son type. C'est à dire que la variable en question ne pourra changer de valeur que dans l'intervalle défini par le type qui lui est assigné.

Dans un algorithme, on se contente de 5 types de base, à savoir :

- Les entiers: qui sont des nombres sans virgule et qui peuvent être positifs ou négatifs. On parle alors de nombres entiers signés ;
- Les réels: qui sont des nombres avec virgule (dite virgule flottante) et qui peuvent être positifs ou négatifs aussi ;
- Les booléens: qui définissent deux valeurs (dites binaires) qui sont Vrai ou Faux (ou encore 1 ou 0) ;
- Les caractères: qui représentent tous les caractères alphanumériques ;
- Les chaînes de caractères: qui représentent des textes constitués de tout type de caractères comme les caractères alphabétique, numériques et symboles.

# OPERATEURS

## TYPES D'OPÉRATEURS :

Un opérateur est un outil qui permet d'agir sur une variable ou d'effectuer des calculs.

Il existe plusieurs types d'opérateurs:

- L'affectation qui confère une valeur à une variable ou à une constante. Il est représenté par le symbole «←» (= en C/C++ …) ;
- Les opérateurs arithmétiques qui permettent d'effectuer des opérations arithmétiques entre opérandes numériques :

| | |
|---|---|
| + | addition |
| - | soustraction |
| * | multiplication |
| / | division |
| mod (% en C/C++) | modulo |
| ^ (Algo) | puissance |
| div (Algo) | division entière |

# OPERATEURS

## TYPES D'OPÉRATEURS

- Les Opérateurs relationnels :

| | |
|---|---|
| > | supérieur |
| < | inférieur |
| >= | Supérieur ou égal |
| =< | Inférieur ou égal |
| = (== en C/C++) | égal |
| < > (!= en C/C++) | différent |

- Les opérateurs logiques :
  - Opérateur unaire : «non» « ! en C/C++» (négation) ;
  - Opérateurs binaires : «et» «&& en C/C++» (conjonction), «ou» «|| en C/C++» (disjonction).
- La concaténation : qui permet de créer une chaine de caractères à partir de deux chaînes de caractère en les mettant bout à bout. Il est représenté par le symbole « + ».

- sur les entiers et les réels : addition, soustraction, multiplication, division, division entière, puissance, comparaisons, modulo ;
- sur les booléens : comparaisons, négation, conjonction, disjonction ;
- sur les caractères : comparaisons ;
- sur les chaînes de caractères : comparaisons, concaténation

# OPÉRATIONS

## PRIORITÉ DES OPÉRATIONS :

Lors de l'évaluation d'une expression, la priorité de chaque opérateur permet de définir l'ordre d'exécution des différentes opérations. Pour changer la priorité d'exécution, on utilise les parenthèses.

- Ordre de priorité décroissante des opérateurs arithmétiques et de concaténation :
  - «^» ;
  - «*», «/» et «div» ;
  - «mod» ;
  - «+» et «-» ;
  - «+» (concaténation).
- Ordre de priorité décroissante des opérateurs logiques :
  - «non» ;
  - «et» ;
  - «ou».

# NOTION DE VARIABLE EN C

En C, lorsqu'on crée une variable, il faut toujours indiquer son type. Le tableau ci-dessous, donne les noms des types et leur plage.

| type | Min | Max |
|---|---|---|
| Signed char | -127 | 127 |
| int | -32 767 | 32 767 |
| long | -2 147 483 647 | 2 147 483 647 |
| float | $-1x10^{37}$ | $1x10^{37}$ |
| double | $-1x10^{37}$ | $1x10^{37}$ |

Ces types sont signés (signed) mais il existe d'autre types non signés (unsigned) qui ne stockent que des nombres positifs.

- **Exemple** : signed int peut aller jusqu'à 32 767 alors que unsigned int va jusqu'à 65 535.

# NOTION DE VARIABLE EN C

- type *nom_variable* permet de déclarer une variable. Mais le nommage des variables est régit par les règles suivantes :
  - elle commence par une lettre ;
  - les espaces sont interdits. On peut utiliser « underscore » pour cela ;
  - on ne peut pas utiliser des accents ;
  - on peut utiliser des minuscules, des majuscules et des chiffres.
- Pour déclarer une constante, il faut utiliser le mot const devant le type et il est obligatoire de lui donner une valeur au moment de sa déclaration.
- Pour afficher le contenu d'une variable avec printf, on utilise le format du type de la variable (ex : **%d** pour les **int** ou **%f** pour les **float**) à l'endroit ou l'on souhaite afficher la valeur de la variable.
- Il est possible d'afficher la valeur de plusieurs variables dans un seul printf. Il vous suffit pour cela d'indiquer des %d ou des %f là où vous voulez, puis d'indiquer les variables correspondantes dans le même ordre, séparées par des virgules.

# NOTION DE VARIABLE EN C++

En C++, c'est presque pareil, sauf que coté initialisation d'une variable, on peut décider de faire comme le C ou avec une syntaxe propre à C++ qui est : TYPE NOM (VALEUR). Aussi, pour l'affichage du contenu d'une variable avec cout et les chevrons (<<), il suffit simplement de mettre le nom de la variable à l'endroit du texte à afficher.

- Opérateur ternaire: Il permet l'affectations du type.
  - Syntaxe: Si *condition* est vraie alors *variable* vaut *valeur*, sinon *variable* vaut *autre valeur*.
  - Exemple: int *a* = (b > 0) ? 10 :20;
- Les opérateurs de manipulation de bit :
  - & : ET bit à bit
  - | : OU bit à bit
  - ^ : OU Exclusif bit à bit
  - << : Décalage à gauche
  - >> : Décalage à droite
  - ~ : Complément à un (bit à bit)
- La fonction *sizeof* est utilisée pour connaitre la taille en mémoire d'une variable passé en paramètre.
  - int a = 1; sizeof(a) donne 4; double a = 3,14; sizeof(a) donne 8.

# EXERCICES D'APPLICATIONS

## Application 1 :

Ecrire un algorithme/programme permettant de déclarer deux variables de type réel, de saisir les valeurs, de calculer et d'afficher leur somme, produit et moyenne.

## Application 2 :

Ecrire un algorithme/programme qui permet de permuter les valeurs de A et B sans utiliser de variable auxiliaire.

## Application 3 :

Ecrire un algorithme/programme permettant de déclarer trois variables A, B, C de type réel, d'initialiser leurs valeurs et ensuite d'effectuer la permutation circulaire des trois variables.

## Application 4 :

Ecrire un algorithme/programme qui permet de saisir les paramètres d'une équation du second degré et de calculer son discriminant delta.

## Application 5 :

Ecrire un algorithme/programme qui à partir de la valeur saisie du côté d'un carré donné, permet de calculer son périmètre et sa surface et affiche les résultats à l'écran.

# STRUCTURES DE CONTRÔLES

# STRUCTURES DE CONTRÔLES

Un ordinateur exécute un programme de manière séquentielle. Pour lui doter de l'intelligence relative afin d'être capable d'effectuer des choix ou des boucles sur une bloc d'instructions et de casser cette linéarité, il va falloir utiliser les structures de contrôle.

Parmi les structures de contrôle nous avons :

- LES STRUCTURES CONDITIONNELLES
- LES STRUCTURES ITERATIVES

# LES STRUCTURES CONDITIONNELLES

## INSTRUCTION CONDITIONNELLE

Syntaxe en Algorithme

```
Si(condition) alors
        {instructions}
Fin Si
```

Syntaxe en C/C++

```
if(condition) {
        /*instructions*/
}
```

Condition est une expression booléenne

**Exemple:**

```
.....

Si(jour <> 7) alors
   ecrire(''Je vais à l'école'');
Fin si
....
```

```
.....

if(jour != 7) {
   cout <<''Je vais à l'école''<<endl;
}
....
```

# LES STRUCTURES CONDITIONNELLES

## INSTRUCTION CONDITIONNELLE

Syntaxe en Algorithme

```
Si(condition) alors
        {instructions}
Sinon
        {instructions}
Fin Si
```

Syntaxe en C/C++

```
if(condition) {
        /*instructions*/
} else {
        /*instructions*/
}
```

**Exemple:**

```
.....

Si(jour <> 7 ET greve=Faux) alors
  ecrire(''Je vais à l'école'')
Sinon
  ecrire(''Il n'y a pas école'')
Fin si
....
```

```
.....

if(jour != 7 && greve==Faux) {
  cout <<''Je vais à l'école''<<endl;
} else {
  cout <<''Il n'y a pas école''<<endl;
}
....
```

# LES STRUCTURES CONDITIONNELLES

## INSTRUCTION CONDITIONNELLE - IF IMBRIQUÉES

Syntaxe en Algorithme

```
Si(condition 1) alors
        {instructions}
Sinon Si(condition 2) alors
        {instructions}

....

Sinon Si(condition n) alors
        {instructions}
Sinon
        {instructions}
Fin Si
```

**Exemple:**

```
...
Si(jour=1) alors
   Ecrire(''Lundi'')
Sinon Si(jour=2) alors
   Ecrire(''Mardi'')
Sinon Si(jour=3) alors
   Ecrire(''Mercredi'')
Sinon Si(jour=4) alors
   Ecrire(''Jeudi'')
Sinon Si(jour=5) alors
   Ecrire(''Vendredi'')
Sinon Si(jour=6) alors
   Ecrire('Samedi'')
Sinon
   Ecrire(''Dimanche'')
Fin Si
...
```

# STRUCTURE A CHOIX MULTIPLE

Elle permet dans certain cas d'éviter une abondance d'instruction if imbriquées.

---

Syntaxe en Algorithme

```
Choix selon(expression)
        cas valeur1
          {instruction 1}
        interrompre
        cas valeur2
          {instruction 2}
        interrompre
                ...
        par défaut
          {suite_instruction}
Fin cas
```

**Exemple:**

```
...
    Choix selon(jour)
            cas 1
              Ecrire(''Lundi'')
             interrompre
            cas 2
              Ecrire(''Mardi'')
             interrompre
            cas 3
              Ecrire(''Mercredi'')
             interrompre
            cas 4
              Ecrire(''Jeudi'')
             interrompre
            cas 5
              Ecrire(''Vendredi'')
            interrompre
            cas 6
              Ecrire('Samedi'')
             interrompre
             par défaut
              Ecrire(''Dimanche'')
    Fin cas
...
```

# STRUCTURE A CHOIX MULTIPLE

Syntaxe en C/C++

```
switch (expression)
{
        case valeur1:
          {instruction 1};
        break;
        case valeur2:
          {instruction 2};
        break;
                ...
        default:
          {suite_instruction} ;
}
```

**Exemple:**

```
...
   switch(jour)
    {
            case 1:
              cout <<''Lundi''<<endl;
             break;
            case 2:
              cout <<''Mardi''<<endl;
             break;
            case 3:
              cout <<''Mercredi''<<endl;
             break;
            case 4:
              cout <<''Jeudi''<<endl;
             break;
            case 5:
              cout <<''Vendredi''<<;
            break;
            case 6:
              cout <<''Samedi''<<endl;
             break;
             default:
              cout <<''Dimanche''<<endl;
    }
...
```

# EXERCICES D'APPLICATIONS

## Application 6 :

Écrivez un programme qui calcule les solutions réelles d'une équation du second degré $ax^2+bx+c = 0$ en discutant la formule:

- Utilisez une variable d'aide d pour la valeur du discriminant $b^2$ - 4*a*c et décidez à l'aide de d, si l'équation a une, deux ou aucune solution réelle. Utilisez des variables du type entier pour a, b et c. Affichez les résultats et les messages nécessaires sur l'écran.

## Application 7 :

Écrivez un programme qui permet de calculer la superficie d'un cercle, d'un rectangle ou d'un triangle. L'utilisateur saisira "C", "R" ou "T" selon la superficie de la figure qu'il souhaite calculer, ensuite il saisira les dimensions.

Selon le choix de l'utilisateur, le programme doit pouvoir lui demander de saisir les dimensions appropriées.

Afficher ensuite à l'écran selon son choix la superficie demandée

# STRUCTURES ITERATIVES

Une itération consiste en la répétition d'un blocs d'instructions jusqu'à ce qu'une certaine condition soit vérifiée.

Il en existe 2 sortes:

- Le nombre d'itérations est connu d'avance
- Le nombre d'itération dépend du résultat précédemment obtenue.

Supposons qu'on veut afficher tous les nombres entiers comprises entre 9 et 999. il va falloir faire:

```
Ecrire(''9'')
Ecrire(''10'')
…
Ecrire(''999'')
```

Une tache répétitive fastidieuse. D'où la nécessité de trouver une solution alternative.

# STRUCTURES ITERATIVES
## ITERATION POUR

Syntaxe en Algorithme

```
Pour i allant de MIN à MAX par pas de PAS faire
        {instructions}
Fin pour
```

Syntaxe en C/C++

```
for (initialisation ; condition ; incrémentation){
        /*instructions*/
 }
```

BOUCLE AVEC COMPTEUR

**Exemple:**

```
…..
Pour i allant de 9 à 999 faire
   ecrire(''i='', i)
Fin pour
….
```

```
…..
int compteur;
for (compteur = 9; compteur <1000 ; compteur++)
{
        cout << compteur << endl;
}
….
```

# STRUCTURES ITERATIVES

## ITERATION FAIRE…TANT QUE

Syntaxe en Algorithme

```
Faire
        {instructions}
Tant que (condition_de_reprise)
```

```
Répéter
        {instructions}
Jusqu'à(condition_de_sortie)
```

Syntaxe en C/C++

```
do{
        /*instructions*/
}while(condition_de_reprise);
```

le contenu de la boucle sera toujours lu au moins une fois.

**Exemple:**

```
…..
Faire
   Ecrire(''veuillez entrer un entier ?'')
   Lire(nombre)
Tant que (nombre<0)
….
Répéter
   Ecrire(''veuillez entrer un entier ?'')
   Lire(nombre)
Jusqu'à (nombre>0)
….
```

```
….
int nombre(0);
do
{
   cout << ''veuillez entrer un entier ?'' << endl;
   cin >> nombre;
} while (nombre< 0);
…
```

# STRUCTURES ITERATIVES

## ITERATION TANT QUE … FAIRE

Syntaxe en Algorithme

```
Tant que (condition) faire
        {instructions}
Fin tant que
```

Syntaxe en C/C++

```
while(condition ){
        /*instructions*/
}
```

La condition d'entrée doit être définie au préalable sinon, en implémentant votre algorithme, vous risquez d'avoir des comportements étranges.

**Exemple:**

```
…..
Ecrire(''Entrez un entier naturel'')
Lire(n)
result ← 0;
i←1;
Tant que(i<=n) faire
        result ← result + i;
        i ← i+1;
Fin tant que
Ecrire(''Somme ='', result)
….
```

```
…..
int result(0), i(1), n;
cout << ''Entrez un entier naturel ?'' << endl;
cin >> n;
while(i<=n)
{
    result = result + i;
    i = i+1;
}
cout << ''Somme ='' << result << endl;
….
```

# EXERCICES D'APPLICATIONS

## Application 8 :

Écrivez un programme qui calcule les solutions réelles d'une équation du second degré $ax^2+bx+c = 0$ en discutant la formule:

- Utilisez une variable d'aide d pour la valeur du discriminant $b^2$ - 4*a*c et décidez à l'aide de d, si l'équation a une, deux ou aucune solution réelle. Utilisez des variables du type entier pour a, b et c. On suppose que les valeurs saisies sont non nulles. Affichez les résultats et les messages nécessaires sur l'écran

## Application 9:

Ecrire un programme qui permet de faire les opérations suivantes :

- Ecrire un programme qui affiche la somme des n premiers entiers naturels. La valeur de n est saisie au clavier lors de l'exécution.
- Ecrire un programme qui affiche la somme des entiers compris entre les entiers d et f. Les valeurs de d et f sont saisies au clavier lors de l'exécution.
- Ecrire un programme qui affiche la somme des valeurs absolues des entiers compris entre les entiers relatifs d et f. Les valeurs de d et f sont saisies au clavier lors de l'exécution.

# TABLEAUX

# TABLEAU

Un tableau est une liste d'éléments ayant le même type et désignés sous le même nom et accessibles par indices (commençant par 1) .

- TABLEAU UNIDIMENSIONNEL
  - Il est déclarer comme suit:

    **nomTableau: tableau[taille] de type**
  - **Exemple**: Notes : tableau[10] de réels
- TABLEAU BIDIMENSIONNEL (on peut avoir un tableau multidimensionnel)
  - Il est déclarer comme suit:

    **nomTableau: tableau[ligne][colonne] de type**
  - **Exemple**: matrice: tableau[2][3] de réels
- **Lecture**: écrire(nomTableau[i][j][k]…[n])
- **Ecriture**: lire(nomTableau[i][j][k]…[n])
- **Affectation**: nomTableau[i][j][k]…[n] ← Valeur

# TABLEAU EN C++

- TABLEAU UNIDIMENSIONNEL
  - Il est déclaré comme suit:

    **Type nom [taille] ;**
  - **Exemple**: double notes[10] ;
- TABLEAU BIDIMENSIONNEL (on peut avoir un tableau multidimensionnel)
  - Il est déclaré comme suit:

    **type nom [ligne][colonne] ;**
  - **Exemple**: double matrice[2][3] ;
- **Lecture**: cout << nomTableau[i][j][k]…[n] << endl;
- **Ecriture**: cin >> nomTableau[i][j][k]…[n];
- **Affectation**: nomTableau[i][j][k]…[n] = Valeur;

# DÉCLARATION TYPEDEF

- Elle permet d'attribuer un nom à un type. Sa forme générale est: **typedef** <*déclaration*>
- C'est très pratique pour nommer certains types de tableaux :

```
typedef int Matrice[2][3];   // définit un type Matrice
```

- Et de l'utiliser pour déclarer ensuite, son équivalent :

```
Matrice M;
```

# TABLEAUX DYNAMIQUES - VECTOR

Un tableau dynamique est un tableau dont la taille peut varier.

- Syntaxe: **vector**<TYPE> nom(TAILLE); // Il va falloir inclure la bibliothèque <vector>
- Quelques fonctions utiles:
  - push_back (): Ajout à la fin du vecteur
  - pop_back (): retire de la fin du vecteur
  - size() : retourne le nombre d'éléments du vecteur
  - erase() : supprime un éléments ou un intervalle d'un vecteur et déplace les éléments suivants.
- **Exemples**:
  - Créer un tableau vide: vector <double> tab;
  - Créer un tableau de 10 éléments: vector <int> tab(10);
  - Créer un tableau de 10 éléments initialisés à 0:
    - vector <int> tab(10, 0);
    - tab.push_back(3); // ajout du 11ème case au tableau de valeur 3;
    - tab.pop_back(); // suppression de la dernière case du tableau;
    - int const taille(tab.size()); // variable contenant la taille du tableau;
    - tab.erase(tab.begin()+5); // suppression du 6iéme élément;
    - tab.erase(tab.begin(), tab.begin()+5); // suppression des 5 premiers éléments;

# TABLEAUX DYNAMIQUES - VECTOR

Il est également possible de créer des tableaux multidimensionnels de taille variable en utilisant les vector.

- Syntaxe pour 2D: **vector**<**vector**<TYPE>> nom; // nous avons plus tôt un tableau de ligne.

  **Exemples**:

  - vector<vector<int>> mat;
  - mat.push_back(vector(3)); //ajout d'une ligne de 3 cases;
  - mat[0].push_back(4); //ajout d'une case contenant 4 à la 1ière ligne du tableau;
  - mat[0][2] = 6; // change la valeur de la cellule (0, 2) du tableau;

# STRING ET TABLEAUX

Une chaine de caractère est en réalité un tableau de caractères. Ce qui veut dire qu'il a beaucoup de point communs avec les vector.

- Pour pouvoir utiliser la classe standard, il faut rajouter la bibliothèque <string>;
  - Elle embarque toutes les opérations de base sur les chaînes:
    - Déclaration: string s1; string s2="Hello";
    - Saisie et Affichage: cin>>s1; cout<<s2;
    - Concaténation: string s3=s1+s2;
      - s3.size(); // pour connaitre le nombre de lettres;
      - s3.push_back("!"); // pour ajouter des lettres à la fin;
      - s3.at(i); // pour récupérer le i-ème caractère;
      - getline(cin, s4); // pour saisir une chaîne de caractères en utilisant le passage à la ligne comme séparateur (notre chaîne de caractères peut alors comporter des espaces);

# STRING ET TABLEAUX

**Exemple**:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
	string prenom("Masamba");
	cout << "Je suis" << prenom << "et toi ?" <<endl;
	prenom[2] = ‘d';
	prenom[3] = ‘e';
	cout << "moi c’est" << prenom << "!" << endl;

	return 0;
}
```

# EXERCICES D'APPLICATIONS

## Application 10:

Ecrivez un programme qui lit au clavier une suite de nombres entiers positifs ou nuls et qui les affiche dans l'ordre inverse de leur lecture. La frappe d'un nombre négatif indique la fin de la série. Nous avons des raisons de penser qu'il n'y aura pas plus de 100 nombres.

## Application 11 :

On considère un tableau tab de N entiers. Ecrire un programme permettant:

a) de compter le nombre d'éléments nuls de tab
b) de chercher la position et la valeur du premier élément non nul de tab
c) de remplacer les éléments positifs par leur carré

## Application 12 :

Ecrire un programme qui permet de saisir des nombres entiers dans un tableau à deux dimensions TAB [10][20] et de calculer les totaux par ligne et par colonne dans des tableaux TOTLIG[10] et TOTCOL[20].

## Application 13 :

Ecrire un programme qui permet de chercher une valeur x dans un tableau à deux dimensions t[m][n]. Le programme doit aussi afficher les indices ligne et colonne si x a été trouvé.

# TRI D'UN TRAVAUX, RECHERCHE D'UN ÉLÉMENT

# TRI

- Etant donné une collection d'entier placés dans un tableau. L'idée fondamentale est de trier le tableau dans l'ordre croissant.
- Les opérateurs de comparaison (<=, >=, >, <, …) sont activement utilisés.
- On peut citer quelques algorithmes de tris:
  - Tris élémentaires (tris naïfs)
    - Tri par insertion
    - Tri par sélection
    - …
  - Tris avancés (Diviser pour régner)
    - Tri fusion
    - Tri rapide
    - …

# TRI
## PAR INSERTION

Le tri par insertion consiste à pré-trier une liste afin d'entrer les éléments à leur bon emplacement dans la liste triée. à l'itération i, on insère le i i-ième élément à la bonne place dans la liste des i−1 éléments qui le précède.

- Principe:
  - On commence par comparer les deux premiers éléments de la liste et de les trier dans un ordre;
  - puis un troisième qu'on insère à sa place parmi les deux précédents;
  - puis un quatrième qu'on insère à sa place parmi les trois autres;
  - ainsi de suite jusqu'au dernier.

# TRI
## PAR INSERTION

Considérons un tableau d'entiers de n éléments à trier.

- Algorithme

```
Pour (i allant de 2 à n) faire
  j ← i;
  tampon ← tab[i] ;
  Tant que (j>1 ET tab[j-1]>tampon) faire
      Tab[j] ← tab[j-1];
      j ← j-1;
  Fin tant que
  Tab[j] ← tampon;
Fin pour
```

- Complexité

Pour apprécier la complexité de cet algorithme, il suffit d'analyser le nombre de comparaisons effectué ainsi que le nombre d'échange lors du tri. On remarque qu'il s'exécute en $\Theta(n^2)$.

- $n|1\,(|2 + |3 + n|4(|5 + |6) + |7) = n(|2 + |3 + |7) + n^2(|5 + |6) \rightarrow \Theta(n^2)$

# TRI
## PAR SÉLECTION

Le tri par sélection consiste à recherche le minimum parmi les éléments non triés pour le placer à la suite des éléments déjà triés.

- Principe:
  - Il suffit de trouver le plus petit élément et le mettre au début de la liste;
  - Ensuite, de trouver le deuxième plus petit et le mettre en seconde position;
  - Puis, de  trouver le troisième plus petit élément et le mettre à la troisième place;
  - Ainsi de suite jusqu'au dernier.

# TRI
## PAR SÉLECTION

Considérons un tableau d'entiers de n éléments à trier.

- Algorithme

```
Pour (i allant de 1 à n-1) faire
    Pour (j allant de i+1 à n) faire
        Si(Tab[i] > tab[j]) alors
            tampon ← tab[i];
            tab[i] ← tab[j];
            tab[j] ← tampon;
        Fin Si
    Fin pour
Fin pour
```

- Complexité

On remarque qu'il s'exécute en $\Theta(n^2)$.

# TRI
## PAR FUSION

Le tri par fusion consiste à fusionner deux tableaux triés pour former un unique tableau trié. Il s'agit d'un algorithme "diviser-pour-régner".

- Principe:
- Etant donné un tableau tab[n]:
  - si n= 1, retourner le tableau tab;
  - Sinon:
    - ✓ Trier le sous-tableau tab[1 . . . n/2];
    - ✓ Trier le sous-tableau tab[n/2 + 1 . . . n];
    - ✓ Fusionner ces deux sous-tableaux…

# TRI
## PAR FUSION

Considérons un tableau d'entiers de n éléments à trier.

- Algorithme

[Devoir maison]

- Complexité

On remarque qu'il s'exécute en $\Theta(n \log_2 n)$ opérations.

# TRI
## RAPIDE

Le tri rapide ou encore tri de Hoare (du mon de l'inventeur) est aussi un tri basé sur le principe "diviser-pour-régner".

- Principe:
  - Il consiste à placer un élément du tableau (le pivot) à sa place définitive, en permutant tous les éléments qui lui sont inférieurs à gauche et ceux qui lui sont supérieurs à droite (le partitionnement).
  - Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement.

# TRI
## RAPIDE

Considérons un tableau d'entiers de n éléments à trier.

- Algorithme

[Devoir maison]

- Complexité

On remarque qu'il s'exécute en $\Theta(n^2)$ dans le pire des cas. Mais elle peut être en $\Theta(n \log_2 n)$ en moyenne.

# RECHERCHE D'UN ÉLÉMENT
## RECHERCHE LABORIEUSE

Soit x l'élément à rechercher dans un tableau t de n entiers.

- Principe:
  - On parcourt complétement le tableau et pour chaque élément, on teste l'égalité avec x.
  - En cas d'égalité, on mémorise la position.
- Algorithme

```
.....
indice← 0;
Pour i allant de 1 à n faire
    Si(t[i]=x) alors
        indice ← i;
    Fin Si
Fin Pour
retourner indice ;
.....
```

# RECHERCHE D'UN ÉLÉMENT
## RECHERCHE SEQUENTIELLE

Soit x l'élément à rechercher dans un tableau t de n entiers.

- Principe:
  - On parcourt séquentiellement le tableau jusqu'à trouver l'élément dans une séquence.
  - Si on arrive à la fin sans le trouver c'est qu'il n'est pas contenu dans la séquence.
- Algorithme

```
.....
Pour i allant de 1 à n faire
    Si(t[i]=x) alors
        retourner i;
    Fin Si
Fin Pour
retourner 0;
.....
```

# RECHERCHE D'UN ÉLÉMENT

## RECHERCHE DICHOTOMIQUE

Soit x l'élément à rechercher dans un tableau t **ordonné** de n entiers.

- Principe:
  - On compare l'élément à rechercher avec celui qui est au milieu du tableau.
  - Si les valeurs sont égales, la tâche est accomplie sinon on recommence dans la moitié du tableau pertinente.
- Algorithme

```
.....
bas ← 1;
haut ← taille(t);
position ← -1;
Repeter
    Si(x = t[milieu]) alors
        position ← milieu;
    Sinon  Si (t[milieu]<x) alors
        bas ← milieu + 1
    Sinon
        haut ← milieu-1
    Fin Si
jusqu'à (x = t[milieu] OU bas > haut)
retourner position
.....
```

# POINTEURS

# POINTEURS
## DÉFINITION

Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.

- Si un pointeur P contient l'adresse d'une variable N, on dit que *'P pointe sur N*'.
- Les pointeurs et les noms de variables ont presque le même rôle (à exception prés):
  - Ils donnent accès à un espace mémoire.
  - Un pointeur  peut 'pointer' sur différentes adresses tant disque le nom d'une variable reste toujours lié à la même adresse.

# POINTEURS

## DÉCLARATION, AFFECTATION & MANIPULATION

- **Déclaration**
  - Syntaxe: <pointeur>:^<type>
  - **Exemple**: Variable monPointeur: ^entier
- **Affectation**
  - Syntaxe: <pointeur> ←<adresse(variable)>
  - Ici on considère que adresse est une fonction. Elle nous renvoie l'adresse mémoire d'une variable
- **Manipulation**
  - pointeur^←valeur
  - Lire(pointeur^)
  - Ecrire(pointeur^)

# POINTEURS EN C/C++

## DÉCLARATION, AFFECTATION & MANIPULATION

- **Déclaration**
  - Syntaxe: <type> *<nom>
  - **Exemple**: int *p;
- **Affectation**
  - Syntaxe: <pointeur> = <&(variable)>
  - **Exemple**:
    - Int n=10;
    - Int *p(0);
    - p=&n;
- **Manipulation**
  - cout << "entrer une valeur";
  - cin >> *p; // écrire dans la case mémoire pointée par p
  - cout << "La valeur est : " << *p<< endl;
  - cout << "L'adresse est : " << p<< endl;

# POINTEURS EN C
## ALLOCATION DYNAMIQUE, LIBÉRATION DE LA MÉMOIRE

- Pour demander manuellement une case mémoire, on utilise l'opérateur malloc qui signifie « Memory ALLOCation ».
- malloc est une fonction ne retournant aucune valeur (void)(on n'en reviendra plus tard) :
  - void* malloc(size_t nombreOctetsNecessaires);
    - **Exemple**:
      - int* p= NULL;
      - p = malloc(sizeof(int));
- On peut libérer la ressource après usage via l'opérateur *free*
- *Free également est une fonction ne revoyant aucune valeur.*
  - *void free(void* p);*
    - **Exemple**: *free*(p);

# POINTEURS EN C/C++

## ALLOCATION DYNAMIQUE D'UN TABLEAU

- L'allocation dynamique d'un tableau est un mécanisme très utile. Elle permet de demander à créer un tableau ayant exactement la taille nécessaire (pas plus, ni moins).
- Si on veut créer un tableau de n élément de type int (par exemple), on fera appel à malloc.
  - **Exemple**:
    - int *t = NULL;
    - t = (int *)malloc(n*sizeof(int));
- Autres fonctions
  - calloc: identique à malloc mais avec initialisation des cases réservées à 0.
    - void* calloc(size_t taille, size_t nombreOctetsNecessaires);
  - realloc : permet d'agrandir une zone mémoire déjà réservée
    - void* realloc(void* tableau, size_t nombreOctetsNecessaires);
    - **Exemple**:
      - t = (int *) calloc (taille, sizeof(int));
      - taille  = taille+10;
      - t =(int *) realloc(t, taille*sizeof(int));

# POINTEURS EN C++

## ALLOCATION DYNAMIQUE, LIBÉRATION DE LA MÉMOIRE

- Pour demander manuellement une case mémoire, on utilise l'opérateur new.
- Syntaxe: <pointeur> = new type
  - **Exemple**:
    - int *p(0);
    - p = new int;
- On peut accéder à la case et modifier sa valeur
  - **Exemple**: *p = 10;
- On peut libérer la ressource après usage via l'opérateur *delete*
  - **Exemple**: *delete* p;

# POINTEURS EN C++

## ALLOCATION DYNAMIQUE D'UN TABLEAU

- Un tableau dynamique est un tableau dont le nombre de cases peut varier au cours de l'exécution du programme. Il permet d'ajuster la taille du tableau au besoin du programmeur.
- L'utilisation de delete[] permet de détruire un tableau précédemment alloué grâce à new [].
- **Tableau unidimensionnel**
  - **Exemple**:
    - int i, taille;
    - …
    - cout << « Entrez la taille du tableau: ";
    - cin >> taille;
    - int *t;
    - t = new int[taille];
    - …
    - delete[] t;

# POINTEURS EN C++

## ALLOCATION DYNAMIQUE D'UN TABLEAU

- **Tableau à deux dimensions**
  - **Exemple**:
    - `int **t;`
    - `int nColonnes;`
    - `int nLignes;`
    - `…`
    - `t = new int* [nLignes];`
    - `for (int i=0; i < nLignes; i++)`

      `t[i] = new int[nColonnes];`
    - `…`
    - `delete[] t;`

# EXERCICES D'APPLICATIONS

## Application 12 :

Ecrivez un programme déclarant une variable i de type int et une variable p de type pointeur sur int. Affichez les dix premiers nombres entiers en :

- n'incrémentant que *p
- n'affichant que i

## Application 13 :

Écrire un programme qui lit un entier n au clavier, alloue un tableau de n entiers initialisés à 0, remplir le tableau par des valeurs saisies au claviers et affiche le tableau.

## Application 14 :

Ecrire un programme qui place dans un tableau T les N premiers nombres impairs, puis qui affiche le tableau. Vous accèderez à l'élément d'indice i de t avec l'expression *(t + i).

## Application 15 :

Ecrivez un programme qui demande à l'utilisateur de saisir un nombre n et qui crée une matrice T de dimensions n*n avec un tableau de n tableaux de chacun n éléments. Nous noterons tij=0 j-ème élément du i-ème tableau. Vous initialiserez T de la sorte : pour tous i, j, tij=1 si i=j (les éléments de la diagonale) et tij=0 si i≠j (les autres éléments). Puis vous afficherez T.

# FONCTIONS, PROCÉDURES

# FONCTIONS & PROCÉDURES

## DÉFINITIONS

Lorsque l'Algorithme à écrire devient de plus en plus important (volumineux), des difficultés d'aperçu global sur son fonctionnement se posent. Il devient donc très difficile de coder et de devoir traquer les erreurs en même temps.

- Il devient utile de découper le problème en de sous problème,
- de chercher à résoudre les sous problèmes (sous-algorithmes),
- puis de faire un regroupement de ces sous-algorithmes pour reconstituer une solution au problème initial.

Un sous-algorithme est une partie d'un algorithme. Il est d'habitude déclaré dans la partie entête et est réutiliser dans le corps de l'algorithme.

- Un sous-algorithme est un algorithme. Il possède donc les même caractéristiques d'un algorithme.

# FONCTIONS & PROCÉDURES

## DÉFINITIONS

- Un sous-algorithme peut utiliser les variables déclarés dans l'algorithme. Dans ce cas, ces variables sont dites globales. Il peut également utiliser ses propres variables. Dans ce cas; les variables sont dites locales. Ces dernières ne pourront alors être utilisable qu'à l'intérieur du sous-programme et nulle part ailleurs (**notion de visibilité**). Ce qui signifie que leur allocation en mémoire sera libérer à la fin de l'exécution du sous-programme.
- Un sous-programme peut être utilisable plusieurs fois avec éventuellement des paramètres différents.
- **Un sous-algorithme peut se présenter sous forme de fonction ou de procédure:**
  - Une fonction est un sous-algorithme qui, à partir de donnée(s), calcul et rend à l'algorithme un et un seul résultat;
  - alors qu'en général, une procédure affiche le(s) résultat(s) demandé(s).

# FONCTIONS

- Syntaxe d'une fonction:

```
Fonction Nom_Fonction (Nom_Paramètre:Type_paramètre;...): type_Fonction ;
Variable
        Nom_variable : Type_variable ;        } Variables locales
        ...
Début
        ...
        Instructions ;                        }
        ...                                   } Corps de la fonction
        Nom_Fonction←resultat ;               }
Fin ;
```

Un appel de fonction est une expression d'affectation de manière à ce que le résultat soit récupéré dans une variable globale de même type: Nom_variable_globale ← Nom_Fonction (paramètres) ;

# FONCTIONS
## EXEMPLE D'APPLICATION

```
Algorithme Calcul_des_n_premiers_nombres_entiers;
Variable
        I, Som, N : entier;
Fonction Somme: entier ;
Variable
        S : entier ;
Debut /*Début de la fonction*/
        S ←0 ;
        Pour I ← 1 a N Faire
                        S ←S + I;
        FinPour ;
        Somme ← S
Fin /*Fin de la Fonction */
Debut /*Début de l'algorithme*/
        Som ← Somme ;
        Ecrire ('La somme des ', N, 'premiers nombres est', Som) ;
Fin. /*Fin de l'algorithme*/
```

# PROCÉDURES

- Syntaxe d'une procédure:

```
Fonction Nom_Procedure (Nom_Paramètre:Type_praramètre;...) ;
Variable
        Nom_variable : Type_variable ;      } Variables locales
        ...
Début
        ...
        Instructions ;                      } Corps de la fonction ...
Fin ;
```

L'appel d'une procédure peut être effectué en spécifiant, au moment souhaité, son nom et éventuellement ses paramètres; cela déclenche l'exécution des instructions de la procédure.

# PROCÉDURES
## EXEMPLE D'APPLICATION

```
Algorithme Calcul_des_n_premiers_nombres_entiers;
Variable
        I, Som, N : entier;
Procedure Somme ;
Debut /*Début de la procédure*/
        Som ←0 ;
        Pour I ← 1 a N Faire
                    Som ←Som + I;
        FinPour ;
        Ecrire ('La somme des ', N, 'premiers nombres est', Som) ;

Fin /*Fin de la Fonction */
Debut /*Début de l'algorithme*/
        Somme ;
Fin. /*Fin de l'algorithme*/
```

# MODE DE PASSAGES DE PARAMÈTRES

## PASSAGE PAR VALEUR

On distingue deux types de passage de paramètres: par valeur et par variable (dite aussi par référence ou encore par adresse).

- Le mode de passage par valeur qui est le mode par défaut, consiste à copié la valeur des paramètres effectifs dans les variables locales issues des paramètres formels de ma fonction ou de la procédure appelée.
  - Dans ce mode, nous travaillons pas directement avec la variable, mais avec une copie. Ce qui veut dire que le contenu des paramètres effectifs n'est pas modifié. À la fin de l'exécution du sous-programme, la variable conservera sa valeur initial.
  - **Syntaxe**:
    - Procedure nom_procédure (param1:type1 ; param2, param3:type2) ;
    - Fonction nom_fonction (param1:type1 ; param2:type2):Type_fonction ;

# PASSAGE PAR VALEUR
## EXEMPLE D'APPLICATION

```
Algorithme valeur_absolue_d-un_nombre_entier;
Variable
        val: entier;
Procedure Abs(nombre: entier);
Debut /*Début de la procédure*/
        Si nombre<0 Alors
                nombre ← - nombre;
        FinSi ;
        Ecrire (nombres) ;
Fin /*Fin de la Fonction */
Debut /*Début de l'algorithme*/
        Lire (val);
        Abs (val);
        Ecrire (val);
Fin. /*Fin de l'algorithme*/
```

- Ici, val reprend sa valeur initiale. Il a juste servi de données pour Abs.

# MODE DE PASSAGES DE PARAMÈTRES
## PASSAGE PAR ADRESSE

- Dans le mode de passage par variable , il s'agit pas simplement d'utiliser la valeur de la variable, mais également son emplacement mémoire.
  - Le paramètre formel se substitue au paramètre effectif tout au long de l'exécution du sous-programme et à la sortie il lui transmet sa nouvelle valeur.
  - Un tel passage se fait par l'utilisation du mot-clé **Var**.
  - **Syntaxe**:
    - Procedure nom_procédure (Var param1:type1 ; param2, param3:type2) ;
    - Fonction nom_fonction (Var param1:type1 ; param2:type2):Type_fonction ;

# PASSAGE PAR ADRESSE
## EXEMPLE D'APPLICATION

```
Algorithme valeur_absolue_d-un_nombre_entier;
Variable
        val: entier;
Procedure Abs(Var nombre: entier);
Debut /*Début de la procédure*/
        Si nombre<0 Alors
                nombre ← - nombre;
        FinSi ;
        Ecrire (nombres) ;
Fin /*Fin de la Fonction */
Debut /*Début de l'algorithme*/
        Lire (val);
        Abs (val);
        Ecrire (val);
Fin. /*Fin de l'algorithme*/
```

- Ici, val prend une nouvelle valeur.

# FONCTIONS EN C++

## RAPPEL

- Une fonction est un bloc paramétré et nommé
- Permet de découper un programme en plusieurs modules.
- Dans certains langages, on trouve *deux sortes de modules*:
  - Les fonctions, assez proches de la notion mathématique
  - Les procédures (Pascal) ou sous-programmes (Fortran, Basic) qui élargissent la notion de fonction.
- En C/C++, il n'existe qu'une seule sorte de module, nommé fonction
  - **Syntaxe:**

```
typeDeRetour nomFonction([arguments]){
        //instructions;
}
```

# FONCTIONS EN C++

## EXEMPLE

- Une fonction peut ne pas renvoyer de valeur. Dans ce cas, le type de la fonction est void. **Exemple**: void *abs*(int nombre){…}.
- Lorsqu'une fonction s'appelle elle-même, on dit qu'elle est « récursive ».

```
#include <iostream>
using namespace std;

int abs(int nombre)
{
        if (nombre<0)
            nombre=-nombre;
        return nombre; // Valeur renvoyée
}

int main()
{
         int val, valAbs;
         cout << "Entrez un nombre : ";
         cin >> val;
         valAbs = abs(val); // Appel de la fonction et affectation
         cout << "La valeur absolue de" << val << "est" << valAbs << endl;
         return 0;
}
```

# FONCTIONS EN C++

## PASSAGE PAR VALEUR

Supposons que l'on souhaite faire une permutation de deux entiers a et b. exemple:

```
#include <iostream>
using namespace std;

void permute(int a, int b)
{
        int tempon = a;
        a = b;
        b =tempon;
}

int main()
{
        int a=2, b=6;
        cout <<  "a: " << a << " b: " <<b <<endl; // avant
        permute(a, b);
        cout <<  "a: " << a << " b: " <<b <<endl; // après
        return 0;
}
```

Après exécution, on constate qu'on a pas le résultat attendu. Par défaut, le passage des arguments à une fonction se fait par valeur. Pour remédier à cela, il faut passer par adresse ou par référence.

# FONCTIONS EN C++

## PASSAGE PAR ADRESSES

Pour modifier le paramètre réel, on passe son adresse plutôt que sa valeur. exemple:

```
#include <iostream>
using namespace std;

void permute(int *a, int *b)
{
        int tempon = *a;
        *a = *b;
        *b =tempon;
}

int main()
{
        int a=2, b=6;
        cout <<  "a: " << a << " b: " <<b <<endl; // avant
        permute(&a, &b);
        cout <<  "a: " << a << " b: " <<b <<endl; // après
        return 0;
}
```

# FONCTIONS EN C++

## PASSAGE PAR RÉFÉRENCE

On peut également faire ceci:

```
#include <iostream>
using namespace std;

void permute(int& a, int& b)
{
        int tempon = a;
        a = b;
        b =tempon;
}

int main()
{
        int a=2, b=6;
        cout <<  "a: " << a << " b: " <<b <<endl; // avant
        permute(a, b);
        cout <<  "a: " << a << " b: " <<b <<endl; // après
        return 0;
}
```

Ici, le compilateur se charge de la gestion des adresses: le paramètre formel est un alias de l'emplacement mémoire du paramètre réel.

# FONCTIONS EN C++

## PASSAGE D'UN TABLEAU

On peut faire passer un tableau en paramètre. Nous avons dans ce cas, deux cas de figure: par pointeur ou par semi-référence.

**Par pointeur**:

```
#include <iostream>
using namespace std;

void affiche(int *tableau, int taille)
{
        for(int i=0; i<taille; i++)
            cout << tableau[i] << "   " << endl;
}

int main()
{
        int tab[5] = {1, 2, 3, 4, 5 };
        affiche(tab, 5);
        return 0;
}
```

# FONCTIONS EN C++

## PASSAGE D'UN TABLEAU

**Par semi-référence**:

```
#include <iostream>
using namespace std;

void affiche(int tableau[], int taille)
{
        for(int i=0; i<taille; i++)
            cout << tableau[i] << "   " << endl;
}

int main()
{
        int tab[5] = {1, 2, 3, 4, 5 };
        affiche(tab, 5);
        return 0;
}
```

# RÉCURSIVITÉ

# RÉCURSIVITÉ
## DÉFINITIONS

On appelle récursivité tout sous-programme qui s'appelle dans son traitement.

- Il est impératif de prévoir une condition d'arrêt puisque le sous-programme va s'appeler récursivement. sinon, il ne s'arrêtera jamais.
  - On teste la condition,
  - Si elle n'est pas vérifié, on lance à nouveau le sous-programme.

# RÉCURSIVITÉ
## EXEMPLE

Algorithme

```
fonction factorielle (n : entier) : entier
Début
   Si(n<2) alors
      retourner 1
   Sinon
      retourner n*factorielle(n-1)
   Fin si
Fin
```

C++

```
int factoriel(int n)
{
        if(n<=1)
                return 1;
        else
                return(n*factoriel(n-1));
}
```

# RÉCURSIVITÉ CROISÉE

Il également possible qu'un sous-programme appel un second qui a son tour appel le premier. On dit que la récursivité est indirecte, cachée, croisée ou mutuelle.

Algorithme

```
fonction pair (n : entier) : booleén
Début
   Si(n=0) alors
      retourner VRAI
   Sinon Si(n=1) alors
      retourner FAUX
   Sinon
      retourner impair(n-1)
   Fin si
Fin
```

```
fonction impair (n : entier) : booleén
Début
   Si(n=1) alors
      retourner VRAI
   Sinon Si(n=0) alors
      retourner FAUX
   Sinon
      retourner pair(n-1)
   Fin si
Fin
```

# FICHIERS

# FICHIERS
## DÉFINITION

Jusque-là, nous ne savons que lire et écrire sur une console. Dans cette section, nous allons apprendre à interagir avec les fichiers.

- Par définition, un fichier est une suite d'informations stocker sur un périphérique (disque dur, clé USB, CDROM, etc. …).
- On peut accéder à un fichier soit en lecture seule, soit en écriture seule ou soit enfin en lecture/écriture.
- Pour pouvoir manipuler les fichiers en C++, il va falloir inclure la bibliothèque fstream (#include <fstream>) qui signifie "file stream" ou "flux vers les fichiers" en français.
- **Il existe 2 types de fichiers**:
  - les fichiers textes qui contiennent des informations sous la forme de caractères. Ils sont lisibles par un simple éditeur de texte.
  - les fichiers binaires dont les données correspondent en général à une copie bit à bit du contenu de la RAM. Ils ne sont pas lisibles avec un éditeur de texte.

# FICHIERS

## MANIPULATION DES FICHIERS TEXTES

lorsqu'on inclut fstream, il ne faut pas inclure iostream car ce fichier est déjà inclut dans fstream.

- Lecture d'un fichier texte:
  - Pour ouvrir un fichier en lecture, la syntaxe est la suivante:

    **ifstream** *nom_fichier* ("chemin_vers_le_fichier");
  - Pour savoir si le fichier a bien été ouvert en lecture, la méthode is_open() est utilisée. Elle renvoie **true** si le fichier est effectivement ouvert.

    Ex: nom_fichier.is_open();
  - Pour fermer le fichier, on fait:

    *nom_fichier*.close();
  - Pour tester si on est arrivé à la fin du fichier, on fait:

    nom_fichier.eof();
  - La lecture dans un fichier se fait par:

    nom_fichier >> variable1 [>> variable2>> …];

❖ Ici, l'espace et le saut de ligne sont des séparateurs

# FICHIERS

## MANIPULATION DES FICHIERS TEXTES

```
# include <fstream>
# include <string>

int main(void)
{
	string nom;
	string prenom;
	string tel;
	ifstream f ("data.txt"); // ouverture du fichier en lecture

	f >> nom >> prenom >> tel;

	while(!f.eof()) // tant qu'on n'est pas arrivé à la fin du fichier
	{
		cout << nom << " \t"<< prenom << " \t"<< tel << "\n"; // on affiche
		f >> nom >> prenom >> tel; // on lit les informations suivantes
	}

	f.close();

	return 0;
}
```

# FICHIERS

## MANIPULATION DES FICHIERS TEXTES

- Ecrire dans un fichier texte:
  - La création d'un nouveau fichier ou l'écriture dans un fichier existant se fait comme suit:

**ofstream** *nom_fichier* ("chemin_vers_le_fichier");

  - L'écriture dans un fichier se fait par:

```
nom_fichier <<"toto"<<" "<<"pathe"<<"\n"<<"sidi"<<" "<<"rama";
```

    - Il va falloir écrire le séparateur soi-même.

# FICHIERS

## MANIPULATION DES FICHIERS TEXTES

```
# include <fstream.h>

using namespace std;

int main (void)
{
	string nom;
	String prenom;
	string tel;
	ofstream f ("data.txt"); // ouverture du répertoire en écriture

	for(int i=0; i<10; i++)
	{
		cout << "\n p"<< i+1 << ":\n"
		cout << "nom:";
		cin >> nom;
		f <<nom << " ";
		cout << "\n prenom:";
		cin >> prenom;
		f <<prenom<< " ";
		cout << "\n tel:";
		cin >> tel;
		f << tel << "\n";
	}

	f.close();

	return 0;
}
```

# STRUCTURES

# STRUCTURES

- Plus haut, nous avons vu les tableaux qui sont une sorte de regroupement de données de même type. Il serait aussi intéressant de regrouper des données de types différents dans une même entité.
- Nous allons donc créer un nouveaux type de données (plus complexes) à partir des types que nous connaissons déjà: les structures.

Une structure permet donc de rassembler sous un même nom, des informations de type différent. Elle peut contenir des données entières, flottantes, tableaux, caractères, pointeurs, structure, etc… . Ces données sont appelés les membres de la structure.

**Exemple**: la carte d'identité d'une personne: (nom, prenom, date_de_naissance, lieu_de_naissance, quartier, etc… ).

# STRUCTURES
## DÉCLARATION

Pour déclarer une structure, on utilise le mon clé struct: syntaxe

```
struct nomStructure{
        type_1 nomMembre1 ;
        type_2 nomMembre2 ;
        …
        type_n nomMembren ;
        }
```

**Exemple**;

```
struct Personne
{
   int age;
   double poids;
   double taille;
};
```

Une fois la structure déclarée, on pourra définir des variables de type structuré.

**Exemple**:

```
Personne massamba, mademba;
```

- Massamba pourra accéder à son age en faisant massamba.age.

# STRUCTURES
## DÉCLARATION

**Un exemple complet**:

```
#include<iostream>
using namespace std;

struct Personne
{
    int age;
    double poids;
    double taille;
};
int main()
{
      Personne massamba;
      massamba.age=25;
      massamba.poids=90,5;
      massamba.taille=185,7;
      cout << '' Massamba a '' << massamba.age << '' ans, il pèse '' << massamba.poids << '' kg et il fait '' << massamba.taille << '' cm de long .'' << endl;
      return 0;
}
```

# STRUCTURES
## DÉCLARATION

Il est possible de déclarer une structure dans un fichier d'entête et puis l'inclure dans le fichier du programme principal.

Exemple:

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
        int age;
        double poids;
        double taille;
};

#endif
```

```
#include<iostream>
#include''person.hh''
using namespace std;
Int main()
{
        Personne massamba;
        massamba.age=25;
        massamba.poids=90,5;
        massamba.taille=185,7;

        cout << '' Massamba a ''
        << massamba.age << '' ans, il pèse ''
        << massamba.poids << '' kg et il fait ''
        << massamba.taille << '' cm de long .''
        << endl;

        return 0;
}
```

# STRUCTURES
## INITIALISATION

Dans l'exemple précèdent, nous avons attribué une valeur champ après champ. Ce qui peut s'avérer long et peu pratique.

Il est en fait possible d'initialiser les champs d'une structure au moment de son instanciation grâce à l'opérateur {}.

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
        int age;
        double poids;
        double taille;
};

#endif
```

```
#include<iostream>
#include''person.hh''
using namespace std;
Int main()
{
        Personne massamba={25, 90.5, 185.7);

        cout << '' Massamba a ''
        << massamba.age << '' ans, il pèse ''
        << massamba.poids << '' kg et il fait ''
        << massamba.taille << '' cm de long .''
        << endl;

        return 0;
}
```

# STRUCTURES
## STRUCTURES ET TABLEAU

Une structure peut contenir un tableau. De ce fait, un espace mémoire lui sera réservé à sa création.

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
        char nom[20];
        int age;
        double poids;
        double taille;
};

#endif
```

```
#include<iostream>
#include''person.hh''
using namespace std;
Int main()
{
    Personne m={"Massamba", 25, 90.5, 185.7);

    cout << m.nom <<''  a ''
    << m.age << '' ans, il pèse ''
    << m.poids << '' kg et il fait ''
    << m.taille << '' cm de long .''
    << endl;

    return 0;
}
```

# STRUCTURES
## TABLEAUX DE STRUCTURES

Il est possible de créer des tableaux contenant des instances d'une même structure.

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
        char nom[20];
        int age;
        double poids;
        double taille;
};

#endif
```

```
#include<iostream>
#include''person.hh''
using namespace std;
Int main()
{
    Personne m[2]={
        {"Massamba", 25, 90.5, 185.7),
        {"Mademba", 30, 73.5, 175.3)
    };

    cout << m[1].nom <<''  a ''
    << m[1].age << '' ans, il pèse ''
    << m[1].poids << '' kg et il fait ''
    << m[1].taille << '' cm de long .''
    << endl;

    return 0;
}
```

# STRUCTURES
## STRUCTURES IMBRIQUÉES

Une structure peut être membre d'une autre structure. On peut donc utiliser ce nouveau type comme champ d'une autre structure comme dans l'exemple suivant:

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct date {
        int jour;
        int mois;
        double annee;
};
struct Personne
{
        char nom[20];
        struct date date_de_naissance;
        double poids;
        double taille;
};

#endif
```

```
#include<iostream>
#include''person.hh''
using namespace std;
Int main()
{
    Personne m[2]={
        {"Massamba", {8,8,2008}, 25.0, 185.7),
        {"Mafatou", {5,5,2010}, 30.6, 175.3)
    };

    cout << m[0].nom <<'' est né en''
    << m[0].date_de_naissance.annee << '', il
    pèse ''
    << m[0].poids << '' kg et il fait ''
    << m[0].taille << '' cm de long .''
    << endl;

    return 0;
}
```

# STRUCTURES
## STRUCTURES ET FONCTIONS

- Une structure peut être passer à une fonction:

```
#include<iostream>

using namespace std;

struct date
{
		int jour;
		int mois;
		double annee;
};
struct Personne
{
		char nom[20];
		struct date date_de_naissance;
		double poids;
		double taille;
};

// la suite →
```

```
void saisirUser(Personne &p)
{
		cout << "Tapez le nom : ";
		cin >> p.nom;
		// ...
		cout << "Tapez la taille: ";
		cin >> p.taille;
}

int main()
{
	Personne p;

	cout << "SAISIE DE P" << endl;
	saisirUser(p);

	cout << p.nom <<'' est né en''
	<< p.date_de_naissance.annee<< '', il pèse ''
	<< p.poids << '' kg et il fait ''
	<< p.taille << '' cm de long .''
	<< endl;

	return 0;
}
```

# STRUCTURES
## FONCTIONS MEMBRES

- On peut ajouter une fonction dans une structure:

```
#include<iostream>
#include<cmath>
using namespace std;

struct date {
		int jour;
		int mois;
		double annee;
};
struct Personne
{
		char nom[20];
		struct date  date_de_naissance;
		double poids;
		double taille;
		double inMas(double p, double t);
};
double Personne::inMas(double p, double t)
{
		return p/pow(t;2);
}

// la suite →
```

```
void saisirUser(Personne &p)
{
		cout << "Tapez le nom : ";
		cin >> p.nom;
		// …
		cout << "Tapez la taille: ";
		cin >> p.taille;
}

int main()
{
	Personne p;

	cout << "SAISIE DE P" << endl;
	saisirUser(p);

	cout << p.nom <<'' est né en''
	<< p.date_de_naissance.annee<< '', il pèse ''
	<< p.poids << '' kg, il fait ''
	<< p.taille << '' cm de long et son IMC est de :''
	<< p.inMas(p.poids, p.taille)
	<< endl;

	return 0;
}
```

# LISTES CHAINÉES, PILES, FILES

# LISTES CHAÎNÉES

- Lorsqu'une structure contient une donnée avec un pointeur vers un élément de même composition, on parle alors de liste chainée.
  - Les listes chainées sont basées sur les pointeurs et sur les structures;
  - Quand une variable pointeur ne pointe sur aucun emplacement, elle doit contenir la valeur ***Nil*** - Not In List (qui est une adresse négative).

Par définition, une liste chaînée est une structure linéaire qui n'a pas de dimension fixée lors de sa création.

- Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs;
- Chaque élément (dit ***nœud***) est lié à son successeur. Chaque prédécesseur contient le pointeur du successeur;
- Le dernier élément de la liste ne pointe sur rien (***Nil***);
- La liste est uniquement accessible via sa tête de liste qui est son premier élément.

# LISTES CHAÎNÉES
## EXEMPLES


- Tête est le pointeur contenant l'adresse du premier élément alors que chaque nœud est une structure avec une case contenant la valeur à manipuler (20, 3, -11, 0 et 6) et une case contenant l'adresse de l'élément suivant;
- Contrairement au tableau, les éléments n'ont aucune raison d'être voisins ni ordonnés en mémoire;
- Selon la mémoire disponible, il est possible de rallonger ou de raccourcir une liste;
- Pour accéder à un élément de la liste il faut toujours débuter la lecture de la liste par son premier élément dans le pointeur duquel est indiqué la position du deuxième élément. Dans le pointeur du deuxième élément de la liste on trouve la position du troisième élément. Ainsi de suite jusqu'à obtenir la position de l'élément… ;
- Pour ajouter, supprimer ou déplacer un élément il suffit d'allouer une place en mémoire et de mettre à jour les pointeurs des éléments.

# LISTES CHAÎNÉES
## DIFFÉRENTS TYPES

Il existe différents types de listes chaînées :

- *Liste chaînée simple* constituée d'éléments reliés entre eux par des pointeurs;
- *Liste doublement chaînée* où chaque élément dispose de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. *Ceci permet donc la lecture dans les deux sens;*
- *Liste circulaire* où le dernier élément pointe sur le premier élément de la liste.

# LISTES CHAÎNÉES

## EXEMPLES: INSERTION/SUPPRESSION PAR L'AVANT

```
#include<iostream>
using namespace std;

int main(){
        int pos_noeud, num_noeud;
        typedef struct noeud
        {
                int data;       // pour stocker l'information
                noeud *suivant; // reference au noeud suivant
        };
        // insertion par l'avant
        noeud *tete = NULL;
        // premier noeud
        noeud *noeud1 = new noeud;;
        noeud1->data=10;
        noeud1->suivant=tete;
        tete = noeud1;
        // deuxiéme noeud
         noeud *noeud2 = new noeud;
        noeud2->data=20;
        noeud2->suivant=tete;
        tete = noeud2;
        // Affichage
        cout<<"TETE -> ";
         while(tete!=NULL)
        {
                cout<< tete->data <<" -> ";
                 tete = tete->suivant;
        }
        cout<<"NULL";

         return 0;
}
```

```
// suppression par l'avant
noeud *cellule=new  noeud;
cellule=tete;
tete=cellule->suivant;
delete cellule;
```

# LISTES CHAÎNÉES

## EXEMPLES: INSERTION À UNE POSITION SPÉCIFIQUE

```
// ……
cout<<"Entrer la position du noeud: ";
cin>>pos_noeud;
noeud *curseur=new noeud;
curseur->suivant=tete;
for(int i=1;i<pos_noeud;i++){
      curseur=curseur->suivant;
      if(curseur==NULL){
                  cout<<"La position"<<pos_noeud<<" n'est pas dans la liste"<< endl;
                  break;
      }
}
noeud *nouveau=new noeud;
nouveau->data=30;
nouveau->suivant=curseur->suivant;
curseur->suivant=nouveau;
// …
```

# LISTES CHAÎNÉES

## EXEMPLES: INSERTION/SUPPRESSION PAR L'ARRIÈRE

```
#include<iostream>
using namespace std;

int main(){;
        typedef struct noeud
        {
                int data;       // pour stocker l'information
                noeud *suivant; // reference au noeud suivant
        };
        // insertion par l'arrière
        noeud *tete = NULL;
        // premier noeud
        noeud *noeud1 = new noeud;;
        tete = noeud1;
        noeud1->data=10;
        noeud1->suivant=NULL;

        // deuxiéme noeud
        noeud *noeud2 = new noeud;
        noeud1->suivant = noeud2
        noeud2->data=20;
        noeud2->suivant=NULL;

        // Affichage
        cout<<"TETE -> ";
         while(tete!=NULL)
        {
                cout<< tete->data <<" -> ";
                 tete = tete->suivant;
        }
        cout<<"NULL";

         return 0;
}
```

```
// suppression par  l'arrière
noeud *cellule=new  noeud;
cellule=tete;
noeud *encien=new  noeud;
while(cellule->suivant!=NULL)
{
        encien=cellule;
        cellule=cellule->suivant;
}
encien->suivant=NULL;
delete cellule;
```

# PILES, FILES

Les piles et les ***files*** sont des *listes chaînées particulières* permettant d'ajouter et de supprimer des éléments uniquement à une des deux extrémités de la liste.

- Une structure **pile** est assimilable à une superposition d'assiettes . on pose et on prend à partir du sommet de la pile. C'est du principe LIFO (Last In First Out);
- Une structure **file** est assimilable à une file d'attente de caisse. le premier client entré dans la file est le premier à y sortir. C'est du principe FIFO (First In First Out).

# PILES

une Pile est donc un ensemble de valeurs ne permettant des insertions ou des suppressions qu'a une seule extrémité, le sommet.

- l'opération insertion d'un objet sur une pile consiste à empiler cet objet au sommet de celle-ci. Exemple: ajouter une nouvelle assiette au dessus de celle qui se trouve au sommet.
- l'opération suppression d'un objet sur une pile consiste à dépiler celui-ci au sommet de celle-ci. Exemple: supprimer ou retirer l'assiette qui se trouve au sommet.

Une pile sert essentiellement à stocker des données ne pouvant pas être traitées immédiatement.

# PILES

une Pile est un enregistrement avec une variable sommet indiquant le sommet de la pile et une structure données pouvant enregistrer les données. La manipulation d'une pile en C++ nécessite d'inclure la bibliothèque stack. Dans cette bibliothèque nous trouvons les fonctions pour:

- La déclaration. syntaxe: stack<type> pile;
- Connaitre la taille de la pile (qui nous renvoie le nombre d'élément): pile.size();
- Vérifier si la pile est vide ou non: pile.empty();
- Ajouter une nouvelle valeur à la pile(empiler): pile.push(element);
- Accéder au premier élement de la pile: pile.top();
- Supprimer la valeur se trouvant au sommet de la pile(depiler): pile.pop(); // ici, la pile ne doit pas être vide !

# PILES

## EXEMPLE

```
#include<iostream>
#include<stack>
using namespace std;
int main(){
		int n;
		stack<int> pile;
		// remplissage
		cout << "veuillez saisir un element: ";
		cin >> n;
		while(n>0){
				pile.push(n);
				cout << "entrer un autre element: ";
				cin >> n;
		}
		// affichage
		cout << endl;
		if(pile.size()==0){
				cout <<"la pile est vide ";
		}else if(pile.size()==1){
				cout<<"la pile contient un element qui est: "<<pile.top();
		}else{
				cout <<"la pile contient " << pile.size() << " elements que sont :" << endl;
				while(!pile.empty()){
						cout << pile.top() << " ";
						pile.pop();
				}
		}

		return 0;
}
```

# PILES

## PILE ET FONCTION

Passer une pile en paramètre à un sous-programme se fait par références.
***Exemple***: remplissage(stack<int>& pile) , affichage(stack<int>& pile);

```
...
void remplissage(stack<int>& pile)
{
		cout << "veuillez saisir un element: ";
		cin >> n;
		while(n>0){
				pile.push(n);
				cout << "entrer un autre element: ";
				cin >> n;
		}
}
------
void affichage(stack<int>& pile)
{
		cout << endl;
		if(pile.size()==0){
				cout <<"la pile est vide ";
		}else if(pile.size()==1){
				cout<<"la pile contient un element qui est: "<<pile.top();
		}else{
				cout <<"la pile contient " << pile.size() << " elements que sont :" << endl;
				while(!pile.empty()){
						cout << pile.top() << " ";
						pile.pop();
				}
		}
}
```

# FILES

une file est un ensemble de valeurs ne permettant des insertions à la fin et des suppressions en début de file.

- l'opération insertion d'un objet sur une file consiste à empiler cet objet à la fin de celle-ci. Exemple: dans la file d'attente, si une nouvelle personne arrive, elle sera mise derrière la dernière personne arrivée.
- l'opération suppression d'un objet sur une file consiste à dépiler celui-ci en début de celle-ci. Exemple: dans une file d'attente, seule la première personne peut être servie en premier.

Une file sert essentiellement à stocker des données qui doivent être traitées selon leur ordre d'arrivée.

# FILES

une File est donc un enregistrement avec une variable ***Début*** indiquant le premier élément, ***Queue*** indiquant le dernier élément et une structure données pouvant enregistrer les données. La manipulation d'une file en C++ nécessite d'inclure la bibliothèque queue. Dans cette bibliothèque nous trouvons les fonctions pour:

- La déclaration. syntaxe: queue<type> file;
- Connaitre la taille de la file (qui nous renvoie le nombre d'élément): file.size();
- Vérifier si la file est vide ou non: file.empty();
- Ajouter une nouvelle valeur à la pile(empiler): file.push(element);
- Accéder au premier élément de la file: file.front();
- Accéder au dernier élément de la file: file.back();
- Supprimer le premier élément de la file(depiler): file.pop(); // ici, la file ne doit pas être vide !

# FILES

## EXEMPLE

```
#include<iostream>
#include<queue>
using namespace std;
int main(){
		int n;
		queue<int> file;
		// remplissage
		cout << "veuillez saisir un element: ";
		cin >> n;
		while(n>0){
				file.push(n);
				cout << "entrer un autre element: ";
				cin >> n;
		}
		// affichage
		if(file.size()==0){
				cout <<"la file est vide ";
		}else if(file.size()==1){
				cout<<"la file contient un element qui est: "<<file. front();
		}else{
				cout <<"la file contient " << file.size() << endl;
				cout << "Le premier élément est : " << file.front() << endl;
				cout << "Le dernier élément est : " << file.back() << endl;
				cout << « Les elements sont :" << endl;
				while(!file.empty()){
						cout << file.front() << " ";
						file.pop();
				}
		}

		return 0;
}
```

# ARBRES
# (RECHERCHE ET EXPOSÉ)

# PROGRAMMATION ORIENTÉE OBJETS

---

# POO

# LA POO

Comme introduit en début de ce cours, le C++ est un langage à la fois procédural et orienté objet contrairement au C qui est seulement procédural.

Un langage est dit procédural s'il permet seulement la définition des données grâce à des variables, et des traitements grâce aux fonctions. il sépare données et traitements sur ces données.

Un langage est dit orienté objet lorsqu'il offre un mécanisme de classe rassemblant données et traitements. Le paradigme de programmation orientée objet implique une méthode différente pour concevoir et développer des applications.

Dans cette deuxième partie, nous parlerons de Classe, d'Objet, de Méthode, d'Encapsulation, de Constructeur/Destructeur, de Surcharge, d'Héritage et de Polymorphisme.

# LA POO
## CLASSE

Une classe est un type de données dont le rôle est de rassembler sous un même nom à la fois données et traitements. En C++, la programmation d'une classe se fait en trois phases: déclaration, définition et utilisation.

- **Déclaration**: c'est la partie interface de la classe. Elle se fait dans un fichier dont le nom se termine par .h ou .hpp, .H, ou .h++ (appelé fichier d'entête).
  - La syntaxe est la suivante:

```
class Nom_de_la_classe {
        public:
        // déclarations des données et fonctions-membres publiques
        private:
        // déclarations des données et fonctions-membres privées
    };
```

  - Exemple:

```
// Personne.hpp
class Personne {
        public:
        Personne( string prenom, string nom );
        void toString();
        private:
        string p_nom;
        string p_prenom;
    };
```

# LA POO
## CLASSE

- **Définition**: c'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient les définitions des fonctions-membres de la classe.
  - Exemple: Personne.cpp

```
#include <iostream.h>
#include "Personne.hpp"

// constructeur (Il permet d'initialiser une nouvelle personne)
Personne::Personne(string prenom, string nom){
        p_prenom = prenom;
        p_nom = nom;
}

// une méthode (fonctions-membres)
void Personne::toString(){
        cout <<"Je m'appel"<<p_prenom<<" "<<p_nom<<endl;
}
```

# LA POO
## CLASSE

- **Utilisation**: Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient le traitement principal (la fonction ***main***).
  - Exemple: test.cpp

```
#include <iostream.h>
#include "Personne.hpp"

// traitement principal
void main(){
        //  appel implicite du constructeur
        Personne p("Nabi" , "Barro");

        p.toString();
}
```

# LA POO
## OBJET

Un objet est une instance d'une classe (c'est-à-dire, une variable dont le type est une classe). Un objet occupe donc de l'espace mémoire. Il peut être alloué :

- ***statiquement***: dans ce cas, on met le nom de la classe suivi du nom de l'objet et éventuellement suivi par des arguments d'appel donnés à un constructeur de la classe.

  **Exemple**: Personne p("Nabi" , "Barro");

- ou ***dynamiquement***: dans ce cas, un pointeur sur la zone mémoire ou l'objet a été alloué est retourné. Lorsqu'on n'en a plus besoin, on le libère avec l'opérateur *delete*.

  **Exemple**:

```
Personne * p = new Personne("Nabi" , "Barro");
p->toString();
delete p;
```

# LA POO

## VISIBILITÉ DES MEMBRES D'UNE CLASSE

La visibilité des attributs et méthodes d'une classe est définie dans l'interface de la classe grâce aux mots-clés public, private ou protected qui permettent de préciser leurs types accès.

- ***public***: autorise l'accès pour tous. Exemple: le constructeur ainsi que la méthode *toString()* de l'exemple précèdent peuvent être utilisés partout sur une instance de Personne.
- ***private***: restreint l'accès aux seuls corps des méthodes de cette classe. Sur l'exemple précédent, les attributs privés *p_prenom* et *p_nom* ne sont accessibles sur une instance de Personne que dans les corps des méthodes de la classe. Ainsi, la méthode *toString()* a le droit d'accès sur ses attributs *p_prenom* et *p_nom*. Elle aurait aussi l'accès aux attributs *p_prenom* et *p_nom* d'une autre instance de Personne.
- *protected*: comme *private* sauf que l'accès est aussi autorisé aux corps des méthodes des classes qui héritent de cette classe.

# LA POO
## CONSTRUCTEURS ET DESTRUCTEURS

- Un ***constructeur*** est une fonction-membre déclarée du même nom que la classe, et sans type : *Nom_de_la_classe(<paramètres>);*
- ***Fonctionnement:*** à l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.

  **Exemple**: Personne p("Nabi" , "Barro");
- Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils différent par le nombre ou le type des paramètres. Un constructeur sans paramètre s'appelle ***constructeur par défaut***.
- Un ***destructeur*** est une fonction-membre déclarée du même nom que la classe mais précédé d'un tilde (~) et sans type ni paramètre : ~Nom_de_la_ classe();
- ***Fonctionnement***: à l'issue de l'exécution d'un bloc, le destructeur est automatiquement appelé pour chaque objet de la classe déclaré dans ce bloc. Cela permet par exemple de programmer la restitution d'un environnement, en libérant un espace mémoire alloué par l'objet.

# LA POO
## HÉRITAGE

Le principe est d'utiliser la déclaration d'une classe (appelée classe de base ou classe parente) comme base pour déclarer une seconde classe(appelée classe dérivée). La classe dérivée héritera de tous les membres (données et fonctions) de la classe parente.

- Exemple de déclaration de la classe parente:

```
class ClasseParente {
        public:
        void fonctionParente(string stringParente);
        protected:
        int valeurParente;
};
```

- Exemple de déclaration de la classe dérivée de la classe parente:

```
// héritage public
class ClasseDerivee : public ClasseParente {
        public:
        void fonctionDerivee(string stringDerivee);
        protected:
        int valeurDerivee;
};
```

Un objet de la classe Dérivée possède alors ses propres données et fonctions-membres, plus les données-membres et fonctions-membres héritées de la classe parente:

# LA POO

## HÉRITAGE - CONSTRUCTEURS ET DESTRUCTEURS

Quand un objet est créé, si cet objet appartient à une classe dérivée, le constructeur de la classe parente est d'abord appelé. Quand un objet est détruit, si cet objet appartient à une classe dérivée, le destructeur de la classe parente est appelé après.

**Voici un exemple**:

```
#include<iostream.h>
class GrandPere
{       // données membres
        public:
                GrandPere(void);
                ~GrandPere();
};
class Pere : public GrandPere
{       // données membres
        public:
                Pere(void);
                ~Pere();
};
class Fils : public Pere
{       // données membres
        public:
                Fils(void);
                ~Fils();
};
```

```
void main()
{
        Fils *junior;
        junior = new Fils;
        // appels successifs des
        constructeurs de GrandPere,
        Pere et Fils
        ........
        delete junior;
        // appels successifs des
        destructeurs de Fils, Pere et
        GrandPere
}
```

# LA POO

## HÉRITAGE - CONSTRUCTEURS ET DESTRUCTEURS

**Exemple**:

```
#include<iostream.h>
class  Rectangle{
        public:
                Rectangle(int lo, int la);
                void toString();
        protected:
                int longueur, largeur;
};
Rectangle::Rectangle(int lo, int la){
        longueur = lo;
        largeur = la;
}
void Rectangle::toString(){
        cout <<"surface= "<<longueur*largeur<<endl;
}
        -------
class Carre : public Rectangle {
        public:
                Carre(int cote);
};
Carre::Carre(int cote) : Rectangle(cote, cote) {
}
```

```
void main()
{
        Carre *monCarre;
        Rectangle *monRectangle;
        monCarre = new Carre(5);
        monCarre->toString();
        // affiche 25
        monRectangle = new Rectangle(5, 10);
        monRectangle->toString();
        // affiche 50
}
```

# LA POO
## POLYMORPHISME, HÉRITAGE MULTIPLE ET CLASSES ABSTRAITES

Dans l'exemple précèdent, la méthode toString() de Carre réutilise celle qui se trouve dans Rectangle. *On peut aussi avoir une version différente pour chacune des classes dérivée*.

Supposons que l'on souhaite creer une collection d'objets de type Rectangle, en demandant toString() pour chacune de ces formes. Ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : *on dit que toString() est polymorphe*. Ce choix de la version adéquate de toString() sera réalisé au moment de l'exécution.

Toute fonction-membre de la classe parente devant être redéfinie dans une classe dérivée doit être précédée du mot *virtual*.

Exemple:

```
#include<iostream.h>
class Rectangle{
        public:
            virtual void toString(); // fonction destinée à être surchargée
};
void Rectangle::toString(){
        cout << « Je suis un rectangle !" <<endl;
}
```

# LA POO

## POLYMORPHISME, HÉRITAGE MULTIPLE ET CLASSES ABSTRAITES

Exemple: suite …

```
class Carre : public Rectangle{
        public:
            void toString();
};

void Carre::toString(){
        cout <<"Je suis un carré!"<<endl;
}

void main()
{
      Carre *monCarre;
      Rectangle *monRectangle;
      monCarre = new Carre;
      monCarre->toString();
      monRectangle = new Rectangle;
      monRectangle->toString();
}
```

# LA POO
## POLYMORPHISME, HÉRITAGE MULTIPLE ET CLASSES ABSTRAITES

Il est possible de faire dériver une classe de plusieurs autres classes simultanément (héritage multiple). Exemple: une classe C peut hériter de la classe A et de la classe B. une instance de la classe C possèdera alors à la fois les données et fonctions-membres de la classe A et celles de la classe B.

***Exemple***:

```
class A {
.....
};
```

```
class B {
.....
};
```

```
class C : public A, public B{
.....
};
```

- À la création de C, les constructeurs des classes parentes sont appelés: celui de A, puis celui de B.
- À la destruction, les destructeurs des classes parentes sont appelés, celui de B, puis celui de A.
- il peut arriver que des données ou fonctions-membres des classes A et B aient le même nom. Dans ce cas, il faut utiliser l'opérateur de porté comme suit: A::var et B::var ;

# LA POO
## POLYMORPHISME, HÉRITAGE MULTIPLE ET CLASSES ABSTRAITES

- Une fonction-membre virtuelle d'une classe est dite purement virtuelle lorsque sa déclaration est suivie de = 0. Exemple:

```
class A {
    public:
    virtual void fonct() = 0;
};
```

- Une fonction purement virtuelle n'a pas de définition dans la classe. Elle ne peut qu'être surchargée dans les classes dérivées.

- Une classe comportant au moins une fonction-membre purement virtuelle est appelée classe abstraite.
- Aucune instance d'une classe abstraite ne peut être créée.
- L'intérêt d'une classe abstraite est uniquement de servir de "canevas" à ses classes dérivées, en déclarant l'interface minimale commune à tous ses descendants.

# FIN