# INFOH2001 Object Oriented Programming

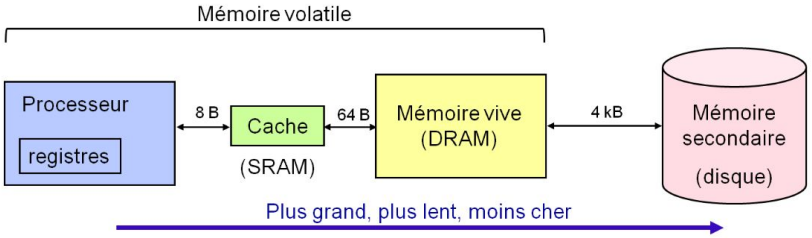
Mahmoud SAKR <mahmoud.sakr@ulb.be>

École polytechnique de Bruxelles

2025/26

Lecture 3

# Hiérarchie Mémoire




| Attribut | Registres | Cache (L1/L2/L3) | DRAM (DDR5) | Disque (NVMe SSD) |
|---|---|---|---|---|
| **Taille** | ~ 0,5 KB par cœur | ~L1: 32–64 KB<br>L2: 0,5–2 MB<br>L3: 16–128 MB | 16–64 Go (système typique) jusqu'à plusieurs To (serveurs) | ~0,5–4 To (grand public) jusqu'à 100+ To (entreprise) |
| **Temps d'accè** | 0,1–0,3 ns | L1 : 0,3–1,2 ns<br>L2 : 2–5 ns<br>L3 : 10–20 ns | 40–60 ns (latence CAS DDR5) | 10–100 µs (0,01–0,1 ms) |
| **Coût** | Très élevé (intégré au CPU) | Très élevé (intégré au CPU) | ~ 12–16 $/GB (fin 2025) | ~ 0,05–0,10 $/GB |
| **Unité de transfert** | 8 bytes (64 bits) | 64 bytes (ligne de cache standard) | 64 bytes (ligne de cache) | 4–16 kB (page/bloc SSD) |



ULB

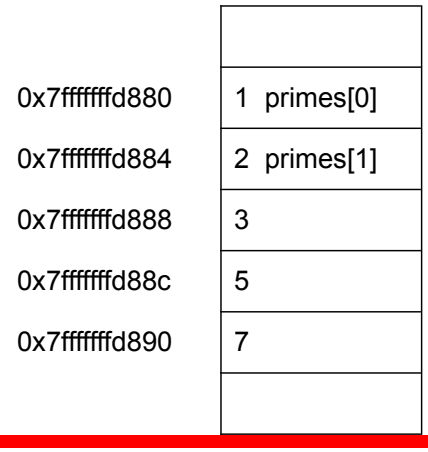
Comme les notes contiennent des figures et des tableaux, elles n'ont pas pu être intégrées en tant que notes de diapositives. Veuillez les consulter dans le fichier externe: https://drive.google.com/file/d/1QDYOSqsreEgbaItMHIKJcO_Kmo2YmvWY/view?usp=sharing

## Array Variable

- Une variable array est une adresse mémoire pointant sur la première case (index 0)
- Chaque case occupe un nombre fixe de bytes selon le type (int: 4 bytes, double: 8 bytes, etc.).
- Toutes les cases de l'array sont stockées de manière contiguë en mémoire.
- C++ ne vérifie pas si on sort du array. C'est au programmeur de rester dans les limites
- Un dépassement de array peut provoquer une erreur de segmentation (segmentation fault) ou corrompre silencieusement la mémoire voisine

```
int primes[]= {1, 2, 3, 5, 7};
for(int i=0; i<5; i++)
  cout<< "Memory address " << primes + i <<
         " contains " << primes[i] <<'\n';
```

```
Memory address 0x7fffffffd880 contains 1
Memory address 0x7fffffffd884 contains 2
Memory address 0x7fffffffd888 contains 3
Memory address 0x7fffffffd88c contains 5
Memory address 0x7fffffffd890 contains 7
```




```
int arrsize= sizeof(primes) / sizeof(primes[0]);
```

ULB

En C++, le nom d'un array représente l'**adresse fixe** de son premier élément. L'arithmétique sur les adresses opère en **unités d'éléments**, pas en bytes. Ajouter un entier n à un array déplace l'adresse de n × sizeof(type) bytes, pour pointer vers le n-ième élément suivant. Exemple : primes + 3 avance de 3 × 4 = 12 bytes (si int occupe 4 bytes).
Opérations autorisées :

- p + n ou p - n -> nouvelle adresse (décalage de n éléments)
- p1 - p2 -> entier représentant la distance en éléments entre deux cellule du même tableau
- Comparaisons (<, <=, ==, etc.) -> résultat booléen

**C++ n'effectue aucune vérification des bornes à l'exécution.** Lorsque vous écrivez arr[i], le compilateur se contente de calculer l'adresse cible et d'y accéder directement, sans tester si i est compris entre 0 et taille-1. Cette absence de contrôle est un choix délibéré du langage: C++ évite toute surcharge en temps d'exécution, au prix de la responsabilité totale du programmeur.

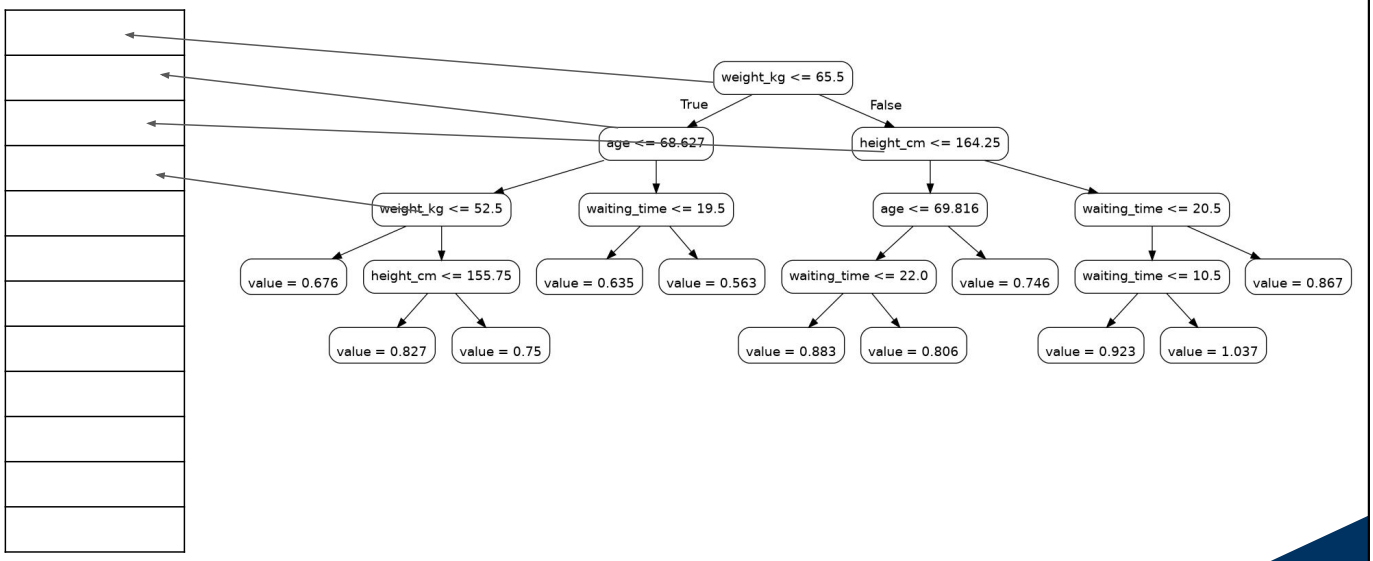
Un accès hors limites a deux conséquences possibles, toutes deux dangereuses :

- **Erreur de segmentation** (*segmentation fault*) : tentative d'accès à une zone mémoire non allouée au processus (votre programme) -> arrêt immédiat du programme.
- **Corruption silencieuse** : l'accès se fait dans une zone mémoire allouée mais appartenant à une autre variable. Le programme continue mais avec des données corrompues -> bug subtil, difficile à diagnostiquer, pouvant se manifester bien plus tard.

L'expression sizeof(primes) / sizeof(primes[0]) est une **idiome classique en C/C++** pour calculer le nombre d'éléments d'un array:

- sizeof(primes) retourne la **taille totale en bytes** du array complet (ex. : 20 bytes pour int primes[5] si int = 4 bytes)
- sizeof(primes[0]) retourne la **taille d'un seul élément** (ex. : 4 bytes pour un int)
- Le quotient donne le **nombre d'éléments** : 20 / 4 = 5 donc le nombre d'éléments d'array primes

## BMD Regression - v3.0 - Chargement de l'Arbre dans un Array






ULB

La version 2.0 que nous avons vue la fois précédente présentait une source d'inefficacité : nous lisions le fichier une fois par nœud. Pour chaque nœud, nous lisions le fichier des députés depuis le début jusqu'à la ligne du nœud concerné. Les entrées/sorties disque sont généralement beaucoup plus lentes que les entrées/sorties vers la mémoire. Donc, quand c'est possible, il faut toujours privilégier la mémoire RAM aux accès disque.

Dans cette version, l'idée principale est de charger le fichier en mémoire RAM. Nous le stockerons dans un array, puis appliquerons la même logique que dans la V2.0. La seule différence est qu'au lieu de rechercher les nœuds dans le fichier, nous les recherchons directement dans l'array en mémoire.

Encore mieux : nous pouvons stocker chaque nœud à l'indice correspondant dans le array. Ainsi, le nœud 1 va dans la case 1, le nœud 10 dans la case 10, et ainsi de suite. De cette façon, rechercher un nœud revient à accéder directement à sa case dans l'array.

Dans la suite, nous expliquerons la version 3.0 de notre programme d'arbre de régression, ainsi que les nouveaux concepts C++ nécessaires pour implémenter cette version.

# read_tree

```
const int MAX_NODES = 32;
Node tree[MAX_NODES];
bool read_tree(char* filename) {
   for (int i = 0; i < MAX_NODES; ++i)  // Initialize all nodes as null
       tree[i] = Node();
   ifstream fp(filename);
   if (!fp) return 0.0; // or handle error

   char line[256];
   int node_id, left_id, right_id;
   char cond_val[128];
   while (fp.getline(line,  sizeof(line))) {
       if (sscanf(line,  "%d,%d,%d,%127[^\n]" , &node_id, &left_id, &right_id, cond_val) !=
4)
           return false; //signal error
       // fill-in the node
       tree[node_id].set_children(left_id, right_id);
       if (left_id == -1 && right_id == -1) {
           // Leaf node: rest is a value
           tree[node_id].mark_as_leaf();
           tree[node_id].set_value(atof(cond_val));
       } else {
           // Internal node: cond_val is a condition
           if (!tree[node_id].parse_condition(cond_val)) {
               cerr <<  "Error: Failed to parse condition: "  << cond_val << endl;
               fp.close();
               return false;
   }   }   }
   fp.close();
   return true;
}
```





- Le programme commence par la fonction read_tree, qui charge l'arbre depuis le fichier dans un tableau.
- On remarque Node tree[MAX_NODES]; : cela signifie qu'il faut définir un type de données pour un nœud d'arbre. Nous le verrons ensuite.
- La fonction ouvre le fichier avec ifstream, puis démarre une boucle.
- Dans la boucle, on lit ligne par ligne, c'est-à-dire nœud par nœud.
- Chaque ligne est parsée avec sscanf.
- On stocke ensuite le nœud dans la case de l'array correspondant à son node_id : tree[node_id].set_children(left_id, right_id);
- Et on remplit les autres données du nœud à partir de la ligne, en utilisant les fonctions du type Node.
- Voyons maintenant comment la classe Node est créée - ce sera la première introduction à la programmation orientée objet dans ce cours.

## Introduction aux Classes

- **Class:** un type de données défini par le programmeur
- Sert à définir des objets
- Constitue un modèle (patron) pour créer des objets

  Ex:

  - std::string fName, lName;

    crée deux objets de la classe string

```
class myType
{

}; // Erreur courante : oublier le ';'


int main()
{
   myType x;
}
```



ULB

**Classe (class) – Type défini par le programmeur**

- Une **classe** est un type de données personnalisé, conçu par le développeur
- Elle sert de **patron** (*blueprint*) pour créer des **objets** (instances concrètes)
- Chaque objet possède sa propre copie des données définies par la classe
- La classe définit quoi stocker et comment agir ; l'objet représente une entité concrète créée à partir de ce modèle.

## Programmation Procédurale et Programmation Orientée Objet

- **Procédurale:** centrée sur les processus et fonctions ; découpage du programme en fonctions manipulant des données de base
- **Orientée objet (POO):** définit des objets modélisant des entités réelles (ex. : Box, Account) avec leurs données et opérations intégrées
- **Encapsulation:** les classes regroupent données et comportements, permettant des expressions intuitives comme bigBox = box1 + box2 + box3
- **Maintenabilité:** code POO plus lisible et évolutif sur les grands projets (malgré une conception initiale plus longue) ; code procédural plus rapide à démarrer mais difficile à faire évoluer
- **Abstraction du domaine:** la POO résout les problèmes avec les entités métier du domaine, pas seulement avec des nombres/caractères -> structures de programme plus naturelles et expressives



ULB

**Programmation procédurale**

- Approche centrée sur les **processus** et les **fonctions** à exécuter
- Découpage du problème en unités de calcul autonomes (fonctions)
- Manipulation directe de types fondamentaux (`int`, `double`, `char[]`)
- Exemple : pour gérer des boîtes, on stocke longueur, largeur et hauteur dans des variables séparées et on calcule manuellement les dimensions résultantes

**Programmation orientée objet (POO)**

- Approche centrée sur les **entités du domaine métier** (objets réels ou conceptuels)
- Identification préalable des types d'objets pertinents (`Box`, `Account`, `Player`) et des opérations associées
- Définition de **classes** qui encapsulent à la fois données *et* comportements
- Exemple : définition d'un type `Box` permettant d'écrire `bigBox = box1 + box2 + box3` — l'opérateur `+` est redéfini pour signifier « créer une boîte contenant les autres »

**Comparaison des approches**

- **POO** : temps de conception plus long, mais code plus lisible, maintenable et évolutif - particulièrement avantageux pour les grands projets
- **Procédurale** : démarrage rapide, mais complexité croissante avec la taille du code ; difficile à faire évoluer sans refactorisation majeure
- La POO modélise le problème avec les entités du domaine métier, non avec les primitives machine (nombres, caractères), ce qui rend la logique plus naturelle et expressive

## Encapsulation - Spécificateurs d'Accès

- Les membres d'une classe sont déclarés avec un spécificateur d'accès qui contrôle leur visibilité:
  - **public:** accessible depuis n'importe quelle fonction du programme (interface externe)
  - **private:** accessible uniquement par les fonctions membres de la classe elle-même (comportement par défaut si non spécifié)
  - **protected:** réservé à l'héritage (similaire à private mais accessible aux classes dérivées)
- Ces sections peuvent apparaître plusieurs fois et dans n'importe quel ordre dans la définition d'une classe.
- Le choix stratégique de ces accès structure la frontière entre l'interface publique et l'implémentation interne.

```
class Square {
 private:
   float side;
 public:
   void setSide(float s){
     side = s; }
   float getSide(){
     return side; }
};

int main() {
 Square x;
 x.side= 5; // erreur: 'int Square::side'
est privé dans ce contexte
 x.setSide(5);}
```

76

ULB

**Encapsulation**
L'encapsulation regroupe au sein d'un même type les **données** (attributs) et les **fonctions** (méthodes) qui les manipulent. Chaque objet instancié à partir de la classe contient :

- Un état défini par ses membres de données
- Un comportement défini par ses méthodes membres
  Cette combinaison permet de modéliser une entité du domaine métier de façon cohérente: l'objet devient une unité autonome qui porte à la fois ses propriétés et les opérations légitimes qui s'y appliquent.

**Masquage des données (*data hiding*)**
Le masquage des données consiste à déclarer les membres de données en `private` afin de :

- Préserver l'**intégrité** de l'objet (ex. : empêcher un taille négatif)
- Contrôler les modifications via des méthodes publiques qui appliquent des règles métier
- Séparer clairement **interface** (méthodes `public`) et **implémentation** (données `private`)
  Cette séparation permet de modifier l'implémentation interne sans impacter le code client - condition essentielle à la maintenabilité et à l'évolutivité des grands systèmes logiciels. La POO ne manipule plus

- des bits bruts, mais des objets cohérents dont l'état est protégé et les interactions sont définies par une interface explicite.

**Spécificateurs d'accès et masquage des données**
Dans la classe `Square`, le membre `side` est déclaré `private` : il n'est accessible qu'aux méthodes membres de la classe. Toute tentative d'accès direct depuis l'extérieur (`x.side = 5;`) provoque une **erreur de compilation** - le compilateur refuse de violer la frontière d'encapsulation. À l'inverse, les méthodes `setSide()` et `getSide()` sont déclarées `public` et forment l'**interface contrôlée** de la classe.

**Encapsulation en action**
La classe `Square` regroupe :

- **État interne** (`side`) : représentation privée de la donnée
- **Interface publique** (`setSide()`, `getSide()`) : opérations autorisées sur l'objet
  Cette combinaison constitue l'encapsulation : l'objet devient une entité cohérente qui porte à la fois ses propriétés et les moyens légitimes de les manipuler.

**Pourquoi masquer `side` ?**

1. **Intégrité de l'objet** : une méthode `setSide()` peut valider l'entrée (ex. : rejeter une valeur négative) avant modification - impossible avec un accès direct.
2. **Indépendance de l'implémentation** : on peut ultérieurement changer la représentation interne (ex. : stocker la surface au lieu du côté) sans modifier le code client, car celui-ci ne dépend que de l'interface publique.
3. **Respect du paradigme objet** : on programme en termes d'*objets* (`Square`) et non en termes de leurs *bits internes* (`side`). L'utilisateur de la classe ne doit pas connaître ni manipuler directement sa structure interne.

# Types de Fonctions Membres

- **Fonctions d'accès (get, getter) :** permettent de lire une variable membre sans la modifier.
  - ex: getSide
- **Fonctions de modification (set, setter) :** permettent de modifier une variable membre.
  - ex: setSide
- **Fonctions membres :** effectuent la logique propre à la classe.
- **Constructeurs :** appelés automatiquement lors de la création d'un objet.
- **Destructeur :** appelé automatiquement juste avant la suppression de l'objet.

```
class Square {
 private:
   float side;
 public:
   void setSide(float s){
     side = s; }
   float getSide(){
     return side; }
};

int main() {
 Square x;
 x.side= 5; // erreur: 'int Square::side'
est privé dans ce contexte
 x.setSide(5);}
```

## Séparation de l'Interface et de l'Implémentation

Separating interface from implementation

- Facilite la modification des programmes
- Fichiers d'en-tête (.h) Contient les définitions de classes et les prototypes de fonctions
- Fichiers de code source (.cpp) Contient les définitions des fonctions membres

```
//square.h
class Square {
private:
  float side;
public:
  void setSide(float s){ // fonction inline
    side = s; }
  float getSide(){ //fonction inline
    return side; }
  bool intersects(Square other); //prototype
};
```

```
//square.cpp
bool Square::intersects(Square other){ //définition
// Test d'intersection ici
}
```

62

ULB

**Principe fondamental -** Séparation stricte entre :

- **Interface** (`.h`) : déclaration de la classe, prototypes des méthodes, spécificateurs d'accès (`public`/`private`). Contient uniquement ce que l'utilisateur de la classe *doit connaître* pour l'utiliser.
- **Implémentation** (`.cpp`) : définitions concrètes des méthodes et constructeurs. Contient la logique métier cachée à l'utilisateur.

**Avantages clés**

- **Maintenabilité** : modification de l'implémentation sans recompiler le code client (tant que l'interface `.h` reste stable)
- **Encapsulation renforcée** : l'utilisateur ne voit que *quoi faire*, pas *comment c'est fait*
- **Compilation incrémentale** : changement dans `.cpp` → recompilation locale uniquement
- **Réutilisabilité** : distribution possible de la bibliothèque compilée + fichier `.h` sans divulguer le code source

**Note technique**

Les méthodes définies *dans* la classe `.h` sont implicitement `inline` (On se reverra plus tard) ; celles définies *hors* classe (dans `.cpp`) ne le sont pas - ce qui est souhaitable pour les méthodes non triviales.

## Class Node - node.h

```
class Node {
private:
   //member variables
   bool is_leaf;
   float value;
   int left_id;
   int right_id;

public:
   //setter and getters
   void mark_as_leaf() {is_leaf= true;}
   void set_value(double v) {value= v;}
   void set_children(int l, int r) {left_id= l; right_id= r;}
   bool test_leaf() {return is_leaf;}
   float get_value() {return value;}
   int get_left() {return left_id;}
   int get_right() {return right_id;}

   // Parsed condition (for internal nodes)
   Feature feature;
   Operator op;
   float threshold;
   // constructor
   Node();
   // member functions
   bool parse_condition(char* cond);
   bool eval_condition(float features[FEATURE_COUNT]);
};
```

```
enum Feature {
   WEIGHT_KG = 0,
   AGE,
   HEIGHT_CM,
   WAITING_TIME,
   FEATURE_COUNT=4
};

enum Operator {
   OP_LE,  // <=
   OP_LT,  // <
   OP_EQ,  // =
   OP_GE,  // >=
   OP_GT   // >
};
```

80

ULB

## Principes OOP appliqués à la classe `Node`

### Encapsulation et masquage des données

Les attributs internes (`is_leaf`, `value`, `left_id`, `right_id`) sont déclarés `private` :

- Protection de l'**intégrité de l'objet** - empêche des états incohérents (ex. : nœud feuille avec `left_id ≠ -1`)
- L'utilisateur ne manipule pas directement les bits internes mais passe par l'**interface publique** (`mark_as_leaf()`, `set_children()`, etc.)

### Interface contrôlée via accesseurs/mutateurs

Les méthodes `public` forment une barrière de validation implicite :

- `mark_as_leaf()` garantit que le passage en feuille suit la logique métier (sans exposer `is_leaf` directement)
- `set_children()` permet de valider ultérieurement les identifiants (`l`, `r ≥ 0` ou `= -1`) sans modifier le code client
    - Respect du principe *programmer en termes d'objets, pas de bits*

**Séparation interface / implémentation**
Dans un projet réel, cette classe serait structurée ainsi :

Fichier `Node.h` (interface)
- Déclaration de la classe avec membres `private`/`public`
- Prototypes des fonctions (`bool eval_condition(...);`)
- Enum types `Feature`, `Operator` (déclarés)

Fichier `Node.cpp` (implémentation)
- Définitions des méthodes hors classe (`Node::parse_condition(...) { ... }`)
- Logique complexe cachée (parsing, évaluation)
- Constructeur `Node::Node() { ... }` initialisant l'état

**Note sur les types composés**
Les membres `Feature` et `Operator` illustrent l'encapsulation de **logique métier** : la condition d'un nœud interne est modélisée comme une entité structurée plutôt qu'une string brute - préparant le terrain pour une évaluation type-sûre via `eval_condition()`.

## Constructor - node.cpp

```
Node::Node()
   : is_leaf(false), value(0.0),
     feature(WEIGHT_KG), op(OP_LE), threshold(0.0),
     left_id(-1), right_id(-1)
{}
```



ULB

## Constructeurs : principes et initialisation

### Définition et rôle fondamental

Un constructeur est une fonction spéciale appelée **automatiquement** lors de la création d'un objet. Il porte obligatoirement le **même nom que la classe** et **n'a pas de type de retour** (pas même `void`). Son rôle : garantir que tout objet démarre dans un **état valide et cohérent**, sans valeurs résiduelles indéterminées. Une class peut avoire plusieure constructeurs.

### Constructeur par défaut

- Si aucun constructeur n'est défini, le compilateur en fournit un *par défaut* sans paramètre - mais il ne garantit **aucune initialisation explicite** des membres.
- Dès qu'un constructeur est défini par le programmeur, le constructeur par défaut **disparaît** (sauf à le déclarer explicitement - on se reverra plus tard).
- Pour les classes utilisées dans des arrays (`Node tree[MAX_NODES]`), un constructeur sans paramètre est **obligatoire** : chaque case du array sera initialisée automatiquement à la création.

**Liste d'initialisation : syntaxe et avantages**
La syntaxe `: membre1(valeur1), membre2(valeur2), ...` placée après la signature du constructeur permet d'initialiser les membres **au moment même de leur création**, et non par affectation ultérieure dans le corps. Avantages :

**Affectation dans le corps**

- `length = l;` -> construction par défaut puis copie
- Inefficace pour objets complexes
- Impossible pour `const` ou références

**Liste d'initialisation**

- `: length(l)` -> construction directe avec valeur
- Plus performant (évite copies inutiles)
- **Seule méthode possible** pour les variables `const`

**Constructeur par défaut de `Node` avec liste d'initialisation**

- L'idée consiste à initialiser les membres avec des valeurs fictives (par défaut), représentant logiquement un nœud vide.

**Déclaration vs définition**
Le constructeur est **déclaré** dans la classe (fichier `.h`) mais **défini** séparément dans le fichier `.cpp` :

Syntaxe `Node::Node()`

- `Node::` opérateur de **résolution de portée** (`scope resolution operator`)
- Lie la définition à la classe `Node`, indiquant au compilateur : cette fonction appartient à la classe `Node`
- Nécessaire car la définition se trouve **hors du bloc `{}` de la classe**

## Enumerated Data Types

- Type de données créé par le programmeur.
- Contient un ensemble de const int nommées, numérotées automatiquement à partir de zéro.
- Possibilité de redéfinir l'association par défaut.

  enum Fruit {apple= 2, grape= 4, orange= 5}

- Les enum améliorent la lisibilité d'un programme.
- Les variables énumérées ne peuvent pas être utilisées avec des instructions d'entrée, telles que 'cin': syntax error
- Le nom associé à la valeur d'un type de données énuméré ne s'affiche pas lorsqu'il est utilisé avec `cout`. cout affichera la valeur int.

```
enum Days {Mon, Tue, Wed, Thur, Fri,
    Sat, Sun};
enum Fruit {apple=15, grape, orange};

int main ()
{
 /* To define variables, use the
enumerated data type name */
 Fruit snack;
 Days workDay, vacationDay;
 /* Variable may contain any valid
value for the data type */
 snack = orange; // no quotes
 if (workDay == Wed)
   cout<<"Wednesday";
}
```

84

ULB

## Enums

Comme vu avant, le C++ est un langage statiquement typé, ce qui signifie que chaque variable et expression possède un type spécifique connu à la compilation. Bien que les types intégrés (comme `int`, `double`, `char`) soient fondamentaux, le développement de logiciels robustes nécessite la définition de types personnalisés pour modéliser des données spécifiques au domaine, améliorer la lisibilité du code et renforcer la sécurité. Au-delà des capacités orientées objet complètes des définitions de `class`, le C++ propose plusieurs mécanismes plus légers pour la définition de types : les énumérations (`enum`), les structures (`struct`) que nous avons deja vu, et les alias de types (`typedef` et `using`). Comprendre les nuances entre ces mécanismes est crucial pour écrire du code C++ moderne, efficace et maintenable.

### 1. Énumérations (Enums)

Les énumérations permettent de définir un type constitué d'un ensemble de constantes entières nommées. Elles sont idéales pour représenter des états, des drapeaux (flags) ou des catégories où une variable ne doit contenir qu'une valeur parmi un ensemble spécifique. Une type énumérés est en interne un integer. Par défaut, le premier identificateur d'une enum reçoit la valeur `0`, et chaque identificateur suivant est incrémenté de 1. Le programmeur peut attribuer explicitement des valeurs entières aux énumérateurs, et les valeurs suivantes continuent l'incrémentation à

partir de la dernière valeur spécifiée. Les types énumérés améliorent considérablement la lisibilité d'un programme. Au lieu d'utiliser des "nombres magiques" cryptiques dans le code :

```
if (workDay == Wed) cout << "Wednesday"; // Plus lisible avec enum
```

**Analyse de l'exemple dans le slide:**

1. **Définition :** `enum Days` crée un type où les jours sont numérotés de 0 à 6. `enum Fruit` montre l'initialisation personnalisée : `apple=15`, donc `grape=16` et `orange=17`.
2. **Déclaration de variables :** On utilise le nom de l'énumération comme type (`Fruit snack;`).
3. **Affectation :** On assigne directement le nom de l'énumérateur (`snack = orange;`) sans guillemets, car ce sont des constantes entières, pas des chaînes de caractères.
4. **Comparaison :** La condition `if (workDay == Wed)` fonctionne car les enums se comportent comme des entiers dans les expressions.

## Function Definition - node.cpp

```
// parse_condition: parses string and stores structured data
bool Node::parse_condition(char* cond) {
   istringstream iss(cond);
   string feat_str, op_str;
   float th;
   if (!(iss >> feat_str >> op_str >> th))
       return false;
   // Parse feature
   if (feat_str == "weight_kg")      feature = WEIGHT_KG;
   else if (feat_str == "age")       feature = AGE;
   else if (feat_str == "height_cm") feature = HEIGHT_CM;
   else if (feat_str == "waiting_time") feature = WAITING_TIME;
   else return false;
   // Parse operator
   if (op_str == "<=") op = OP_LE;
   else if (op_str == "<")  op = OP_LT;
   else if (op_str == "=")  op = OP_EQ;
   else if (op_str == ">=") op = OP_GE;
   else if (op_str == ">")  op = OP_GT;
   else return false;

   threshold = th;
   return true;
}
```

```
// eval_condition: uses pre-parsed members
bool Node::eval_condition(float
features[FEATURE_COUNT]) {
   float feat_val = features[feature];

   switch (op) {
       case OP_LE: return feat_val <= threshold;
       case OP_LT: return feat_val < threshold;
       case OP_EQ: return feat_val == threshold;
       case OP_GE: return feat_val >= threshold;
       case OP_GT: return feat_val > threshold;
       default:    return false;
   }
}
```



ULB

Maintenant que la fonction `read_tree` a été expliquée, nous allons voir la fonction `parse_condition`.
Cette fonction ressemble beaucoup à `eval_condition` de la version 2.0, mais leur but est un peu différent.

- **Dans la version 2.0 :** La fonction servait à vérifier la condition d'un nœud et à dire quelle branche prendre: l'enfant de gauche ou celui de droite.
- **Ici :** Le but est d'analyser le texte de la condition et de le stocker dans l'objet nœud. L'idée est donc de remplir les valeurs des membres : `feature`, `op`, `threshold`.

Ensuite, la fonction `eval_condition` prendra une liste de caractéristiques (les données d'un patient) et renverra la décision de ce nœud : `vrai` pour aller à gauche et `faux` pour aller à droite.

## Increment/Decrement operators

- Increment: augmente la valeur d'une variable
  - ++ ajoute 1 à la variable
  - val++; équivaut à  val = val + 1;
- Decrement – reduce value in variable
  - -- soustrait 1 à la variable
  - val--; équivaut à val = val – 1;
- Ces opérateurs peuvent être utilisés en préfixe (avant) ou en suffixe (après) une variable.

# Prefix v.s. Postfix Modes

- ++val et --val incrémentent ou décrémentent la variable, puis renvoient sa nouvelle valeur.
- val++ et val-- renvoient l'ancienne valeur de la variable, puis l'incrémentent ou la décrémentent.

```
int x = 1, y = 1;
x = ++y; // y est incrémenté à 2
// Puis x est affecté à la valeur 2
cout << x << " " << y; // Affiche 2 2
x = --y;   // y est décrémenté à 1
           // Puis x prend la valeur 1
cout << x << " " << y; // Affiche 1 1
```

```
int x = 1, y = 1;
x = y++; // y++ return  1
// Ensuite, 1 est attribuée à x et y est incrémenté à
2.
cout << x<< " " << y; // Affiche 1 2
x = y--; // y-- return a 2
 // Ensuite, 2 est attribuée x et y est dérémenté à 1
cout << x << " " << y; // Affiche 2 1
```

# estimate

```
// Inference function using array layout
float estimate(float features[FEATURE_COUNT]) {
   int idx = 1; // start at root

   while (idx < MAX_NODES) {
       if (tree[idx].test_leaf()) {
           return tree[idx].get_value();
       }
       bool go_left = tree[idx].eval_condition(features);
       idx = go_left ? tree[idx].get_left() : tree[idx].get_right();
   }

   return 0.0; // a non-void function must always return
}
```