INFOH2001 Object Oriented Programming

Mahmoud SAKR <mahmoud.sakr@ulb.be>

École polytechnique de Bruxelles

2025/26

Lecture 4

Number Systems

Decimal:

- Chiffres: 0,1,2,3,4,5,6,7,8,9
- Base 10
- Utilisé en mathématiques scolaires

10^3	10^2	10^1	10^0	1234
1	2	3	4	

Binary:

- Chiffres: 0,1
- Base 2
- Utilisé dans les ordinateurs et l'électronique

2^3	2^2	2^1	2^0	valeur décimale= 13
1	1	0	1	

Octal:

- Chiffres: 0,1,2,3,4,5,6,7
- Base 8
- rarement utilisé pour l'affichage

8^3	8^2	8^1	8^0	valeur décimale=
1	2	3	4	512+128+24+4= 668

Hexadecimal:

- Chiffres: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
- Base 16
- Utilisé dans les ordinateurs pour l'affichage

16^3	16^2	16^1	16^0	valeur décimale=
1	2	3	4	4096+512+48+4=4660

Conversion décimal → binaire

Méthode manuelle : divisions successives par 2

Algorithme :

1. Diviser le nombre par 2
2. Noter le reste (0 ou 1)
3. Continuer avec le quotient jusqu'à 0
4. Lire les restes **de bas en haut**

Exemple : 42 en décimal au binaire

42 ÷ 2 = 21 reste 0 ← bit le moins significatif (LSB)
21 ÷ 2 = 10 reste 1
10 ÷ 2 = 5 reste 0
5 ÷ 2 = 2 reste 1
2 ÷ 2 = 1 reste 0
1 ÷ 2 = 0 reste 1 ← bit le plus significatif (MSB)
Résultat : $42_{10} = 101010_2$

Conversion décimal → hexadécimal

Meme algorithm, mais divisions successives par 16

Algorithme :

1. Diviser le nombre par 16
2. Noter le reste (0–15, où 10=A, 11=B, ..., 15=F)
3. Continuer avec le quotient jusqu'à 0
4. Lire les restes **de bas en haut**

Exemple : 255 en décimal au hexadécimal

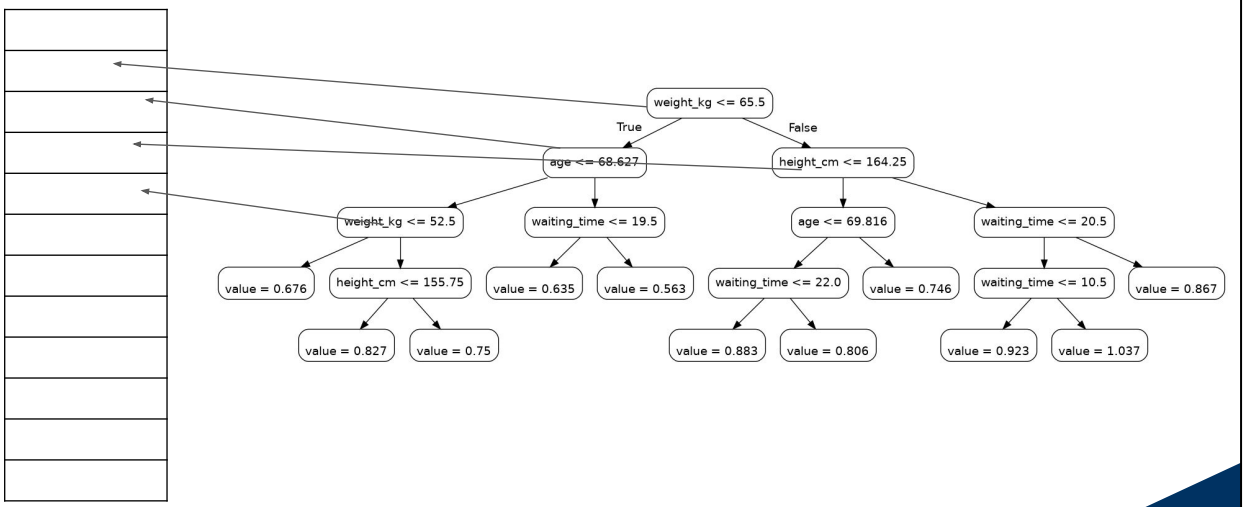
255 ÷ 16 = 15 reste 15 -> F
15 ÷ 16 = 0   reste 15 -> F
Résultat : $255_{10} = FF_{16}$

Code C++ : conversion décimal -> binaire

```cpp
#include <iostream>
#include <string>

std::string decimalToBinary(int n) {
    if (n == 0) return "0";

    std::string binary = "";
    while (n > 0) {
        // Préfixe le bit courant (évite std::reverse)
        binary = ((n % 2) ? '1' : '0') + binary;
        n /= 2;
    }
    return binary;
}

int main() {
    int num = 42;
    std::cout << num << " en binaire = " << decimalToBinary(num) <<
std::endl;
    // Affiche : 42 en binaire = 101010
    return 0;
}
```

```cpp
#include <iostream>
#include <string>

std::string decimalToHex(int n) {
    if (n == 0) return "0";

    std::string hex = "";
    while (n > 0) {
        int remainder = n % 16;
        char digit;
        if (remainder < 10)
            digit = '0' + remainder;      // 0-9
        else
            digit = 'A' + (remainder - 10); // A-F (grace à tableau ASCII)
        hex = digit + hex;                // préfixe le chiffre à gauche
        n /= 16;
    }
    return hex;
}

int main() {
    int num = 255;
    std::cout << num << " en hexadécimal = " << decimalToHex(num) << std::endl;
    // Affiche : 255 en hexadécimal = FF

    // Autres exemples
    std::cout << "10  = " << decimalToHex(10) << std::endl;   // A
    std::cout << "42  = " << decimalToHex(42) << std::endl;   // 2A
    std::cout << "1000 = " << decimalToHex(1000) << std::endl; // 3E8
    return 0;
}
```

Limitations de v3.0 - l'Allocation Statique de Mémoire



ULB

La version précédente du programme d'arbre de régression souffrait de deux limitations majeures liées à l'utilisation d'un array statique:

- Tout d'abord, la taille d'array devait être fixée à l'avance **Node tree[MAX_NODES]**, avant même de connaître la taille réelle de l'arbre contenu dans le fichier. Cette contrainte rendait le programme peu flexible face à des modèles de tailles variables.
- Deuxièmement, l'array présentait des cases vides ou *gaps*, car les identifiants de nœuds (node_id) stockés dans le fichier ne sont pas nécessairement contigus. Les cases 0, 16, 17, 20, 21, 22, 23, 26, 27 restent réservée dans la mémoire mais inutilisées. Cette discontinuité entraînait un gaspillage de mémoire.

Ces problèmes illustrent les limites fondamentales des arrays statiques en C++. Dans de nombreuses applications, notamment celles impliquant le chargement de données externes, la quantité de données ne peut pas être prédite à l'avance. Le nombre d'éléments peut varier considérablement d'une exécution à l'autre, voire évoluer pendant l'exécution du programme lui-même. Un array statique, dont la taille est déterminée à la compilation, ne peut pas s'adapter à ces variations dynamiques.
Une solution naïve consiste à déterminer une valeur maximale possible *N* pour la taille des données, comme on a fait dans v2.0. La, on a prévu une taille maximale de 32. Mais plus réalistement, nous devrions un nombre beaucoup

plus important. Par exemple, on pourrait définir MAX_NODES à 10 000 et déclarer Node tree[10000]. Cependant, cette approche présente de sérieux inconvénients. Si l'arbre réel ne contient que quelques dizaines de nœuds, la grande majorité d’array reste inutilisée, gaspillant ainsi la mémoire. À l'inverse, si l'arbre dépasse la limite fixée, le programme risque un dépassement d’array avec des conséquences imprévisibles. Cette rigidité montre la nécessité d'adopter des structures de données plus flexibles.

Allocation Dynamique de Mémoire

- La mémoire allouée dynamiquement est définie à l'exécution.
- L'allocation dynamique de mémoire permet une utilisation plus efficace de celle-ci.
- Elle est souvent utilisée pour gérer les structures de données dynamiques telles que les tableaux et les arbres dynamiques.
- La mémoire allouée dynamiquement doit être libérée pendant l'exécution lorsqu'elle n'est plus nécessaire.

Avant toute chose, comprenons comment la mémoire est gérée en programmation.

ULB

Mémoire statique vs mémoire dynamique
L'allocation statique, utilisée avec les tableaux classiques comme `Node tree[MAX_NODES]`, réserve la mémoire **à la compilation**. La taille est fixe, connue à l'avance, et l'espace alloué reste constant pendant toute l'exécution du programme. Cette approche est simple mais rigide: on gaspille de la mémoire si les données sont plus petites que prévu, ou on risque un dépassement si elles sont plus grandes.
À l'inverse, l'allocation dynamique réserve la mémoire **à l'exécution**, au moment précis où le programme en a besoin. La taille peut être déterminée par des données externes (comme la taille d'un fichier), par l'utilisateur, ou par la logique même du programme. Cette flexibilité permet d'adapter exactement la quantité de mémoire allouée à la quantité de données réelles à traiter.

Jusqu'ici, la seule méthode étudiée pour réserver de la mémoire était de définir des variables. Par exemple, en déclarant `int x`, on réserve quatre bytes en mémoire pour la variable `x`. La réservation (et la libération) de cette mémoire était effectuée automatiquement, sans contrôle de votre part. Le compilateur réserve ces bytes au début de la fonction où la variable est définie, puis libère cette mémoire lorsque la fonction se termine.
C++, cependant, fournit des outils explicites permettant au programmeur de contrôler directement la mémoire : réserver un nombre précis de bytes et les libérer manuellement. Ce contrôle direct nous permet ainsi de gérer la

mémoire de façon dynamique pendant l'exécution, en réservant exactement la quantité nécessaire et en libérant celle qui ne l'est plus.

The Address Operator

- Chaque variable d'un programme est stockée à un emplacement unique en mémoire, qui possède une adresse.
- Utilisez l'opérateur d'adresse & pour obtenir l'adresse d'une variable :

```
int num = -23;
cout << &num<< '\t'<< num;
```

```
0x7fffffffd8e4  -23
```

Pointer Variables

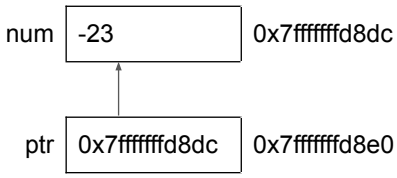
- Pointer variable (pointer): variable contenant une adresse.
- Les pointeurs offrent une méthode alternative d'accès à la mémoire.
- int * : peut stocker une adresse mémoire, où cette adresse contient une valeur entière.
- Les espaces dans la définition sont indifférents : `int* ptr`, `int *ptr`, `int * ptr`.
- L'astérisque `*` fait partie du nom de la variable :

  `int *ptr, x, *y=nullptr, z=12;`

- nullptr est une adresse spéciale qui veut dire: ne pointe vers rien.
- L'astérisque `*` est appelé opérateur d'indirection ; il permet d'accéder à la valeur pointée par le pointeur.

```
int num = -23;
int *ptr= &num;
cout <<ptr<<'\t'<<*ptr<<'\t'<<&ptr;
```

```
0x7fffffffd8dc  -23     0x7fffffffd8e0
```

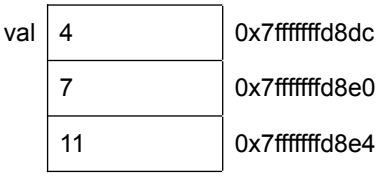


ULB

Arrays and Pointers

```
int val[] = {4, 7, 11};
cout<< val[0]<< '\t'<< val[1]<< '\t'<< val[2]<< endl;
cout<< &val[0]<< '\t'<< &val[1]<< '\t'<< &val[2]<< endl;
cout<< val<< '\t'<< val+1<< '\t'<< val+2<< endl;
cout<< *val<< '\t'<< *(val+1)<< '\t'<< *(val+2)<< endl;
```

```
4    7    11
0x7fffffffd8dc  0x7fffffffd8e0  0x7fffffffd8e4
0x7fffffffd8dc  0x7fffffffd8e0  0x7fffffffd8e4
4    7    11
```



ULB

Arrays et pointeurs : relation fondamentale

Nom de array = adresse du premier élément

Le nom d'un array (`val`) se comporte comme un **pointeur constant** vers son premier élément (`&val[0]`). Contrairement à un vrai pointeur, cette adresse est **fixe** et ne peut pas être modifiée (`val = autre_array;` -> erreur de compilation). Exemple avec `int val[] = {4, 7, 11};`

Arithmétique des pointeurs : décalage en unités d'éléments

Ajouter un entier `n` à un pointeur déplace l'adresse de `n × sizeof(type)` bytes, pas de `n` bytes bruts :

- `val + 1` -> adresse de `val[1]` (décalage de 4 bytes pour un `int`)
- `val + 2` -> adresse de `val[2]` (décalage de 8 bytes)

L'accès par indice et le déréférencement avec arithmétique sont strictement équivalents :

- `val[i]` ≡ `*(val + i)`
- `val[0]` ≡ `*val`
- `val[2]` ≡ `*(val + 2)`

Règles critiques de sécurité

- L'arithmétique est valide uniquement dans l'intervalle `[&array[0], &array[taille]]`
- Dépasser ces limites -> comportement indéfini (plantage ou corruption mémoire silencieuse).
- Parenthèses obligatoires pour le déréférencement après arithmétique : `*(ptr + 1)` (et non `*ptr + 1`, qui ajoute 1 à la *valeur* pointée).

Initialisation d'un pointeur depuis un array

```
int tab[5];
int* p = tab;   // Équivalent à : int* p = &tab[0];
```

-> `p` devient un pointeur modifiable pointant vers l'array. Contrairement à `tab`, `p` peut être réaffecté (`p = autre_array;`).

Pourquoi cette relation existe ?
Les arrays C/C++ sont stockés **contiguëment en mémoire**. L'arithmétique des pointeurs exploite cette contiguïté pour accéder efficacement aux éléments sans calcul manuel d'offsets. Cette abstraction permet d'écrire du code générique fonctionnant sur n'importe quel array ou segment de mémoire alloué dynamiquement.

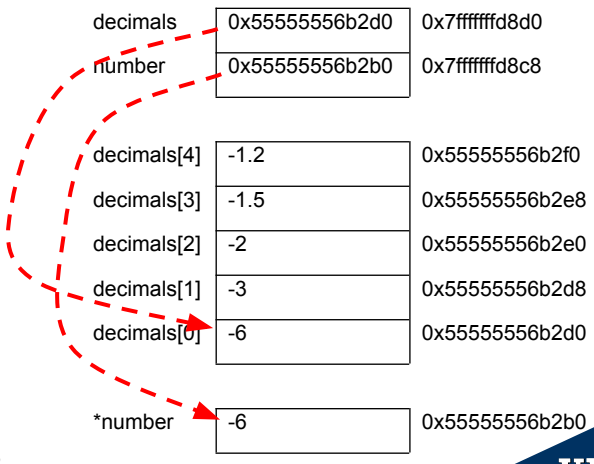
Allocation Dynamique de Mémoire

Deux opérateurs de pointeur (extrêmement puissants) pour gérer dynamiquement la mémoire pendant l'exécution du programme :

- new: alloue de la mémoire et renvoie l'adresse ;
- delete: libère la mémoire pointée par le pointeur.

```
int main(){
 float *number;
 double *decimals;

 number= new float(-6);
 decimals= new double[5];
 for(int i= 1; i<=5; ++i){
   decimals[i]= *number / (i+1);
   cout<< decimals[i]<<'\n';
 }
 delete number;
 delete[] decimals;
 return 0;
}
```

```
-6
-3
-2
-1.5
-1.2
```



ULB

Opérateurs `new` et `delete`

L'opérateur `new` permet d'allouer dynamiquement de la mémoire à l'exécution. Il retourne l'adresse d'un bloc réservé pour le type demandé, que l'on stocke dans un pointeur. On peut initialiser la valeur lors de l'allocation : `p = new double(3.14)`. L'accès aux données s'effectue par déréférencement (`*p = 3.14`).

La libération explicite s'effectue avec `delete p`, qui rend la mémoire au système mais **ne modifie pas le pointeur** : `p` conserve une adresse désormais invalide (*dangling pointer*). Omettre `delete` entraîne des fuites (*memory leaks*).

- `new Type(valeur)` alloue un **objet unique** et retourne son adresse (`Type*`)
- `delete ptr` libère la mémoire pointée (sans modifier `ptr` lui-même -> d'où la nécessité de `ptr = nullptr`
- **Initialisation** possible à l'allocation : `new double{3.14}`

```
float* number = new float(-6);      // Alloue un float
initialisé à -6 sur le tas
*number = 3.14;                     // Accès via
déréférencement
delete number;                      // Libère la mémoire
number = nullptr;                   // Bonne pratique :
réinitialiser le pointeur
```

L'allocation dynamique de petits types comme double ou int est plutôt rare en pratique. On l'utilise plus souvent pour allouer dynamiquement des structures plus grandes comme les arrays ou les arbres.

Allocation d'arrays dynamiques

L'allocation dynamique d'array s'effectue avec l'opérateur `new` suivi de crochets indiquant la taille : `double* decimals = new double[n];` réserve un bloc contigu de `n` éléments, l'adresse du premier élément étant stockée dans le pointeur. Contrairement aux arrays statiques, la taille `n` peut être déterminée à l'exécution (ex. : saisie utilisateur), offrant une flexibilité essentielle pour les structures de données de taille variable. L'accès aux éléments utilise la notation usuelle (`decimals[i]` ou `*(decimals + i)`). Toutefois, les'arryas dynamiques ne peuvent pas être initialisés directement lors de l'allocation : chaque élément doit être assigné explicitement après création.

La libération mémoire exige l'opérateur `delete[]` (avec crochets) : `delete[] decimals;`. Omettre les crochets (`delete decimals;`) provoque un comportement indéfini, car le compilateur ne libérerait qu'un seul élément au lieu du tableau entier. Après suppression, le pointeur doit être réinitialisé à `nullptr` pour éviter les *dangling pointers*. Bien que puissante, cette gestion manuelle est sujette aux erreurs (fuites, corruptions).

- `new Type[n]` alloue un array contigu de `n` éléments
- `delete[] ptr` libère l'array complet (les crochets `[]` sont indispensables : `delete` seul -> comportement indéfini)

Pourquoi utiliser l'allocation dynamique ?

- Taille des données **inconnue à la compilation** (ex. : nombre de nœuds d'un arbre lu depuis un fichier)
- Structures de données **évolutives** (listes chaînées, arbres) dont la taille change pendant l'exécution
- Optimisation mémoire : allouer *exactement* ce qui est nécessaire, sans gaspillage

Conséquences de l'oubli de `delete`

Si l'on oublie d'appeler `delete` (ou `delete[]` pour un array) sur un pointeur alloué dynamiquement, la mémoire reste réservée jusqu'à la fin du programme : c'est une **fuite mémoire** (*memory leak*). Le bloc alloué devient inaccessible dès que le pointeur est réaffecté ou sort de portée, mais il n'est jamais restitué au système. Dans un programme court, l'impact est limité (le système récupère toute la mémoire à la terminaison). En revanche, dans une boucle ou un service longue durée (ex: gaming server, database server, learning management system), les fuites s'accumulent progressivement, épuisant la mémoire jusqu'à provoquer un échec d'allocation (`new` lance une exception) ou un plantage du programme. Contrairement aux variables statiques ou automatiques, la mémoire dynamique **n'est jamais libérée automatiquement** - chaque `new` exige un `delete` explicite.

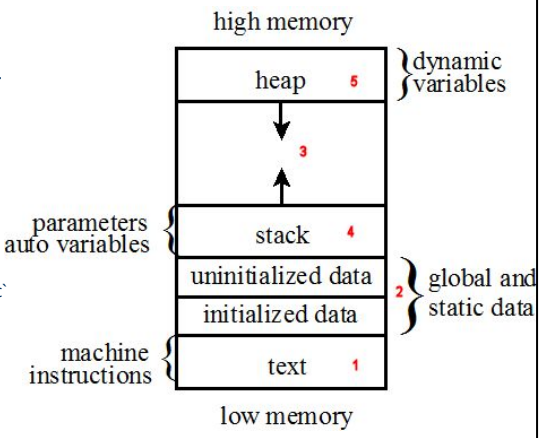
Structure de la Mémoire

Lors de l'exécution d'un programme, l'operating system lui alloue un espace mémoire.

Le programme partitionne cette mémoire comme illustré :

- **texte :** pour les instructions machine du programme (c'est-à-dire le code exécutable).
- **Les variables globales** sont définies dans la portée globale, en dehors de toute fonction ou objet ; **les variables statiques** contiennent le mot-clé `static` dans leur définition.
- **Stack (pile) :** contient toutes les variables locales de la fonction courante.
- **heap(tas) :** la partie de la mémoire qui peut être allouée/libérée dynamiquement avec les opérateurs `new` et `delete`:

  Contrairement àu stack, le heap n'est pas libéré automatiquement à la fin d'une fonction. Il est nécessaire de le libérer manuellement (en C/C++) ou par l'intermédiaire 'garbage collector' (en Java ou Python).



ULB

Gestion mémoire : stack et heap

Un programme C++ en exécution organise sa mémoire en plusieurs régions fonctionnelles distinctes. Le segment de texte contient les instructions machine du programme. Une zone dédiée stocke les variables globales et statiques, allouées une fois au démarrage et libérées uniquement à la terminaison du programme.

Les deux régions dynamiques essentielles sont la stack (pile) et le heap (tas), qui se distinguent fondamentalement par leur mode de gestion et leurs cas d'usage.

La stack: fonctionne selon le principe LIFO (*Last-In First-Out*) : les allocations et désallocations ne peuvent se produire qu'au sommet de la pile. Elle accueille automatiquement toutes les variables locales non-statiques ainsi que les paramètres des fonctions. Lorsqu'une fonction est appelée, ses variables sont "poussées" sur la stack ; à son retour, elles sont immédiatement "dépilées" et la mémoire est restituée. Cette gestion entièrement automatique, couplée à une implémentation matérielle simple (ajustement d'un pointeur de sommet), rend les opérations sur la stack extrêmement rapides.

Toutefois, cette efficacité s'accompagne de contraintes : la taille de la stack est limitée (typiquement quelques mégabytes), et la durée de vie des objets est rigidement liée à la durée de leur fonction. La stack convient donc aux données temporaires de taille modeste et prévisible.

Le heap offre une flexibilité que la stack ne permet pas. Contrairement à la pile, il autorise l'allocation et la libération de blocs mémoire à n'importe quelle position et dans n'importe quel ordre. Cette liberté permet de créer des structures dont la taille n'est connue qu'à l'exécution - comme un array de `n` éléments où `n` provient d'une saisie utilisateur - ou dont la durée de vie dépasse celle de la fonction qui les a créées. En C++, cette gestion explicite s'effectue via les opérateurs `new` (allocation) et `delete`/`delete[]` (libération). Toutefois, cette souplesse a un coût : le gestionnaire de heap doit rechercher un bloc contigu suffisamment grand pour chaque allocation, puis fusionner les blocs libres adjacents pour limiter la fragmentation. Ces opérations rendent l'allocation sur le heap plus lente que sur la stack. De plus, la responsabilité de la libération incombe entièrement au programmeur ; un oubli de `delete` entraîne une fuite mémoire, et une libération incorrecte (`delete` au lieu de `delete[]` pour un array) provoque un comportement indéterminé.

En pratique, un programme robuste exploite les deux régions de manière complémentaire : la stack pour les objets éphémères et de petite taille, le heap pour les structures dynamiques de grande taille ou de durée de vie étendue. Cette dualité constitue le fondement de la gestion mémoire efficace et adaptable en C++.

Pointeur Vers une Classe (ou struct)

Accès aux membres via ->

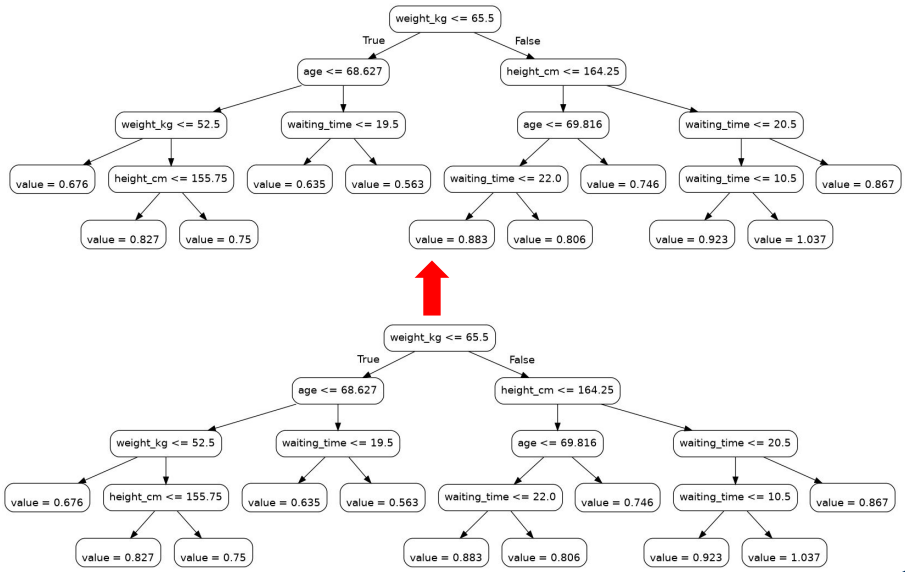
- Lorsqu'un pointeur pointe vers un objet de classe, l'opérateur -> (flèche) permet d'accéder aux membres de l'objet
- Syntaxe : ptr->membre équivaut à (*ptr).membre (déréférencement + accès)
- L'opérateur . (point) s'utilise uniquement sur l'objet lui-même, pas sur un pointeur

```
class Box {
public:
   double length;
   double volume() { return length * length * length; }
};
int main() {
   Box* pbox = new Box();  // Pointeur vers un objet Box
   pbox->length = 5.0;     // Accès au membre via ->
   double v = pbox->volume();  // Appel de méthode via ->
     // Équivalent (mais plus verbeux) :
   (*pbox).length = 5.0;
   double v2 = (*pbox).volume();

   delete pbox;
   return 0;
}
```

BMD Regression - v4.0 - Charger un Arbre dans un Arbre

Mémoire

Fichier



ULB

Dans la version 4, au lieu de lire l'arbre dans un fichier, nous le lisons dans un arbre ayant exactement la même structure.
Avez-vous des idées sur la façon de créer une structure arborescente dans le code ?

Class Node - node.h

```
class Node {
private:
   float value;
   float threshold;
   Node* left;
   Node* right;
public:
   //setter et getters
   void set_value(double v) {value= v;}
   void set_left(Node * l) {left= l;}
   void set_right(Node * r) {right= r;}
   bool test_leaf() {return (left==nullptr && right==nullptr);}
   float get_value() {return value;}
   Node* get_left() {return left;}
   Node* get_right() {return right;}

   // Parsed condition (pour les nœuds internes)
   Feature feature;
   Operator op;

   Node();
   Node(const char* cond_val, const bool is_leaf);
   ~Node();

   bool parse_condition(const char* cond, const bool is_leaf);
   float eval_condition(const float features[FEATURE_COUNT]) const;
   void print_tree(Node * cur);
};
```

```
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   // >
};
```

81

ULB

Voici l'astuce fondamentale pour modéliser un arbre : il faut créer un **type de données récursif**. La classe `Node` est conçue de manière à ce que ses enfants gauche et droit soient eux-mêmes des `Node`, formant ainsi une structure arborescente. Pour éviter une récursion infinie dans la définition du type (un `Node` contenant directement d'autres `Node` entraînerait une taille mémoire infinie), les enfants sont stockés sous forme de **pointeurs** (`Node* left; Node* right;`). Les pointeurs ayant une taille fixe (8 bytes sur une machine 64 bits), la taille de la classe reste finie et bien définie.

Pourquoi des pointeurs pour les enfants ? Sans pointeurs, la définition `Node left; Node right;` serait invalide : chaque `Node` contiendrait deux `Node` complets, eux-mêmes contenant chacun deux `Node`, etc. -> taille infinie. C++ l'empêche-t-il lors de la compilation et génère-t-il une erreur de syntaxe ?

Les pointeurs brisent cette récursion en stockant uniquement des *adresses*, permettant ainsi une structure arborescente dynamique et efficace en mémoire.

Cette conception récursive est à la base de nombreuses structures de données. Elle est expliquée plus en détail dans le module BA3 du cours INFO-H304 Compléments de programmation et d'algorithmique.

Constructor and Destructor - node.cpp

```
Node::Node()
   : left(nullptr), right(nullptr), value(0),
feature(WEIGHT_KG)
{}

Node::Node(const char* cond_val, const bool
is_leaf)
   : left(nullptr), right(nullptr),
feature(WEIGHT_KG) {
   parse_condition(cond_val, is_leaf);
}

Node::~Node() {
   if(left !=nullptr) delete left;
   if(right !=nullptr) delete right;
}
```

```
void Node::print_tree(Node* cur) {
   if(cur->left != nullptr) print_tree(cur->left);
   cout<<cur->value<<", "<<cur->feature<<", "
<<cur->threshold<<endl;
   if(cur->right != nullptr) print_tree(cur->right);
}
```


Le constructeur par défaut initialise l'objet dans un état cohérent et sûr dès sa création :

```
Node::Node()
   : left(NULL), right(NULL), value(0), feature(WEIGHT_KG)
{}
```

Les pointeurs `left` et `right` sont mis à `NULL` pour indiquer l'absence d'enfants, `value` prend une valeur neutre (`0`), et `feature` reçoit une valeur par défaut. Cette initialisation garantit qu'un `Node` non encore rempli reste dans un état valide - essentiel pour éviter les comportements indéfinis lors de l'inférence.
Le constructeur paramétré permet de créer un nœud directement à partir d'une chaîne de caractères représentant soit une condition (`"HEIGHT_CM <= 175"`), soit une valeur feuille (`"3.14"`), selon le paramètre `is_leaf`. Après avoir initialisé les pointeurs enfants à `NULL`, il délègue l'analyse syntaxique à `parse_condition()`, qui remplit les champs `threshold`, `op`, `feature` (pour un nœud interne) ou `value` (pour une feuille). Cette encapsulation sépare clairement la construction de l'objet de la logique de parsing.

Destructeur récursif
Le destructeur libère récursivement tout le sous-arbre : en supprimant `left` et `right`, il déclenche automatiquement leurs propres destructeurs, qui à leur tour libèrent leurs enfants. Cette cascade assure la **libération complète et sécurisée** de l'ensemble de l'arbre à partir de la racine. Il est crucial de tester `(left != NULL)` et `(right != NULL)` comme conditions d'arrêt dans le destructeur récursif. Sans ces

gardes, la suppression d'un nœud feuille (dont les enfants sont `NULL`) déclencherait un appel récursif infini : `delete left` sur un pointeur `NULL` ne provoque pas d'erreur en C++ (c'est une opération définie qui ne fait rien), mais l'absence de test entraînerait une descente sans fin dans l'arbre - chaque appel au destructeur invoquerait à nouveau le destructeur sur des pointeurs nuls, accumulant des appels sur la stack jusqu'à provoquer un **dépassement de pile** (*stack overflow*) et un plantage du programme. Les tests `if (left != NULL)` brisent explicitement la récursion aux feuilles, garantissant que la libération s'arrête exactement là où l'arbre se termine. Cette discipline est indispensable pour tout type de données récursif utilisant des pointeurs.

`Node::print_tree`

La méthode `print_tree` effectue un **parcours infixe** de l'arbre : elle visite d'abord le sous-arbre gauche, puis affiche les informations du nœud courant (`value`, `feature`, `threshold`), puis visite le sous-arbre droit. Les tests `if (cur->left != NULL)` et `if (cur->right != NULL)` servent de conditions d'arrêt implicites : lorsqu'un enfant est `NULL` (feuille), la récursion s'arrête naturellement pour cette branche.
La récursivité - où une fonction s'appelle elle-même sur des sous-problèmes plus petits - est un concept que nous allons réviser dans une autre leçon.

read_tree

```
Node* read_tree(const char* filename) {
   // ouvrir le fichier et déclarer les variables...
   Node* root = NULL;
   while (fp.getline(line, sizeof(line))) {
       // parse line …
       // Dans la première itération, construire la racine
       if(root == nullptr){
           if(node_id == 1)
               root= new Node(cond_val, is_leaf);
           else {
               cerr<<"Tree root is missing. First line in file must have node_id= 0"
               fp.close();
               return nullptr; // ou gérer l'erreur
       }   }
       else{
           Node* parent= find_parent(root, node_id);
           if(node_id % 2== 0) // enfant gauche
               parent->set_left(new Node(cond_val, is_leaf));
           else // enfant droite
               parent->set_right(new Node(cond_val, is_leaf));
   }   }
   fp.close();
   return root; // L'appelant est désormais propriétaire de ce pointeur.
```


ULB

estimate

```
// Fonction d'inférence utilisant une structure arborescente
float estimate(Node* root, float features[FEATURE_COUNT]) {
   Node* cur=root; // commencer à la racine

   while (cur != nullptr) {
       float res= cur->eval_condition(features);
       if(res== -1)         // valeur réservée pour aller vers gauche
           cur= cur->get_left();
       else if (res== 0)    // valeur réservée pour aller vers droite
           cur= cur->get_right();
       else                 // le régresseur a conclu
           return res;
   }

   return 0.0; // fallback
}
```

find_parent

```
// Utilise la structure d'arbre binaire équilibré pour localiser le pointeur d'un index de nœud donné
Node* find_parent(Node* root, int node_idx) {
   if (node_idx == 1) return 0; //racine;
   Node* cur = root, *prev;
   int level= floor(log2(node_idx));
   // Construire le chemin de la racine à node_idx en examinant la représentation binaire
   int mask= pow(2, level-1); // Le masquage commence après le 1 initial (qui correspond à la racine).
   while (mask > 0) {
       prev= cur;
       if (node_idx & mask) { // obtenir le chiffre
           cur = cur->get_right(); // vers droite
       } else {
           cur = cur->get_left() ; // vers gauche
       }
       mask >>= 1; // Décaler d'une position vers la droite pour identifier l'enfant suivant
   }
   return prev;
}
```

Bitwise Operations

```
// 1. Décalage gauche/droite : multiplier par des puissances de deux
int x = 5, y;   // 00000101
y= x << 1; // 000001010, 10
y= x << 2; // 000010100, 20

y >>= 1; // 00001010, 10
y >>= 2; // 000010, 2

// 3. Bitwise AND: masking
int flags = 0b11010110; // binary literal
int mask  = 0b00001111;
cout <<  (flags & mask) ; // 6

// 4. Bitwise OR (set bits) and XOR (toggle bits)
int reg = 0b00000001;
reg |= (1 << 3);     // set bit 3
cout<< reg;         // 9
```


ULB

Les opérations bit à bit permettent de manipuler directement la représentation binaire des entiers, offrant une efficacité maximale pour certaines tâches de calcul ou de configuration. Le décalage à gauche (`<<`) et à droite (`>>`) constitue l'une des opérations les plus intuitives : décaler les bits d'un entier vers la gauche revient à multiplier ce nombre par une puissance de deux, tandis que le décalage vers la droite équivaut à une division entière par une puissance de deux. Ainsi, avec `x = 5` (représenté en binaire par `00000101`), l'expression `x << 1` décale tous les bits d'une position vers la gauche, produisant `00001010`, soit la valeur décimale 10 (5 × 2). De même, `x << 2` donne 20 (5 × 4). Inversement, appliquer `y >>= 1` à une valeur `y = 20` (`00010100`) ramène à 10 (`00001010`), puis `y >>= 2` aboutit à 2 (`00000010`), illustrant la division successive par 2 et 4.

L'opérateur AND bit à bit (`&`) sert principalement à extraire ou isoler des bits spécifiques grâce à un masque. Dans l'exemple, la variable `flags` contient la valeur binaire `11010110` (214 en décimal). En appliquant un masque `00001111` via `flags & mask`, seuls les quatre bits de poids faible sont conservés (`0110`), les autres étant forcés à zéro - le résultat est donc 6. Cette technique, appelée *masquage*, est couramment utilisée pour lire des drapeaux ou extraire des champs dans des registres matériels ou des structures de données compactes.

L'opérateur OR bit à bit (`|`) permet quant à lui d'activer délibérément certains bits sans altérer les autres. Partant d'un registre `reg` initialisé à `00000001`

(valeur 1), l'instruction `reg |= (1 << 3)` crée d'abord un masque avec un `1` à la position 3 (`00001000`), puis combine ce masque avec le registre via OR, produisant `00001001`, soit la valeur décimale 9. Le bit 3 est ainsi activé tout en préservant l'état initial du bit 0. Cette approche est fondamentale dans la configuration de périphériques ou la gestion de permissions, où chaque bit représente un paramètre indépendant.

Ces opérations, exécutées en un seul cycle processeur sur la plupart des architectures, offrent une performance inégalée pour les traitements intensifs. Leur maîtrise permet d'écrire du code à la fois compact et extrêmement efficace.

The main function

```
int main() {
   Node* root = read_tree('bmd_tree_transition.txt'); // dynamic allocation
   float age, weight_kg, height_cm, waiting_time;
   char choice;
   do {
       // read input from user
       ...
       float features[FEATURE_COUNT] = {weight_kg, age, height_cm, waiting_time};
       float bmd = estimate(root, features);
       cout << "\n--> Predicted BMD: " << bmd << "\n\n";

       cout << "Estimate another patient? (y/n): ";
       cin >> choice;
   } while (choice=='y');

   delete root; // Désallocation propre et sans fuites de mémoire
   return 0;
}
```