



INFOH2001 Object Oriented Programming

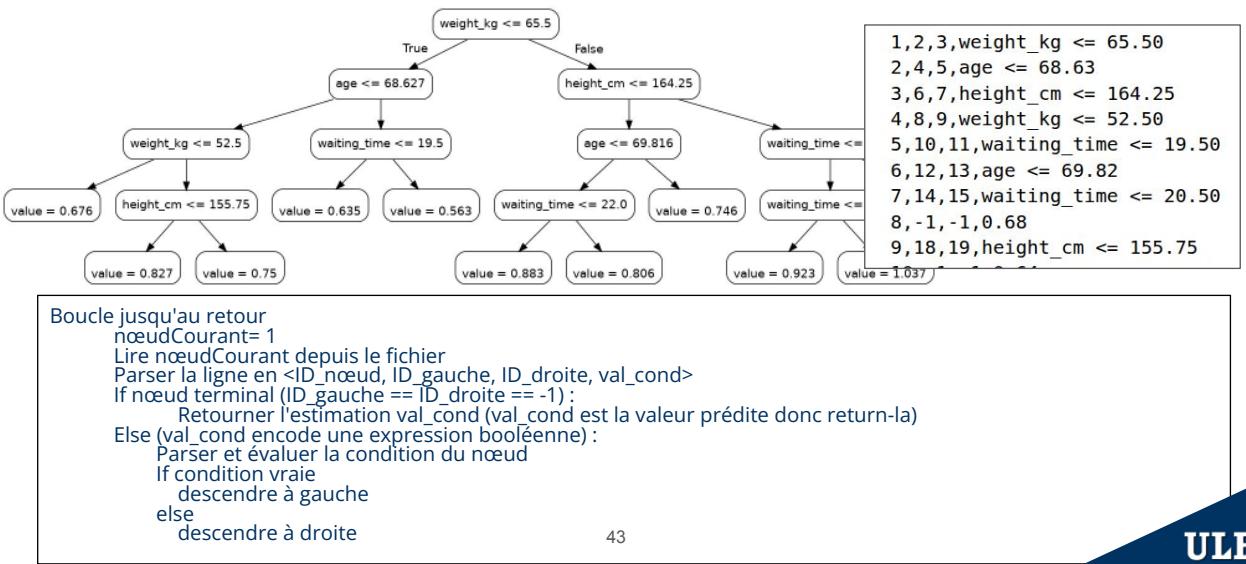
Mahmoud SAKR <mahmoud.sakr@ulb.be>

École polytechnique de Bruxelles

2025/26

Lecture 2

BMD Regression - v2.0 - Conception de l'Algorithme



43

ULB

La version développée pendant le cours précédent présentait un caractère très statique: l'arbre de décision était codé sous la forme d'une longue cascade d'instructions if-then-else imbriquées. Dès lors que la structure de l'arbre évoluait, il devenait nécessaire de modifier manuellement le code source, puis de recompiler l'ensemble pour produire un nouvel exécutable capable d'utiliser la nouvelle version de l'arbre pour la régression.

Une telle approche s'avère incompatible avec les exigences d'un outil de Machine Learning généraliste, dont le fonctionnement repose sur deux phases distinctes : l'apprentissage d'un modèle à partir de données, puis son déploiement pour l'inférence. Or, dans un contexte réel, la structure de l'arbre peut varier considérablement - selon le jeu de données utilisé, les hyperparamètres choisis ou encore les objectifs de modélisation - alors que le code de déploiement doit rester générique et réutilisable. Il est donc impératif de privilégier une approche dynamique, contrairement à la méthode statique adoptée en v1.0.

Dans ce cours, nous développerons la version 2.0 de notre projet, conçue précisément pour répondre à ce besoin. L'idée centrale consiste à lire dynamiquement la structure de l'arbre depuis un fichier externe au moment de l'inférence. Ainsi, la phase d'entraînement du modèle produira un fichier décrivant l'arbre appris; ce fichier sera ensuite chargé à l'exécution par notre programme d'inférence pour effectuer la régression.

Plusieurs stratégies d'implémentation sont envisageables; celle que nous

mettrons en œuvre durant ce cours est illustrée par l'algorithme présenté dans le slide accompagnant ces notes.

Function main()

```
int main() {  
    ...  
    do {  
        cout << "Enter patient details:\n";  
        // Input with error handling  
        cout << "Age (years): ";  
        cin >> age;  
        ...  
        float bmd = estimate(age, weight_kg, height_cm, waiting_time);  
        std::cout << "\n--> Predicted BMD: " << bmd << "\n\n";  
        // Ask to continue  
        std::cout << "Estimate another patient? (y/n): ";  
        std::cin >> choice;  
        std::cout << "\n";  
  
    } while (choice == 'y' || choice == 'Y');  
    ...  
}
```

La fonction main() reste identique à la version précédente: une boucle interactive qui demande les données de patients successifs jusqu'à ce que l'utilisateur vaut arrêter. L'inférence (régression/estimation) est réalisée par un appel à la fonction estimate.

Lire l'Arbre à Partir du Fichier

```
const char* filename = "bmd_tree_transitions.txt";
float estimate(float age, float weight_kg, float height_cm, float waiting_time) {
    ifstream tree(filename);
    if (!tree.is_open())
        return 0.0; // ou gérer l'erreur
    int current = 1; // ID du nœud racine - première ligne du fichier
    char line[256];
    while (1) {
        tree.seekg(0); // Relisez le fichier depuis le début
        while (!tree.eof()) {
            tree.getline(line, 256);
            // Extraire l'identifiant du nœud de la ligne (avant la première virgule)
            int node_id;
            sscanf(line, "%d,", &node_id);

            if (node_id == current) {
                struct ParseResult res = parse_eval_line(
                    line, weight_kg, age, height_cm, waiting_time);
                if (res.is_leaf) {
                    tree.close();
                    return res.value;
                } else
                    current = res.next_node;
            }
        }
    }
}
```

38

ULB

La solution adoptée dans la version 2.0 consiste à externaliser la description de l'arbre dans un fichier texte (`bmd_tree_transitions.txt`). La fonction `estimate` est ainsi repensée pour charger dynamiquement ce fichier à chaque inférence. Son fonctionnement suit ces étapes principales :

- Ouverture du fichier dont le nom est défini comme constante globale de type string (`const char*`) (strings sera expliqué dans ce cours)
- Initialisation du parcours à la racine de l'arbre (`current = 1`)
- Recherche linéaire du nœud courant dans le fichier (à l'aide de `seekg(0)` pour repartir du début à chaque itération)
- Analyse et évaluation de la condition du nœud via la fonction auxiliaire `parse_eval_line`, qui reçoit les caractéristiques du patient (âge, poids, taille, temps d'attente) (`parse_eval_line` sera vu dans ce cours)
- Arrêt et retour de la valeur prédite lorsque le nœud est une feuille (`is_leaf = true`)
- Sinon, descente vers l'enfant gauche ou droit selon le résultat de l'évaluation (`current = res.next_node`)

Cette architecture permet à la phase d'entraînement d'écrire simplement l'arbre appris dans un fichier, tandis que le même exécutable peut effectuer l'inférence pour n'importe quelle structure d'arbre

String Parsing - sscanf

```
// Parse une ligne au format: nœud, gauche, droite, valeur_ou_condition
ParseResult parse_eval_line(
    char* line, float weight_kg, float age, float height_cm, float waiting_time) {
    struct ParseResult res = {0};
    int node_id, left_id, right_id;
    char cond_val[128];
    // Parse: "0,1,6,weight_kg <= 65.0"
    // Utilisez %[^,] pour lire jusqu'à la virgule ou le saut de ligne
    if (sscanf(line, "%d,%d,%d,%127[^\\n]", &node_id, &left_id, &right_id, cond_val) != 4)
    {
        res.is_leaf = true;
        res.value = 0.0;
        return res;
    }
    if (left_id == -1 && right_id == -1) {
        // Nœud feuille : cond_val est la valeur numérique estimée
        res.is_leaf = 1;
        res.value = atof(cond_val);
    } else {
        // Nœud interne
        res.is_leaf = false;
        int take_left = eval_condition(cond_val, weight_kg, age, height_cm, waiting_time);
        res.next_node = take_left ? left_id : right_id;
    }
    return res;
}
```

```
struct ParseResult {
    int is_leaf; // 1 = leaf, 0 = internal
    double value; // if leaf
    int next_node; // if internal
};
```

47

ULB

La fonction `parse_eval_line` analyse une ligne du fichier décrivant un nœud de l'arbre, au format `nœud, gauche, droite, cond_ou_valeur`.

- Elle extrait les quatre champs via `sscanf` avec le format `%d,%d,%d,%127[^\\n]` pour capturer la condition ou la valeur (jusqu'à 127 caractères). (`sscanf` sera expliqué dans ce cours)
- **Détection de feuille** : si `gauche == droite == -1`, le champ `cond_val` contient une valeur numérique convertie via `atof()` → `res.is_leaf = true` et `res.value` est renseigné.
- **Nœud interne** : `cond_val` encode une expression booléenne évaluée par `eval_condition()` à l'aide des caractéristiques du patient (poids, âge, taille, temps d'attente).
- La descente dans l'arbre utilise **l'opérateur conditionnel ternaire (?)** pour sélectionner dynamiquement le prochain nœud :
`res.next_node = take_left ? left_id : right_id;`
équivalent compact d'un `if-else`
`if(a > b) { c = a; } else { c = b; }`
- En cas d'échec de parsing (moins de 4 champs lus), la fonction retourne une feuille par défaut avec `value = 0.0`.

La fonction `parse_eval_line` retourne un **type composite** : une structure nommée `ParseResult`. En C++, une `struct` permet de regrouper plusieurs champs hétérogènes dans un seul objet, facilitant ainsi le retour de plusieurs valeurs corrélées depuis une fonction. (struct sera expliqué dans ce cours)

Cette structure agit comme un conteneur de résultat à deux usages mutuellement exclusifs :

- **Nœud feuille** : `is_leaf = 1` et `value` contient l'estimation finale
- **Nœud interne** : `is_leaf = 0` et `next_node` indique l'identifiant du nœud enfant à visiter

L'utilisation d'une structure évite les paramètres de sortie multiples ou les variables globales, rendant l'interface de la fonction plus propre et le code d'inférence plus lisible.

String Parsing - istringstream

```
#include <sstream>
int eval_condition(char* cond,
    float weight_kg, float age, float height_cm, float waiting_time) {
    istringstream iss(cond); // Connectez le stream à la C-String
    string feat, op;
    float threshold;
    if (!(iss >> feat >> op >> threshold)) {
        return 0; // ou gérer l'erreur - parsing a échoué
    cout<<"Evaluating " << cond<< endl;
    float feat_val = 0.0;
    if (feat == "weight_kg")      feat_val = weight_kg;
    else if (feat == "age")       feat_val = age;
    else if (feat == "height_cm") feat_val = height_cm;
    else if (feat == "waiting_time") feat_val = waiting_time;
    else return 0;

    if(op== "<=")   return (feat_val <= threshold) ? 1 : 0;
    else if(op== "<")   return (feat_val < threshold) ? 1 : 0;
    else if(op== "=")   return (feat_val == threshold) ? 1 : 0;
    else if(op== ">=")  return (feat_val >= threshold) ? 1 : 0;
    else if(op== ">")   return (feat_val > threshold) ? 1 : 0;
    else return 0;
}
```

40

ULB

La fonction `eval_condition` évalue dynamiquement une condition stockée sous forme de string (ex., "weight_kg <= 65.50") à l'aide des caractéristiques réelles du patient.

- Elle utilise un `istringstream` pour découper la condition en trois composantes: nom de la variable (`feat`), opérateur (`op`), et seuil numérique (`threshold`). (`istringstream` sera expliqué dans ce cours)
- En cas d'échec de parsing (moins de trois éléments extraits), elle retourne `0` (faux).
- La valeur réelle de la caractéristique est récupérée par comparaison du nom (`feat`) avec les paramètres d'entrée (`weight_kg`, `age`, `height_cm`, `waiting_time`); une caractéristique inconnue entraîne un retour à `0`.
- L'opérateur est ensuite interprété (`<=`, `<`, `=`, `>=`, `>`) et appliqué pour comparer la valeur réelle au seuil.
- Le résultat est retourné sous forme d'entier: `1` si la condition est vraie donc brancher vers gauche, `0` sinon donc brancher vers droite
- Une instruction `cout` affiche la condition évaluée (utile pour le débogage pendant le développement).

C++ File I/O

C++ propose les classes suivantes pour lire et écrire des caractères dans des fichiers :

- `ofstream` : classe pour écrire dans un fichier
- `ifstream` : classe pour lire un fichier
- `fstream` : classe pour lire et écrire dans un fichier

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    return 0;
}
```

```
int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( getline (myfile,line) )
            cout << line << '\n';
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

50

ULB

Travailler avec des fichiers grâce à la classe `fstream`

Travailler avec des fichiers ressemble à travailler avec l'entrée et la sortie standard. Les classes `ifstream`, `ofstream` et `fstream` sont dérivées respectivement de `istream`, `ostream` et `iostream`. En tant que classes filles, elles héritent des opérations d'insertion (`<<`) et d'extraction (`>>`), ainsi que des autres fonctions membres, tout en ajoutant leurs propres méthodes pour gérer les fichiers. Pour les utiliser, incluez `<fstream>`.

- Utilisez `ifstream` pour lire un fichier.
- Utilisez `ofstream` pour écrire dans un fichier.
- Utilisez `fstream` pour lire et écrire dans le même fichier.

Donnez le nom du fichier comme argument du constructeur.

Exemple : écriture dans un fichier

```
ofstream myfile;
```

```
myfile.open("example.txt");
myfile << "Writing this to a file.\n";
myfile.close();
```

Ce code crée un stream (flux) de sortie `myfile`, ouvre explicitement le fichier "example.txt" en écriture (`ios::out` par défaut), insère une phrase avec l'opérateur `<<`, puis ferme le fichier. La fermeture est essentielle : elle vide le tampon mémoire et libère immédiatement le fichier, permettant à d'autres

programmes d'y accéder sans conflit - un point crucial en environnement partagé ou concurrent.

Exemple : lecture ligne par ligne

```
string line;  
  
ifstream myfile("example.txt");  
if (myfile.is_open()) {  
    while (getline(myfile, line))  
        cout << line << '\n';  
    myfile.close();  
}  
else cout << "Unable to open file";
```

Ici, le fichier est ouvert directement dans le constructeur en mode lecture (`ios::in`). La méthode `getline()` lit chaque ligne complète (y compris les espaces), contrairement à `>>` qui s'arrête aux espaces. Le test `is_open()` garantit que le fichier existe et est accessible avant toute tentative de lecture. Là encore, `close()` est appelé pour libérer proprement la ressource.

Ouvrir un fichier

En C++, on ouvre un fichier soit directement dans le constructeur (`ofstream f("fichier.txt")`), soit explicitement après déclaration avec `f.open("fichier.txt")`. La première méthode est plus simple et sûre. Dans les deux cas, vérifiez toujours si l'ouverture a réussi avec `if (!f)`.

Le mode d'ouverture est construit en combinant des options avec `|` (« ou » logique). Ces options viennent du type `open_mode` :

```
enum open_mode { binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,  
    nocreate=0x20, noreplace=0x40 };
```

ouvrir un fichier en lecture et écriture :

```
fstream inoutFile("someName", ios::in | ios::out);
```

Lire et écrire avec `<<` et `>>`

Une fois ouvert, utilisez `<<` pour écrire dans un fichier de sortie (`ofstream`) et `>>` pour lire depuis un fichier d'entrée (`ifstream`). Ces opérateurs fonctionnent comme avec `cout` et `cin` : `f << "texte"` écrit, `f >> variable` lit. Ils gèrent automatiquement les types de base (nombres, strings).

Fermer les fichiers

Fermez toujours un fichier avec `f.close()` dès que vous avez fini de l'utiliser. Cela vide les tampons mémoire (pour sauvegarder les données) et libère le

fichier immédiatement. En environnement partagé, un fichier non fermé reste verrouillé : d'autres programmes ne peuvent pas y accéder, ce qui crée des conflits. La fermeture automatique à la fin de la portée de l'objet est possible, mais fermer explicitement reste plus sûr en cas de concurrence.

Streams & Modes de Transfert des Données

- Un stream (flux) est une idée simple: c'est un courant de bytes qui circule entre votre programme et un appareil externe (fichier, clavier, écran, etc.).
- Toutes les entrées et sorties utilisent des bytes. Mais comment les lire ou les écrire dépend du contexte : le programmeur doit connaître le format des données.
- Il existe deux modes de transfert :
 - Mode texte
 - Les données sont vues comme du texte, organisées en lignes (séparées par '\n').
 - On utilise << pour écrire et >> pour lire.
 - Le système peut faire des changements automatiques : par exemple sous Windows, '\n' devient '\r\n'.
 - Mode binaire
 - Les bytes sont copiés tels quels, sans modification.
 - Ce qu'on écrit en mémoire est sauvegardé exactement pareil sur le disque (un entier de 4 bytes prend 4 bytes).
 - Aucune information supplémentaire n'est ajoutée : le programmeur doit savoir comment les données sont organisées pour les relire correctement.

53

ULB

Flux et modes de transfert des données

Un flux (*stream*) est une abstraction pour les entrées-sorties séquentielles : c'est un flot d'octets entre votre programme et un périphérique externe (fichier, clavier, écran, etc.). Toutes les opérations d'entrée-sortie manipulent des octets, mais leur interprétation dépend du contexte : c'est au programmeur de connaître le format des données échangées.

Il existe deux modes de transfert :

Mode texte

Les données sont interprétées comme des caractères, organisées en lignes séparées par le symbole '\n'. On utilise les opérateurs formatés << et >> pour lire et écrire. Le système peut effectuer des transformations automatiques selon la plateforme : par exemple sous Windows, le retour à la ligne '\n' est converti en '\r\n' (retour chariot + saut de ligne) lors de l'écriture.

Mode binaire

Les octets sont transférés bruts, sans aucune conversion. L'écriture et la lecture sont non formatées : la représentation exacte en mémoire est préservée (par exemple un entier sur 4 octets occupe bien 4 octets sur le disque). Aucune métadonnée n'est ajoutée — le programmeur doit impérativement connaître le format et l'ordre des données pour les relire correctement.

Par exemple, si vous écrivez l'entier 1000000 dans un fichier en mode texte,

cela produira 7 caractères ('1','0','0','0','0','0','0') et occupera donc 7 bytes sur le disque.

En revanche, si vous écrivez le même entier 1000000 en mode binaire, les 4 bytes qui représentent cet entier en mémoire seront copiés tels quels dans le fichier, n'occupant ainsi que 4 bytes sur le disque. Si vous ouvrez ensuite ce fichier avec un éditeur de texte, vous ne verrez pas « 1000000 » ; vous verrez des caractères illisibles ou des symboles étranges (données binaires brutes). Ce n'est qu'en relisant le fichier avec un programme C++ (en mode binaire) que ces octets seront correctement interprétés à nouveau comme la valeur 1000000.

À retenir :

- Utilisez le mode texte pour les fichiers lisibles par l'homme (texte, CSV, etc.).
- Utilisez le mode binaire pour les données structurées (images, fichiers de données brutes, sérialisation) où la fidélité byte par byte est cruciale.

Gestion Normalisée des Streams

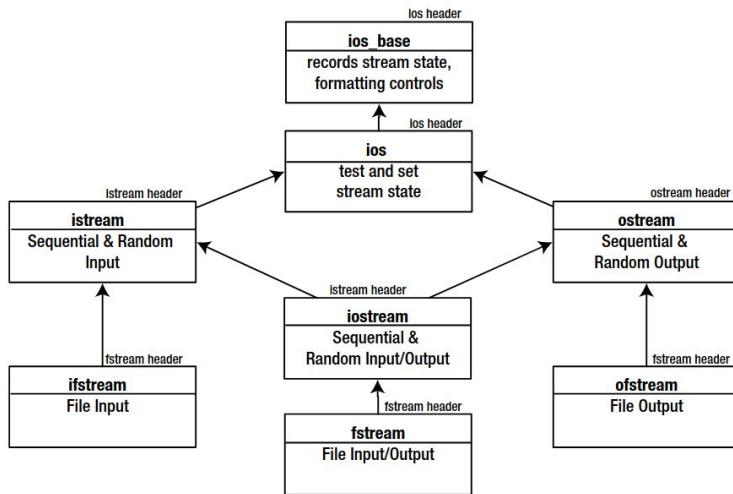


Figure 17-2. The main classes that represent streams

Beginning C++, Ivor Horton

ULB

Les classes de flux en C++ – Explication simple

En C++, les entrées et sorties utilisent une famille de classes toutes partagent des fonctionnalités communes.

À la base se trouve la classe **ios**, qui gère l'**état du flux** (est-il ouvert ? y a-t-il une erreur ?) et les **options de formatage** (afficher les nombres en décimal, hexadécimal, etc.). Tous les flux héritent de ces caractéristiques.

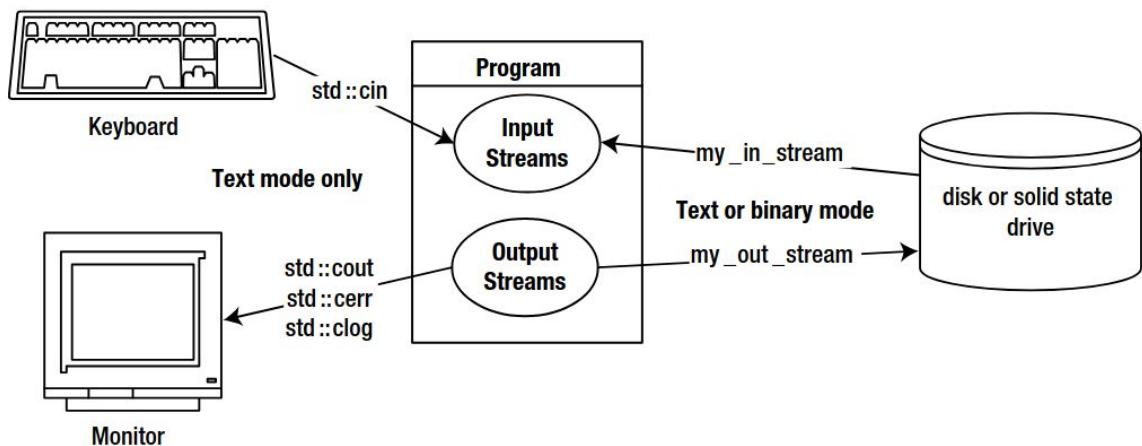
Au-dessus :

- **istream** gère la **lecture** (entrée) avec l'opérateur **>>**.
- **ostream** gère la **écriture** (sortie) avec l'opérateur **<<**.
- **iostream** combine les deux pour lire et écrire.

Les flux de fichiers fonctionnent exactement de la même façon grâce à l'héritage :

- **ifstream <- istream** : pour **lire** un fichier (comme **cin**, mais depuis un fichier).
- **ofstream <- ostream** : pour **écrire** dans un fichier (comme **cout**, mais vers un fichier).
- **fstream <- iostream** : pour **lire et écrire** dans le même fichier.

Gestion Normalisée des Streams



Pourquoi c'est pratique ?

Parce que l'héritage garantit que les mêmes opérateurs (`<<`, `>>`) et méthodes fonctionnent partout :

```
cin >> x;      // Lecture au clavier  
fichier >> x;  // Lecture dans un fichier (ifstream)
```

```
cout << y;      // Écriture à l'écran  
fichier << y;   // Écriture dans un fichier (ofstream)
```

Les flux standards:

- `cin` est un objet de type `istream` (lecture au clavier).
- `cout`, `cerr` et `clog` sont des objets de type `ostream` (écriture à l'écran).

Manipulateurs de stream - Exemples

```
int main(){
    //Setting floating-point precision (default vs. fixed)
    double pi = 3.14159;
    cout << setprecision(3) << pi << endl;           // Output: 3.14
    cout << fixed << setprecision(3) << pi << endl; // Output: 3.142

    //Controlling field width and fill character:
    cout << setw(5) << setfill('.') << 42 << endl; // Output: ...42

    //Left- vs. right-justified output
    cout << left << setw(6) << setfill('.') << 100 << "end" << endl; // Output: 100...end
    cout << right << setw(6) << setfill('.') << 100 << "end" << endl; // Output: ...100end

    //Scientific notation with controlled precision
    double val = 123.456;
    cout << scientific << setprecision(2) << val << endl; // Output: 1.23e+02
}
```

57

ULB

Manipulateurs de flux en C++

`std::setprecision(n)`

Fixe la précision des nombres à virgule flottante. Par défaut, `n` indique le nombre total de chiffres affichés. En mode `fixed` ou `scientific`, `n` devient le nombre de chiffres *après* la virgule. Ce réglage reste actif jusqu'à changement.

`std::setw(n)`

Définit la largeur du champ d'affichage à `n` caractères, *uniquement pour la prochaine valeur*. Les affichages suivants reprennent leur largeur normale (ajustée au contenu).

`std::setfill(ch)`

Remplace le caractère de remplissage (espace par défaut) par `ch` lorsque la valeur affichée est plus courte que la largeur du champ (`setw`). Ce réglage reste actif pour les sorties suivantes.

`std::fixed`

Affiche les nombres à virgule en notation décimale fixe (ex: `3.141590` au lieu de `3.14159e+00`).

`std::scientific`

Force la notation scientifique (ex: `3.141590e+00`). Un chiffre avant la virgule, suivi de l'exposant.

`std::defaultfloat`

Revient au format flottant par défaut (ni `fixed` ni `scientific`).

`std::dec, std::hex, std::oct`

Définissent la base d'affichage des entiers : décimal (par défaut), hexadécimal ou octal. Restent actifs jusqu'à changement.

`std::showbase / std::noshowbase`

`showbase` ajoute le préfixe de base (`0x` pour l'hexa, `0` pour l'octal). `noshowbase` le supprime.

`std::left, std::right, std::internal`

Alignement dans le champ défini par `setw` :

- `left` : valeur à gauche, remplissage à droite.
- `right` : valeur à droite (défaut), remplissage à gauche.
- `internal` : signe ou base à gauche, chiffres à droite, remplissage entre les deux (ex: `- 123`).

`std::endl`

Insère un retour à la ligne (`\n`) et vide immédiatement le tampon mémoire vers le périphérique (écran, fichier...).

`std::flush`

Vide uniquement le tampon mémoire sans insérer de retour à la ligne.

Exemple : `cout << "Traitement..." << flush;` -> affiche immédiatement « Traitement... » sans sauter de ligne.

C++ Array

- C++ Array est une collection de deux cellules mémoire contiguës ou plus, appelées éléments du array
- Tous les éléments partagent un même nom symbolique et un type unique
- Déclaration d'un tableau unidimensionnel :

```
type nom[taille];           // non initialisé
type nom[] = { liste_valeurs }; // initialisé, la taille est automatiquement déterminée à partir de la liste_of_values

    ○ type : n'importe quel type de données (ex. : int, float, char)
    ○ nom : identifiant du tableau
    ○ taille : nombre d'éléments (constante entière connue à la compilation)
    ○ liste_valeurs : valeurs initiales entre accolades (optionnelles)
```

```
float grades[5];
//int primes[5] = {1, 2, 3, 5, 7};
int primes[] = {1, 2, 3, 5, 7};
for(int i=0; i<5; i++)
    cout << primes[i] << '\t';
```

46

ULB

Array en C++

- Collection de deux cellules mémoire contiguës ou plus, appelées *éléments du tableau*
- Tous les éléments partagent un même nom symbolique et un type unique
- Déclaration d'un tableau unidimensionnel :

C string (char[], char *)

- **Character Array:**

- array simple de valeurs char
- Exemples:

- `char vowels[5] {'a', 'e', 'i', 'o', 'u'}; // 5 chars, Pas une string`
- `cout << vowels; // Fonctionnement non garanti`

- **C-String (char[] with \0)**

- Array de caractères terminé par un caractère nul (\0).
- Interprété comme une string par les fonctions d'entrée/sortie et les fonctions de la bibliothèque C (par exemple, 'cout', 'strlen').
- Peut être initialisé à partir d'une string littérale (le compilateur ajoute '\0').
- Exemples:

- `char name[] = "Mae"; // 4 elements: 'M','a','e','\0'`
- `char msg[10] = "Hi"; // 'H','i','\0', puis 7 inutilisés (initialisés à zéro)`
- `cout << name; // Imprime « Mae » (s'arrête à '\0')`

60

ULB

Une C-string est définie comme un array de caractères *explicitement terminé par un octet nul*. Ce marqueur '\0' constitue la convention fondamentale du langage C pour délimiter la fin logique d'une séquence textuelle. Lorsqu'une initialisation utilise un littéral string (entourée dans "") comme dans `char name[] = "Mae";`, le compilateur alloue automatiquement un array de quatre éléments (`sizeof("Mae") == 4`) contenant les bytes 'M' (77), 'a' (97), 'e' (101) et '\0' (0).

De même, `char msg[10] = "Hi";` produit un array de dix bytes où les positions 0 et 1 contiennent 'H' et 'i', la position 2 contient '\0', et les positions 3 à 9 sont initialisées à zéro conformément à la règle d'initialisation partielle du C++. Cette structure mémoire permet aux fonctions de la bibliothèque standard C - telles que `strlen()`, `strcpy()` ou `printf()` - ainsi qu'à l'opérateur `<<` de `std::ostream`, de déterminer la fin de le string en temps linéaire sans mét-information supplémentaires. `strlen(name)` retourne 3 car il compte les bytes jusqu'au premier '\0' (non inclus), tandis que `cout << name` produit exactement la séquence "Mae" suivie d'un retour à la ligne, sans corruption mémoire.

L'absence du terminateur nul '\0' rompt cette convention et invalide toute opération textuelle standard. Une fonction comme `strlen(vowels)` entrerait dans une boucle infinie ou retournerait une valeur erronée, car elle incrémenterait indéfiniment jusqu'à rencontrer un zéro par hasard en mémoire. De même, `strcpy(dest, vowels)` copierait des bytes indéfinis au-delà des cinq

caractères prévus. Cette fragilité structurelle explique pourquoi les C-strings exigent une discipline stricte: toute modification manuelle du contenu (par exemple via une boucle d'affectation) doit impérativement préserver ou réinsérer le '\0' final.

En C++, `std::string` élimine ces risques en encapsulant la gestion du tampon, en stockant explicitement la longueur et en garantissant la null-termination interne lorsque nécessaire - mais comprendre le modèle sous-jacent des C-strings reste essentiel pour interagir avec les API système, les bibliothèques C héritées et les protocoles réseau bas niveau où les bytes bruts prévalent.

C++ string (string.h)

- std::string: classe de la bibliothèque standard (<string>), pas un array de char
- Séquence de caractères avec gestion automatique de la mémoire
- Pas de terminateur nul requis: la longueur est stockée en interne
- Conçue pour être sûre, efficace et facile d'usage
- Encapsule de nombreuses fonctions utiles : size(), substr(), [], insert(), replace(), find(), etc.

```
#include <string>

std::string name = "Mae West"; // Creates a string object
std::cout << name;           // Prints "Mae West"
```

48

ULB

Contrairement aux *C-strings*, qui sont de simples array de caractères terminés par \0, les C++ (`std::string`) constituent un type objet complet de la bibliothèque standard. Elles encapsulent la mémoire, la longueur et les opérations de manipulation, offrant ainsi sécurité, simplicité et expressivité accrues.

Une C++ String se déclare simplement avec `std::string nom;` ou `std::string nom{"valeur"}.` L'initialisation peut utiliser des littéraux, d'autres objets `string`, ou être laissée vide. L'accès aux caractères individuels s'effectue via l'opérateur [] (ex. : `nom[0]`), tandis que la concaténation utilise naturellement l'opérateur +. La longueur de la string est obtenue par la méthode `length()` ou `size()`. Ces objets gèrent automatiquement l'allocation mémoire, évitant les débordements de tampon fréquents avec les *C-strings*.

La comparaison de strings s'effectue intuitivement avec les opérateurs relationnels (<, <=, >, >=, ==, !=). Ces comparaisons sont lexicographiques, basées sur les codes ASCII des caractères, et sensibles à la casse. Lorsque deux strings partagent un préfixe commun, la plus courte est considérée comme inférieure. Par exemple, "age" < "beauty" retourne `true`. Cette propriété permet de trier facilement des collections de string.

La recherche dans une string s'appuie principalement sur `find()`, qui retourne la position de la première occurrence d'une sub-string ou d'un caractère, ou `string::npos` si rien n'est trouvé. Plusieurs surcharges permettent de spécifier une position de départ (`find("an", 3)`) ou de limiter le nombre de caractères recherchés dans un littéral C (`find("akat", 1, 2)` recherche "ak").

Stringstream - Traitement des string Comme Console et Fichiers

<sstream> permet de lire ou d'écrire dans des objets std::string en utilisant les mêmes opérations de iostream ; idéal pour 'parsing'.

```
#include <sstream>
#include <string>
int main(){
    std::string input = "100 3.14";
    std::istringstream inStr(input);
    // Connecter le stream àu string
    long value;
    double data;
    inStr >> value >> data;
    // Extraire les valeurs
    // value = 100, data = 3.14
}
```

```
#include <sstream>
std::ostringstream outStr;
double number = 2.5;
outStr<< "number = " << (number/2.0); //Écrivez comme cout
std::string result = outStr.str(); // Obtenez string
// result == "number = 1.25"
```

```
std::stringstream ss;
ss << "Count: " << 42;
ss << " Price: " << 19.99;
std::string word; int n; double p;
ss >> word >> n >> word >> p;
```

49

ULB

La bibliothèque **<sstream>** fournit des stream en mémoire permettant de lire ou d'écrire dans des objets **std::string** en utilisant les opérateurs familiers **>>** et **<<**, exactement comme avec **cin** ou **cout**. Cette approche facilite le *parsing* ou le formatage, ce qui la rend particulièrement adaptée au traitement de données structurées stockées sous forme textuelle - comme les conditions encodées dans les nœuds de notre arbre de décision ("weight_kg <= 65.5"). Trois classes principales composent cette fonctionnalité :

- **istringstream** : lecture depuis une string (extraction avec **>>**)
- **ostringstream** : écriture vers une string (insertion avec **<<**)
- **stringstream** : lecture et écriture bidirectionnelle dans la même string

C struct

- Un struct regroupe des variables de types différents (int, string, bool, etc.) dans une seule unité nommée
- Accès aux membres via l'opérateur point (.) pour lecture ou modification
- Une struct nommée définit un type réutilisable (ex., ParseResult)
- Disposition mémoire : les membres sont stockés de façon contiguë
- Cas d'usage principal : retourner ou transmettre plusieurs valeurs corrélées depuis/vers une fonction

```
struct Car { // Named struct= new data type
    std::string brand;
    std::string model;
    int year;
};

Car car1, car2; // Multiple variables of type Car

car1.brand = "BMW"; car1.model = "X5"; car1.year =
1999;
car2.brand = "Ford"; car2.model = "Mustang";
car2.year = 1969;
```

Conversion entre Types en C++

C++ effectue automatiquement des conversions entre types compatibles lorsque nécessaire, par exemple de **int** vers **double** dans l'instruction `double x = 5;`.

Le cast style C s'écrit sous la forme `(type)expression` ou `type(expression)`, comme **(int)3.14** ou **int(3.14)**. Cette syntaxe peut être risquée car elle contourne les vérifications du compilateur et masque potentiellement des erreurs.

Une alternative plus sûre est `static_cast` : **int n = static_cast<int>(d);** - explicite, lisible et limitée aux conversions compatibles.

La conversion d'un type plus grand vers un type plus petit (par exemple `long` vers `char`) ou d'un flottant vers un entier peut entraîner une perte de données; **static_cast** rend cette opération intentionnelle et visible.

Recommandation générale : éviter les casts style C et préférer `static_cast`, plus explicite, facilement repérable dans le code et offrant une meilleure sécurité de typage, ce qui contribue à prévenir les bogues subtils.

Functions

```
int addition (int, int);

int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
}

int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
```

The result is 8

52

ULB

Prototype et signature de fonction

Un prototype (**type nom(params);**) déclare l'interface d'une fonction avant sa définition ou son utilisation. Il spécifie le type de retour, le nom et la liste typée des paramètres, permettant au compilateur de valider statiquement la cohérence des appels (arité et types). L'absence de paramètres se déclare avec les parenthèses vides () équivalent à **void**).

Passage de paramètres et retour de valeur

Les fonctions peuvent recevoir des paramètres. Le mot-clé **void** comme type de retour indique une fonction procédurale qui ne produit pas de valeur exploitable par l'appelant. À l'inverse, un type explicite (**int**, **double**, etc.) permet de transmettre un résultat via l'instruction **return**, qui transfère la valeur au contexte appelant.

Modularité et encapsulation

Les fonctions réalisent l'encapsulation logicielle : elles exposent une interface minimalistre (prototype) tout en cachant leur implémentation interne. Cette séparation permet la réutilisation de code, la réduction de la complexité cognitive et l'isolation des effets de bord. Une architecture bien modulaire forme un graphe d'appels hiérarchique où chaque fonction assume une responsabilité unique, facilitant la maintenance et les tests unitaires.