INFOH2001 Programmation Orientée Objet

Mahmoud SAKR <mahmoud.sakr@ulb.be>

École polytechnique de Bruxelles

2025/26

Lecture 1

Contenu du cours

Le cours a trois points principaux :

- Programmer en C++
- Le modèle de Programmation Orientée Objet POO
- Les bases de l'informatique

Le cours a un ton pratique, avec des projets et des exercices de programmation.

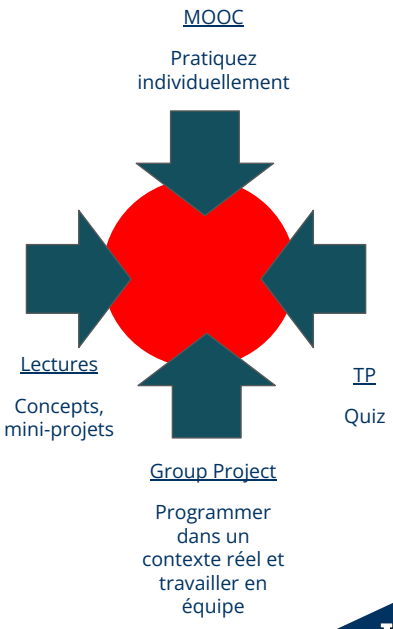
Organisation du cours

Cours magistral
- Une séance par semaine, suivie d'un TP associé la semaine suivante
- Présentation de nouveaux concepts (programmation, bases de l'informatique, conception orientée objet).
- Chaque séance inclut également un mini-projet illustrant les principaux concepts abordés, qui sera expliqué en classe (project-based learning)

MOOC
- Immédiatement après chaque cours, un nouveau module s'ouvrira sur le MOOC sur UV pour s'entraîner sur les thèmes vus en cours

Travaux Pratique
- Chaque TP prend la forme d'un quiz. Une semaine sur deux, il sera noté. Au total, 6 TP notés.
- Pas d'enseignement pendant le TP - uniquement un quiz.

Projet
- Projet de groupe de 4, pendant 6 semaines [~semaine 29-35]



ULB

Évaluation

Il n'y a **pas d'examen final en juin**. L'évaluation se déroule tout au long du semestre selon les modalités décrites ci-dessus - il s'agit d'une évaluation continue !

```
Note finale juin = 14/20 * Note_quiz + 4/20 * Note_projet + 2/20 * Note_oral
```


ULB

Deuxième session:

- Il n'y a pas de deuxième session pour le projet de groupe ni pour l'oral associé. Vos notes obtenues en première session sont définitives.
- Si votre note totale (quiz + projet + oral) en première session est inférieure à 10/20, vous ne pouvez repasser que la partie quiz (sur 14/20). La note du projet et de l'oral obtenue en première session est conservée.
- La deuxième session des quizzes se déroule sous la forme d'un examen de 3 heures sur UV, couvrant l'ensemble du programme du cours

Note finale août = 14/20 * Note_quizz_août + 4/20 * Note_projet + 2/20 * Note_oral

Évaluation - Note_quiz

- Quiz une séance de travaux pratiques sur deux, évalué à 70% (14/20)
- La note de l'ensemble des quiz est la moyenne résultats aux quiz
- **Durée:** Chaque quiz dure 1 heure
- **Contenu:** matériel vu depuis le début du cours jusqu'à la date du quiz, avec une attention accordée aux deux semaines précédentes
- **Format:** des questions QCM et des questions de programmation, portant sur les sujets abordés en cours
- Le quiz a lieu pendant le créneau habituel du TP, dans la salle de TP
- Les séries de quiz sont affichées sur UV.
- Le quiz se déroule entièrement sur UV, via votre compte ULB
- Vous pouvez utiliser **votre propre ordinateur portable ou un PC de la salle informatique**. Les tablettes ne sont pas autorisées.
- Le quiz ne fonctionne que sous **Windows** (compatibilité avec MacOS incertaine).

ULB

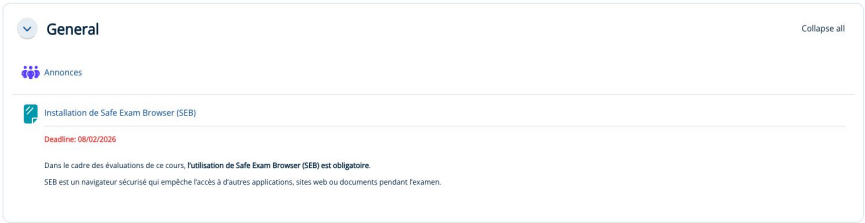
- En cas d'absence justifiée le jour d'un quiz, vous devez envoyer votre justificatif à l'assistant(e) du cours dès que possible, et au plus tard avant le TP de la semaine suivante. Un quiz de rattrapage vous sera alors organisé la semaine suivant celle du quiz manqué.
- En dehors des absences justifiées, aucun quiz individuel ne peut être repassé.

Safe Exam Browser

Dans ce cours, les quiz sur l'UV seront réalisées à l'aide de Safe Exam Browser (SEB) afin de garantir des conditions d'examen équitables pour tous.

- SEB est obligatoire pour les quiz
- L'installation doit être faite avant le **09/02/2026**
- Linux n'est pas supporté, utilisation des ordinateurs des salles informatiques si nécessaire

Pour plus de details, consultez la page du cours

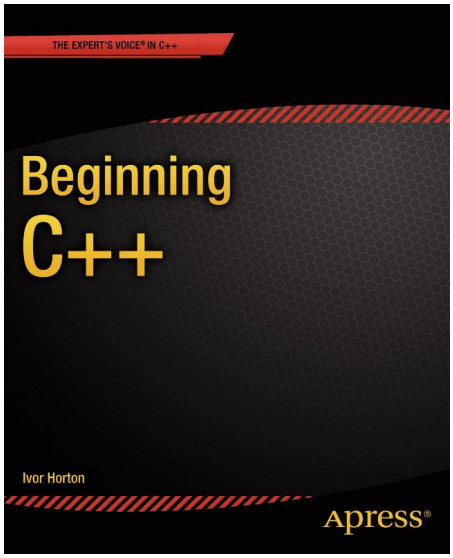


Évaluation - Note_projet, Note_oral

- Projet de groupe évalué à 20 % (4/20)
  - Groupe de quatre étudiants (ni 5, ni 3)
  - Le projet sera annoncé vers la semaine 29 et à remettre vers la semaine 35. Un calendrier plus détaillé suivra ultérieurement
  - Chaque groupe présentera son projet devant un jury et recevra une note collective.
- Examen oral sur le projet évalué à 10 % (2/20).
  - Chaque membre du groupe passera un examen oral individuel sur le projet. Oui, chaque étudiant doit connaître l'ensemble du projet et pourra être interrogé sur n'importe quelle partie de celui-ci
  - Cet examen oral ne portera pas sur l'ensemble du cours, mais uniquement sur le projet. Bien entendu, le projet met en œuvre des concepts abordés en cours, il y aura donc des points de recoupement

Cette présentation et cet examen oral auront lieu pendant la session d'examens de juin. Les dates seront annoncées plus tard.

Lectures Recommandées



ULB

Première source :

- Syllabus : diapositives + notes
- MOOC
- Mini-projets

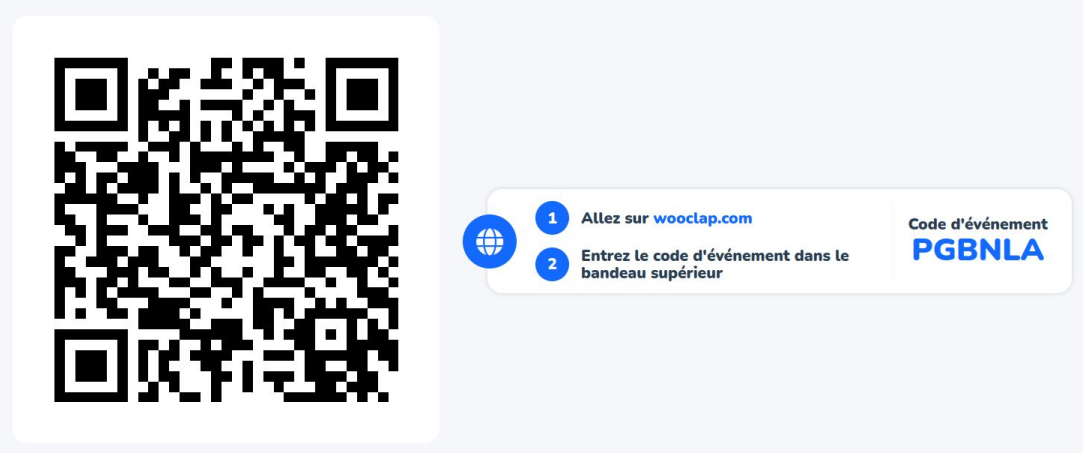
Pour des lectures supplémentaires, si quelque chose n'est pas clair :

- Beginning C++ – disponible en téléchargement gratuit sur Cible+.
- Ou toute autre ressource utile sur Internet.

Si vous avez besoin d'aide ou des questions :

- Posez-les au professeur ou à l'assistant (Gaspard Merten).
- Des assistants étudiants seront disponibles à partir de mars. Leur emploi du temps sera visible sur TimeEdit.

Test - Organisation du cours



ULB

Pourquoi C++ ?

PYPL PopularitY of Programming Language

Worldwide, Dec 2025 :

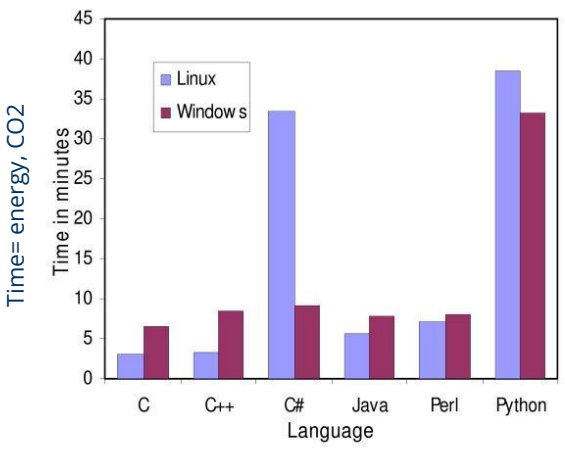
Rank	Change	Language	Share	1-year trend
1		Python	25.91 %	-3.9 %
2	↑↑	C/C++	13.02 %	+5.8 %
3	↑↑↑↑↑↑↑	Objective-C	11.37 %	+8.7 %
4	↓↓	Java	11.36 %	-4.0 %
5	↑	R	5.84 %	+1.2 %

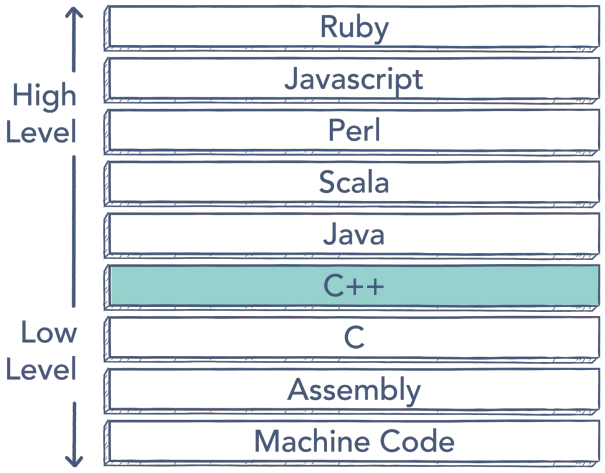
Logiciels développés en C++






Informatique verte





12

ULB

TYPES DE LANGAGES DE PROGRAMMATION

Il existe de nombreux langages pour programmer un ordinateur. Le plus basique est le *langage machine*: une série d'instructions très détaillées et difficiles à lire, qui contrôlent directement les circuits internes de l'ordinateur. C'est le *langage naturel* de la machine.

Très peu de programmes sont écrits directement en langage machine, pour deux raisons principales :
1. Il est très difficile à utiliser.
2. Chaque type d'ordinateur a son propre jeu d'instructions. Un programme écrit pour une machine ne fonctionne pas sur une autre sans modifications importantes.

La plupart du temps, on utilise des langages de haut niveau, plus proches des langues humaines et de notre façon de penser. Parmi eux, il y a des langages généraux comme C, C++, Python, Pascal, Fortran ou BASIC. Il existe aussi des langages spécialisés, conçus pour des tâches précises: par exemple, SIMAN pour les simulations.

Une seule instruction en langage de haut niveau correspond souvent à plusieurs instructions en langage machine. Cela rend la programmation

beaucoup plus simple. De plus, un même programme écrit en langage de haut niveau peut généralement fonctionner sur différents ordinateurs avec peu ou pas de modifications. Les langages de haut niveau offrent donc trois grands avantages: simplicité, uniformité et portabilité (indépendance vis-à-vis de la machine).

Cependant, un programme écrit en langage de haut niveau doit être traduit en langage machine avant de pouvoir s'exécuter. Cette traduction se fait par compilation (tout le programme est traduit d'un coup) ou par interprétation (les instructions sont traduites et exécutées une par une). Le langage C++ utilise généralement un compilateur, ce qui rend les programmes plus rapides.

Le programme original s'appelle le programme source; celui traduit en langage machine s'appelle le programme objet. Chaque ordinateur a besoin de son propre *compilateur* ou *interpréteur* pour chaque langage.

Performance, efficacité… et responsabilité environnementale
En pratique, C++ est un langage de bas niveau: il est proche du langage machine, plus difficile à apprendre, mais il permet de créer des programmes très rapides et efficaces. Python, au contraire, est un langage de haut niveau: facile à apprendre et à utiliser, mais plus lent à l'exécution.

C++ est surtout utilisé pour développer de grands systèmes. Python est idéal pour créer rapidement des prototypes, car on peut proposer des solutions en quelques lignes de code. Mais pour les grands systèmes où la performance est essentielle, le choix se porte sur C++. À noter en particulier que les bibliothèques d'IA sont écrites en C++, car la performance est cruciale lors de l'entraînement et de l'inférence des réseaux neuronaux.

Des comparaisons objectives montrent que, pour un même programme, **C++ s'exécute beaucoup plus vite que Python**, que ce soit sous Linux ou Windows. Moins de temps de calcul signifie **moins de consommation d' électricité**… et donc **moins d'émissions de CO_2**. Dans un monde soucieux de durabilité, cela compte.

Par exemple, si on exécute le même programme en C++ et en Python, C++ sera bien plus rapide. Moins de temps de calcul = moins d'électricité = moins de CO_2.

Comprendre, pas seulement coder

En tant qu'ingénieur, il ne suffit pas de savoir écrire du code qui *marche*. Il faut

comprendre comment et pourquoi il fonctionne. Le C++ vous oblige à penser comme une machine : gérer la mémoire, concevoir des structures efficaces, anticiper les erreurs. C'est cette **profondeur de compréhension** qui vous distinguera des simples utilisateurs de langages modernes - ceux qui savent écrire quelques lignes, mais ignorent ce qui se passe *sous le capot*.

C'est pour ces raisons que de nombreux cursus d'ingénieurs incluent le C++. Ce cours n'est donc pas simplement un changement de langage: c'est une **étape vers une véritable culture d'ingénieur** - fondée sur les principes, la rigueur, et la maîtrise technique.

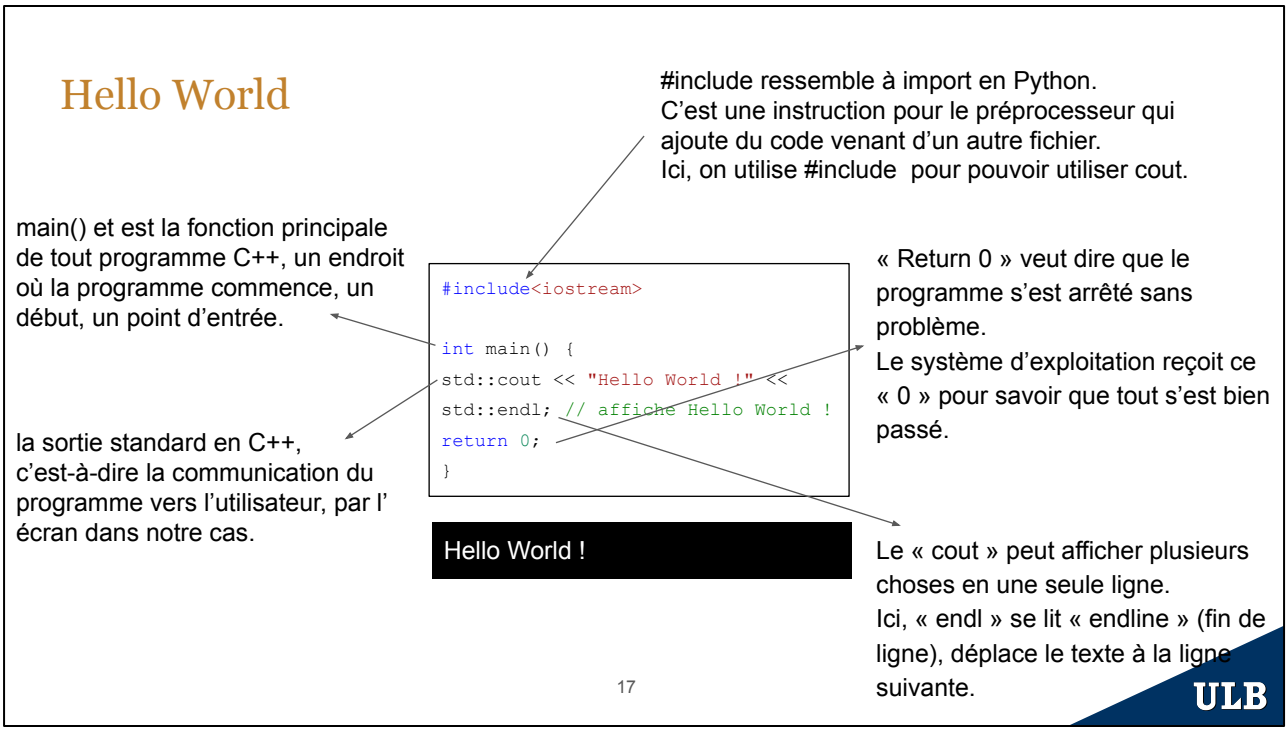
Cours EPB utilisent le C++.

- ELEC-H301 Electronique appliquée
- INFO-H304 Compléments de programmation et d'algorithmique
- ELEC-H310 Digital electronics
- ELEC-H410 Real-time computer systems
- INFO-H417 Database system architecture
- INFO-H502 3D graphics in VR
- INFO-H503 GPU computing
- INFO-H518 Immersive Multimedia Technologies
- INFO-H516 Visual Media Compression

Cours 1 : De Python à C++

ULB

- Ce cours passe rapidement en revue la syntaxe du C++.
- Vous savez déjà programmer en Python.
- Aujourd'hui, vous allez apprendre à écrire en C++ des éléments de base comme les boucles, les conditions, les fonctions, etc.
- J'ai mis des exercices pratiques sur UV. Il est très important que vous les étudiiez et que vous vous entraîniez avant le prochain cours.
- À partir du prochain cours, je supposerai que vous connaissez la syntaxe du C++ et que vous êtes capables d'écrire et d'exécuter des programmes.
- Nous aborderons alors des concepts de programmation plus avancés.
- Si vous ne pratiquez pas maintenant, il vous sera difficile de suivre dans les semaines à venir.



Voici un programme C++ simple qui affiche **Hello World** à l'écran :

Un programme C++ commence toujours par inclure les outils nécessaires à son fonctionnement. Pour afficher du texte, on utilise la directive #include <iostream>. Ce fichier fait partie de la **bibliothèque standard C++**: il contient des fonctions et objets déjà écrits par des développeurs experts, notamment ceux liés aux entrées et sorties (iostream signifie *Input/Output Stream*).

Le mot-clé #include est une **directive de préprocesseur**: elle demande au compilateur d'insérer le contenu de ce fichier avant de traduire le code en langage machine.

Le point de départ de tout programme C++ exécutable est la fonction **main()**. C'est ici que le système d'exploitation 'operating system' commence à exécuter votre programme.
main est alors la seule fonction obligatoire dans un programme.

À l'intérieur de main, on écrit des **instructions**, chacune terminée par un **point-virgule (;)**. Ici:

- std::cout permet d'**afficher du texte** à la console. C'est la sortie vers la console (*console output*).
- Les **chevrons <<** servent à envoyer du contenu vers la sortie.

- "Hello World !" est le message affiché - string.
- std::endl insère un **retour à la ligne** - new line.
- Le préfixe std:: indique que ces éléments appartiennent à l'namespace std, qui regroupe tous les composants de la bibliothèque standard C++ (std vient de *standard*).

La dernière instruction, return 0;, **termine la fonction main** et renvoie la valeur 0 au système d'exploitation. Par convention, 0 signifie que le programme s'est déroulé **sans erreur**.
Le code peut aussi contenir des **commentaires**, ignorés par le compilateur mais très utiles pour les humains :

- // introduit un commentaire sur **une seule ligne**.
- /* ... */ permet des commentaires **sur plusieurs lignes** ou même au milieu d'une instruction (bien que cela soit déconseillé pour des raisons de lisibilité).

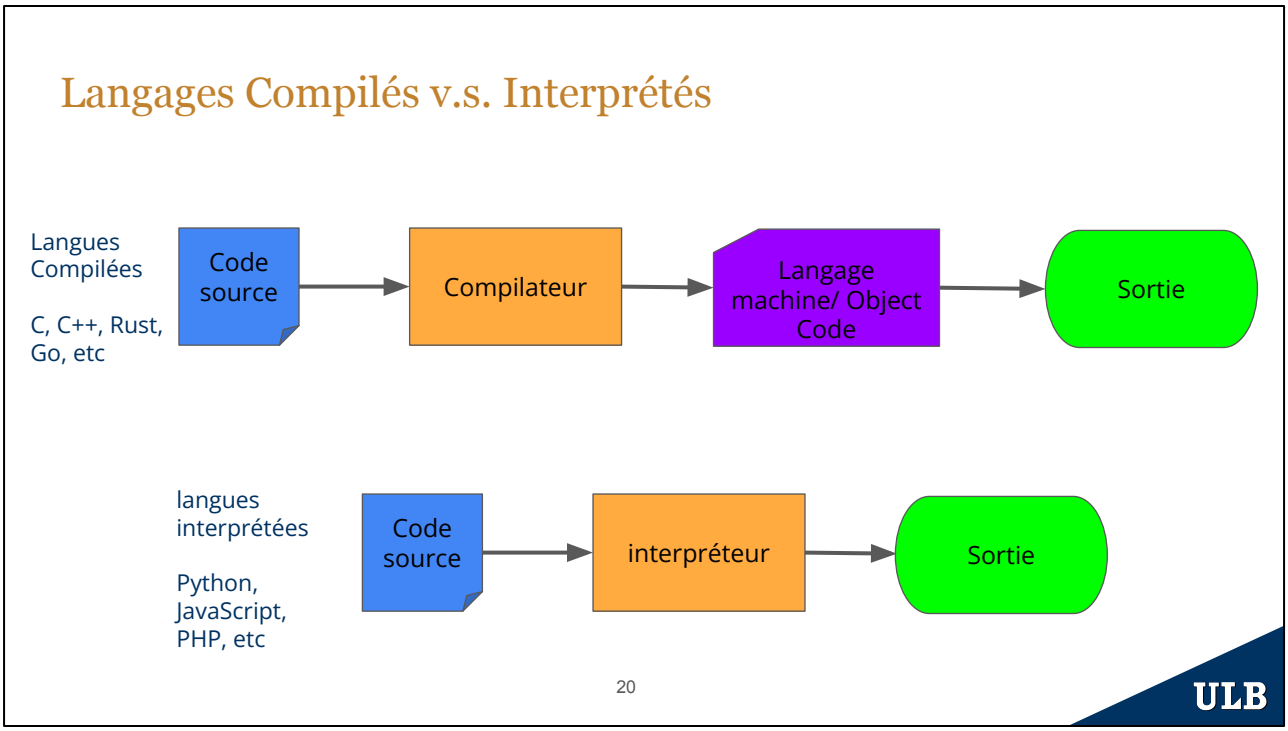
Les commentaires peuvent être rédigés dans n'importe quelle langue - ici, en français - et servent à expliquer le code, clarifier la logique, ou aider d'autres développeurs. Un bon code est souvent **lu plus souvent qu'il n'est écrit**, donc la clarté compte beaucoup.

g++

```
$ g++ hello.cpp -o hello
$ ./hello

Hello World !
```


Pour executer le programme, d'abord, on compile le fichier hello.cpp avec la commande g++, et on crée un programme appelé hello.
Ensuite, on lance ce programme avec ./hello.
g++ est un compilateur pour C++ : il transforme le code source en un programme exécutable.



Compilateurs et interpréteurs

Le but fondamental des **compilateurs** et des **interpréteurs** est de traduire du code source écrit par un humain (dans un langage de haut niveau proche de l'anglais ou d'une notation mathématique) en **instructions exécutables par une machine**, c'est-à-dire en séquences binaires (0 et 1) que le processeur peut comprendre. Puisque les machines ne comprennent que leur propre jeu d'instructions (ISA - Instruction Set Architecture), ces outils servent de pont entre l'abstraction expressive du code source et la rigueur mécanique du matériel.

Compilateur

Un **compilateur** effectue une **traduction statique**. Il analyse entièrement le programme source, le transforme en une représentation intermédiaire, l'optimise, puis génère un **code cible autonome** (souvent du code machine ou de l'assembleur). Ce code est ensuite exécuté directement par le processeur (ou une machine virtuelle). La compilation se fait **avant l'exécution**, ce qui permet des optimisations globales (propagation de constantes, élimination de code mort, inlining, etc.) et produit un binaire performant.

Des exemples de langages compilés sont : C, C++, Rust et Go. Leur code source est traduit en code machine avant l'exécution.

interpréteur

Un **interpréteur**, en revanche, **exécute directement** le programme source (ou une forme intermédiaire comme du bytecode) sans produire de fichier exécutable persistant. Il lit le code instruction par instruction, analyse sa structure, et simule son comportement à l'aide d'une boucle d'évaluation. L'exécution et l'analyse sont donc **couplées dynamiquement**, ce qui entraîne un surcoût à l'exécution mais offre plus de flexibilité (chargement dynamique, introspection).

Des exemples de langages interprétés sont : Python, JavaScript, Ruby et PHP. Ils sont exécutés dynamiquement par un interprète au moment de leur utilisation.

Architecture commune

Les deux partagent un **front-end** composé de :

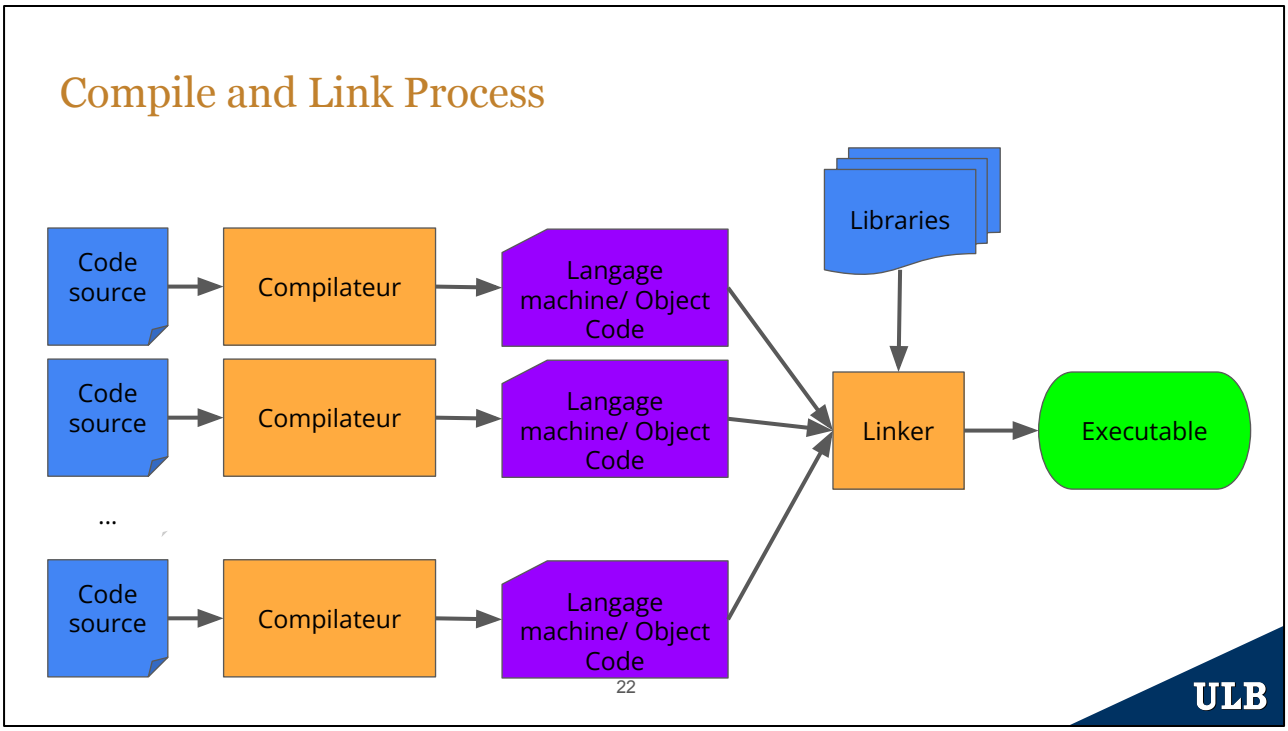
1. **Analyse lexicale** : découpage du texte source en *tokens* (identifiants, opérateurs, littéraux).
2. **Analyse syntaxique** : construction d'un **Arbre Syntaxique Abstrait (AST)** conforme à la grammaire du langage.
3. **Analyse sémantique** : vérification de typage, résolution de portées, et annotation de l'AST.

Le compilateur ajoute un **back-end** :

- Génération et optimisation d'une **Représentation Intermédiaire (IR)** (ex., LLVM IR, JVM bytecode).
- Sélection d'instructions, allocation de registres, et génération de code machine spécifique à l'architecture cible.

Hybrides modernes

De nombreux systèmes combinent les deux approches. La **compilation JIT (Just-In-Time)**, utilisée par la JVM, .NET CLR ou V8 (JavaScript), compilé dynamiquement du bytecode en code natif au moment de l'exécution, en appliquant des optimisations adaptatives basées sur le profil d'exécution réel.



La création d'un module exécutable à partir de votre code source C++ est essentiellement un processus en deux étapes.

Première étape : la compilation.
Le compilateur traite chaque fichier .cpp pour produire un **fichier objet** contenant le code machine équivalent au code source.

Deuxième étape: linkage
L'éditeur de liens (le *linker*) combine tous les fichiers objets d'un programme en un seul fichier exécutable complet. Pendant cette étape, il intègre aussi toutes les fonctions de la **bibliothèque standard** que vous utilisez.

La figure montre trois fichiers source compilés pour produire trois fichiers objets correspondants. L'extension utilisée pour identifier les fichiers objets varie selon les environnements (Windows, Linux, etc.), c'est pourquoi elle n'est pas indiquée ici.
Les fichiers source de votre programme peuvent être compilés **séparément**, lors de différentes exécutions du compilateur, ou bien **ensemble** en une seule commande. Quelle que soit la méthode, le compilateur traite chaque fichier source comme une entité indépendante et produit **un fichier objet par fichier .cpp**.

L’étape linkage combine ensuite ces fichiers objets, ainsi que les bibliothèques nécessaires, en un seul fichier exécutable.
En pratique, la compilation est un **processus itératif**. Il est presque certain que vous aurez fait des erreurs dans votre code. Une fois ces erreurs corrigées dans chaque fichier source, vous pouvez passer à l’étape de linkage, où de nouvelles erreurs peuvent encore apparaître (par exemple, une fonction déclarée mais non définie).

Même si l’étape de linkage produit un module exécutable, votre programme peut encore contenir des **erreurs logiques**: il ne donne pas les résultats attendus. Pour les corriger, vous devez retourner modifier le code source, puis recompiler. Vous répétez ce cycle jusqu’à ce que le programme fonctionne comme vous le souhaitez.

Data types

```
int myNum = 5;
float myFloatNum = 5.99;
double myDoubleNum = 9.98;
char myLetter = 'D';
bool myBoolean = true;
string myText = "Hello";
```

Data Type	Size	Description
`boolean`	1 byte	Stores true or false values
`char`	1 byte	Stores a single character/letter/number, or ASCII values
`int`	2 or 4 bytes	Stores whole numbers, without decimals
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

Types de base

- **int** : entier (généralement 2 ou 4 bytes)
- **char** : caractère (1 byte) – peut être vu comme un petit entier
- **float** : nombre à virgule flottante simple précision (4 bytes)
- **double** : nombre à virgule flottante double précision (8 bytes)

La taille exacte dépend du compilateur et de la machine.

Qualificateurs de type

On peut modifier les types entiers avec :

- **short** -> entier court (≤ int)
- **long** -> entier long (≥ int)
- **signed** -> valeur positive ou négative (par défaut pour int)
- **unsigned** -> valeur ≥ 0 seulement -> éviter erreurs & plage de valeurs plus grande

Exemples :
short int (souvent écrit short)
unsigned long int
unsigned char

Typed v.s. Untyped languages

C++ (langage typé – vérification à la compilation)

```cpp
int main() {
   int a = 5;   // 'a' est un entier
   std::string b = "hello";
   // a + b;   // Erreur de compilation : impossible d'ajouter int et string
   return 0;
}
```

Python (langage dynamiquement typé – vérification à l'exécution)

```python
a = 5          # 'a' contient un entier
b = "hello"    # 'b' contient une chaîne
print(a + b)  # Erreur à l'exécution : TypeError
```


ULB

Langages typés (ex. : C, C++, Java, Rust)

- Chaque variable, expression et fonction a un type déclaré ou inféré au moment de la compilation.
- Le compilateur vérifie la compatibilité des types avant l'exécution (vérification statique).
- Erreurs comme 5 + "hello" sont détectées à la compilation -> plus sûr, plus rapide.
- Les types permettent d'optimiser le code (taille mémoire connue, accès direct).
- En C++, les types contrôlent la représentation en mémoire (bytes, alignement, etc.).

Langages non typés / dynamiquement typés (ex. : Python, JavaScript, Ruby)

- Les variables n'ont pas de type fixe ; les valeurs ont un type, connu à l'exécution.
- Pas de vérification de type à la compilation -> plus de flexibilité, mais erreurs détectées tardivement (ex. : TypeError pendant l'exécution).
- Le même nom de variable peut contenir un entier, puis une chaîne, puis une liste.
- Le système gère automatiquement la mémoire et les conversions ->

- plus simple à écrire, mais moins efficace.
- Moins de contrôle sur la représentation mémoire → moins adapté aux systèmes critiques ou embarqués.

Control Characters

Code	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (where N is an octal constant)
\xN	Hexadecimal constant (where N is a hexadecimal constant)

Octal	Hexadecimal	Decimal	ASCII
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	'0'
'\137'	'\x05f'	95	'_'

Control Characters - Examples

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

int main()
{
  cout << "Hello\n\tWorld\n";  // Affiche :
                               // Hello
                               //     World
  cout << "Elle a dit : \"Salut !\"\n";  // Affichage : Elle a dit : "Salut !"
  cout << "Bip !\a\n";  // Affichage : Bip ! (+ éventuel son système)
  // Caractère nul dans une chaîne — les fonctions C s'arrêtent à \0
  const char* s = "Jens\0Munk";
  cout << "strlen(s) = " << strlen(s) << "\n";  // Affichage : strlen(s) = 4
  cout << "s = " << s << "\n";                        // Affichage : s = Jens
  return 0;
}
```

Control structures - if statement

```
int i= 10;
if(i == 10)
   cout<<"test passed";
else
   cout<<"test failed";
```

test passed

```
if(false)
   cout<<"test passed";
else
   cout<<"test failed";
```

test failed

```
if(5)
   cout<<"test passed";
else
   cout<<"test failed";
```

test passed

```
if(0)
   cout<<"test passed";
else
   cout<<"test failed";
```

test failed

```
int i= 10;
if(i = 5)
   cout<<"test passed";
else
   cout<<"test failed";
```

test passed

```
if(true)
   cout<<"test passed";
cout<<"test failed";
```

test passed
test failed

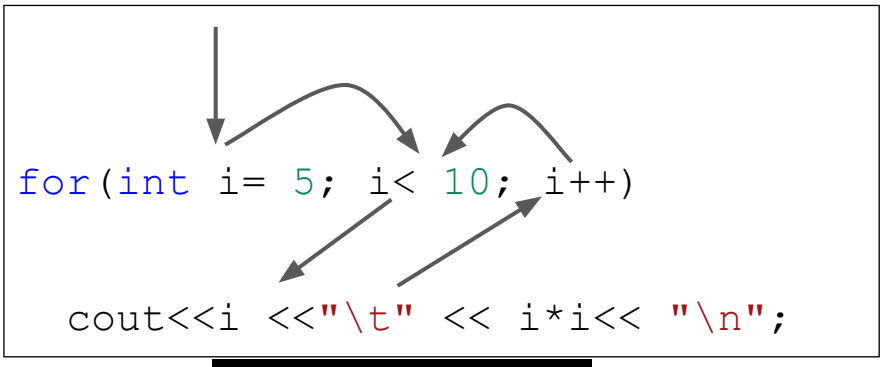
Control structures - switch case

```
int day = 4;
if(day == 1)
   cout << "Monday";
else if (day == 2)
       cout << "Tuesday";
else if (day == 3)
       cout << "Wednesday";
else if (day == 4)
       cout << "Thursday";
else if (day == 5)
       cout << "Friday";
else if (day == 6)
       cout << "Saturday";
else if (day == 7)
       cout << "Sunday";
```

Thursday

```
int day = 4;
switch (day) {
   case 1:
       cout << "Monday";  break;
   case 2:
       cout << "Tuesday"; break;
   case 3:
       cout << "Wednesday"; break;
   case 4:
       cout << "Thursday"; break;
   case 5:
       cout << "Friday"; break;
   case 6:
       cout << "Saturday"; break;
   case 7:
       cout << "Sunday"; break;
   }
```

Control structures - for loop




```
5    25
6    36
7    49
8    64
9    81
```

Control structures - for loop

```
for(int i= 5; i<= 10; i++)
 cout<< i << ", ";
```

5, 6, 7, 8, 9, 10,

```
for(int i= 5; i< 10; i+=2)
 cout<< i << ", ";
```

5, 7, 9,

```
for(int i= 5; i< 5; i++)
 cout<< i << ", ";
```

```
for(int i= 5; ; )
 cout<< i << ", ";
```

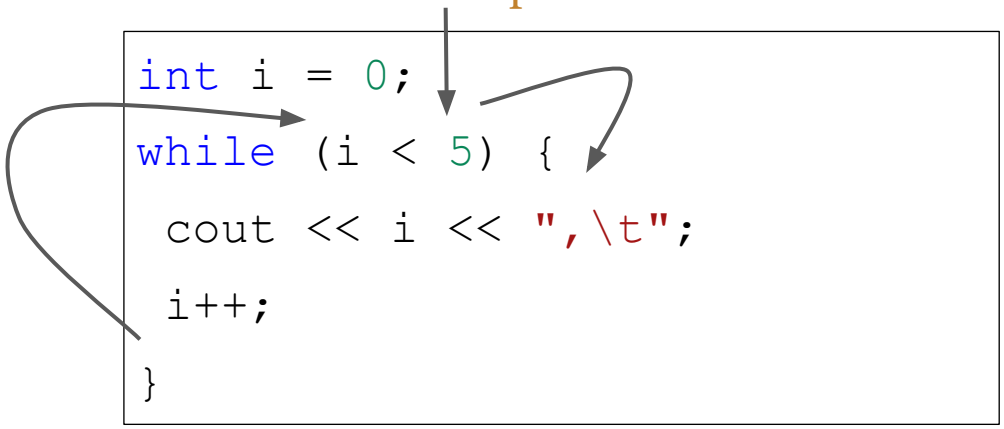
5, 5, 5, 5, … infinite

```
for(; ;)
 cout<< 2 << ", ";
```

2, 2, 2, 2, 2, … infinite

```
for(int i= 5; i--; )
cout<< i << ", ";
```

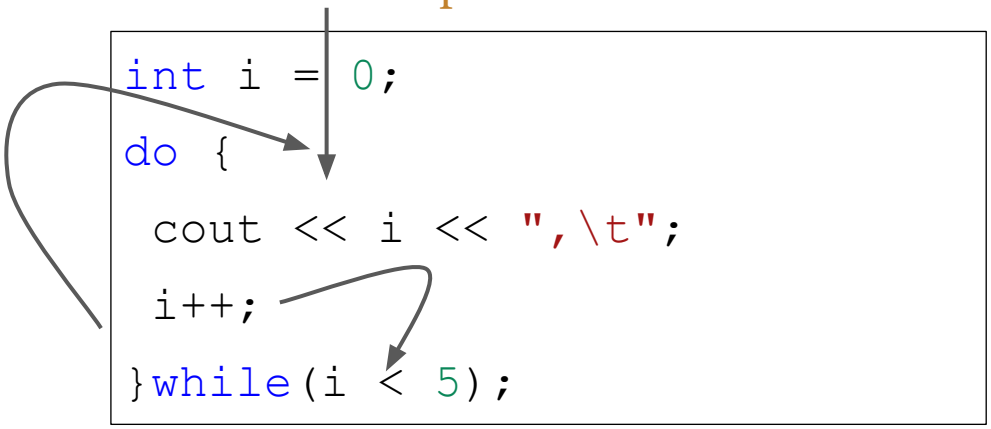
Control structures - while loop




```
0,	1,	2,	3,	4,
```


ULB

Control structures - do loop




```
0,	1,	2,	3,	4,
```

Statement

- Une instruction C++ est une unité de base d'un programme qui se termine par un point-virgule, et non par un retour à la ligne.
- Une fonction est une séquence d'instructions.
- Les instructions sont délimitées par des points-virgules et des brackets {}.
- L'indentation et les espaces blancs n'ont aucun rôle en C++. Ils servent uniquement à améliorer la lisibilité.
- Les brackets regroupent des instructions en un bloc (instruction composée).

Principaux types d'instructions en C++ :

- Instructions-expression
- Instructions composées ({ ... })
- Instructions de sélection (if, switch)
- Instructions de boucle (for, while, do-while)
- Instructions de control (return, break, continue, goto)
- Déclarations de variables
- Blocs try / gestion des exceptions

1. Qu'est-ce qu'une instruction ?

Une **instruction** (ou *statement*) est une unité d'exécution dans un programme C++.
Chaque instruction se termine généralement par un point-virgule (;), sauf les blocs ({}) et les structures de contrôle qui les contiennent.
Dans main(), chaque ligne exécutable est une instruction. Ensemble, elles forment le **flux d'exécution** du programme.

2. Types d'instructions

a) Instruction-expression

C'est une **expression** suivie d'un point-virgule.
Exemples :

- `x = 5;` (affectation)
- `std::cout << "Bonjour";` (appel de fonction)
- `i++;` (incrémentation)

  La plupart des lignes de code dans un programme C++ sont des instructions-expression.

b) Instruction composée (bloc)

Un **bloc** regroupe plusieurs instructions entre accolades `{ }`.
Il permet de traiter plusieurs lignes comme **une seule instruction**, utile dans les `if`, les boucles, etc.

Chaque bloc crée aussi une **portée (scope)**: les variables déclarées dedans sont détruites à la fin du bloc.

c) Instructions de sélection

Elles permettent de choisir un chemin d'exécution :

- `if (condition) { ... }`
- `if (condition) { ... } else { ... }`
- `switch (valeur) { case ... }`

On peut même déclarer une variable dans la condition :
`if (int x = getValue(); x > 0) { ... }`

d) Instructions de boucle (itération)

Elles répètent une instruction (ou un bloc) :

- `while (condition) { ... }`
- `do { ... } while (condition);`
- `for (init; cond; inc) { ... }`

e) Instructions de saut

Elles modifient brusquement le flux normal

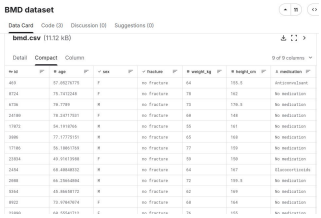
f) Déclarations comme instructions

En C++, **déclarer une variable est une instruction valide**: Cela diffère de certains langages où les déclarations doivent être séparées du code exécutable.

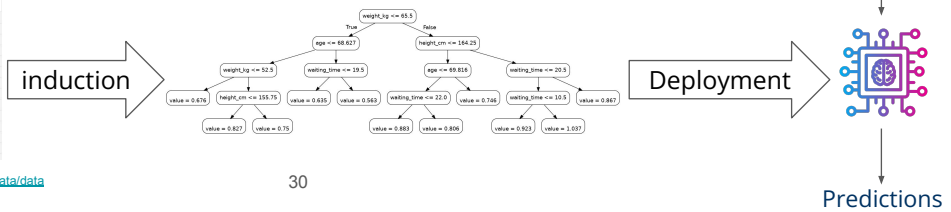
g) Autres types (avancés)

Regression Tree

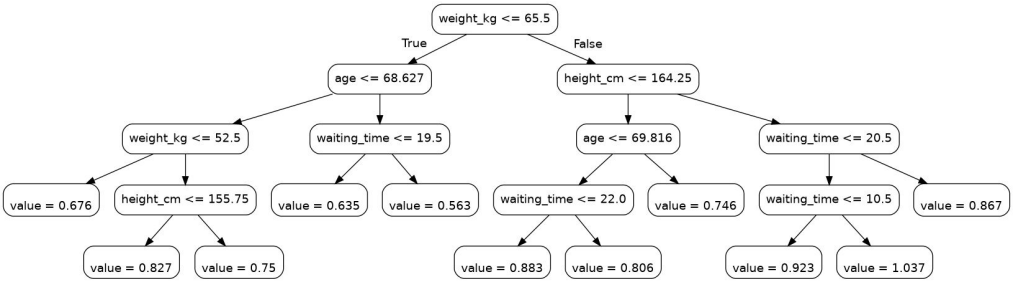
- Regression Tree est un modèle de machine learning (une forme d'intelligence artificielle).
- À partir d'un ensemble de données, on construit un modèle capable d'estimer ou de prédire une variable en fonction d'autres variables. Ce modèle a la forme d'un arbre.
- La construction de ces arbres (c'est-à-dire l'apprentissage du modèle / training) sera étudiée dans le cours INFO-H-423 « Data Mining » du master M-IRIFS.
- Ici, nous allons écrire un programme pour utiliser ce modèle (c'est-à-dire l'inférence).



https://www.kaggle.com/datasets/amarsharma768/bmd-data/data



BMD Regression Model



bmd_tree_transitions.txt

```
1   1,2,3,weight_kg <= 65.50
2   2,4,5,age <= 68.63
3   3,6,7,height_cm <= 164.25
4   4,8,9,weight_kg <= 52.50
5   5,10,11,waiting_time <= 19.50
6   6,12,13,age <= 69.82
7   7,14,15,waiting_time <= 20.50
8   8,-1,-1,0.68
9   9,18,19,height_cm <= 155.75
10  10,-1,-1,0.64
```

ULB

BMD Regression - v0.1

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

float estimate(float age, float weight_kg, float height_cm, float waiting_time) {...

// test step 1
int main() {
   // Example: matches a patient from your data
   float bmd = estimate(60, 70, 165, 30);
   std::cout << "Predicted BMD: " << bmd << std::endl;  // Should be 0.87
   return 0;
}
```

```
Predicted BMD: 0.87
```

BMD Regression - v1.0

```
int main() {
   char choice;
   float age, weight_kg, height_cm, waiting_time;
   cout << "=== BMD Estimator (Based on Trained Regression Tree) ===\n\n" ;
   do {
       cout << "Enter patient details:\n" ;
       cout << "Age (years): ";
       cin >> age;

       cout << "Weight (kg): ";
       cin >> weight_kg;

       cout << "Height (cm): ";
       cin >> height_cm;

       cout << "Waiting time (days): " ;
       cin >> waiting_time;

       float bmd = estimate(age, weight_kg, height_cm, waiting_time);
       cout << "\n--> Predicted BMD: " << bmd << "\n\n";
       // Ask to continue
       cout << "Estimate another patient? (y/n): " ;
       cin >> choice;
       cout << "\n";
   } while (choice == 'y' || choice == 'Y');
   cout << "Thank you for using the BMD estimator!\n" ;
   return 0;
}
```

BMD Regression - v1.0

```
float estimate(float age, float weight_kg, float height_cm, float waiting_time) {
   if (weight_kg <= 65.5) {
       if (age <= 68.63) {
           if (weight_kg <= 52.5) {
               return 0.68;
           } else {
               if (height_cm <= 155.75) {
                   return 0.83;
               } else {
                   return 0.75;
               }
           }
       } else {
           if (waiting_time <= 19.5) {
               return 0.64;
           } else {
               return 0.56;
           }
       }
   } else {
       if (height_cm <= 164.25) {
           …
       }
   }
}
```