Cours 01 - C++ pour les mathématiques appliquées

Introduction

De quoi ça parle?

Pourquoi C++ alors que Python est plus simple ?

- Python : Prototypage rapide
- C++ : Performance critique

Pour simuler 1 million de particules en temps réel

Python	~45 secondes
C++	~0.8 secondes

Performance x50 pour des applications scientifiques

C++ moderne - Aperçu de ce qui vous attend

Objectif du cours

Maîtriser ces concepts C++

```
// Lambda expressions (Cours 7-8)
[&, n] (int a) mutable { m = ++n + a; }(127);
[](auto a, auto b) { return a + b; }
```

```
// Modules (Cours 11-12)
export module Particles;
import Logger;
```

Ces exemples montrent où nous allons, pas où nous commençons.
Les éléments éléments seront abordés progressivement.

Maîtriser les concepts d'analyse et conception.

C++ efficace - Pattern avancé (fin de parcours)

CRTP : Curiously Recurring Template Pattern

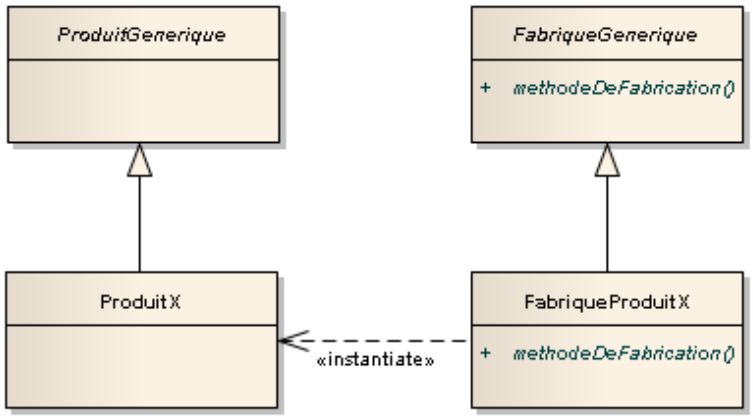
```cpp
template <typename T>
class Total {
    public :
    double getValue() const{
        return static_cast<const T&>(*this).getValue();
    }
};

class Constant42 : public Total<Constant42> {
    public :
    double getValue() const{return 42;}
};
```

- Cours 1 : Introduction
- Cours 9 : Templates avancés

Pourquoi ce pattern ?

- Polymorphisme sans coût runtime
- Utilisé dans Eigen, Boost

Conception avec des modèles



Factory

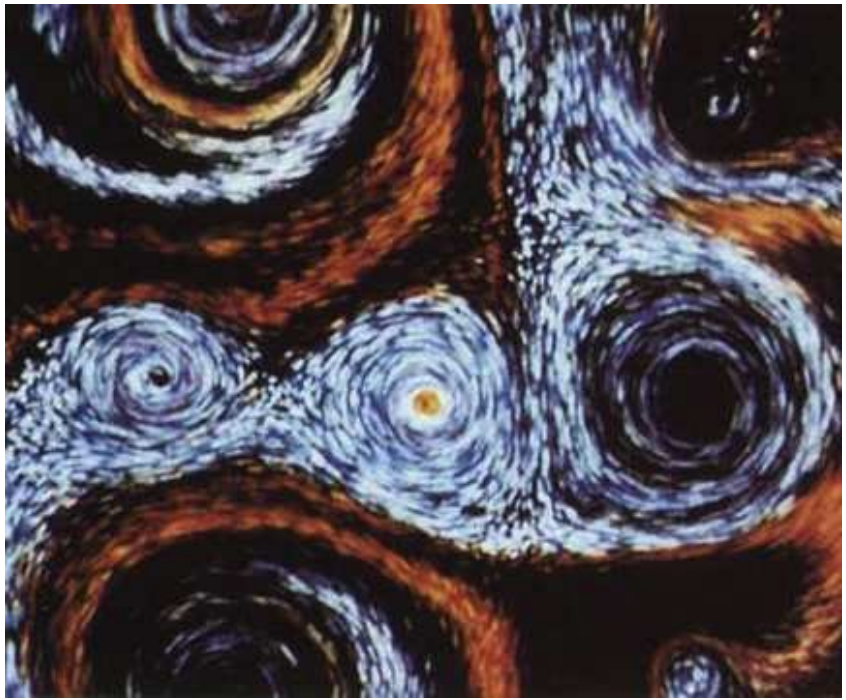
Quand vous l'utiliserez :

- TP : Créer différents types de particules
- TP : Système modulaire extensible

Dans l'industrie :

- Moteurs de jeux (Unity, Unreal)
- Simulations CFD (OpenFOAM)

… d’applications - Projet fil rouge



Système de particules physiques

Progression du projet :

Semaine	Concept	Fonctionnalité
2	Fonctions	Initialisation simple
4	Classes	Objet `Particle`
7	Héritage	Différents types
10	Templates	Optimisation
11	**Résultat final**	**Simulation temps réel**

Organisation du cours

Équipe pédagogique
Christophe Picard - Amphi, Groupe 2
Benoît Massé - Groupe 1, Groupe 3

Horaires
Mercredi 11h15 – 12h45 : Cours en amphi. (exceptionnellement le lundi)
Jeudi 8h15 – 9h45/9h45 – 11h15 : TP par binômes.

Objectifs

- Se familiariser avec les différents concepts.
- Comprendre les concepts présentés dans le cours.
- Développer une maîtrise des concepts nécessaires aux applications.
- Appréhender divers aspects de la programmation scientifique.
- Développer un code scientifique en langage C++.
- Mettre en oeuvre des outils de travail performants.

Consignes Généralités

- Les TP sont en binômes.
- Les rendus doivent se présenter sous la forme d'une archive au format zip (ou tar.gz) qui créera à l'extraction un répertoire ***TPi_nom1_nom2***.
- Les rapports doivent être au format PDF uniquement et ne contiennent pas de code.
- Les codes doivent être réfléchis.

Consignes Présentation du code

Le code doit :

- être facile à lire.
- s'articuler logiquement.
- être déboggable.
- être transmissible.

Vous devez :

- commenter le code abondamment, précisément, clairement et immédiatement.
- indenter proprement et uniformément.
- donner des noms significatifs aux "objets" que vous manipulez.

Consignes : Exemple

Non maintenable

```
// Pas de commentaire
double c(double x) {
    return x*1.8+32;
}

int main() {
    double t = 25;
    double r = c(t);
}
```

Maintenable

```
/// Convertit Celsius en Fahrenheit
/// @param celsius température en °C
/// @return température en °F
double celsius_to_fahrenheit(
    double celsius) {
    return celsius * 1.8 + 32.0;
}

int main() {
    double temp_celsius = 25.0;
    double temp_fahrenheit =
        celsius_to_fahrenheit(temp_celsius);
}
```

Quelques hypothèses

- Vous savez programmer dans un langage quelconque.
- Vous savez utiliser un terminal.
- Vous avez accès à un compilateur C++ moderne.
- Vous connaissez vos algorithmes et structures de données de bases.
- Vous savez résoudre une EDO.

Calcul scientifique

- Le calcul scientifique, les sciences des données, l'IA consistent à déplacer une grande quantité d'information entre des mémoires et des unités de traitement, les traiter et les déplacer vers la mémoire à nouveau

C'est la raison de l'existence de **FORTRAN**

Mais cela reste un langage avec des capacités génériques réduites.

C++ est un langage qui permet de construire des abstractions et de les combiner.

Calcul scientifique - Exemple concret

Problème type

Résoudre $\frac{\partial u}{\partial t} = \nabla^2 u$ sur un domaine 3D

Volume de données

- Grille : 1000 × 1000 × 1000 points
- Mémoire : ~8 Go (double précision)
- Calculs : ~10^9 opérations/itération

Pourquoi C++ ?

Aspect	Impact
Gestion mémoire	Contrôle fin des allocations
Vectorisation	SIMD automatique
Cache	Localité spatiale optimale
Parallélisme	OpenMP, MPI natifs

Motivations

- Simulation d'un processus physique :
  - Processus physiques $\Longrightarrow$ modèle mathématique $\Longrightarrow$ algorithme $\Longrightarrow$ logiciel $\Longrightarrow$ simulation numérique
- Application d'algorithmes numériques, utilisation massive des approches numériques
- Domaines d'application :
  - Simulation de phénomènes naturels (Code de météo France)
  - Applications industrielles (Dassault System)
  - Application en médecine, en finance
  - Cybersécurité (Linbox)

Logiciel scientifique

Quelques propriétés

- **Validation**
  - Tests unitaires (Google Test)
  - Benchmarks de référence
- **Efficacité** : vitesse, mémoire, stockage, énergétique
  - Profiling avec `perf`, `gprof`
  - Optimisation compilateur
- **Maintenabilité**
  - Documentation (Doxygen)
  - Versioning (Git)
- **Extensibilité**
  - Architecture modulaire
  - Plugins et interfaces

Développement d'applications en mathématiques appliquées

Objectifs

- Comprendre le modèle mathématiques.
- Comprendre l'approche numérique.
- Concevoir les algorithmes et les structures de données.
- Choisir et utiliser les bibliothèques et outils adaptés.
- Vérifier les résultats.
- S'adapter aux nouveaux langages et concepts.

Code de calcul type

- Point de départ : problème computationnel
- Pre-processing
  - Données d'entrée et préparation.
  - Mise en place des structure de données
- Coeur du calcul
- Post-processing
  - Analyse des résultats
  - Affichage, sortie et visualisation

Logique de développement

L'implementation correcte de problèmes numériques évolués est une tâche difficile.

Elle se divise en deux étapes :

- Exprimer le problème numérique sous forme d'un algorithme.
- Traduire l'algorithme en langage machine par l'intermédiaire d'un langage de programmation

Quel langage de programmation?

- Il existe plusieurs centaines de langages de programmation.
- Pour le calcul scientifique, les choix sont beaucoup plus restreints : Fortran (77, 95, 2003, 2008), C, C++, Matlab (GNU Octave), Python, Julia, Maple/Mathematica.
- Comment le choisir
  - Efficacité calculatoire.
  - Coût de développement, maintenabilité.
  - Eco-système efficace et évolué.
  - Support pour des types utilisateurs.
- Les projets sont protéiformes. Des langages différents sont requis. L'interopérabilité est nécessaire.

Propriétés : compilés vs interprétés

Langages compilés	**Langages interprétés**
Plus rapides en général	Plus lents en général
Cycles de développement plus longs	Cycles de développement plus rapides
C, C++, Fortran	Python, Matlab, Julia

Les fonctionnalités numériques intensives sont typiquement dans des langages compilés.

Types utilisateurs

Les primitives de types ne sont pas toujours suffisantes pour les codes numériques!

Les primitives sont groupées dans un nouveau type de données

- `struct` en C
- `class` en Java, Python, et C++

Les hiérarchies de classes sont indispensables en programmation scientifique.

La programmation orientée objet peut conduire à des pertes de performances importantes.

Le C++ c'est quoi?

- Tout usage.
- Flexible en permettant aux développeurs/utilisateurs de construire des abstractions.
- Performance et efficacité.
- Se faire comprendre par l'utilisateur.

On en est où?

- 1979 : Création des classes pour C par Bjarne Stroustrup
- 1983 : Changement de nom.
- 1985 : Livre de référence.
- 1998 : Standardisation ISO/IEC 14882:1998 $\Longrightarrow$ C++98
- 2003 : révision mineure
- 2011 : nouveau standard $\Longrightarrow$ C++11
- 2014 : révision mineure $\Longrightarrow$ C++14
- **2017 : révision intermédiaire** $\Longrightarrow$ C++17
- 2020 : révision majeure $\Longrightarrow$ C++20 (gcc 12)
- 2023 : révision mineure C++23

On apprend quoi dans le cours?

Des concepts du `C++`

- Outils de gestion de code : compilation, liens et gestion des dépendances, intégration continue, tests.
- Gestion de ressources: RAII, smart pointers, references, copy/move
- Programmation procédurale : fonctions, paramètres , lambdas, surcharges, et gestion d'erreur…
- OOP: classes, héritage, polymorphisme.
- Programmation générique : templates et variations ….
- Programmation à la compilation : template récursifs, constexpr, et traits.
- Conteneurs et itérateurs : STL, iterator, vues.

Premier exemple de code

On commence par quoi?

```cpp
#include <iostream>
int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

```
$ g++ --std=c++17 hello.cpp -o hello
$ ./hello
Hello, world!
```

À ce stade, vous devez pouvoir compiler et exécuter ce programme.
C'est un rappel de 1A !

Ce que fait le compilateur

1. Le préprocesseur s'exécute en premier.
2. La directive `#include` recopie l'ensemble du fichier dans l'unité de compilation courante.
3. Les symboles `< ... >` indiquent de rechercher dans les répertoires systèmes.
4. Le fichier inclus est `iostream` de la bibliothèque standard.

En résumé : Le préprocesseur inclut les en-têtes avant la compilation proprement dite.

Ce que fait l'OS

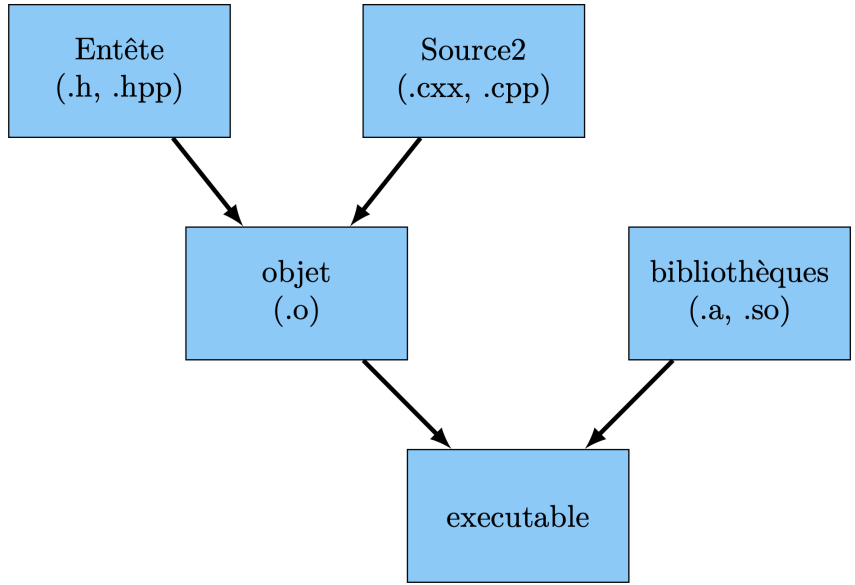
- Chaque programme ne peut contenir qu'une et une seule fonction `main`.
- Le compilateur et le système d'exploitation identifie cette fonction spéciale afin qu'elle soit exécutée lors du lancement.

Ce que ça signifie

- `std::` est l'**espace de nom** standard.
- Un espace de nom est un regroupement d'objets avec une portée limitée.
- Pour accéder à un élément d'un espace de noms, on utilise l'**opérateur** `::` .
- `std::cout` représente la sortie standard, `std::cin` l'entrée standard.
- La bibliothèque standard utilise l'opérateur `<<` pour l'insertion dans un flux.

Quelques Outils

Compilation



Compilation et édition de lien

```
$g++ -std=c++17 -Wall -Wextra -O3 -c -o source.o source.cxx
```

```
$g++ -std=c++17 -Wall -Wextra -O3 -o exe source.o source.a
```

Automatisation : cmake

Il s'agit maintenant de générer des `Makefile` qui soient multi plateformes. Nous allons donc utiliser l'outil `cmake`.

- Fichier équivalent

```
cmake_minimum_required (VERSION 3.16.3)
project(Demo CXX)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "-Weverything -Wall")
# Commentaire
add_executable(exe source.cxx)
```

- Invocation

```
mkdir build; cd build; cmake ..; make
```

Deboggage

Afin de trouver les erreurs dans votre code, l'utilisation de **gdb** est fortement recommandée.

Quelques commandes pour gdb :

- Lancer gdb : `gdb ./exe`
- Lancer le programme sous gdb avec argument et redirection de l'affichage: `(gdb) run arg1 arg2 > sortie`
- Afficher les données : `(gdb) print i j`
- Fonction : `(gdb) call fonction(arg1,arg2, ... )`
- Variable : `(gdb) set variable name = exp`
- Naviguer dans la liste des appels `up` ou `down`

Le programme doit être compilé avec l'option `-g`.

Deboggage - Exemple pratique

Code avec bug

```cpp
int factorial(int n) {
    int result = 1;
    for(int i = 1; i < n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int f = factorial(5);
    // f = 24 au lieu de 120 !
}
```

Session gdb

```
$ g++ -g factorial.cpp -o fact
$ gdb ./fact
(gdb) break factorial
(gdb) run
(gdb) print i
(gdb) print n
(gdb) next
(gdb) print result
# Identifier : i < n au lieu de i ⩽ n
```

💡 **TP1 :** Pratiquez gdb sur des bugs régulièrement.

Gestion de la mémoire

- L'un des aspects critiques de la programmation dans des langages compilés évolués est la gestion de la mémoire.
- `valgrind` est un outil qui permet de vérifier que la mémoire est gérée de façon correcte, et que toute la mémoire utilisée a bien été libérée.
- Quelques options pour `valgrind`
  - Analyser les fuites mémoires `valgrind --leak-check=yes myprog arg1 arg2`
  - Montrer les zones encore accessible `--show-reachable`
  - Être bavard `-v`
  - Enregistrer dans un fichier `--log-file=filename`

Gestion de la mémoire - Exemple

Code avec fuite mémoire

```cpp
void process_data() {
    int* data = new int[1000];
    // Traitement ...
    // Oups, pas de delete[] !
}

int main() {
    for(int i = 0; i < 10; i++) {
        process_data();
    }
    return 0;
}
```

Détection avec valgrind

```
$ g++ -g memory_leak.cpp -o leak
$ valgrind --leak-check=full ./leak

==1234== LEAK SUMMARY:
==1234==    definitely lost: 40,000 bytes
==1234==    in 10 blocks
==1234==
==1234== at process_data()
==1234==    (memory_leak.cpp:2)
```

Conseil : Toujours utiliser valgrind sur vos TPs avant de rendre.
Remarque : Sur MacOS, l'alternative est leaks

La documentation

- Il existe plusieurs types de documentations : besoin, conception, technique, utilisateur.
- La documentation technique peut-être générée en partie automatiquement
- Quelques commandes pour `doxygen`
  - `@param` : En instrumentant votre code, vous pouvez, par exemple, spécifier les arguments retour de vos fonctions.
  - `@brief` : Résumé de description.
  - `@fn` : Documentation de fonction.
  - `@todo` : Un bon moyen de communiquer sur l'état d'un morceaux de code.
  - `@bug` : Un bug a été identifié. Donner les informations nécessaire.

La documentation - Exemple Doxygen

Code documenté

```
/// @brief Calcule la norme d'un vecteur
/// @param x composante x
/// @param y composante y
/// @param z composante z
/// @return norme euclidienne
/// @todo Optimiser avec SIMD
double norm(double x, double y,
            double z) {
    return sqrt(x*x + y*y + z*z);
}
```

Génération HTML

```
$ doxygen -g Doxyfile
$ doxygen Doxyfile
# Génère html/index.html
```

Résultat : Documentation navigable avec :

- Liste des fonctions
- Paramètres et types
- Graphes de dépendances
- TODOs et bugs recensés

Types

String

- La bibliothèque standard possède une classe `string` qui permet de manipuler une chaîne de caractères.
- Elle s'utilise avec l'entête approprié `#include <string>`

```cpp
#include <iostream>
#include <string>
int main() {
    std::string message = "Hello, world";
    std::cout << message << std::endl;
}
```

String - Opérations courantes

Manipulation

```cpp
std::string s1 = "Hello";
std::string s2 = " World";

// Concaténation
std::string s3 = s1 + s2;

// Longueur
size_t len = s3.length();

// Sous-chaîne
std::string sub = s3.substr(0, 5);

// Recherche
size_t pos = s3.find("World");
```

Conversion

```cpp
// String vers nombre
std::string num_str = "42";
int num = std::stoi(num_str);
double d = std::stod("3.14");

// Nombre vers string
int x = 100;
std::string x_str =
    std::to_string(x);
```

Nous verrons les conteneurs STL en détail dans les dernières semaines.

Fonctions

Fonction 101

Une fonction encapsule un morceau de code entre des accolades. La fonction peut être réutilisée ultérieurement.

```
void say_hello() {
    std::cout << "Hello, world!" << std::endl;
}
int main(int argc, char* argv[]) {
    say_hello();
    return 0;
}
```

Signature de fonction

Une fonction déclare ses types de retour et de paramètre.

Les paramètres sont des variables locales initialises par la fonction d'appel

Une valeur est renvoyée avec la mot clé `return`. Le type de l'expression retournée doit être compatible avec le type de retour de la fonction.

```
int sum(int a, int b) {
    return a + b;
}
```

Le type de retour peut être induit:

```
auto sum(int a, int b) → int {
    return a + b;
}
```

Fonctions - Progression vers les lambdas

Fonctions classiques

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int r = add(3, 4);
}
```

Pointeurs de fonctions

```
int (*op)(int, int);
op = add;

int apply(int a, int b,
  int (*f)(int, int)) {
    return f(a, b);
}
```

Lambdas

```
auto add =
  [](int a, int b) {
    return a + b;
  };

auto compute =
  [x=10](int y) {
    return x + y;
  };
```

Surcharge

Des fonctions avec le même nom, mais des arguments différents peuvent coexister.

```
int sum(int a, int b) {
    return a + b;
}
double sum(double a, double b) {
    return a + b;
}
```

Lorsque la fonction `sum` est invoquée, le compilateur identifie les arguments et recherche le meilleur candidat.

Le meilleur candidat peut-être une fonction standard ou une fonction utilisateur.

Surcharge - Résolution à la compilation

Code source

```
int sum(int a, int b) {
    return a + b;
}

double sum(double a, double b) {
    return a + b;
}

int main() {
    int x = sum(3, 4);       // Appel 1
    double y = sum(3.0, 4.0); // Appel 2
    auto z = sum(3, 4.0);    // Appel 3 ?
}
```

Résolution du compilateur

- **Appel 1** : `sum(int, int)` - correspondance exacte
- **Appel 2** : `sum(double, double)` - correspondance exacte
- **Appel 3** : Promotion `int → double` → `sum(double, double)`

⚠️ **Attention :** Ambiguïtés possibles !
Le compilateur peut refuser de compiler si plusieurs candidats sont équivalents

Conclusion

Checkpoint - Ce que vous devez savoir maintenant

Concepts de base

- Compiler et exécuter un programme C++
- Utiliser `std::cout` et `std::string`
- Écrire une fonction simple
- Comprendre la surcharge

Outils

- Compiler avec `g++`
- Utiliser `cmake` (basique)
- Notions de `gdb` et `valgrind`

Questions à vous poser

- Puis-je compiler le Hello World ?
- Ai-je compris la différence compilé/interprété ?
- Suis-je à l'aise avec mon environnement de dev, y compris git?

La prochaine fois

Cours 2 : Les classes

- Encapsulation et abstraction
- Constructeurs et destructeurs
- Membres et méthodes
- Premier pas vers le projet fil rouge

Questions ?