MU4MA056 : Programmation en C++

Table des matières :

Organisation générale

- cours le lundi matin 8h30-10h30, salle 24.34.207
- cours **supplémentaire** jeudi 20 janvier, 13h45-15h45, amphi 55A.
- documents et annonces sur Moodle.
- Travaux Pratiques : à partir du lundi 24 janvier, salle 16.26.401, 3 groupes de 20 : lundi, jeudi, vendredi, 13h45–16h45.
- Règles sanitaires **strictes**.

Évaluation

Habituellement :

$$\textbf{Note sur 100} \stackrel{?}{=} \frac{\text{TP1 sur 25} + \text{TP2 sur 25}}{2} + \text{Examen sur 75}$$

- TP noté 1 : sixième semaine des TP (début mars)
- TP noté 2 : douzième semaine des TP (mi-avril)

Conseils pour réussir cette UE

- *UE traditionnellement difficile* : le C++ est un langage très développé et très puissant et donc très exigeant.
- UE d'informatique au milieu d'UE de mathématiques : autre philosophie. Un nouveau langage avancé en 6 ECTS.

Conseils pour réussir cette UE

- *UE traditionnellement difficile* : le C++ est un langage très développé et très puissant et donc très exigeant.
- UE d'informatique au milieu d'UE de mathématiques : autre philosophie. Un nouveau langage avancé en 6 ECTS.
- Une **règle d'or** :

  Codez, codez, codez ! ! !
- *UE professionnalisante* : ce que demande une entreprise est que vous soyez à l'aise devant un clavier et du C++ (pas que vous soyez superficiellement au courant des derniers gadgets).

Conseils pour réussir cette UE

- *UE traditionnellement difficile* : le C++ est un langage très développé et très puissant et donc très exigeant.
- UE d'informatique au milieu d'UE de mathématiques : autre philosophie. Un nouveau langage avancé en 6 ECTS.
- Une **règle d'or** :

  Codez, codez, codez ! ! !
- *UE professionnalisante* : ce que demande une entreprise est que vous soyez à l'aise devant un clavier et du C++ (pas que vous soyez superficiellement au courant des derniers gadgets).
- *Informatique pour mathématiciens* : pas d'interface, pas de fantaise, mais de la performance numérique. Algorithmique + code optimisé.

Quelques généralités.

Langages informatiques

- Langages interprétés : Python, Ruby, matlab/scilab...
  Très pratique pour petit script, bac à sable, appels à des bibliothèques, syntaxe simplifiée,... mais **lent pour le calcul intensif**

Langages informatiques

- Langages interprétés : Python, Ruby, matlab/scilab...
  Très pratique pour petit script, bac à sable, appels à des bibliothèques, syntaxe simplifiée,... mais **lent pour le calcul intensif**
- Langages compilés : Fortran, C, C++... **Rapide pour le calcul intensif** mais *plus long à écrire, plus de détails à surveiller, plus de subtilités à maîtriser...*

Langages informatiques

- Langages interprétés : Python, Ruby, matlab/scilab...
  Très pratique pour petit script, bac à sable, appels à des bibliothèques, syntaxe simplifiée,... mais **lent pour le calcul intensif**
- Langages compilés : Fortran, C, C++... **Rapide pour le calcul intensif** mais *plus long à écrire, plus de détails à surveiller, plus de subtilités à maîtriser...*
- Interprétation : traduction en langage machine+exécution de chaque ligne, pas de vue globale sur programme. Compilation : traduction en langage machine globale sur l'ensemble du programme puis exécution globale donc possibilité de nombreuses optimisations.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.
- **C et C++ :** 2 évolutions mais volonté de rétrocompatibilité.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.
- **C et C++ :** 2 évolutions mais volonté de rétrocompatibilité.
- C++, depuis 1990 : ajout de la **programmation générique** et début de la **bibliothèque standard**. Fortes disparités avec le C malgré le passé commun.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.
- **C et C++ :** 2 évolutions mais volonté de rétrocompatibilité.
- C++, depuis 1990 : ajout de la **programmation générique** et début de la **bibliothèque standard**. Fortes disparités avec le C malgré le passé commun.
- C++, 2011 : nouveau standard majeur. Nouvelles fonctionnalités et *forte influence sur l'écriture du code*.

  C++, standards 2014 et 2017 : poursuite de cette évolution.

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.
- **C et C++ :** 2 évolutions mais volonté de rétrocompatibilité.
- C++, depuis 1990 : ajout de la **programmation générique** et début de la **bibliothèque standard**. Fortes disparités avec le C malgré le passé commun.
- C++, 2011 : nouveau standard majeur. Nouvelles fonctionnalités et *forte influence sur l'écriture du code*.

  C++, standards 2014 et 2017 : poursuite de cette évolution.
- C++, standard 2020 : nouveau standard majeur (pas encore sur tous les compilateurs !).

Histoire de C et C++

- 1970 : langage C par Ken Thompson pour écrire Unix. *Avoir un langage adapté pour manipuler la mémoire : pointeurs*.
- 1980 : développement du C++ par Bjarne Stroustrup. *Augmentation du C pour laprogrammation orientée objet.* Ajout des **classes**.
- **C et C++ :** 2 évolutions mais volonté de rétrocompatibilité.
- C++, depuis 1990 : ajout de la **programmation générique** et début de la **bibliothèque standard**. Fortes disparités avec le C malgré le passé commun.
- C++, 2011 : nouveau standard majeur. Nouvelles fonctionnalités et *forte influence sur l'écriture du code*.

  C++, standards 2014 et 2017 : poursuite de cette évolution.
- C++, standard 2020 : nouveau standard majeur (pas encore sur tous les compilateurs !).

Conclusion : même si on peut encore écrire du C++ un peu comme du C, les deux langages n'ont plus grand chose à voir.

Attention, si vous connaissez déjà C ! ! !

Langage informatique : de quoi parle-t-on ?

Dans un ordinateur :

- processeur(s) (CPU)
- mémoire vive (RAM)

- carte(s) graphique(s) (GPU) avec leur processeurs et leurs mémoires

- mémoire morte (disque dur)
- périphériques (clavier, écran, souris...)

- tout ça branché sur la carte-mère.

Langage informatique : de quoi parle-t-on ?

Dans un ordinateur :

- processeur(s) (CPU)
- mémoire vive (RAM)
- carte(s) graphique(s) (GPU) avec leur processeurs et leurs mémoires
- mémoire morte (disque dur)
- périphériques (clavier, écran, souris...)
- tout ça branché sur la carte-mère.

Une **variable** en informatique ($\neq$ variable en mathématique) :

- un emplacement dans la RAM (avec adresse et taille)
- un type (manière d'encoder en une suite de bits 0 ou 1)
- un nom dans votre programme
- une *valeur*

Calculs faits par le CPU ; la valeur ensuite sauvegardée dans le disque dur.

Les outils du programmeur

1. Un éditeur de code (Geany, XCode, Code : :blocks, etc.)

Les outils du programmeur

1. Un éditeur de code (Geany, XCode, Code : :blocks, etc.)
2. Un compilateur (g++, clang++, etc.)

Les outils du programmeur

1. Un éditeur de code (Geany, XCode, Code : :blocks, etc.)
2. Un compilateur (g++, clang++, etc.)
3. quelques bibliothèques (libeigen3 pour nous pour l'algèbre linéaire).

Les outils du programmeur

1. Un éditeur de code (Geany, XCode, Code : :blocks, etc.)
2. Un compilateur (g++, clang++, etc.)
3. quelques bibliothèques (libeigen3 pour nous pour l'algèbre linéaire).
4. **du papier et des crayons ! ! !**

Les outils du programmeur

1. Un éditeur de code (Geany, XCode, Code : :blocks, etc.)
2. Un compilateur (g++, clang++, etc.)
3. quelques bibliothèques (libeigen3 pour nous pour l'algèbre linéaire).
4. **du papier et des crayons ! ! !**
5. des documentations :

   `https://en.cppreference.com/w/`

   et

   `http://www.cplusplus.com/reference/`

   savoir les lire fait partie du cours !

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.
- savoir définir et utiliser des **classes** (POO)

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.
- savoir définir et utiliser des **classes** (POO)
- savoir définir et utiliser des **templates** (programmation générique)

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.
- savoir définir et utiliser des **classes** (POO)
- savoir définir et utiliser des **templates** (programmation générique)
- savoir lire la documentation

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.
- savoir définir et utiliser des **classes** (POO)
- savoir définir et utiliser des **templates** (programmation générique)
- savoir lire la documentation
- connaître les parties les plus utilisées la bibliothèque standard.

Buts du cours

- connaître et *être à l'aise* avec *toute* la syntaxe de base, de la plus ancienne à la plus moderne.
- savoir définir et utiliser des **classes** (POO)
- savoir définir et utiliser des **templates** (programmation générique)
- savoir lire la documentation
- connaître les parties les plus utilisées la bibliothèque standard.

NE FAIT PAS PARTIE DU COURS :

les tout derniers standards C++17 et C++20, l'intégralité de la bibliothèque standard, les subtilités fines de l'héritage de classes, les notions les plus récentes des templates, les espaces de noms, le C.

Syntaxe élémentaire du C++.

Premier programme

```cpp
#include <iostream>
int main() {
    std::cout << "Bienvenue en MU4MA056.\n";
    return 0;
}
```

Compilation (dans le terminal) avec :

```
g++ essai1.cpp -o essai1.exe
```

Lancement (dans le terminal) avec

```
./essai1.exe
```

Fonction, entrée/sortie

```
#include <iostream>
#include <cmath>
double circle_area(double r) {
    return M_PI*r*r;
}
int main() {
    std::cout << "Entrez le rayon du cercle:\n";
    double x;
    std::cin  >> x;
    std::cout << "L'aire du cercle est "
          << circle_area(x) << "\n";
         return 0; }
```

Bibliothèque externe

```
#include <iostream>
#include <Eigen/Dense>// chemin vers bibliothèque
int main()
{
  Eigen::Matrix<double,4,4> A;
  A << 1, 2 , 3, 4 , 1,-1,1,-1,4,3,2,1,1,-1,0,0 ;
  std::cout << "La matrice A est:\n" << A << "\n";
  std::cout << "Son determinant est: "
            << A.determinant() << "\n";
  std::cout << "Son inverse est:\n"
        << A.inverse() << "\n";
  return 0;}
```

Compilation avec :

```
g++ -I /usr/include/eigen3 essai3.cpp -o essai3.exe
```

L'affichage produit

```
La matrice A est:
 1  2  3  4
 1 -1  1 -1
 4  3  2  1
 1 -1  0  0
Son determinant est: 40
Son inverse est:
-0.075 -0.125  0.175    0.5
-0.075 -0.125  0.175   -0.5
 0.175  0.625 -0.075   -0.5
 0.175 -0.375 -0.075    0.5
```

Déclaration des variables

Langage fortement typé : déclaration avec type.

```
// Nombres réels:
   double x=4.; //initialisé à 4
// Nombres entiers:
   int n=-2;
   long int m=-45654553365433556;
// Tableaux :
   std::vector<int> v(50,3);//50 entiers égaux à 3
   std::vector<double> w{2.3,-4.,1.1,0.};//4 réels
// Chaîne de caractères:
   std::string S("Le C++, c'est top.");
// Syntaxe alternative > C++11 préférée désormais:
   auto v = vector<int>(50,3);
   auto w = vector<double>{2.3, 4.2, 1.1, 0.};
   auto S = "Le C++, c'est top"s;
```

Variables

1. un nom (dans le code, disparaît dans l'exécutable)
2. un type (donne la taille de l'emplacement mémoire **et** spécifie le codage binaire)
3. un emplacement mémoire (une adresse dans la mémoire vive RAM)
4. une valeur (stockée à l'emplacement mémoire via le codage du type)

mais aussi :

- une durée de vie : la variable est automatiquement supprimée à la fin du bloc où elle a été déclarée.
- toutes les fonctions et opérations qui sont définis sur le type de la variable

Exemple des entiers

- `int` : stocké sur 4 octets, un bit de signe, 31 bits pour la valeur absolue (les entiers relatifs de -2^{31} à $2^{31}-1$)
- `long unsigned int` : stocké sur 8 octets, pas de bits de signes, 64 bits pour la valeur absolue (les entiers naturels de 0 à $2^{64}-1$)

Les opérations :

- `+`, `-`, `*`, `/` (quotient euclidien), % (reste euclidien), etc.
- les opérations associées `+=`, `*=`, `++`, etc.

Attention, arithmétique modulaire si on dépasse les bornes maximales et minimales, **sans avertissement** !

(structure similaire pour les réels)

Chaînes de caractères standard

- requiert la bibliothèque `<string>`
- type `std::string`
- concaténation, affichage, accès à un caractère
- exemple :

```
std::string s1 = "Hello ";
std::string s2 = "World!";
std::string s3 = s1+s2;
std::cout << s3 << "\n";//retour à la ligne après
std::cout <<s1[0] <<s1[4] <<"\n" ; //affiche: Ho
std::cout << s2.size(); //nombre de caractères
```

- voir la documentation :
  `http://www.cplusplus.com/reference/string/string/`

Structures

[**attention**, on fera mieux avec les classes bientôt !]

- Possibilité de combiner plusieurs types dans un nouveau type.

```
struct Point2D {
        double x;
        double y;
        std::string nom;//chaîne de caractères
};
```

- déclaration par :

```
Point2D U;
```

- accès aux champs/attributs par

```
U.x = 3.1 ;
U.nom = "A";
```

Tableaux/vecteurs (à l'ancienne)

- bibliothèque `<vector>`
- `https://en.cppreference.com/w/cpp/container/vector`

```
std::vector<double> u(7,3.5);//7 éléments égaux à 3.5
u.push_back(2.1);// ajout d'un huitième élément
std::cout << u.size(); //nombre d'éléments
u[0] += u[6]; //lire/modifier des éléments
std::vector<double> v(u);//nouvelle copie de u
for( int j=0; j< u.size() ; j++ )  {
     v[j] = 2*u[j]-1;
}
std::cout << "Les éléments sont: ";
for( int j=0; j< v.size() ; j++ )  {
        std::cout << v[j] << " ";
}
```

- ressemble aux tableaux du C mais avec outils en plus et possibilité de faire varier la taille (`std::array` sinon).

Les vecteurs : version moderne

Beaucoup d'évolution pour **limiter les erreurs**, écrire un code plus lisible et plus facilement modifiable : utilisez-les !
Reprise du code précédent avec outils STL et C++11 :

```
using namespace std; //bibliothèque standard !
auto u=vector<double>(7,3.5);//7 éléments égaux à 3.5
u.push_back(2.1);// ajout d'un huitième élément
std::cout << u.size(); //nombre d'éléments
u.front() += u.back(); //lire/modifier des éléments
auto v= vector<double>{u};//nouveau même taille
transform( begin(u),end(u),
   begin(v),
   [](double x) { return 2.*x-1.;} );
cout << "Les éléments sont: ";
for( auto x: v )  {
   cout << x << " ";
}
```

Structures de contrôle

- si... alors...

```
if ( test ) {  .../// instructions  }
else {  .../// instructions  }
```

- boucle d'itérations

```
for( init ; test_de_sortie ; incrementation ) {
    .../// instructions
}
```

- boucle *tant que*

```
while( condition ) {
    .../// instructions
}
```

Les références : une spécificité du C++

- si `T` est un type connu, on a un type `T &` : référence sur objet de type `T`
- *sa valeur* : un emplacement immuable dans la mémoire
- déclaration :

```
double a=1.;
double & r=a;// r n'est pas un double !
std::vector<double> v(100000,12.1);
const std::vector<double> & ref=v;
   // ref n'est pas un vecteur mais un "entier"
```

- Intérêt : évite des copies inutiles
- attention aux modifications induites :

```
r=2.7;
r=ref[2]; // OK
ref[3]=2.1; //impossible car const
```

Les fonctions

- Passage par valeurs : copies des arguments ! **ATTENTION, cela peut être très long. !**
- Sinon pointeurs/références pour éviter les copies. Attention alors au `const`.
- 3 types de prototypes pour les arguments :

```
double f(T a);
//copie de la valeur
double f( T & a);
//copie de l'adresse, modif de la valeur ?
double f(const T & a);
//copie de l'adresse, valeur non modifiée.
```

- 2 manières de définir des fonctions.

Les fonctions : déf. 1

En **dehors** de toute autre fonction :

```
T   NOM( T1 a1, T2 a2, ...,Tn an) {
    //...code...
    return ... ;// de type T
} // T: type du résultat.
```

Ne pas oublier d'ajouter les prototypes dans les en-têtes :

```
T   NOM( T1 a1, T2 a2, ...,Tn an);
```

Les fonctions : déf. 2

Attention : depuis les standars C++11 et C++14 seulement : possibilité de définir une fonction n'importe où dans le code avec son type propre.

Définition d'une λ-fonction :

```
auto f= [captures](arguments)->TYPE { code };
```

Exemple :

```
const int u=4;
auto mult_u = [u](int n) ->int { return n*u; };
std::vector<int> v(100);
std::transform(v.begin(),v.end(),v.begin(),mult_u);
bool b= std::any_of(v.begin(),v.end(),
        [](int n){ return n>= 12; });
//cherche s'il existe un élément plus grand que 12
```

Les fonctions : déf. 2

Dans une λ-fonction : les captures :

- capture par copie : `[a,b,c]`
- capture par adresse : `[a,&b,&c]`
- tout par copie : `[=]`
- tout par adresse : `[&]`

Dans une fonction normale :

- définition en dehors de toute autre fonction donc la question ne se pose pas.
- difficulté pour fixer l'un des arguments pour un appel par une fonction de `<algorithm>` (cf. exemple)

Les classes.

Les classes

But : créer de nouveaux types d'objets

Philosophie : séparer l'implémentation de l'objet de son utilisation :

- utilisation : doit avoir des noms de méthodes simples, qui ressemble le plus possible à des objets déjà existants, l'utilisateur ne doit pas avoir à réfléchir à ce qui passe en coulisse.
- implémentation : structuration des données, optimisation des calculs, gestion de la mémoire, initialisations correctes, format de lecture et d'écriture.

Exemple d'utilisation de classe (I).

```
const unsigned int N=40;
Graph G(N); //constructeur
G.read_from_file("graph_exemple.dat");//lecture
unsigned int k2= G.nb_of_neighbours(2);
bool b = G.test_connect_from_to(4,5);
std::cout<<"Nb of edges: "<< G.nb_of_edges()<<"\n";
{
   Graph H(G);//copie
   for (int i=1; i<G.size() ; i++)
      H.add_link(2,i);
   std::ofstream output("graph_modified.dat");
   output << H; //écriture
   output.close();
}//effacement de H.
G.reverse_all_edges();
std::vector<int> v2 = G.edges_from(4);
```