UE 3.1 C++ : Programmation Orientée Objet

Simon Rohou

2025/2026

Supports de cours disponibles sur `www.simon-rohou.fr/cours/c++/`
Slides inspirées du cours de Olivier Marguin (Univ. Lyon)

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

Planning prévisionnel

1. (4cr) Introduction au C++

2,3. (8cr) Programmation orientée objet

4. (8cr) Outils de développement (atelier)

5. (4cr) Tableaux et pointeurs

6. (4cr) Fichiers et conteneurs

7,8,9. (12cr) Projet Modèle Numérique de Terrain

10. (4cr) Évaluation

Section 2

Les classes

Structure des fichiers

La programmation d'une classe se fait en 3 phases :

1. **Déclaration**
   Fiche descriptive des données et fonctions-membres des objets : interface avec le monde extérieur.

2. **Définition**
   Partie implémentation, contenant la programmation des fonctions membres.

3. **Utilisation**
   Instanciation d'objets de cette classe.

Structure des fichiers

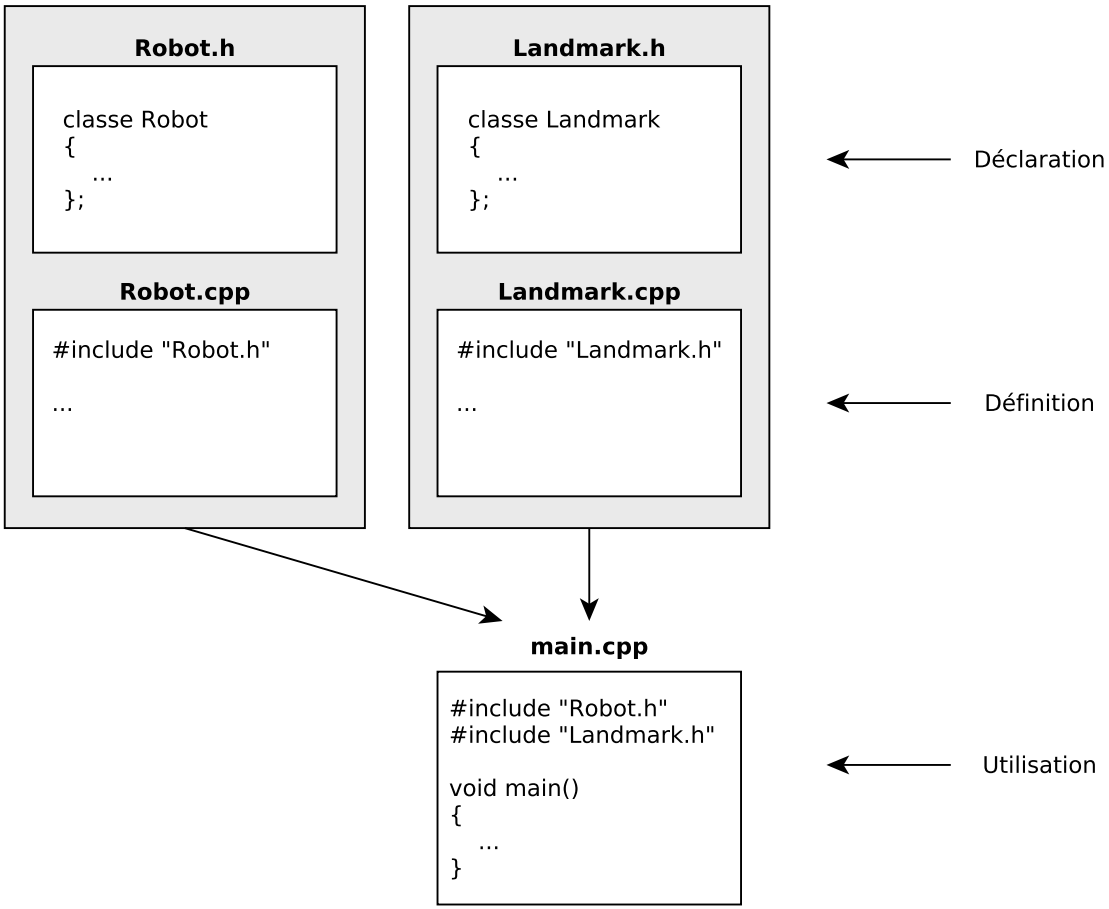
La programmation d'une classe se fait en 3 phases :

1. **Déclaration**
   Fiche descriptive des données et fonctions-membres des objets : interface avec le monde extérieur.
   → dans un fichier `MaClasse.h`

2. **Définition**
   Partie implémentation, contenant la programmation des fonctions membres.
   → dans un fichier `MaClasse.cpp`

3. **Utilisation**
   Instanciation d'objets de cette classe.
   → dans un fichier `.cpp`, par exemple : `main.cpp`

Structure des fichiers



Rappel : la directive d'inclusion `#include` permet d'inclure un fichier de déclarations dans un autre fichier.

Exemple de déclaration (Robot.h)

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17  #endif
```

Exemple de déclaration (Robot.h)

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7
8
9
10
11
12
13    private:
14
15  };
16
17  #endif
```

Exemple de déclaration (Robot.h)

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7
8
9
10
11
12
13    private:
14      float _x = 0., _y = 0.; // variables de classe
15  };
16
17  #endif
```

Exemple de déclaration (Robot.h)

```
1   #ifndef __ROBOT_H__ // pour éviter les inclusions cycliques
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7       Robot(); // constructeur par défaut (non paramétré)
8       Robot(float x, float y); // constructeur paramétré
9       ~Robot(); // destructeur
10      float pos_x() const; // accesseur (lecture de _x)
11      void set_pos_x(float x = 0.); // mutateur (écriture de _x)
12
13    private:
14      float _x = 0., _y = 0.; // variables de classe
15  };
16
17  #endif
```

Exemple de définition (Robot.cpp)

```
1   #include "Robot.h" // permet l'accès aux déclarations
2
3   Robot::Robot() : _x(0.), _y(0.)
4   {
5     // valeurs par défaut spécifiées ci-dessus
6   }
7
8   Robot::Robot(float x, float y) : _x(x), _y(y)
9   {
10    // valeurs configurées ci-dessus
11  }
12
13  Robot::~Robot()
14  {
15    // destructeur
16  }
17
18  // ...
```

Exemple de définition (Robot.cpp)

```
1   #include "Robot.h" // permet l'accès aux déclarations
2
3   Robot::Robot() : _y(0.)
4   {
5     _x = 0.; // équivalent à la précédente slide
6   }
7
8   Robot::Robot(float x, float y) : _x(x), _y(y)
9   {
10    // valeurs configurées ci-dessus
11  }
12
13  Robot::~Robot()
14  {
15    // destructeur
16  }
17
18  // ...
```

Exemple de définition (Robot.cpp)

```
1   // ...
2
3
4     // Accesseur : lecture de la variable de classe _x
5     // La méthode est 'const' -> elle ne modifie pas l'objet
6     float Robot::pos_x() const
7     {
8       return _x;
9     }
10
11    // Mutateur : modification de la variable _x
12    // L'utilisation du 'const' n'est pas possible ici
13    void Robot::set_pos_x(float x)
14    {
15      _x = x;
16    }
```

Exemple d'utilisation (main.cpp)

```
1   #include "Robot.h" // permet l'accès aux déclarations
2
3   int main()
4   {
5     Robot r1; // appel implicite du constructeur non paramétré
6     Robot r2(3., 5.); // appel du constructeur paramétré
7
8     cout << r1.pos_x() << endl; // affiche 0.
9     r2.set_pos_x(6.);
10    cout << r2.pos_x() << endl; // affiche 6.
11
12    return EXIT_SUCCESS;
13  }
```

Compilation avec g++ :

```
g++ Robot.cpp main.cpp -o output
```

Constructeurs et destructeurs

Constructeur :
Fonction-membre appelée automatiquement à la création d'un objet.

- il initialise l'objet
- plusieurs constructeurs peuvent être définis
  (s'ils diffèrent par le nombre ou le type de paramètres)

Destructeur :
Fonction membre unique qui s'oppose au constructeur.

- n'a pas de paramètres
- est préfixé par ~

```
1    Robot(); // constructeur par défaut (non paramétré)
2    Robot(float x, float y); // constructeur paramétré
3    ~Robot(); // destructeur
```

Surcharge d'opérateurs

Possibilité de **reprogrammer** la plupart des opérateurs usuels du langage.

Exemple d'une classe `Interval` :

- un *intervalle* est un ensemble mathématique : $[x] = [x^-, x^+]$
- on définit des opérations telles que $[x] - [y] = [x^- - y^+, x^+ - y^-]$

```
1   class Interval
2   {
3     public :
4       Interval(double x); // intervalle dégénéré (x^- == x^+)
5       Interval(double lb, double ub); // 2 bornes différentes
6
7     private:
8       double _lb; // x^- (lower bound)
9       double _ub; // x^+ (upper bound)
10  };
```

Surcharge d'opérateurs

Méthodes de classe `Interval` :

Surcharge d'opérateur unaire : $-[x] = [-x^+, -x^-]$

```
1   const Interval operator-(const Interval& x)
2   {
3     return Interval(-x.ub(), -x.lb());
4   }
```

Surcharge d'opérateur binaire : $[x] - [y] = [x^- - y^+, x^+ - y^-]$

```
1   const Interval operator-(const Interval& x, const Interval& y)
2   {
3     return Interval(x.lb() - y.ub(), x.ub() - y.lb());
4   }
```

Conversion de types

Des **conversions de types** peuvent se faire de manière implicite.

Par exemple, le calcul $[a] + 2$...

```
1   Interval a(1,3);
2   Interval b = a + 2;
```

... additionne un intervalle avec un réel.

Dans ce cas, le réel `2.0` est converti en objet `Interval` par un **appel implicite** du constructeur `Interval(double x)`.

Section 3

Héritage

Classe dérivée

Une classe dérivée **hérite** de tous les membres (données et fonctions) de la classe de base.

Dans un fichier `UnderwaterRobot.h` :

```
1   class UnderwaterRobot : Robot // héritage de Robot
2   {
3     public:
4       UnderwaterRobot(float x, float y, float z); // constructeur
5       ~UnderwaterRobot(); // destructeur
6       // ...
7
8     private:
9       float _z = 0.; // ajout d'une variable de classe
10  };
```

Classe dérivée

Une classe dérivée **hérite** de tous les membres (données et fonctions) de la classe de base.

Dans un fichier `UnderwaterRobot.cpp` :

```
1  UnderwaterRobot::UnderwaterRobot(float x, float y, float z) :
       Robot(x, y), _z(z) // appel au constructeur de Robot
2  {
3    // contenu du constructeur
4  }
5
6  UnderwaterRobot::~UnderwaterRobot()
7  {
8    // destructeur optionnel
9  }
```

Contrôle des accès

Par qualificatifs dans le fichier de déclaration `.h` :

- `protected` :
  Réserve l'accès aux données et fonctions membres
  de la classe aux seules classes dérivées.
- `private` :
  Les données et fonctions membres privées
  sont inaccessibles aux classes dérivées.

```
1  class Robot
2  {
3    public:
4      // ... membres accessibles partout
5    protected:
6      // ... membres accessibles aux classes dérivées
7    private:
8      // ... membres complètement privés
9  };
```

Héritage public ou privé

Restrictions (optionnelles) sur héritage :

- `class UnderwaterRobot : public Robot`
  Les membres hérités conservent les mêmes droits d'accès.
- `class UnderwaterRobot : private Robot`
  Tous les membres dérivés deviennent privés.

Par défaut, tous les membres hérités deviennent privés.

Héritage public ou privé

```
1   class A
2   {
3     public:
4       int x;
5
6     protected:
7       int y;
8
9     private:
10      int z;
11  };
```

```
1   class B : public A
2   {
3     // x est public
4     // y est protected
5     // z n'est pas accessible depuis B
6   };
7
8   class C : protected A
9   {
10    // x est protected
11    // y est protected
12    // z n'est pas accessible depuis C
13  };
14
15  class D : private A  // 'private' par défaut
16  {
17    // x est private
18    // y est private
19    // z n'est pas accessible depuis D
20  };
```

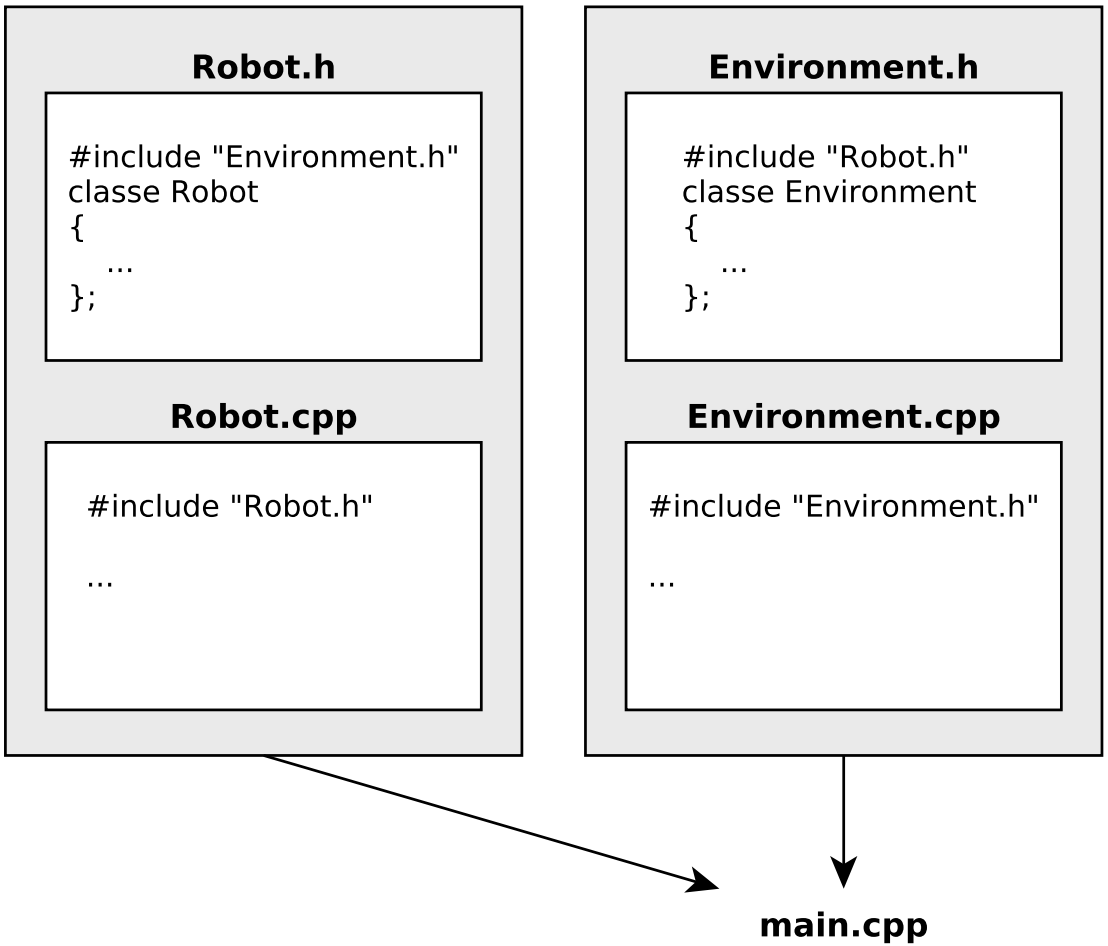
Fonctions amies : le mot-clé friend

Le mot-clé `friend` permet de **redéfinir des droits d'accès**. Il permet par exemple à une fonction d'avoir accès aux éléments privés d'une classe.

```
1   class Robot
2   {
3     private:
4       string _name; // attribut privé
5       friend void rename(Robot& r, const string& new_name);
6   };
7
8   void rename(Robot& r, const string& new_name)
9   {
10    r._name = new_name;
11  }
12
13  int main()
14  {
15    Robot r;
16    rename(r, "nouveau nom");
17  }
```

Dépendance cyclique entre deux classes

Il se peut qu'une **dépendance cyclique** ait lieu : une classe ayant besoin d'une autre pour se déclarer, et inversement.



Dépendance cyclique entre deux classes

Dans ce cas, on réalise une **pré-déclaration** de l'autre classe en en-tête du fichier de déclaration.

Fichier `Robot.h` :

```
1
2
3
4  #include "Environment.h"
5  // Pré-déclaration :
6  class Environment;
7
8  class Robot
9  {
10   // ...
11
12
13 };
```

Fichier `Environment.h` :

```
1
2
3
4  #include "Robot.h"
5  // Pré-déclaration :
6  class Robot;
7
8  class Environment
9  {
10   // ...
11
12
13 };
```

Dépendance cyclique entre deux classes

Et on indique au compilateur de n'inclure les fichiers de déclaration **qu'une seule fois** lors de la compilation.

Fichier `Robot.h` :

```
1  #ifndef __ROBOT_H__
2  #define __ROBOT_H__
3
4  #include "Environment.h"
5  // Pré-déclaration :
6  class Environment;
7
8  class Robot
9  {
10   // ...
11 };
12
13 #endif
```

Fichier `Environment.h` :

```
1  #ifndef __ENVIRONMENT_H__
2  #define __ENVIRONMENT_H__
3
4  #include "Robot.h"
5  // Pré-déclaration :
6  class Robot;
7
8  class Environment
9  {
10   // ...
11 };
12
13 #endif
```

ENSTA INSTITUT POLYTECHNIQUE DE PARIS

D'autres notions d'héritage en C++ ...

... que l'on verra par la suite :

- Opérateurs de portée
- Héritage multiple
- Classes abstraites
- Relations d'amitiés
- ...

Section 4

Passage d'objets

Valeurs, références et pointeurs

- **Passage par valeur** :

```
1    void display(Robot r);
2    // r est en lecture seule
```

- **Passage par référence** :

```
1    void move(Robot &r);
2    // r est en lecture/écriture
```

- **Passage par pointeur** :

```
1    void move(Robot *r);
2    // valeur accessible via l'adresse de l'objet
```

Les références pour la modification de valeurs (rappel)

Pour **modifier la valeur** d'un paramètre dans une fonction, il faut passer ce paramètre par référence.

Une référence sur une variable est un synonyme de cette variable, c'est-à-dire une autre manière de désigner le même emplacement de la mémoire.

On utilise le symbole & pour déclarer une référence.

Les références pour l'efficacité

Un passage de paramètre par valeur recopie l'objet dans le corps de la fonction. Lorsque des objets lourds sont passés par valeur (en lecture), leur copie peut prendre du temps.

On privilégie alors un **passage par référence** pour indiquer une référence directe sur l'objet à lire. Mais pour empêcher sa modification, on ajoute le mot-clé `const`. C'est la **const-correctness**.

```
1   // Fonctions équivalentes :
2
3   void display(Robot r)
4   { // ici, une copie locale de r est faite
5     cout << r.name() << endl;
6   }
7
8   void display(const Robot& r)
9   { // plus rapide à l'exécution : pas de copie de r
10    cout << r.name() << endl;
11  }
```

Const-correctness

```
1   #ifndef __ROBOT_H__
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7       Robot();
8       Robot(float x, float y);
9       std::string name();
10      void set_name(std::string& name);
11
12    private:
13      std::string _name;
14      float _x = 0., _y = 0.;
15      float _size; // attribut constant
16  };
17
18  #endif
```

Const-correctness

```
1   #ifndef __ROBOT_H__
2   #define __ROBOT_H__
3
4   class Robot
5   {
6     public:
7       Robot();
8       Robot(float x, float y);
9       const std::string& name() const; // accesseur = const
10      void set_name(const std::string& name);
11
12    private:
13      std::string _name;
14      float _x = 0., _y = 0.;
15      const float _size; // attribut constant
16  };
17
18  #endif
```

Const-correctness sur variables simples

Cette technique n'est utile que sur des objets relativement lourds en mémoire. On s'en affranchira pour des types simples.

```
1  Road::Road(const float& l) // 'const' inutile ici
2  {
3    // ...
4
5  }
```

Écrire simplement :

```
1  Road::Road(float l)
2  {
3    // ...
4
5  }
```

Les pointeurs (introduction)

Les pointeurs sont très utilisés en C/C++.

- un pointeur est un type de variable renseignant une adresse

```
1   int a = 2;
2   int *ptr_a = nullptr; // init. : aucune variable pointée
3   ptr_a = &a; // le pointeur pointe sur a
```

- il permet d'accéder aux valeurs d'une variable en ne conservant que son adresse

```
1   cout << *ptr_a << endl; // affiche la valeur de a : 2
2   ptr_a = 3; // opération interdite (changement d'adresse)
3   *ptr_a = 3; // modifie la valeur de a : a == 3
```

Les pointeurs : exemple

```
1   void display1(const string *name) {
2     *name = "autre"; // operation interdite (ne compile pas)
3     cout << *name << endl;
4   }
5
6   void display2(string *name) {
7     *name = "autre"; // operation permise
8     cout << *name << endl;
9   }
10
11  int main()
12  {
13    string n = "test";
14    // Passage par pointeur de l'adresse de la variable 'n'
15    display1(&n);      // affiche "test"
16    display2(&n);      // affiche "autre"
17    cout << n << endl; // affiche "autre"
18  }
```

À retenir : la notion de variables locales

Fichier `Car.h` :

```
1  class Car
2  {
3    public:
4      // ...
5      void set_front_car(Car front);
6
7    private:
8      Car *_front_car;
9  };
```

Fichier `Car.cpp` :

```
1  void Car::set_front_car(Car front)
2  {
3    _front_car = &front; // impossible, car objet 'front' local
4                         // à la méthode set_front_car() :
5                         // il disparait à la fin de l'appel
6  }
```

Section 5

Compléments

Structures de données : classe vector

Il existe de **nombreuses structures de données**, telles que `stack`, `list`, `map`, largement documentées. On s'intéresse pour le moment à la classe `vector` qui s'utilise comme un tableau.

```
1   #include <vector>
2
3   std::vector<float> v_headings; // un tableau de décimaux
4   // Un tableau de tableaux d'entiers :
5   std::vector<std::vector<int> > v_v_numbers;
6
7   v_headings.push_back(2.5);
8   v_headings.push_back(0.3);
9
10  float theta0 = v_headings[0]; // theta0 == 2.5
11  double theta1 = v_headings[1]; // cast automatique en double
12
13  int size = v_headings.size(); // size == 2
14  bool is_empty = v_v_numbers.empty(); // is_empty == true
```

Structures de données : classe vector

Pour parcourir une structure de données, on utilise des `iterator`.
Pour la classe `vector`, on peut simplement itérer les valeurs :

```
1   std::vector<float> v_sin;
2
3   // Ajout de valeurs dans le tableau :
4   for(int i = 0 ; i < 100 ; i++)
5   {
6     v_sin.push_back(sin(0.1 * i));
7   } // la variable i n'existe plus
8
9   // Mise à jour de ces valeurs
10  for(int i = 0 ; i < v_sin.size() ; i++)
11    v_sin[i] += M_PI / 2.;
12
13  // Affichage
14  for(int i = 0 ; i < v_sin.size() ; i++)
15    cout << v_sin[i] << endl;
```

Lire les messages du compilateur

Code :

```
1
2     using namespace std;
3
4     int main()
5     {
6       cout << "Hello" << endl;
7       return EXIT_SUCCESS;
8     }
```

Compilation :

```
g++ main.cpp -o output
```

Erreur :

```
main.cpp: In function 'int main()':
main.cpp:6:3: error: 'cout' was not declared in this scope
  cout << "Hello" << endl;
  ^~~~
```

Lire les messages du compilateur

Code :

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    cout << "Hello" << endl;
7    return EXIT_SUCCESS;
8  }
```

Compilation :

```
g++ main.cpp -o output
```

Erreur :

```
main.cpp: In function 'int main()':
main.cpp:6:3: error: 'cout' was not declared in this scope
  cout << "Hello" << endl;
  ^~~~
```

Lire les messages du compilateur (autre exemple)

Code :

```
1  #include "vibes.h"
2  using namespace std;
3
4  int main()
5  {
6    vibes::beginDrawing();
7    return EXIT_SUCCESS;
8  }
```

Compilation :

```
g++ main.cpp -o output
```

Erreur :

```
/tmp/cc7VAZ4D.o: In function ‘main’:
main.cpp:(.text+0x5): undefined reference to ‘vibes::beginDrawing()’
collect2: error: ld returned 1 exit status
```

Lire les messages du compilateur (autre exemple)

Code :

```
1  #include "vibes.h"
2  using namespace std;
3
4  int main()
5  {
6    vibes::beginDrawing();
7    return EXIT_SUCCESS;
8  }
```

Compilation :

```
g++ main.cpp vibes.cpp -o output
```

Erreur :

```
/tmp/cc7VAZ4D.o: In function ‘main’:
main.cpp:(.text+0x5): undefined reference to ‘vibes::beginDrawing()’
collect2: error: ld returned 1 exit status
```