Rappels de C et bases du C++

F. Lassabe

Table des matières

1 Complément du cours

Cette section complète le cours de syntaxe de base du C++. Sa lecture n'est pas requise pour réaliser le TP, mais les éléments abordés seront utilisés lors des TP ultérieurs.

1.1 Le mot-clé auto

Ce mot-clé permet de faire de la déduction de type sous réserve que la déduction puisse être réalisée sans ambiguïté. La déduction de type ne fonctionne pas sur des méthodes virtuelles. La déduction de type est particulièrement intéressante sur des types longs, par exemple des types dépendants de l'espace de nom d'un autre type. En voici un exemple :

```
int a;
auto *p = &a; // déduit int *
auto v = 6; // déduit int
```

```cpp
double f() {/*...*/}
const auto r = f(); // déduit le retour de double
```

1.2 Les boucles sur ensembles

Les boucles sur ensembles sont un type de boucles `for` avec une syntaxe particulière, permettant d'éviter l'usage explicite d'itérateurs ou l'usage d'indices. Par exemplen le code suivant va parcourir une séquence d'entiers dans un vecteur avec une boucle sur ensemble :

```cpp
std::vector<int> v{1, 2, 3, 4};
for (const auto &e: v) {
  std::cout << e << " ";
}
```

Cette syntaxe s'apparente au `foreach` de certains langages. En C++, elle s'appuie sur les itérateurs (*forward iterator*), ainsi que les méthodes begin() et end() qui définissent le début et la fin d'une séquence.

1.3 Les espaces de noms

Les espaces de noms C++ permettent de définir des ensembles de types, classes, etc. Par exemple, l'espace de nom `std` contient des éléments standards du langage, tels que :

- les E/S de base
- les conteneurs de la STL
- les algos standards
- etc.

Pour accéder à un membre d'un `namespace`, utilise l'opérateur "double deux points" `::`. Voici un exemple avec les opérateurs de flux pour afficher à l'écran :

```cpp
#include <iostream>

int main() {
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

Avec les espaces de noms vient un nouveau mot-clé : `using`. Il permet deux choses :

- De chercher automatiquement dans un espace de nom
  - il ne sera plus nécessaire de spécifier le `namespace`
  - attention aux ambiguïtés : on n'utilise pas `using namespace` dans un .h ou un .hpp !
- De déclarer un alias de type
  - Équivalent à typedef en C

Voici un exemple des deux utilisations possibles :

```cpp
#include <iostream>

using namespace std;

using mon_int = int;

int main() {
  mon_int mi = 42;
  cout << "Hello " << mi << endl; // affiche "Hello 42"
  return 0;
}
```

1.4 Les flux d'entrée-sortie

En C++, il est possible de faire des E/S de deux manières : soit en s'appuyant sur les fonctions des bibliothèques C, soit à l'aide des flux définis par la bibliothèque standard C++. Une règle empirique est d'utiliser les outils du C++ dans toute situation, excepté si la bibliothèque standard C++ ne permet pas de

faire ce qu'on veut, ou si la bibliothèque C est nettement plus efficace pour y parvenir (en lignes de code, en temps d'exécution). Être à l'aise avec les bibliothèques C, et mal connaître la bibliothèque standard C++, n'est pas un argument pour utiliser des API C.

La bibliothèque standard C++ propose une hiérarchie de classes de flux entrants et sortants, notamment pour ce qui relève de la saisie au clavier et de l'affichage à l'écran. L'héritage permet également, sous réserve d'une bonne architecture d'application, de facilement adapter des lectures/écritures à des flux différents (de/vers le réseau, des fichiers, l'écran et le clavier).

Pour utiliser les flux de base (clavier et écran en mode texte), il faut :

- inclusion de `iostream`
- `cout` : envoi dans le flux de sortie (≈ stdout du C)
- `cin` : lecture du clavier (≈ stdin de C) dans une variable
- Utilisation avec les opérateurs de flux :
  - `<<` pour les sorties (vers le flux)
  - `>>` pour les entrées (vers des variables)

L'exemple ci dessous permet de lire des valeurs au clavier et de les afficher ensuite à l'écran :

```cpp
#include <iostream>
#include <string>

int main() {
  double d;
  std::string s{};
  std::cout << "Entrez un double: ";
  std::cin >> d;
  std::cout << "Vous avez saisi " << d << std::endl;
  std::cout << "Entrez une chaîne: ";
  std::cin >> s;
  std::cout << "Vous avez saisi " << s << std::endl;
  return 0;
}
```

2 Un programme de base

Pour écrire un programme en C++, vous aurez besoin d'un éditeur de texte pour écrire le code, puis d'un compilateur pour convertir le code écrit en C++ vers un fichier binaire exécutable par votre processeur. Pour les premiers TP, vous utiliserez soit VSCode, soit CLion comme éditeur, et vous compilerez vos programmes à la main. Voyons un programme simple :

```cpp
#include <iostream>

using namespace std;

int main() {
  cout << "Hello, World!" << endl; // Pourrait être: cout << "Hello, World!\n"
  return 0;
}
```

L'exemple présenté consiste en 6 lignes de code :

- `#include <iostream>` inclut le fichier d'entête *iostream*. C'est ce dernier qui permet d'utiliser les entrées sorties de base, et notamment de manipuler *cin* et *cout*.
- `using namespace std;` permet de ne pas précéder les appels à des éléments de *std* par `std::`
- `int main() {` est le début de la fonction *main* du programme (un binaire exécutable contient une et une seule fonction main). La signature de la fonction dans ce programme spécifie que le main retourne une entier, et ne prend aucun paramètre. L'entier retourné est récupéré par le terminal qui a exécuté le programme. En bash, la variable `$?` contiendra la valeur de retour après appel du programme. Une valeur de 0 (zéro) signifie que le programme s'est exécuté sans erreur. Une valeur différente signifie qu'une erreur a eu lieu. C'est à vous de spécifier les valeurs de retour autres que zéro. La ligne se termine sur une accolade ouvrante indiquant le début du code de la fonction *main*.
- `cout << "Hello, World!" << endl;` demande au programme d'afficher la phrase *Hello, World !* (chaîne de caractères, entre doubles quotes), suivie d'un retour à la ligne (*endl*)
- `return 0;` indique que le main retourne la valeur zéro pour utilisation éventuelle par le bash qui a lancé le programme.
- `}` est l'indication que le code de la fonction *main* se termine là. L'accolade fermante "s'équilibre" avec l'accolade ouvrante qui suit la signature du *main*.

Recopiez ce code dans un fichier nommé hello.cpp, puis compilez le avec la commande suivante (depuis le répertoire où hello.cpp a été enregistré) :

```
g++ -std=c++17 hello.cpp -o hello
```

Cette commande consiste à exécuter le compilateur C++ **g++** (un composant de la GCC, GNU Compiler Collection) sur le fichier de code source nommé **hello.cpp** en appliquant le standard C++17 et en produisant un binaire nommé **hello** (grâce à l'option **-o** suivie du nom du binaire généré).

Puis exécutez le programme avec la commande (depuis le répertoire où vous avez compilé) :

```
./hello
```

Le programme doit afficher *Hello, World !* puis revenir à la ligne et rendre la main au shell.

2.1 Exercice : concaténer des chaînes

Modifiez le programme précédent pour ajouter une variable de type *string* à laquelle vous affecterez votre prénom. Faire ensuite en sorte que le programme n'affiche plus *Hello, World !* mais *Hello,* , suivi de votre prénom (lu dans la variable), suivi d'un point d'exclamation. Nommer ce programme *hello-me.cpp*.

2.2 Exercice : saisir le nom

Reprenez l'exercice précédent, mais cette fois, il faut que le nom à afficher soit lu au clavier. Utilisez pour cela *cin* pour saisir la valeur au clavier et la stocker dans la variable contenant le prénom. Nommez le programme *hello-cin.cpp*, compilez le et testez le.

3 Les pointeurs

Dans ce nouvel exercice, vous utiliserez les paramètres pouvant être fournis au programme par la ligne de commande. Il sera nécessaire pour cela de fournir des paramètres à la fonction *main*, qui aura alors la signature suivante :

```
int main(int argc, char *argv[]);
```

Dans ce cas, deux paramètres sont passés au programme par le système :
- **argc** qui contient le nombre d'arguments passés à la fonction
- **argv** qui contient un tableau de chaînes de caractères avec chacun des paramètres (les paramètres sont séparés par des espaces, sauf si plusieurs mots sont encadrés par des double quotes.

Le programme reçoit au minimum un argument, qui est le nom avec lequel il est appelé. Par exemple, la commande

```
./hello-args
```

transmettra au main un argument *argc* de valeur 1 et un argument *argv* qui sera un tableau de un élément contenant *./hello-args*. En invoquant la commande comme suit :

```
./hello-args a b c 123 kjlm
```

le programme recevra comme valeurs de paramètres :
- **argc** égal à 6
- **argv** comme tableau de 6 éléments : `{"./hello-args", "a", "b", "c", "123", "kjlm"}`

Notez bien que tous les éléments de argv sont, à ce stade, des chaînes de caractères. S'ils doivent être considérés de types différents (notamment numériques), il faudra faire appel à des fonctions de conversion de chaîne de caractères vers le type souhaité.

3.1 Exercice : Hello avec arguments

Écrivez un programme qui affichera *Hello,* , suivi de la valeur du premier argument passé après le nom du programme, suivi d'un point d'exclamation. Le programme s'appellera *hello-arg.cpp*.

3.2 Exercice : listage des arguments

Écrivez un programme qui lit les arguments passés au programme et les liste de la manière suivante :
- Sur la première ligne, le nombre d'argument(s) transmis au programme
- Sur les lignes suivantes, un argument par ligne

Le programme s'appelle *liste-args.cpp*.

Modifiez le programme pour que l'affichage ne compte ni n'affiche le nom du programme.

3.3 Passage de paramètres

Dans un programme nommé passage-params.cpp, recopiez le code suivant, puis implémentez une fonction *main* qui utilise les 3 fonctions et teste le résultat produit (avec un affichage des variables).

```
void swap1(int a, int b) {
  int c = a;
  a = b;
  b = c;
}

void swap2(int *a, int *b) {
  int c = *a;
  *a = *b;
  *b = c;
}

void swap3(int &a, int &b) {
  int c = a;
  a = b;
```

```
  b = c;
}
```

Qu'observez vous ? Comment l'expliquer ?

3.4 Valeur, adresse, référence

3.4.1 Introduction

En C, il existe principalement 2 familles de types de données : d'une part, les variables, qui sont des données d'un type concret, avec toutes les propriétés qui y sont liées (arithmétique, etc.), telles que les données numériques ou les structures. D'autre part, les pointeurs, qui sont un type de donnée dont la valeur est une adresse en mémoire correspondant à une autre donnée (quelle qu'elle soit). La langage C++ ajoute aux variables et aux pointeurs un troisième type : la référence. Le principe d'utilisation est similaire au pointeur (la référence indique une autre variable en mémoire), mais en simplifie la syntaxe, c'est à dire qu'on manipule la référence comme si c'était directement la variable, ce qui permet de continuer à utiliser la notation pointée de la POO et de ne pas explicitement référence/dé-référencer un pointeur.

On déclare une référence avec l'opérateur `&` qui précède le nom de la variable référence. Par exemple :

```
int main() {
  int a = 10;
  int &ref = a; // ref "pointe" sur a
  cout << ref << endl; // affiche 10
  return 0;
}
```

Notez que :

- Une référence doit **toujours** être initialisée à la déclaration
- Cet exemple peut être dangereux si a est invalidé alors que ref existe encore
- Les références sont beaucoup utilisées pour le passage de paramètres, comme dans l'exemple suivant :

```
void swap(int *a, int *b) {
  int c = *a;
  *a = *b;
  *b = c;
}
// ...
int a=10, b=42;
swap(&a, &b);

// a le même effet que :

void swap(int &a, int &b) {
  int c = a;
  a = b;
  b = c;
}
// ...
int a=10, b=42;
swap(a, b);
```

3.4.2 Mise en œuvre

Recopiez le code ci-dessous et complétez le dans un fichier nommé parametres.cpp :

```
#include <iostream>
class C {
  public:
  friend std::ostream &operator<<(std::ostream &os, const C &c);
}
```

```
void display_c(const C &c) {
  std::cout << c << std::endl;
}

void display_int(int v) {
  std::cout << v << std::endl;
}

void set_c(C *c) {
  *c = C{};
}

void set_int(int *i) {
  *i = 42;
}

int get_int(int i) {
  return i + 1;
}

C *make_c() {
  return /* your code */;
}

int main() {
  C/* your code */ = make_c(); // variable name is c1
  C c2{};
  display_c(/* your code */); // with c1
  display_c(/* your code */); // with c2
  int i1 = 55;
  int *i2 = /* your code */; // shall get memory for one int and set it to 25
  display_int(/* your code */); // for i1
  display_int(/* your code */); // for i2
  set_int(/* your code */); // for i1
  set_int(/* your code */); // for i2
  std::cout << get_int(/* your code */) << std::endl; // for i1
  std::cout << get_int(/* your code */) << std::endl; // for i2
  set_c(/* your code */); // for c1
  set_c(/* your code */); // for c2
  return 0;
}

std::ostream &operator<<(std::ostream &os, const C &c) {
  os << "I'm a C";
  return os;
}
```

3.5 Pointeurs de fonctions

Cette section est à faire si la section précédente ne vous a pas posé de problème. Dans un fichier nommé ptr_fonctions.cpp, réaliser l'implémentation suivante :

1. Déclarer un main sans paramètre
2. Déclarer une fonction `make_odd` sans paramètre, qui renvoie un pointeur sur un entier impair
3. Déclarer une fonction `make_even` sans paramètre, qui renvoie un pointeur sur un entier pair
4. Dans le main, déclarer un pointeur de fonction correspondant aux signatures de `make_even` et `make_odd`
5. En fonction d'une entrée utilisateur, affecter la bonne fonction au pointeur, l'appeler pour obtenir un nombre et l'afficher.

Vérifier que le programme n'a pas de fuite mémoire avec valgrind.

4 Les classes

4.1 Introduction

La POO en C++ est similaire à celle pratiquée en Java, que vous connaissez déjà. Il y a néanmoins des subtilités de syntaxe et de langage qui différencient le C++ du Java concernant les classes. Une première différence, assez importante, est la capacité de C++ à séparer l'implémentation (la définition) des méthodes d'une classe de sa déclaration (les signatures des méthodes). Cette possibilité est la même que celle proposée par le langage C. Par convention, on place les déclarations dans un fichier d'entête dont le suffixe est `.h` (qui pourra être inclus par ses utilisateurs) et les définitions dans un fichier suffixé `.cpp`.

La seconde différence majeure est qu'il est possible de nommer le fichier contenant une classe d'un nom différent de cette classe. Cela permet également de déclarer plusieurs classes dans une seule paire de fichiers h/cpp. Deux mot-clés peuvent être utilisés pour définir une classe : soit `class`, soit `struct`. Il n'y a pas de différence fondamentale entre les deux si ce n'est que, dans une `class`, tout est par défaut privé, alors que la visibilité par défaut de la `struct` est publique. On peut définir des membres avec une visibilité différente en définissant des sections qui commencent par `public:` (respectivement `protected:`, `private:`) pour que les membres qui suivent (jusqu'à un autre modificateur de visibilité, ou la fin de la classe) soient publiques (respectivement protégés, privés) :

```
class C {
  private: // Inutile : class donc visibilité par défaut private
    // jusqu'à la ligne ci dessous, membres privés
  protected:
    // jusqu'à la ligne ci dessous, membres protégés
  public:
    // jusqu'à la fin de la classe, membres publics
};
```

Il y a deux manière d'instancier un objet d'une classe : en déclarant une variable du type de la classe souhaitée, ou en allouant dynamiquement un objet et en le gérant avec un pointeur (ce qui suppose de ne pas oublier la dé-allocation une fois l'utilisation de l'objet terminée). Cette seconde manière repose sur le couple d'instructions `new` pour créer l'instance, et `delete` pour la dé-allouer. Voici quelques exemples d'instanciations :

```
#include <string>
// ...
string s{"Hello, "}; // on recommande la syntaxe en {} = liste d'init.
QWidget widget{}; // Une classe Qt
string *ptr_s = new string{"World!"};
cout << s << *ptr_s << endl; // Affiche: Hello, World!
delete ptr_s; // NE PAS OUBLIER ! Sinon, fuite mémoire
int *t = new int[100]; // tableau dynamique de 100 entiers
// faire quelque chose avec t
delete [] t; // libérer un tableau: ne pas oublier []
```

Quelques méthodes particulières existent dans une classe C++, notamment les constructeurs, le destructeur, ainsi que l'opérateur d'affectation. Il existe 3 types de constructeurs : le constructeur par défaut (sans paramètre), le constructeur par copie (copie les valeurs d'une autre instance du même type), et le ou les constructeurs avec paramètres, auxquels on fournit des paramètres permettant d'initialiser différemment l'instance. Une méthode peut avoir des paramètres qui ont des valeurs par défaut (elles sont alors indiquées par un `= v` où v est la valeur par défaut). Les paramètres avec valeur par défaut ne peuvent être positionnés qu'après les paramètres sans valeur par défaut dans la liste des paramètres. Voici quelques exemples de constructeurs :

```
1 // class C
2 C(); // Constructeur par défaut
3 C(const C &other); // par copie
4 C(int une_val_init); // avec paramètre
5 C(int une_val = 0); // éq. à ctor défaut + avec paramètre
6   //(ne peut pas exister en même temps que la ligne 2)
```

La seconde méthode capitale en C++ est le destructeur. Il est appelé quand un objet est détruit, soit par l'utilisation de `delete` (objet alloué dynamiquement), ou à la sortie du *scope* de la variable. Quand des ressources gérées par l'instance sont allouées, le destructeur sera chargé de dé-allouer ces ressources (mémoire, ouverture de flux, etc.). Voici un exemple d'utilisation avec de la mémoire dynamique :

```cpp
// Déclaration
class C {
  private:
    int *mon_ptr;

  public:
    C();
    ~C();
};
// Définition
C::C() {
  mon_ptr = new int{42};
}
C::~C() {
  delete mon_ptr;
}

int main() {
  C c{}; // alloue un int
  C *c_ptr = new C{};
  delete c_ptr; // Delete : appel du destructeur -> libère le int
  return 0; // Sortie du scope : appel du destructeur -> libère le int
}
```

Il est également possible avec C++ de procéder à la surcharge des opérateurs, qui consiste à définir le comportement des opérateurs lorsqu'ils sont appliqués sur des instances d'une classe car il n'y a pas d'implémentation automatique possible pour la plupart des opérateurs, et que l'implémentation par défaut n'est pas nécessairement celle qui est souhaitable. Voici les déclarations de quelques opérateurs C++ pouvant être surchargés :

```cpp
C &operator=(const C &other); // affectation
C operator+(const C &other); // addition/concaténation
bool operator<(const C &other); // Comparaison
// Certains opérateurs (sauf () -> = et []) déclarables hors classe :
C operator+(const C &lhs, const C &rhs);
bool operator<(const C &lhs, const C &rhs);
// etc.
```

La définition de ces opérateurs s'effectue ensuite comme pour n'importe quelle autre méthode. La conjonction de ces 3 types de méthodes particulières permet de définir la forme de Coplien pour une classe. Cette forme est une classe comprenant :

- le constructeur par défaut
- le destructeur
- le constructeur par copie
- l'opérateur d'affectation

Par exemple :

```cpp
class C {
  private:
    string s;
  public:
    C(): s("hello") {}
    C(const C &other): s(other.s) {}
    ~C() {}
    C &operator=(const C &other);
};

C &C::operator=(const C &other) {
  if (this != &other) {
    s = other.s;
  }
```

```
  return *this;
}
```

4.2 Application au C++

Pour cette partie, vous vous baserez sur l'exemple ci dessous :

```
#include <iostream>

class Mobile {
  double x;
  double y;
public:
  Mobile(const double x=0.0, const double y=0.0): x(x), y(y) {}
  Mobile(const Mobile &other);
  ~Mobile() {}
  Mobile &operator=(const Mobile &other);
  friend std::ostream &operator<<(std::ostream &os, const Mobile &m);
};

Mobile::Mobile(const Mobile &other): x(other.x), y(other.y) {
}

Mobile &Mobile::operator=(const Mobile &other) {
  if (*this != other) {
    x = other.x;
    y = other.y;
  }
  return *this;
}

std::ostream &operator<<(std::ostream &os, const Mobile &m) {
  os << x << "," << y;
  return os;
}

int main() {
  Mobile m1{1.0, 42.0};
  Mobile m2{m1};
  Mobile m3;
  m3 = m4;
  std::cout << m1 << " " << m2 << " " << m3 << " " << m4 << std::endl;
  return 0;
}
```

Vous y remarquerez notamment que les méthodes définies hors de la déclaration de la classe sont préfixées par `Mobile::`, c'est à dire le nom de la classe dont on définit une méthode suivi de `::`, puis du membre défini (devant correspondre à un membre déclaré).

4.3 Une classe simple

À partir de l'exemple, implémentez une classe `Piece` dont les attributs sont :

- `name`, de type `std::string`, contenant le nom de la pièce (pion, roi, etc.)
- `column`, de type int, contenant le numéro de la colonne où se situe la pièce (entre 1 et 8)
- `line`, de type int, contenant le numéro de la colonne où se situe la pièce (entre 1 et 8)
- Un constructeur par défaut qui crée un pion aux coordonnées (2, 1)
- Un constructeur paramétré (peut être fusionné avec le constructeur par défaut avec des paramètres ayant des valeurs par défaut)
- Un constructeur par copie
- Un opérateur d'affectation

— Un destructeur
— Un opérateur de flux sortant

4.4 Une classe avec mémoire gérée

Implémentez maintenant une autre classe, que l'on nommera `array_int`. L'objectif de cette classe est de fournir l'accès à un tableau d'entiers. Cette classe a 2 attributs :
— `size`, la taille du tableau
— `content`, la mémoire disponible pour le tableau. C'est un type `int *`
Vous implémenterez les méthodes suivantes :
— Constructeur par défaut. Initialise le tableau à une taille de 10.
— Constructeur avec paramètre : le paramètre spécifie la taille du tableau en mémoire.
— Destructeur : pensez à libérer la mémoire.
— Constructeur par copie : fait une copie profonde (i.e. alloue autant de mémoire que la source, et recopie le tableau de la source)
— Affectation
— Flux sortant
— Accès positionnel : opérateur `[]` pour accéder à un élément du tableau par sa position.
Les constructeurs (sauf par copie) doivent initialiser les éléments du tableau à zéro. Testez le programme, et assurez vous qu'il n'y a pas de fuite mémoire avec Valgrind.

5 Héritage

5.1 Présentation

L'héritage permet d'étendre une classe en lui ajoutant des membres et/ou des méthodes. L'extension se fait en définissant une nouvelle classe qui sera basée sur la classe qu'on souhaite étendre. On nommera la nouvelle classe **classe enfant** ou **classe fille** et la classe sur laquelle elle se base **classe parente** ou **classe mère**. On dira que **la classe fille hérite de la classe mère**. La classe fille possède les membres de sa classe parente ainsi que ceux qui lui sont ajoutés. Une classe peut hériter de plusieurs classes à la fois en C++.
La structure en classes et l'héritage permettent le polymorphisme :

Polymorphisme

Le polymorphisme est la capacité pour les objets par sous-classe à se comporter de façon différente. Cette notion est en étroite relation avec la redéfinition des méthodes d'une classe enfant. Un pointeur sur une classe mère peut pointer sur une des classes filles, le polymorphisme permettant l'appel de la méthode correspondant à la classe effectivement instanciée.

Avec le polymorphisme, on peut déclarer un type pointeur sur une classe parente, puis instancier un objet d'une classe enfant, et avoir le comportement spécialisé. Par exemple, le parent peut être de type *Shape* et les instances référencées par les pointeurs sur *Shape* être en réalité de type *Square*, *Rectangle*, etc. (**Square** et **Rectangle** sont des enfants de **Shape**). L'appel des méthodes à partir des pointeurs aboutira aux méthodes définies dans les enfants. Voici un autre exemple pour mieux comprendre :

```
class vehicle {
  public:
  virtual void say_hello() {cout << "Hello, I'm a";}
};

class car: public vehicle { // héritage
  public:
  void say_hello()override {vehicle::say_hello(); cout << " car.\n";}
};

class plane: public vehicle { // héritage
  public:
  void say_hello()override {vehicle::say_hello(); cout << " plane.\n";}
};
```

```
int main() {
  vehicle *v1 = new car{};
  vehicle *v2 = new plane{};
  v1->say_hello();
  v2->say_hello();
  delete v1;
  delete v2;
  return 0;
}
```

Cet exemple introduit deux nouveaux éléments :

— Le mot-clé `virtual` : devant une signature de fonction, il indique que la méthode (non statique) est virtuelle. Le compilateur cherche si elle est redéfinie dans les classes filles, et appelle la méthode redéfinie quand elle existe. Ce mot-clé est obligatoire pour permettre le polymorphisme.

— Le mot-clé `override` : il indique qu'on redéfinit une méthode virtuelle. Il est facultatif mais générera une erreur de compilation s'il est présent dans la déclaration d'une méthode non virtuelle. L'utilisation de `override` est également recommandée pour la qualité et la compréhension du code.

Un dernier mot-clé complète `virtual` et `override`. il s'agit du mot-clé `final`. Il ne peut figurer que sur une méthode virtuelle, pour indiquer au compilateur que cette méthode ne pourra plus être redéfinie dans les classes dérivées.

Il existe également des classes et des méthodes virtuelles pures. On indique une méthode virtuelle pure par un `=0` après sa signature. Il n'est alors pas nécessaire de définir la méthode (mais on le peut quand même, et on le doit quand il s'agit du destructeur). À partir du moment où il existe au moins une méthode virtuelle pure, alors la classe devient également virtuelle pure et elle ne peut plus être instanciée. Seules des classes enfants qui définissent la ou les méthodes virtuelle(s) pure(s) de leur parent pourront être instanciées. Il est en revanche possible de déclarer un pointeur sur une classe virtuelle pure, permettant le polymorphisme à partir de celle-ci. Voici un exemple de classe virtuelle pure :

```
class C {
  public:
  C();
  ~C();
  do_something() = 0; // virtuelle pure
};
class C_child: public C {
  public:
  do_something(); // implémentée
};
/* ... */
C c{}; // Erreur : C ne peut être instanciée
C *ptr_c; // Ok
C *ptr_c2 = new C{}; // Erreur : C ne peut être instanciée
C *ptr_c3 = new C_child{}; // Ok
```

5.2 Héritage simple

Pour cet exercice ; vous définirez deux classes :

— point2D :
  — Attributs : x et y, de type double
  — Constructeurs : par défaut, avec paramètres pour x et y, par copie
  — Destructeur
  — Méthode distance avec un paramètre point2D
  — Opérateur de flux de sortie

— point3D, qui hérite de point2D :
  — Ajoute un attribut z de type double
  — Constructeurs : par défaut, avec paramètres pour x et y, par copie
  — Destructeur
  — Méthode distance avec un paramètre point2D (considère $z = 0$ pour le point2D)
  — Méthode distance avec un paramètre point3D

— Opérateur de flux de sortie

Vous testerez ces deux classes avec le main suivant (aucune modification ne doit lui être apportée) :

```
int main() {
  point2D p0{2, 42};
  point2D p1{1.0, 2.0};
  point2D p2{p1};
  point3D p3{1, 2, 3};
  point3D p4{p3};
  point3D p5{p2};
  cout << "Distance " << p0 << " " << p2 << " = " << p0.distance(p2) << endl;
  cout << "Distance " << p3 << " " << p5 << " = " << p3.distance(p5) << endl;
  cout << "Distance " << p3 << " " << p0 << " = " << p3.distance(p0) << endl;
  return 0;
}
```

5.3 Polymorphisme

Pour ce premier test de polymorphisme, vous utiliserez le main fourni ci dessous :

```
int main() {
  point2D p0{2, 42};
  point2D p1{1.0, 2.0};
  point2D p2{p1};
  point3D p3{1, 2, 3};
  point3D p4{p3};
  point3D p5{p2};
  cout << "Distance " << p0 << " " << p2 << " = " << p0.distance(p2) << endl; // affiche 40.0125
  cout << "Distance " << p3 << " " << p5 << " = " << p3.distance(p5) << endl; // 3
  cout << "Distance " << p3 << " " << p0 << " = " << p3.distance(p0) << endl; // 40.1248
  return 0;
}
```

Si le résultat n'est pas exact, corrigez votre code en rendant distance virtuelle.

5.4 Méthodes virtuelles

Utilisez le main suivant :

```
int main() {
  vector<point2D *> v{new point2D{1, 2}, new point3D{2, 3, 4}, new point3D{3, 4, 5}, new point2D{4, 5}};
  for (const auto &p: v) {
    cout << *p << endl;
    delete p;
  }
  return 0;
}
```

Tous les points seront affichés comme point2D. Pour obtenir un affichage correct, la boucle sera modifiée comme suit :

```
for (const auto &p: v) {
  p->display();
  cout << endl;
  delete p;
}
```

Vous adapterez les deux classes pour permettre au code de fonctionner.

5.5 Classe virtuelle pure

Vous construirez pour cet exercice une hiérarchie de classes, dont certaines seront purement virtuelles :

- `vehicule` (virtuel pure) :
  - 2 attributs : x et y (double) la position
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getters/setters de x et y
  - Méthode `display`, affiche "vehicule (X, Y) de type " (remplacer X, Y par les valeurs de l'instance)
- `terrestre` (virtuel pure), hérite de vehicule
  - Un nouvel attribut : type_moteur, de type motorisation_t (enum contenant ESSENCE, DIESEL, GPL, ELECTRIQUE)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getter et setter de type_moteur
  - Méthode `display`, affiche "terrestre (moteur T) : " (remplacer T par le mot correspondant à la motorisation)
- `aerien` (virtuel pure), hérite de vehicule
  - 2 nouveaux attributs : altitude (double), et nb_passagers (int)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getters/setters de altitude et nb_passagers
  - Méthode `display`, affiche "aérien (A m, N passagers) : " (remplacer A et M par les valeurs de l'instance)
- `voiture` (hérite de terrestre)
  - 2 attributs en plus : puissance_chevaux (double), nb_portes (int)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getters/setters de puissance_chevaux et nb_portes
  - Méthode `display`, affiche "voiture (P portes, C chevaux)" (remplacer P et C par les valeurs de l'instance)
- `camion` (hérite de terrestre)
  - 2 attributs en plus : capacite_m3 (double), capacite_tonnes (double)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getters/setters de capacite_m3 et capacite_tonnes
  - Méthode `display`, affiche "camion (M m3, T tonnes)" (remplacer M et T par les valeurs de l'instance)
- `avion` (hérite de aerien)
  - 1 attribut en plus : envergure (double)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getter/setter de envergure
  - Méthode `display`, affiche "avion (E m d'envergure)" (remplacer E par la valeur de l'instance)
- `helicoptere` (hérite de aerien)
  - 1 attribut en plus : nb_rotors (int)
  - Constructeurs (défaut, paramétré, copie), destructeur virtuel pur, opérateur d'affectation
  - Getter/setter de nb_rotors
  - Méthode `display`, affiche "hélicoptère (R rotor(s))" (remplacer R par la valeur de l'instance)

Écrivez et testez le code avec le main suivant :

```
int main() {
  vehicule **vehicules = new vehicule*[5];
  vehicules[0] = new voiture{5};
  vehicules[1] = new avion{};
  vehicules[2] = new voiture{(voiture&) (*vehicules[0])};
  vehicules[3] = new camion{};
  vehicules[4] = new helicoptere{};
  for (int i=0; i<5; ++i)
    vehicules[i]->display();
  for (int i=0; i<5; ++i)
    delete vehicules[i];
  delete v2;
  delete []vehicules;
  return 0;
}
```