République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur

et de la Recherche Scientifique

Université de Tissemsilt

Faculté des Sciences et de la Technologie

Département des Sciences et de la Technologie

Polycopié de cours

Programmation Orientée Objet en C++

Filière : Électronique
Spécialité : Master Instrumentation

Préparé par : Dr. HAMDANI Mostefa
Maître de conférences classe « B »

Tissemsilt - 2021/2022

Avant propos

Ce polycopié de cours "Programmation Orientée Objet en C++" s'adresse aux étudiants de la première année Master "Instrumentation". Le manuscrit est composé de six chapitres :

- Introduction à la programmation orientée objets
- Notions de base
- Classes et objets
- Héritage et polymorphisme
- Les conteneurs, itérateurs et foncteurs
- Notions avancées

L'objectif principal de ce polycopié est de mettre à la disposition des étudiants un support de cours avec des travaux pratiques afin qu'ils puissent apprendre les fondements de base de la programmation orientée objet ainsi que la maitrise des techniques de conception des applications en langage C++.

Pré-requis : Les connaissances préalables recommandées sont l'algorithmique et la programmation procédurale entre autre la programmation en C.

Canevas

Objectifs de l'enseignement :
Apprendre à l'étudiant les fondements de base de la programmation orientée objets ainsi que la maitrise des techniques de conception des programmes avancés en langage C++.

Connaissances préalables recommandées : Programmation en langage C.

Contenu de la matière :

Chapitre 1. Introduction à la POO **(2 semaines)**
Principe de la POO, Définition du langage C++, Mise en route de langage C++, Le noyau C du langage C++.

Chapitre 2.Notions de base **(2 semaines)**
Les structures de contrôle, Les fonctions, Les tableaux, La récursivité, Les fichiers, Pointeurs, Pointeurs et références, Pointeurs et tableaux, L'allocation dynamique.

Chapitre 3. Classes et objets **(3 semaines)**
Déclaration de classe, Variables et méthodes d'instance, Définition des méthodes, Droits d'accès et encapsulation, Séparations prototypes et définitions, Constructeur et destructeur, Les méthodes constantes, Association des classes entre elles, Classes et pointeurs.

Chapitre 4. Héritage et polymorphisme **(3 semaines)**
Héritage, Règles d'héritage, Chaînage des constructeurs, Classes de base, Préprocesseur et directives de compilation, Polymorphisme, Règles à suivre, Méthodes et classes abstraites, Interfaces, Traitements uniformes, Tableaux dynamiques, Chaînage des méthodes, Implémentation des méthodes virtuelles, Classes imbriquées.

Chapitre 5. Les conteneurs, itérateurs et foncteurs **(3 semaines)**
Les séquences et leurs adaptateurs, Les tables associatives, Choix du bon conteneur,

Itérateurs : des pointeurs boostés, La pleine puissance des list et map, Foncteur : la version objet des fonctions, Fusion des deux concepts.

Chapitre 6. Notions avancées **(2 semaines)**

La gestion des exceptions, Les exceptions standard, Les assertions, Les fonctions templates, La spécialisation, Les classes templates.

TP Programmation orientée objet en C++

- TP1 : Maitrise d'un compilateur C++
- TP2 : Programmation C++
- TP3 : Classes et objets
- TP4 : Héritage et polymorphisme
- TP5 : Gestion mémoire
- TP6 : Templates

Mode d'évaluation : Contrôle continu : 40% ; Examen : 60%.

Références bibliographiques :

1. Bjarne Stroustrup (auteur du C++), Le langage C++, Pearson.
2. Claude Delannoy, Programmer en langage C++, 2000.
3. Bjarne Stroustrup, Le Langage C++, Pearson Education France, 2007.
4. P.N. Lapointe, Pont entre C et C++ (2ème Édition), Vuibert, Edition 2001.

Table des matières

Avant Propos **3**

Canevas **i**

Table des matières **iii**

Table des figures **vi**

Liste des Algorithmes **viii**

Glossaire **ix**

Introduction **1**
- 1 Introduction . . . 1
- 2 Pourquoi apprendre le C++ ? . . . 1
- 3 Installation et configuration de l'environnement de programmation . . . 2
  - 3.1 IDE (Integrated Development Environment) . . . 3
    - 3.1.1 Microsoft © Visual Studio Express ® . . . 3
    - 3.1.2 Eclipse . . . 3
    - 3.1.3 Code : :Blocks . . . 4
    - 3.1.4 XCode . . . 4
  - 3.2 Des outils en ligne . . . 4

Chapitre 1 Introduction à la POO **5**
- 1.1 Introduction . . . 5
- 1.2 Problématique de la programmation . . . 6
- 1.3 Programmation structurée . . . 6
- 1.4 Programmation Orientée Objet . . . 6
  - 1.4.1 Objet . . . 7
  - 1.4.2 Encapsulation . . . 7
  - 1.4.3 Classe . . . 7
  - 1.4.4 Héritage . . . 7

1.4.5 Polymorphisme . . . 7
1.5 Language C++ . . . 8
1.6 Premier programme avec C++ . . . 8

Fiche TD 01 - Chapitre 1. Introduction à la POO **11**

Chapitre 2 Notions de base **12**
2.1 Introduction . . . 12
2.2 Variables en C++ . . . 12
2.2.1 Noms des variables . . . 12
2.2.2 Type de variables . . . 13
2.2.3 Mots-clés – pourquoi sont-ils les clés ? . . . 14
2.3 Structures de contrôle . . . 15
2.3.1 Traitement conditionnel . . . 15
2.3.1.1 L'instruction if . . . 15
2.3.1.2 L'instruction switch . . . 16
2.3.2 La boucle While . . . 18

Fiche TD 01 - Chapitre 2. Notions de base **20**

Chapitre 3 Classes et objets **22**
3.1 Classe . . . 22
3.1.1 Déclaration de classe . . . 23
3.1.2 Les méthodes . . . 26
3.1.3 Constructeur et destructeur . . . 27
3.1.3.1 Règles de destructeur : . . . 31
3.1.4 Variables automatiques / statiques . . . 32
3.1.5 Droits d'accès et encapsulation . . . 33

Chapitre 4 Héritage et polymorphisme **35**
4.1 L'héritage . . . 35
4.1.1 L'héritage simple . . . 37
4.1.2 L'héritage multiple . . . 38
4.1.2.1 Mise en œuvre de l'héritage multiple . . . 38

Chapitre 4. TD 01 - Héritage et polymorphisme **41**

Chapitre 5 Les conteneurs, itérateurs et foncteurs **43**
5.1 Conteneur . . . 43
5.2 Itérateur . . . 44

5.2.1 begin - end 44
5.2.2 advance 45
5.2.3 next - prev 45
5.2.4 distance 46

Chapitre 6 Notions avancées **48**
6.1 Gestion des exceptions 48
6.2 Syntaxe des exceptions 49
6.3 Les assertions 49
6.4 Les fonctions templates 50
6.5 Classe Template 52

Bibliographie **53**

Table des figures

1.1 Programme Hello World 10

2.1 Keywords 14

3.1 Exemple de classe 23

4.1 Héritage 36
4.2 Héritage 38
4.3 Héritage Multiple - Problème 39

Liste des Algorithmes

1.1 Programme Hello World ! 8

2.1 L'instruction if 15
2.2 Exemple - Instruction If 16
2.3 Exemple 2 - Instruction If 16
2.4 Instruction Switch 17
2.5 Instruction While 18
2.6 Exercice 1 - MAX 20
2.7 Exercice 2 - MAX of three numbers 21

3.1 Déclaration de classe 25
3.2 Exemple de classe comportant un constructeur 28
3.3 classe point 29
3.4 Destructeur 31
3.5 Variables automatiques / statiques 33

4.1 Héritage Simple 37
4.2 Héritage Multiple 40
4.3 Héritage + Protected - 1 41
4.4 Héritage + Protected - 2 42

5.1 Itérateur begin - end 44
5.2 Itérateur - advance 45
5.3 Itérateur - next - prev 45
5.4 Itérateur - next - prev - 2 46
5.5 Itérateur - distance -1 46
5.6 Itérateur - distance -2 47

6.1 throw, try, catch 49
6.2 Assertions 49
6.3 Fonctions templates - 1 50
6.4 Fonctions templates - 2 51

6.5 Fonctions templates - max 51
6.6 Classe template - 1 52

Glossaire

IDE Integrated Development Environment 3

GCC GNU Compiler Collection 4

GNU GNU's Not Unix 4

GPL GNU General Public License 4

GUI Graphical User Interface 3

POO Programmation Orientée Objet 6

Introduction Générale

Sommaire

1	Introduction	1
2	Pourquoi apprendre le C++ ?	1
3	Installation et configuration de l'environnement de programmation	2

1 Introduction

C++ est un langage de programmation informatique puissant qui convient aux personnes orientées vers la technique avec peu ou pas d'expérience en programmation, et aux programmeurs expérimentés à utiliser dans la construction de systèmes d'information substantiels.

Construire des logiciels rapidement, correctement et économiquement reste un objectif difficile à atteindre à une époque où la demande de nouveaux logiciels plus puissants monte en flèche.

2 Pourquoi apprendre le C++ ?

le C++ a les avantages [1] :

- Sa popularité : le C++ est un langage qui est utilisé dans de nombreux projets important (citons Libre Office , 7-zip ou encore KDE). Il est au programme de

beaucoup de formations informatiques. Il possède une communauté très importante, beaucoup de documentation et d'aide, surtout sur l'internet anglophone.

- Sa rapidité : C++ offre un grand contrôle sur la rapidité des programmes. C'est cette caractéristique qui fait de lui un des langages de choix pour les programmes scientifiques, par exemple.
- Sa facilité d'apprentissage : depuis sa version de 2011, C++ est beaucoup plus facile à apprendre que par le passé. Et ça tombe bien, c'est sur cette version et les suivantes que va se baser ce cours.
- Son ancienneté : C++ est un langage ancien d'un point de vue informatique (30 ans, c'est énorme), ce qui donne une certaine garantie de maturité, de stabilité et de pérennité (il ne disparaîtra pas dans quelques années).
- Son évolution : C++11 est un véritable renouveau de C++, qui le rend plus facile à utiliser et plus puissant dans les fonctionnalités qu'il offre aux développeurs. C++14 et C++17 améliorent encore la chose.
- Il est multi-paradigme : il n'impose pas une façon unique de concevoir et découper ses programmes mais laisse le développeur libre de ses choix, contrairement à d'autres langages comme Java ou Haskell .

Bien entendu, tout n'est pas parfait et C++ a aussi ses défauts [1].

- Son héritage du C : C++ est un descendant du langage C, inventé dans les années 1970. Certains choix de conception, adaptés pour l'époque, sont plus problématiques aujourd'hui, et C++ les traine avec lui.
- Sa complexité : il ne faut pas se le cacher, avoir une certaine maîtrise du C++ est très long et demandera des années d'expérience, notamment parce que certaines des fonctionnalités les plus puissantes du C++ requièrent de bien connaître les bases.
- Sa bibliothèque standard : bien qu'elle permette de faire beaucoup de choses (et d'ailleurs, nous n'aurons même pas le temps d'en faire un tour complet dans ce cours), elle n'offre pas de mécanisme natif pour manipuler des bases de données, faire des programmes en fenêtres, jouer avec le réseau, etc. Par rapport à d'autres langages comme Python , C++ peut paraître plus «limité».

3 Installation et configuration de l'environnement de programmation

Généralement, nous pouvons utiliser deux méthodes :

— En utilisant un Integrated Development Environment (IDE) installé localement,

— à l'aide des outils qui existent en ligne.

3.1 IDE (Integrated Development Environment)

IDE est une application logicielle qui se compose généralement d'un éditeur de code, d'un compilateur, d'un débogueur et d'un constructeur de Graphical User Interface (GUI).

Programmer avec un IDE présente de nombreux avantages : vous obtenez une boîte à outils contenant tout ce dont vous pourriez avoir besoin. Les vrais programmeurs utilisent généralement aussi un IDE. Un IDE vous offre un bureau confortable équipé de tous les moyens, fournitures et aides.

Il y a aussi quelques inconvénients. Les bureaux confortables pèsent généralement beaucoup. Les IDE aussi. Ils peuvent consommer beaucoup de ressources et, franchement, vous n'avez probablement pas besoin de la plupart des fonctions qu'ils peuvent exécuter.

Il existe de nombreux IDE sur le marché, à la fois gratuits et payants :

3.1.1 Microsoft © Visual Studio Express ®

Un environnement de développement à plate-forme unique spécialement conçu pour créer des programmes C/C++, à la fois sous et pour le système d'exploitation MS Windows.

- Site d'accueil : `https://www.visualstudio.com`
- Téléchargement : `https://www.visualstudio.com/products/free-developer-offers-vs.aspx`
- Licence : une version gratuite propriétaire mais limitée nommée Visual Studio Community est disponible en téléchargement ; enregistrement requis.

3.1.2 Eclipse

Environnement de développement multiplateforme spécialement conçu pour Java. Programmation C++ possible sans configuration supplémentaire (version dédiée C/C++ disponible en téléchargement).

- Site d'accueil : `https://netbeans.org`

- Téléchargement : https://netbeans.org/downloads/index.html
- Licence :Common Development and Distribution License ou GNU General Public License (GPL) version 2 (gratuite et ouverte).

3.1.3 Code : :Blocks

Environnement de développement multipla-teforme conçu pour la programmation C/C++. Le programme d'installation Windows par défaut n'inclut pas le compilateur C++ - utilisez plutôt celui contenant "mingw-setup" dans le nom du fichier

- Site d'accueil : http://www.codeblocks.org
- Téléchargement : http://www.codeblocks.org/downloads/binaries
- Licence :Common Development and Distribution Licence publique GNU's Not Unix (GNU) version 3 (gratuite et ouverte)

3.1.4 XCode

Environnement de développement à plate-forme unique spécialement conçu pour créer des applications pour les systèmes d'exploitation conçus par Apple Inc. Programmation en C++ entièrement disponible.

- Site d'accueil : https://developer.apple.com/xcode
- Téléchargement : https://developer.apple.com/xcode/download
- Licence : propriétaire mais gratuit pour les utilisateurs de Max OS X ; intégré à OS X et pré-installé.

3.2 Des outils en ligne

Aujourd'hui'hui, grâce à Internet, nous pouvons accéder à de nombreuses ressources dont des compilateurs C++ dans leur dernière version. L'un deux s'appelle Wandbox [1] . Ce site fournit des compilateurs en ligne pour C++ qui sont parmi les plus connus : Clang et GNU Compiler Collection (GCC) [1].

1. https ://melpon.org/wandbox/

CHAPITRE 1

Introduction à la POO

Sommaire

1.1 Introduction 5
1.2 Problématique de la programmation 6
1.3 Programmation structurée 6
1.4 Programmation Orientée Objet 6
1.5 Language C++ 8
1.6 Premier programme avec C++ 8

1.1 Introduction

Les développeurs de logiciels découvrent que l'utilisation d'une approche de conception et de mise en œuvre modulaire et orientée objet peut rendre les groupes de développement de logiciels beaucoup plus productifs qu'il n'était possible avec des techniques populaires antérieures telles que la « programmation structurée » - les programmes orientés objet sont souvent plus faciles à comprendre, à corrige, à corriger et à modifier.

Construire des logiciels rapidement, correctement et économiquement reste un objectif difficile à atteindre à une époque où la demande de nouveaux logiciels plus puissants monte en flèche. Les objets, ou plus précisément les classes dont sont issus les objets,

sont essentiellement des composants logiciels réutilisables [2].

1.2 Problématique de la programmation

La programmation vise à produire des logiciels avec des exigences de qualité qu'on tente de mesurer suivant certains critères, notamment [3] :

- l'exactitude : aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation (par exemple, données correspondant aux spécifications) ;
- la robustesse : aptitude à bien réagir lorsque l'on s'écarte des conditions normales d'utilisation ;
- l'extensibilité : facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications ;
- la réutilisabilité : possibilité d'utiliser certaines parties (modules) du logiciel pour résoudre un autre problème ;
- la portabilité : facilité avec laquelle on peut exploiter un même logiciel dans différentes implémentations ;
- l'efficience : temps d'exécution, taille mémoire...

1.3 Programmation structurée

En programmation structurée, un programme est formé de la réunion de différentes procédures et de différentes structures de données, généralement indépendantes de ces procédures (équation de Wirth, créateur de Pascal) [3] :

$$\text{Programmes} = \text{algorithmes} + \text{structures de données} \tag{1.1}$$

1.4 Programmation Orientée Objet

La Programmation Orientée Objet (POO) est axée sur la conception, la mise en œuvre et l'utilisation classes hiérarchiques [4].

1.4.1 Objet

L'idée de base de la POO est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

$$\text{Objet} = \text{Méthodes} + \text{Données} \tag{1.2}$$

1.4.2 Encapsulation

qu'il n'est pas possible d'agir directement sur les données d'un objet ; il est nécessaire de passer par l'intermédiaire de ses méthodes, qui jouent ainsi le rôle d'interface obligatoire [3].

1.4.3 Classe

Une classe est le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable [5].

1.4.4 Héritage

L'héitage permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données et de nouvelles méthodes [3].

1.4.5 Polymorphisme

Le polymorphisme, conséquence directe de l'héritage, permet à un même message, dont l'existence est prévue dans une super-classe, de s'exécuter différemment, selon que l'objet qui le reçoit est d'une sous-classe ou d'une autre. Cela permet à l'objet responsable de l'envoi du message de ne pas avoir à se préoccuper dans son code de la nature ultime de l'objet qui le reçoit et donc de la façon dont il l'exécutera [6].

1.5 Language C++

C++ a été développé à partir du langage de programmation C et, à quelques exceptions près, conserve C comme sous-ensemble [4]. Le langage C++ a été conçu à partir de 1982 par Bjarne Stroustrup (*AT&T Bell Laboratories*), dès 1982, comme une extension du langage C, lui-même créé dès 1972 par Denis Ritchie, formalisé par Kerninghan et Ritchie en 1978. L'objectif principal de B. Stroustrup était d'ajouter des classes au langage C et donc, en quelque sorte, de « greffer » sur un langage de programmation procédurale classique des possibilités de « programmation orientée objet » [3].

1.6 Premier programme avec C++

Nous allons écrire et comprendre le premier programme en C++. Nous écrivons un programme C++ simple qui affiche le message "Hello World!" . Voyons d'abord le programme, puis nous discuterons chaque partie en détail

Algorithme 1.1 : Programme Hello World!

```
1   /*
2           Commentaires
3           sur plusieurs lignes
4     */
5   #include<iostream>
6
7   //Commentaire sur une seule ligne
8   using namespace std;
9
10  //C'est ici que commence l'exécution du programme
11  int main()
12  {
13    // affiche Hello World ! à l'écran
14    cout<<"Hello World!"<< endl;
15
16    return 0;
17  }
```

1. **Commentaires :** Les commentaires sont complètement ignorés par le compilateur, ils ne servent qu'aux humains. On s'en sert pour documenter son code,

expliquer des passages un peu ardus, décrire des passages un peu moins lisibles ou tout simplement pour offrir quelques compléments d'information [1].

Il existe deux types de commentaires :

- les commentaires « libres » (sur plusieurs lignes), hérités du langage C ;
- les commentaires de fin de ligne (introduits par C++).

2. **#include<iostream>**, acronyme de «Input Output Stream», Ce fichier fait partie de la bibliothèque standard C++, un ensemble de fonctionnalité déjà précodées et incluses partout avec chaque compilateur C++. Pour utiliser les fonctionnalités offertes par ce fichier, notamment écrire un message, on doit l'importer dans notre programme. On dit qu'on l'inclut, d'où l'anglais «*include*». «#» est une directive de préprocesseur. Le préprocesseur est un programme qui se lance automatiquement au début de la compilation, notamment pour importer les fichiers qu'on lui demande [1].

3. **using namespace std; :** Les espaces de noms sont principalement utilisés pour organiser des composants de programme plus importants, tels que des bibliothèques. Ils simplifient la composition d'un programme à partir de parties développées séparément [4].

4. **int main() :** fait partie de chaque programme C++. Les parenthèses après *main* indiquent que *main* est un bloc de construction de programme appelé une *fonction*. Les programmes C++ consistent généralement en une ou plusieurs fonctions et classes. Exactement une fonction dans chaque programme doit être nommée *main* [2]. Quant l'ordinateur exécute un programme, il lui faut bien un endroit où commencer, un début, un point d'entrée. À partir de ce point d'entrée, il exécutera des instructions, des ordres, que nous aurons au préalable écrits. Cet endroit s'appelle main et est la fonction principale de tout programme C++ [1].

5. **cout « "Hello World!"; :** demande à l'ordinateur d'effectuer une action, à savoir d'imprimer la chaîne de caractères contenue entre les guillemets doubles. Un *string* est parfois appelé une chaîne de caractères ou un littéral de chaîne. Nous nous référons aux caractères entre guillemets doubles simplement comme des chaînes. Les espaces blancs dans les chaînes ne sont pas ignorés par le compilateur [2].

6. **endl; :** ermet d'aller à la ligne, comme son nom l'indique («end line» que l'on traduit par «retour à la ligne»).

7. **return 0; :** Au système d'exploitation. Le zéro, par convention, signifie que tout s'est bien passé [1]. Nous aurons l'occasion de comprendre mieux tout cela quand nous aborderons le chapitre sur les fonctions.

La figure 1.1 représente l'interface graphique programme précédent (Alg. 4.4) écrit dans Microsoft Visual Studio.

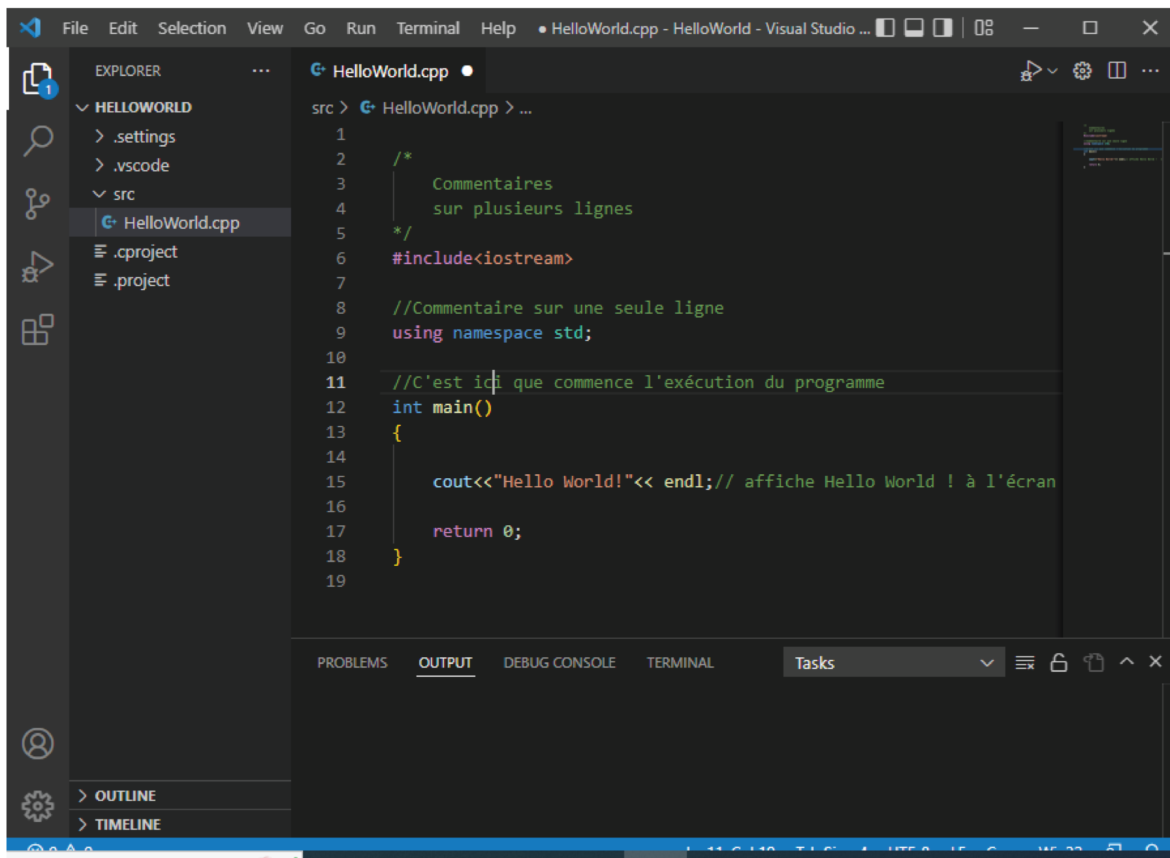

Figure 1.1 – Programme Hello World

Fiche TD 01 - Chapitre 1. Introduction à la POO

Questions (*rappel*)

- C ++ est un language de pragrammation purement orientée objet. Vrai/Faux ? Justifier.
- Quelles est la différence entre la POO et la programmation structurée ?
- Quelles sont les avantages de la POO ?
- Quelles sont les principes de base de la POO ?

Exercice

- Écrire un programme qui permet d'afficher le message suivant : *Bonjour*
- Expliquer le rôle de :
  - #include<iostream> ;
  - using namspace std ;
- Le « **;** » est il facultatif en C++ ?

CHAPITRE 2

Notions de base

Sommaire

2.1 Introduction **12**
2.2 Variables en C++ **12**
2.3 Structures de contrôle **15**

2.1 Introduction

2.2 Variables en C++

Qu'est-ce que chaque variable a ?

- un nom ;
- un type ;
- une valeur ;

2.2.1 Noms des variables

Commençons par les problèmes liés au nom d'une variable. Si vous voulez donner un nom à la variable, vous devez suivre quelques règles strictes :

- le nom de la variable doit être composé de lettres latines majuscules ou minuscules, de chiffres et du caractère _ (trait de soulignement) (*underscore*) ;
- le nom de la variable doit commencer par une lettre ;
- le caractère souligné est une lettre (étrange mais vrai) ;
- les lettres majuscules et minuscules sont traitées différemment (un peu différemment que dans le monde réel – Alice et ALICE sont les mêmes prénoms mais ce sont deux noms de variables différents, par conséquent, deux variables différentes) ;

Ces mêmes restrictions s'appliquent à tous les noms d'entités utilisés en C++. Voici quelques noms de variables corrects, mais pas toujours pratiques :

- variable
- i
- t10
- Exchange_Rate
- counter
- DaysToTheEndOfTheWorld
- TheNameOfAVariableWhichIsSoLongThatYouWillNotBeAbleToWriteItWithoutMistakes
- _

Le nom de famille peut être ridicule de votre point de vue, mais votre compilateur pense que c'est correct. Et maintenant quelques noms incorrects :

- 10t : ne commence pas par une lettre.
- Adiós_Señora : contient des caractères non autorisés.
- Exchange Rate : contient un espace.

Vous pouvez trouver plus d'informations sur le style et les conventions de dénomination C++ dans : C++ Core Guidelines [1]

2.2.2 Type de variables

Le type est un attribut qui définit de manière unique quelles valeurs peuvent être stockées dans la variable. La variable existe à la suite d'une déclaration. Une déclaration est une structure syntaxique qui lie un nom fourni par le programmeur à un type spécifique proposé par le langage C++.

1. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#nl8-use-a-consistent-naming-style

```
int counter;
float f;
```

2.2.3 Mots-clés – pourquoi sont-ils les clés ?

On les appelle mots-clés ou (plus précisément) mots-clés réservés. Ils sont réservés car vous ne pouvez pas les utiliser comme noms : ni pour vos variables, ni pour vos fonctions ou toute autre entité nommée que vous souhaitez créer. La signification du mot réservé est prédéfinie et ne peut en aucun cas être modifiée.

Ne soyez pas surpris par le fait que chaque version de C++ utilise un ensemble différent de mots-clés. Comme vous vous en doutez probablement, la liste s'allonge avec chaque nouveau standard C++. La liste, illustrée dans la figure 2.1, provient de C++17 et certains des mots-clés ne sont pas présents dans les versions antérieures.

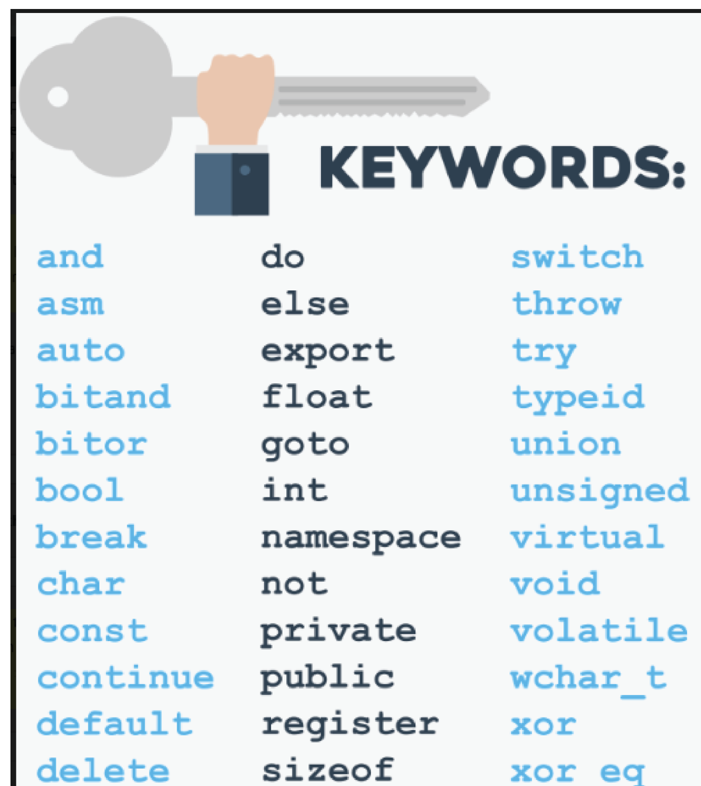

Figure 2.1 – Keywords

comme le compilateur C++ est sensible à la casse, vous pouvez modifier n'importe lequel de ces mots en changeant la casse de n'importe quelle lettre, créant ainsi un nouveau mot, qui n'est plus réservé.

2.3 Structures de contrôle

C++ fournit un ensemble conventionnel d'instructions pour exprimer la sélection et la boucle, telles que les instructions if, les instructions switch, les boucles while et les boucles for [7].

2.3.1 Traitement conditionnel

2.3.1.1 L'instruction if

Cette déclaration conditionnelle se compose des éléments suivants, strictement nécessaires, dans cet ordre et cet ordre uniquement :

- mot clé *if* ;
- Parenthèses () ;
- Une expression dont la valeur sera interprétée uniquement en termes de vrai/Faux ;
- une instruction (une seule). En cas ou le nombre d'instruction est superieure à 1, il faut les mettre entre ....

Nb. Le mot *else* et l'instruction qu'il introduit sont facultatifs, de sorte que cette instruction if présente deux formes :

Algorithme 2.1 : L'instruction if

```
if (expression)
  | instruction_1
else
  | instruction_2
```

```
if (expression)
  | instruction_1
```

Algorithme 2.2 : Exemple - Instruction If

```
// Program to check whether an integer is positive or negative
// This program considers 0 as a positive number

#include <iostream>
using namespace std;
int main() {
  int number;
  cout << "Enter an integer: ";
  cin >> number;

  if (number >= 0) {
    cout << "You entered a positive integer: " << number << endl;
  }
  else {
    cout << "You entered a negative integer: " << number << endl;
  }
  return 0;
}
```

Remarque : Un *else* se rapporte toujours au dernier if rencontré auquel un *else* n'a pas encore été attribué.

Algorithme 2.3 : Exemple 2 - Instruction If

```
if(the_weather_is_good)
   if (nice_restaurant_is_found)
      have_lunch();
   else
      eat_a_sandwich();
else
   if(tickets_are_available)
       go_to_the_theater();
   else
      go_shopping();
```

2.3.1.2 L'instruction switch

L'instruction *switch* permet, comme l'instruction if , de déclencher des traitements en fonction d'une condition (d'un test). Une instruction switch fournit un moyen pratique

de sélectionner parmi un nombre (éventuellement important) d'alternatives fixes [8]. L'instruction switch se compose d'une série d'étiquettes "case" et d'une case "default" facultative.

Algorithme 2.4 : Instruction Switch

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

int main()
{
  int weeknumber;
  //Reading week no from user
  cout<<"Enter week number(1-7): ";
  cin>>weeknumber;

  switch(weeknumber)
  {
    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;
    default: cout<<"Invalid input! Please enter week no. between
        1-7.";
  }
  return 0;
}
```

Remarque :

- Oublier "Break" est une source courante de bugs.
- Le dernier "case" d'une instruction *switch* ne nécessite pas une instruction *break*.
- Les instructions qui suivent "**default**" sont exécutées lorsqu'aucune étiquette ne correspond à la valeur de l'expression *switch*.

2.3.2 La boucle While

Une instruction de répétition spécifie qu'un programme doit répéter une action tant qu'une condition reste vraie.

```
while(conditional_expression)
    statement ;

int product = 3 ;
while ( product <= 100 )
    product = product * 3 ;
```

Algorithme 2.5 : Instruction While

```
1  #include <iostream>
2  using namespace std;
3   int main()
4   {
5    int sum = 0, val = 1;
6    // keep executing the while as long as val is less than or
         equal to 10
7    while (val <= 10) {
8      sum += val; // assigns sum + val to sum
9      ++val; // add 1 to val
10   }
11   std::cout << "Sum of 1 to 10 inclusive is "
12   << sum << std::endl;
13
14    return 0;
15 }
```

Commentaires [3]

1. Là encore, notez bien la présence de parenthèses pour délimiter la condition de poursuite. Remarquez que, par contre, la syntaxe n'impose aucun point-virgule de fin (il s'en trouvera naturellement un à la fin de l'instruction qui suit, si celle-ci est simple).
2. L'expression utilisée comme condition de poursuite est évaluée avant le premier tour de boucle. Il est donc nécessaire que sa valeur soit définie à ce moment.

3. Lorsque la condition de poursuite est une expression qui fait appel à l'opérateur séquentiel, n'oubliez pas qu'alors toutes les expressions qui la constituent seront évaluées avant le test de poursuite de la boucle. Ainsi, cette construction : while (cout « "donnez un nombre : " , cin » n, som<=100) som += n ; n'est pas équivalente à celle de l'exemple d'introduction.
4. La construction : while (expression1, expression2) ; est équivalente à : do expression1 while (expression2) ;

Fiche TD 01 - Chapitre 2. Notions de base

Algorithme 2.6 : Exercice 1 - MAX

```
1  /* finding the larger of two numbers */
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    int number1, number2;   /* the two numbers */
8
9      /* read two numbers */
10   cin >> number1;
11   cin >> number2;
12  zss&
13   /* we temporarily assume that the former number is the larger
         one */
14   /* we will check it soon */
15   int max = number1;
16
17   /* we check if the assumption was false */
18   if (number2 > max)
19   max = number2;
20
21   /* we print the result */
22   cout << "The larger number is " << max << endl;
23 }
```

Algorithme 2.7 : Exercice 2 - MAX of three numbers

```cpp
/* finding the largest of three numbers */

#include <iostream>
using namespace std;

int main()
{
  /* the three numbers */
  int number1, number2, number3;

  /* read three numbers */
  cin >> number1;
  cin >> number2;
  cin >> number3;

  /* we will save the larger number here */
  /* we temporarily assume that the former number is the larger
        one */
  /* we will check it soon */
  int max = number1;

  /* we check if the second value is the largest */
  if (number2 > max)
  max = number2;

  /* we check if the third value is the largest */
  if (number3 > max)
  max = number3;

  /* we print the result */
  cout << "The largest number is " << max << endl;
}
```

Excercice : Write C++ program to print month name using switch case

CHAPITRE 3

Classes et objets

Sommaire

3.1 Classe **22**
3.1.1 Déclaration de classe 23
3.1.2 Les méthodes 26
3.1.3 Constructeur et destructeur 27
3.1.4 Variables automatiques / statiques 32
3.1.5 Droits d'accès et encapsulation 33

L'orienté objet s'est trouvé à l'origine ces dernières années, compétition oblige, d'une explosion de technologies différentes, mais toutes intégrant à leur manière les mécanismes de base de l'OO : classes, objets, envois de messages, héritage, encapsulation, polymorphisme... Ainsi sont apparus une multitude de langages de programmation, qui intègrent ces mécanismes de base à leur manière, à partir d'une syntaxe dont les différences sont soit purement cosmétiques, soit légèrement plus subtiles [6].

3.1 Classe

En POO apparaît généralement le concept de classe, qui correspond simplement à la généralisation de la notion de type que l'on rencontre dans les langages classiques. En effet, une classe n'est rien d'autre que la description d'un ensemble d'objets ayant

une structure de données commune2 et disposant des mêmes méthodes. Les objets apparaissent alors comme des variables d'un tel type classe (on dit aussi qu'un objet est une « instance » de sa classe) [3].

Une classe est un agrégat composé de variables (également appelées champs ou propriétés) et de fonctions (parfois appelées méthodes). Les variables et les fonctions sont des composants de classe [9].

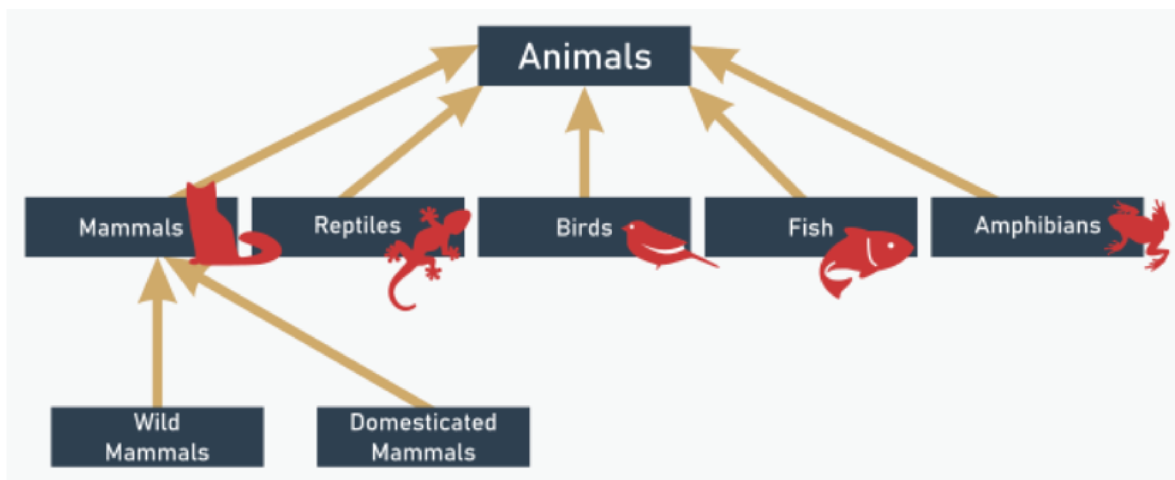

Figure 3.1 – Exemple de classe [9]

Nous pouvons dire que tous les animaux (notre classe de niveau supérieur) peuvent être divisés en cinq sous-classes :

- mammifères
- reptiles
- des oiseaux
- poisson
- amphibiens

regardons la première sous-classe pour une analyse plus approfondie. On peut identifier les sous-classes suivantes (bien sûr, ce n'est qu'une des dizaines de divisions possibles) :

- mammifères sauvages
- mammifères domestiqués

3.1.1 Déclaration de classe

Prenons l'exemple suivant, les composants set_val et get_val sont publics - ils sont accessibles à tous les utilisateurs de la classe. Le composant de valeur est privé - il n'est accessible qu'au sein de la classe.

```
class Class {
    public :
        void set_val( int value);
        int get_val();
    private :
        int value;
};
```

Déclarer l'objet de la classe, en le faisant de la manière suivante :

```
Class object;
```

Algorithme 3.1 : Déclaration de classe

```
#include <iostream>
using namespace std ;
/* ------------ Déclaration de la classe point ------------- */
class point
{ /* déclaration des membres privés */
  private :
  int x ;
  int y ;

  /* déclaration des membres publics */
  public :
  void initialise (int, int) ;
  void deplace (int, int) ;
  void affiche () ;
} ;
/* ----- Définition des fonctions membres de la classe point
      ---- */
void point::initialise (int abs, int ord)
{
  x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
  x = x + dx ; y = y + dy ;
}

void point::affiche ()
{
  cout << "Je suis en " << x << " " << y << "\n" ;
}

/* -------- Utilisation de la classe point -------- */
main()
{
  point a, b ;
  a.initialise (5, 2) ; a.affiche () ;
  a.deplace (-2, 4) ; a.affiche () ;
  b.initialise (1,-1) ; b.affiche () ;
}
```

Remarques [3]

1. Dans le jargon de la P.O.O., on dit que a et b sont des instances de la classe point,

ou encore que ce sont des objets de type point ; c'est généralement ce dernier terme que nous utiliserons.

2. Dans notre exemple, tous les membres données de point sont privés, ce qui correspond à une encapsulation complète des données. Ainsi, une tentative d'utilisation directe (ici au sein de la fonction main) du membre a : a.x = 5 conduirait à un diagnostic de compilation (bien entendu, cette instruction serait acceptée si nous avions fait de x un membre public). En général, on cherchera à respecter le principe d'encapsulation des données, quitte à prévoir des fonctions membres appropriées pour y accéder.

3. Dans notre exemple, toutes les fonctions membres étaient publiques. Il est tout à fait possible d'en rendre certaines privées. Dans ce cas, de telles fonctions ne seront plus accessibles de l'« extérieur » de la classe. Elles ne pourront être appelées que par d'autres fonctions membres.

3.1.2 Les méthodes

Une classe se compose normalement d'une ou plusieurs fonctions membres qui manipulent les attributs appartenant à un objet particulier de la classe. Les attributs sont représentés sous forme de variables dans une définition de classe. Ces variables sont appelées membres de données et sont déclarées à l'intérieur d'une définition de classe mais en dehors du corps des définitions de fonction membre de la classe. Chaque objet d'une classe conserve sa propre copie de ses attributs en mémoire. Ces attributs existent tout au long de la vie de l'objet [2].

Une méthode se compose généralement d'un type de retour, d'un nom, d'une liste de zéro ou plusieurs paramètres et d'un corps. Les paramètres sont spécifiés dans une liste séparée par des virgules entre parenthèses [8].

chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès. Il existe deux droits d'accès différents [10] :

- public : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- private : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. Par défaut, tous les éléments d'une classe sont private.

NB. Il existe d'autres droits d'accès mais ils sont un peu plus complexes. Nous les verrons plus tard.

la définition des fonctions membres se fait par une définition (presque) classique de fonction. Voici ce que pourrait être la définition de initialise :

```
void point : :initialise (int abs, int
ord)
{
    x = abs ;
    y = ord ;
}
```

Dans l'en-tête, le nom de la fonction est :

point : :initialise

Le symbole : : correspond à ce que l'on nomme l'opérateur de "résolution de portée", lequel sert à modifier la portée d'un identificateur. Ici, il signifie que l'identificateur initialise concerné est celui défini dans point. En l'absence de ce "préfixe" (point : :), nous définirions effectivement une fonction nommée initialise, mais celle-ci ne serait plus associée à point ; il s'agirait d'une fonction "ordinaire" nommée initialise, et non plus de la fonction membre initialise de la structure point [3].

C++ nous autorise à surdéfinir les fonctions ordinaires. Cette possibilité s'applique également aux fonctions membres d'une classe, y compris au constructeur (mais pas au destructeur puisqu'il ne possède pas d'arguments) [11].

3.1.3 Constructeur et destructeur

C++ nécessite un appel de constructeur pour chaque objet créé, ce qui permet de s'assurer que chaque objet est initialisé correctement avant d'être utilisé dans un programme. L'appel du constructeur se produit implicitement lors de la création de l'objet. Si une classe n'inclut pas explicitement un constructeur, le compilateur fournit un constructeur par défaut, c'est-à-dire un constructeur sans paramètres [2]. Il s'agit d'une fonction avec un nom identique à son nom de classe d'origine est appelée un constructeur. Le constructeur est destiné à construire l'objet lors de sa création, c'est-à-dire pour initialiser les valeurs de champ, allouer de la mémoire, créer d'autres objets, etc. Le constructeur peut accéder à tous les composants de l'objet comme n'importe quelle autre fonction membre de la classe mais ne doit pas être invoqué directement.

De plus, le constructeur ne doit pas être déclaré à l'aide de spécifications de type de retour, y compris les spécifications de type void.

Les constructeurs peuvent également être surchargés, en fonction des besoins et des exigences spécifiques.

Algorithme 3.2 : Exemple de classe comportant un constructeur

```
class point
{ /* déclaration des membres privés */
  int x ;
  int y ;
  public : /* déclaration des membres publics */
  point (int, int) ; // constructeur
  void deplace (int, int) ;
  void affiche () ;
} ;
```

voici comment pourrait être adapté le programme précédent pour qu'il utilise maintenant notre nouvelle classe point :

Algorithme 3.3 : classe point

```
#include <iostream>
using namespace std ;
/* ------------ Déclaration de la classe point ------------- */
class point
{ /* déclaration des membres privés */
  int x ;
  int y ;
  /* déclaration des membres publics */
  public :
  point (int, int) ; // constructeur
  void deplace (int, int) ;
  void affiche () ;
} ;
/* ----- Définition des fonctions membre de la classe point
      ---- */
point::point (int abs, int ord)
{ x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{ x = x + dx ; y = y + dy ;
}
void point::affiche ()
{ cout << "Je suis en " << x << " " << y << "\n" ;
}
/* -------- Utilisation de la classe point -------- */
main()
{ point a(5,2) ;
  a.affiche () ;
  a.deplace (-2, 4) ; a.affiche () ;
  point b(1,-1) ;
  b.affiche () ;
}
```

Remarques [3] :

1. Supposons que l'on définisse une classe point disposant d'un constructeur sans argument. Dans ce cas, la déclaration d'objets de type point continuera de s'écrire de la même manière que si la classe ne disposait pas de constructeur :

```
point a ; // déclaration utilisable avec un constructeur sans argument
```

Certes, la tentation est grande d'écrire, par analogie avec l'utilisation d'un constructeur comportant des arguments :

```
point a() ; //incorrect
```

En fait, cela représenterait la déclaration d'une fonction nommée a, ne recevant aucun argument, et renvoyant un résultat de type point. En soi, ce ne serait pas une erreur, mais il est évident que toute tentative d'utiliser le symbole a comme un objet conduirait à une erreur...

2. Nous verrons dans le prochain chapitre que, comme toute fonction (membre ou ordinaire) un constructeur peut être surdéfini ou posséder des arguments par défaut.

3. Lorsqu'une classe ne définit aucun constructeur, tout se passe en fait comme si elle disposait d'un « constructeur par défaut » ne faisant rien. On peut alors dire que lorsqu'une classe n'a pas défini de constructeur, la création des objets correspondants se fait en utilisant ce constructeur par défaut. Nous retrouverons d'ailleurs le même phénomène dans le cas du « constructeur de recopie », avec cette différence toutefois que le constructeur par défaut aura alors une action précise.

En ce qui concerne la chronologie, on peut dire que :

- le constructeur est appelé après la création de l'objet,
- le destructeur est appelé avant la destruction de l'objet.

Le destructeur, quant à lui, est une méthode appelée, automatiquement, dès la destruction d'un objet. Cette méthode ne peut recevoir d'argument car le programmeur n'est pas à l'origine de son appel [6].

Fonctionnement : Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via des "*delete*") qui a été allouée ***dynamiquement*** [10].

Algorithme 3.4 : Destructeur

```
1     #include <iostream>
2     using namespace std ;
3  class Message{
4    public:
5    //Constructeur
6    Message(){
7      cout << "Constructeur est appelé" << endl;
8    }
9    //Destructeur
10   ~Message(){
11     cout << "Destructeur est appelé" << endl;
12   }
13   //Fonction membre
14   void afficher(){
15     cout << "Bienvenu sur WayToLearnX!" << endl;
16   }
17 };
18 int main(){
19   //Objet créé
20   Message msg;
21   //Fonction appelée
22   msg.afficher();
23   return 0;
24 }
```

3.1.3.1 Règles de destructeur :

- Le nom doit commencer par le signe tilde ( ) et doit correspondre au nom de la classe.
- On peut pas avoir plus d'un destructeur dans une classe.
- Contrairement aux constructeurs qui peuvent avoir des paramètres, les destructeurs n'ont pas de paramètre. Ils n'ont aucun type de retour, comme les constructeurs.
- Lorsque vous ne spécifiez aucun destructeur dans une classe, le compilateur génère un destructeur par défaut et l'insère dans votre code.

3.1.4 Variables automatiques / statiques

Toutes les variables de votre code appartiennent à l'une des deux catégories. Elles sont :

- variables automatiques, créées et détruites, parfois de manière répétée, et automatiquement (d'où leur nom) lors de l'exécution du programme.
- variables statiques, existant en permanence pendant toute l'exécution du programme

Le langage de programmation "C++" suppose que toutes les variables sont automatiques par défaut à moins qu'elles ne soient déclarées explicitement comme statiques.

Algorithme 3.5 : Variables automatiques / statiques

```
#include <iostream>
using namespace std;
void fun()
{
  int var = 99;
  cout << "var = " << ++var << endl;
}

int main()
{
  for(int i = 0; i < 5; i++)
  fun();
}

///////////////////

#include <iostream>
using namespace std;

void fun()
{
  static int var = 99;
  cout << "var = " << ++var << endl;
}

int main()
{
  for(int i = 0; i < 5; i++)
  fun();
}
```

3.1.5 Droits d'accès et encapsulation

Les attributs sont encapsulés dans l'objet et qu'il n'est pas possible d'y accéder en dehors des méthodes elles-mêmes. Ainsi, avec notre classe *Point* précédente, nous ne pouvons pas connaître ou modifier l'abscisse d'un point en nous référant directement à l'attribut *abs* correspondant [12] . Il estpossible de doter une classe de méthodes appropriées permettant :

- d'obtenir la valeur d'un attribut donné ; nous parlerons de « méthodes d'accès » ;
- de modifier la valeur d'un ou de plusieurs attributs ; nous parlerons de « méthodes

d'altération »

> Par défaut, les attributs d'une classe sont privés et les méthodes sont publiques.

CHAPITRE 4

Héritage et polymorphisme

Sommaire

4.1 L'héritage **35**
4.1.1 L'héritage simple 37
4.1.2 L'héritage multiple 38

L'un des principaux objectifs de la programmation orientée objet est de fournir un code réutilisable.

Lorsque vous développez un nouveau projet, en particulier si le projet est important, il est agréable de pouvoir réutiliser du code éprouvé plutôt que de le réinventer. L'utilisation d'un ancien code permet de gagner du temps et, comme il a déjà été utilisé et testé, peut aider à supprimer l'introduction de bogues dans un programme.

Aussi, moins vous aurez à vous soucier des détails, mieux vous pourrez vous concentrer sur la stratégie globale du programme.

4.1 L'héritage

Voici quelques avantage de l'héritage [13] :

- Vous pouvez ajouter des fonctionnalités à une classe existante. Par exemple, étant donné une classe de tableau de base, vous pouvez ajouter des opérations arithmé-

tiques.

- Vous pouvez ajouter aux données qu'une classe représente. Par exemple, étant donné une classe de chaîne de base, vous pouvez dériver une classe qui ajoute un membre de données représentant une couleur à utiliser lors de l'affichage de la chaîne.
- Vous pouvez modifier le comportement d'une méthode de classe. Par exemple, étant donné une classe *Passenger* qui représente les services fournis à un passager aérien, vous pouvez dériver une classe *FirstClassPassenger* qui fournit un niveau de services supérieur.

Bien sûr, vous pouvez atteindre les mêmes objectifs en dupliquant le code de la classe d'origine et en le modifiant, mais le mécanisme d'héritage vous permet de continuer en fournissant simplement les nouvelles fonctionnalités. Vous n'avez même pas besoin d'accéder au code source pour dériver une classe. Ainsi, si vous achetez une bibliothèque de classes qui fournit uniquement les fichiers d'en-tête et le code compilé pour les méthodes de classe, vous pouvez toujours dériver de nouvelles classes basées sur les classes de la bibliothèque.

Vous pouvez distribuer vos propres classes à d'autres, en gardant secrètes certaines parties de votre implémentation, tout en donnant à vos clients la possibilité d'ajouter des fonctionnalités à vos classes.

L'héritage est un concept important et sa mise en œuvre de base est assez simple. Mais gérer l'héritage pour qu'il fonctionne correctement dans toutes les situations nécessite quelques ajustements.

L'héritage peut être simple ou multiple (4.2)

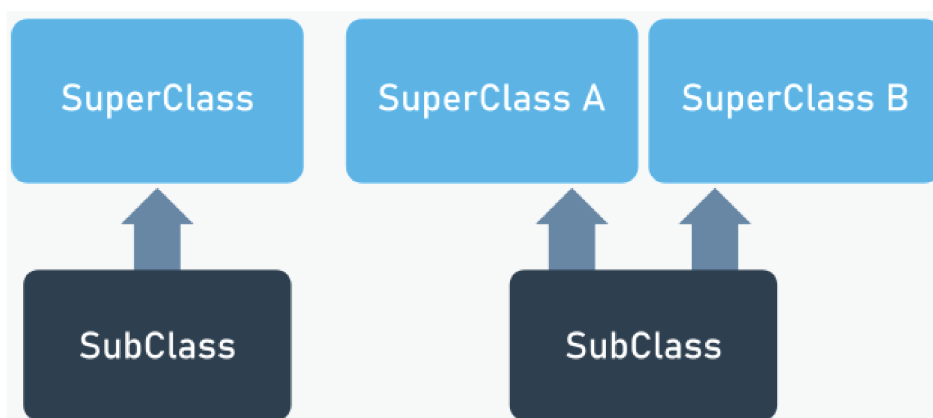

FIGURE 4.1 – Héritage [9]

L'héritage multiple est la possibilité de dériver une classe de plusieurs classes de

base directes. Une classe dérivée de manière multiple hérite des propriétés de tous ses parents. Bien que simple dans son concept, les détails de l'entrelacement de plusieurs classes de base peuvent présenter des problèmes délicats au niveau de la conception et de l'implémentation [8].

> Le mot-clé ***protected*** signifie que tout composant marqué avec celui-ci se comporte comme un composant public lorsqu'il est utilisé par l'une des sous-classes et ressemble à un composant privé pour le reste du code.

4.1.1 L'héritage simple

On parle d 'héritage simple, quand une classe fille hérite d 'une seule classe mère.

Algorithme 4.1 : Héritage Simple

```
#include <iostream>
#include "point.h" // incorporation des déclarations de point
using namespace std ;

/* --- Déclaration et définition de la classe pointcol ----- */
class pointcol : public point // pointcol dérive de point
{
  short couleur ;
  public :
     void colore (short cl)
     {
        couleur = cl ;
     }
} ;

main() {
  pointcol p ;
  p.initialise (10,20) ; p.colore (5) ;
  p.affiche () ;
  p.deplace (2,4) ;
  p.affiche () ;
}
```

4.1.2 L'héritage multiple

Certains langages offrent la possibilité dite d'héritage multiple, dans laquelle un classe peut hériter simultanément de plusieurs classes de base. L'héritage multiple reste cependant peu usité car [12] :

- peu de langages en disposent ;
- il entraîne des difficultés de conception des logiciels. Il est, en effet, plus facile de « structurer » un ensemble de classes selon un « arbre » (cas de l'héritage simple) que selon un « graphe orienté sans circuit » (cas de l'héritage multiple).

Bien entendu, il faut prévoir un mécanisme d'appel des constructeurs, ce qui ne présente pas de difficultés. En revanche, certaines situations peuvent poser problème. Considérons par exemple cette situation [12] :

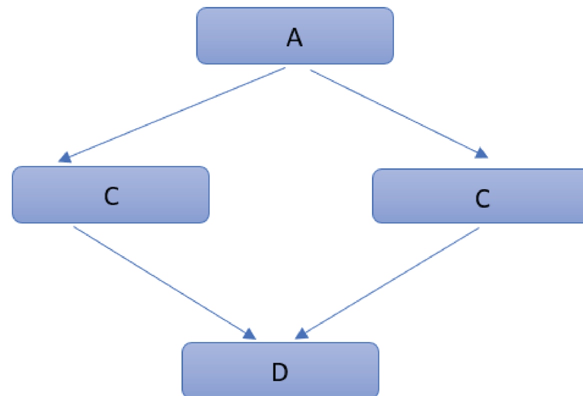

Figure 4.2 – Héritage [9]

Ici, on voit que, en quelque sorte, D hérite deux fois de A. Dans ces conditions, les méthodes et les attributs de A apparaissent deux foix dans D. En ce qui concerne les méthodes, cela est manifestement inutile, mais sans importance puisqu'elles ne sont pas dupliquées. En revanche, en ce qui concerne les attributs, il faudra disposer d'un mécanisme permettant de choisir de les dupliquer ou non. Dans le cas où on les duplique, il faudra être capable d'identifier ceux obtenus par l'intermédiaire de B de ceux obtenus par l'intermédiaire de C. Conceptuellement, on voit que les choses ne sont pas particulièrement claires.

4.1.2.1 Mise en œuvre de l'héritage multiple

Considérons une situation simple, celle où une classe, que nous nommerons pointcoul, hérite de deux autres classes nommées point et coul :

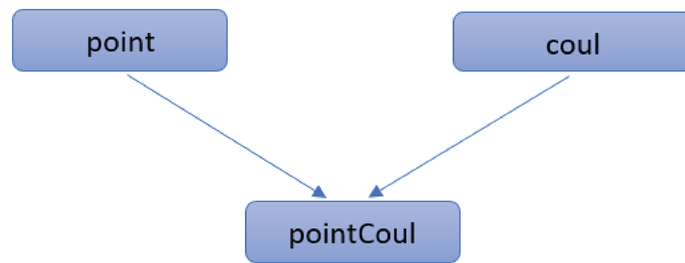

Figure 4.3 – Héritage Multiple

Algorithme 4.2 : Héritage Multiple

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
  public :
    point (int abs, int ord)
      { cout << "++ Constr. point \n" ; x=abs ; y=ord ;}
    ~point () { cout << "-- Destr. point \n" ; }
    void affiche ()
    { cout << "Coordonnees : " << x << " " << y << "\n" ; }
} ;
class coul
{   short couleur ;
  public :
      coul (int cl)
       { cout << "++ Constr. coul \n" ; couleur = cl ; }
      ~coul () { cout << "-- Destr. coul \n" ; }
      void affiche ()
      { cout << "Couleur : " << couleur << "\n" ; }
} ;
class pointcoul : public point, public coul
{
  public :
  pointcoul (int, int, int) ;
  ~pointcoul () { cout << "---- Destr. pointcoul \n" ; }
  void affiche ()
  { point::affiche () ; coul::affiche () ;  }
} ;
pointcoul::pointcoul (int abs, int ord, int cl) : point (abs,
      ord), coul (cl)
{ cout << "++++ Constr. pointcoul \n" ; }

main()
{ pointcoul p(3,9,2) ;
  cout << "------------\n" ;
  p.affiche () ; // appel de affiche de pointcoul
  cout << "------------\n" ;
  p.point::affiche () ; // on force l'appel de affiche de point
  cout << "------------\n" ;
  p.coul::affiche () ; // on force l'appel de affiche de coul
  cout << "------------\n" ;
}
```

Algorithme 4.3 : Héritage + Protected - 1

```
#include <iostream>
using namespace std;
class Pet {
  protected: string name;
  public: Pet(string n)
           { name = n;  }
        void run()
           { cout << name << ": I'm running" << endl; }
};
class Dog : public Pet {
  public:
    Dog(string n) : Pet(n) {};
    void make_sound()
        { cout << name << ": Woof! Woof!" << endl; }
};
class Cat : public Pet {
  public:
    Cat(string n) : Pet(n) {};
    void make_sound()
     { cout << name << ": Meow! Meow!" << endl; }
};
int main(){
  Pet a_pet("pet");
  Cat a_cat("Tom");
  Dog a_dog("Spike");
  a_pet.run();
  a_dog.run(); a_dog.make_sound();
  a_cat.run(); a_cat.make_sound();
}
```

Algorithme 4.4 : Héritage + Protected - 2

```
1  # include <iostream.h>
2  class Rectangle
3  {
4    public:
5    Rectangle(short l, short h);
6    void AfficheAire(void);
7    protected:
8    short largeur, hauteur;
9  };
10 Rectangle::Rectangle(short l, short h)
11 {
12   largeur = l;
13   hauteur = h;
14 }
15 void Rectangle::AfficheAire()
16 {
17   cout << "Aire = " << largeur * hauteur << "\n";
18 }
19 class Carre : public Rectangle
20 {
21   public:
22   Carre(short cote);
23 };
24 Carre::Carre(short cote) : Rectangle(cote, cote)
25 {
26 }
27 void main()
28 {
29   Carre *monCarre;
30   Rectangle *monRectangle;
31   monCarre = new Carre(10);
32   monCarre -> AfficheAire(); // affiche 100
33   monRectangle = new Rectangle(10, 15);
34   monRectangle -> AfficheAire(); // affiche 150
35 }
```

CHAPITRE 5

Les conteneurs, itérateurs et foncteurs

Sommaire

5.1 Conteneur **43**
5.2 Itérateur **44**
5.2.1 begin - end 44
5.2.2 advance 45
5.2.3 next - prev 45
5.2.4 distance 46

La bibliothèque standard fournit un ensemble de classes dites conteneurs, permettant de représenter les structures de données les plus répandues telles que les vecteurs, les listes, les ensembles ou les tableaux associatifs.

5.1 Conteneur

La bibliothèque standard du C++ propose un grand nombre de conteneurs qui sont représentés par des classes *template* et que vous pouvez utiliser pour stocker vos données, des pointeurs, des références ou des objets. Un conteneur (container) est un objet qui contient d'autres objets. Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets. Par exemple, on pourra construire une liste d'entiers, un vecteur de flottants

ou une liste de points (point étant une classe) par les déclarations suivantes :

list <int> li ; /* *liste vide d'éléments de type int* */
vector <double> ld ; /* *vecteur vide d'éléments de type double* */
list <point> lp ; /* *liste vide d'éléments de type point* */

5.2 Itérateur

Littéralement, il s'agit de quelque chose dont on se sert pour itérer. Les itérateurs présentent de nombreuses similitudes avec les pointeurs et sont utilisés pour pointer vers des éléments de conteneur de première classe. Les itérateurs contiennent des informations d'état sensibles aux conteneurs particuliers sur lesquels ils opèrent ; ainsi, les itérateurs sont implémentés de manière appropriée pour chaque type de conteneur [2].

5.2.1 begin - end

Algorithme 5.1 : Itérateur begin - end

```
1  #include<iostream>
2  #include<string>
3  #include<iterator> // classe itérateur
4  #include<vector> // classe vecteur (conteneur)
5  using namespace std;
6  int main ( ){
7    // définir un vecteur
8    vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
         "Vendredi", "Samedi","Dimanche" };
9    cout<< "Le premier element est : "<< *std::begin(jrs);
10   cout<< '\n';
11   // Afficher des éléments à l'aide de begin() et end()
12   vector<string>::iterator it;
13   cout << "Les jours par begin() et end() : "<<endl;;
14   for (it = std::begin(jrs); it < std::end(jrs); it++)
15   cout << *it << " \t";
16   return 0;
17 }
```

5.2.2 advance

Algorithme 5.2 : Itérateur - advance

```
1   int main ( ){
2     vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
          "Vendredi", "Samedi","Dimanche" };
3     vector<string>::iterator it;
4     it=std::begin(jrs);
5     std::advance(it, 3);  // faire avancer l'itérateur de 3 pas
6     cout << "La position de l'iterateur apres avoir avance : ";
7     cout << *it << " ";
8     return 0;
9   }
```

5.2.3 next - prev

Algorithme 5.3 : Itérateur - next - prev

```
1   int main ( ){
2     vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
          "Vendredi", "Samedi","Dimanche" };
3     vector<string>::iterator it;
4     it=std::begin(jrs);
5     while(it < std::end(jrs)){
6       cout<< *it << '\t';
7       it=std::next(it);
8     }
9     cout<< '\n'; cout<< "ordre inverse : "<<endl;
10    it=std::end(jrs);
11    it=std::prev(it,1);
12    while(it>std::begin(j// pointer vers le dernier
          élémentrs)){
13      cout<< *it << '\t';
14      it=std::prev(it);
15    }
16    return 0;
17  }
```

Algorithme 5.4 : Itérateur - next - prev - 2

```
1     int main ( ){
2     vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
          "Vendredi", "Samedi","Dimanche" };
3     vector<string>::iterator it;
4     it=std::begin(jrs); // ou simplmement it=jrs.begin();
5     cout<< "4eme element est : "<< *std::next(it,3);
6     cout << '\n';
7     it=std::end(jrs);// ou simplmement it=jrs.end();
8     cout<< "Avant dernier element est : "<< *std::prev(it,2);
9     return 0;
10  }
```

5.2.4 distance

Algorithme 5.5 : Itérateur - distance -1

```
1     include<iostream>
2     #include<string>
3     #include<iterator> // classe itérateur
4     #include<vector> // classe vecteur (conteneur)
5     using namespace std;
6
7     int main ( ){
8       // définir un vecteur
9       vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
            "Vendredi", "Samedi","Dimanche" };
10
11      vector<string>::iterator debut=jrs.begin();
12      vector<string>::iterator fin=jrs.end();
13
14      cout<< "Distance entre le pointeur end et debut est : "<<
            std::distance(debut,fin);
15
16      return 0;
17    }
18
```

Algorithme 5.6 : Itérateur - distance -2

```
1  #include <iostream>
2  #include <string>
3  #include <iterator> // classe itérateur
4  #include <vector> // classe vecteur (conteneur)
5  using namespace std;
6  int main ( ){
7    // définir un vecteur
8    vector<string> jrs = {"Lundi", "Mardi", "Mercredi","Jeudi",
         "Vendredi", "Samedi","Dimanche" };
9
10   vector<string>::iterator it=jrs.begin();
11
12   cout<< "déplacer l'iterateur de 3 pas : "<< * (it+3);
13
14   cout<< '\n';
15
16   it=jrs.end();
17
18   cout<< "déplacer l'iterateur de 3 pas a partir de fin: "<<
         * (it-3);
19
20   return 0;
21 }
```

CHAPITRE 6

Notions avancées

Sommaire

6.1 Gestion des exceptions **48**
6.2 Syntaxe des exceptions **49**
6.3 Les assertions **49**
6.4 Les fonctions templates **50**
6.5 Classe Template **52**

6.1 Gestion des exceptions

Au cours de l'histoire du génie logiciel, plusieurs mécanismes ont été proposées pour permettre de gérer les erreurs. Une exception est mécanisme qui permet à un morceau de code, comme une fonction, de signaler au morceau de code qui l'a appelé que quelque chose d'exceptionnel - c'est-à-dire indépendant du développeur - s'est passé [1].

6.2 Syntaxe des exceptions

Algorithme 6.1 : throw, try, catch

```
try {
  ... if (x == 0) { throw string("valeur nulle"); }
  ... if (j >= 3) { throw j; }
}
catch(const string& excep) // capture les exceptions de type
      string
{
  cerr<<"Erreur: "<<excep<<endl;
}
catch(int excep) // capture les exceptions de type int
{
  cerr<<"Avertissement: "<<excep<<endl;
}
```

6.3 Les assertions

Les exceptions c'est bien mais il y a des cas où mettre en place tous ces blocs *try / catch* est fastidieux. Il existe un autre mécanisme de détection et de gestion qui vient du langage C : les assertions.

Algorithme 6.2 : Assertions

```
#include <cassert>
using namespace std;
int main()
{
  int a(5);
  int b(5);
  assert(a == b) ; //On vérifie que a et b sont égaux
  //reste du programme
  return 0;
}
```

Lors de l'exécution, rien ne se passe. Normal, les deux variables sont égales. Par contre, si vous modifiez la valeur de b, le message suivant s'affiche alors à l'exécution :

```
monProg : main.cpp :9 : int main() : Assertion ‘a == b’ failed.
Abandon
```

6.4 Les fonctions templates

Le terme français pour *template* est modèle. Le nom est bien choisi car il décrit précisément ce que nous allons faire. Nous allons écrire un modèle de fonction et le compilateur va utiliser ce modèle dans les différents cas qui nous intéressent [10].

Algorithme 6.3 : Fonctions templates - 1

```
#include <iostream>
using namespace std;
template <typename T>
T maximum(const T& a,const T& b)
{
  if(a>b)
      return a;
  else
      return b;
}
int main()
{
  double pi(3.14);
  double e(2.71);
  cout << maximum<double>(pi,e) << endl; //Utilise la "version
          double"de la fonction
  int cave(-1);
  int dernierEtage(12);
  cout << maximum<int>(cave,dernierEtage) << endl; //Utilise la
          "version int" de la fonction

  unsigned int a(43);
  unsigned int b(87);
  cout << maximum<unsigned int>(a,b) << endl; //Utilise la
          "version unsigned int" de la fonction.
  return 0;
}
```

Algorithme 6.4 : Fonctions templates - 2

```
#include<iostream>
using namespace std;
template<typename T, typename S>
S moyenne(T tableau[], int taille)
{
  S somme(0); //La somme des éléments du tableau
  for(int i(0); i<taille; ++i)
  somme += tableau[i];
  return somme/taille;
}
int main()
{
  int tab[5];
  //Remplissage du tableau
  cout << "Moyenne : " << moyenne<int,double>(tab,5) << endl;
  return 0;
}
```

Fonction template pour trouver le plus grand nombre :

Algorithme 6.5 : Fonctions templates - max

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

template <class T>
T maxi(T n1, T n2)
{
  return (n1 > n2) ? n1 : n2;
}

// programme principal
int main()
{
  cout<< maxi(5,2)<<endl; // max de deux entiers
  cout<< maxi(3.5,2.75)<<endl; // max de deux réels
  cout<< maxi('Z','M')<<endl; // max de deux réels
  return 0;
}
```

6.5 Classe Template

un *template* permet de définir des classes et fonctions génériques et fournit ainsi un support pour la programmation. générique. les classes *templates* sont utiles lorsqu'une classe définit quelque chose qui est indépendant du type de données.

Algorithme 6.6 : Classe template - 1

```
1  #include<iostream>
2  using namespace std;
3  template<class T, class U = char>
4  class A  {
5    public:
6          T x;
7          U y;
8  };
9
10 int main()  {
11   A<int> a;  // Cela appellera A<int, char>
12   return 0;
13 }
```

Algorithme 6.7 : Classe template - 2

```
1  // création d'un patron de classe
2  template <class T> class point{
3    T x ; T y ;
4    public :
5       point (T abs=0, T ord=0)
6        { x = abs ; y = ord ;}
7       void affiche () ;
8  } ;
9  template <class T> void point<T>::affiche ()
10 { cout << "Coordonnees : " << x << " " << y << "\n" ;}
11 main (){
12   point <int> ai (3, 5) ; ai.affiche () ;
13   point <char> ac ('d', 'y') ; ac.affiche () ;
14   point <double> ad (3.5, 2.3) ; ad.affiche () ;
15 }
```

Bibliographie

[1] Claude Delannoy. *Programmer en C++ moderne : De C++11 à C++20*. Eyrolles, Paris, 2021.

[2] Paul Deitel and Harvey Deitel. *C++ How to Program*. Pearson Education , Inc, New Jersey, 10 edition, 2017.

[3] Claude Delannoy. *Apprendre le C++*. Eyrolles, Paris, 3 edition, 2008.

[4] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, USA, 4 edition, 2013.

[5] Jean Michel Doudoux. Développons en Java, version 2.2. https://www.jmdoudoux.fr/accueil_java.htm#dej/. [Online ; Page consultée le 25/05/2022].

[6] Hugues Bersini. *La programmation orientée objet Cours et exercices en UML2, Python, PHP, C#, C++ et Java*. Eyrolles, Paris, 2017.

[7] Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley, 3 edition, 2019.

[8] Barbara Moo Stanley Lippman, Josée Lajoie. *C++ Primer*. Addison-Wesley, 5 edition, 2013.

[9] Cisco Networking Academy. CPA - Programming Essentials in C++. https://www.netacad.com/. Accessed on 13/09/2022.

[10] Mathieu Nebra and Matthieu Schaller. *Programmez avec le langage C++*. Eyrolles, 2 edition, 2015.

[11] Claude Delannoy. *C++ pour les programmeurs C*. Eyrolles, Paris, 6 edition, 2007.

[12] Claude Delannoy. *S'initier a la programmation et a l'oriente objet : avec des exemples en C, C++, C#, Java, Python et PHP*. Eyrolles, Paris, 2 edition, 2016.

[13] Stephen Prata. *C++ Primer Plus*. Addison-Wesley, 6 edition, 2012.