INTRODUCTION
À LA CONCEPTION OBJET
ET À C++

Philippe Dosch

Date : Janvier 2001

UNIVERSITÉ NANCY 2
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
2 ter boulevard Charlemagne
CS 5227
54052 • NANCY cedex

Tél : 03.83.91.31.31
Fax : 03.83.28.13.33
http://www.iuta.univ-nancy2.fr

Table des matières

0. Table des matières

Chapitre 1

La conception objet

1.1 Introduction

Le but de ce premier chapitre est d'introduire la notion même de conception objet, d'un point de vue théorique. Ce chapitre n'a pas la prétention d'expliquer en détail ce type de conception, mais se propose d'en rappeler les idées maîtresses en préambule à l'étude du langage C++. Le chapitre est très largement inspiré de [Mey 91] et [Gau 96] auquel le lecteur pourra se reporter pour plus de précisions. La philosophie du langage C++, comme celle des autres langages à objets, est en effet directement inspirée de ce type de conception.

La conception par objet trouve ses fondements dans une réflexion menée autour de la vie du logiciel. D'une part, le *développement* de logiciels de plus en plus importants nécessite l'utilisation de règles permettant d'assurer une certaine qualité de réalisation. D'autre part, la *réalisation* même de logiciel est composée de plusieurs phases, dont le développement ne constitue que la première partie. Elle est suivie dans la plupart des cas d'une phase dite de *maintenance* qui consiste à corriger le logiciel et à le faire évoluer. On estime que cette dernière phase représente 70 % du coût total d'un logiciel [Mey 91], ce qui exige plus encore que la phase de développement de produire du logiciel de qualité.

La conception objet est issue des réflexions effectuées autour de cette qualité. Celle-ci peut-être atteinte à travers certains critères :

- la *correction* ou la *validité* : c'est-à-dire le fait qu'un logiciel effectue exactement les tâches pour lesquelles il a été conçu ;
- l'*extensibilité* : c'est-à-dire la capacité à intégrer facilement de nouvelles spécifications, qu'elles soient demandées par les utilisateurs ou imposées par un événement extérieur ;
- la *réutilisabilité* : les logiciels écrits doivent pouvoir être réutilisables, complètement ou en partie. Ceci impose lors de la conception une attention particulière à l'organisation du logiciel et à la définition de ses composantes ;
- la *robustesse* : c'est-à-dire l'aptitude d'un logiciel à fonctionner même dans des conditions anormales. Bien que ce critère soit plus difficile à respecter, les conditions anormales étant par définition non spécifiées lors de la conception d'un logiciel, il peut être atteint si le logiciel est capable de détecter qu'il se trouve dans une situation anormale.

Nous détaillons dans un premier temps les critères utilisés pour concevoir du logiciel de qualité, tels que la modularité [§ 1.2] ou encore la réutilisabilité [§ 1.3]. Nous définissons ensuite les bases générales de la programmation par objets [§ 1.4] avant d'introduire la terminologie associée à cette approche [§ 1.5].

1.2 La modularité

Les critères énoncés au paragraphe précédent influent sur la façon de concevoir un logiciel, et en particulier sur l'architecture logicielle. En effet, beaucoup de ces critères ne sont pas respectés lorsque l'architecture d'un logicielle est obscure, monolithique. Dans ces conditions, le moindre changement de spécification peut avoir des répercutions très importantes sur le logiciel, imposant une lourde charge de travail pour effectuer les mises à jour.

On adopte généralement une architecture assez flexible pour parer ce genre de problèmes, basée sur les *modules*. Ceux-ci sont des entités indépendantes intégrées dans une architecture pour produire un logiciel. L'ensemble des modules utilisés, ainsi que les relations qu'ils entretiennent entre eux, sont dénominés *système*. L'intérêt de ce type de conception est de concentrer les connaissances liées à une entité logique à l'intérieur d'un module qui est seul habilité à exploiter ces connaissances. L'une des conséquences immédiates est que lorsqu'une maintenance est à effectuer sur une entité logique, celle-ci ne doit concerner qu'un seul module, ce qui confine la maintenance.

1.2.1 Deux méthodes de conception de modules

Si la définition de modules est une approche communément admise, il faut également une méthode de construction de systèmes qui permette de déduire quels sont les bons modules. Il existe deux grandes familles de méthodes modulaires :

- Les méthodes *descendantes* qui procèdent par décomposition de problème. Un problème est ainsi divisé en un certain nombre de sous-problèmes, chacun de complexité moindre. Cette division est ensuite appliquée aux sous-problèmes générés, et ainsi de suite, jusqu'à ce que chacun des sous-problèmes soit trivial.
- Les méthodes *ascendantes* qui procèdent par composition de briques logicielles simples, pour obtenir des systèmes complets. C'est en particulier le cas des bibliothèques de sous-programmes disponibles avec tous les systèmes, langages, environnements...

Les deux méthodes ne sont pas automatiquement à opposer, et sont souvent utilisées en même temps lors de la conception d'un logiciel. On peut cependant noter que l'approche descendante ne favorise pas toujours la réutilisabilité des modules produits.

1.2.2 Quelques critères de qualités

En dehors de la démarche même menant aux modules, il est bon de préciser quelques critères de qualité à respecter lors de la définition des modules :

- *Compréhensibilité modulaire* : les modules doivent être clairs et organisés de manière compréhensible dans le système. Ceci implique que les modules doivent communiquer avec peu de modules, ce qui permet de les « situer » plus facilement. De même, l'enchaînement des différents modules doit être logique, et on ne doit pas avoir par exemple à utiliser plusieurs fois de suite un module pour produire une action atomique.
- *Continuité modulaire* : ce critère est respecté si une petite modification des spécifications n'entraîne qu'un nombre limité de modifications au sein d'un petit nombre de modules, sans remettre en cause les relations qui les lient.
- *Protection modulaire* : ce critère signifie que toute action lancée au niveau d'un module doit être confiné à ce module, et éventuellement à un nombre restreint de modules. Ce critère ne permet pas

de corriger les erreurs introduites, mais de confiner autant que possible les erreurs dans les modules où elles sont apparues.

Ces notions, qui sont assez intuitives, et qui découlent des réflexions menées autour de la vie du logiciel, doivent être considérées lors de la définition et de la maintenance des modules, même si elle ne sont pas accompagnées d'une méthodologie précise permettant d'y arriver. C'est à travers elles que la qualité globale d'un logiciel peut être atteinte.

1.2.3 Les principes de définition

À partir des critères exposés ci-dessus, quelques principes de conception ont été retenus pour la réalisation de modules :

- *Interface limitée* : le but n'est pas de borner les actions associées à un module, mais de se restreindre à un nombre limité d'actions bien définies, ce qui supprime une part des erreurs liées à l'utilisation de modules.
- *Communications limitées* : les communications entre modules, réalisées *via* leur interface, doivent être limitées de façon quantitative. Ceci est une conséquence du principe de modularité, qui est d'autant mieux respecté que les modules jouent leur rôle. Si les échanges sont trop importants, la notion même de module devient floue, limitant l'intérêt de cette technique.
- *Interface explicites* : les communications entre modules doivent ressortir explicitement.
- *Masquage de l'information* : toutes les informations contenues dans un module doivent être privées au module, à l'exception de celles explicitement définies publiques. Les communications autorisées sont ainsi celles explicitement définies dans l'*interface* du module, *via* les services qu'il propose.
- Les modules définis lors de la conception doivent correspondre à des unités modulaires syntaxiques liées au langage de programmation. En clair, le module spécifié ne doit pas s'adapter au langage de programmation, mais au contraire le langage de programmation doit proposer une structure permettant d'implanter le module tel qu'il a été spécifié. Par exemple, si le langage de programmation ne permet pas d'effectuer le masquage de l'information (comme le langage C), il n'est pas adéquat pour implanter les modules de manière satisfaisante selon les critères de la conception objet.

Ce genre de critères proscrit ainsi des comportements tels que l'utilisation de variables globales par exemple, qui va à l'encontre des principes énoncés. En effet, les variables globales peuvent être utilisées et modifiées par n'importe quelle composante d'un programme, ce qui complique d'autant la maintenance autour de ce genre de variables.

1.3 La réutilisabilité

La réutilisabilité n'est pas un concept nouveau en informatique et a été utilisée dès les balbutiements. En effet, les types de données à stocker sont toujours construits autour des mêmes bases (tables, listes, ensembles) et la plupart des traitements comportent des actions atomiques telles que l'insertion, la recherche, le tri, ... qui sont des problèmes résolus en informatique. Il existe une bibliographie assez abondante décrivant des solutions optimales à chacun de ces problèmes. La résolution des problèmes actuels passe par la composition des solutions de chacun de ces problèmes basiques.

Les bibliothèques (systèmes, mathématiques, etc.) sont des bons exemples de réutilisabilité et sont couramment utilisées par les programmeurs. Elles montrent cependant parfois leurs limites. En effet, les

fonctions qu'elles comportent ne sont pas capables de s'adapter aux changements de types ou d'implantation. La solution dans ce cas est de fournir une multitude de fonctions, chacune adaptée à un cas particulier, ou d'écrire une fonction prenant tous les cas en considération. Dans un cas comme dans l'autre, ce n'est que peu satisfaisant. C'est pourquoi la conception objet se propose de formaliser un peu plus cette notion de réutilisabilité et de proposer de nouvelles techniques pour l'atteindre pleinement.

1.3.1 Les principes de la réutilisabilité

Le paragraphe précédent a introduit le notion de module, en insistant sur les avantages de la conception modulaire, mais n'a pas donné de détails sur la conception même d'un module. On conviendra ici qu'un « bon » module est un module réutilisable, c'est-à-dire conçu dans l'optique d'être placé dans une bibliothèque à des fins de réutilisation. Afin de marier modularité et réutilisabilité, quelques conditions nécessaires à la conception de bons modules ont été définies :

- un module doit pouvoir manipuler plusieurs types différents. Un module de listes par exemple doit pouvoir manipuler aussi bien des entiers que des types composites ;
- de même, un module doit pouvoir s'adapter aux différentes structures de données manipulées dotées de méthodes spécifiques. Il devra ainsi par exemple pouvoir rechercher de la même manière une information contenue dans un tableau, une liste, un fichier ;
- un module doit pouvoir offrir des opérations aux clients qui l'utilisent sans que ceux-ci connaissent l'implantation de l'opération. Ceci est une conséquence directe du masquage de l'information préconisé [§ 1.2.3]. C'est une condition essentielle pour développer de grands systèmes : les clients d'un module sont ainsi protégés de tout changement de spécifications relatif à un module ;
- les opérations communes à un groupe de modules doivent pouvoir être factorisées dans un même module. Ainsi par exemple, les modules effectuant du stockage de données, tels que les listes, les tables, etc. doivent être dotés d'opérations de même nom permettant d'accéder à des éléments, d'effectuer un parcours, de tester la présence d'éléments. Ceci peut permettre entre autres de définir des algorithmes communs, tels que la recherche, quelle que soit la structure de données utilisée pour stocker les données.

1.3.2 De nouvelles techniques

L'idée, afin de faire cohabiter les principes inhérents à la modularité et à la réutilisabilité, est d'utiliser la notion de *paquetage*, introduite par des langages tels que Ada ou Modula-2. Un paquetage correspond à un regroupement, au sein d'un même module, d'une structure de données et des opérations qui lui sont propres. Ceci satisfait en particulier les critères de modularité, en isolant chaque entité d'un système, ce qui la rend plus facile à maintenir et à utiliser. En ce qui concerne les critères de réutilisabilité, il est possible d'aller encore un peu plus loin. Nous introduisons ici de nouvelles notions qui apparaissent avec les paquetages et vont permettre de franchir ce pas :

- La **surcharge** : cette notion prévoit que des opérations appartenant à des modules différents peuvent être associées au même nom. Les opérations ne sont donc plus indépendantes, elle prennent leur signification contextuellement en fonction du cadre dans lequel elles sont utilisées. Parmi ces opérations, on trouve les fonctions, mais également les opérateurs. Cela peut permettre par exemple de définir une fonction `insérer` dans chaque module de stockage, permettant d'écrire de manière uniforme : `insérer(elt, container)` quelque soit le type de `container` (liste, tableau, fichier...) ;

– La **généricité** : cette notion permet de définir des modules paramétrés par le type qu'ils manipulent. Un module générique n'est alors pas directement utilisable : c'est plutôt un *patron* de module qui sera « instancié » par les types paramètres qu'il accepte. Cette notion est très intéressante, car elle va permettre la définition de méthodes (façon de travailler) plus que de fonctions (plus formelles).

Ces définitions et ces nouveaux outils vont nous permettre de définir de nouvelles manières de concevoir des systèmes informatiques.

1.4 Principes de conception objet

1.4.1 Introduction

Après avoir énuméré les qualités souhaitables nécessaires à l'élaboration d'un système de qualité, il nous reste maintenant à déterminer les règles de construction de tels systèmes. D'un point de vue général, la construction d'un système informatique se résume par la formule :

$$\boxed{\textit{Algorithmes} + \textit{Structures de données} = \textit{Programme}}$$

Le concepteur d'un système informatique a donc deux grandes options pour l'architecture d'un système : orienter sa conception en se basant sur les données ou sur les traitements.

Dans les méthodes de conception par traitements, qui constituent l'approche traditionnelle, la base de la réflexion est effectuée autour des traitements. Le concepteur considère ainsi les tâches que doit accomplir le programme, en décomposant celui-ci en une série de tâches simples (approche descendante) ou en le construisant par composition de traitements (fonctions) disponibles (approche ascendante). On peut faire les remarques suivantes sur ce type d'approche :

– Les modules trouvés par cette approche se trouvent souvent être des modules *ad hoc*, adaptés au type de problème posé au départ et ne permettant que peu d'extensibilité du système obtenu et que peu de réutilisabilité.
– Les traitements définis ne prennent pas assez en considération les structures de données sous-jacentes, qui se retrouvent partagées entre plusieurs modules. Il devient difficile dans ces conditions de maintenir et de protéger ces structures de données.
– Il n'est pas toujours évident d'identifier les traitements, ou leur enchaînement, impliqués dans un système informatique. Sur des cas simples, cela reste aisé, mais sur des cas plus compliqués (définition d'un système d'exploitation par exemple) cela ne permet pas de déduire une architecture naturelle.
– Les traitements sont généralement beaucoup moins stables que les données. En effet, un programme une fois terminé se verra souvent étendre par de nouvelles fonctionnalités, parfois en concordance avec les objectifs premiers du programme, mais pas toujours. Par exemple, n'importe quel programme devant initialement calculer des fiches de paie ou archiver des données devra par la suite effectuer des statistiques ou être capable de répondre interactivement alors qu'il était prévu initialement pour fonctionner chaque nuit ou chaque mois. Les données en revanche sont beaucoup plus stables, et si des modifications interviennent dans la représentation des données (ajout de la CSG pour les fiches de paie, ajout d'un champ supplémentaire dans le classement des données archivées), elles ne changent pas radicalement la représentation des données.

Bien sûr, les approches basées sur les traitements ont quand même quelques avantages, dont celui d'être relativement intuitives et facilement applicables. Elles peuvent à ce titre être utilisées pour la réalisation

d'applications de taille raisonnable. Elles se révèlent cependant rapidement inadaptées lors de la réalisation de systèmes plus conséquents ou sensibles aux changements de spécification.

1.4.2 La conception par objets

Afin d'établir de façon stable et robuste l'architecture d'un système, il semble maintenant plus logique de s'organiser autour des données manipulées, les *objets*. En effet, les données étant de par leur nature plus stables que les traitements, la conception en est simplifiée. De plus il apparaît que la conception par traitement ne favorise pas l'utilisation des principes de qualité mis en évidence, tels que la modularité ou la réutilisabilité. Il reste maintenant à éclaircir les fondements mêmes de la conception par objets. Dans cette optique, voici une première définition de la conception par objets, énoncée par B. Meyer [Mey 91] :

> « *La conception par objet est la méthode qui conduit à des architectures logicielles fondées sur les OBJETS que tout système ou sous-système manipule* »

ou encore

> « *Ne commencez pas par demander ce que fait le système, demandez À QUOI il le fait !* »

La spécification d'un système va donc maintenant s'axer principalement sur la détermination des objets manipulés. Une fois cette étape réalisée, le concepteur n'aura plus qu'à réaliser les fonctions de haut-niveau qui s'appuient sur les objets et les familles d'objets définis. C'est cette approche, préconisant de considérer d'abord les objets (c'est-à-dire en première approche les données) avant l'objectif premier du système à réaliser, qui permet d'appliquer les principes de réutilisabilité et d'extensibilité. Reste maintenant à décrire comment trouver, décrire et exploiter les objets !

1.4.3 Détermination des objets

Pour la détermination même des objets, il faut simplement se rattacher aux objets physiques ou abstraits qui nous entourent. Typiquement, dans un programme de fiches de paie, le bulletin de salaire, l'employé, l'employeur, la date d'émission, etc. sont des objets. Les objets vont couvrir tous les types d'entités, des plus simples aux plus complexes, et sont à ce titre partiellement similaires aux structures utilisées dans des langages tels que C ou Pascal. Mais ils sont également dotés de nombreuses propriétés supplémentaires très intéressante, comme nous le découvrirons au fil de ce cours.

Pour représenter ou décrire de tels objets, nous allons nous intéresser non pas aux objets directement mais aux *classes* qui les représentent. Les classes vont constituer le modèle dont seront issus chacun des objets, appelés aussi *instances de classes*. Et pour décrire ces classes, nous n'allons pas faire intervenir des champs (ou attributs — des données) comme cela pouvait être le cas auparavant, nous allons les spécifier formellement en fonction des services (ou fonctions) qu'elles offrent. Ceci apporte l'avantage de se détacher de la représentation physique des données et de raisonner uniquement sur les services qu'elles doivent offrir. À titre d'exemple, la figure [FIG. 1.1] présente la description d'une classe `Pile`.

- La partie **Type** indique le type de données décrit, paramétré ici par le type `T`. Ceci permet de définir un type *générique* de pile, qui sera ensuite instancié lors de son utilisation par le type d'objets que l'on souhaite empiler : des entiers, des caractères, d'autres types...
- La partie **Fonctions** décrit les services disponibles sur les piles. Ces fonctions sont exprimées sous une forme mathématique, décrivant les paramètres nécessaires à l'utilisation de la fonction et les résultats fournis.

Type

`Pile[T]`

Fonctions

`créer:`		$\rightarrow$ `Pile[T]`
`vide:`	`Pile[T]`	$\rightarrow$ `Booléen`
`sommet:`	`Pile[T]`	$\nrightarrow$ `T`
`empiler:`	`Pile[T]` $\times$ `T`	$\rightarrow$ `Pile[T]`
`dépiler:`	`Pile[T]`	$\nrightarrow$ `Pile[T]`

Préconditions

$\forall$ `p : Pile[T]`

`dépiler(p)`	**défini ssi**	**non** `vide(p)`
`sommet(p)`	**défini ssi**	**non** `vide(p)`

FIG. 1.1 – *Description simplifiée d'une pile d'après [Gau 96].*

- La partie **Préconditions** précise les conditions d'utilisation des fonctions définies à l'aide d'une flèche barrée ($\nrightarrow$), qui ne sont pas définies pour toutes les valeurs de paramètres possibles.
- D'autres parties peuvent être ajoutées :
  - une partie **Postconditions** qui s'utilise comme la partie **Préconditions**, et qui précise les propriétés des résultats qui seront fournis par les différentes fonctions de la classe ;
  - une partie **Axiomes** qui précise les propriétés que chacun des objets de type `Pile` doit respecter à tout moment.

Cette méthode est appliquée sur chacun des types d'objets recensés nécessaires à l'élaboration du système. L'approche est conforme à la philosophie énoncée car le module est bien défini en fonction des services qu'il offre, c'est-à-dire à partir de son interface. Les principes de masquage d'information sont notamment bien respectés : l'utilisateur du module ne sait pas quelle est la représentation physique des données. Il connaît juste les services offerts par la classe. Cette démarche s'avère nécessaire pour le développement d'un système conséquent et respecte les règles de qualités établies auparavant.

Un des intérêts de ce type de démarche est par ailleurs de permettre d'améliorer les principes de réutilisabilité. Cette étape de spécification de type devance celle de l'implantation de la classe correspondant au type. Et toute classe définie pourra être potentiellement étendue par une nouvelle classe qui la *spécialisera*. Ce mécanisme, appelé *héritage*, est étudié en détails dans le chapitre 4 et permet l'utilisation de nouvelles techniques telles que la *liaison dynamique* et le *polymorphisme*.

1.4.4 Conclusion

Pour conclure, faisons le point sur les objectifs souhaités pour obtenir du logiciel de qualité et les techniques introduites pour atteindre ce but :

- modularité : cette technique permet de découper un système complet en un ensemble de modules qui sont indépendant ;
- réutilisabilité : les classes produites peuvent être regroupées en bibliothèques et être réutilisés. L'héritage permet également de réutiliser des classes en les spécialisant ;
- abstraction de données et masquage de l'information: les classes n'indiquent pas la représentation physique des données qu'elles utilisent, mais se contentent de présenter les services qu'elles offrent.

Le concept de généricité permet encore d'accroître cette abstraction, en proposant des classes qui sont paramétrées par des types de données ;

- extensibilité : les classes sont définies en terme de services. Dès lors, un changement de représentation interne de données ou une modification de celles-ci n'altère pas la façon dont les autres classes les utilisent.
- lisibilité : l'interface (documentée) permet d'avoir un mode d'emploi clair et précis de l'utilisation d'une classe, qui est d'autant plus clair que l'implantation des classes est cachée.

Pour des détails plus précis sur la conception objet, se reporter à des ouvrages détaillant ce type d'approche, tels que [Mey 91] ou [Gau 96].

1.5 Résumé et terminologie

Ce paragraphe se propose de faire le point sur toutes les notions abordées et d'introduire le vocabulaire spécifique à la conception objet.

- De part la prédominance de la maintenance dans la conception d'un logiciel, il est essentiel d'utiliser des règles permettant de construire du logiciel de qualité.
- Dans ce contexte, certains critères tels que la réutilisabilité et l'extensibilité ont été définis, répondant en partie à cette exigence.
- La modularité est une approche qui permet de parvenir à ces objectifs. Ce concept s'applique à l'implantation, mais surtout à la conception.
- La programmation est une tâche qui s'avère assez répétitive et qui nécessite de se doter de techniques favorisant la réutilisabilité. Deux de ces techniques ont été présentées : la généricité et la surcharge.
- L'architecture d'un système peut se concevoir autour des traitements ou autour des données. Si la première approche est assez intuitive et facile d'utilisation, elle ne convient que pour des systèmes de taille limitée. En revanche, la deuxième approche permet de construire des systèmes plus stables et répond mieux aux objectifs de qualités définis.
- La conception par objets se propose de décrire un système à travers les classes d'objets qui sont manipulées. Ces classes sont regroupées en familles de classes qui peuvent être implantées à travers l'héritage.
- Une classe est une structure regroupant des données, appelées *attributs* ou *variables d'instance* et des fonctions disponibles, appelées *méthodes*.
- Une instance de classe est appelée *objet*.
- La classe dont est issu un objet est appelée *type* de l'objet.
- Une classe A est appelée *client* d'une classe B si A contient un attribut de type B. La classe B est alors appelée *fournisseur* de A.

Chapitre 2

Introduction au langage C++

2.1 Historique

Le langage C++ est un langage complexe que ce seul polycopié ne peut pas exhaustivement couvrir. Le lecteur pourra se reporter à des ouvrages tels que [Str 97] et [Lip 98] (ou sa traduction française [Lip 92]) pour de plus amples informations. Un excellent ouvrage est aussi disponible gratuitement sur le Web à l'adresse `http://www.EckelObjects.com/ThinkingInCPP2e.html`. C++ peut être considéré comme une extension de C, si bien que tout programme C valide est un programme C++ valide. La différence par rapport à C est essentiellement faite au niveau des fonctionnalités objets ajoutées, et de quelques conventions différentes étudiées dans ce polycopié.

Tout comme le langage C, le langage C++ a été conçu dans les laboratoires AT&T Bell. Son concepteur, Bjarne Stroustrup, désirait étendre les fonctionnalités du langage C afin d'y intégrer les concepts de la programmation par objet. Il désirait en effet conserver les habitudes des programmeurs C, leur permettre de récupérer une bonne partie des programmes qu'ils avaient développés, tout en intégrant les avantages de la programmation objet, en s'inspirant de langages tels que *simula*. Ceci a donné naissance en 1979 au langage C+, qui intégrait quelques fonctionnalités objet, et en 1983 au langage C++, un peu plus complet, incluant notamment la généricité, la liaison dynamique... Le langage a continué ensuite à évoluer, jusqu'à l'adoption d'une norme qui est assez récente, puisqu'elle date de la fin de 1997. Ce polycopié ne s'attarde pas sur les dernières fonctionnalités offertes par la norme définitive du langage, que peu de compilateurs supportent dans leur globalité, mais plutôt sur les bases mêmes offertes par ce langage. De même, les fonctionnalités communes aux langages C et C++ ne sont pas reprises ; seules les différences sont soulignées. Le lecteur doit donc avoir une bonne connaissance du langage C pour aborder cette introduction.

2.2 Implantation de modules en C++

Tout comme en C, un module C++ est généralement implanté par deux fichiers : un fichier d'interface qui regroupe les services offerts par le module (définition de types, constantes, déclaration de fonctions) et un fichier d'implantation qui permet de définir les fonctions déclarées dans l'interface. Les extensions conventionnellement utilisées en C++ sont :

– `.hh, .H` pour les fichiers d'interface C++ (les *headers*). Il n'est pas rare cependant de voir des fichiers d'extension `.h` contenir des interfaces C++.
Remarque : la norme finale du langage C++ prévoit que les fichiers d'interface perdent leur extension.

C'est la raison pour laquelle les interfaces spécifiques au C++ sont désormais désignées sans leur extension (comme `iostream` par exemple). Il en est de même pour les *headers* déjà disponibles en C. Ceux-ci sont de plus préfixés par le caractère « `c` ». Ainsi, `stdio.h` est maintenant dénominé par `cstdio`.

– `.cc, .cpp, .C` pour les fichiers d'implantation C++. Ici par contre les extensions sont bien différenciées et sont souvent utilisées directement par les compilateurs pour déterminer le type de compilation (C, C++) à effectuer.

2.3 Types de base

Les types de bases disponibles en C, tels que `char, int, float, double, void` sont également disponibles en C++. Un type supplémentaire a été introduit pour manipuler de manière plus rigoureuse et plus explicite les booléens, c'est le type `bool` [PROG. 2.1]. Les variables de type `bool` peuvent avoir deux valeurs différentes : `true` ou `false`. Cependant, afin de rester compatible avec les variables booléennes utilisées en C, il est toujours possible de stocker et de manipuler des booléens à partir de variables de type `int`.

```
1  bool stop = false;
2
3  while (!stop) {
4    ...
5  }
```

PROG. 2.1 – *Exemple d'utilisation d'un booléen.*

2.4 Commentaires

Il existe deux moyens de définir des commentaires en C++. Le premier moyen correspond à des commentaires longs et est déjà disponible sous C. Il correspond aux délimiteurs « `/* */` » : tout ce qui se trouve entre ces deux délimiteurs est considéré comme commentaire, même s'il y a des changements de lignes. Le deuxième moyen permet de définir des commentaires courts, tenant sur une seule ligne. Il correspond au délimiteur « `//` » : tout ce qui se trouve à droite de ce délimiteur sur la *même* ligne est considéré comme commentaire [PROG. 2.2].

```
1  /*
2    Ceci est un commentaire de
3    plusieurs lignes
4  */
5
6  bool stop; // Commentaire court
```

PROG. 2.2 – *Deux types de commentaires disponibles.*

2.5 Structure d'un programme C++

La structure minimale d'un programme C++ est similaire à celle d'un programme C. Elle peut ensuite être étendue par des éléments (fonctions, instructions, structures de contrôle, etc...) abordés lors de l'étude du langage C, et par les éléments propres au C++ présentés dans ce polycopié. La fonction principale, appelée lors du début de l'exécution d'un programme, est la fonction `main` qui peut être définie de deux manières différentes [PROG. 2.3].

```
1  int main() { ... } // Programme sans paramètre
2
3  /* Programme avec paramètres :
4     argc : nombre de paramètres (y compris le nom du programme)
5     argv : tableau de paramètres, argc entrées
6  */
7
8  int main(int argc, char *argv[]) { ... }
```

PROG. 2.3 – *Structure minimale d'un programme C++.*

2.6 Les fonctions

Les fonctions C++ se déclarent et se définissent comme en C. Plusieurs caractéristiques ont cependant été ajoutées ou modifiées.

- **Vérification stricte des types.** Contrairement à C qui est un langage faiblement typé, et qui autorise ainsi l'utilisation d'une fonction avant sa définition ou sa déclaration[1], C++ est un langage fortement typé. L'utilisation d'une fonction sans une déclaration ou une définition préalable conduit à une erreur de compilation.
- **Possibilité de définir des valeurs par défaut pour certains paramètres de fonctions.** Certaines fonctions sont appelées avec des paramètres qui changent rarement. Considérons par exemple une fonction `EcranInit` qui est chargée d'initialiser un écran d'ordinateur (en mode caractères). Dans 90 % des cas, l'écran a les dimensions 24 lignes $\times$ 80 caractères et doit être initialisé dans 99 % des cas avec le caractère ' ', qui provoque l'effacement de l'écran. Plutôt que de contraindre le programmeur à énumérer des paramètres qui sont *généralement* invariants, C++ offre la possibilité de donner des valeurs par défaut à certains paramètres lors de la déclaration de la fonction [PROG. 2.4].

  Quelques remarques sur cette fonctionnalité :

  1. Une fonction peut définir des valeurs par défaut pour tous ses paramètres ou seulement pour une partie. Les paramètres acceptant des valeurs par défaut doivent se trouver *après* les paramètres sans valeur par défaut dans la liste des paramètres acceptés par une fonction.
  2. Les valeurs par défaut de chaque paramètre ne peuvent être mentionnées qu'une seule fois parmi les définitions / déclarations d'une fonction. Ainsi, par convention, ces valeurs sont généralement mentionnées dans la *déclaration* de la fonction et pas dans sa *définition* (donc dans le `.H` et pas dans le `.C`).
  3. L'ordre de déclaration des paramètres est important : dans l'exemple [PROG. 2.4] il est en effet impossible de donner une valeur à `col` sans en donner une auparavant à `lig`. D'une façon générale, il faut donc positionner parmi les paramètres ayant des valeurs par défaut en premier ceux qui ont le plus de chances d'être modifiés.

1. La fonction en question est alors supposée par défaut renvoyer un entier (type `int`).

```
 1  void EcranInit(Ecran ecran, int lig = 24, int col = 80, char fond = ' ');
 2
 3  void EcranInit(Ecran ecran, int lig, int col, char fond)
 4  {
 5    ...
 6  }
 7
 8  int main()
 9  {
10    Ecran ec;
11
12    EcranInit(ec);              // Éq. à : EcranInit(ecran, 24, 80, ' ');
13    EcranInit(ec, 26);          // Éq. à : EcranInit(ecran, 26, 80, ' ');
14    EcranInit(ec, 26, 92);      // Éq. à : EcranInit(ecran, 26, 92, ' ');
15    EcranInit(ec, 26, 92, '+');
16  }
```

PROG. 2.4 – *Exemple d'initialisations par défaut.*

– **Définition de fonctions *inline***. Les fonctions offrent beaucoup d'avantages, mais ont un inconvénient majeur lorsque le nombre d'instructions qu'elles comportent est réduit : elles sont coûteuses en temps d'exécution (exemple : fonction `int min(int a, int b);`). En effet, à chaque appel de fonction, le système met à jour un certain nombre de valeurs (essentiellement avec la pile) et effectue d'autres mises à jour lors de la terminaison de la fonction. Le coût de ces mises à jour est d'autant plus élevé (relativement) que le nombre d'instructions contenues dans la fonction est réduit, ce qui peut être critique dans des traitements exigeant de hautes performances. On peut cependant vouloir garder la structure offerte par les fonctions pour des raisons de lisibilité, d'uniformisation, de réutilisation. La solution est de définir ce genre de fonctions comme *inline* [PROG. 2.5].

```
 1  // Définition dans une interface (.H)
 2
 3  inline int min(int a, int b)
 4  {
 5    return ((a < b)? a : b);
 6  }
 7
 8  // Utilisation dans un programme
 9
10  ...
11  int m = min (i, j);  //   Remplacé par :    m = ((i < j)? i : j);
12  ...                  // (à la compilation)
```

PROG. 2.5 – *Exemple de fonction* inline.

En rajoutant le mot-clé `inline` lors de la définition de la fonction, la fonction est considérée comme *inline* : chaque appel fait à la fonction sera remplacé par le code source correspondant à la fonction [PROG. 2.5]. On dit que la fonction est *expansée* à son point d'appel. Quelques remarques :

1. Pour bénéficier de cette fonctionnalité, la fonction doit être définie (et non pas seulement déclarée) dans une interface, c'est-à-dire dans un fichier `.H`.
2. La spécification `inline` n'est qu'une recommandation faite au compilateur, qui peut décider ou non d'effectivement laisser cette fonction *inline*. En particulier, si le corps de la fonction est trop important, il y a peu de chances que la fonction reste *inline* ; il y a peu de gain aussi de toute façon, cette technique s'avérant rentable uniquement sur les fonctions comportant peu d'instructions.

2.7 Définition des variables

Contrairement au C qui impose de définir toutes les variables d'un bloc au début de ce bloc, C++ offre la possibilité de définir les différentes variables utilisées au fur et à mesure des besoins. Seule condition requise : une variable doit être soit déclarée, soit définie, avant de pouvoir être utilisée. Voir [PROG. 2.6] (et notamment les lignes 10 et 12) pour des exemples de définition de variables. Toutefois, dans un souci de lisibilité, il est conseillé de définir autant que possible les variables en début de bloc.

```
 1  int main()
 2  {
 3    int a = 5;
 4    int b = 4;
 5    int c;
 6
 7    c = a * b;
 8    b += c;
 9
10    int j = 0;
11
12    for (int i = 0; i < c; i++)
13      j+= i;
14  }
```

PROG. 2.6 – *Définition de variables en C++.*

2.8 Les références

Les références permettent de bénéficier d'un mécanisme d'*alias* sur les variables. Le langage C avait déjà introduit la notion de pointeur sur variable, qui permet d'accéder à la valeur contenue dans une variable grâce à l'adresse de cette variable. Une référence est un nouveau nom donné à une variable existante, ce qui lui confère ce rôle d'alias. Une référence est également représentée par une adresse (un pointeur), mais s'utilise comme une variable : les opérations effectuées sur la référence sont en fait réalisées sur l'objet référencé [PROG. 2.7]. Pour définir une référence : `Type &nomRéf = variableÀRéférencer`.

```
 1  int main() {
 2    int val = 10;
 3    int i;
 4    int *pval = &val;   // Pointeur sur val
 5    int &refVal = val;  // Référence sur val
 6    int &refAutre;      // Erreur !
 7
 8    *pval = 12;
 9    i = refval;         // 'i' vaut 12, valeur de 'val'
10  }
```

PROG. 2.7 – *Exemple d'utilisation des références.*

Il existe une différence entre les pointeurs et les références : les références ne peuvent être vides. C'est pour cela qu'elles sont toujours initialisées lors de leur déclaration, et qu'on préférera leur usage aux pointeurs dans les situations où on ne souhaite pas avoir de valeur inconsistante, comme cela peut être le cas avec un pointeur `NULL`. Voici ci-dessous un exemple de fonction réalisée avec des pointeurs, et la même version réalisée avec des références [PROG. 2.8].

```
 1  void swapC(int *i, int *j)
 2  {
 3    int tmp = *i;
 4    *i = *j;
 5    *j = tmp;
 6  }
 7
 8  void swapCpp(int &i, int &j)
 9  {
10    int tmp = i;
11    i = j;
12    j = tmp;
13  }
14
15  int main() {
16    int m = 1, n = 2;
17
18    swapC(&m, &n);  // D'où : m=2 et n=1
19    swapCpp(m, n);  // D'où : m=1 et n=2
20  }
```

PROG. 2.8 – *Comparaison pointeur / référence.*

2.9 Les constantes

C++ offre la possibilité de définir des entités constantes (variables, fonctions). La valeur de ces entités ne peut alors plus être modifiée lors de l'exécution du programme, ce qui suppose, comme pour les références, qu'elles doivent être initialisées à la déclaration. Des exemples typiques d'utilisation de cette fonctionnalité pour des variables sont [PROG. 2.9] :

– la représentation des bornes de boucles. C'est le cas de la variable `taille` qui ne doit pas être modifiée une fois qu'elle est fixée ;
– le passage de paramètres dans des fonctions qui ne sont pas sensées modifier la valeur de ces paramètres. C'est le cas de la fonction `contient` qui effectue une recherche et qui ne doit pas modifier la valeur des paramètres qui lui sont fournis.

Toute tentative de modification du contenu d'une variable déclarée `const` génère une erreur à la compilation : seules les opérations de lecture sont autorisées. On est donc assuré à la déclaration d'une telle variable qu'aucune modification de valeur ne sera permise. Cette restriction entraîne qu'il n'est pas possible de définir un pointeur « normal » sur une variable constante : le contenu de la variable pourrait alors être modifié *via* le pointeur. On est alors obligé de définir un pointeur sur variable constante [PROG. 2.10]. De même, si on souhaite que la valeur du pointeur ne soit pas modifiée au cours du programme, il faut définir un pointeur constant sur variable constante [PROG. 2.10].

Il est toujours possible de considérer temporairement (le temps d'une fonction par exemple) comme constante une variable non constante. L'inverse n'est pas possible : cela génère automatiquement une erreur à la compilation. L'utilisation de fonctions constantes est étudié [§ 3.6].

2.10 Allocation dynamique de mémoire

Tout comme C, C++ permet de manipuler dynamiquement la mémoire. En C, ces manipulations étaient implantées grâce aux fonctions `malloc` et `free` disponibles dans les bibliothèques standards C. En C++,

```
 1  // Un tableau 'tab' de 'taille' éléments contient-il 'elt' ?
 2
 3  bool contient(const int tab[], const int taille, const int elt)
 4  {
 5    int i;
 6
 7    for (i = 0; (i < taille) && (tab[i] != elt); i++)
 8      ;
 9
10    return (i != taille);
11  }
12
13  int main()
14  {
15    const int j;                // Erreur à la compilation !
16    int taille = 1000;
17    int tab[1000];
18    bool appartient;
19    int i;
20
21    ...
22    appartient = contient(tab, taille, i);
23    ...
24  }
```

PROG. 2.9 – *Utilisation des constantes.*

```
 1  const int taille = 1000;
 2  int compteur;
 3  int *ptaille1 = &taille;              // Erreur à la compilation !
 4  const int *ptaille2 = &taille;        // Ok
 5  const int *const ptaille3 = &taille;  // Ok
 6
 7  *ptaille2 = 1010;      // Erreur à la compilation !
 8  ptaille2 = &compteur;  // Ok. 'compteur' devient constant via ptaille2
 9  *ptaille2 = 10;        // Erreur à la compilation !
10  compteur = 10;         // Ok
11  ptaille3 = &compteur;  // Erreur à la compilation !
```

PROG. 2.10 – *Pointeurs et constantes.*

ces manipulations sont implantées *via* les opérateurs `new` et `delete` et il n'est plus nécessaire d'utiliser l'opérateur `sizeof` conjointement avec la fonction `malloc`. Le principe d'utilisation reste globalement inchangé, le programmeur doit être vigilant : à chaque allocation doit correspondre une désallocation de mémoire.

Lorsque les allocations portent sur des tableaux plutôt que des objets simples, les opérateurs `new` et `delete` sont remplacés respectivement par `new []` et `delete []`. Voir [PROG. 2.11] pour des exemples d'utilisation de l'allocation dynamique de mémoire en C++ et la comparaison par rapport aux mêmes exemples en C.

```
1  // Allocation d'une variable et d'un tableau en C
2
3  #include <stdlib.h>
4
5  main()
6  {
7    int *pi = malloc(sizeof(int));
8    int *tab = malloc(sizeof(int) * 10);
9
10   if ((pi != NULL) && (tab != NULL)) {
11     ...
12     free(pi);
13     free(tab);
14   }
15 }
16
17 // Allocation d'une variable et d'un tableau en C++
18
19 main()
20 {
21   int *pi = new int;
22   int *tab = new int[10];
23
24   if ((pi != NULL) && (tab != NULL)) {
25     ...
26     delete pi;
27     delete [] tab;
28   }
29 }
```

PROG. 2.11 – *Allocation dynamique de mémoire en C++.*

Attention : les mécanismes d'allocation dynamique de mémoire en C ou en C++ font intervenir des structures internes gérées par le langage. Ces structures sont différentes selon que l'on utilise la paire `malloc/free` ou la paire `new/delete`. Il est ainsi d'usage de n'utiliser *que* la première en C et *que* la deuxième en C++.

2.11 La surcharge

La *surcharge* est une des nouvelles techniques présentées [§ 1.3.2] permettant d'améliorer la réutilisabilité en conception objet. Elle permet d'attribuer le même nom à plusieurs opérateurs ou à plusieurs fonctions. L'ambiguïté sur la fonction appelée est alors levée après examen du contexte, c'est-à-dire du nombre et/ou du type des paramètres. Cette technique est disponible en C++, qui offre ainsi la possibilité de définir plusieurs fonctions portant le même nom, à la condition que ces fonctions aient des profils différents. Cela

permet par exemple de déclarer dans un même programme les fonctions présentées [PROG. 2.12].

```
1  int max(int a, int b);
2  int max(int *tab, int taille);
3  int max(int a, int b, int c);
```

PROG. 2.12 – *Exemples de surcharge de fonction.*

Une définition séparée de la fonction `max` est nécessaire pour chaque prototype de la fonction. Les fonctions ont cependant toutes le même but : extraire la valeur maximale en exploitant les paramètres qui leurs sont fournis. Ceci facilite donc la tâche du programmeur, qui peut associer un nom unique à une action déterminée, quels que soient les arguments fournis. La fonction correspondant aux paramètres fournis est alors exécutée. Si les types des paramètres fournis à une fonction surchargée ne concorde pas totalement avec l'une des définitions, un certain nombre de règles sont utilisées pour choisir (ou ne pas choisir) l'implantation de la fonction qui est utilisée. Ces règles ne sont pas toujours triviales, consulter un ouvrage de référence pour plus d'informations.

Il est également possible de surcharger des opérateurs. Une application est présentée [§ 2.12], et le paragraphe [§ 3.8] présente la surcharge dans le contexte particulier des classes.

2.12 Les entrées-sorties en C++

Les entrées-sorties ont été remaniées en C++, notamment pour profiter des avantages offerts par la surcharge. Comme en C, elle ne font pas partie intégrante du langage, mais sont définies dans les bibliothèques standards fournies avec le langage. Pour pouvoir les utiliser, il faut inclure l'en-tête `<iostream>`. Les entrées-sorties sont ensuite réalisées par l'intermédiaire de trois *flots* :

- `cin` : flot d'entrée standard (par défaut le clavier) ;
- `cout` : flot de sortie standard (par défaut l'écran) ;
- `cerr` : flot de sortie des messages d'erreur (par défaut l'écran).

Les opérateurs `<<` et `>>` permettent ensuite de réaliser respectivement les écritures et les lectures. Ils sont surchargés pour supporter tous les types de base disponibles en C++ [PROG. 2.13]. L'opération `endl` permet de passer à la ligne et de vider le tampon[2] correspondant au flot. Elle a donc le même rôle que le caractère `'\n'` en C, qui peut être utilisé en C++ aussi, mais ne permet alors que de passer une ligne.

Il y a plusieurs avantages à utiliser les entrées-sorties C++ par rapport aux entrées-sorties C :

- La vitesse d'exécution est plus rapide. En effet, à chaque appel de la fonction `printf`, les paramètres sont analysés (en particulier la chaîne de formatage). Avec les flots, la traduction est effectuée au moment de la compilation.
- Il n'y a plus de problème de type. Les erreurs typiques en C (associer un spécificateur `%d` avec un réel par exemple) ne peuvent plus se produire.
- Les instructions générées sont plus réduites : il n'y a plus que le code correspondant effectivement à la valeur manipulée, alors que tout le code correspondant à `printf` est inclus sinon.
- L'utilisateur a la possibilité de surcharger les opérateurs `<<` et `>>` pour manipuler les types qu'il définit, ce qui permet d'homogénéiser les entrées-sorties.

2. *Buffer.*

```
 1  #include <iostream>
 2
 3  main()
 4  {
 5    int age;
 6    int annee;
 7
 8    cout << "Entrez votre age : ";
 9    cin >> age;
10    cout << "Entrez l'année courante : ";
11    cin >> annee;
12    cout << "Vous êtes né en " << (annee - age) << endl;
13  }
```

PROG. 2.13 – *Exemples d'entrées-sorties en C++.*

– De toute façon, tout comme pour l'allocation dynamique de mémoire, les entrées-sorties font intervenir des structures internes gérées par le langage (les tampons). Les fonctions du type `printf` ou `scanf` ne sont donc plus du tout utilisées en C au profit des opérateurs manipulant les flots, sous risque d'interférences entre ces deux mécanismes.

Le format des entrées-sorties est décrit en annexe A. La gestion des fichiers en C++ est présentée brièvement [§ A.4].

2.13 Les fonctions *template*

Parmi les techniques présentées [§ 1.3.2] pour améliorer la réutilisabilité, nous avons présenté la notion de *généricité*, qui permet de paramétrer les fonctions (et les classes, voir pour cela [§ 3.9]) par un type de données. Quelle peut en être l'utilité ? L'utilité se mesure dans l'abstraction offerte par cette fonctionnalité.

Prenons un exemple concret. Supposons que l'on souhaite écrire une fonction `min` qui accepte deux paramètres et qui renvoie la plus petite des deux valeurs qui lui est fournie. On désire bénéficier de cette fonction pour certains types simples disponibles en C++ (`int, char, float, double`). La première solution pour atteindre ce but est d'utiliser la surcharge [§ 2.11] et de définir 4 fonctions `min`, une pour chacun des types considérés [PROG. 2.14].

```
 1  int min (int a, int b) {
 2    return ((a < b)? a : b);
 3  }
 4
 5  float min (float a, float b) {
 6    return ((a < b)? a : b);
 7  }
 8
 9  double min (double a, double b) {
10    return ((a < b)? a : b);
11  }
12
13  char min (char a, char b) {
14    return ((a < b)? a : b);
15  }
```

PROG. 2.14 – *Définition des fonctions `min` grâce à la surcharge.*

Lors d'un appel à la fonction `min`, le type des paramètres est alors considéré et l'implantation correspondante est finalement appelée. Ceci présente cependant quelques inconvénients :

- La définition des 4 fonctions mène à des instructions identiques, qui ne sont différenciées que par le type des variables qu'elles manipulent. On s'aperçoit ici que plus qu'une fonction, on souhaiterait exprimer une méthode, valable pour n'importe quel type manipulé : la fonction `min` est la fonction qui renvoie le plus petit des paramètres qui lui est fourni. Cet élément est déterminé grâce à l'opérateur `<` qui établit une relation d'ordre sur le type d'élément considéré.
- Si on souhaite étendre la définition de cette fonction à de nouveaux types, il faut définir une nouvelle implantation de la fonction `min` par type considéré.

Une autre solution est de définir une fonction *template*, c'est-à-dire générique [PROG. 2.15]. Cette définition définit en fait un patron de fonction, qui est *instancié* par un type de données (ici le type `T`) pour produire une fonction par type manipulé.

```
 1  template <class T>
 2  T min (T a, T b)
 3  {
 4    return ((a < b)? a : b);
 5  }
 6
 7  main()
 8  {
 9    int a = min(10, 20);          // int min(int, int)
10    float b = min(10.0, 25.0);    // float min(float, float)
11    char c = min('a', 'W');       // char min(char, char)
12  }
```

PROG. 2.15 – *Définition de la fonction `min` générique.*

Il n'est donc plus nécessaire de définir une implantation par type de données. De plus, la fonction `min` est valide avec tous les types de données dotés de l'opérateur `<`. On définit donc bien plus qu'une fonction, on définit une méthode permettant d'obtenir une certaine abstraction en s'affranchissant des problèmes de type. Quelques remarques :

- Il est possible de définir des fonctions *template* acceptant plusieurs types de données en paramètre. Chaque paramètre désignant une classe est alors précédé du mot-clé `class`, comme dans l'exemple : `template <class T, class U> ....`
- Chaque type de données paramètre d'une fonction *template* doit être utilisé dans la définition de cette fonction.
- Pour que cette fonctionnalité soit disponible, les fonctions génériques doivent être définies dans des fichiers d'interface (fichiers `.H`)[3]. Les fonctions *template* sont en effet expansées elles aussi. Ainsi, chaque appel fait à ce genre de fonctions est remplacé, à la précompilation, par le code source correspondant à la fonction.

3. Les fonctions *inline* et *template* sont ainsi les **seules** fonctions à être définies dans les interfaces. Toutes les autres sont définies dans les fichiers d'implantation (`.C`) et sont seulement déclarées dans les interfaces.

Chapitre 3

Les classes

3.1 Définition

Les classes C++ permettent d'implanter la notion de paquetage introduite [§ 1.3.2]. Une classe C++ peut se rapprocher de la notion de module définie en C, et se trouve implantée dans deux fichiers : l'interface et l'implantation. On trouve à l'intérieur d'une classe C++ des *attributs*, correspondants aux variables définies dans la classe, et des *méthodes* correspondant aux services offerts par la classe [PROG. 3.1, 3.2].

```
 1  #ifndef __ARTICLE_H_INCLUDED__
 2  #define __ARTICLE_H_INCLUDED__
 3
 4  class Article {
 5  public:
 6    // Nom de l'article
 7    const char* nom();
 8
 9    // Prix brut de l'article
10    float prixBrut();
11
12    // Quantité en stock
13    int quantite();
14
15    // Ajout d'articles dans le stock
16    void ajout(int quantite);
17
18    // Suppression d'articles dans le stock
19    void suppression(int quantite);
20
21  protected:
22    char* _nom;       // Nom de l'article
23    float _prixBrut;  // Prix brut de l'article
24    int   _quantite;  // Quantité en stock
25  };
26
27  #endif /* __ARTICLE_H_INCLUDED__ */
```

PROG. 3.1 – *Interface de la classe Article (`Article.H`).*

Pour déclarer une classe C++, il faut utiliser le mot-clé `class`. Les différents attributs et méthodes sont ensuite énumérés de la même manière que des champs sont définis à l'intérieur d'une structure C. Il y

```
 1  #include "Article.H"
 2
 3  // ****************************************************************************
 4
 5  const char*
 6  Article::nom()
 7  {
 8    return _nom;
 9  }
10
11  // ****************************************************************************
12
13  float
14  Article::prixBrut()
15  {
16    return _prixBrut;
17  }
18
19  // ****************************************************************************
20
21  int
22  Article::quantite()
23  {
24    return _quantite;
25  }
26
27  // ****************************************************************************
28
29  void
30  Article::ajout(int quantite)
31  {
32    _quantite += quantite;
33  }
34
35  // ****************************************************************************
36
37  void
38  Article::suppression(int quantite)
39  {
40    _quantite -= quantite;
41  }
```

PROG. 3.2 – *Implantation de la classe Article (`Article.C`).*

a cependant quelques différences :

- Les structures C ne permettent que de regrouper des variables, alors que les classes C++ permettent également de regrouper des fonctions (les méthodes). Ces fonctions sont également appelées *fonctions membres* pour les différencier des fonctions qui sont définies en dehors de toute classe.
- Une classe C++ peut comporter plusieurs sections, chacune d'elles étiquetée par l'un des mots-clés suivants : `public, protected, private`. Ces mots-clés permettent de définir le niveau de masquage de l'information tel qu'il a été défini [§ 1.2.3] :
  - une section `public` permet de déclarer les attributs ou les méthodes qui sont accessibles par n'importe quel client de la classe, c'est-à-dire à n'importe quel autre endroit du programme, que ce soit à l'intérieur de cette classe, d'une autre classe, ou en dehors du contexte de toute classe ;
  - une section `protected` ou `private` permet de déclarer les attributs et méthodes qui sont inaccessibles aux clients extérieurs à la classe, mais pas à la classe elle-même. Toute tentative d'accès à ces informations par un client extérieur se soldera alors par une erreur de compilation. La distinction qui existe entre les sections `protected` et `private` ne concerne pas les clients d'une classe, mais est relative à l'héritage ; elle est expliquée [§ 4.2].

  Remarque : afin de bien différencier les identificateurs (attributs, méthodes) publics de ceux qui ne le sont pas, on utilise généralement une convention d'écriture, qui correspond ici au préfixe `'_'` utilisé.
- Dans ce contexte, les attributs de classe sont conventionnellement protégés, en les déclarant dans une section autre que `public`. La conséquence de ce masquage de l'information est que les attributs d'une classe ne sont plus *directement* accessibles. On définit alors des méthodes permettant d'accéder à leur valeur. Ces méthodes sont désignées sous l'appellation *fonctions d'accès*. C'est ce qu'on appelle de l'**encapsulation** : la structure de données réelle est masquée et peut donc évoluer. Du coup, on ne montre que ce que l'on veut, et sous la forme que l'on veut.

La définition des méthodes d'une classe se fait dans le fichier d'implantation de la classe (sauf les méthodes *inline* [§ 3.3] et *template* [§ 3.9]). Le prototype[4] des méthodes est repris et est préfixé par le nom de la classe à laquelle appartiennent les méthodes et par l'*opérateur de résolution de portée* `'::'`.

3.2 Utilisation

Un exemple d'utilisation de cette classe se trouve [PROG. 3.3]. Une classe définit un type, qui peut être utilisé comme tout autre type disponible. On note au passage la notation pointée (`'.'`) utilisée pour appliquer une méthode sur un objet ou sur une référence sur objet. Cette notation se transforme en notation fléchée (`'->'`) si on dispose d'un pointeur sur objet. La ligne 11 par exemple présente l'application de la méthode `supprimer` sur l'objet `unArt` de type `Article`.

Syntaxiquement, c'est une des grandes différences entre C et C++. En C [PROG. 3.4], la conception est axée autour des traitements : les appels fonctionnels sont donc réalisés en appelant des fonctions auxquelles sont passés en paramètres les données à traiter. En C++ [PROG. 3.5], la conception est axée autour des données : les appels fonctionnels sont donc réalisés en *appliquant* des fonctions (les méthodes) sur les données (les objets). Du coup, la notation est affectée, même si, intuitivement, cela ne représente que peu de changement : il suffit de considérer qu'en C++, l'objet sur lequel on applique une méthode serait, en C, le premier paramètre de la fonction correspondant à la méthode.

4. C'est-à-dire l'identificateur correspondant à la méthode, le type de ses paramètres et le type de son résultat.

```
 1  #include "Article.H"
 2
 3  int main()
 4  {
 5    Article unArt;
 6    Article *unAutre;
 7    Article &refArt = unArt;
 8    ...
 9    cout << "Quantité disponible de " << unArt.nom()
10         << " : " << unArt.quantite() << endl;
11    unArt.suppression(3);
12    cout << "Nouvelle quantité de " << unArt.nom()
13         << " : " << unArt.quantite() << endl;
14
15    cout << "Nom de l'autre : " << unArt->nom() << endl;
16    cout << "Nom du référencé : " << refArt.nom() << endl;
17  }
```

PROG. 3.3 – *Exemple d'utilisation de la classe* `Article`.

```
 1  /*
 2    Extrait de l'interface d'un module C 'Article'
 3  */
 4
 5  typedef struct {
 6    ...
 7  } Article;
 8
 9  void ajoutArticle(Article art, int quantite);
10
11  /*
12    Exemple d'utilisation
13  */
14
15  int main()
16  {
17    Article a;
18
19    ...
20
21    ajoutArticle(a, 10);
22  }
```

PROG. 3.4 – *Appel fonctionnel réalisé en C.*

```
1  /*
2    Extrait de l'interface d'une classe C++ 'Article'
3  */
4
5  class Article {
6    ...
7  public:
8    void ajout(int quantite);
9  };
10
11
12 /*
13   Exemple d'utilisation
14 */
15
16 int main()
17 {
18   Article a;
19
20   ...
21
22   a.ajout(10);
23 }
```

PROG. 3.5 – *Appel fonctionnel réalisé en C++.*

Au niveau mémoire, chaque définition d'un nouvel article entraîne la réservation d'un espace mémoire permettant de représenter tous les attributs de la classe `Article` (le nom, le prix, la quantité). Chaque article possède donc sa *propre* copie d'attributs, ce qui est identique au comportement que le langage C propose avec les structures. La différence, encore une fois, n'est que syntaxique. Ainsi, en reprenant l'exemple d'appel à la méthode `supprimer` (ligne 11 du [PROG. 3.3]), la diminution de stock effectuée ligne 40 [PROG. 3.2] va donc affecter les attributs associés à l'objet `unArt` (lignes 5 et 11 du [PROG. 3.3]).

3.3 Les méthodes *inline*

Nous avons vu que C++ permettait de définir des fonctions *inline* [§ 2.6]. Cette fonctionnalité est également disponible avec les méthodes. Il est possible de déclarer de telles méthodes de deux façons différentes [PROG. 3.6] :

- Soit en *définissant* directement une méthode à l'intérieur d'une classe plutôt que d'effectuer une simple déclaration (méthode `nom`).
- Soit en utilisant le mot-clé `inline` lors de la déclaration, ce qui permet de définir la méthode en dehors de la partie déclarative de la classe. La définition doit cependant dans ce cas être présente dans le fichier d'interface de la classe (méthode `prixBrut`).

Ainsi, dans tous les cas, une méthode *inline* se trouve définie dans le fichier d'interface d'une classe. La raison est la même que dans le cas des fonctions *inline* : la méthode est expansée lors de l'étape de précompilation, ce qui implique que le précompilateur doit avoir à sa disposition le corps de la méthode en question.

```
 1  #ifndef __ARTICLE_H_INCLUDED__
 2  #define __ARTICLE_H_INCLUDED__
 3
 4  class Article {
 5  public:
 6    // Nom de l'article
 7    const char* nom() {return _nom}
 8
 9    // Prix brut de l'article
10    inline float prixBrut();
11
12  protected:
13    char* _nom;       // Nom de l'article
14    float _prixBrut;  // Prix brut de l'article
15  };
16
17  inline float
18  Article::prixBrut()
19  {
20    return _prixBrut;
21  }
22
23  #endif /* __ARTICLE_H_INCLUDED__ */
```

PROG. 3.6 – *Exemples de méthodes* inline.

3.4 Les constructeurs

Il est souvent nécessaire d'initialiser les objets au moment de leur création. Dans le cas de la classe `Article` en particulier, on souhaite pouvoir attribuer un nom, un prix et une quantité à tout nouvel article créé. Dans d'autres cas, ce peut être pour initialiser certains attributs spéciaux (comme des compteurs), pour effectuer une allocation mémoire, etc. Une solution pourrait être de définir pour toutes ces classes une méthode `init` qui réaliserait les initialisations souhaitées. Mais cela est problématique : pour toute création d'un nouvel objet, deux actions vont être nécessaires (déclaration + appel de la méthode `init`) alors que la création d'un objet est *a priori* une action atomique. Et que faire si un client oublie d'appliquer la méthode `init` ?

Pour résoudre ce problème, C++ possède un mécanisme d'initialisation automatique d'objets de classe. Une ou plusieurs méthodes particulières, appelées *constructeurs*, sont appliquées implicitement dès qu'un objet est défini. Ces constructeurs, généralement publics, portant le même nom que la classe à laquelle ils appartiennent [PROG. 3.7, 3.8]. Quelques commentaires généraux et relatifs à l'exemple présenté :

– Les constructeurs n'ont aucun type de retour (même pas `void`) et ne sont jamais appelés explicitement par le programmeur. C'est le compilateur qui se charge de le faire à chaque création d'objet, après avoir choisit le constructeur à utiliser en fonction des paramètres d'initialisation fournis (principe de la surcharge). Voir [PROG. 3.9] pour des exemples relatifs à chacun des constructeurs définis.

– Le premier constructeur de la classe `Article` est un constructeur particulier, dit *constructeur par défaut*, qui est utilisé lorsqu'un objet est instancié sans paramètre.

  L'action réalisée ici est d'initialiser le pointeur correspondant au nom de l'article à la valeur nulle pour indiquer que l'article est incomplet.

– Le second constructeur est également un constructeur particulier, dit *constructeur de copie*, utilisé lorsqu'un objet est créé à partir d'un autre objet de même type. On peut noter la syntaxe à utiliser

```
 1  #ifndef __ARTICLE__
 2  #define __ARTICLE__
 3
 4  #include <string.h>
 5
 6  class Article {
 7  public:
 8    // Constructeur par défaut
 9    Article();
10
11    // Constructeur de copie
12    Article(const Article& unArt);
13
14    // Constructeur normal
15    Article(const char* nom, float prixBrut, int quantite = 0);
16
17    // Destructeur
18    virtual ~Article();
19
20  protected:
21    char* _nom;       // Nom de l'article
22    float _prixBrut;  // Prix brut de l'article
23    int   _quantite;  // Quantité en stock
24  };
25
26  #endif
```

PROG. 3.7 – *Constructeurs et destructeur de la classe* `Article` *(interface).*

pour spécifier à C++ que l'on souhaite définir un constructeur de copie : `Type (const Type& unObjet)` ; cette syntaxe est obligatoire pour définir ce type de constructeur. Remarques sur ce constructeur :

  - les attributs de l'objet passé en paramètre sont recopiés dans les attributs de l'objet à créer, en prenant garde d'allouer un nouvel espace mémoire pour stocker le nom de l'article ;
  - les attributs ne sont pas recopiés si l'objet passé en paramètre est incomplet. Le nouvel objet devient alors lui-même un objet incomplet ;
  - on peut ici accéder directement aux attributs de l'objet passé en paramètre, sans avoir à utiliser les fonctions d'accès définies. Ceci n'est bien entendu possible que parce que l'objet à créer appartient à la *même* classe que l'objet passé en paramètre. Il n'est donc pas un « client ordinaire ».

– Le troisième constructeur se charge de l'initialisation d'un objet à partir de toutes les données nécessaires pour cela. Si la quantité est omise, elle prend alors la valeur `0` par défaut. Ici encore, on alloue un nouvel espace mémoire pour stocker le nom de l'article. Ce nom se trouve ainsi dans un espace propre à l'objet et ne peut donc pas être affecté par un événement extérieur, comme cela aurait pu être le cas si on avait simplement effectué une copie de pointeurs.

– Un constructeur marche en deux temps : il effectue tout d'abord l'initialisation des attributs de la classe en utilisant leur constructeur par défaut, et exécute seulement ensuite le corps du constructeur. Il n'est cependant pas toujours souhaitable d'utiliser le constructeur par défaut des attributs d'une classe. Il faut dans ce cas placer l'appel aux constructeurs des attributs entre le prototype du constructeur de l'objet et son corps, en les séparant du symbole `':'` [PROG. 3.10]. Plusieurs appels situés à ce niveau devront être séparés par des virgules.

```
 1  #include "Article.H"
 2
 3
 4  Article::Article()
 5  {
 6    _nom = 0;
 7  }
 8
 9  // *************************************************************************
10
11  Article::Article(const Article& unArt)
12  {
13    if (unArt._nom) {
14      _nom = new char[srlen(unArt._nom) + 1];
15      strcpy(_nom, unArt._nom);
16      _prixBrut = unArt._prixBrut;
17      _quantite = unArt._quantite;
18    }
19    else
20      _nom = 0;
21  }
22
23  // *************************************************************************
24
25  Article::Article(const char* nom, float prixBrut, int quantite)
26  {
27    _nom = new char[strlen(nom) + 1];
28    strcpy(_nom, nom);
29    _prixBrut = prixBrut;
30    _quantite = quantite;
31  }
32
33  // *************************************************************************
34
35  Article::~Article()
36  {
37    if (_nom)
38      delete [] _nom;
39  }
```

PROG. 3.8 – *Constructeurs et destructeur de la classe* `Article` *(implantation).*

```
 1  #include "Article.H"
 2
 3  int main()
 4  {
 5    Article a;                    // Constructeur par défaut appelé
 6    Article b("livre", 100, 10);  // 3ème constructeur appelé
 7    Article c(b);                 // Constructeur par copie appelé
 8    Article *d;
 9
10    d = new Article(a);           // Constructeur par copie appelé
11    ...
12    delete d;                     // Destructeur de 'd' explicitement appelé
13  }                               // Destructeurs de 'a', 'b' et 'c'
14                                  // implicitement appelés
```

PROG. 3.9 – *Appels des constructeurs et du destructeur de la classe* `Article`.

```
 1  #include "Article.H"
 2
 3  class LotArticles {
 4  public:
 5    LotArticle(const char* nom, float prixBrut, int quantite, int nbLot)
 6      : _art(nom, prixBrut, quantite), _nbLot(nbLot)
 7    {
 8    }
 9
10  protected:
11    Article _art;    // Article du lot
12    int     _nbLot;  // Nombre d'articles par lot
13  };
```

PROG. 3.10 – *Exemple d'initialisation sans appel au constructeur par défaut.*

3.5 Le destructeur

Le mécanisme automatique d'initialisation proposé par les constructeurs est très pratique. Mais d'autres problèmes surviennent : à quel moment libérer l'espace mémoire alloué pour stocker le nom d'un article en particulier ? C++ apporte également une solution à ce problème grâce à un second mécanisme automatique, permettant d'appliquer une méthode particulière, appelée *destructeur*, au moment de la destruction de l'objet. Quelques remarques sur cette méthode :

- il n'y a qu'un seul destructeur possible par classe ;
- l'identificateur désignant le destructeur est composé du nom de la classe préfixé par le symbole `'~'` (ce qui donne ainsi le nom `~Article` pour désigner le destructeur de la classe `Article`) ;
- les destructeurs n'acceptent aucun paramètre et ne renvoient aucun résultat ;
- dans l'exemple présenté [PROG. 3.8], le destructeur de la classe n'a qu'un seule rôle : libérer l'espace mémoire alloué dynamiquement pour stocker le nom de l'article ;
- il est préférable de préfixer les destructeurs du mot-clé `virtual`. Une explication de cette convention est fournie [§ 4.5] ;
- les destructeurs, comme les constructeurs, ne sont **jamais** explicitement appelés dans un programme. Ils sont utilisés automatiquement et implicitement par le compilateur au moment de la création et de la destruction des objets.

3.6 Méthodes constantes

Après avoir étudié les variables constantes [§ 2.9], nous étudions dans ce paragraphe les méthodes constantes. Nous avons vu à propos des variables constantes que toute tentative de modification qui leur est relative se solde par une erreur de compilation. Dans le contexte particulier des instances de classe, cela n'est pas toujours approprié car les objets sont rarement directement modifiés, vu que leurs attributs sont généralement cachés. En fait, ce sont généralement des méthodes publiques qui sont invoquées pour réaliser ces modifications. Et c'est pour assurer la constance des objets que les *méthodes constantes* ont été introduites. Le concepteur d'une classe peut ainsi indiquer les méthodes qui peuvent être utilisées en toute sécurité, sans risquer de modifier les objets de cette classe.

Pour rendre une méthode constante, il suffit de placer le spécificateur `const` après le prototype de la méthode. Les méthodes constantes sont ensuite les seules à pouvoir être appliquées sur des objets constants.

Elles ne peuvent ainsi pas comporter d'instruction permettant de changer la valeur de l'objet sous peine d'erreur de compilation. Un exemple de définition et d'utilisation est présenté [PROG. 3.11].

```
 1  #include <iostream>
 2
 3  class Article {
 4  public:
 5    // Nom de l'article
 6    const char* nom() {return _nom;}
 7
 8    // Nom de l'article (const)
 9    const char* nomCste() const {return _nom;}
10
11    // Change le nom de l'article (const)
12    // -> ERREUR À LA COMPILATION
13    void changeNom(char *nom) const {_nom = nom};
14
15  protected:
16    char* _nom;       // Nom de l'article
17  };
18
19  void testAffichage(Article a, const Article b)
20  {
21    cout << a.nom() << endl;       // Ok
22    cout << a.nomCste() << endl;   // Ok
23
24    cout << b.nom() << endl;       // Erreur compilation
25    cout << b.nomCste() << endl;   // Ok
26  }
```

PROG. 3.11 – *Définition et utilisation des méthodes constantes.*

3.7 Le mot-clé `this`

On a parfois besoin de désigner à l'intérieur d'une fonction membre l'objet qui est manipulé par la méthode. Comment le désigner cependant alors qu'il n'existe aucune variable le représentant dans la fonction membre ? Les fonctions membres travaillent en effet directement sur les attributs de classes : ceux qui sont atteints correspondent alors à ceux de l'objet courant. C++ apporte une solution à ce problème en introduisant le mot-clé `this` qui permet à tout moment dans une fonction membre d'accéder à un pointeur sur l'objet manipulé. Un exemple d'application se trouve [PROG. 3.12].

3.8 La surcharge

Il est possible d'utiliser la surcharge à l'intérieur des classes, de la même manière que cela est possible sur les fonctions classiques [§ 2.11]. Les constructeurs de classe en sont d'ailleurs un très bon exemple ! La surcharge est ainsi beaucoup utilisée dans le contexte des classes, que ce soit pour surcharger des méthodes ou des opérateurs. Imaginons en effet que nous souhaitions disposer de l'opérateur d'affectation `'='` pour nous permettre de réaliser une affectation d'articles. Un opérateur d'affectation par défaut existe, mais il n'effectue qu'une copie champ-à-champ des attributs de la classe. Ce comportement est ennuyeux dans le cas de la classe `Article` car il ne permet pas de dupliquer le nom des articles, mais juste de réaliser une copie des pointeurs.

```
 1  #include <iostream>
 2  #include "Article.H"
 3
 4  // Fonction présente pour les besoins du test
 5
 6  void testAffichage(Article *unArt)
 7  {
 8    cout << "Article : " << unArt->nom() << endl;
 9  }
10
11  // Méthode de la classe 'Article'
12
13  void
14  Article::methodeQuelconque()
15  {
16    // Comment appeler la fonction 'testAffichage' ?
17    // Avec le mot-clé 'this' !
18
19    testAffichage(this);
20  }
```

PROG. 3.12 – *Utilisation du mot-clé* `this`.

Pour que l'opérateur d'affectation effectue les actions que nous attendons de lui, nous pouvons le surcharger [PROG. 3.13]. Un grand nombre d'opérateurs peuvent être surchargés [TAB. 3.14], même s'il n'est

```
 1  Article& Article::operator=(const Article &unArt)
 2  {
 3    // Si l'article courant a un nom, on le libère
 4
 5    if (_nom)
 6      delete [] _nom;
 7
 8    // On recopie l'article passé en paramètre
 9
10    if (unArt._nom) {
11      _nom = new char[srlen(unArt._nom) + 1];
12      strcpy(_nom, unArt._nom);
13      _prixBrut = unArt._prixBrut;
14      _quantite = unArt._quantite;
15    }
16    else
17      _nom = 0;
18  }
```

PROG. 3.13 – *Surcharge des opérateurs de la classe* `Article`.

pas toujours souhaitable d'abuser de cette pratique, qui peut entraver la clarté du programme et être à l'origine d'erreurs difficiles à localiser. En effet, ces opérateurs sont dans la plupart des cas définis par défaut. Un programmeur qui n'a pas regardé attentivement la documentation d'une classe peut s'attendre à un comportement donné en considérant un opérateur appliqué à un objet, alors que le comportement peut être tout autre si l'opérateur est surchargé[5]. Ce genre de désagrément n'arrive pas avec les méthodes, aucune méthode n'étant définie par défaut pour une classe donnée. Cependant, il est courant de surcharger des opé-

5. On peut par exemple s'amuser à surcharger `new` pour qu'il effectue ce qui effectué par `delete`, et *vice-versa* ! Mais même sans atteindre ce type de situations contre nature, la surcharge peut réserver quelques surprises... Cette fonctionnalité a d'ailleurs été supprimée dans le langage Java, plus récent et souhaitant éliminer les causes d'erreurs fréquentes en C++.

`+`	`-`	`*`	`/`	`%`
`~`	`!`	`=`	`<`	`>`
`<=`	`>=`	`++`	`-`	`==`
`!=`	`&&`	`||`	`+=`	`-=`
`new`	`delete`	`()`	`->`	`[]`

TAB. 3.14 – *Opérateurs pouvant être surchargés (liste non exhaustive).*

rateurs tels que l'affectation (=), l'inférieur (<), la différence (!=) ou l'égalité (==). C'est en particulier le cas lors de l'utilisation de la bibliothèque STL livrée en standard avec C++ et détaillée chapitre 5.

3.9 Les classes *template*

Il est possible, comme pour les fonctions [§ 2.13], de définir des classes *template*, c'est-à-dire paramétrées par un type de données. Cette technique évite ainsi de définir plusieurs classes similaires pour décrire un même concept appliqué à plusieurs type de données différents. Elle est largement utilisée pour définir tous les types de containers (comme les listes, les tables, les piles, etc.), mais aussi des algorithmes génériques par exemple. La bibliothèque STL (chapitre 5) en particulier propose une implantation d'un bon nombre de types abstraits et d'algorithmes génériques.

La syntaxe permettant de définir une classe *template* est similaire à celle qui permet de définir des fonctions *template*. Voir [PROG. 3.15, 3.16] pour un exemple de classe *template*, portant sur des points dont la précision de représentation (à partir d'entiers, de réels, etc.) est le type paramètre de la classe. Quelques remarques :

- Comme dans le cas des fonctions *template*, tout le code source correspondant à des classes *template* (y compris la définition de leurs méthodes) doit se trouver dans l'interface de la classe correspondante.
- Une classe *template* permet de définir des attributs, des paramètres ou des valeurs de retour de méthodes *template*. De façon réciproque, pour pouvoir définir des entités *template* à l'intérieur d'une classe, la classe doit elle-même être *template*.
- Attention à la syntaxe des méthodes *template* définies en dehors du corps de la classe. La définition se fait : `template <class T> typeRetour nomClasse<T>::nomMéthode(params)`.

3.10 Les amis

Il arrive parfois que l'on souhaite accorder des accès plus fins que les accès `public`, `protected` ou `private` proposés par défaut. En particulier, il peut être intéressant d'accorder des accès aux attributs ou aux méthodes à certaines classes clientes, tout en protégeant ce même accès vis-à-vis des autres classes. Un des exemples les plus fréquents est la surcharge de l'opérateur << permettant d'afficher un objet. En effet, afin de concaténer les opérateurs successivement (p.ex. `cout << a << b << c << endl;`), le profil de la fonction est défini comme :

```
ostream& operator<<(ostream&, Classe&);
```

où `Classe` est la classe qui est manipulée. Dans ce contexte, l'opérateur << n'est alors pas défini comme membre de chaque classe, mais comme opérateur à portée globale, devant être surchargé pour chaque type à manipuler. Problème : comment autoriser cet opérateur à accéder à des attributs de classe, tout en

```
 1  template <class T>
 2  class Point {
 3  public :
 4    // Constructeur par défaut
 5    Point() : _x(0), _y(0) {}
 6
 7    // Constructeur
 8    Point(T x, T y) : _x(x), _y(y) {}
 9
10    // Accès à x
11    const T x() const {return _x;}
12
13    // Accès à y
14    const T y() const {return _y;}
15
16    // Translation
17    void translation(T x, T y);
18
19  protected:
20    T _x;      // Abcisse
21    T _y;      // Ordonnée
22  };
23
24  template <class T> void
25  Point<T>::translation(T x, T y)
26  {
27    _x += x;
28    _y += y;
29  }
```

PROG. 3.15 – *Définition d'une classe* template.

```
1  #include "Point.H"
2
3  int main()
4  {
5    Point<int> pointEntier(2, 3);
6    Point<float> pointReel(3.14, 2.27);
7
8    pointReel.translation(pointEntier.x(), pointEntier.y());
9  }
```

PROG. 3.16 – *Utilisation d'une classe* template.

protégeant ce même accès vis-à-vis des autres classes ? C++ introduit pour cela la notion d'*amitié*, dont une définition intuitive est : « *Un ami est quelqu'un qui peut toucher vos parties privées.* » En clair, un ami peut accéder aux attributs et aux méthodes protégées ou privées d'une classe. Cela permet de surcharger `<<` comme présenté [PROG. 3.17].

```
1  #include <iostream>
2
3  class Article {
4  public:
5    // Un exemple de fonction amie
6    friend ostream& operator<<(ostream&, Article&);
7
8    // Un exemple de classe amie
9    friend class uneClasse;
10
11   // Un exemple de méthode amie
12   friend void autreClasse::methode();
13
14 protected:
15   char* _nom;       // Nom de l'article
16   float _prixBrut;  // Prix brut de l'article
17   int   _quantite;  // Quantité en stock
18 };
19
20 ostream& operator<<(ostream& os, Article& art)
21 {
22   os << "Article " << art._nom
23      << ", prix : " << art._prixBrut
24      << ", quantité : " << art._quantite
25      << endl;
26
27   return os;
28 }
```

PROG. 3.17 – *Amitié et Surcharge de* `<<` *pour la classe* `Article`.

C++ permet de définir de cette manière des classes et des fonctions amies, suivant les accès que l'on souhaite autoriser. La notion d'amitié constitue une infraction aux règles d'encapsulation, mais s'avère utile dans certaines situations ; elle est généralement utilisée avec parcimonie.

3.11 Membres statiques

Jusqu'à présent, les attributs et méthodes tels que nous les avons définis ont toujours été liés aux objets. En effet, dans le cas d'un attribut, une valeur différente de cet attribut peut être associée à chaque objet. De même, les méthodes sont toujours invoquées par rapport à un objet, et ne peuvent pas être appelées en dehors de tout contexte.

Dans certains cas, ce comportement n'est cependant pas toujours souhaité. On peut en effet avoir envie de rattacher un attribut non pas à chaque instance d'une classe, mais à la classe elle-même. De la même manière, il peut parfois être utile de pouvoir invoquer une méthode dans le contexte d'une classe, et non dans celui d'un objet. Cette fonctionnalité est possible en C++ grâce aux *membres statiques*, appelés également *variables* et *méthodes de classe* (par opposition aux variables et méthodes *d'instance* que nous avons étudiés jusqu'à présent).

Par exemple, si nous souhaitons attribuer automatiquement un numéro d'article à chaque nouvel article que nous créons, il est possible de définir un compteur statique au niveau de la classe article. Ce compteur n'aura plus alors qu'à être incrémenté à chaque nouvel création d'article. La modification s'opère donc au niveau de l'interface de la classe (pour la définition de ce nouvel attribut), ainsi que dans le ou les constructeurs de cette classe [PROG. 3.18, 3.19]. Dans cet exemple, l'attribut `_prochainNumero` est

```
 1  class Article {
 2  public:
 3    // Constructeur.
 4    Article();
 5
 6    // Accès au numéro de l'article.
 7    int numero() const {return _numero;}
 8
 9  protected:
10    int _numero;                 // Numéro de l'article
11    static int _prochainNumero;  // Variable de classe
12  };
```

PROG. 3.18 – *Utilisation d'un membre statique pour la création d'un compteur (extrait de l'interface).*

```
1  #include "Article.H"
2
3  int Article::_prochainNumero = 0;
4
5  Article::Article()
6  {
7    _numero = _prochainNumero++;
8  }
```

PROG. 3.19 – *Utilisation d'un membre statique pour la création d'un compteur (extrait de l'implémentation).*

donc **commun** à toutes les instances de la classe `Article`. En effet, étant donné que cette variable est une variable de classe, elle est partagée par **tous** les objets issus de la classe `Article`. Attention à la syntaxe de l'initialisation de telles variables : l'initialisation est impérativement unique, et se trouve dans le fichier d'implémentation de la classe, en rappelant tous les éléments du contexte (type de la variable et classe d'appartenance).

Cette variable possède des propriétés supplémentaires : il est possible (mais pas nécessaire) d'y accéder sans passer par un objet. En effet, cette variable existant indépendamment des objets qui sont créés, il est également possible d'y accéder ou de la modifier, même si aucune instance de la classe `Article` n'est disponible. Pour cela, il suffit de définir des méthodes de classes, c'est-à-dire des méthodes statiques, qui peuvent être appelées en dehors du contexte d'un objet ; il est cependant alors nécessaire de préciser le contexte de la classe [PROG. 3.20, 3.21].

Attention : une méthode statique pouvant être appelée en dehors du contexte d'un objet, il est impossible d'y intégrer des accès aux membres (attributs ou méthodes) non-statiques de la classe. Il est également impossible, pour les mêmes raisons, de faire référence au pointeur `this` à l'intérieur d'une méthode statique.

La définition de membres statiques est très utile pour définir et utiliser des variables dont le comportement est proche de variables globales. Il existe cependant des avantages à définir des membres statiques

```
 1  class Article {
 2  public:
 3    // Constructeur.
 4    Article();
 5
 6    // Accès au numéro de l'article.
 7    int numero() const {return _numero;}
 8
 9    // Accès au prochain numéro d'article.
10    static int prochainNumero() const {return _prochainNumero;}
11
12  protected:
13    int _numero;                 // Numéro de l'article
14    static int _prochainNumero;  // Variable de classe
15  };
```

PROG. 3.20 – *Définition d'une méthode statique (extrait de l'interface).*

```
1  int main()
2  {
3    int pn;
4
5    pn = Article::prochainNumero();
6  }
```

PROG. 3.21 – *Utilisation d'une méthode statique.*

plutôt que de vraies variables globales :

- Les règles de masquage de l'information sont respectées. En particulier, une variable statique peut être publique ou privées, alors qu'une variable globale ne peut être que publique.
- Une variable membre statique est définie à l'intérieur d'une classe, et ne peut donc pas interférer avec les membres définis dans les autres classes (pas de conflit de noms).

Chapitre 4

L'héritage

4.1 Introduction

L'héritage, qui a été évoqué [§ 1.4.3], est une technique de plus permettant de servir la réutilisabilité. Son objectif est de permettre la définition aisée de sous-types, correspondant à une spécialisation (à l'extension) de types existants. Imaginons en effet que nous souhaitions implanter une classe `Boisson`. Les boissons sont des *sortes* d'articles, et doivent comporter à ce titre les même attributs que tout article (`_nom, _prixBrut, _quantite`), ainsi qu'un autre qui est spécifique aux boissons : `_volume`. L'héritage permet de réutiliser la définition de la classe `Article` pour définir la classe `boisson`.

4.2 La dérivation

La dérivation de classe est la technique qui consiste à faire hériter une classe d'une autre classe en C++. Au niveau terminologique, on dit ainsi qu'une classe `B` est *dérivée* d'une classe `A` si elle en hérite. La classe `A` est alors appelée *superclasse* ou *classe de base* de la classe `B`. Dans l'exemple évoqué au paragraphe précédent, la classe `Boisson` est une classe dérivée de la classe `Article` puisqu'elle la spécialise. Pour spécifier en C++ qu'une classe `B` hérite d'une classe `A`, on déclare la classe `B` de la manière suivante :

```
class B : <mode_dérivation> class A { ... };
```

où le mode de dérivation permet de fixer le statut des membres (attributs, fonctions) de la classe `B` en fonction du statut des membres de la classe `A` (voir le tableau [TAB. 4.1] pour un récapitulatif des différents

		Statut des membres de base		
		public	**protected**	**private**
Mode de dérivation	**public**	public	protected	*inaccessible*
	protected	protected	protected	*inaccessible*
	private	private	private	*inaccessible*

TAB. 4.1 – *Statut des membres de la classe dérivée en fonction du statut des membres de la classe de base et du mode de dérivation.*

cas possibles). Le mode de dérivation par défaut est le mode `private`, mais le mode de dérivation le plus courant est le mode `public` : c'est celui qui permet de donner les même statuts aux membres dérivés que ceux des membres de base. Une définition de la classe `Boisson` dans ce contexte est donnée [PROG. 4.2].

```
 1  #include <string.h>
 2
 3  //
 4  // Classe Article
 5  //
 6
 7  class Article {
 8  public:
 9    // Constructeur
10    Article(const char* nom, float prixBrut,
11            int quantite = 0, float tva = 1.206)
12      : _prixBrut(prixBrut), _quantite(quantite), _tva(tva)
13    {
14      _nom = new char[strlen(nom) + 1];
15      strcpy(_nom, nom);
16    }
17
18    // Prix brut de l'article
19    float prixBrut() const;
20
21    // Prix net de l'article
22    float prixNet() const {return _prixBrut * _tva;}
23
24  protected:
25    char* _nom;       // Nom de l'article
26    float _prixBrut;  // Prix brut de l'article
27    int   _quantite;  // Quantité en stock
28    float _tva;       // TVA appliquée à l'article
29  };
30
31  //
32  // Classe Boisson
33  //
34
35  class Boisson : public Article {
36  public:
37    // Constructeur
38    Boisson(const char* nom, float prixBrut, int volume, int quantite = 0)
39      : Article(nom, prixBrut, quantite), _volume(volume) {}
40
41    // Accès au volume
42    int volume() const {return _volume;}
43
44  protected:
45    int _volume;     // Volume de la boisson (cl)
46  };
```

PROG. 4.2 – *Rappel de la classe* `Article` *et définition de la classe* `Boisson`*. Note : un attribut correspondant à la TVA a également été ajouté au niveau de la classe* `Article`*.*

La classe `Boisson`, qui est avant tout un article, hérite des attributs et des méthodes définis dans la classe `Article`, en dehors des constructeurs, du destructeur, du constructeur de copie et de l'opérateur d'affectation. Le statut de ces membres hérités est fixé par le mode de dérivation. La classe `Boisson` peut aussi définir de nouveaux membres, comme l'attribut `_volume` et la nouvelle méthode d'accès à cet attribut.

Lors de la création d'une boisson (c'est-à-dire d'un objet de ce type), tous les constructeurs de la hiérarchie d'héritage sont automatiquement appelés, du plus général (ici `Article`) au plus particulier (ici `Boisson`). Le type d'appel fait au constructeur de la classe `Article` est implanté dans le constructeur de la classe `Boisson`, en utilisant la même syntaxe que pour initialiser les attributs de classe. Si cet appel n'est pas spécifié, c'est le constructeur par défaut de la classe `Article` qui est invoqué.

Lors de la destruction de cette même boisson, les différentes destructeurs en présence dans la hiérarchie d'héritage sont successivement appelés, dans l'ordre *inverse* de celui des constructeurs, c'est-à-dire du plus particulier (`Boisson`) au plus général (`Article`).

Note : C++ permet également d'utiliser l'héritage multiple. Cette partie n'est pas abordé dans ce poly-copié.

4.3 Redéfinition de méthodes

L'héritage permet de bénéficier d'autres fonctionnalités, telles que la redéfinition de méthodes. Supposons, pour illustrer cette technique, que nous souhaitions implanter une classe `BoissonAlcoolisee`. Cette classe hérite de la classe `Boisson` et introduit deux nouveaux attributs :

- `_degre` qui correspond au degré d'alcool contenu dans la boisson,
- `_accise` qui contient le montant des accises[6].

Cette classe hérite donc des attributs et méthodes définis successivement dans les classes `Article` et `Boisson`. Cependant, le calcul effectué pour calculer le prix net de la boisson doit être modifié pour prendre en compte les droits d'accise. Pas de problème, l'héritage offre la possibilité de redéfinir la méthode `prixNet` dans la classe `BoissonAlcoolisee` [PROG. 4.3]. Pour cela, l'entête de la méthode redéfinie doit être absolument identique à l'original. La méthode `prixNet` originale reste applicable sur un objet la classe `BoissonAlcoolisee`, mais il faut pour cela la préfixer du nom de la classe où elle a été définie [PROG. 4.4].

L'avantage de cette technique est qu'il est ainsi possible de disposer de deux méthodes `prixNet` différentes sur des articles — en fonction du type exact de l'article —, effectuant leur calcul de deux manières différentes, tout en conservant une homogénéité de nom.

Attention : lorsqu'une méthode surchargée est redéfinie dans une classe dérivée, la redéfinition masque *toutes* les définitions de la méthode de base, et pas seulement celles redéfinies.

4.4 Le polymorphisme

Un certain nombre de conversions standards sont automatiquement définies entre une classe de base et ses classes dérivées de façon publique. Ainsi, pour une classe `A` de base et une classe `B` dérivée de `A`, des

6. Taxe appliquée à certains alcools, proportionnelle au degré d'alcool et au volume.

```
 1  class BoissonAlcoolisee : public Boisson {
 2  public:
 3    // Constructeur
 4    BoissonAlcoolisee(const char *nom, float prixBrut, int volume,
 5                      float degre, float accise, int quantite = 0)
 6      : Boisson(nom, prixBrut, volume, quantite),
 7        _degre(degre), _accise(accise) {}
 8
 9    // Prix net de l'article
10    float prixNet() const
11    {
12      return ((_prixBrut + _accise * _degre * _volume) * _tva);
13    }
14
15  protected:
16    float _degre;   // Degrés de la boisson
17    float _accise;  // Droits d'accise par degré et litre
18  };
```

PROG. 4.3 – *Définition de la classe* `BoissonAlcoolisee`.

```
 1  #include "BoissonAlcoolisee.H"
 2
 3  int main()
 4  {
 5    Article table("Table", 250, 1);
 6    BoissonAlcoolisee biere("Bière", 3.40, .25, 4.7, .062);
 7
 8    cout << "Prix de la table : " << table.prixNet() << endl;
 9    cout << "Prix de la bière : " << biere.prixNet() << endl;
10    cout << "Prix de la bière si elle n'était pas alcoolisée : "
11         << biere.Article::prixNet() << endl;
12  }
```

PROG. 4.4 – *Utilisation de la classe* `BoissonAlcoolisee`.

conversions implicites sont définies :

- d'un objet de type B vers un objet de type A,
- d'un pointeur sur un objet de type B vers un pointeur sur un objet de type A,
- d'une référence sur un objet de type B vers une référence sur un objet de type A.

Ce type de conversions n'est pas risqué puisqu'un objet de type B est avant tout un objet de type A. Dans le premier cas (celui des objets), c'est une conversion d'*objet* qui est effectuée : l'objet de type A est affecté à l'objet de type B. Dans ce cas, seul les attributs définis dans A sont pris en compte, les éventuels attributs supplémentaires définis dans B ne sont pas pris en considération. Dans les deux autres cas (ceux des pointeurs et références), c'est seulement une conversion de *type* qui est réalisée : l'objet en lui-même n'est pas affecté. Ce qui implique qu'un pointeur sur type A peut pointer vers un objet de type B. C'est ce qu'on appelle le *polymorphisme*[7]. Voir [PROG. 4.5] pour des exemples pratiques de polymorphisme.

```
 1  #include <iostream>
 2  #include "Article.H"
 3  #include "BoissonAlcoolisee.H"
 4
 5  int main()
 6  {
 7    // Avec des objets
 8
 9    Article a("Sucre", 4.55);
10    BoissonAlcoolisee b("Biere", 3.4, .25, 4.7, .062);
11
12    cout << a.nom() << endl;   // Sucre
13    cout << b.nom() << endl;   // Biere
14
15    a = b;
16
17    cout << a.nom() << endl;   // Biere
18
19    // Avec des pointeurs
20
21    Article *pa = new Article("Sucre", 4.55);
22    BoissonAlcoolisee *pb = new BoissonAlcoolisee("Biere", 3.4, .25,
23                                                  4.7, .062);
24
25    cout << pa->prixNet() << endl;   // Article::prixNet()
26    cout << pb->prixNet() << endl;   // BoissonAlcoolisee::prixNet()
27
28    pa = pb;
29    cout << pa->prixNet() << endl;   // Article::prixNet()
30
31    pb = (BoissonAlcoolisee*) pa;    // Juste car pa pointe vers un alcool
32    cout << pb->prixNet() << endl;   // BoissonAlcoolisee::prixNet()
33  }
```

PROG. 4.5 – *Exemples de polymorphisme.*

Quelques remarques :

- On ne peut pas accéder aux membres spécifiques à B à travers un pointeur ou une référence de type A. Les seuls membres accessibles sont donc ceux qui sont définis au niveau de la classe A. Un

7. En biologie, c'est la caractéristique d'un organisme qui peut se présenter sous diverses formes sans changer de nature.

exemple pratique de cette remarque se trouve ligne 29 : même si `pa` pointe vers un objet de type `BoissonAlcoolisee`, c'est bien la méthode `Article::prixNet` qui est appelée.

– Si la conversion implicite marche très bien du type `B` vers le type `A`, ce n'est pas le cas de la conversion inverse (d'un objet de type `A` vers un objet de type `B`) qui nécessite une conversion de type explicite (un *cast*). Voir en particulier l'exemple de la ligne 31 qui illustre cette situation.
  D'autre part, pour réaliser cette conversion, le programmeur **doit être sûr** qu'elle a un sens, c'est-à-dire que `pa` pointe effectivement vers une boisson alcoolisée. Il existe plusieurs moyens d'avoir cette assurance (ce qui est utile sur des exemples plus complets que celui présenté) :
  1. Définir dans la classe de base un attribut permettant de stocker un identificateur unique pour chaque classe de la hiérarchie. En interrogeant préalablement cet identificateur avant une conversion de type, on sait exactement quel est le type de l'objet en présence, et on peut donc appliquer la conversion *ad hoc*. Cette technique a longtemps été la seule disponible dans ce genre de situation.
  2. Utiliser les nouveaux opérateurs de conversion définis dans la norme finale C++, qui permettent de réagir dynamiquement face aux types des objets manipulés, sans avoir à implanter quelque chose de plus. Ces nouveaux opérateurs ne sont cependant pas encore supportés par tous les compilateurs C++ et il est toujours recommandé d'utiliser l'ancienne méthode si le programme C++ est destiné à être porté sur d'autres compilateurs que celui utilisé lors de la conception. Ces opérateurs sont décrits [§ C.2].

4.5 La liaison dynamique

L'héritage et le polymorphisme sont des fonctionnalités très puissantes pour représenter et manipuler non seulement des objets, mais également des familles d'objets. Il existe cependant une entrave à la bonne maniabilité de ces familles d'objets : c'est la liaison statique des méthodes à la compilation. En effet, dans les exemples que nous avons étudiés jusqu'à présent, le choix de la méthode à appliquer à un objet est déterminé à la compilation. C'est pour cette raison d'ailleurs que l'appel de la ligne 29 présent [PROG. 4.5] a provoqué l'appel de la fonction `Article::prixNet`. Le compilateur s'est basé sur le type du pointeur `pa` pour déterminer la méthode concernée et a effectué une liaison statique.

Ce comportement est parfois assez contraignant. Imaginons en effet que nous ayons un tableau de pointeurs sur des objets de type `Article`, et que ces pointeurs pointent vers des articles ou des objets issus des classes dérivées de `Article` (comme des alcools par exemple), ceci grâce au polymorphisme. Comment faire pour afficher le prix net de chacun des articles contenus dans ce tableau ? Une des solutions est de convertir chacun de ces pointeurs, afin d'accorder leur type avec celui de l'objet pointé, et de calculer ensuite le prix net [PROG. 4.6]. Cette solution n'est pas très élégante, et le programme doit être modifié à chaque ajout d'un article redéfinissant la méthode `prixNet`, ce qui pose des problèmes évidents de maintenance...

Pour éviter ce genre de situations, C++ permet de bénéficier de la *liaison dynamique*. Pour cela, il suffit de définir les méthodes qui sont concernées par ce type de liaison comme *virtuelles*. Cette définition s'effectue en préfixant la méthode concernée du mot-clé `virtual`[8] lors de la première déclaration de la méthode [PROG. 4.7]. Une méthode déclarée virtuelle dans une classe de base le reste en effet dans toutes les classes dérivées de la hiérarchie. Dès lors, il n'est plus nécessaire de convertir au fur et à mesure les différents articles. La méthode qui correspond au type d'objet manipulé est dynamiquement déterminée, c'est-à-dire pendant l'exécution du programme et non pas lors de sa compilation [PROG. 4.8].

Remarque importante : afin de pouvoir détruire tous les objets comme indiqué ligne 11 [PROG. 4.8],

8. Attention : ce mot-clé n'est utilisé que lors de déclarations, il ne faut pas le reprendre lors de la *définition* des méthodes.

```
 1  #define MAXELTS 1000
 2
 3  int main()
 4  {
 5    Article *lesArticle[MAXELTS];
 6
 7    // Initialisation du tableau
 8    // avec des articles hétérogènes
 9
10    for (int i = 0; i < MAXELTS; i++)
11      switch (lesArticle[i]->type()) {
12      case ARTICLE:
13        cout << lesArticle[i]->prixNet() << endl;
14        break;
15
16      case ALCOOL:
17        BoissonAlcoolisee *ba;
18
19        ba = (BoissonAlcoolisee*) lesArticle[i];
20        cout << ba->prixNet() << endl;
21        break;
22
23        // Et ainsi de suite pour les autres articles...
24      }
25  }
```

PROG. 4.6 – *Résolution statique de l'application d'une méthode à une famille d'objets.*

```
1  class Article {
2  public:
3
4    virtual float prixNet() const;
5  };
```

PROG. 4.7 – *Déclaration d'une méthode virtuelle dans la classe* `Article`.

```
 1  #define MAXELTS 1000
 2
 3  int main()
 4  {
 5    Article *lesArticle[MAXELTS];
 6
 7    ...
 8
 9    for (int i = 0; i < MAXELTS; i++) {
10      cout << lesArticle[i]->prixNet() << endl;
11      delete lesArticle[i];
12    }
13
14    // Plus de distinction à faire : la bonne méthode est
15    // automatiquement appelée en fonction du type
16    // dynamique de l'article courant
17  }
```

PROG. 4.8 – *Résolution dynamique de l'application d'une méthode à une famille d'objets.*

il est *nécessaire* de déclarer comme virtuel le destructeur de la classe de base (`Article`). Dans le cas contraire, c'est uniquement le destructeur de la classe de base qui sera appelé sur chacun des articles, quelque soit leur type réel... On déclare donc conventionnellement *tous*[9] les destructeurs de classe comme virtuels, à moins d'avoir de bonnes raisons de ne pas le faire.

4.6 Les classes abstraites

Nous avons vu dans le paragraphe précédent comment déclarer des fonctions virtuelles. C++ autorise la déclaration de *fonctions virtuelles pures*, c'est-à-dire de fonctions virtuelles dont la définition n'est pas donnée. Dans l'exemple présenté [PROG. 4.9], une classe `FigureGeometrique` est définie, destinée

```
1  class FigureGeometrique {
2  public:
3    // Constructeur
4    FigureGeometrique();
5
6    // Dessin (Fonction virtuelle pure)
7    virtual void dessin() const = 0;
8
9    ...
10 };
```

PROG. 4.9 – *Déclaration d'une fonction virtuelle pure.*

à être la classe de base de toutes les figures géométriques. Suivant leurs caractéristiques, ces figures vont être implantées dans diverses classes dérivées (on pourrait imaginer les figures en 2D, en 3D, etc.). Dans ce contexte, la classe `FigureGeometrique` sert de cadre générique pour la définition des méthodes dans les classes dérivées. Ainsi, chaque figure doit pouvoir être dessinée, et c'est ce qui est exprimé par l'intermédiaire de la fonction virtuelle pure `dessin`.

La fonction `dessin`, qui est une fonction virtuelle pure, n'a pas de définition. Cette définition devra être donnée dans les classes dérivées. Lorsqu'une classe, comme la classe `FigureGeometrique`, possède au moins une méthode virtuelle pure, elle ne peut pas être instanciée. Elle est alors qualifiée de *classe abstraite*. Si les classes abstraites ne peuvent être instanciées, elles peuvent cependant être utilisées dans le cadre du polymorphisme, ce qui est très pratique pour manipuler de façon homogène une famille d'objets [PROG. 4.10].

9. Même sur des classes qui ne sont pas destinées à être classe de base d'une hiérarchie. Cela peut éviter des problèmes si ces classes sont finalement dérivées.

```
 1  #include "FigureGeometrique.H"
 2
 3  #define NBFIG 1000
 4
 5  int main()
 6  {
 7    FigureGeometrique *lesFigures[NBFIG];
 8    ...
 9    // Dessin de toutes les figures géométriques
10
11    for (int i = 0; i < NBFIG; i++)
12      lesFigures[i]->dessin();
13    ...
14  }
```

PROG. 4.10 – *Utilisation d'une classe abstraite dans le cadre du polymorphisme.*

4. L’héritage

Chapitre 5

La bibliothèque STL

5.1 Introduction

La bibliothèque STL (*Standard Template Library*[10]) est certainement l'un des atouts de C++. Cette bibliothèque fournit un ensemble de composants C++ bien structurés qui marchent de façon cohérente et peuvent aussi être adaptés facilement. En effet, il est possible d'utiliser les structures de données proposées par STL avec des algorithmes personnels, les algorithmes de la bibliothèque avec des structures de données personnelles, ou d'utiliser toutes les composantes STL ! Lors de sa conception, l'accent a été mis sur l'efficacité et sur l'optimisation des composants, ce qui en fait un outil très puissant.

Ce chapitre présente les généralités liées à STL. Pour en tirer pleinement partie, une bonne documentation s'avère nécessaire. Un des meilleurs ouvrages de référence de cette bibliothèque est celui de Musser & Saini [Mus 96]. Il est également possible de trouver des informations sur internet :

- `http://www.sgi.com/tech/stl/` : manuel d'utilisation de STL.
- `http://www.cs.rpi.edu/~musser/doc.ps` : autre manuel d'utilisation.
- `http://www.infosys.tuwien.ac.at/Research/Component/tutorial/prwmain.htm` : tutoriel d'utilisation de STL.

STL contient cinq types de composants : des containers, des itérateurs, des algorithmes, des objets-fonctions et des adaptateurs. Nous nous intéressons dans ce chapitre aux trois premiers composants.

5.2 Les containers

Les containers sont des objets qui permettent de stocker d'autres objets. Ils sont décrits par des classes génériques représentant les structures de données logiques les plus couramment utilisées : les listes, les tableaux, les ensembles... Ces classes sont dotées de méthodes permettant de créer, de copier, de détruire ces containers, d'y insérer, de rechercher ou de supprimer des éléments. La gestion de la mémoire, c'est-à-dire l'allocation et la libération de la mémoire, est contrôlée directement par les containers, ce qui facilite leur utilisation. L'exemple [PROG. 5.1] présente une application où les valeurs entières saisies par un utilisateur sont stockées dans une liste et dans un tableau ; un exemple d'exécution est présenté [PROG. 5.2].

10. Bibliothèque standard générique.

```
 1  #include <iostream>
 2  #include <vector>
 3  #include <list>
 4
 5  int main()
 6  {
 7    vector<int> tableauEntiers;  // Crée un tableau d'entiers vide
 8    list<int> listeEntiers;      // Crée une liste d'entiers vide
 9    int unEntier;
10
11
12    // Saisie des entiers
13
14    cout << "Saisir le prochain entier (-1 pour finir) : ";
15    cin >> unEntier;
16
17    while (unEntier != -1) {
18      tableauEntiers.push_back(unEntier);
19      listeEntiers.push_back(unEntier);
20
21      cout << "Saisir le prochain entier (-1 pour finir) : ";
22      cin >> unEntier;
23    }
24
25    // Nombre d'éléments des containers
26
27    cout << "Il a y " << tableauEntiers.size()
28         << " éléments dans le tableau" << endl;
29
30    cout << "Il a y " << listeEntiers.size()
31         << " éléments dans la liste" << endl;
32
33    // Accès à des éléments
34
35    cout << "Premier élément du tableau : "
36         << tableauEntiers.front() << endl;
37
38    cout << "Premier élément de la liste : "
39         << listeEntiers.front() << endl;
40
41    int milieu = tableauEntiers.size() / 2;
42
43    cout << "Élément de milieu de tableau : "
44         << tableauEntiers[milieu] << endl;
45  }
```

PROG. 5.1 – *Utilisation des listes et tableaux* STL.

```
 1  Saisir le prochain entier (-1 pour finir) : 4
 2  Saisir le prochain entier (-1 pour finir) : 5
 3  Saisir le prochain entier (-1 pour finir) : 3
 4  Saisir le prochain entier (-1 pour finir) : 7
 5  Saisir le prochain entier (-1 pour finir) : 6
 6  Saisir le prochain entier (-1 pour finir) : 3
 7  Saisir le prochain entier (-1 pour finir) : -1
 8  Il a y 6 éléments dans le tableau
 9  Il a y 6 éléments dans la liste
10  Premier élément du tableau : 4
11  Premier élément de la liste : 4
12  Élément de milieu de tableau : 7
```

PROG. 5.2 – *Exemple d'utilisation de [*PROG. *5.1].*

Quelques remarques sur les containers et sur l'exemple présenté :

- Un certain nombre de méthodes sont disponibles sur tous les types de containers, ce qui permet d'homogénéiser leur utilisation. C'est le cas par exemple de la méthode `push_back` (lignes 18 et 19) qui insère un nouvel élément à la fin d'un container.
- D'autres méthodes ou opérateurs sont disponibles en fonction du type de container utilisé. L'opérateur `[]` est disponible sur les objets de type `vector`, mais pas sur ceux de type `list` : il permet d'accéder directement à un élément. Plus de précisions sur ces méthodes sont disponibles [§ B.1].
- L'utilisateur n'a pas à se soucier de l'allocation ou de la libération de la mémoire. C'est vrai lors de l'insertion d'éléments (lignes 18 et 19) et aussi à la fin du programme : aucune instruction particulière n'est nécessaire pour restituer la mémoire occupée par les containers. À la sortie du bloc dans lequel ils sont définis, leur destructeur se charge de libérer toutes les ressources occupées.
- Les containers peuvent manipuler n'importe quel type de données, à partir du moment où la classe correspondante est dotée d'un certain nombre de méthodes nécessaires à STL (pour une classe `X`) :
  - `X()` : un constructeur par défaut,
  - `X(const X&)`, un constructeur par copie,
  - `operator=(const X&)` : l'opérateur d'affectation,
  - `operator==(const X&)` : l'opérateur d'égalité,
  - `operator<(const X&)` : l'opérateur inférieur (utile uniquement pour les tris).

Les différentes sortes de containers disponibles sont :

- `vector` : container implantant les tableaux, qui autorise les accès directs sur ses éléments. Les opérations de mise à jour (insertion, suppression) sont réalisées en un temps constant à la fin du container, et en un temps linéaire (dépendant du nombre d'éléments) aux autres endroits.
- `list` : container implantant les listes doublement chaînées, dédié à la représentation séquentielle de données. Les opérations de mise à jour sont effectuées en un temps constant à n'importe quel endroit du container.
- `deque` : container similaire au `vector`, effectuant de plus les opérations de mise à jour en début de container en un temps constant.
- `set` : container implantant les ensembles où les éléments ne peuvent être présents au plus qu'en un seul exemplaire.
- `multiset` : container implantant les ensembles où les éléments peuvent être présents en plusieurs exemplaires.
- `map` : container implantant des ensembles où un type de données appelé *clé* est associé aux éléments à stocker. On ne peut associer qu'une seule valeur à une clé unique. On appelle aussi ce type de

container *tableau associatif*.
- `multimap` : container similaire au `map` supportant l'association de plusieurs valeurs à une clé unique.
- `stack` : container implantant les piles, qui sont des listes spéciales, dites LIFO [11].
- `queue` : container implantant les files, qui sont des listes spéciales, dites FIFO [12].

Une carte de référence sur les containers existants et sur les méthodes dont ils dont dotés est disponible [§ B.1].

5.3 Les itérateurs

Les itérateurs sont une généralisation des pointeurs, ce qui permet au programmeur de travailler avec des containers différents de façon uniforme. Ils permettent de spécifier une position à l'intérieur d'un container, peuvent être incrémentés ou déréférencés (à la manière des pointeurs utilisés avec l'opérateur de déréférencement `'*'`), et deux itérateurs peuvent être comparés. Tous les containers sont dotés d'une méthode `begin` qui renvoie un itérateur sur le premier de leurs éléments, et d'une méthode `end` qui renvoie un itérateur sur une place se trouvant *juste après* le dernier de leurs éléments. On ne peut ainsi pas déréférencer l'itérateur renvoyé par la méthode `end`. Un exemple d'utilisation des itérateurs se trouve [PROG. 5.3].

```
 1  #include <iostream>
 2  #include <list>
 3
 4  int main()
 5  {
 6    list<int> lesEntiers;
 7
 8    // Ici, des instructions pour initialiser la
 9    // liste des entiers
10
11    ...
12
13    // Affichage des éléments contenus dans la liste
14
15    list<int>::iterator it;
16
17    for (it = lesEntiers.begin(); it != lesEntiers.end(); it++)
18      cout << *it << endl;
19  }
```

PROG. 5.3 – *Utilisation typique des itérateurs pour un parcours de container.*

Ces itérateurs sont dotées de méthodes permettant de les manipuler, décrites [§ B.2]. Il existe une hiérarchie d'itérateurs (qui n'est pas liée à un quelconque héritage) :

- Les itérateurs d'entrée (*input iterators*) : ils permettent d'accéder séquentiellement à des sources de données. Cette source peut-être un container, un flot.
- Les itérateurs de sortie (*output iterators*): ils permettent de préciser la localisation d'une destination permettant de stocker des données. Cette source peut-être un container, un flot.

11. *Last In, First Out* : dernier entré, premier sorti.
12. *First In, First Out* : premier entré, premier sorti.

- Les itérateurs à sens-unique (*forward iterators*) : ils sont dotés de toutes les méthodes des itérateurs d'entrée et de sortie. Ils sont utilisés pour parcourir séquentiellement une séquence de données dans un sens. Ils ne peuvent pas être utilisés pour effectuer des retours en arrière.
- Les itérateurs à double-sens (*bidirectional iterators*) : ils sont dotés de toutes les méthodes des itérateurs à sens-unique. Ils sont également utilisés pour effectuer des parcours séquentiels de données, qu'ils peuvent effectuer dans les deux sens.
- Les itérateurs à accès direct (*random-access iterators*) : ils sont dotés de toutes les méthodes des itérateurs à double-sens. Ils permettent d'accéder directement à des valeurs contenues dans un container, sans être obligé d'y accéder séquentiellement.

Tous les containers disponibles sous STL fournissent au moins des itérateurs à double-sens, et certains fournissent des itérateurs à accès direct (voir [§ B.1] pour les itérateurs par défaut renvoyés par chaque type de container).

5.4 Les algorithmes

Les algorithmes sont des fonctions C++ génériques qui permettent d'effectuer des opérations sur les containers. Afin de pouvoir s'appliquer à plusieurs types de containers, les algorithmes ne prennent pas de containers en arguments, mais des itérateurs qui permettent de désigner une partie ou tout un container. De ce fait, il est même possible d'utiliser ces algorithmes sur des objets qui ne sont pas des containers. On peut par exemple utiliser un `istream_iterator` (voir [§ B.2]) comme paramètre d'un algorithme, qui va alors s'appliquer à l'entrée standard. Certains algorithmes ne nécessitent que des itérateurs de base (d'entrée ou de sortie), et d'autres nécessitent des itérateurs plus évolués, comme la fonction `sort` [13] (effectuant un tri) qui nécessite un itérateur à accès direct.

Les algorithmes disponibles sont décrits en annexe [§ B.3]. Pour les utiliser, il suffit d'inclure l'en-tête `algorithm`. Un exemple d'utilisation des algorithmes est présenté [PROG. 5.4], et un résultat produit à partir de cet exemple est présenté [PROG. 5.5].

13. Du coup, cet algorithme n'est pas applicable sur les listes qui ne fournissent que des itérateurs à double sens. C'est pour cette raison que la classe `list` est dotée d'une méthode `sort`.

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  int main()
6  {
7    vector<int> tableauEntiers;  // Crée un tableau d'entiers vide
8    int unEntier;
9
10   // Saisie des entiers
11
12   cout << "Saisir le prochain entier (-1 pour finir) : ";
13   cin >> unEntier;
14
15   while (unEntier != -1) {
16     tableauEntiers.push_back(unEntier);
17
18     cout << "Saisir le prochain entier (-1 pour finir) : ";
19     cin >> unEntier;
20   }
21
22   // Tri du tableau
23
24   sort(tableauEntiers.begin(), tableauEntiers.end());
25
26   // Affichage des éléments triés
27
28   vector<int>::iterator it;
29
30   for (it = tableauEntiers.begin(); it != tableauEntiers.end(); it++)
31     cout << *it << " ";
32
33   cout << endl;
34 }
```

PROG. 5.4 – *Utilisation de l'algorithme de tri.*

```
1  Saisir le prochain entier (-1 pour finir) : 5
2  Saisir le prochain entier (-1 pour finir) : 3
3  Saisir le prochain entier (-1 pour finir) : 8
4  Saisir le prochain entier (-1 pour finir) : 10
5  Saisir le prochain entier (-1 pour finir) : 3
6  Saisir le prochain entier (-1 pour finir) : 6
7  Saisir le prochain entier (-1 pour finir) : 9
8  Saisir le prochain entier (-1 pour finir) : -1
9  3 3 5 6 8 9 10
```

PROG. 5.5 – *Exemple d'utilisation de [*PROG. 5.4*].*

Annexe A

Format des entrées-sorties

A.1 La classe `ios`

La classe `ios` est la classe de base des classes représentant les flots. Toutes les classes de flot présentées ci-dessous héritent donc de cette classe et des méthodes qu'elle définit (non exhaustif) :

- `int ios::good()` : retourne une valeur nulle s'il y a eu un échec lors de la dernière opération d'entrée-sortie, et une valeur non nulle sinon.
- `int ios::fail()` : retourne le contraire de la méthode précédente.
- `int ios::eof()` : retourne une valeur non nulle si la fin de fichier est atteinte, et la valeur nulle sinon.

La classe `ios` définit aussi un certain nombre de méthodes relatives au format des informations manipulées (non exhaustif) (voir [PROG. A.1] pour un exemple d'utilisation) :

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5    cout << "Largeur standard : "
 6         << cout.width() << endl;     // Affiche : 0
 7
 8    cout.width(10);
 9    cout.fill('#');
10
11    cout << 654 << endl;             // Affiche : #######654
12  }
```

PROG. A.1 – *Exemples de formatage de texte.*

- `int ios::width(int n)` : positionne la largeur du champ (le nombre de caractères) de sortie.
- `int ios::width()` : retourne la largeur du champ de sortie.
- `char ios::fill(char c)` : positionne le caractère de remplissage (utilisé pour remplir toute la largeur des champs).
- `char ios::fill()` : retourne le caractère de remplissage.
- `int ios::precision(int p)` : positionne la précision, c'est-à-dire le nombre de caractères (y compris le point) qu'occupe un réel.

– `int ios::precision()` : retourne la précision.

Enfin, cette classe définit un certain nombre d'opérations qui peuvent être utilisées sans les préfixer du flot sur lequel on travaille, et autorise ainsi l'écriture de lignes du type :

```
cout << "La valeur octale de 154 est : " << oct << 154 << endl;
```

Parmi ces opérations, on trouve :

– `endl` : écrit un `'\n'` et vide le tampon du flot.
– `ends` : écrit un `'\0'` et vide le tampon du flot.
– `flush` : vide le tampon du flot.
– `dec` : la prochaine opération d'entrée-sortie se fera en décimal.
– `hex` : la prochaine opération d'entrée-sortie se fera en hexadécimal.
– `oct` : la prochaine opération d'entrée-sortie se fera en octal.
– `ws` : saute les espaces lors d'une lecture sur un flot d'entrée.
– `setfill(int c)` : positionne le caractère de remplissage pour la prochaine opération d'entrée-sortie.
– `setprecision(int p)` : positionne la précision de la prochaine opération d'entrée-sortie à `p` chiffres.
– `setw(int n)` : positionne la largeur de la prochaine entrée-sortie à `n` caractères.

A.2 La classe `ostream`

Cette classe est dédiée aux sorties formatées ou non. Un objet de type `ostream` est défini par défaut dans tout programme C++ : c'est la variable `cout`. La classe `ostream` surcharge l'opérateur `<<` pour tous les types prédéfinis du C++. Il faut, si besoin est, le surcharger pour les nouveaux types introduits. Un exemple de surcharge est présenté [§ 3.10]. En plus de l'opérateur `<<`, la classe `ostream` est dotée des méthodes suivantes (non exhaustif) :

– `ostream &ostream::put(char c)` : insère un caractère dans le flot.
Exemple : `cout.put('\n')`.
– `ostream &ostream::write(const char *, int n)` : insère `n` caractères dans le flot.
– `ostream &ostream::flush()` : vide le tampon du flot.

A.3 La classe `istream`

Cette classe est dédiée aux entrées formatées ou non. Un objet de type `istream` est défini par défaut dans tout programme C++ : c'est la variable `cin`. La classe `istream` surcharge l'opérateur `>>` pour tous les types prédéfinis du C++. Il faut, si besoin est, le surcharger pour les nouveaux types introduits. Un exemple de surcharge est présenté [§ 3.10]. En plus de l'opérateur `>>`, la classe `istream` est dotée des méthodes suivantes (non exhaustif) :

– `int istream::get()` : retourne la valeur du caractère lu (EOF si la fin du flot est atteinte).
– `istream &istream::get(char &c)` : extrait le premier caractère du flot (même si c'est un espace) et le place dans `c`.

- `int &istream::peek()` : lit le prochain caractère du flot sans l'enlever du flot (renvoie EOF si la fin du flot est atteinte).
- `istream &istream::get(char *ch, int n, char delim = '\n')` : extrait `(n - 1)` caractères du flot et les place à l'adresse `ch` (le tampon). La lecture s'arrête éventuellement après le délimiteur s'il est rencontré.
- `istream &istream::getline(char *ch, int n, char delim = '\n')` : identique à la méthode précédente sans placer le délimiteur dans le tampon.
- `istream &istream::read(char *ch, int n)` : extrait au plus `n` caractères du flot et les place à l'adresse `ch`. Le nombre de caractères effectivement lus peut être obtenu grâce à la méthode `gcount`.
- `int istream::gcount()` : retourne le nombre de caractères extraits lors de la dernière lecture.
- `istream &istream::flush()` : vide le tampon du flot.

A.4 Les fichiers

Les entrées-sorties sur les fichiers sont également réalisées avec des flots en C++. Ce type d'opérations nécessite l'inclusion de l'en-tête `fstream` en plus de l'en-tête `iostream`. Les deux grandes classes permettant de réaliser ces opérations sont :

- La classe `ofstream` : cette classe est dédiée aux écritures réalisées dans des fichiers. La classe `ofstream` est dérivée de la classe `ostream` et bénéficie donc de toutes les méthodes définies dans cette classe. Un exemple d'utilisation est présenté [PROG. A.2].

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6    // Deux modes d'ouverture sont possibles :
7    //   - ios::out -> création, fichier écrasé si existant
8    //   - ios::app -> ajout en fin de fichier
9
10   ofstream fichierSortie("donnees.txt", ios::out);
11
12   // Test d'ouverture du fichier
13
14   if (!fichierSortie) {
15     cerr << "Problème d'ouverture de fichier" << endl;
16     exit(1);
17   }
18
19   fichierSortie << "J'écris des caractères dans le fichier "
20                 << "et des nombres : " << 10 << " " << 20
21                 << endl;
22
23   // Fermeture du fichier
24
25   fichierSortie.close();
26 }
```

PROG. A.2 – *Écritures dans un fichier.*

– La classe `ifstream` : cette classe est dédiée aux lectures réalisées dans des fichiers. La classe `ifstream` est dérivée de la classe `istream` et bénéficie donc de toutes le méthodes définies dans cette classe. Un exemple d'utilisation est présenté [PROG. A.3].

```
1  #include <iostream>
2  #include <fstream>
3
4  int main()
5  {
6    // Ouverture du fichier
7
8    ifstream fichierEntree("donnees.txt", ios::in);
9
10   // Test d'ouverture du fichier
11
12   if (!fichierEntree) {
13     cerr << "Problème d'ouverture de fichier" << endl;
14     exit(1);
15   }
16
17   char buf[1024];
18
19   // Tant qu'il a y des lignes dans le fichier, on les
20   // lit et on les affiche à l'écran
21
22   while (!fichierEntree.eof()) {
23     fichierEntree.getline(buf, 1024);
24     cout << buf << endl;
25   }
26
27   // Fermeture du fichier
28
29   fichierEntree.close();
30 }
```

PROG. A.3 – *Lectures dans un fichier.*

Annexe B

Carte de référence STL

B.1 Les containers

Tous les containers sont dotés des caractéristiques suivantes :

- Certains types prédéfinis (`typedefs`) :
  - `size_type`
  - `pointer`
  - `const_pointer`
  - `reference`
  - `const_reference`
- Des types prédéfinis pour la création d'itérateurs :
  - `iterator`
  - `const_iterator`
  - `reverse_iterator`
  - `const_reverse_iterator`
- `constructor()` : pour la création de containers vides
- `copy-constructor()`
- `destructor()`
- `bool empty() const`
- `size_type size() const` : nombre d'éléments du container
- `size_type max_size() const` : capacité (mémoire occupée) maximum du container
- `container-ref operator=(const-container-ref)` : remplacement de tout le contenu
- `void swap(container-ref)` : inverse tout le contenu
- `bool operator==(const-container-ref)` : teste l'égalité sur tous les éléments
- `bool operator<(const-container-ref)`
- `begin()` et `end()` : méthodes pour accéder au contenu
- `insert()`

Propriétés des containers

Container	Iterateur par déf.	Constructeurs	Accesseurs	Méthodes
array	-	-	op[]	-
vector	random-acc	copy	front(), back(), op[], at()	push_back(), pop_back()
bit_vector	random-acc	copy	front(), back(), op[]	push_front(), pop_back(), flip(), assign()
list	bidirectional	copy	front(), back()	push_front(), push_back(), pop_front(), pop_back, sort(), splice(), remove(), reverse(), unique(), merge()
deque	random-acc	copy	front(), back(), op[], at()	push_front(), push_back(), pop_front(), pop_back()
Associative				
set	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
multiset	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
map	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count(), op[]
multimap	bidirectional	copy	find(), lower_bound(), upper_bound(), equal_range()	count()
Adaptor				
stack	n/a	copy	top()	push(), pop()
queue	n/a	copy	front(), back()	push(), pop()
priority_queue	n/a	copy	top()	push(), pop()
Special				
bitset	n/a	copy	front(), back(), op[]	push_front(), pop_back(), test(), any(), none(), op&=, op|=, opˆ=, op<<, op>>, set(), reset(), to_ulong(), to_string(), count(), flip()

B.2 Les itérateurs

Propriété des itérateurs

Iterateur	Constructeurs	Accesseurs	Déplacement	Comparaison
all	copy		op++(), op++(int)	
output	"	op*() *write only*	"	
input	"	op*() *read only*	"	op=, =()
forward	"	op*() *read write*	", operator=()	"
bidirectional	"	"	", op–(), op–(int)	"
random-access	"	"	", op+=, op+, op-=, op-, op[]	", op<()
C style pointer	"	"	"	"

Itérateurs spéciaux

istream_iterator	lit un flot d'entrée C++
ostream_iterator	écrit dans un flot de sortie C++
back_insert_iterator	insère à la fin d'un container
front_insert_iterator	insère au début d'un container
insert_iterator	insère dans un container à n'importe quelle position
raw_storage_iterator	itérateur sur mémoire allouée non initialisée
reverse_bidirectional_iterator	itérateur dans le sens inverse

B.3 Les algorithmes

Dans le tableau suivant, les arguments des fonctions et des valeurs de retour sont codés de la manière suivante :

- r random access iterator
- b bidirectional iterator
- f forward iterator
- i input iterator
- o output iterator
- V value
- R reference
- P pair
- B bool
- p unary predicate
- p2 binary predicate
- c compare function
- F unary function
- F2 binary function
- n count
- [...] optional args

Algorithmes en STL

Nom	Retourne	Arguments	Description
Finding			
adjacent_find	i	i,i[,p2]	find sequence of equal elements
binary_search	B	f,f,V[,c]	find a value in a sorted range
count	void	i,i,V,R	count matching elements
count_if	void	i,i,p,R	count elements which satisfy *p*
find	i	i,i,V	locate an equal element
find_if	i	i,i,p	locate an element which satisfies *p*
search	f	f,f,f,f[,p2]	locate a subrange within a range
search	f	f,f,n,V[,p2]	locate a subrange within a range
find_end	f	f,f,f,f[,p2]	find the last subrange which satisfies; like *search* but from the end
lower_bound	f	f,f,V[,c]	returns the *first* possible insert location into a sorted collection
upper_bound	f	f,f,V[,c]	returns the *last* possible insert location into a sorted collection
equal_range	P	f,f,V[,c]	returns the *range of possible insert locations* into a sorted collection
min_element	i	i,i[,c]	find the *smallest*
max_element	i	i,i[,c]	find the *largest*
Applying			
for_each	F	f,f,F	apply a function to a range
transform	o	i,i,o,F or i,i,i,o,F2	apply an operation against a range
replace	v	f,f,V,V	replace all matching elements with a new one
replace_if	v	f,f,p,V	replace all matching elements with a new one
replace_copy	o	i,i,o,V,V	replace during copy, all matching elements with a new one
replace_copy_if	o	i,i,o,p,V	replace during copy, all matching elements with a new one
Filling			
fill	v	f,f,V	fill with a value
fill_n	v	f,n,V	fill with a single value
generate	v	f,f,unary_op	fill with generated values
generate_n	v	f,n,unary_op	fill with generated values
Enumerating			
count	v	i,i,V,R	count the number of matches
count_if	v	i,i,p2,R	count the number of matches, using pred
mismatch	P	i,i,i[,p2]	returns the first subrange than does not match
equal	B	i,i,i[,p2]	*true* if the ranges match
lexigraphical_compare	B	i,i,i,[,c]	*true* if the ranges match

Nom	Retourne	Arguments	Description
Copying			
copy	o	i,i,o	copy one range to another
copy_backward	b	b,b,b	reverse copy one range to another
swap_ranges	f	f,f,f	swap one range with another
Ordering			
remove	f	f,f,V	move unwanted entries to the end of the range
remove_if	f	f,f,p	move unwanted entries to the end of the range
remove_copy	o	i,i,o,V	copy and remove unwanted entries
remove_copy_if	o	i,i,o,p	copy and remove unwanted entries
unique	f	f,f[,p2]	collapse the range so that multiple copies of *equal* elements are removed
unique_copy	o	i.i,o[,p2]	copy the range skipping multiple copies of *equal* elements
reverse	v	b,b	reverse the order of a range
reverse_copy	o	b,b,o	reverse the order of a range
rotate	v	f,f,f	rotate a range, given first, middle and last
rotate_copy	o	f,f,f,o	rotate and copy, given first, middle and last
random_shuffle	v	r,r[,rand_gen]	shuffle the order of a range
Sorting			
partition	b	b,b,p	swaps to make all the pred-successes precede the pred-failures
stable_partition	b	b,b,p	swaps to make all the pred-successes precede the pred-failures; preserves relative order
sort	v	r,r[,c]	sorts the elements in the range
stable_sort	v	r,r[,c]	sorts the range; preserve relative order on the "equal" ones
partial_sort	v	r,r,r[,c]	sorts the range into the subrange
partial_sort_copy	r	i,i,r,r[,c]	sorts the range into the subrange at a new location
nth_element	v	r,r,r[,c]	sorts the range so that one specific one is in the right place
next_permutation	B	b,b[,c]	transforms range to next permutation
prev_permutation	B	b,b[,c]	tranforms range to previous permutation

B. Carte de référence STL

Nom	Retourne	Arguments	Description
Merging			
merge	o	i,i,i,i,o[,c]	merges two input ranges
inplace_merge	v	b,b,b[,c]	merges two input ranges
Set Support			
includes	b	i,i,i,i[,c]	tests for the elementf of one range present in another
set_union	o	i,i,i,i,o[,c]	builds the sorted union
set_intersection	o	i,i,i,i,o[,c]	intersection
set_difference	o	i,i,i,i,o[,c]	difference
set_symmetric_difference	o	i,i,i,i,o[,c]	symmetric_difference
Heap Support			
push_heap	v	r,r[,c]	adds the last element of a range to a heap
pop_heap	v	r,r[,c]	changes a heap into a smaller heap *plus* a last element
make_heap	v	r,r[,c]	changes a range into a heap
sort_heap	v	r,r[,c]	sorts a heap

Annexe C

Compléments

C.1 Les *namespaces*

Les *namespaces* (espaces de noms) ont été principalement introduits dans la norme définitive de C++ pour gérer les gros projets. Dans ce type de projet, il n'est pas rare d'utiliser plusieurs bibliothèques C++ qui peuvent parfois définir les mêmes identificateurs, ce qui génère des conflits. Avec les *namespaces*, il ne doit plus y avoir de conflits de noms : les déclarations restent cachées dans un *namespace* jusqu'à ce qu'on fasse explicitement appel à lui.

Pour définir un *namespace*, il faut utiliser le mot-clé `namespace` comme cela est présenté [PROG. C.1]. La déclaration d'un même *namespace* peut être réalisée dans plusieurs fichiers d'interface, le *namespace*

```
1  namespace MonNameSpace
2  {
3    // Toutes les déclarations sont regroupées
4    // dans ce bloc
5
6    int f();
7
8    // D'autres déclarations...
9
10   // Fin du namespace
11 };
```

PROG. C.1 – *Définition d'un* namespace.

complet résultant alors de l'union des déclarations. Dans l'exemple présenté, le nom complet de la fonction `f` devient `MonNameSpace::f`, selon une syntaxe qui est similaire à celle des méthodes membres de classe. Cependant, afin de ne pas être contraint de désigner la fonction `f` par rapport à son *namespace* lorsqu'il n'y a pas de risque de conflit, des facilités d'utilisation sont disponibles grâce à la définition d'alias ou à l'utilisation du mot-clé `using` [PROG. C.2]. L'utilisation de ces directives est cependant à bannir des fichiers d'interface, pour éviter des conflits qui pourrait apparaître dans les modules incluant ces interfaces.

Du coup, la plupart des déclarations de la bibliothèque standard C++ ont été regroupées dans un *namespace*, appelé `std`.

```
 1  // Si on souhaite définir un alias sur le nom d'un
 2  // namespace
 3
 4  namespace mon = MonNameSpace;
 5
 6  mon::f();   // Appelle MonNameSpace::f()
 7
 8  // Si la fonction f est la seule à être présente,
 9  // on peut déclarer
10
11  using MonNameSpace::f;
12
13  f();   // Appelle MonNameSpace::f()
14
15  // Si on souhaite bénéficier de toutes les déclarations
16  // d'un namespace sans avoir à les préfixer du nom
17  // du namespace
18
19  using namespace MonNameSpace;
```

PROG. C.2 – *Utilisation d'un* namespace.

C.2 Les nouveaux *casts*

Dans les nouveautés introduites dans la norme finale du langage, on trouve également de nouveaux opérateurs de conversion (*casting*). Ils sont particulièrement utiles dans des contextes tels que le polymorphisme, afin de convertir un objet d'une classe de base vers une classe dérivée [§ 4.4]. En effet, jusqu'à l'introduction de ces opérateurs, ce type de conversion délicate reposait entièrement sur les épaules du programmeur, qui avait la charge de vérifier la validité de la conversion avant de la réaliser. Du coup, cela pouvait engendrer quelques problèmes de sécurités, que ces nouveaux opérateurs sont sensés résoudre.

Certains nouveaux *casts* se basent sur une fonctionnalité ajoutée il y a peu au langage C++ : la *Run-Time Type Identification (RTTI)*[14]. L'objectif[15] de ces nouveaux opérateurs est de disposer d'une syntaxe améliorée, plus claire, d'une sémantique moins ambiguë et de réaliser des conversions de types en toute sécurité. Ces nouveaux opérateurs sont au nombre de 4 :

1. L'opérateur `static_cast<T> (expr)` : cet opérateur est utilisé pour effectuer des conversions qui sont résolues à la compilation. Il peut être utilisé pour convertir un pointeur (ou une référence) sur une classe de base vers un pointeur (ou une référence) sur une classe dérivée. L'opérateur n'effectue aucune vérification au cours de l'exécution (comme son nom l'indique) et doit donc être utilisé pour des conversions non-ambiguës. Mal utilisé, il renvoie un résultat indéfini. Il doit surtout être utilisé pour effectuer des conversions arithmétiques. Il est assez proche de la conversion traditionnelle, mais permet de supprimer des trous de sécurité qui existaient.
2. L'opérateur `const_cast<T> (expr)` : cet opérateur permet de supprimer la constance d'un objet. Ce n'est pas très naturel, mais utile dans certaines situations ; il doit être ainsi utilisé avec parcimonie. Un exemple d'utilisation se trouve [PROG. C.3]. De même, il peut être utilisé à l'intérieur d'une méthode de classe constante : en l'appliquant sur le pointeur `this`, on peut modifier par la suite l'objet courant (!).
3. L'opérateur `dynamic_cast<T> (expr)` : c'est certainement l'un des nouveaux opérateurs les plus intéressants. Il peut uniquement être utilisé sur des pointeurs ou des références pour naviguer

14. Identification des types au cours de l'exécution.
15. Ça ne reste qu'un objectif pour certains d'entre eux (NdA).

```
1  void f(Article &i)
2  {
3  }
4
5  void g(const Article &j)
6  {
7    f(j);  // Erreur : j est constant et f n'attend pas un const
8    f(const_cast<Article&> (j));  // Ok
9  }
```

PROG. C.3 – *Utilisation de l'opérateur* `const_cast`.

dans une hiérarchie de classes. Il peut être utilisé pour convertir un objet d'une classe dérivée vers un objet d'une classe de base ou inversement. Dans le premier cas, c'est une classique conversion statique qui est effectuée tandis que dans le second cas c'est une conversion dynamique qui est réalisée, en se basant sur le système RTTI. Dans ce cas, si la conversion est possible, l'opérateur de conversion renvoie un pointeur valide, ou un pointeur nul sinon. Cette fonctionnalité est très puissante comme le montre l'exemple [PROG. C.4].

```
1  #define MAXELTS 1000
2
3  int main()
4  {
5    Article *lesArticle[MAXELTS];
6
7    // Initialisation du tableau avec des articles hétérogènes
8    // Deux cas différenciés : Alcools ou autres articles
9
10   for (int i = 0; i < MAXELTS; i++) {
11       BoissonAlcoolisee *ba;
12
13       ba = dynamic_cast<BoissonAlcoolisee*> (lesArticle[i]);
14
15       // Si l'article est un alcool, affichage du nom et du
16       // degré d'alcool. Affiche du nom uniquement sinon.
17
18       if (ba)
19         cout << ba->nom() << " (" << ba->degre() << ")" << endl;
20       else
21         cout << lesArticle[i]->nom() << endl;
22   }
23 }
```

PROG. C.4 – *Utilisation de l'opérateur* `dynamic_cast`.

4. L'opérateur `reinterpret_cast<T> (expr)` : cet opérateur peut être utilisé pour convertir des objets dont les types ne sont pas en relation. Le résultat de la conversion est dépendante de l'implantation, et n'est ainsi pas portable. Il peut être utilisé dans certains contextes particuliers de conversion entre types de pointeurs de fonctions.

Conclusion : parmi les nouveaux opérateurs introduits, seuls le `static_cast` et le `dynamic_cast` sont relativement conformes à l'approche objet. Ils sont à utiliser en priorité.

C.3 Où trouver un compilateur C++ ?

Il existe un vaste choix de compilateurs C++ commerciaux. Des solutions libres et gratuites existent également, quelques pointeurs sont donnés ici :

- **Sous Windows** :
  - DJGPP (`http://www.delorie.com/djgpp/`). C'est un système complet de développement C/C++ tournant sous DOS, donc en mode texte. Il comprend un éditeur multi-fichiers, un compilateur, un gestionnaire de projet et un débogueur. Si l'environnement est plus austère que celui de CodeWarrior, le compilateur en revanche est plus récent (il s'agit en fait de GCC 2.95). Il peut être téléchargé librement à partir de `ftp://ftp.lip6.fr/pub/simtelnet/gnu/djgpp/` par exemple. Nous recommandons de sélectionner les packages suivants : `bnu281b, djdev202, gcc2951b, gpp2951b, lgp2951b, rhide14b` et de se référer au fichier `lisezmoi.1er` (répertoire `v2`) pour la procédure d'installation.
  - DEV-C++ (`http://www.bloodshed.net/devcpp.html`). Là aussi, c'est un système complet de développement C/C++, tournant cette fois sous Windows directement. Comme DJGPP, il comprend un éditeur multi-fichiers, un compilateur, un gestionnaire de projet et un débogueur. Le compilateur de base est d'ailleurs le même : GCC 2.95. Il peut être téléchargé librement à partir de `http://www.telecharger.com/`.
  - PRODUITS BORLAND (`http://www.borland.fr/download/compilateurs/`). Borland met gratuitement à la disposition de la communauté des développeurs ses compilateurs, dont les célèbres Turbo C++ et Borland C++. Le premier comprend un environnement complet en mode semi-graphique, et le deuxième correspond à un compilateur utilisable en ligne de commande, fourni avec quelques outils annexes.
- **Sous Unix** : GCC. Les sources sont téléchargeables à partir de `ftp://ftp.gnu.org/pub/gnu/gcc/`, mais l'installation est plus délicate. Il est livré en standard avec certains systèmes Unix (comme Linux), ce qui permet d'éviter l'étape d'installation. Les débogueurs du monde Unix livrés en standard ne sont généralement pas très conviviaux, mais il est possible de récupérer un débogueur graphique (DDD) à l'adresse `http://www.cs.tu-bs.de/softech/ddd/`. Les personnes travaillant sous Unix utiliseront généralement leur éditeur de texte favori (comme Emacs ou XEmacs) pour compléter cet environnement. Il existe cependant une version sous Unix similaire à celle proposée pour Windows (donc en mode texte) : c'est RHIDE, qui est disponible à `http://www.tu-chemnitz.de/~sho/rho/rhide-1.4/rhide.html`.

Bibliographie

[Gau 96] M. Gautier, G. Masini et K. Proch. *Cours de programmation par objets — Principes et applications avec Eiffel et C++*. Masson, Paris, 1996.

[Lip 92] S.B. Lippman. *L'essentiel du C++, 2ème édition*. Addison-Wesley, France, 1992.

[Lip 98] S.B. Lippman et J. Lajoie. *C++ Primer, 3rd Edition*. Addison-Wesley, Reading, MA, USA, 1998.

[Mey 91] B. Meyer. *Conception et programmation par objets, Seconde édition*. Informatique Intelligence Artificielle. InterÉditions, Paris, 1991.

[Mus 96] D. R. Musser et A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading, MA, 1996.

[Str 97] B. Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, Reading, MA, USA, 1997.

Remerciements

Je tiens à remercier tout particulièrement Gérald Masini et Karl Tombre qui m'ont aidé dans la rédaction de ce polycopié en me fournissant des documents sur le langage C++, utilisés en partie comme base de travail.