

Table des matières

Bases et vocabulaire	11
<i>À propos de la forme</i>	<i>12</i>
<i>Mise en contexte</i>	<i>13</i>
Qu'est-ce qu'une classe?	15
<i>Un peu de vocabulaire.....</i>	<i>16</i>
Encapsulation – Contrôler l'accès à l'information	18
<i>Membres privés.....</i>	<i>19</i>
<i>Membres publics.....</i>	<i>20</i>
<i>Instances constantes.....</i>	<i>21</i>
<i>Méthodes const.....</i>	<i>22</i>
<i>Paramètres const.....</i>	<i>22</i>
<i>Qualification const et pointeurs.....</i>	<i>23</i>
Exemple concret (petit compte bancaire).....	24
<i>Exercices – Série 00</i>	<i>25</i>
Exemple détaillé : une classe Rectangle	26
<i>Plusieurs niveaux d'opérations.....</i>	<i>27</i>
<i>Considérations techniques</i>	<i>29</i>
<i>Déclaration de la classe Rectangle.....</i>	<i>30</i>
Opérations de base	31
Accesseurs.....	31
Mutateurs.....	33
Attributs.....	34
Attributs publics... Jamais?	36
<i>Définition des méthodes</i>	<i>37</i>
Méthodes et messages	38
Définir les accesseurs	39
Question d'identité et d'appartenance	41
Définir les mutateurs	42
Remarque technique	43
<i>Dans d'autres langages</i>	<i>45</i>
<i>Exercices – Série 01</i>	<i>49</i>

<i>Le principe ouvert/ fermé (Open/ Closed Principle)</i>	53
<i>Pourquoi encapsuler?</i>	54
Possibilité de contrôle d'accès	55
Parenthèse technique – validation des entrées	57
Séparation de l'interface et de l'implémentation	58
Accesseurs de second ordre.....	60
Pourquoi loger, à même l'objet, des accesseurs de second ordre?.....	60
Autres méthodes	62
<i>Encapsulation, mutateurs et validation</i>	64
<i>Objets et contraintes de projet</i>	65
Membres de classe ou membres d'instance	65
<i>Membres d'instance</i>	66
Cas concret : hauteur minimale et hauteur maximale d'un Rectangle.....	67
Utilisation d'un membre de classe	68
Les accesseurs et les mutateurs constituent-ils un bris d'encapsulation?	69
Encapsulation, intégrité et invariants	70
Méthodes de classe.....	75
Référer à soi-même : le mot clé <code>this</code>	77
En résumé.....	78
Réflexions.....	78
<i>Dans d'autres langages</i>	79
<i>Exercices – Série 02</i>	85
Réflexions.....	85
<i>Exercices – Série 03</i>	86
Réflexions.....	86
Le mot-clé <code>auto</code>	87
<i>Répétitives <code>for</code> généralisées</i>	91
Limites du mot clé <code>auto</code>	92
<i>Dans d'autres langages</i>	94
<i>Énumérations en tant que constantes</i>	95
Étymologie d'un membre de classe	96
Un effet secondaire agaçant	97

Utiliser une énumération comme source de constantes entières	98
Les constantes entières et les autres	98
<i>Dans d'autres langages</i>	99
Parenthèse techniques – énumérations fortes	100
Vie d'un objet	101
<i>Le constructeur</i>	102
Constructeur par défaut	103
Constructeurs par défaut et types primitifs	105
Constructeur paramétrique	106
Détail technique : paramètres avec valeur par défaut	109
Constructeur de copie	111
La mécanique de copie	112
Signature d'un constructeur de copie	112
Constructeur de copie et opérateur d'affectation	114
Remarque technique au sujet de l'opérande du constructeur par copie	114
Génération automatique du constructeur de copie	115
Méthodes =default	116
Cacher des constructeurs	117
Suppression d'un service – méthode = delete	118
Copies et classes d'objets immuables	119
Préconstruction	120
<i>Constructeurs de délégation</i>	122
<i>Dans d'autres langages</i>	123
<i>Initialisation immédiate d'attributs d'instance</i>	129
<i>Dans d'autres langages</i>	130
<i>Assertions statiques et robustesse du code</i>	134
<i>Le moteur d'inférence de types</i>	135
Le moteur d'inférence de types en action	136
À quoi cela sert-il?	136
Construction implicite ou construction explicite	137
Nuances fines entre conversion implicite et conversion explicite	139
<i>Dans d'autres langages</i>	140
<i>Destructeur</i>	142

Exemple où le destructeur est essentiel.....	143
<i>Dans d'autres langages</i>	148
<i>Exercices – Série 04</i>	153
<i>Approche OO et code efficace</i>	154
Encapsulation et vitesse	154
Déclarer avant d'utiliser?	157
Constantes d'instance	158
Écriture plus professionnelle.....	159
Instanciation tardive	160
<i>Dans d'autres langages</i>	164
<i>Vie d'un objet et tableaux</i>	166
<i>Vie d'un objet et allocation dynamique de mémoire</i>	167
<i>Dans d'autres langages</i>	168
<i>Exercices – Série 05</i>	171
Objets et opérateurs	172
<i>Modalités générales</i>	175
Fonctions globales ou méthodes?.....	175
Enrichir l'interface de classes existantes.....	178
<i>Planter l'affectation</i>	179
Note sur la Sainte-Trinité	180
Empêcher la copie	181
Un cas douloureux.....	182
L'idiome d'affectation sécuritaire.....	184
<i>Planter les comparaisons</i>	185
Égalité ou équivalence	186
<i>Planter l'ordonnement</i>	188
<i>Planter l'autoincrément et l'autodécrément</i>	189
<i>Opérateur d'écriture sur un flux</i>	191
Nature des opérands	192
Implémentation de l'opérateur	192
Fruits de la généralisation	195
<i>Opérateur de lecture sur un flux</i>	196

Brève parenthèse sur la sérialisation et la persistance.....	198
<i>Opérateurs de conversion et accesseurs</i>	199
<i>Opérateurs et pratiques</i>	200
Une saine maxime	200
Incarnation du mal : la surcharge de l'opérateur & unaire.....	200
Mise en garde : surcharge de && et de 	201
Le cas de la méthode <code>operator bool()</code>	201
Mise en garde : surcharge des opérateurs de conversion	202
<i>Dans d'autres langages</i>	203
<i>Exercices – Série 06</i>	206
<i>Exercices – Série 07</i>	207
Réflexions.....	207
<i>Exercices – Série 08</i>	208
Traitement d'exceptions	209
<i>Pourquoi le traitement d'exceptions?</i>	209
Affichage d'un message d'erreur	210
Ajouter un paramètre pour le code de succès ou d'échec	211
Ajouter un paramètre pour le résultat et retourner un code d'échec	212
Vices du traitement d'erreur traditionnel	212
<i>Utiliser le traitement d'exceptions</i>	213
Reconnaître et lever une exception	214
Gérer une exception	215
La mécanique (en détail).....	216
Lorsque le traitement d'exception est obligatoire : une technique.....	219
<i>Signaler ses intentions en C++</i>	221
La garantie no-throw	221
Spécifications d'exceptions.....	222
Clause <code>noexcept</code> en tant qu'opérateur.....	223
Catch Any et Re-Throw	223
<i>Dans d'autres langages</i>	224
<i>Exercices – Série 09</i>	227
Réflexions.....	227

<i>Exceptions et classes : portrait plus riche</i>	228
<i>L'avantage de classes différentes selon les types d'exceptions</i>	232
<i>Exceptions, constructeurs et destructeurs</i>	234
<i>Exercices – Série 10</i>	236
La classe <code>Joueur</code>	237
L'exécution du programme	238
Raffinements conceptuels et technique	239
<i>Déclarations a priori</i>	239
Rôle d'une déclaration a priori	242
Utilisation de déclarations a priori	243
<i>Dans d'autres langages</i>	246
<i>Simuler un passage par nom</i>	247
Thématiques diverses	250
<i>Sens historique du mot objet</i>	250
<i>Reconnaître la présence d'objets et décrire les relations entre objets</i>	252
<i>Manières de penser un objet</i>	254
Penser subjectivement	255
Choix de stratégie statique	256
Choix de stratégie dynamique	257
Sur la piste des objets intelligents	258
<i>Comparatif d'approches OO pragmatiques</i>	259
La génération et l'exécution des programmes.....	259
Taille sur disque vs taille en mémoire.....	262
Ressemblances et différences syntaxiques	263
Comparatifs sur la vie des objets.....	268
Instanciation et libération	268
Tableaux	269
Gestion de la mémoire.....	270
Comparatif explicite : une classe, trois langages	270
Appendice 00 – Sémantiques directes et indirectes	279
<i>La sémantique directe : types valeurs et types concrets</i>	281
<i>Sémantique indirecte : les adresses</i>	282

Adresses et pointeurs.....	283
Adresses et indirections.....	284
À quoi s’associe l’astérisque?	285
Comprendre l’astérisque	285
Le pointeur nul	286
Pointeurs et arithmétique.....	286
Pointeurs et tableaux	286
Les pointeurs, un concept à bannir?	289
Visualiser les pointeurs	290
<i>Sémantique indirecte : les références</i>	291
Déclarer une référence.....	291
<i>Allocation dynamique de mémoire</i>	292
Les opérations d’allocation et de libération de la mémoire.....	292
Gestion de la mémoire allouée dynamiquement	293
Allocation dynamique et scalaires.....	293
Allocation dynamique et tableaux.....	294
Allocation dynamique de mémoire et erreurs	295
Les aléas de la responsabilité	297
<i>Exemple de type concret : la monnaie</i>	298
<i>Exemple d’allocation dynamique : la pile</i>	302
<i>Exemple d’allocation dynamique : l’arbre binaire</i>	306
Appendice 01 – Penser un modèle OO	311
<i>Un exemple visuel : la fourmi affamée</i>	312
Design par ébauche de code	312
Design par diagrammes	313
<i>Ébauches de classes</i>	315
Classe <code>Position</code>	315
Pour réfléchir.....	315
Classe <code>Bouffe</code>	316
Pour réfléchir.....	316
Représenter l’orientation	317
Pour réfléchir.....	317

Classe Fourmi.....	318
Classe Commande.....	319
Pour réfléchir.....	319
Classe Menu	320
Pour réfléchir.....	320
Classe Jeu.....	321
<i>Petites fautes de design.....</i>	322
Corriger le problème de la convivialité d'une Commande	323
Solutions alternatives	323
Corriger le problème structurel de la répétitive principale	324
Pour réfléchir.....	324
Encapsuler l'affichage de la bouffe.....	325
Ajuster l'affichage du jeu.....	325
Encapsuler l'affichage de la fourmi	326
Pour réfléchir.....	326
<i>La classe manquante : l'espace de jeu</i>	327
Ajuster l'affichage de l'espace de jeu	328
Pour réfléchir.....	328
Valider les déplacements et les positions dans l'espace de jeu.....	329
<i>Remarques d'ensemble.....</i>	331
À propos du design initial	331
À propos de la responsabilité opératoire	332
À propos de la conception des classes	333
À propos de la comparaison d'objets concrets.....	334
À propos de l'introduction d'une valeur dans un système	335
Remarques diverses quant au développement.....	337
Appendice 02 – Nommer et documenter	338
<i>Généralités</i>	339
<i>Groupement et nomenclature</i>	340
Groupement par modules	340
Groupement par classes.....	340
Exemple concret : la classe Integer de Java.....	341

<i>Noms significatifs de constantes</i>	342
Noms historiques ou associés à un objet courant	342
Dénombrement, bornes et énumération	343
<i>Noms significatifs de variables</i>	344
Composition de noms.....	345
Les noms et les verbes.....	345
Variables jetables	346
Clarté ou concision?	346
Préfixes, suffixes et autres décorations	346
<i>La notation hongroise</i>	347
Bref historique.....	348
Problèmes de décision de nomenclature	349
Aperçu de la notation	350
Raison d’être et avantages.....	351
Le cas particulier des interfaces personne/ machine	352
Inconvénients	353
Pour réfléchir.....	355
<i>Noms significatifs d’attributs</i>	356
<i>Noms significatifs de sous-programmes globaux</i>	357
<i>Noms significatifs de méthodes</i>	358
<i>Noms significatifs de types et de classes</i>	359
<i>À propos des commentaires</i>	359
<i>Lectures complémentaires</i>	361
Appendice 03 – Dans l’objet ou hors de l’objet?	362
Annexe 04 – Sémantique des constructeurs	365
Annexe 00 – Résumé de la notation UML abordée dans ce document	367
Annexe 01 – Discussions sur quelques réflexions choisies	368
<i>Réflexion 00.0 : tout est-il objet?</i>	368
<i>Réflexion 00.1 : Machiavel ou Murphy?</i>	369
<i>Réflexion 00.2 : quelles méthodes exposer?</i>	370
<i>Réflexion 00.3 : des correctifs silencieux?</i>	372
<i>Réflexion 00.4 : choix de représentation</i>	373

<i>Réflexion 00.5 : clients hostiles</i>	374
<i>Réflexion 00.6 : une saine gamme de constructeurs</i>	374
<i>Réflexion 00.7 : erreurs à la construction</i>	376
Annexe 03 – Concepts et pratiques du langage C++	377
<i>Idiome RAIT</i>	377
<i>ODR – la One Definition Rule</i>	378
<i>Sainte-Trinité</i>	381
Annexe 04 – Principe de localité et efficacité	382
<i>Remarques préliminaires</i>	383
<i>Primitifs</i>	383
<i>Objets</i>	383
Réponse générale.....	384
Variations selon les types de constructeurs.....	384
Implémentations par défaut.....	385
Individus	386
Références	387

Bases et vocabulaire

Rôle de ce volume

Nous verrons ici les concepts tout à fait à la base de l'approche objet : du vocabulaire, l'idée de ce qu'est un objet, de ce que des objets de même famille ont en commun, comment les objets prennent forme et existent de leur construction à leur destruction, *etc.* Tout ça en mettant l'accent sur le premier principe fondamental de l'approche objet : l'encapsulation.

Ce document couvrira quelque chose de fort important : ce qu'est, formellement, une **classe**, et ce que sont des **instances** d'une classe. C'est sur les thèmes de ce document que repose l'essence de la programmation orientée objet (POO). Nous utiliserons OO pour « orienté objet » ou « orientée objet » dans ce volume, de manière générale.

On y verra :

- comment **définir une classe**, au sens du langage de programmation utilisé comme au sens de la réflexion à apporter pour bien la définir en fonction d'un ensemble de besoins – quoique, pour le volet réflexion, on ne pourra faire mieux pour le moment qu'amorcer le processus, manquant un peu de perspective face au sujet;
- comment **instancier une classe**. Une classe étant analogue à un type de données, on voudra en obtenir des instances¹, analogues à des variables ou à des constantes. Ceci nous amènera à discuter de la vie d'un objet, de sa construction à sa destruction, et des questions techniques propres à ce sujet;
- ce que sont les **membres (attributs et méthodes) d'un objet**, qu'il s'agisse de membres d'instance ou de membres de classe;
- ce qu'est le **principe d'encapsulation**, et comment il est appliqué dans le langage de programmation utilisé;
- les **catégories de méthodes** les plus communes; et
- ce qu'est la **surcharge d'opérateurs**.

Nous viserons un double objectif :

- d'un côté, le **développement d'aptitudes techniques de base en POO**, incluant la reconnaissance et la compréhension de certains éléments essentiels, et la capacité de rédiger et d'utiliser soi-même des programmes suivant une approche OO; et
- de l'autre, le **développement d'aptitudes de gestion de développement informatique selon un modèle OO**, incluant la reconnaissance d'attitudes valables et productives dans un contexte de développement de projet informatique, et la compréhension des raisons pour lesquelles ces attitudes sont à favoriser.

Idéalement, à la suite de ce tronçon, l'étudiant(e) saura, si le besoin s'en fait sentir, réaliser lui- ou elle-même certaines tâches simples de POO, et (surtout) assister et encadrer des spécialistes dans la réalisation de leurs propres tâches.

Ce document s'inscrit dans une série d'une dizaine de documents de taille semblable. Il est clair que nous ne ferons ici qu'effleurer la surface d'un sujet au cœur de l'informatique contemporaine.

¹ Ce qu'on nomme communément, et un peu par excès de langage, des **objets**.

À propos de la forme

Ce document étant une première approche, il contient de remarques qui mériteront d'être raffinées et des pratiques sur lesquelles nous reviendrons ultérieurement. Prenez donc soin de garder l'esprit ouvert puisque ce qui convient pour démarrer un processus de réflexion peut ne pas convenir à une pensée plus avancée sur un même sujet.

Vous trouverez à divers endroits des petits encadrés indiquant *Réflexion 00.n* (pour *n* entier positif²). Ces encadrés soulèveront des questions qui méritent une discussion plus approfondie et pour lesquelles les réponses peuvent surprendre ou ne pas être aussi banales qu'il y paraît. Des ébauches de réponses seront proposées pour chacune dans *Annexe 01 – Discussions sur quelques réflexions* choisies.

Puisque ces notes se veulent un appui à l'apprentissage de la POO, pas d'un langage particulier, mais puisqu'il faut aussi utiliser (au moins) un langage pour appuyer formellement nos idées et notre discours, le cœur de ce document utilise un langage OO, C++, pour ses exemples. Cependant, vous trouverez à certains endroits dans le document des sections intitulées *Dans d'autres langages* qui exploreront les ramifications des sections précédentes en Java, C# et VB.NET, ce qui vous permettra de faire le pont avec d'autres technologies. Ces pages ont des bordures différentes des autres, et vous pourrez les omettre si ces nuances ne sont pas au cœur de vos préoccupations.

Enfin, notez qu'au moment d'écrire ces lignes, la norme la plus récente du langage C++ est C++ 14, un ajustement à l'énorme mise à jour que fut C++ 11, le vote pour officialiser C++ 17 (la prochaine mise à jour significative du langage) est en cours, et les travaux sur ce qui devrait être C++ 20 vont bon train. Les volumes plus poussés de cette série de notes de cours couvrent des aspects des plus récentes versions de C++, et montrant parfois comment il est possible d'en arriver au même résultat à partir de versions antérieures du langage, ou en indiquant les raisons qui motivent certaines adaptations de la norme.

² Oui, zéro est positif.

Mise en contexte

Notre monde est rempli d'objets. Supposons l'énoncé de problème suivant.

Veillez réaliser une application permettant de :

- demander à l'utilisateur ce qu'il désire faire. Les choix disponibles sont dessiner un carré, dessiner un rectangle, dessiner un triangle et quitter;
- si l'utilisateur choisit un carré, on devra lui demander d'entrer sa hauteur, qui devra être validée pour être entre 1 et 20 inclusivement;
- si l'utilisateur choisit un rectangle, on devra lui demander d'entrer sa hauteur, qui devra être validée pour être entre 1 et 20 inclusivement, et sa largeur, qui devra être validée pour être entre 1 et 50 inclusivement;
- si l'utilisateur choisit un triangle, on devra lui demander d'entrer sa hauteur, qui devra être validée pour être entre 3 et 19 inclusivement et qui devra être impaire;
- par la suite, la forme choisie sera dessinée, on attendra qu'une touche soit pressée, et on présentera à nouveau le menu. Cela sera fait tant et aussi longtemps que l'utilisateur n'aura pas choisi de quitter.

Prenons le problème sous un angle objet, et examinons-en les constituants. Au strict minimum, on aura besoin de :

Reconnaître les objets

- déterminer une représentation pour un rectangle;
- déterminer une représentation pour un carré; et
- déterminer une représentation pour un triangle.

On remarquera aussi quelques relations amusantes entre ces différentes catégories de formes. Entre autres :

Similitudes structurelles

- *ce sont toutes des formes*, qui ont sur le plan opérationnel et structurel des points en commun, ce dont nous tirerons profit plus tard; et
- on a un carré qui est, du moins en mathématiques, un cas particulier de rectangle³, ce dont nous tirerons aussi profit plus tard.

On pourrait aller beaucoup plus loin que cela, mais arrêtons-nous ici pour le moment. Pour l'instant, nous allons viser :

- en premier lieu, la rédaction d'une classe `Rectangle` qui sera telle qu'on pourra utiliser chaque instance de `Rectangle` aussi simplement qu'on utiliserait un `int`⁴;
- en second lieu, la rédaction d'une classe `Triangle`, avec les mêmes critères d'aisance à l'utilisation que ceux énoncés ci-dessus; et enfin
- la rédaction d'une classe `Point`, toujours selon les mêmes critères.

Nous réserverons pour plus tard la rédaction de la classe `Carre`.

³ Ceci est vrai sur le plan structurel, mais pas vraiment sur le plan opératoire; les mathématiques et l'informatique ne concordent pas ici. Nous discuterons de cette nuance dans [POOv01].

⁴ Cet objectif dénote une vision de l'approche objet; voir *Appendice 00 – Sémantiques directes et indirectes*.

Supposons maintenant l'énoncé de problème suivant.

Votre firme prend le virage Internet et veut implanter un système de transaction B2B (*Business to Business*). On aimerait avoir la possibilité de permettre à un programme assurant le lien avec Internet de consulter (sans pouvoir y apporter de modifications) le registre des transactions passées où intervenait un client donné.

Le volet qui revient à votre équipe est celui de la *représentation des factures*; vous serez en contact avec le système de gestion de bases de données, sans y avoir directement accès. Étant donné un numéro de client, vous devez offrir au minimum, et moyennant une forme d'authentification (disons un mot de passe), l'accès :

- au numéro d'identification de la facture;
- au montant de la facture;
- à la date à laquelle la facture fut émise. Dans votre système, toute facture considérée *devra* avoir été émise, et par conséquent avoir été émise à une certaine date);
- à la date à laquelle la facture fut payée. Il faudra réfléchir à une stratégie pour représenter les cas où la facture n'a pas encore été payée;
- à la nature de ce qui fut facturé (pour le moment, un vulgaire texte explicatif);
- aux intérêts cumulés pour retard de paiement;
- au numéro d'identification de la facture précédente pour ce client; et
- au numéro d'identification de la facture suivante, aussi pour ce client. Il faudra d'ailleurs peut-être réfléchir à une stratégie pour représenter les numéros de facture représentant l'absence de facture.

Vous savez qu'il y aura assurément des ajouts à ce système (entre autres, la nature de ce qui fut facturé risque de devenir quelque chose de plus riche qu'une chaîne de caractères), mais cela ne vous empêche pas de commencer le travail...

Certains objets, dans cet énoncé, sont presque incontournables :

- la facture, en soi, peut être vue comme un objet. Chaque facture aura un certain nombre d'attributs et de méthodes, décrites en partie par l'énoncé, et toutes les factures auront une structure similaire;
- en fonction des besoins et des contraintes, on aura peut-être aussi besoin d'objets pour représenter des dates. Ce type d'objet, cette *classe*, existe peut-être déjà dans une bibliothèque commerciale ou *maison*; à la limite, il sera possible de développer une classe *Date* à l'interne pour nos besoins propres;
- bien que la nature de ce qui fut facturé soit *pour le moment* du texte, il serait prévoyant de penser cet attribut d'une facture sous un angle objet. De la description qu'on nous en fait, la forme de cet attribut est, pour l'instant, assez volatile et sujette à changements imprévisibles.
- il nous faut savoir à quelle entité demander, étant donné une facture, quelle est celle qui la précède et quelle est celle qui la suit (un autre objet, responsable d'organiser les factures?). Les objets peuvent collaborer entre eux, après tout;
- toute facture doit pouvoir nous indiquer les intérêts cumulés, qu'elle les calcule elle-même ou qu'elle les connaisse au préalable... à moins qu'une facture ne collabore avec d'autres classes pour y arriver...
- et on pourrait continuer longtemps.

Qu'est-ce qu'une classe?

⇒ Une **classe** est un *modèle*, un *archétype*. C'est l'abstraction qu'on vise à actualiser, à réifier, donc que l'on souhaite *instancier*.

La définition d'une classe dépend de l'utilisation qu'on compte en faire, et est par conséquent guidée par un souci politique, technique ou philosophique.

Exemple : si on observe tous les clients humains d'une entreprise dans le but d'identifier ce qui fait que chacun est un client et pas autre chose, on obtiendra une liste de ce qui est commun à tous les clients.

À tout hasard :

- un nom, un prénom, un numéro de téléphone et une adresse postale, sans doute; puis
- un solde à payer;
- un numéro d'identification unique à chacun, probablement;
- une représentation des préférences d'achat de l'individu, peut-être; *etc.*

Ce faisant, on détermine ce qu'est un client sans jamais parler de l'un ou l'autre d'entre eux. *On spécifie de manière abstraite ce qu'est un client en listant, essentiellement, ce que tous les clients ont comme points en commun.*

On le fait surtout dans un but opérationnel. Si on veut représenter des clients, dans le cas sous étude, c'est sûrement dans un but commercial, par exemple pour mieux cibler nos interventions publicitaires ou pour faire un suivi des comptes en souffrance. *Comme pour la production d'un algorithme, on se laissera guider dans la création d'une classe par un problème (ou un ensemble de problèmes) à résoudre.*

De même, comme dans le cas d'un algorithme, où il y a souvent plusieurs solutions possibles à un même problème, il y aura souvent plusieurs manières possibles de définir les classes dont on aura besoin, et la manière dont elles seront réalisées. La connaissance du problème à résoudre, et des problèmes envisageables dans le futur, aidera à reconnaître, parmi les solutions possibles, celles qui seront à privilégier.

Un problème différent mènera parfois à la conception d'objets différents. Une bonne solution aura donc souvent le mérite d'être applicable à un large éventail de problèmes, d'être réutilisable dans plusieurs contextes tout en demeurant efficace.

Les sections *Reconnaître la présence d'objets et décrire les relations entre objets* et *Manières de penser un objet*, plus loin, donnent des pistes en ce sens.

Il est probable que vos premières définitions de classes pour résoudre un problème soient moins élégantes que ne le seront celles que vous ferez lorsque vous aurez un peu plus d'expérience. Ne vous en faites pas trop : la pratique est votre meilleure alliée, ici comme ailleurs.

Un peu de vocabulaire

Avant d'aller plus loin, quelques mots de vocabulaire :

- une **classe** est un type de données au potentiel très riche, regroupant des états (des données) et les opérations qui s'y appliquent;
- une **instance** est un cas particulier d'une classe (c'est souvent ce qu'on nomme un *objet*, quoique cela s'avère un léger abus de langage);
- les données qu'on retrouve dans une classe sont ses **attributs** (on dira parfois qu'il s'agit de ses *champs*, par analogie avec les bases de données, ou de ses *propriétés*, quoique ce terme ait une signification spécialisée dans certains cercles);
- les opérations qu'on retrouve dans une classe sont ses **méthodes**, ses *services*;
- les attributs et les méthodes d'une classe sont ses **membres**.

Les mots *déclaration* et *définition* apparaîtront à plusieurs reprises dans ce document et dans les suivants.

Une **déclaration** introduit un nom symbolique dans un programme, alors qu'une **définition** est le lieu où une entité commence à exister (souvent, il s'agit du lieu où le compilateur doit attribuer de l'espace pour une entité).

En C++, il existe une règle de base nommée la règle ODR (pour *One Definition Rule*) qui dit qu'une entité d'un programme ne peut être *définie* qu'une fois. Voir **ODR – la One Definition Rule** pour des détails.

Notez qu'il est possible – et parfois même pertinent – de concevoir des classes sans états, des classes sans comportements. Les classes sans comportements vous sont peut-être familières : elles correspondent, grosso modo, à des enregistrements dans des langages de programmation structurée (les `struct` du langage C, par exemple, ou les `record` du langage Pascal). Les classes sans états sont souvent – mais pas toujours – associées à des contrats, des interfaces, un concept plus sophistiqué que nous couvrirons dans [POOv01].

Il arrive même que l'on conçoive des classes complètement vides, qui représentent des concepts purs de par leur simple existence. Et croyez-le ou non, mêmes ces classes archi simples sont extrêmement utiles! Mais nous y reviendrons dans les volumes ultérieurs de cette série...

Si on veut lier ces mots de vocabulaire à du concret, on peut prendre exemple sur le tableau suivant (la notation graphique respecte la norme UML 2.0).

Exemple d'un client (ci-dessus)	Exemple d'un triangle																																
<p>La classe est Client.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2" style="text-align: center;">Client</th> </tr> </thead> <tbody> <tr> <td>- Nom:</td> <td>texte</td> </tr> <tr> <td>- Prénom:</td> <td>texte</td> </tr> <tr> <td>- Adresse:</td> <td>texte</td> </tr> <tr> <td>- NoTél:</td> <td>texte</td> </tr> <tr> <td>- SoldeImpayé:</td> <td>réel</td> </tr> </tbody> </table> <p>Une instance de Client serait le client <i>c</i> nommé Roger Tromblon vivant au 393 Côte St-Louis Est, Blainville avec un solde impayé de 147,73\$.</p> <p>Le prénom de <i>c</i> (de valeur "Roger") est l'un de ses attributs.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2" style="text-align: center;">c : Client</th> </tr> </thead> <tbody> <tr> <td>- Nom =</td> <td>"Tromblon"</td> </tr> <tr> <td>- Prénom =</td> <td>"Roger"</td> </tr> <tr> <td>- Adresse =</td> <td>393 Côte ...</td> </tr> <tr> <td>- NoTél =</td> <td>(XXX) XXX-XXXX</td> </tr> <tr> <td>- SoldeImpayé =</td> <td>147,73</td> </tr> </tbody> </table> <p>Si je veux connaître le solde impayé de Roger Tromblon, je devrai, si la classe Client est bien conçue, interroger l'instance <i>c</i> par l'une de ses méthodes. <i>Je ne devrais pas avoir directement accès à son solde impayé.</i></p>	Client		- Nom:	texte	- Prénom:	texte	- Adresse:	texte	- NoTél:	texte	- SoldeImpayé:	réel	c : Client		- Nom =	"Tromblon"	- Prénom =	"Roger"	- Adresse =	393 Côte ...	- NoTél =	(XXX) XXX-XXXX	- SoldeImpayé =	147,73	<p>La classe est Triangle.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2" style="text-align: center;">Triangle</th> </tr> </thead> <tbody> <tr> <td>- Hauteur:</td> <td>entier</td> </tr> </tbody> </table> <p>Une instance de Triangle serait le triangle <i>t</i> de hauteur 13.</p> <p>La hauteur de <i>t</i> (de valeur 13) est l'un de ses attributs.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2" style="text-align: center;">t : Triangle</th> </tr> </thead> <tbody> <tr> <td>- Hauteur =</td> <td>13</td> </tr> </tbody> </table> <p>Si je veux que le triangle <i>t</i> se dessine, <i>je devrais lui demander de se dessiner</i>. C'est le travail de chaque Triangle, donc <i>de chaque instance de la classe Triangle</i>, de bien se dessiner.</p>	Triangle		- Hauteur:	entier	t : Triangle		- Hauteur =	13
Client																																	
- Nom:	texte																																
- Prénom:	texte																																
- Adresse:	texte																																
- NoTél:	texte																																
- SoldeImpayé:	réel																																
c : Client																																	
- Nom =	"Tromblon"																																
- Prénom =	"Roger"																																
- Adresse =	393 Côte ...																																
- NoTél =	(XXX) XXX-XXXX																																
- SoldeImpayé =	147,73																																
Triangle																																	
- Hauteur:	entier																																
t : Triangle																																	
- Hauteur =	13																																

Réflexion 00.0 – Dans la classe **Client**, y aurait-il lieu d'utiliser une classe distincte pour représenter une adresse? Un numéro de téléphone? Expliquez votre position (la réponse est subtile; voir **Réflexion 00.0 : tout est-il objet?**).

Encapsulation – Contrôler l'accès à l'information

L'un des principaux principes de la POO est le **principe d'encapsulation**. La manière la plus appropriée de définir l'encapsulation est sans doute celle-ci : *principe selon lequel un objet est responsable de son intégrité, du début à la fin de sa vie*. Une écriture plus technique de ce principe serait : *un objet est responsable du maintien de ses invariants* (voir **Encapsulation, intégrité et invariants** pour plus de détails).

Pour respecter ce principe, au strict minimum, l'accès aux états d'un objet devrait se faire le plus possible à travers des canaux contrôlés, comme par exemple des *accesseurs* et, plus rarement, des *mutateurs*. Ces deux modes d'accès sont définis plus bas.

Les termes anglais *Encapsulation, Data Hiding et Information Hiding* ont tous un sens analogue à celui que nous donnerons au mot encapsulation.

Notez que selon les sources, les langages et les écoles de pensée, il peut y avoir des variations sur la portée technique de ce terme

Cela dit, la portée du principe d'encapsulation va bien plus loin que cela, comme nous le verrons dans ce volume et dans les volumes suivants.

Bien que le principe d'encapsulation soit des plus importants en POO, il s'agit d'un principe qui peut être appliqué et respecté dans la plupart des langages de programmation, OO ou non.

Appliquer l'encapsulation

Dans un langage n'offrant pas le niveau de support à l'encapsulation attendu des langages OO, cependant, le respect de l'encapsulation repose habituellement sur la discipline des informaticien(ne)s.

Les langages OO ont ceci de particulier qu'ils permettent d'appliquer le principe d'encapsulation de manière stricte et explicite à l'aide de quelques mots clés. Les compilateurs et les interpréteurs y servent entre autres à renforcer la discipline des membres de l'équipe de développement en ajoutant des règles strictes qui *imposent* l'encapsulation aux conventions qui, elles, ne font que la suggérer.

En C++, ces mots clés sont **private**, **protected**, **public**, **friend** (bien que certains, à tort, le contesteraient) et **const**, ce dernier lorsqu'il est appliqué à des méthodes ou à des paramètres.

La section **Encapsulation et regroupement** de [POOv01] décrit quelques-unes des subtilités terminologiques de ces mots avec Java et les langages .NET.

La plupart des autres langages OO commerciaux utilisent `private`, `protected` et `public` à des fins identiques ou très similaires à celles qui seront décrites ici, mais la plupart n'offrent pas un support à l'encapsulation suffisant pour permettre des objets constants.

Note : nous éviterons le mot `protected` jusqu'à ce que nous débutions notre discussion sur le concept d'héritage. De même, comprendre en quoi le mot `friend`, bien appliqué, accroît l'encapsulation plutôt que de la réduire, nous demandera au préalable de progresser un peu dans notre cheminement.

Nous couvrirons les détails syntaxiques propres à chacun des mots clés lorsque nous en présenterons des exemples d'utilisation.

Membres privés

La spécification **private** s’appliquera à tout ce qui est destiné à usage interne dans la classe. On l’appliquera souvent aux attributs, mais il est sain de considérer que `private` soit la meilleure qualification par défaut – moins un objet révélera de détails sur lui-même et plus libre nous serons de le modifier à loisir.

La notation appliquée dans les schémas UML demande qu’on précède un membre privé du signe `-` (*moins*).

En fait, *si on veut respecter le principe d’encapsulation, on devrait au minimum qualifier de privés tous les attributs*. En effet, si un attribut est privé, alors tous les accès qui y sont faits doivent être réalisés par l’objet qui en est propriétaire⁵.

Il y a des exceptions à cette règle; voir *Attributs publics... Jamais?* pour plus d’information.

Ce contrôle absolu sur l’accès aux attributs permet à l’objet d’offrir des promesses d’intégrité qu’il est en mesure de garantir. Si d’autres que lui pouvaient modifier ses attributs, ses états, sans qu’il n’en soit conscient, il deviendrait impossible pour lui d’offrir quelque garantie d’intégrité que ce soit. Il serait impossible pour lui d’appliquer pleinement le principe d’encapsulation.

On aura aussi parfois – et de plus en plus souvent alors qu’on devient expérimenté – des méthodes privées, par exemple lorsqu’une classe permettra des opérations pour usage interne mais qui seraient dangereuses si elles étaient disponibles à tous.

Exemple : vous trouverez à droite une illustration C++ de la syntaxe d’une classe `Priv` dont l’attribut `attr_`, de type `int`, et dont la méthode `f()`, retournant un `float`, sont privés. En C++, la qualification `private` est implicite *dans les classes* (c’est elle qui s’applique par défaut).

L’équivalent UML 2.0 est présenté ci-dessous.

Priv
- attr_ : int
- f() : float

```
class Priv {
    private:
        int attr_;
        float f() { return attr_ + 0.5f; }
};
// ou (équivalent)
class Priv {
    int attr_;
    float f() { return attr_ + 0.5f; }
};
```

⁵ Ou par ses amis, mais ceci ne change en rien l’essence du propos.

Membres publics

La spécification **public** s'appliquera à tout ce qui est destiné à usage externe dans la classe. Normalement, on l'appliquera seulement aux méthodes qui permettront aux autres éléments d'un programme (au *code client*) de manipuler l'objet.

La notation appliquée dans les schémas UML demande qu'on précède un membre public du signe + (*plus*).

Ce qui est public dans un objet constitue son interface, sa barrière d'abstraction. La façade publique d'un objet est ce qu'il dévoile au monde, ce qu'il accepte de révéler et de supporter. Les autres objets d'un système compteront sur la présence du volet public d'un objet pour transiger avec lui.

Exposer une méthode est, pour l'objet qui le fait, un engagement solennel : comptez sur cette méthode, elle demeurera disponible, opérationnelle. Vous pourrez vous en servir. En effet, si les objets d'un système utilisent une méthode publique d'un objet donné, et si cet objet modifie la signature de cette méthode (ou la supprime!), alors tous ses clients seront dans le pétrin. Un objet est responsable l'entretien de son visage public.

Compte tenu du poids de cet engagement, on souhaitera ne révéler de manière publique que ce qu'un objet considère important de révéler. Si le visage public d'un objet ratisse trop large, cet objet se transformera en véritable cauchemar d'entretien.

Ceci explique la suggestion (à la section précédente) de qualifier privés les membres jusqu'à ce qu'il y ait un réel besoin de procéder autrement.

Exemple : vous trouverez à droite une illustration C++ de la syntaxe d'une classe `Pub` dont l'attribut `attr_`, de type `int`, et dont la méthode `f()`, retournant un `float`, sont publics.

L'équivalent UML 2.0 est présenté ci-dessous.

Pub
+ attr_ : int
+ f() : float

Exemple : vous trouverez à droite une illustration C++ de la syntaxe d'une classe `Mixte` dont l'attribut `attr_`, de type `int`, est privé et dont la méthode `f()`, retournant un `float`, est publique.

L'équivalent UML 2.0 est présenté ci-dessous.

Mixte
- attr_ : int
+ f() : float

```
class Pub {
public:
    int attr_; // attribut public
    float f() { return attr_ + 0.5f; }
};
```

```
class Mixte {
    int attr_; // attribut privé
public:
    float f() { return attr_ + 0.5f; }
};
```

Instances constantes

La qualification `const`, lorsqu'elle est appliquée à une instance, indique que cette instance ne peut en aucun temps modifier ses attributs. Tenter une opération ne garantissant pas la constance de l'instance est une erreur, et est rapporté comme tel à la compilation.

Il est important de relever que le début et la fin de la vie d'une instance (sa construction et sa destruction – voir **Le constructeur** et **Destructeur**, plus bas) sont des moments où l'état de cette instance doit changer, ce qui est évident.

La spécification `const` sur une instance entre en vigueur dès que cette instance est construite, et se maintient jusqu'au début de sa destruction. On peut donc, en général, utiliser une instance constante, mais pour fins de consultation seulement. Ceci nous amène à l'idée de *méthodes const*.

*Limites à la
spécification
const sur un
objet*

La possibilité de traiter tous les types sur un pied d'égalité au niveau des possibilités est une particularité des langages OO.

Certains y arrivent en n'offrant aucun type primitif (Smalltalk en est un exemple), d'autres y arrivent en offrant à toutes fins pratiques le même niveau de support aux types primitifs et aux classes (C++), certains essaient mais s'arrêtent à mi-chemin (C#) et d'autres n'essaient même pas, gardant une distinction nette entre types primitifs et classes (Java) ou le simulent en partie par une technique souvent nommée *Boxing* (C#, Java 1.5+) par laquelle des conversions implicites sont réalisées au besoin entre type primitif et objet et (inversement).

C++ est l'un des rares langages OO commerciaux qui permet à un objet d'être véritablement constant. C'est, sur le plan de l'homogénéité de son modèle OO, l'une de ses plus grandes qualités.

Effet appréciable des instances constantes : elles doivent absolument être initialisées dès leur déclaration, ce qui impose aux programmeuses et aux programmeurs de saines pratiques, et mène à du code généralement plus rapide à l'exécution.

Méthodes `const`

La qualification `const`, lorsqu'elle est appliquée à une méthode, indique que l'action de celle-ci ne modifie en rien l'objet auquel elle appartient.

Ceci permet de créer des instances `const`, donc qui ne peuvent changer d'état une fois déclarées, et de restreindre par la suite les opérations possibles sur celles-ci aux seules opérations garanties comme étant sans le moindre risque d'altération sur l'objet auquel elles appartiennent.

Une méthode `const` offre une garantie au compilateur : aucune de ses opérations ne peut modifier l'instance à laquelle cette méthode appartient. Le compilateur peut détecter les violations de cette garantie et prévenir les actions qui y contreviendraient.

Une méthode `const` peut utiliser directement des attributs, mais de manière consultative (à droite d'une opération d'affectation, par exemple). De même, une méthode `const` peut utiliser d'autres méthodes du même objet, mais seulement si celles-ci sont aussi `const`. Les garanties de constance sont transitives à l'intérieur d'un même objet.

Note : si une méthode *peut* être spécifiée `const`, donc si elle peut garantir ne pas modifier son instance propriétaire, alors elle *devrait* généralement l'être.

Note : une distinction importante doit être faite entre les concepts de `const`, qui décrit une immuabilité contextuelle, et d'immuable, qui décrit une immuabilité en tout temps.

Permettre de spécifier qu'une méthode garantit l'intégrité d'un objet est nécessaire pour permettre la mise en place de véritables instances constantes.

Il est possible d'obtenir des classes dont toutes les instances sont constantes par une technique d'immuabilité – concevoir une classe dont les instances, une fois construites, n'offrent aucune méthode permettant de les modifier. C'est toutefois une constance sur la base d'une catégorie entière d'objets, pas sur une base individuelle.

En Java et en C#, l'immuabilité une classe à la fois est la seule manière d'offrir des objets constants. Notons que C# offre des mécanismes simulant la constance sur la base méthode à l'aide de propriétés en lecture seule (voir plus loin pour des informations sur les propriétés).

Paramètres `const`

La spécification `const`, lorsqu'elle est appliquée à un paramètre, empêche le sous-programme recevant ce paramètre de le modifier de quelque manière que ce soit.

En particulier, si le paramètre est un objet, seules ses méthodes spécifiées `const` pourront alors être appelées de manière légale.

Utiliser `const` partout où cela s'avère possible est une excellente pratique, un excellent *comportement par défaut*, au sens de ces réflexes qu'il est sain de développer.

Pour plus d'information à ce sujet, je vous suggère [SutterConst], qui est un peu corsé étant donné nos connaissances du modèle OO pour le moment.

Note : l'emploi par un sous-programme de paramètres constants est une garantie de sécurité offerte aux sous-programmes appelants, particulièrement si ce paramètre est passé par référence; dans le cas d'un passage par copie, la spécification `const` ou non est, dans la plupart des cas, question de style.

Qualification `const` et pointeurs

La qualification `const` mêlée aux pointeurs, tout en étant une abstraction fort utile, est source fréquente de maux de tête du fait qu'un pointeur représente (indirectement) deux choses : le lieu d'une donnée et, par indirection, la donnée en question.

Sachant cela, le tableau suivant⁶ résume le sens du mot clé `const` en fonction de sa position dans un type de donnée (le type `int` est utilisé à titre d'illustration).

Notation	Le pointeur est	La donnée pointée est
<code>int *p;</code>	Modifiable	Modifiable
<code>int *const p;</code>	Constant	Modifiable
<code>const int *p;</code>	Modifiable	Constante
<code>int const *p;</code>	Modifiable	Constante
<code>const int * const p;</code>	Constant	Constante

Les versions les plus fréquemment rencontrées sont celles des troisième et quatrième lignes (donnée non modifiable, pointeur amovible). Pour bien lire ces déclarations, mieux vaut procéder selon la grammaire anglaise et prendre les éléments de droite à gauche :

- la première ligne se lit *pointer to an int*;
- la deuxième ligne se lit *constant pointer to an int*;
- la troisième ligne se lit mal (une critique récurrente de ceux qui prêchent pour l'écriture – équivalente – de la quatrième ligne);
- la quatrième ligne se lit *pointer to a constant int*; et
- la cinquième ligne se lit *constant pointer to a constant int*.

Réflexion 00.1 – Jusqu'où devrait-on pousser la quête de la protection de l'intégrité des objets? (voir *Réflexion 00.1 : Machiavel ou Murphy?*).

⁶ L'idée derrière ce tableau a été prise du site http://photon.poly.edu/~hbr/cs903-F00/lib_design/notes/advanced.html

Exemple concret (petit compte bancaire)

Présumons l’existence d’une classe `CompteBancaire`. Chaque instance de cette classe aura entre autres les membres suivants :

- un solde (réel qu’on voudra positif);
- une méthode pour connaître le solde;
- une méthode pour modifier le solde;
- une méthode pour effectuer un dépôt; et
- une méthode pour effectuer un retrait.

Le *solde*, qui est un attribut, sera **privé**. S’il fallait qu’il soit public, n’importe qui pourrait avoir un accès direct au solde du compte et le modifier, ce qui serait clairement quelque chose d’inacceptable.

La méthode pour *connaître le solde* sera **publique**. Ceci ne veut pas dire qu’on ne puisse pas protéger l’accès au solde par mot de passe ou par numéro d’identification personnel (NIP), ou que l’objet sera dans l’obligation de dévoiler l’information demandée; cela signifie simplement qu’il est possible au code client de demander le solde.

Demander à un compte bancaire de révéler son solde ne devrait pas entraîner une modification de ce compte bancaire⁷; cette méthode devrait donc être **const**.

Il est raisonnable de qualifier la méthode pour *modifier le solde* de méthode **privée**, et vouée à l’usage interne, dans l’optique où les transactions d’un éventuel client devraient se limiter à des retraits ou à des dépôts

CompteBancaire	
-	solde: réel
<hr/>	
+	connaître_solde() const : réel
-	modifier_solde(nouveau_solde : réel) : rien
+	déposer(Montant : réel) : rien
+	retirer(Montant : réel) : rien

Les méthodes pour *effectuer un retrait* et pour *effectuer un dépôt* devraient toutes deux être **publiques**. On s’attend à ce qu’à l’interne, ces méthodes sollicitent l’aide de celle qui modifie le solde et, peut-être, de celle qui permet de connaître le solde (pour vérifier que les fonds sont suffisants avant d’autoriser un retrait, par exemple).

Note sur la notation UML

Vous remarquerez la division d’un diagramme de classe UML en trois parties distinctes. Celle du haut sert à indiquer le nom de la classe, celle du centre liste ses attributs, et celle qui se trouve en bas liste les méthodes.

Philosophiquement, quand on utilise un objet, on souhaite ne pas avoir à connaître la partie du centre. Celle-ci devrait être encapsulée par les deux autres.

NomDeLaClasse	
-	attribut0: type0
-	attribut1: type1
<hr/>	
+	méthode0(param0 : type0, param1 : type1) : type
+	méthode1() : type
+	méthode2(param0 : type0) : type

⁷ ... dans ce modèle simplifié, du moins.

Exercices – Série 00

Allons-y de quelques exercices de reconnaissance d'objets. Essayez d'identifier les classes, les instances, les méthodes et les attributs... Attention : avec des exemples non formels, et même avec des exemples formels mais examinés à partir de points de vue différents, il est possible que plusieurs personnes arrivent à des découpages logiques qui soient à la fois différents et valides.

... dans ce paragraphe :

EX00 *Un poisson rouge nage tranquillement dans un aquarium. Au passage, il gobe glouonnement un filet de nourriture avant qu'un autre habitant du même aquarium, un bêta, ne le lui chipe.*

... dans cet extrait du site Web d'un programme universitaire :

EX01 *Le programme vise à former des professionnels des TI, capables de s'intégrer d'emblée à des équipes de développement ou de maintenance de systèmes informatiques, pour accéder ensuite rapidement à la fonction de chargé ou chef de projet.*

... dans cet extrait d'un article de journal :

EX02 *Au cours de la soirée, un prix hommage sera décerné à une interprète ayant marqué la chanson québécoise. Lady Alys Robi, qui a souvent triomphé au Capitole, viendra non seulement accepter la récompense, mais mettra son grain de sel dans la soirée en interprétant son classique, Tico Tico.*

EX03 Les couleurs pimpantes. L'automne est froid et cérébral, taillé sur les lignes de Hitchcock. La jupe étroite aux genoux en tweed léger et les twin-sets sages refont surface avec une absence totale de...
 Envoyer cet article Imprimer Retour Haut

Dans cet extrait d'une page Web

Dans cet exemple de série harmonique, pris du site

<http://membres.lycos.fr/villemingerard/Iteration/TrgLeibn.htm>

EX04
$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \dots = \text{Diverge}$$

§ **Harmonique** parce que, en musique, si on coupe une corde en 2 on passe à l'octave au-dessus, en 3 à la quinte au-dessus, etc.

Note : il n'y a pas qu'une seule bonne réponse dans chaque cas. L'idée ici est de proposer un découpage en classes et de discuter avec votre enseignant(e) des résultats auxquels vous arrivez, dans le but d'explorer des alternatives et d'ouvrir des horizons.

Exemple détaillé : une classe `Rectangle`

Rôle de cette section

Pour s’attaquer à une approche nouvelle, quoi de mieux que d’y aller d’un exemple concret? Dans cette section, nous examinerons comment concevoir une classe simple, comment l’instancier et comment se servir des instances résultantes. Cela nous permettra de voir comment on pense un objet simple. Puisque nous en sommes à nos premiers pas, ne soyez pas étonné(e)s que nous revenions ultérieurement sur ce qui est présenté pour le raffiner.

Examinons plus en détail ce qui sera notre classe `Rectangle`, maintenant. Que pourrait-on vouloir connaître d’un `Rectangle`? Que pourrait-on vouloir lui faire accomplir?

Eh bien, pour donner quelques exemples, on pourrait vouloir :

- connaître sa position sur un plan (un point d’origine, celui de son coin en haut à gauche par exemple);
- connaître l’extrémité opposée à son point d’origine (un autre point, en bas à droite);
- connaître sa hauteur, sa largeur;
- connaître son périmètre, son aire (sa surface);
- le déplacer (changer son point d’origine);
- en changer la largeur;
- en changer la hauteur;
- savoir le dessiner; *etc.*

Tenons-nous en à ce qui est listé ci-dessus, pour le moment, et ajoutons à tout cela deux opérations fondamentales :

- savoir ce que signifie créer un `Rectangle` (savoir le *construire*); et
- contrôler ce qui se produira à la fin de l’existence du `Rectangle` (savoir le *détruire*).

Nous traiterons de ces deux opérations particulières (mais essentielles) dans la section *Vie d’un objet*. Nous ajouterons aussi d’autres opérations de base à notre rectangle dans la section *Objets et opérateurs*.

Pour le moment, nous avons du pain sur la planche, alors mettons-nous au travail.

Il y a une *énorme* différence entre indiquer qu’on pourrait vouloir connaître une donnée sur un objet et prétendre que l’objet doit avoir un attribut tenant à jour cette donnée.

Nous y reviendrons souvent dans ce document : un objet bien conçu se définit principalement par sa gamme de services, en particulier ceux faisant partie de son visage public, et non pas par ses attributs.

Au sujet des consignes typographiques

Au sens des conventions typographiques pour les classes et les méthodes, il existe plusieurs écoles de pensée, la plupart du temps cohérentes à l’interne. Les standards ISO ont les leurs, *Sun* propose les siennes pour la plateforme Java, *Microsoft* propose les siennes pour la plateforme .NET, la bibliothèque *Boost* a la sienne, et ainsi de suite.

Nous utiliserons un format (proche de celui préconisé par *Boost* et par *Sutter* et *Alexandrescu* dans [CppCodStd]) qui se veut lui aussi cohérent à l’interne, mais il est possible que ce format diffère de celui que vous avez rencontré dans le passé. Ce n’est pas une hérésie, et le format préconisé ici n’est ni meilleur, ni pire qu’un autre, n’ayant comme prétention que celle d’être cohérent.

Plusieurs niveaux d'opérations

Si on examine les opérations listées ci-dessus, on constate qu'il est possible de les regrouper en catégories, d'au moins deux manières distinctes.

L'une d'entre elles serait de distinguer entre *celles qui utilisent le rectangle*⁸ sans permettre de le modifier, et qui servent donc à en tirer de l'information, et *celles qui font en sorte de modifier le rectangle*.

- Dans le premier cas, on retrouvera par exemple une méthode permettant d'obtenir la hauteur d'un rectangle. On préfixera souvent ce type de méthode du vocable **Get**.
- Dans le second cas, on retrouvera par exemple une méthode permettant de modifier la hauteur d'un rectangle. On préfixera souvent ce type de méthode du vocable **Set**.

Voir *Appendice 02 – Nommer et documenter* pour une discussion portant entre autres sur la question du choix de préfixes dans un nom de méthode.

L'autre serait de distinguer celles qui sont de *bas niveau*, qui touchent de très près aux attributs de l'objet, et celles qui sont de plus *haut niveau* qui, pour la plupart, utilisent les opérations de bas niveau pour faire leur travail.

- Dans le premier cas, on retrouvera par exemple une méthode permettant d'obtenir la hauteur d'un rectangle, présumant que cette information est élémentaire⁹.
- Dans le second cas, on retrouvera par exemple une méthode permettant d'obtenir le périmètre d'un rectangle, présumant que ce calcul périmètre repose en partie sur une connaissance de sa hauteur.

Ces catégorisations ont une importance dans notre travail. Ainsi :

- les opérations de bas niveau seront utilisées très souvent, de manière directe (pour connaître la hauteur d'un rectangle) ou indirecte (pour en connaître le périmètre). *Il faudra donc que celles-ci soient particulièrement efficaces côté vitesse*¹⁰;
- les opérations qui ne modifient pas l'objet doivent être identifiées comme telles pour permettre au compilateur d'assurer un degré de sécurité intéressant dans les systèmes auxquels il contribuera. Le compilateur C++ est très strict de ce côté, et il faut développer de bonnes habitudes de travail avec lui pour avoir du succès.

⁸ Par exemple, celles qui offrent simplement de l'information sur le rectangle (nous en donnons la hauteur) ou celles qui opèrent sur celui-ci sans le changer (celle qui le dessine).

⁹ Quoique, selon la représentation interne choisie, cela même puisse reposer sur d'autres méthodes.

¹⁰ L'un des reproches souvent faits aux systèmes développés à l'aide de technologies OO est qu'ils tendent à être plus gros et plus lents que leurs contreparties structurées. Le langage C++ permet d'éviter ce genre d'écueil : un programme C++ *bien écrit* n'a rien à envier, côté vitesse ou taille, à un programme C voué à la réalisation de la même tâche. Il faut toutefois réfléchir un peu plus longtemps à notre design avant de le réaliser, du moins le temps de développer des réflexes de reconnaissance de ce qui fera qu'un design fonctionnera ou ne fonctionnera pas, pour obtenir d'aussi bons résultats. La section *Encapsulation et vitesse* de ce document présente brièvement une technique simple pour optimiser ces accès de base, de même que quelques raisons pour lesquelles vous devriez vous intéresser à la question.

On nommera parfois **accesseur** une méthode permettant une consultation sans modification de l'objet (souvent un `Get`, d'où le néologisme anglais *Getter*), et **mutateur** une méthode vouée à sa modification (souvent un `Set`, d'où le néologisme anglais *Setter*). Certains langages, par exemple Delphi ou C#, ont formalisé cet idiome dans un concept qu'ils nomment des *propriétés* (voir *Propriétés*, plus bas).

Répartissons ces quelques opérations dans un tableau, et essayons de voir comment on pourrait les catégoriser. Ceci nous permettra entre autres de les développer dans un ordre raisonnable par la suite.

<i>Opération par laquelle un Rectangle...</i>	Niveau		Peut modifier l'objet	
	<i>Bas</i>	<i>Haut</i>	<i>Oui</i>	<i>Non</i>
...révèle sa position	X			X
...révèle l'extrémité opposée	X			X
...révèle sa hauteur	X			X
...révèle sa largeur	X			X
...révèle son périmètre		X		X
...révèle son aire		X		X
...se déplace	X		X	
...modifie sa largeur	X		X	
...modifie sa hauteur	X		X	
...se dessine		X		X
...se construit	X		X	
...se détruit	X		X	

Puisqu'il faut faire un choix quant à la structure interne du `Rectangle`, nous ferons celui (pleinement arbitraire) de définir un rectangle par un point d'origine, par sa largeur et sa hauteur. Ainsi, connaître l'extrémité opposée deviendra une opération de plus haut niveau pour nous. Cela dit, une maxime simple pour obtenir du succès dans une approche OO est de concevoir les objets sur une base opératoire plutôt que structurelle : ce qui compte le plus dans un objet, c'est ce qu'il peut faire, pas comment il est structuré.

Vous réaliserez vous-mêmes l'implantation du point d'origine d'un `Rectangle` à l'aide d'une classe `Point` servant à représenter un point dans le plan, dans un exercice proposé plus loin.

Considérations techniques

Une classe en C++ sera par convention **déclarée** dans un fichier d'en-tête (extension `.h`, pour *header file*¹¹) portant son nom.

Déclaration

⇒ On s'attend donc à trouver la déclaration de la classe nommée `Rectangle` dans le fichier `Rectangle.h`¹².

Le fichier d'en-tête, contenant la déclaration d'une classe, sera inclus par tout module ayant besoin d'utiliser cette classe. L'information qui s'y retrouve doit être suffisante pour permettre au compilateur de comprendre la structure de la classe, et pour permettre aux modules voulant utiliser cette classe de comprendre comment s'en servir. Il s'agit, conséquemment, d'un fichier qui doit contenir de la documentation claire et adéquate pour aider à la bonne utilisation éventuelle de l'objet qui y est décrit.

Modifier la déclaration d'une classe demande que tous les modules qui l'incluent soient compilés à nouveau. On souhaitera donc que la déclaration d'une classe soit au point le plus tôt possible dans le cycle de développement d'un système informatique.

Par convention toujours, le code associé à une classe (la **définition** de ses méthodes) se retrouvera dans un fichier source (extension `.cpp`, pour C++) portant son nom.

Définition

⇒ On s'attend donc à trouver la définition de la classe nommée `Rectangle` dans le fichier `Rectangle.cpp`.

Le fichier source, logeant la définition de la classe, contiendra la définition des méthodes de cette classe, et pourra au besoin être livré sous une forme déjà compilée pour ne pas révéler de secrets d'entreprise. La documentation de ce fichier devra servir de support au développement, décrire les révisions¹³ qui y ont été apportées, les stratégies et algorithmes choisis, et toute autre note technique appropriée.

¹¹ Certains (pas nous) utiliseront aussi l'extension `.hpp` pour ces fichiers. L'idée derrière ce choix est d'éviter la confusion avec les en-têtes du langage C. En pratique, cette tentative de standardisation n'a pas eue beaucoup de succès alors on rencontre une majorité de fichiers d'en-tête portant l'extension `.h` encore aujourd'hui.

¹² Les partisans de la notation hongroise (voir *La notation hongroise*, dans *Appendice 02 – Nommer et documenter*) utiliseront généralement `Rectangle.h` pour la déclaration de la classe `CRectangle`, malgré le préfixe `C` apposé au nom de la classe (idem pour `Rectangle.cpp` et la définition de `CRectangle`). Ceci ne cause pas de confusion pour un projet en notation hongroise puisque, selon cette démarche, toutes les classes portent le même préfixe. Si on ne mêle pas les notations dans un projet, il n'y aura pas de conflit de noms entre une classe `Rectangle` et une classe `CRectangle`.

¹³ Datées et signées de leur auteur, bien entendu.

Modifier la définition d'une classe, si sa déclaration n'a pas été changée, ne demande une nouvelle compilation que du fichier source modifié, et une nouvelle édition des liens des systèmes informatiques qui l'utilisent; coûteux, soit, mais beaucoup moins qu'une nouvelle compilation généralisée du système.

Réduire l'impact des changements apportés à une classe ou à un module sur les autres classes et les autres modules contribue à **réduire le couplage** entre ces unités de code. De manière générale, on estime qu'un système à *faible couplage* et à *forte cohésion* est souhaitable¹⁴.

Appliquer la technique **une classe, un module** vous simplifiera fréquemment la vie en C++, et il s'agit d'un design qui se transpose bien dans la plupart des langages OO (Java et C# y compris). C'est une bonne idée d'en développer l'habitude tout de suite.

Nous verrons éventuellement qu'il y a parfois lieu de déroger à cette stratégie, qui se veut plus un guide général qu'une règle stricte.

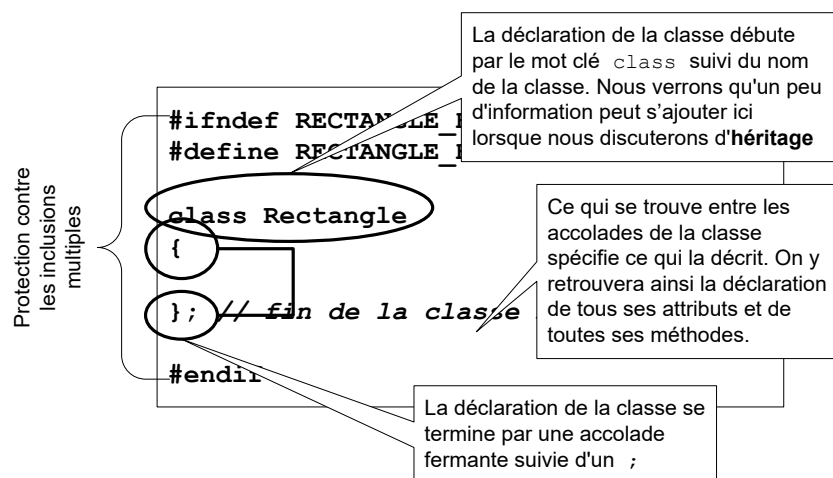
Pour l'initiation au modèle OO que nous faisons ici, évidemment, procéder à partir d'une règle simple et claire sera avantageux à plusieurs égards.

Déclaration de la classe *Rectangle*

La déclaration de la classe suivra une forme telle que celle proposée à droite¹⁵.

Une fois la classe déclarée, on est en droit de s'en servir (quoique, pour le moment, il n'y ait que bien peu de choses à en tirer).

Un petit programme de démonstration utilisant un rectangle pourrait avoir l'air de ceci :



```
#include "Rectangle.h" // inclure la déclaration de la classe Rectangle
int main() {
    Rectangle r; // légal, dans la mesure où la classe Rectangle a été déclarée
                // au préalable. Ici, r est une instance de la classe Rectangle
}
```

Vous pouvez essayer le programme ci-dessus et le compiler. Si la compilation et l'édition des liens se terminent sans erreur, vous êtes sur la bonne voie.

Commençons maintenant à rédiger la définition de la classe, et à rencontrer les conventions du petit monde du développement OO.

¹⁴ Ces concepts sont attribués à *Larry Constantine*, l'un des penseurs derrière la programmation structurée.

¹⁵ Détail technique : il est sage d'insérer la protection contre les inclusions multiples dès le début d'un fichier d'en-tête, avant même les commentaires d'usage, du fait qu'il existe des compilateurs qui procèdent à certaines optimisations particulières du temps de compilation lorsqu'ils rencontrent ce patron de travail.

Opérations de base

Commençons par examiner comment nous pourrions définir les accesseurs et les mutateurs pour la hauteur et la largeur d'un `Rectangle`. Pour simplifier les choses, nous spécifierons ces deux attributs comme étant des entiers.

Accesseurs

On nommera **accesseur** une méthode permettant un accès surtout consultatif à une information sur un objet.

Les accesseurs protègent fréquemment (mais pas toujours) l'information en question contre toute forme de modification par ceux y ayant recours.

On verra parfois le mot observateur (en anglais : *Observer*) appliqué à ce type de méthode. Nous éviterons ce terme ici parce qu'il sert aussi pour identifier un schéma de conception (*Design Pattern*) très connu.

Par convention, le nom d'un accesseur débutera par le préfixe **Get**, suivi du nom de l'information à obtenir.

La classe `Rectangle` telle que nous la concevons aura au moins deux accesseurs, nommés **GetHauteur()** et **GetLargeur()**. Notez que rien n'oblige une classe à exposer un accesseur par attribut; un accesseur est un service offert aux clients d'un objet et leur permettant de poser des questions auxquelles cet objet accepte de répondre.

Un **accesseur strict** est un bon cas de méthode qui ne devrait en rien modifier l'objet consulté. Par exemple, *il n'y a pas de raison pour que `GetLargeur()` modifie la largeur ou tout autre attribut du rectangle examiné.*

Quand on peut garantir que l'action d'une méthode ne modifiera en rien l'objet auquel elle appartient, il est très avantageux de l'indiquer en lui apposant la mention **const**.

Dans la définition de la classe `Rectangle`, une fois les prototypes des méthodes nommées `GetHauteur()` et `GetLargeur()` ajoutés, on aura ceci.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
public:
    int GetHauteur() const;
    int GetLargeur() const;
}; // fin de la classe Rectangle

#endif
```

Le prototype des méthodes nommées `GetLargeur()` et `GetHauteur()`.

Pour ne prendre qu'un seul exemple, si on a deux instances de `Rectangle` nommées `r1` et `r2`, alors:

- la méthode `r1.GetLargeur()` sera un sous-programme de type `int` dont l'exécution garantit ne rien modifier à `r1`; et
- la méthode `r2.GetLargeur()` sera un sous-programme de type `int` dont l'exécution garantit ne rien modifier à `r2`.

Remarquez l'emploi de la spécification **public:** qui indique que ce qui suit, *jusqu'à indication contraire*, sera public, donc accessible à tous. Certains langages OO exigent que chaque membre, qu'il s'agisse d'un attribut ou d'une méthode, soit qualifié individuellement, mais C++ permet de procéder par blocs.

⇒ **Par défaut, en C++, les membres d'une classe sont privés.** Certains langages (Java et C#, en particulier) utilisent une autre qualification, moins contraignante, par défaut.

Bonifions notre programme de démonstration pour montrer comment on peut utiliser une méthode de `r`, notre instance de `Rectangle` :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
#include <iostream>
int main() {
    using namespace std;
    Rectangle r;
    // Appel à la méthode GetLargeur() de r
    cout << "La largeur du Rectangle r est "
         << r.GetLargeur()
         << endl;
}
```

Ceci compilera sans erreur, du fait que la classe `Rectangle` est déclarée dans `Rectangle.h`, que le prototype de la méthode `GetLargeur()` y est correctement indiqué, et que l'appel fait à la méthode `GetLargeur()` est conforme à son prototype.

L'édition des liens échouera, par contre, puisque nous n'avons pas encore écrit la définition de cette méthode (le code à proprement dit). Le travail n'est pas terminé...

Mutateurs

On nommera **mutateur** une méthode permettant un accès sur un objet d'une manière risquant de le modifier.

Les mutateurs les plus fondamentaux sont ceux qui donnent un accès à des attributs. Par convention, le nom d'un tel mutateur débutera par le préfixe **Set**, suivi du nom de l'information à modifier.

Par exemple, la classe `Rectangle` aura au moins deux mutateurs, nommés **SetHauteur(int)** et **SetLargeur(int)**, tel qu'indiqué ci-dessous¹⁶. Encore une fois, rien n'oblige une classe à offrir des mutateurs; ici, offrir ces deux mutateurs pour `Rectangle` signifie que la classe juge pertinent qu'on puisse modifier directement ces deux caractéristiques d'un `Rectangle` (on aurait aussi pu choisir de ne pas le permettre).

On trouvera aussi dans la littérature le terme modificateur (en anglais *Modifier*) pour cette catégorie de méthodes.

J'utilisais auparavant le terme plus classique de *manipulateur* ici, mais l'évolution de la POO a fait dériver ce mot qui désigne maintenant un idiome particulier (mais fort joli) du schéma de conception visiteur. J'ai donc choisi, dans ce cas, d'adapter mon vocabulaire à l'air du temps.

```
class Rectangle
{
public:
    // -- Accesseurs de premier ordre
    int GetHauteur() const;
    int GetLargeur() const;
    // -- Mutateurs de premier ordre
    void SetHauteur(int);
    void SetLargeur(int);
}; // fin de la classe Rectangle
```

On trouve ici le prototype des méthodes `SetLargeur()` et `SetHauteur()`.

Ces méthodes ne sont pas `const`, puisqu'elles auront comme rôle de modifier le `Rectangle` auquel elles appartiennent.

Leurs paramètres pourraient être `const`. Il n'y a pas de raison pour que ceux-ci soient modifiés par l'action de l'une ou l'autre de ces deux méthodes. Plusieurs omettront le `const` appliqué aux paramètres du fait que ceux-ci sont passés par copie, mais indiquer `const` ne peut pas nuire à votre programme (au mieux, vous gagnez un peu au change; au pire, vous ne perdez rien). Ici, j'ai pêché par simplicité.

Modifions notre programme de démonstration en y ajoutant un exemple utilisant un mutateur :

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    Rectangle r;
    r.SetLargeur(3); // Intuitivement, la largeur du Rectangle nommé r devrait devenir 3
    // Appel à la méthode GetLargeur() de r (devrait nous afficher que sa largeur est 3)
    cout << "La largeur du Rectangle r est "
         << r.GetLargeur()
         << endl;
}
```

¹⁶ Notez que les détails du fichier qui ne sont pas pertinents à la classe en tant que telle, comme les directives du préprocesseur servant à protéger le fichier contre les inclusions multiples, seront omis à partir de maintenant dans le but d'alléger le texte.

Attributs

Si nous voulons rédiger la définition de nos premières méthodes, qui auront comme rôle de permettre la consultation et la modification de la largeur et de la hauteur d'un `Rectangle`, il nous faut passer à la déclaration des attributs que seront la largeur et la hauteur d'un `Rectangle`, car c'est bien là notre choix d'implantation, fait plus haut.

Dans le but d'imposer le principe d'encapsulation et de protéger rigoureusement nos données, nous spécifierons nos attributs comme étant **privés** (spécification **`private:`**, qui indique que ce qui suit, *jusqu'à indication contraire*, sera privé).

Aussi, pour fins de clarté et pour éviter la confusion dans certaines circonstances, nous appliquerons au nom de chaque attribut le suffixe `_`¹⁷. Ainsi, par exemple, la largeur du `Rectangle` sera nommée **`largeur_`**¹⁸.

Ici aussi, les us et coutumes sont à la fois nombreux et variés. En effet :

- certains préfixeront les attributs d'un caractère de soulignement (p. ex. : `_hauteur`). Ne le faites pas en C++, cependant, car les compilateurs ont le droit de se réserver les noms préfixés du symbole `_` et certains le font!
- d'autres en utiliseront deux (p. ex. : `__hauteur`), ou même trois. Ne le faites pas (voir ci-dessus);
- certains, dans la tradition de la bibliothèque MFC, utiliseront le préfixe `m_` (pour *Data Member*);
- certains utiliseront simplement le nom à représenter (p. ex. : `hauteur` plutôt que `hauteur_`) et résoudre la confusion qui survient lorsqu'un paramètre d'une méthode portera le même nom que l'attribut en utilisant des mécanismes du langage; *etc.*

Sachez toutefois que les noms préfixés par au moins un caractère de soulignement ne sont pas portables, certains compilateurs sur les réservant. Ceci explique que nous privilégions ici un suffixe.

```
class Rectangle
{
public:
    // -- Accesseurs de premier ordre
    int GetHauteur() const;
    int GetLargeur() const;
    // -- Mutateurs de premier ordre
    void SetHauteur(int);
    void SetLargeur(int);
private:
    int hauteur_,
        largeur_;
}; // fin de la classe
```

Déclaration des attributs `largeur_` et `hauteur_`, qui représenteront respectivement la largeur et la hauteur d'un `Rectangle`. Chaque instance de `Rectangle` aura sa propre hauteur et sa propre largeur.

¹⁷ ...en conformité avec les usages de la bibliothèque *Boost* et les recommandations de *Herb Sutter* et *Andrei Alexandrescu* dans [CppCodingStd]

¹⁸ Lorsque nous programmerons la définition de la méthode `SetLargeur(const int)`, nous pourrons alors nommer `largeur` le paramètre reçu par cette méthode sans créer de conflit de nom avec le membre `largeur_` représentant la largeur.

Notre déclaration de la classe `Rectangle` comprend maintenant deux attributs privés, de même qu'un mutateur et un accesseur publics pour chacun d'entre eux.

Rectangle
- hauteur_ : int - largeur_ : int
+ GetHauteur() const : int + GetLargeur() const : int + SetHauteur(hauteur : int) : void + SetLargeur(largeur : int) : void

La notation UML préconise que les types utilisés dans un diagramme de classe soient ceux qui devront être utilisés lorsque la classe sera effectivement déclarée ou définie.

Dans un standard comme les standards ISO, qui indiquent ce qui sera éventuellement implémenté dans un langage ou un autre et qui dicte des règles sans les réaliser directement, la nature des types primitifs apparaissant dans les diagrammes de classe devra accompagner (et habituellement précéder) leur utilisation dans lesdits diagrammes.

Pour illustrer l'impact de la spécification **private**, examinez le programme ci-dessous, en portant une attention particulière aux commentaires :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
#include <iostream>
int main() {
    using namespace std;
    Rectangle r;
    r.SetLargeur(3); // Légal: SetLargeur(const int) est public
    // r.largeur_ = 8; // Illégal: l'attribut est privé
    cout << r.GetHauteur() << endl; // Légal: GetLargeur() est public
    // cout << r.hauteur_ << endl; // Illégal: l'attribut est privé
}
```

Un effet secondaire désirable

Ayant « caché » (rendu privé) chaque attribut, et s'engageant à rédiger des méthodes pour les consulter (accesseur) et pour les modifier (mutateur), on vient d'imposer à toute entité externe à l'objet lui-même l'obligation de passer par *ses* méthodes pour avoir accès à *ses* attributs.

L'encapsulation des attributs est réalisée, à la base, par cette manœuvre assez simple qui est d'enrober l'accès à chaque attribut à l'aide de méthodes. De manière plus générale, un objet offrira des services publics qu'il juge utiles pour ses clients, et eux seuls auront accès à ses membres privés (en particulier, à ses attributs).

Pour faciliter le débogage et les tests, on visera aussi à réduire au minimum le nombre de méthodes accédant à un attribut donné; quand cet attribut sera accédé, le nombre d'endroits à surveiller sera petit. Cet effet secondaire fait qu'il devient possible, quand l'accès aux attributs via des accesseurs et des mutateurs est rigoureux même à l'intérieur de l'objet¹⁹, de suivre à la trace *chaque accès à un attribut, quel qu'il soit*. Toute consultation, toute modification d'un attribut peut donc être surveillée.

¹⁹ Cette rigueur importe surtout pour le code interne à l'objet, car pour accéder à un attribut à partir de l'extérieur de l'objet, cette façon de procéder est *obligatoire*, du moins si l'encapsulation est stricte.

Attributs publics... Jamais?

Pour réaliser l'encapsulation, devrait-on systématiquement éviter les attributs publics? La réponse est simple... Presque!

En fait, dans les mots d'**Andrew Koenig** (je ne retrouve pas la citation), il est raisonnable d'exposer des attributs publics dans deux cas :

- quand ils sont `const`, ce qui exclut toute possibilité de modification ultérieure; ou
- quand les états de l'objet (ses attributs) sont en fait son interface.

Évidemment, il arrive que ces deux clauses s'avèrent pour un même objet. L'idée ici est que dans un cas, il ne sert à rien d'assurer un suivi des changements d'état d'un objet, ses états étant fixés dès sa construction (voir ***Le constructeur***, plus loin), alors que dans l'autre, on accepte d'office n'importe quelle valeur pour chacun des états visés. Koenig a alors en tête un point 2D, paire $\{x, y\}$, où toutes les valeurs sont admissibles pour x comme pour y , et où le suivi des changements d'états n'est pas pertinent.

Ce sont deux cas limites, disons-le d'office, et il est rare que des classes exposant des attributs publics soient une bonne idée. Les accesseurs sont essentiellement gratuits sur le plan du temps d'exécution, et facilitent l'entretien du code; mieux vaut les utiliser. Ne pas contrôler dès le début l'accès aux attributs par des méthodes signifie qu'il ne sera pas possible, ultérieurement, de contrôler cet accès sans briser du code existant.

Définition des méthodes

Rôle de cette section

Dans un contexte OO, en appliquant le principe d'encapsulation, on s'impose de ne communiquer avec nos objets que de manière polie, sécurisée. Les premières opérations (les premières **méthodes**) que nous examinerons ici nous permettront une telle interaction propre et respectueuse avec nos objets.

Nous savons qu'une classe `Rectangle` existe, ce qui signifie qu'on peut déclarer des instances de `Rectangle`²⁰. Nous savons aussi qu'avec les choix de design faits plus haut, il est possible d'interroger un `Rectangle` pour lui demander quelle est sa hauteur et quelle est sa largeur²¹, tout comme il est possible de lui demander de modifier sa hauteur et sa largeur²².

Relevez la terminologie : *on ne modifie pas la hauteur d'un `Rectangle`, mais on demande au `Rectangle` de modifier lui-même sa hauteur.*

Reste aussi à voir comment écrire le code associé à chacune des méthodes de `Rectangle` pour que la tâche annoncée soit effectivement accomplie.

Rappel : si la déclaration de la classe `Rectangle` se fait, par convention, dans un fichier d'en-tête nommé `Rectangle.h`, la définition de cette classe (le code associé à ses méthodes, essentiellement) se fait, toujours par convention, dans le fichier source nommé `Rectangle.cpp`.

La première ligne du fichier `Rectangle.cpp` devrait, logiquement, être la suivante :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
```

Il est nécessaire d'inclure le fichier contenant la déclaration de la classe pour rédiger la définition de ses méthodes.

²⁰ On dira qu'on peut *instancier* `Rectangle`. Notez que nous faisons ici un abus de langage, puisqu'il existe des classes qu'on ne peut instancier. Ces classes, dites *classes abstraites*, existent pour des raisons spécifiques, et nous seront d'une grande utilité, malgré les apparences.

²¹ À l'aide de ses accesseurs `GetLargeur()` et `GetHauteur()`.

²² À l'aide de ses mutateurs `SetLargeur(int)` et `SetHauteur(int)`.

Méthodes et messages

Dans ce document, nous référons aux actions possibles (aux *comportements*) d'un objet sous le vocable *méthodes*, une méthode étant un sous-programme membre d'un objet. Ainsi, nous utilisons les locutions *invoquer une méthode*, *appeler une méthode* et *utiliser une méthode* pour expliquer comment on peut solliciter l'aide d'un objet, comment interagir avec lui.

Le principe d'encapsulation repose en grande partie sur l'exposition d'un ensemble judicieux de méthodes. Transiger avec un objet à travers ses méthodes équivaut à le responsabiliser, à lui faire confiance. On présume alors que l'objet fera correctement (en fait, aussi bien que possible) son travail; qu'il est un spécialiste de sa propre question.

Chaque objet ayant une vocation claire, on est en droit de le considérer comme un spécialiste dans la réalisation des tâches qui lui sont dévolues. Ceci se reflète dans la terminologie objet²³ : l'encapsulation amène une subdivision des tâches, qui suppose qu'on ait confiance en celles et ceux qui vont les accomplir.

Derrière cette philosophie, donc, se cache un autre vocable pour la communication avec un objet, vocable plus général qu'*invoquer une méthode*²⁴ : ***envoyer un message***.

⇒ L'**envoi de message** est une généralisation conceptuelle de l'appel de méthodes, et englobe cette dernière. Communiquer avec un objet par passage de message, c'est lui demander poliment²⁵ de réaliser une tâche.

Certains livres portant sur la POO n'emploient que ce vocable, prenant une approche plus aérienne que celle prise dans la présente, et c'est une approche tout à fait valide – celle que nous aurions peut-être pris ici si nous n'avions pas eu à couvrir les bases techniques au passage.

²³ ...et dans la terminologie client/ serveur, et pour les mêmes raisons!

²⁴ L'idée d'appel suppose une communication de bas niveau (un appel de sous-programme) avec l'objet contacté, ce qu'*envoyer un message* ne présuppose pas – l'envoi pourrait se faire à travers un lien de télécommunication, sans perte de sens terminologique.

²⁵ En respectant un protocole (ce qui englobe l'idée de réaliser un appel de méthode correct).

Définir les accesseurs

Commençons la rédaction de la définition de nos méthodes par celle d'un petit accesseur, `GetHauteur()`. La syntaxe est un peu particulière de prime abord, alors relevons tous les éléments qui doivent y apparaître et apprenons tout de suite à les reconnaître :

- le nom de la méthode est `GetHauteur()`;
- il s'agit d'une méthode de la classe `Rectangle`;
- son type est `int`, ce qui équivaut à spécifier que la valeur retournée par la méthode sera de type `int`;
- elle ne prend pas de paramètres (il n'y a rien entre les parenthèses qui suivent son nom);
- la méthode est `const`, ce qui signifie qu'elle garantit de ne pas modifier le `Rectangle` auquel elle appartient;
- son rôle est d'informer le sous-programme qui fait appel à ses services de la hauteur du `Rectangle` auquel elle appartient.

```
class Rectangle {
public:
    int GetHauteur() const;
    int GetLargeur() const;
    void SetHauteur(int);
    void SetLargeur(int);
private:
    int hauteur_,
        largeur_;
};
```

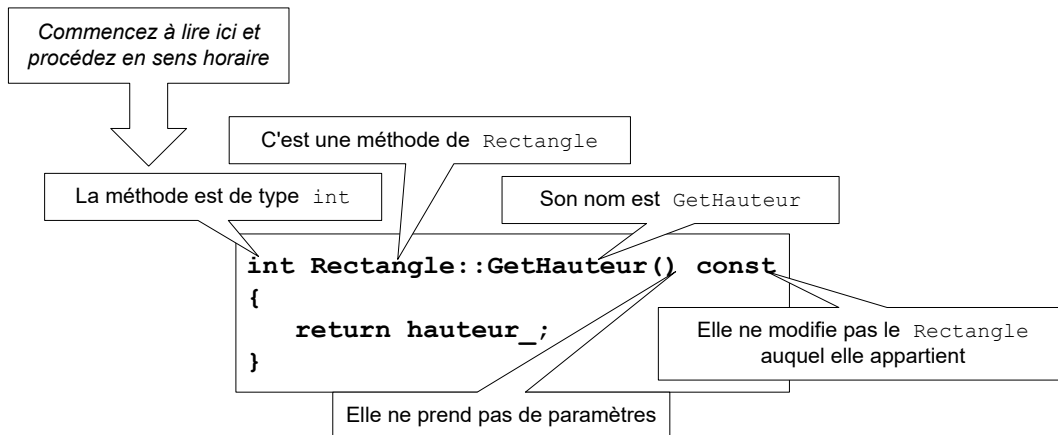
Une fois ces éléments identifiés, la lecture de la définition de la méthode est plus facile à faire. Bien qu'il y ait une certaine lourdeur syntaxique à tout ceci, chacun de ces éléments est important à la bonne spécification de la méthode. Le langage C++ offre beaucoup de latitude aux développeurs, mais il faut que ceux-ci paient cette latitude par un effort de clarté plus grand dans leurs spécifications.

La définition de la méthode est donc telle que proposé dans l'extrait à droite.

On sera d'accord sur le fait que cette méthode n'a rien changé à la valeur de quelque attribut de `Rectangle` que ce soit. La mention `const` de la méthode est par conséquent pleinement justifiée (et légale).

```
// déclaration de la classe Rectangle
#include "Rectangle.h"
int Rectangle::GetHauteur() const {
    return hauteur_;
}
```

Remarquez les similitudes entre la définition d'une méthode et celle d'un sous-programme conventionnel (position des accolades, du type du sous-programme, du `return`, *etc.*). Les points à identifier sont, tels qu'indiqués plus haut :



Sans effort, on peut imaginer ce dont aura l'air la méthode `GetLargeur()`.

Ce patron de méthode est fréquemment rencontré; on peut pratiquement parler ici d'une recette. Cela dit, fréquent ne veut pas dire universel, ce que nous constaterons sous peu.

```

#include "Rectangle.h"
int Rectangle::GetHauteur() const {
    return hauteur_;
}
int Rectangle::GetLargeur() const {
    return largeur_;
}
  
```


Question d'identité et d'appartenance

Ce qui peut poser problème à la compréhension est le `hauteur_` qui apparaît dans la méthode `GetHauteur()`... *à qui appartient-il?*

Pour bien saisir la mécanique en jeu ici, voici un exemple détaillé.

Nous avons la classe `Rectangle` (à droite, simplifiée). Chaque instance de `Rectangle` aura un attribut nommé `hauteur_`, qui est de type `int`.

C'est ce que nous dit la déclaration de cet attribut, dans une section privée de la déclaration de la classe.

```
class Rectangle {
public:
    int GetHauteur() const;
private:
    int hauteur_;
};
```

```
#include "Rectangle.h"
int Rectangle::GetHauteur() const {
    return hauteur_;
}
```

La définition de la méthode (à gauche) fait référence à `hauteur_`. Mais le `hauteur_` de quel `Rectangle`?

C'est à l'utilisation que la question de propriété d'un membre d'instance devient plus claire.

Le programme à droite définit deux instances de `Rectangle` nommées `r1` et `r2`. Chacun a sa propre hauteur (son propre attribut `hauteur_`), et on peut faire appel pour chacun à sa méthode `GetHauteur()`.

Un appel de méthode effectué hors de l'objet possédant cette méthode se fera **toujours** à travers l'objet propriétaire. À droite, le premier appel demande la hauteur de `r1`, et le second la hauteur de `r2`... à l'utilisation, il n'y a jamais d'ambiguïté!

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    Rectangle r1, r2;
    // ...
    cout << r1.GetHauteur() << endl;
    cout << r2.GetHauteur() << endl;
}
```

On peut sans peine étendre cette idée aux autres membres et aux autres méthodes.

Dans le contexte de la définition d'une méthode, les membres appartiennent implicitement à l'instance à laquelle la méthode appartient. Chaque instance connaît ses propres membres. On verra plus loin des cas de méthodes mettant en jeu plus d'une instance à la fois, ce qui nous permettra d'apprécier encore plus le subtil jeu d'identité en cours ici.

Définir les mutateurs

La syntaxe des mutateurs suivra celle des accesseurs. D'ailleurs, il s'agit dans chaque cas de méthodes; la distinction entre catégories de méthodes est d'intérêt conceptuel et ne concerne pas le compilateur.

Les méthodes que sont `SetHauteur()` et `SetLargeur()` sont destinées à modifier le `Rectangle` auquel elles appartiennent, et le font, clairement. *On n'aurait donc pas pu les spécifier `const`* (le compilateur aurait aisément repéré l'accès illégal).

Par contre, les paramètres de chacune de ces méthodes peuvent, eux, être spécifiés `const`, puisqu'ils ne sont en rien modifiés par l'exécution de la méthode à laquelle ils sont passés. Dans `SetHauteur()`, par exemple, le paramètre `Hauteur` est laissé intact lors de l'exécution de la méthode.

```
// déclaration de la classe Rectangle
#include "Rectangle.h"
int Rectangle::GetHauteur() const {
    return hauteur_;
}
int Rectangle::GetLargeur() const {
    return largeur_;
}
void Rectangle::SetHauteur(int hauteur) {
    hauteur_ = hauteur;
}
void Rectangle::SetLargeur(int largeur) {
    largeur_ = largeur;
}
```

On notera ici que la notation choisie pour nos attributs sert surtout à distinguer le paramètre (par exemple : `largeur`) de l'attribut correspondant (nommément `largeur_`).

On retrouve dans certains créneaux du monde ○○ des pratiques cherchant à désambiguïser les attributs et les paramètres aux méthodes sans avoir recours à une convention de nomenclature particulière pour les attributs. Nous discuterons du pour et du contre dans la section **Référez à soi-même : le mot clé `this`**, plus loin.

Remarque technique

La spécification `const` apposée à une méthode fait partie de sa signature. Ainsi, deux méthodes portant le même nom, et prenant le même nombre de paramètres avec le même type pour chacun de leurs paramètres respectifs peuvent être différenciées par le compilateur si l'une est `const` et l'autre ne l'est pas.

Par exemple, ceci est légal :

```
class X {
    int valeur_;
public:
    // la version de valeur() ci-dessous retournera une copie de valeur_,
    // qu'un appelant ne pourra utiliser pour modifier valeur_ ou pour modifier
    // l'instance de X à laquelle valeur_ appartient. La méthode peut donc être
    // spécifiée const
    int valeur() const;
    // la version de valeur() ci-dessous retournera une référence à valeur_.
    // La référence n'étant pas constante, un appelant pourra s'en servir pour
    // modifier l'attribut. La méthode ne peut donc pas être spécifiée const.
    int& valeur();
};
// ce qui suit est habituellement logé dans un fichier .cpp
int X::valeur() const {
    return valeur_;
}
int& X::valeur() {
    return valeur_;
}
```

Le compilateur cherchera à utiliser la version la plus adéquate possible selon lui. Pour cibler cette adéquation, le compilateur se base sur le type de l'objet, utilisant les versions `const` des méthodes si l'objet est lui-même `const` :

```
// ... inclusions et using ...
void f(const X &cx, X &x) {
    // cx est const. Appelle la méthode const
    cout << cx.valeur();
    // x n'est pas const. Appelle la méthode qui n'est pas const, même si dans ce cas
    // bien précis, la version const aurait été tout à fait convenable
    cout << x.valeur();
    // cx est const. Ceci est illégal, car cx.valeur() retourne une temporaire
    // anonyme sur laquelle l'affectation n'a pas de sens
    // cx.valeur() = 3;
    // x n'est pas constant. Appelle la méthode qui n'est pas const, ce qui permet
    // d'affecter directement dans l'attribut valeur_ de x. Ici, par contre, on est
    // en droit de se demander pourquoi ne pas utiliser un mutateur, tout simplement
    x.valeur() = 3;
}
```

Réflexion 00.2 – Comment décider quels accesseurs et quels mutateurs (ou, de manière plus générale, quelles méthodes) une classe devrait exposer? (voir **Réflexion 00.2** : *quelles méthodes exposer?*).

Dans d'autres langages

Comment réaliser la classe `Rectangle` et le programme de démonstration, tous deux proposés plus haut, dans d'autres langages OO commerciaux? Voici des exemples en Java, C# et VB.NET. Ceci vous permettra d'apprécier à la fois les similitudes et les différences.

En Java, la classe `Rectangle` devra être placée dans un fichier nommé `Rectangle.java`. Les membres de la classe seront qualifiés `public`, `private` ou `protected` sur une base individuelle. Notez au passage l'absence de la ponctuation « : » après ces qualifications. L'absence de qualification de protection dans une classe implique une variante un peu spéciale de la qualification `private` nommée *Package Private*, que nous examinerons dans [POOv01], section *Encapsulation et regroupement*.

Remarquez que l'interface et l'implémentation d'une classe se situent toutes deux dans un seul et même fichier. Il n'est pas possible en Java (ou en C#, ou en VB.NET) de séparer les deux dans des fichiers distincts sans concevoir une entité plus abstraite (nous y reviendrons quand nous discuterons d'héritage, en particulier d'héritage d'interfaces).

Les accolades joueront le même rôle qu'en C++ mais on ne placera pas de `;` après la déclaration de la classe. Le mot clé `final` y joue un rôle semblable au mot clé `const` en C++ mais y est moins répandu (Java, comme C# et VB.NET, ne permet pas de définir de vrais objets constants); nous aurions pu qualifier `final` les paramètres aux mutateurs, par exemple.

La disposition des accolades utilisée dans le code en exemple respecte la convention officielle de *Sun Microsystems* [JavaStyle], acquis par Oracle en 2009. Cette convention demande aussi de débiter les noms de méthodes par une lettre minuscule, et de préfixer les attributs de « `this.` » pour les distinguer des paramètres lorsqu'il y a un risque d'ambiguïté.

Il est fréquent qu'un petit programme de test pour valider une classe soit défini à même cette classe, dans une méthode `static` (donc une *méthode de classe*; nous y reviendrons un peu plus loin) nommée `main(String[])` qui est l'équivalent du sous-programme `main()` de C++.

Une classe Java qui expose une méthode de classe publique nommée `main()` est une classe pouvant être exécutée. Ce concept apparaît aussi en C# et en VB.NET. Ceci permet entre autres de mettre en place des tests unitaires pour une classe donnée.

```
public class Rectangle {
    public int getHauteur() {
        return hauteur;
    }
    public int getLargeur() {
        return largeur;
    }
    public void setHauteur(int hauteur) {
        this.hauteur = hauteur;
    }
    public void setLargeur(int largeur) {
        this.largeur = Largeur;
    }
    private int hauteur,
        largeur;
    public static void main(String [] args) {
        Rectangle r0 = new Rectangle();
        r0.setLargeur(3);
        System.out.println(
            "Largeur de r0: " + r0.getLargeur()
        );
    }
}
```

L'écriture à la console se fait ici à l'aide de `System.out.println()` et affiche une `String`. Le langage Java offre un traitement spécial au type `String`, qui n'y est ni un objet tout à fait comme les autres, ni un type primitif.

En C#, les règles syntaxiques seront semblables à celles mentionnées ci-dessus pour Java. L'exemple à droite respecte la norme de disposition généralement acceptée pour un programme C#, à ceci près que j'ai, de manière volontaire, choisi de ne pas utiliser ce qu'on y nomme des propriétés²⁶.

Remarquez la présence obligatoire d'un espace nommé (`namespace`), chose optionnelle en C++ alors que Java préconise (sans l'obliger) un groupement similaire par paquetage (`package`). Le groupement obligatoire en modules logiques est une bonne pratique et se retrouve maintenant dans presque tous les langages de programmation répandus.

La méthode de classe `Main` (avec un `M` majuscule en C#) joue aussi le même rôle qu'un Java. L'affichage se fait avec `System.Console.WriteLine()` et la mention `{0}` est remplacée par le premier paramètre suivant la chaîne de caractères²⁷.

Là où Java réalise de la magie sur les chaînes de caractères, C# et VB.NET construisent implicitement des tableaux à partir de listes optionnelles de paramètres, ce qui est en soi une bonne chose. Depuis C++ 11, C++ permet des mécanismes semblables avec les *Initializer Lists* et avec les *templates* variadiques, deux sujets plus avancés.

En particulier, tout type (primitif ou non) peut se convertir ou être converti en `String`, et on peut créer une `String` qui soit la concaténation de deux `String` à l'aide de l'opérateur `+` alors que la définition d'opérateurs sur des objets y est habituellement interdite. Avec le type `String`, Java fait un peu de magie hors de portée des programmeurs pour d'autres types.

```
namespace Formes
{
    class Rectangle
    {
        public static void Main(string [] args)
        {
            Rectangle r0 = new Rectangle();
            r0.SetHauteur(3);
            System.Console.WriteLine(
                "Hauteur de r0: {0}",
                r0.GetHauteur()
            );
        }
        private int hauteur;
        private int largeur;
        public int GetHauteur()
        {
            return hauteur;
        }
        public int GetLargeur()
        {
            return largeur;
        }
        void SetHauteur(int hau)
        {
            hauteur = hau;
        }
        void SetLargeur(int lar)
        {
            largeur = lar;
        }
    }
}
```

²⁶ En effet, La pratique dans les langages .NET est d'utiliser des propriétés plutôt que des accesseurs et des mutateurs sous forme de méthodes. Nous y reviendrons en temps et lieu, mais se limiter à des accesseurs et à des mutateurs est plus homogène sur le plan de la pratique et n'enlève rien sur le plan conceptuel, alors nous nous limiterons à cela pour le moment.

²⁷ C'est triste : les chaînes de formatage constituent une pratique fragilisant les programmes car ces mécanismes ne permettent pas au compilateur de détecter des erreurs; avec cette stratégie, un nombre incorrect de paramètres ou une erreur de type ne sera captée qu'à l'exécution.

En C# comme en Java, on rencontre des variantes nuancées des qualifications de sécurité standards spécifiées par les mots clés `public`, `protected` et `private`, soit les qualifications `internal` et `protected internal`. Nous les examinerons dans [POOv01], section *Encapsulation et regroupement*.

En langage **VB.NET**, la pratique recommandable est aussi d'insérer les classes dans un espace nommé.

Cette pratique y est moins explicite et moins stricte qu'en C# (le code généré par l'IDE *Visual Studio .NET* n'en tient pas toujours compte), en partie à cause de la clientèle cible de VB qui a toujours compris à la fois des programmeurs d'expérience, qui en font leur gagne-pain, et une partie importante de programmeurs néophytes ou autodidactes.

La syntaxe des classes en VB.NET récupère en partie la syntaxe traditionnelle des langages de la lignée VB :

- les entités typées comme les constantes, les variables et les fonctions sont suivis de `As` et du type (p. ex. : `Dim i As Integer` pour déclarer une variable entière nommée `i`);
- les blocs logiques sont terminés par `End` et par le nom de la catégorie de bloc qu'ils terminent (p. ex. : `End Function` pour terminer une fonction);
- le code se décline ligne par ligne et chaque ligne incomplète d'une opération sur plusieurs lignes doit se terminer par un espace suivi d'un caractère de soulignement (« `_` »);
- les majuscules et les minuscules ne sont pas distinguées les unes des autres (*prudence!*), contrairement à ce qui prévaut avec C++, Java et C#;
- la valeur de retour d'une fonction peut être donnée par l'affectation d'une valeur au nom de la fonction (ce qui ne conclut pas l'exécution de la fonction); *etc.*

Traditionnellement, les langages VB sont plus tolérants envers des pratiques à proscrire, mais un programmeur sérieux utilisant VB.NET comme outil de développement devrait tendre vers de saines pratiques même s'il n'y est pas contraint de force par son outil.

En retour, certains éléments syntaxiques sont des dérivés de pratiques adaptées de langages suivant la stratégie C, comme par exemple la capacité d'initialiser des variables dès la déclaration.

Dans une unité de regroupement comme une classe ou un espace nommé, on qualifiera explicitement de `public`, `private` ou `protected` chaque élément, qu'il s'agisse d'une classe, d'un attribut, d'une méthode ou d'autre chose.

Remarquez que `main()` aurait ici pu être un sous-programme local à un module plutôt qu'une méthode de classe. La syntaxe d'une méthode de classe comme la méthode `main()` de `Rectangle` en `VB.NET` est `Public Shared Sub main()` (notez l'emploi de `Shared`, pas `Static`).

Tel que mentionné précédemment, aucun des trois autres principaux langages OO commerciaux (`Java`, `C#` et `VB.NET`) n'offre un réel support aux méthodes `const` ou aux instances constantes.

```
Namespace Formes
Public Class Rectangle
    Public Function GetLargeur() As Integer
        Return largeur
    End Function
    Public Function GetHauteur() As Integer
        Return hauteur
    End Function
    Public Sub SetLargeur(ByVal lar As Integer)
        largeur = lar
    End Sub
    Public Sub SetHauteur(ByVal hau As Integer)
        hauteur = hau
    End Sub
    Private largeur As Integer
    Private hauteur As Integer
    Public Shared Sub main()
        Dim r0 As New Rectangle
        r0.SetLargeur(3)
        System.Console.WriteLine _
            ("Largeur de r0: {0}", _
            r0.GetLargeur())
    End Sub
End Class
End Namespace
```

Pour la question des objets constants en `Java`, `C#` et `VB.NET`, je vous invite à lire *Appendice 00 – Sémantiques directes et indirectes*.

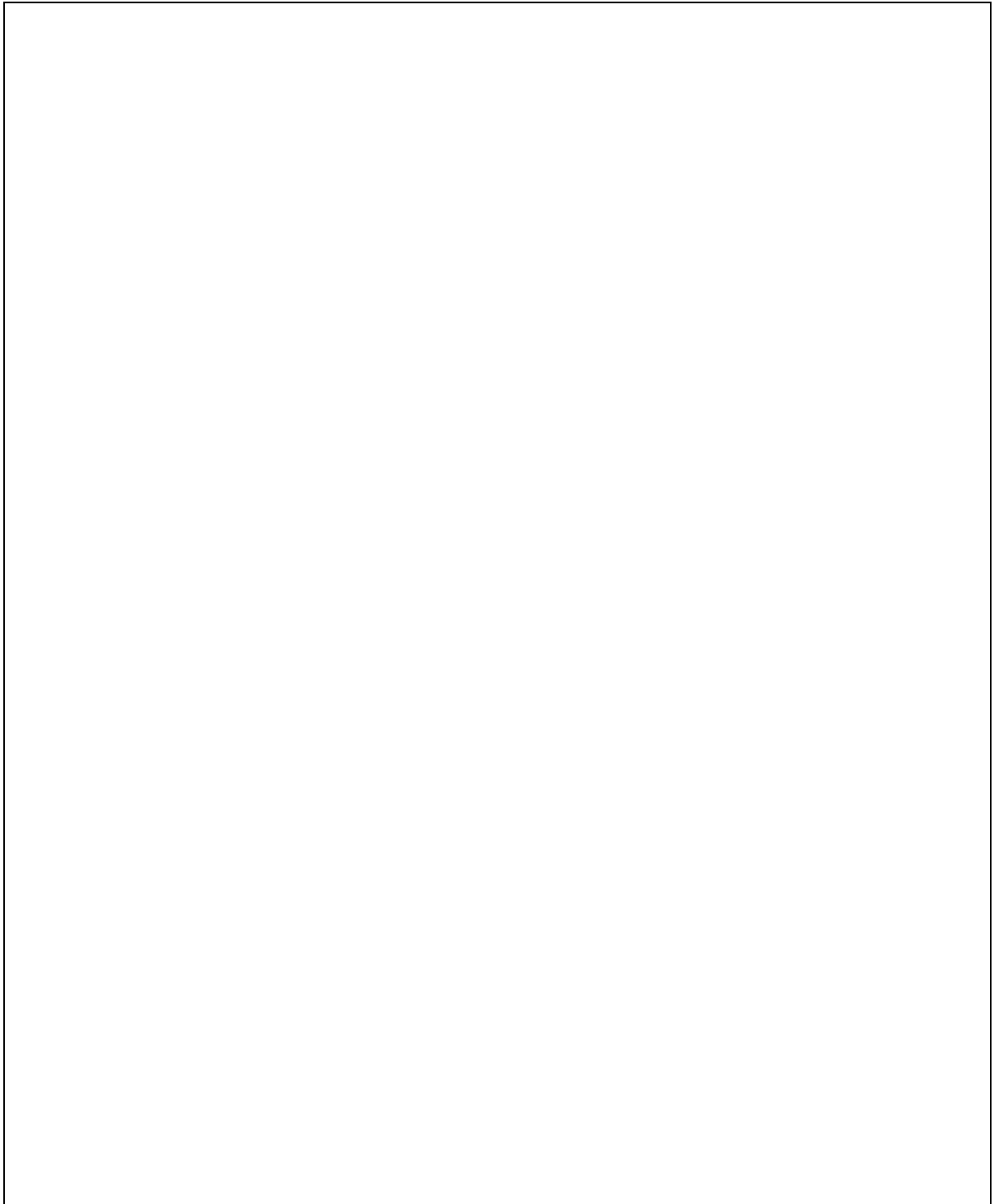
Exercices – Série 01

EX00 – Déclarez, en langage C++, la classe `Tasse` qui permet de représenter des tasses (de café, par exemple) et a les membres suivants :

- un attribut privé `volume_` représentant le volume de liquide que contient la tasse;
- une méthode publique `Verser()` qui prend en paramètre le volume de liquide à verser de la tasse et ne retourne rien;
- une méthode publique `Emplir()` qui prend en paramètre le volume de liquide à verser dans la tasse et ne retourne rien;
- un accesseur public et `const` nommé `GetVolume()` qui retourne le volume présentement contenu dans la tasse;
- un mutateur privé `SetVolume()` qui prend en paramètre la valeur à affecter à l'attribut `volume_` et ne retourne rien.



EX01 – Modifiez la déclaration de Tasse en y ajoutant les membres nécessaires pour permettre de différencier le volume présentement contenu dans une tasse et le volume maximal que cette tasse peut contenir.



EX02 – Étant donné la définition de méthode suivante :

```
double Aliment::GetTauxGlucose() const {  
    return tauxGlucose_;  
}
```

EX02.0 – À quelle classe appartient-elle? _____

EX02.1 – Combien prend-elle de paramètres? _____

EX02.2 – Quel est son type? _____

EX02.3 – Complétez le programme suivant pour qu'il lise un seuil acceptable de glucose; compare ce seuil au taux de glucose d'un pouding, et affiche si le taux de glucose dudit pouding dépasse ou non le seuil en question :

```
#include "Aliment.h"  
#include <iostream>  
int main() {  
    using namespace std;  
    Aliment pouding; // on tiendra pour acquis que dès sa déclaration, cet  
                    // objet « connaît » son taux de glucose; nous verrons  
                    // plus tard à raffiner cette idée  
    float SeuilAcceptable;  
  
    // ...  
    cout << "Seuil de glucose acceptable? ";  
    cin >> SeuilAcceptable;  
    // VOTRE CODE SUIT  
  
}
```

EX03—Étant donné la définition de méthode suivante :

```
void Menu::Presenter(const string &nomProgramme) const {
    cout << "*****\n"
         << " Bienvenue au programme " << nomProgramme << '\n'
         << "*****" << endl;
}
```

EX03.0 – À quelle classe appartient-elle? _____

EX03.1 – Combien prend-elle de paramètres? _____

EX03.2 – Quel est son type? _____

EX03.3 – Complétez le programme présenté ci-dessous pour qu'il affiche "Bienvenue au programme Bingo v1.0" :

```
#include "Menu.h"
#include <iostream>
#include <string>
using namespace std;
int main() {
    const string NOM_PROG = "Bingo v1.0";
    Menu m;
```

// VOTRE CODE SUIT

```
}
```

EX03 ci-dessus utilise à la fois '\n' et endl. Sachez que '\n' est un caractère représentant un saut de ligne (*newline*), alors que endl est une entité plus complexe mais qui fait en séquence '\n' suivi d'un ordre exigeant l'écriture sur le flux qu'est cout. Pour des raisons d'efficacité, C++ (comme beaucoup d'autres langages d'ailleurs) n'écrit réellement que si cela semble utile.

Le principe ouvert/ fermé (Open/ Closed Principle)

L'encapsulation recoupe en particulier l'un des principes importants de la POO qu'on nomme l'*Open/ Closed Principle*, ou en français le principe ouvert/ fermé.

La parenté de ce principe est généralement attribuée à **Bertrand Meyer**, concepteur du langage Eiffel. Pour en savoir plus sur le sujet :

- http://en.wikipedia.org/wiki/Open/closed_principle
- <http://www.objectmentor.com/resources/articles/ocp.pdf>

Selon ce principe, une classe devrait être *ouverte pour extension*, mais *fermée pour modification*. Une classe, une fois construite, peut être enrichie par des opérations externes (des fonctions globales, par exemple, si le langage de programmation le permet), ou par voie de spécialisation, à travers des mécanismes comme l'héritage et le polymorphisme [POOv01]. Cependant, une classe étant une unité de sécurité, elle devrait être fermée aux modifications; on ne devrait pas être en mesure de lui ajouter des attributs, des méthodes, ou d'en modifier la structure interne une fois sa déclaration terminée (en C++ : une fois l'accolade fermante de sa déclaration rencontrée).

Ce principe trouve des adhérents non seulement dans le monde OO mais aussi dans le monde du développement logiciel en général. Il est souvent perçu comme une règle de saine développement logiciel, applicable à tous les composants, objets ou autres.

Dans un langage OO, l'encapsulation contribue à plein au volet fermé pour modification de ce principe. Sans la possibilité de restreindre les accès aux membres d'une classe ou de ses instances, il serait impossible de garantir qu'une classe soit fermée pour modification.

Ce principe ne rencontre pas que des adhérents, cela dit :

- certains langages supportent les objets mais pas les clauses de sécurisation. Par exemple, l'idée de membre privé au sens strict n'existe pas en Python;
- d'autres permettent d'ajouter des membres à loisir à un objet, ou de modifier le comportement d'une méthode une fois celle-ci déclarée. JavaScript en est un cas patent, bien qu'il s'agisse plus d'un langage fonctionnel se prêtant à certaines tactiques OO que d'un langage OO à proprement dit;
- d'autres encore permettent de déclarer et de définir des classes dites *partielles*, par morceaux placés dans des fichiers différents (comme par exemple C#), dans le but de laisser par exemple certains fichiers sous le contrôle d'un éditeur visuel (pour les fenêtres) et les autres sous le contrôle des programmeuses et des programmeurs.

Pourquoi encapsuler?

Rôle de cette section

Au-delà des grands principes, pourquoi met-on tant d'efforts à procéder à l'encapsulation (en particulier, l'encapsulation des attributs) d'un objet? Cette section vise à amener à une réflexion éclairée sur cette question.

Pourquoi utiliser des méthodes pour accéder aux attributs si on ne fait pas vraiment de contrôle d'accès sur ces attributs? Pour clarifier la question, examinons les deux programmes ci-dessous :

P0 .cpp	P1 .cpp
<pre>#include "Rectangle.h" int main() { Rectangle r; int hauteur = r.GetHauteur(); r.SetHauteur(hauteur* 2); }</pre>	<pre>#include "Rectangle.h" int main() { Rectangle r; int hauteur = r.hauteur_; r.hauteur_ = hauteur * 2; }</pre>

Mis à part l'objectif fort louable de faire plaisir à son professeur, ce qui est en soi louable bien sûr, existe-t-il de véritables raisons de préférer celui de gauche à celui de droite? Pourquoi ne pas spécifier les attributs publics, tout simplement?

En fait, il y a un tas de raisons pour préférer celui de *gauche*. En voici quelques-unes.

La qualité dans le détail, ou comment tirer profit de cette section

La différence en apparence mineure entre les deux programmes ci-dessus est des plus trompeuses, du fait que l'une des deux versions représente, à moyen et à long terme, de grandes économies en temps de développement et d'entretien de code. Les sections qui suivent cherchent d'ailleurs à expliquer pourquoi.

Pour tirer au maximum profit de ce qui suit, il est suggéré de porter deux chapeaux à la fois, ou du moins en alternance rapide :

- celui de l'étudiant(e) en POO, qui cherche à comprendre les diverses ramifications de l'emploi systématique de techniques d'encapsulation, qui se demande pourquoi on insiste spécialement sur l'importance de ces petites méthodes qui assurent un contrôle serré sur les membres privés d'un objet; et
- celui du/ de la chef de projet, qui voudra que son entreprise (son département, son service, son groupe) soit en santé, travaille de manière efficace et productive, n'ayant pas à recommencer plusieurs fois la même tâche et étant (autant que faire se peut) protégé contre les inévitables changements technologiques et philosophiques auxquels elle devra faire face.

Le premier chapeau vise à faire prendre conscience des enjeux sur une base personnelle, alors que le second tient au fait que le/ la chef de projet devra être en mesure de justifier et de défendre ses positions face aux spécialistes qui œuvreront pour lui/ pour elle.

Possibilité de contrôle d'accès

Il est vrai que dans la version courante de notre classe `Rectangle`, il n'y a pas de contrôle fait sur l'accès au membre `hauteur_`. Toutefois, *rien ne nous empêche d'en décider autrement un jour ou l'autre*.

Par exemple, on pourrait (et ce serait fort raisonnable) décider de ne pas accepter toute demande de changement à la hauteur d'un `Rectangle` qui ferait en sorte que celle-ci devienne inférieure ou égale à zéro.

La méthode qui devrait alors être modifiée serait le mutateur `SetHauteur()`, qui deviendrait telle que proposé à droite.

```
void Rectangle::SetHauteur(int hauteur) {
    if (hauteur > 0)
        hauteur_ = hauteur;
}
```

Ceci assurerait que, *si la hauteur initiale d'un `Rectangle` est valide*²⁸, aucune tentative de changement à cet attribut ne pourrait faire en sorte qu'il devienne invalide.

Si on prend les deux programmes à droite, très similaires à ceux examinés plus haut, on voit sans peine les conséquences peuvent alors être sérieuses.

Réflexion 00.3 – Le mutateur `SetHauteur()` de `Rectangle`, à droite, protège l'instance de la corruption, mais serait-il sage d'avoir recours à cette écriture en pratique? (voir **Réflexion 00.3 : des correctifs silencieux?**).

P0.cpp

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using std::cin;
    Rectangle r;
    int haut;
    if (cin >> haut)
        r.SetHauteur(haut);
}
```

P1.cpp

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using std::cin;
    Rectangle r;
    int haut;
    if (cin >> haut)
        r.hauteur_ = haut;
}
```

Dans la version de gauche, utilisant la méthode `SetHauteur(int)`, le `Rectangle` nommé `r` est protégé contre une donnée erronée, comme par exemple, par l'entrée d'une largeur négative. Dans la version de droite, l'accès direct à l'attribut représentant la hauteur empêche la classe d'assurer une telle protection.

Plus encore, dans la version de gauche, l'utilisateur de la classe `Rectangle` n'a pas à connaître les détails d'implantation pour utiliser la classe: il demande à l'objet de se modifier avec l'assurance, si le code est bien écrit, que sa demande sera rejetée par l'objet si elle est inacceptable. La version de droite, par contre, demande à l'utilisateur de connaître les détails d'implantation de l'objet, afin de ne pas commettre d'impair dans son utilisation.

²⁸ La question d'assurer un état initial convenable pour tout objet est un dossier à part entière, qui sera sous peu au cœur de nos préoccupations.

On pourrait protéger la hauteur de `r` sans avoir à passer en tout temps par la méthode `SetHauteur(int)`, mais il y a un prix à payer pour cela, et ce prix est d'ajouter le code servant à la protection de la donnée lors de chaque accès pouvant la modifier :

P0.cpp	P1.cpp
<pre>#include "Rectangle.h" // etc. int main() { Rectangle r; int hauteur; if (cin >> hauteur) r.SetHauteur(hauteur); }</pre>	<pre>#include "Rectangle.h" // etc. int main() { Rectangle r; int hauteur; if (cin >> hauteur && hauteur > 0) r.hauteur_ = hauteur; }</pre>

Ceci n'est pas une option viable à long terme. Les risques d'oublier de protéger l'un ou l'autre des accès sont grands, et la redondance dans la protection entraîne un effet secondaire pervers. Sans compter que cela force une répétition du code de validation partout dans le programme (une perte de ressources non négligeable!).

Un objet se veut intègre et responsable; qui voudrait dépendre des autres (innombrables et *a priori* inconnus) pour garantir sa propre intégrité?

Un de ces effets pervers pourrait être que le critère de validité change. Par exemple, on décide qu'il devient acceptable d'avoir des rectangles de hauteur négative, mais pas de hauteur nulle (on peut imaginer de tels cas).

L'emploi systématique d'un mutateur tel que `SetHauteur()` fait d'un tel changement une tâche mineure, puisque la validation se fait à même la classe; la manière de faire illustrée à droite fait d'un tel changement un cauchemar d'entretien : la validation se trouve alors répartie dans le code client et s'assurer qu'elle soit à la fois présente et cohérente partout où l'objet sera utilisé est, en pratique, impossible.

Question de couplage

En pratique, on souhaite réduire au minimum le couplage entre les objets. Par *couplage*, on entend des dépendances qui font en sorte que modifier un objet implique aussi en modifier d'autres.

La réduction de la surface publique des objets, sur laquelle nous avons beaucoup mis l'accent déjà, s'inscrit dans cette démarche. Un couplage plus faible signifie un entretien moins coûteux et des mises à jour comportant moins de ramifications néfastes.

Si le code client doit connaître les règles de validation des objets qu'il manipule, comme dans `P1.cpp` ci-dessus, changer une règle de validation aura un impact sur l'ensemble du code client de la classe, qui peut se trouver répandu partout sur la planète. En retour, localiser dans l'objet les règles de validation et d'intégrité localise aussi les changements, réduisant le douloureux et coûteux couplage.

Parenthèse technique – validation des entrées

Vous avez peut-être remarqué des exemples de la section précédente que la validation des entrées en C++ prend une forme particulière. En effet, reprenons un extrait du précédent exemple :

```
// ...
if(cin >> hauteur && hauteur > 0)
    r.SetHauteur(hauteur);
// ...
```

Cet extrait se lit comme suit : « si on parvient à lire `hauteur` et si `hauteur` est plus grande que zéro ». En effet, l'expression suivante :

```
cin >> val
```

...se teste comme un booléen et est vraie seulement si la lecture sur `cin` fut un succès. Il est donc nettement préférable d'écrire

```
if (cin >> val && val > 0)
    // ...
```

...plutôt que d'écrire

```
cin >> val;
if (val > 0)
    // ...
```

...car la première forme valide la lecture et la qualité de l'intrant, alors que la seconde pose problème dans le cas où la lecture fut erronée. Par exemple, imaginez la situation suivante :

```
int n;
cin >> n;
if (n > 0)
    cout << n << endl;
```

Si un usager entre le texte "patate" plutôt qu'un entier dans ce cas, la lecture de `n` aura échoué et la variable `n` demeurera non-initialisée, une situation pour le moins périlleuse (en C++, on parle alors de comportement indéfini [hdUB], ou *Undefined Behaviour*, ce qui signifie essentiellement que le programme entier devient invalide dû à cette opération malvenue). Bien entendu, si nous écrivons plutôt ceci :

```
int n;
if (cin >> n && n > 0)
    cout << n << endl;
```

...alors l'utilisation de `n` dépendra à la fois de son initialisation et de sa validation, ce qui est clairement préférable.

Séparation de l'interface et de l'implémentation

Rôle de cette section

Séparer l'interface de l'implémentation est l'une des principales clés du succès dans le développement informatique en général. Nous examinons ici ce qu'il en retourne de cette stratégie dans un contexte OO.

Un objet, à la manière d'un humain, d'une armée ou d'une voiture, dévoile (on dira aussi *expose*) publiquement certaines choses, et en conserve d'autres pour lui.

Le volet **public** d'un objet est souvent appelé son **interface**. C'est cet aspect de l'objet qui servira pour communiquer avec le monde extérieur, et qui lui permet d'offrir des services aux autres objets du même système.

Interface

L'interface d'un objet nous dit ce qu'on lui peut lui demander de faire, et comment on peut le lui demander.

Le terme « interface » a aussi une acception technique figée dans certains langages OO; ici, nous utilisons ce terme au sens large.

L'interface comprend à la fois les méthodes de l'objet et les opérations sur l'objet qui accompagnent la déclaration de la classe (nous y reviendrons). Le volet public étant ce qui permet à d'autres de communiquer avec l'objet, il importe que ce volet soit aussi *stable* et *définitif* que possible.

On peut ajouter des éléments à l'interface d'une classe avec le passage du temps (le nombre de méthodes publiques d'une classe peut croître) mais on ne devrait pas modifier une interface déjà publiée²⁹ d'une manière pouvant nuire à toute utilisation déjà en cours de l'objet³⁰.

Tout ce qui est public dans un objet peut être considéré comme un contrat, que l'équipe de développement s'engage à respecter. Une fois l'objet livré, ce qui en est public doit être maintenu tel quel.

Interface = contrat

Il faut donc ne rendre public que ce qui devrait vraiment l'être; ce qui est considéré comme stable et à peu près définitif³¹.

²⁹ Ce qu'on entend par *publier une interface*, en C++ comme ailleurs, est (au sens large) *livrer l'objet pour fins d'utilisation*. Une fois cette étape franchie, ce qui est public dans l'objet *peut* être en cours d'utilisation dans un, deux, cent, mille projets distincts, et tout changement ultérieur apporté à l'interface de l'objet porte à conséquence dans chacun de ces projets. Spécifier un membre public est donc une responsabilité importante, qu'il ne faut pas prendre à la légère.

³⁰ Techniquement, même ajouter des méthodes à une classe après publication est délicat, puisque certains programmes clients pourraient prendre pour acquis des détails de bas niveau comme l'ordre d'apparition des méthodes. Ces problèmes surviennent rarement au niveau du code source des clients, mais peut survenir quand un objet est utilisé par une infrastructure commerciale qui fait le pont entre deux technologies (il en existe beaucoup). Ce couplage binaire, bien qu'il ne soit pas souhaitable, se produit en pratique.

³¹ Les technologies qui acceptent les changements d'interfaces (les services Web, en particulier) ont aussi des mécanismes de gestion du changement sophistiqués et paient le prix de cette souplesse par une réduction des seuils de performance. Les objets, en général, doivent être performants, du moins dans les langages voués à des applications commerciales.

À la limite, on nommera aussi *interface* ce qui est révélé de la classe, ce qu'on retrouve dans sa déclaration, au sens où c'est ce fichier qui sera mis à la disposition des utilisateurs de la classe. Le mot révélé ici s'applique aux humains; pour les entités du système informatique dans lequel habite un objet, les clauses de protection comme `public` et `private` font la loi.

Il existe des techniques pour ne retrouver, dans cette section exposée aux yeux des programmeuses et des programmeurs, que l'interface au sens strict (les membres publics), et ces techniques mènent plusieurs langages OO à donner au mot `interface` un sens propre. Cette thématique s'explorera mieux dans la section sur le polymorphisme [POOv01] ou dans un cours portant sur les systèmes client/ serveur.

Le volet privé d'un objet fait partie de ce qu'on appelle son **implémentation**. L'implémentation de l'objet inclut aussi en général la définition de ses méthodes, ce qui peut être modifié localement sans que cela n'entraîne de modifications au code client.

Implémentation

Les détails d'implémentation indiquent comment l'objet est implanté, et comment il réalisera les tâches qu'on voudra lui confier.

En théorie, seul l'objet lui-même devrait être préoccupé par les questions d'implémentation. Dans la mesure où l'interface de l'objet ne change pas, tout changement apporté à son implémentation devrait être transparent³² pour qui l'utilise. C'est l'un des objectifs poursuivis par la réduction du couplage : réduire les coûts des mises à jour et des améliorations apportées à l'implémentation d'un objet et de ses services.

Tout ce qui est privé dans un objet peut être considéré comme sujet à changement; c'est en partie pourquoi on devrait éviter tout accès direct aux attributs d'un objet, pourquoi on insiste sur l'encapsulation.

Un cas intéressant est le nom de l'attribut `hauteur_`. S'agit-il d'une information publique, qu'on devrait s'engager à garder stable pour assurer le bon fonctionnement de l'objet à moyen et à long terme? La réponse, bien sûr, est négative : le choix d'ajouter au nom de l'attribut le suffixe `_` est arbitraire³³; l'emploi d'une nomenclature française pour cet attribut l'est aussi.

En fait, l'existence même d'un membre tenant à jour la hauteur du `Rectangle` est un choix d'implémentation. On pourrait penser à plusieurs autres stratégies, incluant celle de calculer la hauteur du `Rectangle` par la valeur absolue de la différence entre les ordonnées de deux de ses coins³⁴, qui seraient tout aussi valides que celle maintenant à jour un attribut.

Les choix de représentation dans un objet ne constituent habituellement **pas** une information d'ordre public.

³² On pourra bien sûr voir une amélioration de son fonctionnement, mais pas un changement de nature.

³³ Voir *Appendice 02 – Nommer et documenter* pour une discussion des questions de notation, sur le plan historique comme sur le plan des usages contemporains.

³⁴ Question piège : dans quelles situations privilégierait-on l'option de calculer la hauteur sur demande à celle selon laquelle on conserve en tout temps la hauteur dans un attribut? La réponse, de même qu'une discussion de considérations connexes, dans la section *Réflexion 00.4 : choix de représentation*.

Accesseurs de second ordre

Tel que nous l'avons défini jusqu'ici, tout `Rectangle` sait dévoiler sa hauteur et sa largeur. Il semble raisonnable de vouloir laisser le `Rectangle` nous dévoiler aussi son périmètre et son aire (sa surface).

Il n'est pas nécessaire pour y arriver que le `Rectangle` tienne à jour un attribut spécifiant chacune de ces données; il peut aisément les calculer sur demande. Malgré tout, rien n'empêcherait un `Rectangle` de tenir à jour un attribut dans chaque cas. On ne doit pas, de l'extérieur, présupposer de choses sur son implémentation : il s'agit, pour le `Rectangle`, d'information privée.

Ajoutons donc à notre `Rectangle` une méthode `GetAire()` et une méthode `GetPerimetre()`. Les fichiers `Rectangle.h` et `Rectangle.cpp` modifiés sont visibles plus loin, mais essayez d'écrire les méthodes vous-mêmes avant d'aller voir la solution.

Pourquoi loger, à même l'objet, des accesseurs de second ordre?

On aurait tendance à se demander pourquoi un objet devrait lui-même implanter les accesseurs de second ordre, comme la méthode `GetPerimetre()` d'un rectangle. La raison est que *l'objet lui-même tend à être le meilleur pour implanter correctement les opérations qui le concernent*. L'objet seul sait comment il est constitué, et comment le calcul du périmètre pourrait être le plus efficace étant donné cette constitution.

Vous trouverez une discussion un peu plus poussée – et, surtout, plus nuancée – de cette thématique dans *Appendice 03 – Dans l'objet ou hors de l'objet?*

En effet, peut-être chaque `Rectangle` garde-t-il seulement en note, à l'aide d'attributs, sa hauteur et sa largeur, et se sert-il de ceux-ci pour calculer son périmètre... mais il est aussi possible que chaque `Rectangle` garde à jour son périmètre et son aire, les modifiant à chaque réajustement de sa hauteur ou de sa largeur. Ces données étant privées, seul le `Rectangle` lui-même peut, en toute confiance, prétendre en tirer pleinement profit.

Chaque objet cherchant à encapsuler le détail de son implémentation, il devient *de facto* responsable de la bonne implantation des opérations qui lui sont associées. Chaque objet devrait être le plus grand spécialiste des opérations à accomplir sur lui et lui seul.

Ceci n'est pas un dogme absolu, remarquez bien. Il peut y avoir des cas justifiables où, en toute connaissance de cause, on voudra qu'un algorithme contourne cette règle; on ne peut pas espérer toujours tout prévoir. Mais c'est une bonne pratique que de s'en remettre d'abord à l'objet pour les opérations, élémentaires ou non, qui lui sont propres, et d'exiger une forme de justification écrite (au moins un commentaire) pour les quelques cas où on pourra démontrer l'avantage de contourner cet usage.

C'est là un principe à garder en tête : les règles et pratiques ont leur raison d'être, et devraient en général être respectées. Toute situation où on veut contourner une règle reconnue devrait être accompagnée d'une justification écrite, de manière à ce qu'il soit possible de réexaminer la situation dans le futur, et pour que les raisons derrière le geste demeurent claires à long terme.

Réflexion 00.4 – Est-il avantageux de garder à jour l'aire et le périmètre d'un `Rectangle` à l'aide d'attributs? Quel serait le prix à payer? Quel est le choix optimal? Expliquez votre position (voir *Réflexion 00.4 : choix de représentation* pour une discussion à ce sujet).

Une fois les fichiers modifiés, on obtient ce qui suit (les ajouts sont en caractères gras).

Rectangle.h	Rectangle.cpp
<pre> #ifndef RECTANGLE_H #define RECTANGLE_H class Rectangle { public: // -- Accesseurs de premier ordre int GetHauteur() const; int GetLargeur() const; // -- Accesseurs de second ordre int GetAire() const; int GetPerimetre() const; // -- Mutateurs de premier ordre void SetHauteur(int); void SetLargeur(int); private: int hauteur_, largeur_; }; #endif </pre>	<pre> #include "Rectangle.h" int Rectangle::GetHauteur() const { return hauteur_; } int Rectangle::GetLargeur() const { return largeur_; } int Rectangle::GetAire() const { return GetLargeur() * GetHauteur(); } int Rectangle::GetPerimetre() const { return (GetLargeur() + GetHauteur()) * 2; } void Rectangle::SetHauteur(int hauteur) { hauteur_ = hauteur; } void Rectangle::SetLargeur(int largeur) { largeur_ = largeur; } </pre>

Remarquez que même à l’interne, ce code utilise *autant que possible* les accesseurs de bas niveau (le même principe s’appliquera aux mutateurs). C’est voulu : en diminuant le nombre de lieux par lesquels on peut accéder aux attributs, on est plus en mesure d’identifier les endroits à optimiser dans le code. Lorsque des bogues surviennent, conséquemment, on diminue le nombre d’endroits susceptibles de les abriter.

On a ici, avec `GetAire()` et `GetPerimetre()`, deux bons cas de méthodes pouvant être écrites sans peine, même si elles ne sont pas sollicitées dans le programme en cours de rédaction. Elles ne coûtent à peu près rien en fait d’espace, et qui sait à qui elles pourront servir?

Autres méthodes

Les méthodes d'un objet ne se limitent pas à des accesseurs et à des mutateurs. La gamme de services offerte par un objet donné doit correspondre aux tâches qu'un client peut raisonnablement souhaiter lui demander de réaliser.

Dans le cas du `Rectangle`, pris au sens du problème énoncé en début de document, l'un des objectifs est de bien pouvoir le dessiner... il est donc raisonnable d'envisager y exposer la méthode `Dessiner()`.

L'ajout à notre déclaration de classe de cette méthode nous donnera ce qu'on peut voir à droite.

La spécification `const`, comme toujours, offre la garantie que dessiner un `Rectangle` ne le modifiera d'aucune façon. Seules des méthodes spécifiées `const` y seront utilisées : pour réaliser sa tâche, `Dessiner()` pourra avoir recours à `GetLargeur()`, qui est `const`, mais pas à `SetLargeur(int)`.

```
class Rectangle {
public:
    // -- Accesseurs de 1er ordre
    int GetHauteur() const;
    int GetLargeur() const;
    // -- Accesseurs de 2e ordre
    int GetAire() const;
    int GetPerimetre() const;
    // -- Mutateurs de 1er ordre
    void SetHauteur(int);
    void SetLargeur(int);
    // -- Autres méthodes
    void Dessiner() const;
private:
    int hauteur_,
        largeur_;
};
```

L'ajout à notre définition de classe de cette méthode nous donnera :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
#include <iostream>
using namespace std;
//
// Définition de GetHauteur(), GetLargeur(), SetHauteur(const int) et
// SetLargeur (const int), inchangées. Omises pour épargner de l'espace
//
void Rectangle::Dessiner() const {
    const int LARGEUR = GetLargeur(),
            HAUTEUR = GetHauteur();
    for (int i = 0; i < HAUTEUR; i++) {
        for (int j = 0; j < LARGEUR; j++) {
            cout << "*";
        }
        cout << endl;
    }
}
```

Remarquez que, puisque la méthode `Dessiner()` telle que définie dans `Rectangle.cpp` utilise `std::cout` et `std::endl`, il faut inclure la bibliothèque `<iostream>` dans ce fichier.

Avec la classe `Rectangle` telle que nous l'avons maintenant, il devient simple d'écrire un programme qui créera un `Rectangle` de hauteur 5 et de largeur 10, puis le dessinera.

Le programme principal n'a pas à inclure `<iostream>`: la manière par laquelle le `Rectangle` se dessinera relève du `Rectangle` seul.

```
#include "Rectangle.h"
int main() {
    Rectangle r;
    r.SetLargeur(10);
    r.SetHauteur(5);
    r.Dessiner();
}
```

Nous examinerons un peu plus loin d'autres stratégies pour réaliser une tâche semblable. La pensée OO n'est pas une pensée unique; elle ouvre plus d'une porte et se prête à une multitude d'interprétations. Nous prendrons soin d'explorer des techniques OO plus sophistiquées en temps et lieu.

Encapsulation, mutateurs et validation

Nos mutateurs ont jusqu'ici été bien naïfs, au sens où bien qu'ils introduisent un lieu dans la classe `Rectangle` pour prévenir la corruption des états de l'objet, nous n'avons encore rien fait pour en tirer profit.

Supposons l'implémentation simpliste proposée dans la section précédente, alors quel sens donnerions-nous au programme suivant?

```
#include "Rectangle.h"
int main() {
    Rectangle r;
    r.SetLargeur(-8);
    r.SetHauteur(0);
    r.Dessiner(); // ?
}
```

Ici, dû à l'implémentation choisie pour `Rectangle::Dessiner()`, le programme n'afficherait rien, ce qui pourrait rendre perplexe mais ne causerait pas de dégâts outre mesure. Bien entendu, ce programme est simple, pour ne pas dire simpliste, et nous ne serons pas toujours aussi chanceux dans des systèmes de grande envergure.

Notre `Rectangle` devrait en fait refuser qu'on lui appose une largeur ou une hauteur invalide, le mot « invalide » étant ici pris au sens de « ne respectant pas les règles d'affaire de la classe ».

La question de la validation des tentatives de mutation des états d'un objet est complexe, et nous y reviendrons dans *Traitement d'exceptions* plus loin. D'ici là, nous procéderons selon la philosophie suivante : *si un objet devait être corrompu, alors le programme ne serait plus valide, alors mieux vaut planter immédiatement et permettre un diagnostic.*

En ce sens, ce que nous ferons si une valeur incorrecte est passée à un mutateur sera de lever une exception, donc d'envoyer un signal au code client à l'effet qu'un problème grave est survenu. La saine gestion d'un tel signal sera traitée dans une autre section du présent document.

Pour nos besoins, donc, nous remplacerons le code de gauche par le code de droite ci-dessous :

```
#include "Rectangle.h"
// ...
void Rectangle::SetHauteur(int hauteur) {
    hauteur_ = hauteur;
}
// ...
```

```
#include "Rectangle.h"
#include <exception>
// ...
void Rectangle::SetHauteur(int hauteur) {
    if (hauteur <= 0)
        throw std::exception{"Hauteur incorrecte"};
    hauteur_ = hauteur;
}
// ...
```

Conséquemment, un programme comme le `main()` plus haut plantera plutôt que de nous laisser perplexes devant un écran noir, et on ne pourra pas corrompre un `Rectangle`.

Objets et contraintes de projet

Rôle de cette section

Nous pousserons ici notre réflexion de design pour inclure une distinction entre ce qui est propre à chaque instance d'une classe et ce qui appartient, de manière commune, à toutes les instances d'une même classe.

Revenons à l'application permettant de dessiner des formes, décrite dans la section *Mise en contexte* au début de ce document.

Membres de classe ou membres d'instance

On remarquera que l'utilisateur devra entrer une hauteur, devant être validée entre 1 et 20 inclusivement, et que ce même usager devra aussi entrer une largeur devant être validée entre 1 et 50 inclusivement. Il s'agit bel et bien là de **constantes**. Ces constantes seront propres à ce qu'est, pour nous, un `Rectangle`. Ces constantes sont des états propres à `Rectangle`, et par conséquent des **attributs**³⁵ au sens usuel du terme.

Contrairement aux attributs rencontrés jusqu'ici, ces constantes auront la même valeur quel que soit le `Rectangle`. En effet, pour prendre un exemple simple, la hauteur minimale sera 1 pour tout `Rectangle` dans notre système, quel qu'il soit.

Il se trouve que les attributs tels que nous les avons examinés jusqu'ici sont ce qu'on appelle des **attributs d'instance**, ceci parce que *chaque instance a une copie distincte de chacun de ces attributs*.

*Attributs
d'instance*

Par exemple, tout `Rectangle` respectant les spécifications que nous avons déclarées a son propre attribut `hauteur_`. Ainsi, considérant dix instances de `Rectangle`, chacune d'elles *pourrait* avoir une hauteur différente.

La question qui se pose est la suivante : est-il raisonnable d'avoir une copie distincte de la constante donnant la hauteur minimale d'un `Rectangle` dans chaque instance de `Rectangle` en circulation dans un programme?

Cette question a du sens, même d'un point de vue économique. Si on a un million d'instances de `Rectangle` actives dans un programme, le prix à payer pour un choix mal avisé pourrait devenir fort élevé³⁶.

³⁵ Certaines écoles de pensée prétendraient autrement, par exemples celles orientées surtout sur les fonctions qui maintiennent parfois qu'une constante est, au fond, une fonction retournant toujours la même valeur, mais il est clair que, dans le contexte de notre programme, cet énoncé peut être considéré vrai.

³⁶ Si on présume que la hauteur minimale est un `int`, le coût en espace mémoire serait alors $1'000'000 * \text{sizeof}(\text{int})$, pour une donnée *garantie comme étant toujours identique*, puisque constante. C'est un coût considérable, qu'il convient d'éviter. Les programmes jouets peuvent négliger ces détails, mais les programmes commerciaux ne le peuvent absolument pas.

Le langage C++, tout comme Java, C#, VB.NET et bien d'autres, permet de déclarer des *membres d'instance* et des *membres de classe*.

Rappel : nous entendons par *membre* l'ensemble fait à la fois des attributs et des méthodes.

Les **membres d'instance** sont ceux dont on trouve une copie distincte dans chaque instance de la classe.

Membres d'instance

Pensez bien sûr à hauteur_, pour que chaque instance de Rectangle connaisse sa propre hauteur, mais aussi à GetHauteur() et à SetHauteur(const int) pour que chaque Rectangle puisse nous donner sa hauteur et qu'on puisse modifier sa hauteur.

Par défaut, dans la déclaration d'une classe, un membre, qu'il s'agisse d'une méthode ou d'un attribut, sera un membre d'instance.

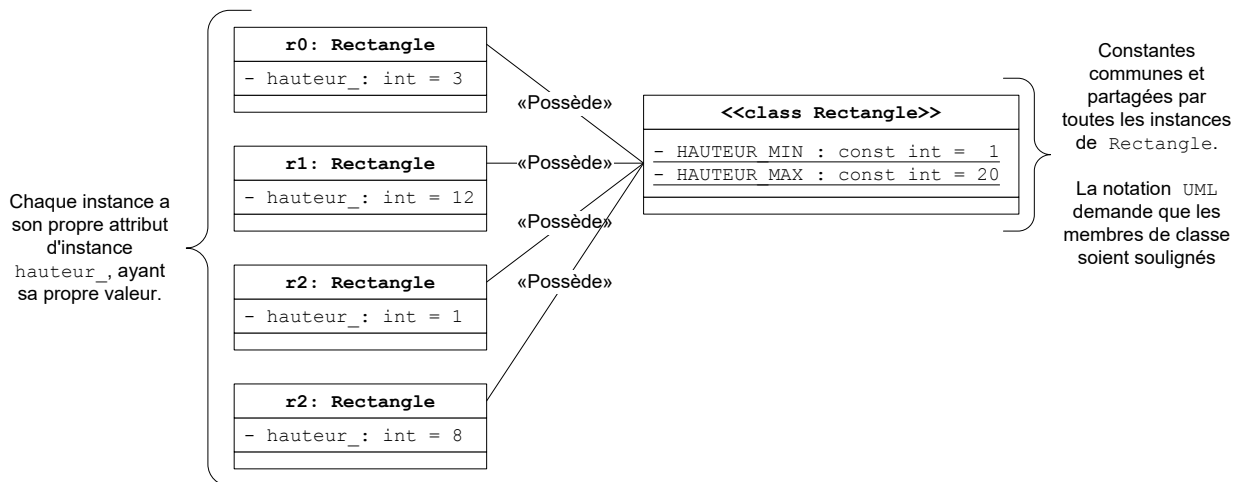
Les **membres de classe** sont ceux dont on ne trouve qu'une seule copie pour la classe en entier.

Membres de classe

Un candidat logique à ce titre serait une constante devant avoir la même valeur *quelle que soit l'instance qui y a recours*. Il y a aussi des cas typiques pour lesquels on voudra avoir recours à des attributs de classe qui seront des variables plutôt que des constantes.

On verra aussi, ci-dessous, des cas où il est intéressant d'avoir recours à des méthodes de classe, et on expliquera les contraintes auxquelles ces méthodes sont soumises.

Pour qu'un membre soit un membre de classe en C++, en C# et en Java, il faut lui apposer la spécification **static**. En VB.NET, le mot clé utilisé à cette fin est **Shared**. Notons qu'en C#, tout primitif constant est implicitement static (on les qualifie const plutôt que static const comme en C++).



Cas concret : hauteur minimale et hauteur maximale d'un Rectangle

Pour ajouter à la classe `Rectangle` des constantes pour indiquer la hauteur minimale et la hauteur maximale qui lui sont acceptables, on procédera comme suit.

Tout d'abord, dans la déclaration de la classe (visible à droite), on ajoutera deux attributs portant les spécifications `static` (pour dire qu'il s'agit d'un membre de classe) et `const` pour indiquer qu'il s'agit d'une constante.

Remarquez qu'on n'initialise pas immédiatement la constante. En effet, notre ajout spécifie qu'il *existe* une constante privée `HAUTEUR_MIN` et une constante privée `HAUTEUR_MAX` appartenant toutes deux à la classe `Rectangle`; cette affirmation de leur existence n'est pas une déclaration à proprement dit.

Ensuite, dans le fichier où on trouve la définition de la classe (dans `Rectangle.cpp`), on déclare (et on initialise) les attributs de classe.

Remarquez d'ailleurs la syntaxe :

```
#include "Rectangle.h" // déclaration de la
                        // classe Rectangle
// ... autres inclusions ...
const int Rectangle::HAUTEUR_MIN = 1,
        Rectangle::HAUTEUR_MAX = 20;
// reste du fichier, définition des
// méthodes incluses
```

Les constantes d'une classe existent avant la première instanciation de cette classe. C'est pourquoi elles sont définies globalement.

Remarquez aussi l'absence du mot clé `static` lors de la *définition* officielle des constantes de classe. Dans la déclaration de la classe, ce mot indique que le membre est un membre de classe; par contre, dans la définition de la classe, le simple fait de spécifier le nom de la classe en préfixe au nom de l'attribut suffit pour indiquer qu'il s'agit d'un attribut de la classe.

```
class Rectangle {
public:
    // -- Accesseurs de premier ordre
    int GetHauteur() const;
    int GetLargeur() const;
    // -- Accesseurs de second ordre
    int GetAire() const;
    int GetPerimetre() const;
    // -- Mutateurs de premier ordre
    void SetHauteur(int);
    void SetLargeur(int);
    // -- Autres méthodes
    void Dessiner() const;
private:
    static const int HAUTEUR_MIN,
                    HAUTEUR_MAX;

    int hauteur_,
        largeur_;
};
```

Remarquez le choix de notation : le recours aux majuscules (c'est une constante entière : `const` et `int`), et pas de préfixe ou de suffixe particulier (le nom d'un membre de classe est qualifié par le nom de sa classe : `Rectangle::HAUTEUR_MAX`).

Cette qualification par le nom de la classe est un signe qu'il s'agit d'attributs appartenant à la classe `Rectangle`, pas à l'une ou l'autre de ses instances

Détail de notation

En vertu de la règle ODR (voir *Annexe 03 – Concepts et pratiques du langage C++* pour des détails), toute entité d'un programme C++ ne doit être définie qu'une fois, bien qu'elle puisse être déclarée plusieurs fois. La déclaration des constantes (dans le `.h`) sera vue par le code client de la classe `Rectangle`, mais la définition est cachée dans le fichier source (le `.cpp`), qu'on n'inclura pas, ce qui évitera de contrevenir à la règle.

Notez que C++ admet une exception pour les constantes entières, pour faciliter certaines manœuvres comme la déclaration de tableaux dans les objets. La forme proposée ici (déclaration dans le `.h`, définition dans le `.cpp`) est la seule qui fonctionne pour tous les types

Utilisation d'un membre de classe

Comment une méthode d'instance peut-elle utiliser un membre de sa classe?

De deux manières. La première est de spécifier le nom du membre en entier, incluant le nom de la classe (exemple à droite)...

```
// SetHauteur (int) avec validation
void Rectangle::SetHauteur(int hauteur) {
    if (hauteur >= Rectangle::HAUTEUR_MIN &&
        hauteur <= Rectangle::HAUTEUR_MAX)
        hauteur_ = hauteur;
    else
        throw std::exception{"Hauteur incorrecte"};
}
```

```
// SetHauteur (int) avec validation
void Rectangle::SetHauteur(int hauteur) {
    if (hauteur >= HAUTEUR_MIN && hauteur <= HAUTEUR_MAX)
        hauteur_ = hauteur;
    else
        throw std::exception{"Hauteur incorrecte"};
}
```

... alors que la seconde omet le nom de la classe, qui est pour cette instance quelque chose d'implicite.

⇒ Toute instance peut utiliser les membres de classe de sa classe comme s'ils lui appartenaient directement³⁷.

Une autre classe voulant accéder à un membre de classe public devra par contre spécifier le nom du membre en entier. Il en va de même pour un sous-programme quelconque, comme dans l'exemple qui suit :

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using std::cout;
    // NOTE: ce qui suit ne fonctionnera que si les constantes de classe utilisées
    // sont publiques (dans notre exemple plus haut, elles étaient privées)
    cout << "La hauteur d'un rectangle doit se situer inclusivement entre "
         << Rectangle::HAUTEUR_MIN << " et " << Rectangle::HAUTEUR_MAX;
}
```

Avec notre implémentation de `Rectangle`, ce programme serait illégal du fait que les constantes de classe ont été qualifiées `private` lors de leur déclaration.

Question : en prenant pour exemple la classe `Rectangle`, serait-il possible pour une méthode de classe de référer implicitement à `hauteur_`? Expliquez clairement votre réponse.

³⁷ Et, en fait, ils lui appartiennent. L'appartenance n'est simplement pas exclusive à l'instance.

Les accesseurs et les mutateurs constituent-ils un bris d'encapsulation?

L'idée d'accesseurs et de mutateurs présentée ici est une forme primaire d'encapsulation, envers laquelle des critiques sont fréquemment lancées. Ces critiques tiennent généralement au fait que des accesseurs et des mutateurs de premier ordre, donnant chacun un accès encapsulé à un attribut, forment en fait un bris d'encapsulation.

La raison de cette critique est la suivante :

- si, pour tout attribut `att_` de type `Type`, on présente un accesseur public constant `GetAtt()` de type `Type` retournant une copie de `att_`; et
- si on présente aussi un mutateur `SetAtt()` prenant un paramètre de type `const Type` ou `const Type&` et affectant directement la valeur du paramètre à `att_`; alors
- on vient de complexifier légèrement un accès direct à `att_`, sans plus.

Conséquemment, sous cette optique, l'attribut n'est encapsulé qu'en apparence. L'objet, en appliquant bêtement l'encapsulation par une paire accesseur/ mutateur, révèle au monde extérieur une partie de sa structure interne, donc un détail d'implémentation.

C'est une critique qui a du vrai, mais qui est mal ciblée. Examinons pourquoi elle manque un peu sa cible, mais aussi comment bien appliquer le principe d'encapsulation dans un contexte réel. Ensuite, nous amorcerons (sans la compléter) une réflexion plus pointue sur la gamme de services que devrait offrir un objet.

```
class X {
    Type att_;
public:
    Type GetAtt() const;
    void SetAtt(const Type&);
};
// ...
Type X::GetAtt() const {
    return att_;
}
void X::SetAtt(const Type &a) {
    att_ = a;
}
```

Le discours que nous tiendrons ici demandera à être raffiné un peu plus tard. Comme c'est souvent le cas, nous pêcherons tout d'abord par rigueur et par souhait de robustesse, puis nous aborderons les bémols lorsque ceux-ci s'imposeront à nous.

Encapsulation, intégrité et invariants

L'encapsulation est un principe, et c'est un principe important. Sa raison d'être est double.

Encapsulation et intégrité

En premier lieu, on applique l'encapsulation pour **permettre à un objet d'assurer sa propre intégrité, de sa construction à sa destruction.**

Forcer toute modification³⁸ des attributs d'un objet à passer par un même lieu, par une même méthode, permet à l'objet de mettre en application ce principe de manière stricte, surtout si l'objet est rigoureux et s'il utilise lui-même ces mécanismes de contrôle d'accès.

Dans la littérature scientifique, on décrit un **objet** comme une **entité garantissant des invariants**. En ce sens, il est possible d'affirmer des faits sur un objet qui s'avéreront toujours *entre deux appels successifs de méthodes*, mais pas nécessairement durant l'exécution d'une méthode, puisque durant cette période il est possible qu'un objet change d'état. Qu'un objet soit garant de sa propre intégrité signifie qu'il soit garant de ses invariants.

À titre d'exemple, supposons la classe `Nom` représentant un nom et offrant les services `vide()`, retournant `true` seulement si le nom est vide, et `longueur()`, retournant la longueur du nom en nombre de caractères.

Si l'un des invariants de cet objet est que pour toute instance `n` de `Nom`, `n.vide() ⇔ n.longueur() == 0`, alors il devrait être impossible, de l'extérieur des services de `Nom`, en tout temps entre la fin de la construction et le début de la destruction de `n`, que `n.vide() && n.longueur() > 0`; assurer cette garantie est la responsabilité de l'instance `n`.

Localiser les accès aux attributs en un seul endroit de manière systématique et rigoureuse a aussi l'effet secondaire de limiter les impacts de tout changement structurel apporté à un objet. Si même l'objet en soi passe toujours par ses accesseurs et ses mutateurs pour consulter ou modifier ses attributs, alors changer le type d'un attribut de cet objet aura un effet *extrêmement local*. Ceci entraînera une diminution importante des coûts d'entretien de tout système utilisant cet objet.

Dans un livre célèbre [ArtMPP], *Maurice P. Herlihy* présente l'encapsulation en tant que garante de l'intégrité des objets comme une forme de garantie de non-interférence (*Freedom from Interference*), ce qui sied aussi fort bien à ce que nous expliquons ici.

Évolution d'un objet vs évolution de son interface

Déjà sous cette lueur, on voit que la critique mentionnée plus haut est à prendre avec un grain de sel : puisque changer le type d'un attribut doit être une opération transparente pour le monde extérieur à l'objet, il est impératif que les accesseurs et les mutateurs déjà existants, ou qu'une strate de protection équivalente, soient maintenus³⁹.

Ceci mène nécessairement, avec l'évolution des objets, à une *disparité de type* entre certains mutateurs et certains accesseurs de premier ordre et les attributs pour lesquels ils offrent une interface d'encapsulation primitive.

On le voit, le volet de la critique prêtant à la stratégie d'encapsulation primitive par accesseurs et mutateurs de premier ordre la propriété de briser l'encapsulation d'un objet est un volet fondamentalement floué dans un contexte de code de production.

³⁸ ... et il en va de même pour toute consultation.

³⁹ Et s'il advient qu'un changement de type apporté à un attribut soit si drastique que maintenir son interface de base soit devenu impossible (qu'on ne puisse plus maintenir les accesseurs et les mutateurs de premier ordre existants sur cet attribut), alors on parle vraiment d'un nouvel objet, opérationnellement différent de l'original. On a donc besoin d'une nouvelle classe.

Interface d'encapsulation secondaire

Notons aussi au passage qu'il est normal pour un objet d'offrir plusieurs accesseurs et plusieurs mutateurs dérivés des accesseurs et des mutateurs de premier ordre. Pensons par exemple à une méthode d'instance `GetPerimetre()` pour un `Rectangle` muni des attributs `largeur_` et `hauteur_`.

Ainsi, l'objet pris au sens opérationnel⁴⁰ offrira normalement une interface beaucoup plus riche que son interface d'encapsulation primitive. La critique omet le volet secondaire de l'encapsulation, qui est pourtant très près du volet primitif.

Encapsulation et opérabilité

En second lieu, on applique l'encapsulation pour que **l'objet soit seul responsable de dévoiler l'interface opérationnelle de son choix**. Un objet est conçu pour être manipulé poliment, avec délicatesse. Il nous offre par ses méthodes l'éventail des possibilités qu'il accepte d'offrir, point à la ligne. Pour le monde extérieur à l'objet, il y a adéquation entre l'objet et ce qu'il sait faire⁴¹.

C'est là qu'on voudra se poser des questions de design et toucher au point le plus pertinent de la critique susmentionnée : quelles sont les opérations qu'on voudra dévoiler au monde extérieur? Voudra-t-on nécessairement dévoiler tous les accesseurs et tous les mutateurs de l'objet? Se limitera-t-on à des interfaces aussi primitives?

En général, la réponse à ces deux questions sera négative. Un objet peut fort bien déclarer des mutateurs privés, destinés à son seul usage, et limiter son interface publique à des méthodes plus sophistiquées. Pensez entre autres à l'exemple du compte bancaire, plus haut, où le mutateur proposé était privé et où l'interface était constituée d'un accesseur primitif et des méthodes `Deposer()` et `Retirer()`. Le mutateur privé est une sorte de dernier rempart, centralisant la validation du respect des garanties d'intégrité de l'objet.

Qu'on présente ou non les interfaces d'encapsulation primitives des objets au monde extérieur variera selon les classes. Dans la plupart des cas, on révélera un sous-ensemble de ces opérations primitives pour une classe donnée. La véritable question à se poser n'est donc pas « l'objet devrait-il avoir une interface primitive d'encapsulation? » mais bien « quelles sont les opérations de cet objet qui devraient être révélées au monde extérieur, qui devraient faire partie de son visage public? ».

Répondre à cette question nous indique lesquelles des opérations d'un objet seront publiques, et lesquelles seront privées. Ce dont il faut se souvenir, c'est que l'interface primitive d'encapsulation d'un objet ne sert pas nécessairement à des fins publiques.

⁴⁰ Un objet ne devrait d'ailleurs être pris qu'au sens opérationnel dans l'immense majorité des cas; s'il est pris au sens structurel, sauf dans le contexte de son développement propre, alors on a un *réel* bris d'encapsulation.

⁴¹ **Confucius** aurait peut-être pu écrire *tu es ce que tu fais* à ce sujet.

Stratégies alternatives

Présumons qu'on veuille exposer, dans un objet, une méthode pour consulter une donnée et une méthode pour la modifier. La question d'utiliser ou non les préfixes `Set` et `Get` pour modifier ou consulter une donnée d'un objet, qu'un attribut existe ou non pour représenter cette donnée, est un choix esthétique.

On pourrait privilégier d'autres options. Par exemple, celle d'utiliser un nom rappelant le nom de la donnée désirée dans les deux cas.

Un exemple d'une telle stratégie est proposé à droite⁴². Remarquez par exemple que pour consulter la hauteur d'un `Rectangle` nommé `r`, on utiliserait la version `const` de la méthode `Hauteur()` :

```
cout << r.Hauteur();
```

alors que pour modifier sa hauteur, on utiliserait la version `non-const` de cette méthode (qui retourne une référence au même attribut) :

```
r.Hauteur() = 10;
```

Constatons qu'ici, il y a un vrai bris d'encapsulation, du fait que la version retournant une référence à la hauteur du `Rectangle` donne *de facto* un accès direct à l'attribut `hauteur_`. On pourrait noter, dans la méthode, qu'un accès direct a été fait à l'attribut, *mais on ne pourrait pas ici valider cet accès*.

Une telle option apporte moins, du point de vue de l'encapsulation, que la stratégie utilisant un couple accesseur/mutateur classique. On pourrait corriger le bris d'encapsulation mais conserver l'esthétique de l'alternative ci-dessus en remplaçant ceci...

```
void Rectangle::Hauteur(int hauteur) {
    hauteur_ = hauteur;
}
```

...par cela. On obtient alors une solution différente de celle constituée par le couple `Set/Get` sur le plan cosmétique, mais identique sur le plan opérationnel.

Il existe une technique avancée qui permet de contourner ce problème en retournant non pas une donnée d'un type primitif comme `int` mais bien un objet à part entière. Si vous êtes imaginative ou imaginatif, le présent document comprend toute l'information requise pour arriver à concevoir une telle solution.

```
class Rectangle {
    int hauteur_,
        largeur_;
public:
    int Hauteur() const;
    int& Hauteur();
    int Largeur() const;
    int& Largeur();
    int Perimetre() const;
};
int Rectangle::Hauteur() const {
    return hauteur_;
}
int& Rectangle::Hauteur() {
    return hauteur_;
}
int Rectangle::Largeur() const {
    return largeur_;
}
int& Rectangle::Largeur() {
    return largeur_;
}
int Rectangle::Perimetre() const {
    return 2 * (Largeur()+ Hauteur());
}
```

```
int & Rectangle::Hauteur() {
    return hauteur_;
}
```

⁴² Cet exemple est légal : une version `const` d'une méthode est considérée différente, par le compilateur, d'une version de même signature mais non `const`.

Propriétés⁴³

```

public class Rectangle
{
    private int hauteur = 0; // défaut
    public int Hauteur
    {
        get // accesseur implicite
        {
            return hauteur;
        }
        set // mutateur implicite
        {
            hauteur = value;
        }
    } // etc.
}

public class PetitProgramme
{
    public static void Main()
    {
        Rectangle r = new Rectangle();
        r.Hauteur = 5; // set
        int hau = r.Hauteur; // get
    }
}

```

Parmi les gens qui formulent des critiques comme celle présentée plus haut, on trouve plusieurs défenseurs du concept de *propriété*. Ce concept est implanté formellement par certains langages⁴⁴, mais pas par C++ ou par Java.

⇒ Une **propriété** sert à associer implicitement une opération Set et une opération Get⁴⁵ à un attribut.

Par exemple, pour le Rectangle ci-dessus, le langage C# aurait motivé l'utilisation d'une propriété plutôt que d'un trio accesseur/ mutateur/ attribut explicite. Il existe même des écritures plus compactes pour en arriver au même résultat⁴⁶.

L'exemple de code C# (à gauche) présente la propriété Hauteur. Son opération get est associée à toute utilisation en lecture de Hauteur, alors que son opération set est associée à toute opération en écriture de Hauteur. Notez que Hauteur est une propriété, mais que hauteur est l'attribut qui lui est associé.

Par souci de conséquence, on associe généralement le set et le get d'une propriété donnée à un seul et même attribut. Le mot clé value du set est toujours la valeur écrite dans la propriété.

Visiblement, une propriété est une formalisation simple d'un couple accesseur/ mutateur à valeur unique. On ne pourrait pas, par exemple, remplacer un Set à deux paramètres par une propriété. L'emploi de propriétés est donc une extension esthétique⁴⁷, mais limitée, du concept d'encapsulation primitif véhiculé par les couples Get/ Set les plus simples.

⁴³ Il est possible d'implémenter des propriétés en C++, mais les techniques pour ce faire sont avancées [hdebProp].

⁴⁴ *Visual Basic 6*, C#, Delphi, la plupart des outils .NET...

⁴⁵ En *Visual Basic 6*, les propriétés pouvaient aussi avoir un *Let*. La nuance entre *Let* et *Set* était analogue à celle entre une référence et une copie. On peut choisir de ne spécifier qu'un get ou un set dans une propriété.

⁴⁶ Ici: `public class Rectangle { public int Hauteur { get; set; } }` si l'on souhaite à la fois un accesseur et un mutateur (chacun est optionnel).

⁴⁷ Ce que les anglophones nomment du sucre syntaxique (*Syntactic Sugar*). Il faut faire attention de ne pas prendre le terme dans un sens négatif: ce n'est pas parce qu'un concept ou un outil n'est pas fondamental qu'il n'est pas appréciable ou utile. C# est un langage très riche en sucre syntaxique, et cela ne nuit pas à sa popularité.

Une propriété représente le même niveau de bris d'encapsulation qu'une paire accesseurs/mutateur, et ouvre les mêmes possibilités pour des accès de second ordre (dans la mesure où un seul paramètre entre en jeu pour le mutateur).

Les amateurs de propriétés soulignent que l'utilisation d'une propriété leur semble plus naturelle, en lecture comme en écriture, qu'un appel explicite à une méthode `Set` ou `Get`. De même, ils mettent de l'avant l'avantage organisationnel de juxtaposer à même le code l'accesseur et le mutateur sur un même attribut – s'il s'agissait de méthodes distinctes, ils pourraient être positionnés à deux endroits différents dans la déclaration de la classe.

Présumant que le programmeur puisse juxtaposer les méthodes comme bon lui semble, utiliser une propriété dans un langage qui le permet est surtout un choix esthétique. Il n'y a pas d'avantage technique à préférer une utilisation homogène de `Set/Get` pour la totalité des interfaces d'encapsulation primitives et secondaires, ou à préférer l'utilisation d'une notation ressemblant à celle d'un accès direct à un attribut comme le propose l'utilisation de propriétés⁴⁸.

Avec une propriété, comme avec une interface primitive d'encapsulation, ce n'est pas l'attribut qui se protège lui-même; c'est l'objet contenant l'attribut qui protège l'attribut. L'attribut peut aussi protéger son intégrité s'il définit ce que signifie lui affecter une valeur – s'il surcharge l'opérateur d'affectation. Nous montrerons comment y arriver plus loin dans ce document.

```
public class Rectangle
{
    public int Hauteur
    {
        get ; private set;
    } = 0;
    // etc.
}
```

Notez que C# supporte le concept de **propriétés auto-générées**, derrière lesquelles l'attribut est anonyme et n'est pas visible directement dans le code.

Dans le cas où il n'y a pas de validation à faire lors d'un accès en écriture, cette notation compacte permet d'en arriver à l'essentiel.

Notez qu'à gauche, les accès à `Hauteur` en écriture sont privés en écriture mais publics en lecture, et la valeur par défaut de l'attribut derrière la propriété `Hauteur` est 0.

⁴⁸ Il y a une exception notable à cette affirmation : certains environnements de développement, comme *Visual Studio*, *Netbeans* ou *Eclipse*, utilisent les propriétés dans leur environnement intégré d'édition, surtout pour les objets à vocation graphique. Les propriétés offrent un niveau primaire d'encapsulation identifié à même le code, pas seulement par convention (ce que sont les méthodes `set` et `get`) pouvant être avantageux lorsque l'alternative est d'exposer directement un attribut. Les *Beans* de Java, pour offrir des outils de développement semblables, demandent aux programmeurs de se discipliner et de rédiger des paires `set/ get` respectant des règles de nomenclature très strictes. Dans cette optique, la stratégie permettant une prise en charge à même les concepts du langage est préférable à celle reposant sur la discipline du programmeur.

Méthodes de classe

Si la hauteur minimale et la hauteur maximale d'un `Rectangle` sont des membres de classe, la question de savoir si une valeur entière donnée serait une hauteur valide ou non est une question pouvant être répondue par la classe `Rectangle`.

Nous allons donc procéder à la rédaction d'une méthode de classe `est_hauteur_valide()` qui retournera `true` si la valeur passée en paramètre serait une hauteur valide pour un `Rectangle`, et retournera `false` dans le cas contraire.

Les méthodes de classe, comme les attributs de classe, sont indiqués par la spécification `static` dans la déclaration de la classe (voir à droite).

```
class Rectangle {
public:
    //...
    static bool est_hauteur_valide(int);
    //...
private:
    static const int
        HAUTEUR_MIN, HAUTEUR_MAX,
        LARGEUR_MIN, LARGEUR_MAX;
    int hauteur_,
        largeur_;
};
```

Cette indication n'a pas à être répétée lors de la définition de la méthode :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
// ...
const int Rectangle::HAUTEUR_MIN = 1,
        Rectangle::HAUTEUR_MAX = 20,
        Rectangle::LARGEUR_MIN = 1,
        Rectangle::LARGEUR_MAX = 50;

// autres méthodes
bool Rectangle::est_hauteur_valide(int hauteur) {
    return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
}

// autres méthodes
```

On spécifiera une méthode comme étant une méthode de classe lorsque celle-ci n'utilise que des membres de classe. C'est le cas ici : de manière implicite, seules des constantes de classe sont utilisées par `est_hauteur_valide(int)`.

Une méthode de classe, par contre, ne pourrait pas accéder de manière implicite à un membre d'instance. Si la méthode `est_hauteur_valide(int)` cherchait toutefois à appeler `GetHauteur()`, la compilation échouerait. *La méthode `GetHauteur()` n'a de sens que pour une instance en particulier*, la hauteur de chaque instance pouvant être différente.

On utiliserait `est_hauteur_valide()` dans le contexte de la validation d'une hauteur proposée à `SetHauteur()`, tel que présenté à droite...

```
void Rectangle::SetHauteur(int hauteur) {
    if (est_hauteur_valide(hauteur))
        hauteur_ = hauteur;
    else
        throw std::exception{"Hauteur incorrecte"};
}
```

```
void Rectangle::SetHauteur(int hauteur) {
    if (Rectangle::est_hauteur_valide(hauteur))
        hauteur_ = hauteur;
    else
        throw std::exception{"Hauteur incorrecte"};
}
```

... ou encore, de façon équivalente, tel que présenté à gauche. Cette écriture est légale et correcte mais un peu lourde; en pratique, on privilégiera généralement celle-ci-dessus, où le nom de la classe est implicite.

Comme nous le verrons dans [POOv01], il y a des cas où il est nécessaire d'utiliser le nom entier d'un membre, incluant le nom de la classe où il est déclaré, pour faire disparaître certaines ambiguïtés.

Un sous-programme quelconque peut maintenant, puisque nous avons spécifié cette méthode comme étant publique, demander à la classe `Rectangle` de valider une hauteur potentielle avant de s'en servir.

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    int hau;
    if (cin >> hau && Rectangle::est_hauteur_valide(hau))
        cout << hau
            << " est une hauteur valide pour un Rectangle"
            << endl;
}
```

Question : si la méthode `est_hauteur_valide()` était plutôt une méthode d'instance, donc déclarée sans le mot clé `static`, comment aurions-nous dû nous y prendre dans le programme ci-dessus pour y avoir recours? *Écrivez le programme C++ correspondant à votre réponse.*

Référent à soi-même : le mot clé `this`⁴⁹

Il peut arriver dans une méthode qu'on veuille explicitement parler de l'instance propriétaire. Ceci est toujours possible à l'aide du mot clé `this`.

⇒ Le mot clé `this` est un pointeur vers l'instance propriétaire de la méthode en cours d'exécution. ***Il n'a donc de sens que dans le cadre d'une méthode d'instance.***

La mécanique de l'invocation des méthodes d'instance fait en sorte de passer silencieusement et implicitement un paramètre supplémentaire, `this`, à la méthode. Certains langages, comme Python, utilisent un passage explicite de ce paramètre (`Self`), alors que Java, C++ et C# procèdent de manière silencieuse et implicite.

L'une des marques d'élégance de l'approche OO tient au caractère implicite des références aux membres d'instance dans une méthode d'instance. La subjectivité de l'approche OO transparaît dans l'expression de cette relation.

```
class X {
public:
    void f(int);
    void g(float) const;
};

int main() {
    X x;
    // X::f() reçoit un int, mais
    // aussi un X* qu'est this
    x.f(3);
    // X::g() reçoit un float, mais
    // aussi un const X* qu'est this
    x.g(3.14159f);
}
```

À titre d'exemple, si une instance de `Rectangle` déclare une méthode `GetHauteur()`, on pourrait remplacer chaque mention faite à ce membre *dans une méthode d'instance*, là où le mot `this` a un sens, par `this->GetHauteur()` ou encore par `(*this).GetHauteur()`.

Dans une méthode `const`, le mot `const` s'applique en fait au pointeur `this`, ce qui empêche toute opération non constante sur l'instance active.

Dans la plupart des cas, on n'aura pas besoin d'utiliser `this`, du fait que l'accès aux membres d'instance d'une instance en particulier est, pour cette instance, implicite.

Il peut par contre arriver qu'un objet veuille explicitement faire référence à lui-même. Dans certains cas, comme on en verra dans le segment sur les opérateurs, c'est même nécessaire d'être capable de le faire.

On parle donc ici d'une idée à laquelle on ne peut échapper, mais aussi d'une idée à laquelle on n'a recours que dans certaines occasions.

Certains préfèrent utiliser le même nom pour un attribut et les paramètres des méthodes, comme dans le cas d'une classe `Rectangle` avec un attribut `Hauteur` et une méthode d'instance `SetHauteur()` prenant un paramètre aussi nommé `Hauteur`. Ces gens distingueront alors l'attribut du paramètre dans la méthode en écrivant explicitement `this->Hauteur`. L'élégance du nom de l'attribut est contrebalancée par la perte de la subjectivité implicite qu'entraîne l'usage forcé de l'expression `this->` devant les attributs.

Chaque approche a du pour et du contre; à l'exécution, les deux s'équivalent. Le biais du rédacteur de ce document est sans doute clair à la lecture des exemples et de la démarche suggérée, mais il reste qu'il s'agit là d'un biais, pas d'un dogme.

⁴⁹ Selon les langages, il y aura des variantes terminologiques pour le même concept (les mots `Me` et `self` sont des cas fréquemment rencontrés).

En résumé

- ⇒ Une classe ne peut accéder implicitement, par ses méthodes de classe, qu'à ses membres de classe.
- ⇒ Une instance peut accéder implicitement, par *ses* méthodes d'instance, à *ses* propres membres d'instance et aux membres de classe de *sa* classe.
- ⇒ La maxime importante est : toute instance connaît nécessairement sa classe, mais une classe ne connaît pas nécessairement chacune de ses instances.

		Méthode...	
		...d'instance	...de classe
Membre...	...d'instance	Une méthode d'une certaine instance peut accéder implicitement à un attribut d'instance ou à une méthode d'instance de la même instance.	Une méthode de classe ne peut pas accéder implicitement à un attribut d'instance ou à une méthode d'instance.
	...de classe	Une méthode d'instance peut accéder implicitement à un attribut de classe ou à une méthode de classe de la classe dont fait partie l'instance à laquelle elle appartient.	Une méthode de classe peut accéder implicitement à un attribut de classe ou à une méthode de classe de la même classe.

Réflexions

Quelques questions pour réfléchir un peu. Imaginons que vous dirigiez une équipe qui contribue au développement d'un projet à saveur informatique...

Q00 – L'un(e) de vos spécialistes prétend qu'il est superflu d'écrire un accesseur si une méthode ne fait que retourner la valeur d'un membre d'instance. Êtes-vous d'accord avec lui/ elle? *Justifiez votre réponse.*

Q01 – On vous demande s'il serait préférable d'implanter les constantes de classe de manière publique ou privée. Quelle est votre position, et pourquoi?

Q02 – L'un(e) de vos stagiaires se demande s'il est nécessaire de documenter un fichier d'en-tête. Après tout, on n'y retrouve pas de code, raisonne-t-il/ raisonne-t-elle. Quelle sera votre réponse? *Expliquez votre position.*

Dans d'autres langages

En Java, comme en C# et en VB.NET d'ailleurs, on remarquera tout de suite le support de caractères accentués à même le code source, ce qui est agréable pour les développeurs francophones (entre autres) mais pose quelques problèmes d'intégration à l'échelle internationale. Nous omettrons ici la question des exceptions, couverte dans *Traitement d'exceptions* plus loin.

Notez qu'en C++, les accents sont interdits même dans les commentaires, du moins quand le compilateur est strict. Nous trichons un peu, dans cette série de documents, pour améliorer la lisibilité.

La question de l'internationalisation des sources est matière à débats, encore aujourd'hui. Cela dit, nous en profiterons sans gêne dans nos exemples.

Les accesseurs et les mutateurs de second ordre ne sont ni plus complexes à implémenter, ni moins recommandables, en Java qu'en C++.

Pour des raisons techniques, que nous ne sommes pas en mesure d'expliquer pour le moment⁵⁰, il est plus difficile pour un compilateur Java d'optimiser correctement les invocations de méthodes simples comme `getHauteur()`. En Java, il est donc préférable d'utiliser des variables ou des constantes temporaires plutôt que d'invoquer de manière répétée de telles méthodes. Voir, à titre d'exemple, les constantes `HAUTEUR` et `LARGEUR` locales à la méthode `dessiner()`, à droite).

```
public class Rectangle {
    private int hauteur,
               largeur;

    public int getHauteur() {
        return hauteur;
    }

    public int getLargeur() {
        return largeur;
    }

    public void dessiner() {
        final int HAUTEUR = getHauteur(),
                 LARGEUR = getLargeur();
        for (int i = 0; i < HAUTEUR; i++) {
            for (int j = 0; j < LARGEUR; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }

    public int getAire() {
        return getLargeur() * getHauteur();
    }

    public int getPérimètre() {
        return 2 * (getLargeur() + getHauteur());
    }
}
```

⁵⁰ La note technique à cet effet va comme suit : en Java, toutes les méthodes d'instances sauf les méthodes qualifiées `final` et les constructeurs sont des méthodes polymorphiques, ce qui empêche le compilateur de prévoir les méthodes véritablement invoquées dans de tels cas. Les thématiques associées sont couvertes dans [POOv01].

Les méthodes et les attributs de classe ont une syntaxe semblable à celle employée en C++.

Pensons par exemple au rôle du mot clé `static` et à l'utilisation implicite par une instance d'un membre de classe comme s'il lui appartenait en propre... car c'est effectivement le cas.

La syntaxe quant à l'accès à un membre de classe mais de l'extérieur de la classe est fait, plus simplement, en passant par l'opérateur `.` plutôt que par l'opérateur `::` comme en C++, ce qui a le mérite d'alléger la syntaxe.

```
public void setHauteur(int hauteur) {
    if (estHauteurValide(hauteur)) {
        this.hauteur = hauteur;
    }
}
public void setLargeur(int largeur) {
    if (estLargeurValide(largeur)) {
        this.largeur = largeur;
    }
}
public static boolean estHauteurValide(int hauteur) {
    return HAUTEUR_MIN <= hauteur &&
           hauteur <= HAUTEUR_MAX;
}
public static boolean estLargeurValide(int largeur) {
    return LARGEUR_MIN <= largeur &&
           largeur <= LARGEUR_MAX;
}
private static final int
    HAUTEUR_MIN = 1, HAUTEUR_MAX = 20,
    LARGEUR_MIN = 1, LARGEUR_MAX = 50;
```

Notez cependant que, puisque `main()` est ici une méthode de classe de la classe `Rectangle`, il n'y est pas vraiment nécessaire de préfixer l'appel qui y est fait à `estHauteurValide()` par le nom de la classe `Rectangle`. Le recours, dans une méthode de classe de `Rectangle`, à une autre méthode de classe de `Rectangle` est compris de manière implicite.

Notez aussi que l'appel à la méthode `estHauteurValide()` fait dans `main()` est une illustration.

En pratique, on devrait s'en remettre à un `Rectangle` pour qu'il garantisse lui-même son intégrité.

```
public static void main(String [] args) {
    Rectangle r0 = new Rectangle();
    int hauteur = 3;
    if (Rectangle.estHauteurValide(hauteur)) {
        r0.setHauteur(hauteur);
        System.out.println
            ("Hauteur de r0: " + r0.getHauteur());
    }
}
```

Pour le reste, les éléments de syntaxe devraient sembler évidents. Remarquez, peut-être, le type `boolean` plutôt que `bool` pour les méthodes booléennes.

En C#, on remarquera un mélange syntaxique de C++ et de Java sans nécessairement voir de grandes différences conceptuelles.

Les constantes *d'une* classe ne peuvent y être que des constantes *de* classe, ce qui signifie que la qualification `static` sur les constantes (mot clé `const`) y serait redondante (elle y est même illégale). En C++, les instances aussi peuvent avoir des constantes (nous y reviendrons), ce qui impose une nuance supplémentaire.

L'invocation d'une méthode de classe (qualifiée `static`) peut y être faite implicitement à l'intérieur de la même classe (voir par exemple l'appel à `EstHauteurValide()` faite dans la méthode `SetHauteur()` ou explicitement à partir du nom de la classe (voir l'appel fait dans `Main()` à `EstHauteurValide()`).

```
namespace Formes
{
    class Rectangle
    {
        private const int LARGEUR_MIN = 1,
                        LARGEUR_MAX = 50,
                        HAUTEUR_MIN = 1,
                        HAUTEUR_MAX = 20;

        public static bool EstHauteurValide(int hauteur)
        {
            return HAUTEUR_MIN <= hauteur &&
                   hauteur <= HAUTEUR_MAX;
        }

        public static bool EstLargeurValide(int largeur)
        {
            return LARGEUR_MIN <= largeur &&
                   largeur <= LARGEUR_MAX;
        }

        public static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            int hauteur = 3;
            if (Rectangle.EstHauteurValide(hauteur))
                r.SetHauteur(hauteur);
        }
    }
}
```

La remarque faite plus haut à l'effet qu'un Rectangle devrait se porter garant de sa propre intégrité s'applique évidemment ici aussi.

Les accesseurs de second ordre sont, ici comme ailleurs, banals.

Notez que dans cet exemple, tel qu'il est idiomatique de le faire en C#, les accesseurs et les mutateurs ont été implémentés sous forme de propriétés.

```
public int Hauteur
{
    get { return hauteur; }
    private set
    {
        if (EstHauteurValide(value))
            hauteur = value;
    }
}
public int Largeur
{
    get { return largeur; }
    private set
    {
        if (EstLargeurValide(value))
            largeur = value;
    }
}
private int hauteur;
private int largeur;
public int Aire
{
    get
    {
        return Largeur * Hauteur;
    }
}
public int Périmètre()
{
    get
    {
        return 2 * (Largeur + Hauteur);
    }
}
public void Dessiner()
{
    for (int i = 0; i < Hauteur; i++)
    {
        for (int j = 0; j < Largeur; j++)
            System.Console.Write('*');
        System.Console.WriteLine();
    }
}
}
```

En **VB.NET**, on retrouvera les éléments vus précédemment :

- mot clé **Shared** pour les méthodes et les attributs de classe;
- droit aux caractères accentués dans le code source;
- manière d'accéder aux membres de classe;
- j'ai utilisé des méthodes **Set/ Get** dans ces exemple, mais des propriétés auraient aussi été convenables;
- mot clé **Const** pour les constantes, sans qualification spécifique pour les constantes de classe (il n'y en a pas de constantes d'instance en **VB.NET**); et

```

Namespace Formes
Public Class Rectangle
    Public Shared Function EstHauteurValide _
        (ByVal hauteur As Integer) As Boolean
        Return HAUTEUR_MIN <= hauteur And _
            hauteur <= HAUTEUR_MAX
    End Function
    Public Shared Function EstLargeurValide _
        (ByVal largeur As Integer) As Boolean
        Return LARGEUR_MIN <= largeur And _
            largeur <= LARGEUR_MAX
    End Function
    Public Shared Sub main()
        Dim r0 As New Formes.Rectangle
        Dim hauteur As Integer = 3
        If (Rectangle.EstHauteurValide(hauteur)) Then
            r0.SetHauteur(3)
        End If
    End Sub
    Private largeur As Integer
    Private hauteur As Integer
    Public Function GetLargeur() As Integer
        Return largeur
    End Function
    Public Function GetHauteur() As Integer
        Return hauteur
    End Function
    Public Function GetAire()
        Return GetHauteur() * GetLargeur()
    End Function
    Public Function GetPérimètre()
        Return 2 * (GetHauteur() + GetLargeur())
    End Function
    Public Sub SetLargeur(ByVal lar As Integer)
        If EstLargeurValide(lar) Then
            largeur = lar
        End If
    End Sub
    Public Sub SetHauteur(ByVal hau As Integer)
        If EstHauteurValide(hau) Then
            hauteur = hau
        End If
    End Sub
    Private Const HAUTEUR_MIN As Integer = 1
    Private Const HAUTEUR_MAX As Integer = 20
    Private Const LARGEUR_MIN As Integer = 1
    Private Const LARGEUR_MAX As Integer = 50

```

- éléments syntaxiques traditionnels de VB (If ... Then ... End If, par exemple, ou For ... Next) mêlés aux éléments adaptés de C comme l'initialisation des variables à la déclaration et le mot clé Return.

Portez attention aux bornes de la répétitive For car la borne supérieure y est inclusive, contrairement à la situation qui prévaut en C++, C# et Java.

```
Public Sub Dessiner()  
    For i As Integer = 0 To GetHauteur() - 1  
        For j As Integer = 0 To GetLargeur() - 1  
            System.Console.Write("*")  
        Next  
        System.Console.WriteLine()  
    Next  
End Sub  
End Class  
End Namespace
```

La surabondance de `-1` dans les bornes en VB.NET n'est pas tant accidentelle que le résultat d'une évolution historique. Le passage de VB6 à VB.NET s'est inscrit dans une entreprise d'homogénéisation partielle (et mitigée dans ses résultats) de certains standards langagiers.

Certaines structures de contrôle de VB6 (la répétitive For en est une) étaient adaptées aux coutumes en place, en particulier aux tableaux de N éléments dont les indices des éléments sont situés entre 1 et N inclusivement, alors qu'en VB.NET un tableau typique de N éléments sera indicé de 0 à N-1.

Exercices – Série 02

Quelques petites tâches à réaliser par vous-mêmes.

EX00 – Ajoutez les constantes spécifiant les bornes valides pour la largeur d'une `Rectangle`, de même que le code requis à `SetLargeur(int)` pour éviter qu'on puisse affliger un `Rectangle` d'une largeur invalide.

EX01 – Ajoutez les constantes `HAUTEUR_DEFAULT` et `LARGEUR_DEFAULT` à la classe `Rectangle`, et faites en sorte que leurs valeurs respectives soient celles de `HAUTEUR_MIN` et de `LARGEUR_MIN`.

EX02 – Faites en sorte que les mutateurs `SetHauteur()` et `SetLargeur()` utilisent les valeurs par défaut pour la hauteur et la largeur, respectivement, lorsqu'on leur passe en paramètre une valeur hors bornes.

Réflexions

Q00 – Si on désire garder, dans un attribut, la couleur d'un `Rectangle`, préférera-t-on utiliser un attribut d'instance ou un attribut de classe? Pourquoi?

Q01 – Si on désire garder, dans un attribut, le nombre d'instances de `Rectangle` en cours d'exécution, aura-t-on recours à un attribut d'instance ou un attribut de classe? Pourquoi?

Exercices – Série 03

EX00 – Ajoutez à la classe `Rectangle` une méthode de classe que vous nommerez `est_largeur_valide(int)` et utilisez convenablement cette méthode dans l'implémentation de `SetLargeur(int)` pour assurer la validité de la largeur proposée.

EX01 – Assurez-vous que les constantes de classe de la classe `Rectangle` soient bel et bien privées, puis ajoutez des accesseurs de classe (des `Get` portant la mention `static`) pour la hauteur et la largeur minimum et maximum d'un `Rectangle`.

EX02 – Sur la base de quels critères devrait-on décider si un attribut devrait être un attribut de classe ou un attribut d'instance? *Expliquez votre réponse.*

EX03 – Sur quels critères devrait-on se baser pour décider si une méthode devrait être une méthode de classe ou une méthode d'instance? *Expliquez votre réponse.*

EX04 – Pouvez-vous imaginer une raison pour avoir un attribut de classe qui ne soit pas une constante? Si oui, donnez un exemple. *Que vous répondiez oui ou non, prenez soin d'expliquer votre réponse.*

Réflexions

On vous propose la maxime suivante : en POO, si un membre peut jouer son rôle de manière pleine et entière en étant soit un membre de classe, soit un membre d'instance, au choix, alors il est généralement préférable d'en faire un membre de classe. Qu'en pensez-vous? Expliquez votre position, et discutez-en avec votre professeur.

Le mot-clé `auto`

Il est possible, depuis C++ 11, d'utiliser le mot-clé `auto` pour déterminer le type d'une variable. Cette particularité du langage, en apparence banale, a d'énormes ramifications dans nos pratiques au quotidien.

Lorsqu'une variable est déclarée de type `auto`, son type est automatiquement déduit du type de l'expression permettant de l'initialiser. Par exemple :

```
auto i = 3; // i est un int
auto flt = 3.14159f; // flt est un float
auto d = 3.5; // d est un double
auto s0 = "J'aime mon prof"; // s0 est un char(&)[16]
auto s1 = std::string{s0}; // s1 est un std::string
auto res = f() + g(); // le type de res est celui de f() + g()
// auto x; // illégal (pas d'expression pour initialiser x; type indéterminé)
```

Il est important de comprendre que dans chaque cas, le type de la variable est connu à la compilation; ce que permet `auto`, c'est de laisser le compilateur déterminer ce type à partir de l'initialisation plutôt que de demander aux programmeuses ou aux programmeurs de l'expliquer. Évidemment, si le type désiré est distinct du type de l'expression utilisée (p. ex. : une définition telle que `double x=3;`), alors `auto` n'est pas adéquat.

Depuis C++ 11, le mot clé `auto` permet de déduire le type d'une variable du type de l'expression qui a permis de l'initialiser. Nous utiliserons ce mot avec parcimonie pour le moment, parce que nous travaillons à développer une conscientisation face aux types impliqués, mais il prendra une place croissante dans nos exemples lorsque nous deviendrons plus confortables avec le sujet que nous étudions en ces pages.

Lorsque nous utilisons un littéral, le type d'une variable lors de sa déclaration peut être déduit du type du littéral lui-même. Il en va de même pour une expression plus complexe. Ainsi :

Ceci...	...équivalent à cela	Raison
<code>auto n = 3;</code>	<code>int n = 3;</code>	Le littéral 3 est de type <code>int</code>
<code>auto d = 3.5;</code>	<code>double d = 3.5;</code>	Le littéral 3.5 est de type <code>double</code>
<code>auto c = 'A';</code>	<code>char c = 'A';</code>	Le littéral 'A' est de type <code>char</code>
<code>auto ps = "Ha!";</code>	<code>const char *ps = "Ha!";</code>	Le littéral "Ha!" est de type <code>const char (&) [4]</code> , ou <i>référence sur un tableau de quatre char</i> , mais l'affectation à <code>ps</code> provoque ce qu'on appelle une décrépitude de pointeur (<i>Pointer Decay</i>) et ne conserve qu'un pointeur sur le premier caractère de la séquence
<code>auto s = string{"Ha!"};</code>	<code>string s{"Ha!"};</code>	L'expression <code>string{"Ha!"}</code> est de type <code>string</code>
<code>auto x = 1 + 2.5f;</code>	<code>float x = 1 + 2.5f;</code>	L'expression <code>1 + 2.5f</code> est de type <code>float</code> car l'entier 1 est promu au type <code>float</code> dû à son addition au littéral <code>2.5f</code>
<code>auto r = Rectangle{2,3};</code>	<code>Rectangle r{2,3};</code>	L'expression <code>Rectangle{2,3};</code> est, manifestement, de type <code>Rectangle</code>
<code>auto i = 0;</code>	<code>int i = 0;</code>	Le littéral 0 pris isolément est de type <code>int</code>
<code>auto x = {};</code>	s/o	Ne compilera pas. Le compilateur ne sait pas quoi faire avec cela
<code>class X { /* ... */ }; X f(); // ... auto x = f();</code>	<code>class X { /* ... */ }; X f(); // ... X x = f();</code>	Un appel de la fonction <code>f()</code> telle que déclarée ici retournera un <code>X</code>

Utiliser `auto` ne nous dispense pas d'agir avec intelligence. Prenez par exemple l'extrait de code ci-dessous (qui est de qualité discutable, mais bon, c'est une illustration) :

```
double somme_elems(const double tab[], unsigned long n) {
    auto cumul = 0.0; // Ok, cumul est un double
    for(auto i = 0; i < n; ++i) // oups! Problème!
        cumul += tab[i];
    return cumul;
}
```

Le problème dans la définition du compteur `i` est que pour le compilateur, le littéral `0` est un `int`, or la variable `n` est un `unsigned long` et il est possible que la comparaison `i < n` soit une tautologie (si `n` dépasse le seuil maximal pour un `int`).

Une solution possible aurait été d'explicitier le type du compteur :

```
double somme_elems(const double tab[], unsigned long n) {
    auto cumul = 0.0; // Ok, cumul est un double
    for(unsigned long i = 0; i < n; ++i) // Ok
        cumul += tab[i];
    return cumul;
}
```

Une alternative aurait été d'utiliser un littéral explicitement `unsigned long` :

```
double somme_elems(const double tab[], unsigned long n) {
    auto cumul = 0.0; // Ok, cumul est un double
    for(auto i = 0UL; i < n; ++i) // Ok (0UL signifie «0 de type unsigned long»)
        cumul += tab[i];
    return cumul;
}
```

Nous aurons parfois recours à `auto` pour déclarer nos variables et nos constantes. Les avantages sont nombreux; en particulier :

- lorsque les déclarations de types seraient complexes ou déplaisantes à rédiger, `auto` allège l'écriture. Par exemple, si `s` est de type `std::string`, alors il est bien plus simple d'écrire la définition `auto sz = s.size();` que de l'exprimer sous la forme (équivalente) `std::string::size_type sz = s.size();`
- le recours à `auto` réduit le couplage dans le code. Par exemple, écrire `auto x = f();` permet de changer le type de ce que retourne `f()` sans affecter le code client de manière adverse. Ceci simplifie les mises à jour du code source et évite certains bogues pernicieux;
- le fait qu'il soit nécessaire d'initialiser immédiatement un objet dont le type est `auto` nous force à ne construire cet objet que lorsque nous pouvons le faire convenablement, et réduit les cas d'objets créés avant que cela ne soit pertinent.

En pratique, les expressions suivantes sont équivalentes :

```
string s = "J'aime mon prof"; auto s = string{"J'aime mon prof"};
```

Dans un souci d'uniformité et de généralité, il est recommandé par de plus en plus d'experts de privilégier l'option de droite.

Répétitives *for* généralisées

Un atout important des déclarations `auto` est qu'elles peuvent être intégrées à des répétitives `for` lorsque celles-ci permettent de parcourir un conteneur standard [POOv02] dont le compilateur connaît la taille, dans la mesure où nous n'avons pas besoin de connaître explicitement les indices utilisés lors du parcours.

Prenons par exemple, le programme suivant :

```
#include <iostream>
// ... using ...
int main() {
    double tab[] = { 2,3,5,7,11 };
    enum { N = sizeof(tab) / sizeof(tab[0]) };
    for(int i = 0; i < N; ++i)
        cout << tab[i] << endl;
}
```

Ici, le tableau `tab` a un nombre `N` d'éléments connu à la compilation (cette valeur peut être déduite de la séquence utilisée pour son initialisation, comme nous le faisons d'ailleurs ici), et nous utilisons l'indice `i` de chaque élément pour accéder à l'élément, pas en tant que tel (nous n'affichons pas la valeur de `i`, à titre d'exemple).

Dans un tel cas, depuis C++ 11, il est possible d'écrire tout simplement :

```
#include <iostream>
// ... using ...
int main() {
    double tab[] = { 2,3,5,7,11 };
    for(auto val : tab) // ou for(int val : tab)
        cout << val << endl;
}
```

Le code est plus simple, plus général, et tout aussi rapide.

Limites du mot clé `auto`

Le mot clé `auto` capture le type sans qualifications. Ainsi, soit l'extrait suivant :

```
const int NMAX = 30;
auto nmax = NMAX;
```

... le type de `nmax` est `int`, pas `const int`. La qualification n'est pas capturée par le recours à `auto`; seul le type l'est.

Il est par contre possible de qualifier un objet déclaré avec `auto`. Quelques exemples :

```
const int NMAX = 30;
auto x = NMAX; // x est int
const auto y = NMAX; // y est const int
auto & z = NMAX; // z est const int& car elle réfère à un const int
```

Comprendre cette nuance est particulièrement important dans les répétitives. En effet, le code suivant est défectueux :

```
int tab[100] = { 0 };
for(auto val : tab)
    val = -1;
```

En effet, le type de `val` ici sera `int`, et chaque `val` dans le parcours sera une copie d'un élément de `tab`. Conséquemment, cette répétitive modifie des variables temporaires et le tableau original demeure inchangé.

Ce problème est simple à régler :

```
int tab[100] = { 0 };
for(auto &val : tab)
    val = -1;
```

Avec ce changement, chaque `val` est un `int&` et réfère à un élément de `tab`. Conséquemment, la répétitive initialise le tableau tel qu'attendu.

Bien saisir ces nuances peut faire toute la différence entre un programme dont l'exécution est rapide et un autre qui serait beaucoup plus lent. Par exemple :

```
#include <string>
#include <iostream>
// ... using ...
void initialiser(string [], size_t);
int main() {
    const int NCHAINES = 10'000;
    string chaines[NCHAINES];
    initialiser(chaines, NCHAINES); // remplir le tableau
    //
    // Ceci est lent, créant à chaque itération une copie de la chaîne à afficher
    //
    for(auto s : chaines)
        cout << s << endl;
    //
    // Ceci est plus rapide
    //
    for(auto & s : chaines)
        cout << s << endl;
    //
    // Ceci est aussi rapide mais plus propre
    //
    for(const auto & s : chaines)
        cout << s << endl;
}
```

D'autres applications du mot clé `auto` apparaîtront au fil de sections qui suivent, dans ce document comme dans les suivants.

Dans d'autres langages

En C#, il est possible de déclarer une variable de manière à ce que le compilateur en déduise le type à partir de l'expression servant à l'initialiser. Le mot clé à utiliser est `var` :

```
var x = new Rectangle(); // x est une référence sur un Rectangle
```

Toujours en C#, si les indices des éléments ne nous intéressent pas, il est possible de parcourir les éléments d'un conteneur avec `foreach` :

```
int [] vals = new int[n];  
// ...initialiser vals...  
foreach(int i in vals)  
{  
    Console.WriteLine(i);  
}
```

C# supporte avec `foreach` le typage implicite à l'aide de `var`. Étant donné que certains objets de C# n'ont pas de type connu du programmeur (p. ex. : les expressions lambda... Voir *Erreur ! Source du renvoi introuvable.*), le mot `var` est parfois nécessaire dans ce contexte :

```
int [] vals = new int[n];  
// ...initialiser vals...  
foreach(var i in vals)  
{  
    Console.WriteLine(i);  
}
```

Avec **VB.NET**, si `Option Infer` est activé, déclarer une variable avec `Dim` sans ajouter de nom est équivalent à utiliser `var` en C#. L'inférence de types est alors faite à la compilation, comme en C# ou en C++. VB.NET offre aussi une répétitive `For Each` (en deux mots) dont l'exécution est analogue au `foreach` de C#.

Avec **Java**, au moment d'écrire ceci, il n'existe pas d'équivalent à `auto` en C++ ou à `var` en C#. Il existe par contre une variante de la répétitive `for` pour qui ne se soucie pas des indices des éléments :

```
int [] vals = new int[n];  
// ...initialiser vals...  
for(int i : vals) {  
    System.out.println(i);  
}
```

Énumérations en tant que constantes

Cette section est une parenthèse technique. Si le sujet ne vous intéresse pas, poursuivez votre lecture à partir de la section *Parenthèse techniques – énumérations fortes*, un peu plus loin.

L'un des principaux problèmes de C++ est aussi l'une de ses principales forces, c'est-à-dire son lien de parenté avec le langage C, qui a pratiquement don d'ubiquité dans les milieux où la concentration d'ingénieurs et d'informaticiens est forte.

Parmi les héritages parfois irritants qu'obtient C++ du langage C, on trouve la méthodologie spécifique qu'ont ces deux langages quant à la compilation séparée des unités de traduction. Une unité de traduction étant un fichier source tel qu'il est suite à l'opération du préprocesseur, donc une fois toutes les lignes débutant par un # remplacées convenablement⁵¹.

Les langages C et C++ impliquent tous deux une compilation séparée de chaque fichier source (chaque .cpp), puis une édition des liens entre les fichiers objets (.o, .a, .obj ou .lib) résultants. L'unicité de certains symboles, liée directement à la règle ODR (voir *Annexe 03 – Concepts et pratiques du langage C++*), y devient une préoccupation importante : en langage C, par exemple, deux symboles globaux⁵² définis dans des unités de traduction devant être liés lors d'une édition de liens ne peuvent avoir le même nom; en C++, le même phénomène s'applique, dans la mesure où on remplace *nom* par *signature*.

La redondance potentielle de symboles globaux a forcé les concepteurs du langage C à développer l'idée de symboles globaux statiques (mot clé `static`) et externes (mot clé `extern`). Un symbole global statique n'apparaît pas lors d'une édition des liens, et peut donc être répété dans plusieurs unités de traduction distinctes, étant unique à chacune. En langage C comme en langage C++, les variables globales sont statiques par défaut⁵³, et les sous-programmes globaux sont externes par défaut⁵⁴.

⁵¹ Comme par exemple les directives `#include` remplacées par le contenu du fichier à inclure.

⁵² Par *symbole global*, on entend ici toute variable, toute constante ou tout sous-programme global. En C++ ISO, *global* signifie *membre de l'espace nommé anonyme*.

⁵³ Détail important : en langage C++, les constantes globales sont statiques par défaut, alors qu'en langage C elles sont externes. Cela signifie qu'en langage C, déclarer une constante globale `const int X;` est légal (et sémantiquement équivalent à `extern const int X;`) dans la mesure où une constante globale `X` de type `int` est définie par l'une des unités de traduction impliquées dans l'édition des liens. C'est clairement un endroit où C aurait dû suivre l'exemple de C++.

⁵⁴ Le langage C++ cherche à éviter l'utilisation du mot clé `static` pour qu'un sous-programme global soit local à l'unité de traduction courante. On préférera, en C++, ajouter un sous-programme local à l'espace nommé anonyme et le déclarer localement, ou dans un fichier d'en-tête n'étant pas destiné à un usage de type interface, pour obtenir précisément le même effet.

Étymologie d'un membre de classe

L'emploi de la spécification `static` pour un membre de classe en C++ tient en partie de l'idée d'avoir une seule définition pour le membre ainsi désigné⁵⁵. Son accessibilité entre unités de traduction est rendue possible par son nom : pour accéder au membre de classe `m` de la classe `X` de l'extérieur de `X`, il faut indiquer `X::m`, et il faut connaître la déclaration de `X` – donc avoir inclus `X.h` dans la plupart des cas.

Le *One Definition Rule*⁵⁶ du langage C++ stipule que toute entité doit n'être définie qu'une fois pour toute unité de traduction y ayant accès.

Pour éviter les multiples définitions d'un membre de classe, ce qui se produirait si les attributs de classe étaient définis à même un fichier d'en-tête, les membres de classe en langage C++ doivent être déclarés à même la déclaration de la classe dont ils font partie (souvent dans un fichier d'en-tête) mais ne peuvent être définis au même endroit, car cela ferait en sorte d'assurer une déclaration distincte du même attribut de classe dans chaque unité de traduction ayant incluse la déclaration de la classe à laquelle il appartient.

Cette déclaration de constante, dans `X.h` (on aurait aussi pu écrire `static float m = {};`, où le `=` est optionnel)...

... entraîne une définition de `X::m` ici...

... de même qu'une autre définition de `X::m` ici, ce qui contrevient à la règle de définition unique des symboles globalement accessibles.

```
class X {
public:
    // ce qui suit est illégal
    static float m = 0.0f;
};

// a.cpp
#include "X.h"

// b.cpp
#include "X.h"
```

C++ fait, depuis la standardisation ISO, une exception pour les attributs de classe s'il s'agit de constantes entières. Cela dit, ce que nous expliquerons à l'instant fait partie des usages couramment rencontrés en C++, et est suffisamment élégant pour mériter une brève discussion.

⁵⁵ L'utilisation de ce mot en Java pour le même concept provient d'un désir de mimétisme de C++, et l'utilisation de ce mot en C# tient d'un souci de mimétisme de Java. Les préoccupations commerciales font en sorte, lorsque nous les laissons nous dominer, de nous éloigner du concept. De manière générale, *statique* signifie connu dès la compilation (*dynamique* implique connu seulement lors de l'exécution du programme).

⁵⁶ Voir *ODR – la One Definition Rule* pour des détails.

Un effet secondaire agaçant

En général, le principe de définir à un seul endroit (en C++) tout attribut de classe ne pose pas de problème conceptuel.

X.h	X.cpp
<pre>class X { public: // déclaration static const int m; };</pre>	<pre>#include "X.h" // définition const int X::m = 0; // ou encore // const int X::m{};</pre>

Le problème survient lorsqu'on veut déclarer une entité qui dépende, à la compilation, de la valeur d'un attribut de classe. L'exemple le plus frappant est celui des tableaux.

En effet, pour que la déclaration à droite soit légale, il faut que `T` ait une taille connue à la compilation... or, les unités de traduction autres que celles qui définiront `M` ne connaissent pas, à la compilation, la valeur de `M`, et se trouvent ainsi incapables de gérer `T` (et de gérer la taille de tout `X`) correctement.

```
class X {
public:
    // déclaration
    static const int M; // ok; X::M à définir
                        // (dans X.cpp?)

    int T[M]; // illégal!
};
```

On voudrait, pourtant, pouvoir déclarer des tableaux dans une classe dont la taille est déterminée par la classe en question sans recourir à de l'allocation dynamique de mémoire.

Une solution (décevante) au problème est de fixer la taille en question par une constante globale. Ça fonctionne, mais on y perd au change sur le plan de l'encapsulation.

Par exemple, dans l'exemple à droite, `TAILLE` est une constante globale, qui est utilisée pour fixer, à la compilation, la valeur de `X::M` et la taille `T[]` pour tout `X`.

```
const int TAILLE = 10; // constante globale

class X {
public:
    // ok; X::M à définir (dans X.cpp?)
    static const int M;

    // déclaration
    int T[TAILLE]; // légal mais suspect
};
```

Ce qui est irritant ici est que `TAILLE` est, visiblement, une rustine pour contourner un problème technique. Ajouter une constante globale pour régler un problème d'encapsulation est plutôt triste.

Sans compter que dans un cas comme celui-ci, l'équivalence entre le nombre d'éléments du membre `T` d'un `X` et la valeur de la constante `X::M` dépend ici de la discipline des programmeuses et des programmeurs, ce qui introduit des risques et de la complexité tous deux inutiles dans l'entretien du code.

```
#include "X.h"

// définition
const int X::M = TAILLE;
```

Utiliser une énumération comme source de constantes entières

La solution la plus simple à ce problème est d'utiliser un type énuméré comme spécification de taille pour un tableau. En effet, un type énuméré (un `enum`) liste des constantes entières dont les valeurs sont connues à la compilation, mais a le privilège d'être soustrait aux règles s'appliquant aux constantes symboliques.

Par exemple, dans l'exemple à droite, `X::TAILLE` est une valeur d'un type énuméré anonyme (on aurait pu donner un nom au type, mais ce n'est pas nécessaire ici).

Les valeurs possibles pour un type énuméré étant positives et entières, cette stratégie comporte les mêmes avantages que la plupart des constantes, mais sans les ennuis habituels et sans les effets secondaires irritants des constantes globales.

```
class X {
    enum { TAILLE = 20 };
public:
    // déclaration
    int tab[TAILLE]; // légal!
};
```

L'emploi de constantes énumérées ne doit pas être vu comme un dénigrement des constantes de classe, qui ont leur rôle à jouer, surtout quand on désire une constante d'un type qui ne fait pas partie des types entiers. Il s'agit d'un mécanisme somme toute élégant permettant de contourner un irritant avec peu d'effort et sans perte réelle de force symbolique.

Les constantes entières et les autres

Cela dit, bien que l'utilisation de constantes énumérées pour représenter des constantes statiques entières soit légal (et souvent utilisé, parce que le code résultant est concis) encore aujourd'hui, il se trouve que la plupart des compilateurs C++ distinguaient, jusqu'à C++ 11, les constantes entières des autres. Pour deux raisons :

- pour faciliter la définition de tailles de tableaux, bien sûr; mais aussi
- pour fins d'optimisation.

Ainsi, les constantes entières sont toutes susceptibles d'être « brûlées » dans le code objet lors de la compilation des fichiers sources. En ce sens, elles ont un traitement différent des autres constantes, qu'il s'agisse d'objet ou de primitifs non-entiers.

Pour cette raison, il est illégal en C++ de prendre l'adresse d'une constante statique entière : le compilateur n'a pas l'obligation de générer une véritable entité dans le code objet pour ces constantes, et les remplacera typiquement par des littéraux lors de la génération du code.

C++ 11 lève cette interdiction de combiner la déclaration et la définition d'une constante statique en un même lieu, mais le support de cette fonctionnalité, au moment d'écrire ceci, varie beaucoup selon les compilateurs.

Dans d'autres langages

En Java, les constantes énumérées ne furent introduites qu'avec la version 1.5 du langage.

La syntaxe est semblable à celle des `enum` de C et de C++ à ceci près que la conversion implicite d'un `enum` en `String` est jolie : le nom du symbole est utilisé, ce qui est plus sophistiqué que ce qu'on trouve en C++ où une valeur d'un `enum` n'est qu'un entier.

```
public class X {
    enum Couleurs {
        Vert, Bleu, Jaune
    }
    public static void test() {
        System.out.println(Couleurs.Vert);
    }
    public static void main(String [] args) {
        // Affiche "Vert"
        System.out.println(Couleurs.Vert);
    }
}
```

En retour, une constante énumérée y est un symbole, pas un entier (du moins pas au sens du langage), et ne peut servir de taille à un tableau. Cela dit, Java ne donnant pas dans la compilation séparée, ceci ne pose aucun problème logistique.

En C#, la situation est identique et les conclusions sont les mêmes, incluant la conversion implicite d'un symbole énuméré en une chaîne de caractères du même nom.

```
namespace Test
{
    class X
    {
        enum Couleurs
        {
            Vert, Jaune, Bleu, NB_COULEURS
        };
        public static void Test()
        {
            System.Console.WriteLine(Couleurs.Vert);
        }
        public static void Main(string [] args)
        {
            Test();// Affiche Vert
        }
    }
}
```

En langage `VB.NET`, la situation est aussi identique, avec les mêmes avantages et les mêmes inconvénients.

La syntaxe seule diffère, surtout de par sa forme *ligne par ligne*.

```
Namespace Test
Public Class X
    Public Enum Couleurs
        Vert
        Jaune
        Bleu
    End Enum
    Public Shared Sub Test()
        System.Console.WriteLine(Couleurs.Vert)
    End Sub
    Public Shared Sub main()
        Test()
    End Sub
End Class
End Namespace
```

Parenthèse techniques – énumérations fortes

Les énumérations traditionnelles (avant C++ 11) de C++ sont « faibles », au sens où elles constituent en quelque sorte une manière concise d'exprimer une séquence de constantes entières de valeurs consécutives, du moins par défaut.

<i>Ceci...</i>	<i>...équivalent généralement à cela</i>
<code>enum { A, B, C };</code>	<code>static const int A=0, B=1, C=2;</code>

Les valeurs d'un `enum` sont traditionnellement des `int`, même si la plage de valeurs couverte par les constantes énumérées d'un même groupe est suffisamment petite pour être représentée en entier par des `char` ou par des `short`.

De même, les valeurs d'un `enum` traditionnel ne sont pas qualifiées par le nom de leur type. Ainsi, le programme suivant est ambigu :

```
enum Medaille { Or, Argent, Bronze };
enum Metal { Argent, Fer, Cuivre };
int main() {
    Metal m = Argent; // ambigu : valeur 1 ou 0?%
}
```

Depuis C++ 11, il est possible d'exprimer des énumérations « fortes ». Ce terme prend deux sens distincts; en effet :

- il est possible de faire en sorte que les valeurs d'une énumération doivent être qualifiées par leur type; et
- il est possible de choisir le substrat pour les valeurs d'une énumération.

Pour qualifier les valeurs d'une énumération par leur type, il suffit d'utiliser l'écriture suivante :

```
enum class Medaille { Or, Argent, Bronze };
enum class Metal { Argent, Fer, Cuivre };
int main() {
    // Metal m = Argent; // illegal
    Metal m = Metal::Argent; // Ok
    // Metal m = Medaille::Argent; // illegal, aucune conversion n'existe...
}
```

Le mot clé `class` entre `enum` et le nom du type indique que le code client souhaite une énumération « forte ».

Pour choisir le substrat des valeurs, il suffit d'utiliser une notation comme la suivante :

```
enum class Medaille : char { Or, Argent, Bronze };
```

Ce faisant, chacune des valeurs sera représentée sur `sizeof(char)` *bytes* plutôt que sur `sizeof(int)` *bytes* comme c'est la tradition. Tout type entier peut bien sûr être utilisé ici.

Vie d'un objet

Rôle de cette section

Les objets sont des entités dynamiques du moment de leur création jusqu'au moment où ils cessent d'exister, *inclusivement*. La construction et la destruction d'un objet sont des moments charnières et sensibles sur lesquels il faut maintenant porter notre regard.

Un objet (au sens restreint d'instance d'une classe) a une durée de vie précise et délimitée par deux événements importants, qui sont sa **construction** et sa **destruction**.

La possibilité de déterminer l'état initial d'un objet, de contrôler ce qui se produit au moment où il débute et conclut son existence, permet (de concert avec les techniques d'encapsulation) d'assurer sur lui un contrôle de qualité entier.

Prenons encore une fois le cas de la classe `Rectangle`, et déclarons-en une instance que nous nommerons `r`.

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    int hauteur,
        largeur;
    // Lecture et validation, hauteur du Rectangle
    do {
        cout << "Entrez une hauteur pour un Rectangle: ";
    } while (cin >> hauteur &&
        !Rectangle::est_hauteur_valide(hauteur));
    // Idem pour la largeur (omis pour économie)
    Rectangle r; // (0)
    r.SetHauteur(hauteur);
    r.SetLargeur(largeur);
    r.Dessiner();
} // (1)
```

Les règles de portée vues dans le segment sur la programmation structurée en C++ s'appliquent aux objets comme aux autres variables et aux autres constantes. Ainsi, la durée de vie de l'instance `r` est délimitée inclusivement par les lignes **(0)** et **(1)** dans le petit programme ci-dessus :

- la ligne **(0)** est la déclaration de l'objet `r`. C'est à ce moment que la vie de `r` débute officiellement : le moment de sa **construction**;
- la ligne **(1)** constitue l'accolade fermante du bloc dans lequel `r` fut déclaré, et marque par conséquent la fin de sa vie : le moment de sa **destruction**.

Quand plusieurs objets rencontrent leur destruction au même endroit dans un programme, l'ordre de leur destruction dépend de l'implémentation.

Un objet est une structure de données pouvant être arbitrairement complexe. Le moment de sa construction et celui de sa destruction sont, par conséquent, fort importants, et méritent un examen approfondi.

Remarquez le passage suivant :

```
Rectangle r; // (a)
r.SetHauteur(hauteur); // (b)
r.SetLargeur(largeur); // (c)
```

Il y a quelque chose de malsain dans ce code : l'état de `r` est (pour le moment) indéterminé après l'instruction (a), et à moitié déterminé après (b). Une telle situation ne respecte pas le principe d'encapsulation décrit dans **Encapsulation – Contrôler l'accès à l'information**, plus haut. C'est en partie ce à quoi nous remédierons dans cette section.

Le constructeur

La construction d'un objet sollicite⁵⁷ chez lui un sous-programme particulier qu'on nomme le **constructeur**.

⇒ Un **constructeur** est une méthode dont la tâche est de mettre en place l'état initial d'un objet.

Un constructeur est un sous-programme spécial et important, lancé automatiquement lorsqu'une instance est déclarée. La définition d'un objet, donc, aura toujours pour impact d'exécuter un sous-programme : le constructeur approprié.

En l'absence d'un constructeur pour une classe (comme ce fut jusqu'ici le cas pour la classe `Rectangle`), le compilateur C++ alloue, lors de la création d'un objet, la mémoire nécessaire pour la bonne gestion de ses attributs et sollicite implicitement le constructeur par défaut de ses attributs (dans le cas des types primitifs, faute d'un tel constructeur par défaut implicite⁵⁸, ceux-ci demeurent dans un état aléatoire, n'offrant aucune garantie quant à leur valeur).

Toutefois, ce n'est pas là un comportement adéquat dans la plupart des cas, et on aimerait que la construction d'un objet soit *a priori* porteuse de garanties quant à l'état de cet objet une fois qu'il aura été construit :

- on voudrait que la déclaration d'un objet fasse en sorte de le créer avec des valeurs par défaut considérées convenables pour chacun de ses attributs. *Assurer qu'on puisse créer un objet avec des valeurs par défaut requiert un **constructeur par défaut***;
- on voudrait aussi, lorsqu'on connaît les valeurs à attribuer à certains attributs d'un objet, pouvoir les lui spécifier dès sa construction. *Assurer qu'on puisse créer un objet avec des valeurs explicites lors de sa construction implique l'existence d'au moins un **constructeur paramétrique***;
- on verra finalement qu'il est essentiel, à des fins pratiques et techniques, de rédiger un troisième type de constructeur⁵⁹ nommé **constructeur de copie**.

Il existe un certain nombre d'autres catégories de constructeurs (en particulier le constructeur de séquence et le constructeur de copie apparentée), sur lesquels nous reviendrons dans des volumes explorant des approches OO plus poussées.

Les catégories explorées dans le présent document sont les plus répandues et celles susceptibles d'apparaître dans la majorité des langages OO.

D'autres types de constructeurs existent (constructeur de conversion, constructeur de séquence, constructeur de mouvement, ...). Ceux-ci correspondent à des thématiques plus poussées, et sont – conséquemment – couverts dans des documents ultérieurs de cette série.

⁵⁷ Il est étrange de parler de la sollicitation d'une méthode servant à la construction d'un objet; après tout, si on s'apprête à construire l'objet, c'est qu'il n'existe au préalable pas encore! Et pourtant, ça fonctionne. Le constructeur, pour être précis, est le sous-programme appelé lors de l'initialisation de l'objet. Son rôle n'est pas tant de construire l'objet en totalité que d'entrer en jeu aussi tôt que possible dans son cycle de construction pour en assurer un état initial contrôlé, et par conséquent connu et convenable. Plus tard dans cette série de documents, nous examinerons en détail les étapes menant à l'attribution d'un espace mémoire pour un objet et celles menant à son initialisation correcte, mais nous n'en sommes pas là pour le moment.

⁵⁸ Le mot *implicite* ici n'est pas accidentel; nous y reviendrons sous peu dans la section **Constructeurs par défaut et types primitifs**.

⁵⁹ Ce constructeur est, au fond, un cas particulier mais extrêmement important de constructeur paramétrique.

Constructeur par défaut

Reprenant l'exemple d'utilisation d'un `Rectangle` ci-dessus, on voit que la déclaration de `r` se fait comme proposé à droite.

```
Rectangle r;
```

Lorsqu'on instancie de cette manière, sans rien spécifier d'autre que le nom de la classe et celui de ses instances, le constructeur sollicité par la mécanique de construction est le **constructeur par défaut** de cette classe.

⇒ On nomme **constructeur par défaut** un constructeur sans paramètres.

⇒ Les attentes face à un tel constructeur sont que l'instance résultante ait, pour chacun de ses attributs, des valeurs initiales convenables.

On ajoutera à la déclaration de la classe `Rectangle` un constructeur par défaut de la manière suivante. Remarquez la syntaxe de la déclaration de cette méthode (à droite).

Les lignes en caractères gras sont celles qui sont vraiment pertinentes à notre sujet.

```
class Rectangle {
public:
    // ...
    // -- Constructeur par défaut
    Rectangle();
    // ...
private:
    static const int
        HAUTEUR_MIN, HAUTEUR_MAX,
        HAUTEUR_DEFAULT,
        LARGEUR_MIN, LARGEUR_MAX,
        LARGEUR_DEFAULT;
    int hauteur_,
        largeur_;
};
```

Le constructeur par défaut de `Rectangle` est celui que le code client utilisera quand il voudra un `Rectangle` **typique**.

La déclaration (le prototype) du constructeur par défaut est telle que présenté à droite.

```
Rectangle();
```

Première remarque : en C++, **un constructeur doit porter le même nom que la classe dont il doit construire une instance**. Ceci est vrai de tous les constructeurs, celui par défaut comme tous les autres.

Deuxième remarque : **un constructeur par défaut est un constructeur sans paramètres**. C'est pourquoi on remarquera que les parenthèses qui en suivent le nom sont *vides*.

Troisième remarque : **un constructeur n'a pas de type**. En langage C++, un constructeur ne retourne rien au sens usuel, mais n'est pas non plus une procédure⁶⁰. *Le fruit de l'exécution d'un constructeur est une instance de la classe qu'il sert à construire, convenablement initialisée.*

Ces remarques s'appliquent aussi à C# et à Java (mais pas à VB.NET)

Notez l'ajout de deux nouvelles constantes à la classe `Rectangle`, nommées `Rectangle::HAUTEUR_DEFAULT` et `Rectangle::LARGEUR_DEFAULT`. Sachant devoir attribuer des valeurs par défaut à la hauteur et à la largeur d'un `Rectangle`, il est propice de déclarer des constantes de classe qui serviront à entreposer ces valeurs.

⁶⁰ ...une fonction `void`.

La définition du constructeur par défaut d'un `Rectangle` se fera comme suit :

```
#include "Rectangle.h" // déclaration de la classe Rectangle
// Autres inclusions
const int Rectangle::HAUTEUR_MIN = 1,
        Rectangle::HAUTEUR_MAX = 20,
        Rectangle::HAUTEUR_DEFAUT = Rectangle::HAUTEUR_MIN,
        Rectangle::LARGEUR_MIN = 1,
        Rectangle::LARGEUR_MAX = 50,
        Rectangle::LARGEUR_DEFAUT = Rectangle::LARGEUR_MIN;
// Autres méthodes
Rectangle::Rectangle() {
    SetLargeur(LARGEUR_DEFAUT);
    SetHauteur(HAUTEUR_DEFAUT);
}
// Autres méthodes
```

La forme proposée ici est correcte, mais peut être raffinée. La notation alternative (et typiquement préférable, du moins en C++) pour initialiser les attributs d'une instance sera couverte dans *Préconstruction*, plus bas.

Encore une fois, on remarque les points saillants de la syntaxe des constructeurs, en particulier celui par défaut :

- son nom est le même que celui de sa classe;
- il n'a pas de type à proprement dit; et
- il n'accepte pas de paramètres (ce qui fait de lui un constructeur par défaut plutôt qu'une autre sorte de constructeur).

Le rôle joué ici par le constructeur par défaut est celui auquel on est en droit de s'attendre de sa part : il initialise chacun des attributs du `Rectangle` à sa valeur par défaut, utilisant les mutateurs appropriés.

Les valeurs par défaut qui ont été attribuées ici à la hauteur et à la largeur d'un `Rectangle` sont purement arbitraires. L'important est, bien sûr, que celles-ci soient valides pour un `Rectangle` tel qu'on l'a défini.

Ce qui constitue une bonne valeur par défaut pour un attribut donné, et ce qui fait un bon constructeur par défaut, de façon générale, dépendra de la classe, du problème... Comme dans bien des cas, les meilleurs choix d'implantation seront ceux qui seront simples, efficaces, sans risque et qui faciliteront la réutilisation des objets dans le plus de contextes différents possibles.

Constructeurs par défaut et types primitifs

Certains langages, comme Java et les langages .NET, imposent des valeurs par défaut aux types primitifs. En Java, ces valeurs sont dictées par la norme du langage, alors que la plateforme .NET fait des types primitifs des alias pour des entités semblables aux classes mais avec des capacités réduites, et définit des constructeurs par défaut pour ces types.

En pratique, compter sur les valeurs par défaut imposées par une norme de langage complique la portabilité du code. Si vous utilisez Java ou un langage .NET, il demeure préférable d'explicitement vos intentions à même le code, pour éviter de compliquer une éventuelle migration technologique. Soyez clair(e)s dans vos intentions (dans la mesure où cela ne réduit pas la performance de vos programmes).

C++ porte sur ses épaules l'héritage du langage C, qui laisse indéfinies les valeurs des variables de types primitifs⁶¹.

Ainsi, l'énoncé suivant initialise `i` à la valeur `0` en Java mais laisse indéfinie la valeur de `i` en C++ :

```
int i;
```

Cependant, cet énoncé a le même impact en Java, C++ et C#, soit initialiser `i` à la valeur `0`, et est donc nettement préférable au précédent :

```
int i = 0;
```

C++ 03 offre toutefois une syntaxe homogène à la construction, pour les types primitifs comme pour les objets, soit celle-ci :

```
int i = int();
```

Cet énoncé se lit *affecter à `i` la valeur d'un `int` par défaut*. La syntaxe `T v = T();` a le mérite de fonctionner pour tout type `T`, qu'il s'agisse d'un type primitif ou d'une classe, et est donc privilégiée par toutes celles et tous ceux qui utilisent C++ pour écrire de la programmation générique [POOv02], où les programmes, les classes et les algorithmes doivent être applicables à de larges gammes de types de manière homogène.

Prudence, toutefois : ne confondez pas cette syntaxe avec celle de `i0`, à droite, qui déclare un prototype de fonction nommée `i0`, ne prenant pas de paramètre et retournant un `int`. Vous auriez de vilaines surprises!

```
int i0(); // prototype de fonction
int i1; // valeur indéfinie
int i2 = 0; // i2 vaut zéro
int i3 = int(); // i3 vaut zéro
```

Depuis C++ 11, les syntaxes suivantes sont possibles, et **c'est celles-ci que nous allons privilégier**.

```
int i = {};
```

```
int i {}; // le = est optionnel
```

Cette écriture est homogène, étant applicable à tous les types, et évite certaines ambiguïtés syntaxiques susceptibles de survenir avec les parenthèses. La raison pour laquelle ce n'est pas l'écriture utilisée dans ce document est qu'elle n'est pas encore pleinement supportée par tous les compilateurs au moment d'écrire ces lignes.

En C++ comme en Java et dans les langages .NET, les valeurs par défaut des types primitifs sont les zéros de ces types (pointeur nul pour un `X*`, entier de valeur zéro pour un `int`, `0.0f` pour un `float`, et ainsi de suite).

⁶¹ Le langage C a pour principe qu'un programme ne doit payer que pour ce dont il a vraiment besoin, et pousse ce principe à l'extrême.

Constructeur paramétrique

Revenons au programme utilisé comme exemple, plus haut, en le résumant brièvement par l'extrait de code ci-dessous, qui présume que `haut` et `lar` sont deux entiers ayant des valeurs convenables pour, respectivement, la hauteur et la largeur d'un `Rectangle`:

```
// ...
Rectangle r;
r.SetHauteur(haut);
r.SetLargeur(lar);
r.Dessiner();
// ...
```

Cet extrait vise à dessiner un `Rectangle` de hauteur `haut` et de largeur `lar`, et procède en trois étapes :

- il crée d'abord un `Rectangle` par défaut;
- il lui affecte une hauteur et une largeur spécifique; puis
- il dessine ce `Rectangle`.

Les deux premières étapes servent toutes essentiellement à la construction d'un `Rectangle` particulier, qui se trouve à être différent du `Rectangle` par défaut. Il est possible de les réunir en une seule étape de manière à simplifier la stratégie d'ensemble, soit :

- créer un `Rectangle` d'une hauteur et d'une largeur spécifique, intégrant effectivement la construction d'un `Rectangle` et l'affectation immédiate à ses attributs de valeurs spécifiques; puis
- dessiner ce `Rectangle`.

On parvient à ce résultat à l'aide d'un *constructeur paramétrique*.

⇒ On nomme **constructeur paramétrique** un constructeur muni de paramètres.

⇒ On veut qu'un tel constructeur utilise les paramètres qu'on lui confie pour initialiser l'instance en cours de construction.

⇒ Un cas particulier de constructeur paramétrique est celui prenant en paramètre une référence vers une constante sur un objet du même type que celui construit. On nomme *constructeur de copie* un tel constructeurs; une section leur est consacrée, plus bas.

⇒ Un autre cas particulier de constructeur paramétrique est celui ne prenant qu'un seul paramètre, d'un type autre que celui construit. Un tel constructeur est dit **constructeur de conversion**, puisque *sauf indication contraire*⁶², il permet de guider la conversion implicite d'une instance d'un type en une instance d'un autre type (par exemple, faire une `Note` à partir d'un nombre à virgule flottante). Le nom constructeur de conversion sert aussi à des fins plus sophistiquées qui seront couvertes dans des volumes ultérieurs.

Le constructeur paramétrique de `Rectangle` est celui que le code client utilisera quand il voudra un `Rectangle` **spécifique**, en fonction de besoins précis, pas seulement un `Rectangle` typique.

⁶² Voir *Construction implicite ou construction explicite*, plus loin.

On ajoutera à la déclaration de la classe `Rectangle` un constructeur paramétrique permettant de créer un `Rectangle` d'une hauteur et d'une largeur spécifique de la manière présentée ci-dessous. Remarquez la syntaxe de la déclaration de cette méthode.

Le prototype du constructeur indique le nombre de paramètres qu'il acceptera, et le type de chacun d'entre eux. Le constructeur paramétrique que nous rédigerons ici prendra deux entiers constants en paramètre, qui seront (dans l'ordre) la hauteur et la largeur désirées pour le `Rectangle` à construire.

Notez que, bien que les noms de paramètres soient optionnels dans une signature de sous-programme, les omettre ici risquerait d'entraîner une confusion chez les utilisateurs de la classe.

On peut avoir plusieurs constructeurs paramétriques distincts, et les règles usuelles servant à différencier deux sous-programmes s'appliquent autant aux constructeurs qu'aux procédures et aux fonctions.

La définition de ce constructeur sera telle que visible à droite.

On y remarquera les mêmes éléments que pour la définition du constructeur par défaut, à ceci près que dans ce cas, les valeurs initiales pour la largeur et la hauteur du `Rectangle` à construire seront celles passées en paramètre, plutôt que des valeurs par défaut.

```
class Rectangle {
public:
    // Autres méthodes
    // -- Constructeur par défaut
    Rectangle();
    // -- Constructeur paramétrique
    Rectangle(int hauteur, int largeur);
    // reste de la classe
};
```

```
#include "Rectangle.h"
// ...
Rectangle::Rectangle(int hauteur, int largeur) {
    SetLargeur(largeur);
    SetHauteur(hauteur);
}
// ...
```

Le programme suivant illustre l'utilisation de constructeurs paramétriques :

```
#include "Rectangle.h"
int main() {
    Rectangle r1,          // r1 est un Rectangle par défaut (1 x 1)
                r2(10, 5 ), // r2 est un Rectangle 10 x 5
                r3( 3, 3 ), // r3 est un Rectangle 3 x 3
                r4{ 2, 4 }; // r2 est un Rectangle 2 x 4
    r1.Dessiner();        // Dessine un Rectangle 1 x 1
    r2.Dessiner();        // Dessine un Rectangle 10 x 5
    r3.Dessiner();        // Dessine un Rectangle 3 x 3
    r4.Dessiner();        // Dessine un Rectangle 2 x 4
}
```

Depuis C++ 11, écrire `Rectangle r(2,3);` et écrire `Rectangle r{2,3};` sont, dans la majorité des cas, manières équivalentes de procéder, et on privilégiera l'écriture avec accolades, que nous avons utilisé pour `r4` ci-dessus.

Les accolades sont plus restrictives que les parenthèses. Par exemple, étant donné l'ébauche de classe `Point` à droite, l'écriture `Point(1.0,-1.0)` serait acceptable, et les `double` que sont `1.0` et `-1.0` seraient implicitement convertis en `float`, mais l'écriture `Point{1.0,-1.0}` ne le serait pas (des `float` sont exigés, alors les `double` sont refusés).

```
class Point {
    // ...
public:
    Point(float, float);
    // ...
};
```

Réflexion 00.5 – Remarquez que notre constructeur paramétrique initialise ses attributs en passant par des mutateurs plutôt qu'en affectant directement aux attributs correspondants les valeurs reçues en paramètre. Pourquoi donc procédons-nous ainsi? (voir *Réflexion 00.5 : clients hostiles*).

Détail technique : paramètres avec valeur par défaut

Un gadget technique qui peut s'avérer pratique à l'utilisation, en particulier dans le cas des constructeurs, est la possibilité d'attribuer à des paramètres des valeurs par défaut.

Réexaminons deux constructeurs de `Rectangle`, soit son constructeur par défaut et un constructeur paramétrique. Tels que nous les avons définis, leurs déclarations respectives sont tel que présenté à droite, et leurs définitions sont telles que visibles ci-dessous.

```
class Rectangle {
public:
    // ...
    Rectangle();
    Rectangle(int, int);
    // ...
};
```

```
#include "Rectangle.h"
// ...
Rectangle::Rectangle() {
    SetLargeur(LARGEUR_DEFAULT);
    SetHauteur(HAUTEUR_DEFAULT);
}
Rectangle::Rectangle(int hau, int lar) {
    SetLargeur(lar);
    SetHauteur(hau);
}
// ...
```

On remarque sans peine des similitudes assez fortes entre les deux :

- dans un cas, le constructeur initialise la largeur et la hauteur à l'aide de valeurs par défaut connues à l'avance; alors que
- dans l'autre, le constructeur initialise la largeur et la hauteur à l'aide de valeurs passées lors de l'appel.

La tâche est la même; seules les valeurs changent.

On voudrait tirer profit de ce fait, tout en s'assurant de ne rien perdre côté fonctionnalité et aisance à l'utilisation. L'application de paramètres avec valeurs par défaut permet de fusionner ces deux constructeurs en un seul, sans quelque perte que ce soit. Voici comment :

- on veut que la déclaration `Rectangle r1;`, où les paramètres sont omis à l'appel du constructeur, fasse en sorte que la hauteur et la largeur initiales soient implicitement celles spécifiées par défaut pour un `Rectangle`;
- on veut aussi que la déclaration `Rectangle r2(10,5);`, où les paramètres ont des valeurs explicitement spécifiées, fasse en sorte que la hauteur initiale de `r2` soit 10 et que sa largeur initiale soit 5;
- on veut donc que la valeur par défaut du premier paramètre (la hauteur) soit `Rectangle::HAUTEUR_DEFAULT` et que celle du second paramètre (la largeur) soit `Rectangle::LARGEUR_DEFAULT`.

L'implantation concrète sera donc de remplacer la déclaration des deux constructeurs par une seule, tel que présenté à droite (voir toutefois l'encadré au bas de la présente page), et dont la définition demeurera précisément celle du constructeur paramétrique, qui demeure la plus générale des deux.

On voit donc qu'un constructeur par défaut peut être vu comme un cas particulier de constructeur paramétrique ayant des paramètres avec valeurs par défaut, ces valeurs étant celles propres à une instance par défaut de cette classe.

```
class Rectangle {
public:
    // ...
    Rectangle(int = HAUTEUR_DEFAULT,
              int = LARGEUR_DEFAULT);
    // ...
};

#include "Rectangle.h"
// ...
Rectangle::Rectangle(int hau, int lar) {
    SetLargeur(lar);
    SetHauteur(hau);
}
```

Une nuance s'impose, cependant : dans le cas du constructeur par défaut, les valeurs utilisées pour initialiser l'objet sont connues au préalable et sont nécessairement valides (présumant un programme écrit correctement). Dans le cas du constructeur paramétrique, aucune présomption de validité ne peut être faite quant aux valeurs des paramètres suppléés par le code client.

Conséquemment, le constructeur par défaut pourrait omettre la validation des valeurs utilisées pour initialiser ses attributs, alors que le constructeur paramétrique, lui, ne peut se permettre ce luxe. Plusieurs décideront de garder deux constructeurs distincts pour cette raison : le constructeur par défaut sera simple et rapide, et le constructeur paramétrique prendra soin de se blinder et de valider rigoureusement ses intrants.

Les paramètres par défaut sont surtout utilisés, en pratique, pour définir des constructeurs paramétriques pour lesquels les premiers paramètres doivent être indiqués par le code client, mais pour lesquels les paramètres ultérieurs peuvent se voir attribuer des valeurs par défaut raisonnables. Ce faisant, ils évitent aux programmeuses et aux programmeurs de rédiger un large éventail de constructeurs distincts, et ils évitent au code client de passer une grande quantité de paramètres à la construction des objets, du moins dans bien des cas.

Dans le cas de `Rectangle`, en pratique, je n'utiliserais pas cette technique parce que le constructeur paramétrique a deux paramètres (hauteur, largeur), et qu'il me semblerait étrange de dire qu'il est possible de suppléer une hauteur tout en profitant implicitement de la largeur par défaut, mais par l'inverse. Cela dit, il faut examiner chaque cas au mérite et se poser la question : est-ce que ce que nous offrons comme possibilité sera utile et pertinent pour le code client?

Constructeur de copie

Un cas particulier (mais combien important) de constructeur paramétrique est celui du *constructeur de copie*. Pour comprendre le rôle joué par ce constructeur, il nous faut revenir sur la notion de paramètre passé par valeur.

Si on prend le sous-programme `decupler(int)`, à droite, on remarque que le paramètre `valeur` utilisé par la fonction `decupler(int)` est une *copie* de ce que le sous-programme appelant a offert à la fonction pour accomplir son travail.

```
int decupler(int valeur) {
    valeur *= 10;
    return valeur;
}
```

Ainsi, si on prend les appels à droite (qui présument qu'on a accès au prototype de `decupler()`), on constate que, bien que les divers appels soient tout aussi valides les uns que les autres, il semble y avoir une légère nuance quant à la nature de l'un et de l'autre.

En effet :

```
void demonstration() {
    const int TEST_A = 8;
    int testB = 4;
    int resultat;
    resultat = decupler(TEST_A); // (0)
    resultat = decupler(testB); // (1)
    resultat = decupler(3); // (2)
    resultat = decupler(decupler(5)); // (3)
}
```

- l'appel **(0)** passe en paramètre à `decupler(int)` une copie de la constante `TEST_A`. Lors cet appel, le paramètre `valeur` de la fonction `decupler(int)` aura initialement la valeur 8;
- l'appel **(1)** passe en paramètre à `decupler(int)` une copie de la variable `testB`. Lors cet appel, le paramètre `valeur` de la fonction `decupler(int)` aura initialement la valeur 4; et
- l'appel **(2)** passe en paramètre à `decupler(int)` une copie du littéral 3. Lors de cet appel, le paramètre `valeur` de la fonction `decupler(int)` aura initialement la valeur 3; et
- l'appel **(3)** passe en paramètre à `decupler(int)` le résultat de l'exécution de la fonction `decupler(5)`.

Question : aurait-il été une bonne idée de spécifier le paramètre à `decupler()` comme étant `const int` plutôt que `int`? Pourquoi?

Pour le sous-programme `decupler(int)`, toutefois, dans chaque cas, `valeur` est une variable `int`, ayant pour valeur initiale une copie de ce qu'on a bien voulu lui donner à l'appel. Les détails quant à la provenance de cette valeur⁶³ lui échappent complètement.

⁶³ S'agit-il d'une copie d'une constante? D'une copie d'une variable? D'une copie d'un littéral? D'une copie du résultat de l'exécution d'une fonction? Est-ce même une question pertinente?

La mécanique de copie

Faire une copie d'une donnée dont le type est l'un des types primitifs du langage est tout à fait à la portée du compilateur, qui connaît la taille et la structure interne de chacun d'entre eux⁶⁴.

Faire une copie d'un objet, par contre, est une tâche plus complexe, puisque *ce que signifie « copier » dépend de la manière dont est construit l'objet à copier*. Ceci est surtout vrai si l'objet à copier contient des données ayant été allouées de manière dynamique⁶⁵.

C'est donc la tâche de l'équipe de développement que de spécifier comment on fait pour construire un objet qui soit une copie d'un autre objet, ce qui équivaut à rédiger un constructeur de copie (*Copy Constructor*).

Il est d'usage, dans le monde anglophone, de documenter un tel objet en indiquant qu'il est *CopyConstructible*. Une telle dénomination pour un type est ce qu'on nomme un **concept** [POOv03].

Signature d'un constructeur de copie

Dans la déclaration de la classe, un constructeur de copie apparaîtra comme présenté ci-dessous. Remarquez la signature: on passe au constructeur une référence sur une instance constante de la même classe (dans cet exemple, un `const Rectangle&`).

Cette signature est délibérée. Cherchant à définir comment faire d'un `Rectangle` une copie d'un autre `Rectangle`, on ne peut pas passer le paramètre par valeur⁶⁶, car cela entraînerait une récursivité infinie!

On spécifie la référence comme menant vers une constante du fait que faire une copie ne doit pas modifier l'original. Ceci nous impose toutefois une contrainte : se limiter aux méthodes `const` (par exemple des `Get`) du paramètre.

La définition de cette méthode devra faire de l'instance en cours de construction une copie de celle (déjà construite) passée en paramètre.

Notre classe `Rectangle` présente une illustration typique de cette fonctionnalité (voir à droite).

```
class Rectangle {
    // ...
public:
    // ...
    // -- Constructeur de copie
    Rectangle(const Rectangle&);
    // ...
};
```

```
#include "Rectangle.h"
// ...
Rectangle::Rectangle(const Rectangle &r) {
    SetLargeur(r.GetLargeur());
    SetHauteur(r.GetHauteur());
}
```

Remarquez que les mutateurs (`Set`) sont utilisés sans qu'on indique *explicitement* à qui ils appartiennent. Ceci signifie qu'ils appartiennent à l'instance active de `Rectangle`, propriétaire du constructeur en cours d'exécution, et donc à l'instance qu'on est en train de construire. Les accesseurs (`Get`), eux, appartiennent explicitement à `r`, qui est l'instance de `Rectangle` dont on veut faire une copie.

⁶⁴ Le compilateur sait pertinemment combien de *bytes* sont requis pour encoder un `int`, par exemple, et dans quel ordre ces *bytes* doivent se trouver pour faire du sens.

⁶⁵ Avec l'opérateur `new`, sur lequel nous reviendrons plus loin.

⁶⁶ Il faudrait alors que le compilateur en fasse une copie, nécessitant encore son constructeur de copie, ce qui introduirait une circularité dans la mécanique.

On utilise les constructeurs de copie surtout de manière implicite, à travers le passage en paramètre par valeur⁶⁷ :

```
#include "Rectangle.h"
// DoublerRectangle() recevra un Rectangle passé par valeur
void dessiner_gros(Rectangle r);
// Petit programme de démonstration
int main() {
    Rectangle r(4, 5); // rectangle de démonstration
    r.Dessiner();
    // on passe r par valeur; il faut un constructeur de copie
    dessiner_gros(r);
    r.Dessiner(); // r n'a pas changé
}
// Utilise une copie de l'original (ne modifie donc que la copie)
void dessiner_gros(Rectangle r) {
    r.SetLargeur(r.GetLargeur() * 2);
    r.SetHauteur(r.GetHauteur() * 2);
    r.Dessiner();
}
```

Notez cette version, plus élégante, qui profite mieux des constructeurs de Rectangle :

```
#include "Rectangle.h"
// DoublerTaille() retourne un Rectangle deux fois la largeur et la hauteur de l'original
Rectangle grossir(Rectangle r);
// Petit programme de démonstration
int main() {
    Rectangle r(4, 5); // rectangle de démonstration
    r.Dessiner();
    grossir(r).Dessiner();
    r.Dessiner(); // r n'a pas changé!
}
Rectangle grossir(Rectangle r) {
    return Rectangle(r.GetHauteur() * 2, r.GetLargeur() * 2);
}
```

Depuis C++ 11, il est possible d'écrire `grossir()` plus simplement encore :

```
Rectangle grossir(Rectangle r) { // la fonction retourne un Rectangle, donc...
    return { r.GetHauteur() * 2, r.GetLargeur() * 2 }; // ... ceci appelle un constructeur
} // de Rectangle!
```

Réflexion 00.6 – À quels signes reconnaît-on que les constructeurs exposés par un objet sont en nombre suffisant, sans plus? (voir **Réflexion 00.6 : une saine gamme de constructeurs**).

⁶⁷ Il y a des cas où C++ fera de la véritable magie avec les paramètres par valeur et les constructeurs de copie, à l'aide de son *moteur d'inférence de types*. Nous y reviendrons.

Constructeur de copie et opérateur d'affectation

Notons au passage une subtilité syntaxique de C++. Étant donné une classe `X` quelconque, et prenant en exemple le code proposé ci-dessous :

- la ligne **(0)** sollicite le constructeur par défaut de `X` (donc la méthode `X::X()`);
- la ligne **(1)** sollicite le constructeur par copie de `X`. *Cette ligne est une construction, pas une affectation.* On le voit au fait qu'elle est constituée d'un nom de type, soit `X`, suivi d'un nouveau nom de variable (`x1`). La syntaxe utilisant `=` lors d'une déclaration de variable est une forme compacte d'invocation d'un constructeur à un seul paramètre, que ce soit un constructeur par copie ou un constructeur paramétrique (selon le type de l'opérande de droite);
- la ligne **(2)** sollicite aussi le constructeur par copie, ceci à cause du type de l'objet entre parenthèses (qui est le même que le type de l'objet en cours de construction). Cette écriture est plus polyvalente que celle de la ligne (1) du fait qu'elle permet de passer plusieurs paramètres à la construction et peut être légèrement plus efficace⁶⁸, mais (1) et (2) sont équivalents dans la majorité des cas; enfin
- la ligne **(3)** est une affectation. L'opérande de gauche est déjà déclaré et a donc été pleinement construit, et l'opérateur d'affectation applicable au type `X` sera utilisé à cet endroit pour réaliser la tâche demandée par le programme.

```
class X {
    // ...
};
int main() {
    X x0;           // (0)
    X x1 = x0;     // (1)
    X x2(x0);      // (2)
    x1 = x2;       // (3)
}
```

Lorsque l'opérateur `=` est utilisé à la définition d'un objet, il indique une invocation de constructeur. Lorsqu'il est utilisé sur un objet déjà construit, il réfère à une affectation. Les ramifications techniques de cette nuance seront explorées plus loin, lorsque nous explorerons la surcharge des opérateurs.

Remarque technique au sujet de l'opérande du constructeur par copie

Pour toute invocation implicite du constructeur par copie d'une classe `X` donnée, il est nécessaire que le paramètre passé à ce constructeur soit de type `const X &`. C'est la forme que le compilateur reconnaîtra.

Il est possible d'écrire un constructeur de `X` prenant un `X &` (non `const`) en paramètre. Cependant, ce constructeur cachera le constructeur par copie (le compilateur ne pourra plus les distinguer à l'usage dans la majorité des cas); il ne sera plus possible de passer un `X` par copie à un sous-programme. Évitez donc la forme non `const` sauf pour des applications très spécialisées, très longuement réfléchies (et encore!).

⁶⁸ On dit que `X x1 = x0;` est une opération d'initialisation par copie (*Copy Initialization*) alors que `X x2(x0);` est une opération d'initialisation directe (*Direct Initialization*). Les deux formes sollicitent le constructeur par copie si l'objet construit et le paramètre passé au constructeur sont de même type. La version utilisant l'initialisation directe est celle que les compilateurs parviennent le mieux à optimiser, mais les deux formes sont importantes pour fin de générique (référez-vous à [POOv02] pour en savoir plus au sujet de la programmation générique). Pour ce qui est de ces deux modes d'initialisation, voir *Nuances fines entre conversion implicite et conversion explicite*, plus loin.

Génération automatique du constructeur de copie

Le constructeur de copie est l'une des fonctions clés de la programmation en C++. En effet, contrairement à Java et aux langages .NET qui ne permettent pas au code client d'utiliser des objets directement (les clients n'ont qu'un accès indirect aux objets dans ces langages), C++ supporte une sémantique d'accès direct aux objets (voir *Appendice 00 – Sémantiques directes et indirectes*) et permet au code client de les manipuler aussi naturellement qu'il manipule des données primitives.

Copier un objet soulève une question profonde et subjective : quel est le sens de cette duplication? Et cette question ne peut être répondue que par l'objet lui-même.

Dès qu'un objet est responsable de la gestion d'autres ressources (allocation dynamique de mémoire, gestion de fichiers, accès à des bases de données, *etc.*), l'acte de le copier devient un sujet d'intérêt et mérite une attention particulière.

C++ offre un support spécial à trois méthodes d'instance, qu'on nomme la *Sainte-Trinité* [hdTri], soit le constructeur de copie, l'affectation et la destruction (nous couvrirons les deux dernières un peu plus bas). Si le programme ne déclare pas un constructeur de copie pour un objet, entre autres, C++ le fera pour lui, à partir des constructeurs de copie de ses attributs.

Par exemple :

```
// ... inclusions et using ...
class Rectangle {
    int hauteur_,
        largeur_;
public:
    Rectangle(int hau, int lar);
    int hauteur() const;
    int largeur() const;
};
void afficher(Rectangle);
int main() {
    Rectangle r{ 2,3 };
    afficher(r); // passe r par copie
}
void afficher(Rectangle r) { // copie implicite des attributs (Sainte-Trinité)
    cout << r.hauteur() << endl; // affichera 2
}
```

Ce comportement conviendra aux classes simples comme `Rectangle`; pour cette classe, notre code aura le même impact que le code généré automatiquement par le compilateur. Voir *Annexe 03 – Concepts et pratiques du langage C++* pour plus de détails.

Méthodes `=default`

Pour chaque méthode que le compilateur pourrait générer de lui-même (celles de la Sainte-Trinité, le constructeur par défaut, et quelques autres que nous couvrirons plus tard), il est possible de déclarer la méthode et d'indiquer notre souhait que le compilateur génère le code par défaut lui-même, en insérant le suffixe `= default`. En général, dans ce cas, le « vide » du compilateur sera de meilleure qualité que le nôtre.

Par exemple, avec le constructeur de copie :

Écriture traditionnelle	Écriture <code>=default</code>
<pre>class X { int val; // ... public: X(); X(const X&); // ... etc. }; // ... dans le .cpp X::X() { val = 0; } X(const X &autre) { val = autre.val; } // ... etc.</pre>	<pre>class X { int val; // ... public: X(); X(const X&) = default; // ... etc. }; // ... dans le .cpp X::X() { val = 0; } // ... etc.</pre>

Il est préférable d'utiliser `=default` plutôt que de tenter de reproduire manuellement ce que le compilateur aurait fait naturellement : l'intention est alors plus clairement inscrite dans le code source du programme, et le compilateur est le mieux placé pour faire son propre travail.

Une application de ce mécanisme est de jouer avec la qualification de ces méthodes « spéciales » que le compilateur peut générer de lui-même. Par exemple, si vous souhaitez un constructeur de copie « typique » mais privé, il suffit de le déclarer `private` et de lui apposer `=default`.

Couplé à la préconstruction (voir *Préconstruction*, plus bas) et aux NSDMI (voir *Initialisation immédiate d'attributs d'instance*, plus bas encore), la notation `=default` permet d'adjoindre la concision à l'efficacité.

Cacher des constructeurs

Une classe qui n'exposerait aucun constructeur se fait suppléer automatiquement par C++ un constructeur par défaut et un constructeur de copie. Le constructeur par défaut automatiquement généré invoque les constructeurs par défaut implicites des attributs de l'objet, et le constructeur de copie automatiquement généré invoque les constructeurs de copie de ces attributs.

Dès qu'au moins un constructeur est déclaré dans une classe, le compilateur C++ ne génère plus automatiquement son constructeur par défaut. Déclarer au moins un constructeur est un engagement, de la part des programmeuses et des programmeurs, à contrôler les états initiaux des instances de cette classe. Le compilateur supplée toutefois un constructeur de copie automatiquement généré jusqu'à ce que la classe ne déclare explicitement sa propre version.

S'il est important que les instances d'une classe ne puissent pas être copiées, il faut donc que la classe déclare son propre constructeur de copie et le rende à la fois inaccessible au code client et inutilisable pour lui-même. Pour ce faire, il suffit de déclarer ce constructeur, typiquement de manière privée, *quitte à ne pas le définir*. Ceci indiquera au compilateur de ne pas générer implicitement de constructeur de copie.

Par exemple :

```
class X {
    X(const X&); // constructeur par copie privé
public:
    X();
};
class Y {
public:
    Y(int); // pas de constructeur par défaut
};
void f(X) { } // voir la note technique ci-dessous
void g(Y) { } // idem
void h() {
    X x; // légal: X::X() est public
    f(x); // illégal: X::X(const X&) est privé
    Y y0; // illégal : pas de constructeur par défaut pour Y
    Y y1{3}; // légal
    g(y1); // légal : Y a implicitement un constructeur de copie
}
```

Petite note technique : avec ce programme, si nous avons nommé les paramètres aux fonctions `f()` et `g()` sans les utiliser, notre compilateur nous aurait donné un avertissement (légitime) pour cause de paramètres nommés mais sans être utilisés. C'est pour éviter cet avertissement que les noms des paramètres ont été omis, et que je n'ai gardé que les types. Cette technique fonctionne aussi en C#, d'ailleurs.

Suppression d'un service – méthode = delete

Le simple fait de « cacher » un constructeur (ou, de manière générale, une méthode) en le déclarant privé puis en « omettant » de le définir fonctionne bien mais représente un *Hack*, un tour de passe-passe. Le plus gros défaut de cette approche est le type d'erreur qu'il génère :

- si le code client d'un objet utilise la méthode ainsi « cachée », alors l'erreur qui sera générée à la compilation sera à l'effet que la méthode est inaccessible. Est-ce que ceci décrit bien l'intention de la programmeuse ou du programmeur?
- si l'objet utilise lui-même (par accident) la méthode en question, alors on aura un échec lors de l'édition des liens, phase tardive suivant la compilation, à l'effet que la méthode n'est pas définie. Ici encore, est-ce bien ce que nous souhaitions exprimer?

Depuis C++ 11, il est possible d'explicitement l'intention réelle, qui est de supprimer complètement le service (de construction, dans ce cas-ci) en suffixant sa signature de `= delete`. Ce faisant, une utilisation (accidentelle ou non) du service provoquera une erreur à la compilation, et cette erreur sera claire : le service n'est pas offert.

Par exemple :

```
class X {
    // ...
public:
    X() = default;
    X(const X&) = delete; // constructeur de copie supprimé
};
class Y {}; // la Sainte-Trinité s'applique
void f(X) {}
void f(Y) {}
int main() {
    X x; // Ok, X::X() existe et est accessible
    Y y; // Ok, Y::Y() existe et est accessible
    // f(x); // Non; constructeur de copie de X supprimé
    f(y); // Ok, Y::Y(const Y&) existe et est accessible
}
```

Copies et classes d'objets immuables

Les langages Java et C# se ressemblent beaucoup; en fait, ils sont tous deux en compétition directe pour un même marché. Ils se ressemblent (sans être identiques) dans leur démarche, dans leurs qualités et dans leurs défauts.

L'une des mécaniques qui manquent cruellement à ces deux langages est celle des méthodes `const` au sens C++, c'est-à-dire des méthodes dont l'utilisation n'entraîne pas d'effets secondaires sur les états de l'objet qui les possède.

Dans ces langages, si un objet possède une méthode publique, alors le monde extérieur peut en faire usage. La constance ou non de l'objet importe peu pour ce qui est de la validité ou non des appels de méthodes.

Une des principales raisons derrière cette situation est que Java et C# ne donnent qu'un accès indirect (à travers des références) aux objets.

En Java, par exemple, on peut déclarer une référence constante, mais pas ce à quoi elle réfère. Voir l'*appendice 00* pour plus de détails.

Java et C# ont une stratégie particulière pour éviter le problème de modification indue des objets. Dans ces langages, toute classe pour laquelle on veut garantir la constance de l'objet une fois celui-ci construit doit omettre d'exposer toute opération permettant sa modification.

Par exemple, il existe en Java une classe nommée `Integer` :

- on peut l'instancier par construction paramétrique avec un appel à `new` (p. ex. : `Integer i=new Integer(3);` pour créer l'instance `i` de la classe `Integer` ayant la valeur `3`);
- on peut demander à un `Integer` sa valeur (appel à sa méthode `getValue()`);
- on peut la dupliquer manuellement (p. ex. : `Integer j=new Integer (i);`); mais
- il n'existe pas de manière de modifier la valeur d'une instance d'`Integer` une fois celle-ci construite.

Ainsi, une addition entre deux `Integer` pourrait avoir l'air de ceci⁶⁹ :

```
Integer Somme = new Integer(i.getValue() + j.getValue());
```

Dans les langages Java et C#, les chaînes de caractères (respectivement, les classes `java.lang.String` et `System.String`) sont de telles classes. On dit d'une classe n'offrant aucune méthode permettant de modifier le contenu d'une instance déjà créée qu'elle est **immuable**. Les classes immuables sont essentielles à la réalisation d'une encapsulation correcte en l'absence de véritables constantes : on remplace ici un concept par une technique.

Notez que cette stratégie est viable en Java et en C# parce que le langage offre une collecte d'ordures automatique. Les nombreuses instanciations dynamiques d'objets (avec `new`) ne demandent pas de suivi spécial puisque les objets auxquels nul ne fait référence seront éventuellement éliminés de la mémoire. En retour, l'immutabilité peut être réalisée dans tous les langages permettant de définir des méthodes et de qualifier les attributs privés, C++ inclus.

⁶⁹ Depuis l'avènement de C#, par mimétisme, Java a adopté une stratégie nommée le *Boxing* (qu'on pourrait traduire par « mise en boîte »), pour réaliser implicitement ces conversions. Cela simplifie le code mais le ralentit aussi beaucoup, alors soyez conscient(e)s de vos actions lorsque vous manipulez ces types!

Préconstruction

Une technique toute simple et qui fait parfois des miracles pour la qualité du code est la préconstruction des attributs. L'idée est la suivante :

- si une instance d'une classe donnée possède des attributs, ces attributs doivent être construits avant que le constructeur de l'objet ne débute son exécution, sinon le constructeur ne pourrait pas initialiser les attributs d'instance;
- ceci implique que, si les attributs sont eux-mêmes des objets, leur propre constructeur doit avoir été invoqué au préalable;
- le seul constructeur raisonnable dans les circonstances est le constructeur par défaut de chaque attribut; donc
- le constructeur de l'objet, pour chaque attribut d'instance qu'il initialisera, remplacera les états d'objets déjà construits par de nouveaux états. Chaque attribut sera initialisé deux fois; clairement, il s'agit d'une fois de trop.

La classe `Nom` proposée à droite démontre ce qui se produit si on n'utilise pas la préconstruction. Ici, à l'accolade ouvrante du constructeur paramétrique, il faut que l'attribut qui contiendra la valeur du nom existe déjà, puisque le constructeur (par le mutateur qu'il invoque) y affectera une valeur.

Le constructeur par défaut de `std::string` a donc nécessairement déjà été appelé, mais cette `std::string` par défaut est une valeur bidon, que nous remplaçons immédiatement par la suite. On parle donc de gaspillage de temps d'exécution.

```
#include <string>
using std::string;
class Nom {
    string valeur_;
    void SetValeur(const string &valeur) {
        valeur_ = valeur;
    }
public:
    Nom(const string &valeur) {
        SetValeur(valeur);
    }
    // ...
};
```

Ce gaspillage serait épargné si le bon constructeur pour chaque attribut était invoqué d'office. Construire un objet, c'est (dans l'ordre) :

- réserver un espace suffisant pour le contenir, que ce soit en instanciant la variable de manière automatique, comme dans l'écriture `Cercle c;`, ou en l'instanciant de façon dynamique, `Cercle *c = new Cercle;`;
- construire ses attributs, en utilisant leurs constructeurs par défaut; puis
- solliciter le constructeur de l'objet en cours de construction pour que celui-ci puisse initialiser ses états comme bon lui semble, ce qui lui permet d'implémenter l'encapsulation en assurant son intégrité depuis le tout début de son existence.

Le texte à gauche est incomplet, et le restera jusqu'à ce que nous ayons abordé les concepts d'héritage simple, d'héritage multiple et d'héritage virtuel

Ici, le mutateur ne réalise aucune validation, et (présumant que `Nom` soit immuable, donc qu'on n'envisage pas offrir de mécanismes pour en modifier les attributs) on pourrait donc éliminer complètement le mutateur.

La préconstruction, qui le remplacerait, consisterait à initialiser directement l'attribut à partir de la valeur du paramètre, réalisant ainsi non pas une construction par défaut suivie d'une affectation pour cet attribut mais bien une construction par copie.

```
#include <string>
using std::string;
class Nom {
    string valeur_;
public:
    Nom(const string &valeur)
        : valeur_{valeur}
    {
    }
    // ...
};
```

Et si la valeur utilisée pour initialiser l'attribut devait être initialisée? Plusieurs stratégies sont alors possibles (dont le traitement d'exceptions, sur lequel nous reviendrons plus loin), mais nous pouvons quand même utiliser la préconstruction.

Supposons par exemple que nous ayons une politique stipulant qu'un `Nom` ne peut être vide, donc que toute tentative d'utiliser un `Nom` vide remplace ce nom par une valeur entendue d'avance ("Joe Blo", peut-être, mais peu importe).

Le mutateur peut alors être remplacé par une méthode de classe qui examine les conditions de validation de la valeur en question, testant ici si elle est vide ou non, et retournant (selon le cas) la valeur reçue en paramètre ou son remplacement. J'ai choisi d'utiliser l'opérateur ternaire (l'opérateur `? :`) pour y arriver mais une version de la méthode utilisant deux points de sortie (deux `return`) aurait aussi été possible, quoique moins jolie.

Ici, `Nom` applique la préconstruction à son attribut après avoir fait subir à la valeur candidate pour l'initialiser une vérification des politiques de validation. Le seul coût à l'exécution est un test (le `if`).

```
#include <string>
using std::string;
class Nom {
    string valeur_;
    static const string NOM_DEFAULT;
    static const string& filtrer
        (const string &valeur)
    {
        return valeur.empty()?
            NOM_DEFAULT : valeur;
    }
public:
    Nom(const string &valeur)
        : valeur_{filtrer(valeur)}
    {
    }
    // ...
};
```

Évidemment, si un `Nom` peut changer de contenu suite à sa construction, conserver le mutateur peut être une bonne chose (préconstruction ou pas). À vous de juger.

En pratique, les accesseurs sont *beaucoup* plus utiles que les mutateurs. Une saine pratique d'encapsulation est de privilégier la mutation des états d'un objet à travers les constructeurs par copie (initialisation) et à travers l'affectation. L'accès aux constructeurs des attributs, par voie de préconstruction, permet d'atteindre cet idéal.

Constructeurs de délégation

Depuis C++ 11, il est possible pour un constructeur de déléguer son travail à un autre constructeur. À titre d'exemple, les deux extraits ci-dessous sont équivalents :

C++ 03	C++ 11
<pre>class EntierSpecial { int val_; static int valider(int); public: EntierSpecial(int val) : val_{valider(val)} { } EntierSpecial() : val_{ } { } // ... };</pre>	<pre>class EntierSpecial { int val_; static int valider(int); public: EntierSpecial(int val) : val_{valider(val)} { } EntierSpecial() : EntierSpecial{0} { } // ... };</pre>

Cette façon de faire est pertinente principalement dans le cas où les constructeurs se ressemblent à quelques détails près, permettant alors de réduire la redondance de code.

L'exemple ci-dessus est une mauvaise application du constructeur de délégation. En effet :

- dans le code à gauche, on trouve une distinction entre le constructeur par défaut, qui place dans `val_` une valeur digne de confiance, et le constructeur paramétrique, qui accepte une valeur candidate pour `val_` d'un tiers externe et, par conséquent, qui doit être validée;
- dans le code à droite, le cas général reposant une valeur quelconque se trouve aussi à couvrir le cas digne de confiance. Ceci implique que même la valeur digne de confiance (ici : le littéral `0`) passera par le filtre de la méthode `EntierSpecial::valider()`.

Le constructeur de délégation est un outil dont il est facile d'abuser, mais qui peut simplifier significativement l'entretien du code.

Dans d'autres langages

En Java, comme en C++, un constructeur est une méthode sans type portant le même nom que celle de la classe dont elle doit construire des instances.

Un exemple (abrégé, par souci d'économie) de classe `Rectangle` en Java munie de constructeurs (par défaut, paramétrique, par copie) est proposé à droite.

Remarquez, comme le montre le programme de test (méthode de classe `main()`), que toute instantiation d'un objet en Java doit se faire avec `new`, et que les parenthèses autour des paramètres sont obligatoires, même pour le constructeur par défaut. Java, comme C# et VB.NET, ne permet pas de manipuler un objet directement. Tous les accès à un objet sont faits, dans ces langages, à travers une référence, et l'opérateur `new` a entre autres pour rôle de solliciter le constructeur indiqué et de retourner une référence vers l'objet nouvellement créé.

L'usage C++ de spécifier comme constant le paramètre passé à un constructeur par copie n'a pas d'équivalent en Java (ou en C#, ou en VB.NET) du fait que ces langages n'ont pas de concept équivalent à celui d'une instance constante ou d'une méthode `const`. Les garanties de constance du paramètre passé à un constructeur par copie reposent sur la confiance qu'a l'appelant envers le constructeur par copie, ou sur l'emploi de classes immuables (n'exposant aucune méthode capable de modifier une instance une fois celle-ci construite).

```
public class Rectangle {
    // ...
    private static final int
        HAUTEUR_MIN = 1, HAUTEUR_MAX = 20,
        HAUTEUR_DÉFAUT = HAUTEUR_MIN;
    private static final int
        LARGEUR_MIN = 1, LARGEUR_MAX = 50,
        LARGEUR_DÉFAUT = LARGEUR_MIN;
    public Rectangle() {
        setLargeur(LARGEUR_DÉFAUT);
        setHauteur(HAUTEUR_DÉFAUT);
    }
    public Rectangle(int largeur, int hauteur) {
        setLargeur(largeur);
        setHauteur(hauteur);
    }
    public Rectangle(Rectangle r) {
        setLargeur(r.getLargeur());
        setHauteur(r.getHauteur());
    }
    public static void main(String [] args) {
        // r0 est un Rectangle par défaut (1 x 1)
        Rectangle r0 = new Rectangle(),
        // r1 est un Rectangle 10 x 5
        r1 = new Rectangle(10, 5),
        // r2 est une copie de r0
        r2 = new Rectangle(r0);
        r0.dessiner();
        r1.dessiner();
        r2.dessiner();
    }
}
```

D'ailleurs, notez que le constructeur prenant un `Rectangle` en paramètre n'est pas, ici, un véritable constructeur de copie, puisque `r` est une référence sur un `Rectangle`, pas un `Rectangle`. Les ramifications de ce constat sont explorées dans *Appendice 00 – Sémantiques directes et indirectes*.

Il est aussi possible en Java d'écrire des constructeurs déléguant leur travail à d'autres constructeurs, ceci à l'aide du mot clé `this()` pris comme une méthode.

Cette stratégie a le bon côté de simplifier la rédaction du code. Le constructeur le plus flexible (souvent un constructeur paramétrique) sera implémenté en détail, alors que les autres lui feront simplement référence en lui passant les paramètres appropriés.

```
public class Rectangle {  
    // ...  
    public Rectangle() {  
        this (LARGEUR_DÉFAUT, HAUTEUR_DÉFAUT);  
    }  
    public Rectangle(int largeur, int hauteur) {  
        setLargeur(largeur);  
        setHauteur(hauteur);  
    }  
    public Rectangle(Rectangle r) {  
        this(r.getLargeur(), r.getHauteur());  
    }  
}
```

Le défaut de cette stratégie est qu'elle ne permet pas de tirer avantage de certains savoirs, comme par exemple le fait que les constructeurs par défaut et par copie d'un objet utilisent assurément des valeurs initiales valides pour les attributs (ce que ne peut garantir le constructeur paramétrique), ce qui implique que ces constructeurs pourraient escamoter l'étape de validation de ces valeurs. S'en remettre ultimement au constructeur paramétrique est plus simple, mais est aussi généralement moins rapide à l'exécution.

Java offre aussi un concept nommé constructeur de classe (constructeur `static`) pour réaliser certaines initialisations propres à la classe et à ses attributs et pour mettre en place certains éléments plus globaux. Plus de détails dans la section sur les *Singletons* dans [POOv02].

En C#, sans surprises, le code est presque un calque de celui qu'on retrouve en Java, mis à part l'impossibilité de qualifier de constants les paramètres (mais souvenons-nous que procéder ainsi n'est pas l'usage le plus répandu en Java non plus) et l'habitude de débiter le nom de chaque méthode par des majuscules.

Notez ici aussi que les objets sont créés avec `new` puis manipulés de manière indirecte (à travers des références), et ramassés implicitement par un moteur automatique de collecte d'ordures. Ceci explique qu'il n'y ait pas lieu de prendre des dispositions particulières pour expliciter le caractère indirect du paramètre dans la signature du constructeur par copie.

Comme Java, le langage C# offre aussi un concept nommé constructeur de classe (constructeur `static`) pour réaliser certaines initialisations propres à la classe et à ses attributs et pour mettre en place certains éléments plus globaux.

```
namespace Formes
{
    class Rectangle
    {
        private const int
            LARGEUR_MIN = 1, LARGEUR_MAX = 50,
            LARGEUR_DÉFAUT = LARGEUR_MIN,
            HAUTEUR_MIN = 1, HAUTEUR_MAX = 20,
            HAUTEUR_DÉFAUT = HAUTEUR_MIN;
        // ...
        public Rectangle()
        {
            Largeur = LARGEUR_DÉFAUT;
            Hauteur = HAUTEUR_DÉFAUT;
        }
        public Rectangle(int largeur, int hauteur)
        {
            Largeur = largeur;
            Hauteur = hauteur;
        }
        public Rectangle(Rectangle r)
        {
            Largeur = r.Largeur;
            Hauteur = r.Hauteur;
        }
        public static void Main(string [] args)
        {
            // r0 est un Rectangle par défaut (1 x 1)
            Rectangle r0 = new Rectangle(),
            // r1 est un Rectangle 10 x 5
            r1 = new Rectangle(10, 5),
            // r2 est une copie de r0
            r2 = new Rectangle(r0);

            r0.Dessiner();
            r1.Dessiner();
            r2.Dessiner();
        }
    }
}
```

La mécanique de Java par laquelle un constructeur peut déléguer vers un autre constructeur existe aussi en C#, mais la syntaxe est légèrement différente.

En effet, la syntaxe pour cette délégation d'un constructeur vers un autre est de suivre la parenthèse fermante de la liste des paramètres au constructeur par un `:` puis par un appel à **this ()** en tant que méthode, appel auquel on passera les paramètres souhaités.

Notons que les désavantages susmentionnés demeurent, en C#, les mêmes qu'en Java. Pour un gain de simplicité, le programmeur paie habituellement le prix d'une perte de performance du fait que le constructeur général ne peut profiter de certains *a priori* spécifiques au constructeur par défaut et au constructeur paramétrique, c'est-à-dire la certitude de la validité des valeurs initiales.

```
namespace Formes
{
    class Rectangle
    {
        // ...
        public Rectangle()
            : this(LARGEUR_DÉFAUT, HAUTEUR_DÉFAUT)
        {
        }
        public Rectangle(int largeur, int hauteur)
        {
            Largeur = largeur;
            Hauteur = hauteur;
        }
        public Rectangle(Rectangle r)
            : this(r.GetLargeur(), r.GetHauteur())
        {
        }
        // Main()
    }
}
```

En VB.NET, toujours sans surprises, on retrouvera un mélange de C# et de syntaxe inspirée des versions plus traditionnelles des langages de la lignée VB. Toutefois, la notation utilisée pour y déclarer un constructeur sera nettement différente.

En effet, un constructeur d'instance en VB.NET est une procédure nommée `New`.

Pour le reste, la syntaxe est tout ce qu'il y a de plus traditionnel dans les circonstances.

La forme utilisée pour instancier un `Rectangle` dans le programme de test (`main()`) évoque à la fois celle utilisée dans d'autres langages OO et celle utilisée dans des versions antérieures de VB pour créer un objet COM.

```

Namespace Formes
  Public Class Rectangle
    ' ...
    Public Sub New()
      SetLargeur(LARGEUR_DÉFAUT)
      SetHauteur(HAUTEUR_DÉFAUT)
    End Sub
    Public Sub New(ByVal largeur As Integer, _
      ByVal hauteur As Integer)
      SetLargeur(largeur)
      SetHauteur(hauteur)
    End Sub
    Public Sub New(ByVal r As Rectangle)
      SetLargeur(r.GetLargeur())
      SetHauteur(r.GetHauteur())
    End Sub

    Private Const HAUTEUR_MIN As Integer = 1
    Private Const HAUTEUR_MAX As Integer = 20
    Private Const HAUTEUR_DÉFAUT As Integer = HAUTEUR_MIN
    Private Const LARGEUR_MIN As Integer = 1
    Private Const LARGEUR_MAX As Integer = 50
    Private Const LARGEUR_DÉFAUT As Integer = LARGEUR_MIN

    Public Shared Sub main()
      ' r0 est un Rectangle par défaut (1 x 1)
      Dim r0 As New Rectangle
      ' r1 est un Rectangle 10 x 5
      Dim r1 As New Rectangle(10, 5)
      ' r2 est une copie de r0
      Dim r2 As New Rectangle(r0)

      r0.Dessiner()
      r1.Dessiner()
      r2.Dessiner()

    End Sub
  End Class
End Namespace

```

Il existe aussi, en VB.NET, une syntaxe permettant à un constructeur de déléguer son mandat à un autre constructeur.

Pour y arriver, la chose à faire est d'insérer un appel à `Me.New()` (donc de solliciter explicitement un autre constructeur de l'instance active) et de lui passer les paramètres appropriés.

Les mentions quant à la perte de performance encourue par cette stratégie, évoquées précédemment pour les autres langages tiennent évidemment ici aussi.

```
Namespace Formes
  Public Class Rectangle
    ' ...
    Public Sub New()
      Me.New(LARGEUR_DÉFAUT, HAUTEUR_DÉFAUT)
    End Sub
    Public Sub New(ByVal largeur As Integer, _
      ByVal hauteur As Integer)
      SetLargeur(largeur)
      SetHauteur(hauteur)
    End Sub
    Public Sub New(ByVal r As Rectangle)
      Me.New(r.GetLargeur(), r.GetHauteur())
    End Sub
    ' main()
  End Class
End Namespace
```

Ni Java, ni C#, ni VB.NET ne supportent la mécanique de méthodes ayant des paramètres dont les valeurs sont suppléées par défaut comme le fait C++.

Initialisation immédiate d'attributs d'instance

L'initialisation immédiate d'attributs d'instance (en anglais : *Non-Static Data Member Initialization*, ou NSDMI) est possible depuis C++ 11. Vous remarquerez que nous n'aurons que peu recours à ce mécanisme dans nos exemples, cherchant à développer l'habitude de bien initialiser les états des objets dans les constructeurs, mais voici un bref survol du mécanisme et de son utilité.

Sans NSDMI

```
class Cercle {
    static const float RAYON_DEFAULT;
    float rayon_;
    float centre_x_,
          centre_y_;
public:
    Cercle() : centre_x{}, centre_y{},
              rayon_{RAYON_DEFAULT}
    {
    }
    // valider rayon (omis pour simplicité)...
    Cercle(float rayon)
        : centre_x{}, centre_y{},
          rayon_{rayon}
    {
    }
    Cercle(float rayon, float cx, float cy)
        : centre_x_{cx}, centre_y_{cy},
          rayon_{rayon} // valider...
    {
    }
    // ...
};
```

Avec NSDMI

```
class Cercle {
    static const float RAYON_DEFAULT;
    float rayon_ = RAYON_DEFAULT;
    float centre_x_ = {},
          centre_y_ = {};
public:
    Cercle() = default;
    // valider rayon (omis pour simplicité)...
    Cercle(float rayon) : rayon_{rayon} {
    }
    Cercle(float rayon, float cx, float cy)
        : centre_x_{cx}, centre_y_{cy},
          rayon_{rayon} // valider...
    {
    }
    // ...
};
```

L'idée ici est que les attributs peuvent se voir attribuer une valeur « par défaut », mais les constructeurs peuvent les initialiser avec une autre valeur si cela leur sied. Ce sont d'ailleurs les constructeurs qui ont le dernier mot : par exemple, un appel à `Cercle{3.5f}` initialisera l'attribut `rayon_` du `Cercle` ainsi construit à `3.5f`; la valeur par défaut `Cercle::RAYON_DEFAULT` ne sera pas utilisée du tout dans ce cas.

Mieux vaut être prudentes et prudents avec ce mécanisme, cependant :

- il devient quelque peu facile d'oublier d'initialiser un attribut dans les constructeurs, puisque l'on peut tendre à ne plus les y initialiser de manière systématique; et
- ce ne sont pas tous les types pour lesquels une valeur par défaut « par attribut » est raisonnable. Pensez par exemple à une date, où les valeurs possibles pour un jour dépendent du mois et de l'année.

Dans d'autres langages

En Java, il est permis depuis longtemps d'initialiser les attributs d'instance dès leur déclaration. Les mêmes réserves et la même prudence s'appliquent ici comme avec C++, et pour les mêmes raisons.

```
class Cercle {
    private static final float RAYON_DÉFAUT = 1.0f;
    private float centreX = 0.0f,
                  centreY = 0.0f;
    private float rayon = RAYON_DÉFAUT;
    public Cercle() {
        // rayon, centreX, centreY par défaut
    }
    public Cercle(float rayon) {
        this.rayon = rayon;
        // centreX, centreY par défaut
    }
    public Cercle(float rayon, float cx, float cy) {
        this.rayon = rayon;
        centreX = cx;
        centreY = cy;
    }
}

public class Z {
    public static void main(String [] args) {
        Cercle c = new Cercle();
    }
}
```

En C#, la situation est aussi la même. Le code proposé à droite en fait la démonstration.

```
namespace z
{
    class Cercle
    {
        private const float RAYON_DÉFAUT = 1.0f;
        private float centreX = 0.0f,
                       centreY = 0.0f;
        private float rayon = RAYON_DÉFAUT;
        public Cercle()
        {
            // rayon, centreX et centreY par défaut
        }
        public Cercle(float rayon)
        {
            this.rayon = rayon;
            // centreX et centreY par défaut
        }
        public Cercle(float rayon, float cx, float cy)
        {
            this.rayon = rayon;
            centreX = cx;
            centreY = cy;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Cercle c = new Cercle();
        }
    }
}
```

C# permet aussi d'initialiser les propriétés à la déclaration, avec la syntaxe proposée à droite..

```
namespace z
{
    class Cercle
    {
        private const float RAYON_DÉFAUT = 1.0f;
        public float CentreX { get; private set; } = 0.0f;
        public float CentreY { get; private set; } = 0.0f;
        public float Rayon { get; private set; } = RAYON_DÉFAUT;
        public Cercle()
        {
            // Rayon, CentreX et CentreY par défaut
        }
        public Cercle(float rayon)
        {
            Rayon = rayon;
            // CentreX et CentreY par défaut
        }
        public Cercle(float rayon, float cx, float cy)
        {
            Rayon = rayon;
            CentreX = cx;
            CentreY = cy;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Cercle c = new Cercle();
        }
    }
}
```

Enfin, **VB.NET** permet aussi l'initialisation des attributs à la déclaration. En ce sens, les quelques langages explorés ici se ressemblent tous.

```
Module Z
    Public Class Cercle
        Private Const RAYON_DÉFAUT As Single = 1.0F
        Private centreX As Single = 0.0F
        Private centreY As Single = 0.0F
        Private rayon As Single = RAYON_DÉFAUT
        Public Sub New()
        End Sub
        Public Sub New(rayon As Single)
            Me.rayon = rayon
        End Sub
        Public Sub New(rayon As Single, cx As Single, cy As Single)
            Me.rayon = rayon
            centreX = cx
            centreY = cy
        End Sub
    End Class

    Sub Main()
        Dim c As Cercle = New Cercle
    End Sub
End Module
```

Assertions statiques et robustesse du code

Ce qui suit est une thématique de C++ 11. Cette thématique peut être implémentée en C++ 03, mais demande pour ce faire que l'on comprenne certaines techniques qui dépassent ce que nous avons couvert jusqu'ici. Si votre compilateur C++ ne supporte pas encore le mot-clé `static_assert`, alors vous ne pourrez pas tirer profit de cette section pour le moment.

La section sur les constructeurs par défaut met de l'avant des constantes pour les valeurs minimale et maximale possibles pour une hauteur et pour une largeur, de même qu'une valeur par défaut dans chaque cas.

Il est entendu que la valeur par défaut devrait se situer inclusivement entre les valeurs minimale et maximale dans chaque cas. Cela semble évident; pourtant, les programmeuses et les programmeurs font des erreurs, comme tout le monde. Les humains sont distraits, préoccupés, et vont parfois trop rapidement.

Il est possible de mettre en place des espèces de chiens de garde à même le code pour valider certaines assertions, certains *a priori* qui doivent s'avérer pour qu'un programme soit correct. Dans le cas de la valeur d'une constante statique devant être située inclusivement entre les valeurs d'autres constantes statiques, nous avons un privilège : toutes les informations sur lesquelles nous baserons notre jugement quant à la validité du programme sont connues au moment de la compilation du programme. En effet, la condition de validité pour la hauteur par défaut des instances de `Rectangle` va comme suit :

```
Rectangle::HAUTEUR_MIN <= Rectangle::HAUTEUR_DEFAUT &&  
Rectangle::HAUTEUR_DEFAUT <= Rectangle::HAUTEUR_MAX
```

Il est donc possible de faire en sorte que le programme ne compile que si ses conditions de validité sont rencontrées. Ce faisant, nous protégeons notre programme contre nos propres bêtises, et nous y arrivons sans que cela ne coûte quoi que ce soit à l'exécution.

C++ permet donc de valider des conditions connues à la compilation avec une assertion statique, ou `static_assert`. Une assertion statique se compose de deux éléments :

- la condition devant s'avérer, qui doit être statique (connue à la compilation du programme); et
- le message d'erreur à produire si la condition ne s'avère pas (pour faciliter le dépiage du problème quand il y a plusieurs candidats possibles).

Dans notre cas, pour profiter de cette sécurité, nous écrivons ceci (probablement quelque part dans le fichier source `Rectangle.cpp`, une fois les constantes à tester définies :

```
// ... divers trucs, dont probablement la définition des constantes ...  
static_assert(Rectangle::HAUTEUR_MIN <= Rectangle::HAUTEUR_DEFAUT &&  
             Rectangle::HAUTEUR_DEFAUT <= Rectangle::HAUTEUR_MAX,  
             "Dans Rectangle, HAUTEUR_DEFAUT n'est pas entre HAUTEUR_MIN et MAX");  
// ...
```

Le moteur d'inférence de types

Comme la plupart des langages OO, C++ est muni d'un puissant moteur d'inférence de types, qui lui permet (entre autres) de déduire à la compilation des opérations qui ne sont pas nécessairement évidentes ou élémentaires.

Prenons par exemple la classe `Cercle` (abrégée et simplifiée) telle que proposée dans la déclaration suivante :

```
class Cercle {
    // ...
public:
    Cercle(); // constructeur par défaut, cercle unitaire centré à l'origine
    Cercle(float rayon); // constructeur paramétrique, origine {0,0}
    // constructeur paramétrique, origine {Centre_X,Centre_Y}
    Cercle(float rayon, float centre_X, float centre_Y);
    Cercle(const Cercle&); // constructeur de copie
    void dessiner() const; // etc.
    // ...
};
```

Présumons aussi qu'on ait le prototype de sous-programme suivant :

```
// Demande au cercle passé en paramètre de se
// dessiner, et affiche un petit texte descriptif
// (p. ex. : Cercle de rayon ... à la position ...)
void afficher(const Cercle&);
```

Examinons brièvement le programme ci-dessous :

```
int main() {
    Cercle c1, // cercle unitaire, centré à l'origine (le cercle « typique »)
           c2{3}, // rayon 3, origine {0,0}
           c3(5, 1, -8.4f); // rayon 5, origine {1,-8.4}
    afficher(c1); // ok
    afficher(c2); // ok
    afficher(c3); // ok
    afficher(8.3f); // ceci est-il légal?
}
```

Que penser de l'appel `afficher(8.3f)`? Est-il légal?

Le moteur d'inférence de types en action

En fait, il s'agit effectivement d'un appel légal. Voici un peu comment le moteur d'inférence de types du langage attaque ce problème :

- il constate un appel à `afficher()` avec un seul paramètre (de type `float`);
- il constate qu'il existe au moins un prototype pour un sous-programme nommé `afficher()` prenant un seul paramètre, mais que les types des paramètres ne concordent pas entre l'appelant et le candidat à être appelé;
- il procède à essayer d'*unifier* les types en jeu⁷⁰, et constate qu'il existe une manière de construire le type requis par le sous-programme à partir du type fourni à l'appel : il existe un constructeur de `Cercle` prenant un `float` (constant, dans ce cas-ci) en paramètre;
- il considère donc l'appel comme légal. Cet appel générera toutefois un appel implicite du constructeur `Cercle::Cercle(const float)`; qui est un constructeur paramétrique capable de convertir implicitement un `const float` en `Cercle` (un constructeur de conversion), avant de réaliser l'appel à la fonction `afficher()`. Il faut en être conscient pour être efficace;
- une fois construit le `Cercle` temporaire résultant, l'appel en soi est effectué.

On le voit, le moteur d'inférence de types est capable, *pour les constructeurs à un seul paramètre*, de générer par lui-même une *brève* séquence de construction menant d'un type à un autre, dans la mesure où il existe une telle séquence, donc dans la mesure où les constructeurs requis pour ce faire existent bel et bien.

Le moteur d'inférence de types de C++ se limite à un seul niveau d'inférence. S'il ne parvient pas à joindre un type et un constructeur lors de la première inférence tentée, il considère avoir échoué dans la tentative.

Limites du moteur d'inférence

C'est là une décision de nature pragmatique : limiter le nombre de niveaux d'inférence à un seul accélère la compilation; aussi, à l'usage, on remarque que les situations où le moteur d'inférence de types devait y aller de plus d'un niveau d'inférence découlaient surtout d'erreurs de programmation.

À quoi cela sert-il?

On a recours aux constructions implicites bien plus souvent qu'on ne le pense. Imaginons par exemple le très pertinent extrait de code proposé à droite.

```
#include <string>
#include <iostream>
int main() {
    using namespace std;
    string mot;
    cout << "Entrez un mot: ";
    if (cin >> mot && mot == "MAGIE")
        cout << "C'est le mot magique!\n";
    cout << "Au revoir!" << endl;
}
```

L'expression en caractères gras compare une instance de `std::string`, `mot`, avec un `const char*`, le littéral "MAGIE". Il s'agit d'une expression valide.

⁷⁰ Les stratégies pour ce faire sont intéressantes, mais débordent de beaucoup le cadre de ce cours.

Il faut comprendre, par contre, qu'il est possible que personne n'ait rédigé une opération spécifiquement pour comparer un `std::string` avec un `char*`, ou avec un tas d'autres types différents. S'il fallait dupliquer toutes les opérations possibles sur un type donné pour chaque type avec lequel il risque d'être comparé (ou autrement manipulé), la redondance des opérations deviendrait très rapidement impossible à gérer.

En fait, pour être honnête, certains types (dont `std::string`) font diverses versions de leurs constructeurs et de leurs opérations pour réaliser des optimisations spécialisées. Cependant, il s'agit de raffinements techniques, pas d'une obligation conceptuelle.

Les concepteurs de `std::string` peuvent simplement exposer un constructeur de `std::string` prenant un `const char*` en paramètre (de même que d'autres prenant en paramètre d'autres types à partir desquels on a jugé raisonnable de construire des `std::string`), et laisser le moteur d'inférence de types découvrir, à la compilation (donc de manière relativement efficace), les conversions à entreprendre.

Construction implicite ou construction explicite

Le prix à payer pour avoir recours aux services du moteur d'inférence de types du langage est d'être prudent. Chaque fois que le moteur entre en scène, une construction (au moins) est entreprise, générant ainsi une variable temporaire qui devra aussi être détruite⁷¹. Au moins deux appels de sous-programmes (un pour le constructeur, un autre pour le destructeur) sont sollicités à chaque fois.

Si vous craignez de devenir imprudent(e), si les pratiques de votre entreprise sont telles que laisser le moteur d'inférence de types prendre charge trop souvent deviendrait pénalisant sur le plan de la qualité du produit, ou si certaines constructions implicites vous sembleraient susceptibles d'être sources d'erreurs difficiles à dépister (le cas de la fonction `afficher()`, plus haut, est patent : est-il clair que le code client veut afficher un Cercle en invoquant `afficher(8.3f)`?), il est possible d'empêcher la construction *implicite* en préfixant la déclaration des constructeurs à un seul paramètre ciblés de la mention `explicit`.

⇒ Un **constructeur explicite** (mention **explicit**) ne peut être utilisé pour fins de conversion implicite de types par le moteur d'inférence de types du langage.

⁷¹ Voir *Destructeur*, plus bas.

Si nous reprenons la classe `Cercle` examinée plus haut, et si notre désir est d'éviter la construction implicite d'un `Cercle` à partir d'un `float`, on ajoutera le mot clé `explicit` à la déclaration du constructeur approprié :

```
class Cercle {
    // ...
public:
    explicit Cercle(float rayon);
    // ...
};
```

Ceci impose un changement à l'utilisation des instances de `Cercle`, bien entendu. Ainsi, le programme suivant devient illégal puisqu'il repose sur une construction implicite de `Cercle` à partir d'un `float`.

Par contre, on peut toujours procéder explicitement à la construction d'un `Cercle` à partir d'un `float`. De ce fait, le programme suivant est légal puisque le `Cercle` est explicitement construit par le code client à l'invocation de la fonction.

La mention `explicit` s'applique de manière générale, incluant pour les valeurs de retour des fonctions. Par exemple, à droite, l'expression (A) ne compilera pas si le constructeur est `explicit`, alors que l'expression (B) compilera dans un tel cas.

```
// ...
void afficher(const Cercle&);
int main() {
    afficher(8.3f); // illégal
}
```

```
// ...
void afficher(const Cercle&);
int main() {
    afficher(Cercle{8.3f}); // légal
}
```

```
// ...
Cercle creer_cercle() {
    // return { 8.3f }; // (A)
    return Cercle{ 8.3f }; // (B)
}
```

Nuances fines entre conversion implicite et conversion explicite

Un impact un peu particulier de la possibilité d'exprimer un constructeur `explicit` est que cela introduit une nuance entre *conversion* explicite et implicite, ce qui influence dans certains cas pathologiques la syntaxe de la construction elle-même. Ceci peut introduire des bogues sophistiqués dans certaines situations (voir [POOv02], section *Pointeurs intelligents et objets partageables* pour un exemple).

Prenons à titre d'exemple la classe `Entier` proposée à droite. Elle offre un constructeur paramétrique qualifié d'explicite, ce qui signifie qu'il n'est pas possible de prendre implicitement un `int` pour un `Entier` si la situation semble s'y prêter.

Nous l'avons vu plus haut, une telle manœuvre peut avoir du sens pour des raisons de performance (éviter des copies inutiles) ou de sémantique (là où la conversion implicite introduirait confusion ou ambiguïté).

Elle peut aussi introduire un certain seuil de sécurité supplémentaire, ce sur quoi nous reviendrons.

Cependant, contrairement peut-être aux attentes, le programme principal à droite ne compilera pas du fait que la ligne déclarant la variable `e3` est illégale, *même si elle joue essentiellement le même rôle que la ligne déclarant `e1`*.

Il faut comprendre ici que la déclaration de `e1` constitue une invocation *explicite* de `Entier::Entier(int)`. La déclaration de `e3`, de son côté, est une invocation *implicite* de `Entier::Entier(const int)`. Le même constructeur est invoqué dans les deux cas mais la manière d'invoquer l'un ou l'autre diffère, et il en va de même pour la mécanique impliquée dans la construction.

```
class Entier {
    int valeur_;
public:
    Entier();
    explicit Entier(int);
    Entier(const Entier&);
    int GetValeur() const;
    // etc.
};
```

```
int main() {
    Entier e0;
    Entier e1{3};
    Entier e2{e1};
    Entier e3 = 3;
}
```

Dans d'autres langages

En Java, une mécanique de conversion automatique (*Boxing*) entre les types primitifs (tels que `int`) et les classes correspondantes dans le monde objet (p. ex. : `Integer`) est en place depuis Java 1.5 et directement inspirée de ce que propose C#.

Le problème de fond est que les langages .NET et Java offrent tous deux un système de types hétérogène où les types primitifs et les objets ont un support différent, ce qui complique certaines opérations comme créer un conteneur capable de prendre diverses sortes d'entités, ce qui, en Java, demande traditionnellement un petit effort supplémentaire lorsqu'il faut y insérer autre chose que des objets.

La construction implicite et automatique est un support spécial pour mieux intégrer les types primitifs au système de types OO du langage.

En plus de ce support spécial, Java offre aussi un support spécial pour le type `String` où tout objet et tout type primitif peut être converti en `String` ou concaténé avec une `String`.

Ce support est ce qu'on nomme parfois du *Compiler Magic* ou du sucre syntaxique, et ne peut être reproduit naturellement pour d'autres types à l'intérieur de ce langage de programmation (on ne pourrait par exemple pas permettre de conversion implicite d'un tableau de paires d'entiers en `Polygone` sauf en appelant explicitement `new Polygone()` et en lui passant le tableau de paires d'entiers en paramètre).

En C#, la situation est la même qu'en Java. Notons que C# est l'instigateur de la technique du *Boxing*. C# n'offre pas d'inférence de types capable de trouver la correspondance entre un type et un autre et capable de solliciter implicitement le bon constructeur au besoin.

Il existe en C# plusieurs mécaniques capables d'alléger la tâche du programmeur à l'aide de tours de magie, comme par exemple celle déjà mentionnée qui crée des tableaux d'objets implicitement à l'aide de certaines syntaxes, mais il n'y a pas d'inférence de types en tant que telle.

```
public class X {
    private int valeur;
    public X(int val) {
        valeur = val;
    }
    public int getValeur() {
        return valeur;
    }
    public static void f(X x) {
        System.out.println(x.getValeur());
    }
    public static void main(String [] args) {
        f(new X(3)); // légal
        f(4);       // illégal
    }
}
```

```
namespace z
{
    public class X
    {
        public int Valeur
        {
            get; private set;
        };
        public X(int valeur)
        {
            Valeur = valeur;
        }
        public static void f(X x)
        {
            System.Console.WriteLine
                (x.Valeur);
        }
        public static void Main()
        {
            f(new X(3)); // légal
            f(4);       // illégal
        }
    }
}
```

En VB .NET, on s'en doutera, la situation est identique à celle rencontrée en C# pour ce qui est de l'inférence de types.

Détail important: si le paramètre passé à `X.Test()` avait été `x` as `X` plutôt que `p` as `X`, alors VB.NET aurait quand même été capable de différencier le petit `x` du grand `X` à partir du contexte.

Cela dit, ce serait vraiment courir après les ennuis que de procéder ainsi, alors simplifiez-vous la vie et ne différenciez pas vos noms en VB.NET à l'aide d'un simple passage de majuscule à minuscule et inversement.

```
Namespace z
  Public Class X
    Private valeur As Integer
    Public Function GetValeur() As Integer
      GetValeur = valeur
    End Function
    Public Sub New(ByVal val As Integer)
      valeur = val
    End Sub
    Public Shared Sub Test(ByVal p As X)
      System.Console.WriteLine(p.GetValeur())
    End Sub
    Public Shared Sub Main()
      Test(New X(3)) ' légal
      Test(4)       ' illégal
    End Sub
  End Class
End Namespace
```

Destructeur

La destruction d'un objet, qui intervient au moment où la vie de cet objet se termine, sollicite chez lui une méthode particulière qu'on nomme le *destructeur*.

⇒ Un **destructeur** est un sous-programme dont la tâche est de clore tout ce que l'objet en cours de destruction risque de laisser en plan : fermer des fichiers, conclure des sessions de télécommunication, libérer de la mémoire allouée dynamiquement, *etc.*

⇒ Certains les nomment aussi des *finisseurs* (*Finalizers*), du fait qu'ils sont plus des moments de nettoyage que de destruction, mais le mot destructeur est l'expression consacrée.

La syntaxe utilisée pour déclarer un destructeur est indiquée à droite.

Les éléments importants à en retenir sont :

- la composition du nom du destructeur, soit le symbole `~` suivi du nom de la classe;
- l'absence de paramètres;
- l'absence d'un type de valeur de retour, similaire à ce que l'on connaît pour la syntaxe des constructeurs; et
- le fait qu'il soit public.

```
class Rectangle {
public:
    // ...
    // -- destructeur
    ~Rectangle();
    // ...
};
```

La définition d'un destructeur doit contenir le code requis pour que, une fois l'objet détruit, le système soit rétabli essentiellement dans le même état (côté ressources) qu'avant sa construction. Évidemment, un objet ne contrôle que ce qui tombe sous sa gouverne, mais il doit agir en bon citoyen et faire en sorte d'éviter que son passage dans un système n'entraîne une dégradation de la qualité de ce système.

⇒ **Toute classe n'aura qu'un seul destructeur**, qui sera appelé lors de la fin de la vie d'un objet.

Une nuance s'impose : en C++, les destructeurs sont déterministes et sont invoqués quand la vie d'un objet se termine; dans les langages où intervient une collecte automatique d'ordures, comme Java et les langages .NET, la finalisation est par défaut indéterministe, et les questions qui y ont trait sont beaucoup plus complexes.

Notre `Rectangle` n'ayant pas sollicité de ressources particulières, son destructeur sera assez simple (à droite).

Dans un tel cas, on aurait même pu l'omettre, ou encore lui apposer plus explicitement `=default`.

```
#include "Rectangle.h"
// ...
Rectangle::~Rectangle() {
}
```

Nous explorerons un peu plus bas un exemple où le destructeur est essentiel.

Ainsi, dans le sous-programme ci-dessous, le `Rectangle` par défaut `r` débutera sa vie utile à la ligne **(0)**, où son constructeur par défaut sera appelé, et terminera cette vie à la ligne **(1)**, où son destructeur sera invoqué⁷². Aux yeux de plusieurs programmeuses et programmeurs C++, l'accolade fermante, qui introduit une finalisation déterministe des objets, est l'une des plus grandes forces de ce langage.

Notez que les destructeurs ont un rôle crucial à jouer dans le bon fonctionnement d'un système, aidant à le maintenir en équilibre et à éviter la dégradation des ressources.

Le fait que le `Rectangle` soit une classe simple ne doit pas nous donner l'illusion que le destructeur est une méthode sans importance.

```
#include "Rectangle.h"
void f() {
    Rectangle r; // (0)
    // ...
} // (1)
```

Pendant l'exécution d'un destructeur, l'objet en cours de destruction n'est pas considéré constant, même si cet objet avait été qualifié `const`. De même, il est possible d'appliquer l'opérateur `delete` à un pointeur constant.

Exemple où le destructeur est essentiel

Imaginons une classe `Notes` servant à représenter les notes d'un élève, mais conçue de manière à ce que le nombre de notes pour un élève ne soit connu qu'au moment de la construction. Ceci permet à chaque instance de `Notes` de représenter un nombre différent de résultats scolaires.

On doit dans un tel cas avoir recours à la mécanique d'allocation dynamique de mémoire : le constructeur allouera dynamiquement un tableau de `float` ou d'un autre type adéquat.

Puisque l'objet aura sollicité des ressources du système, il lui faudra, dans son destructeur, les libérer. Ne pas définir de destructeur dans un tel cas impliquerait (a) ne pas libérer les ressources (ce qui signifie que l'objet est un mauvais citoyen du système); ou (b) offrir des services permettant au code client de libérer explicitement les ressources (ce qui signifie une encapsulation incomplète).

Même dans un langage reposant sur une collecte automatique d'ordures, ce problème demeure entier, puisque les ressources ne se limitent pas à la mémoire (pensez aux fichiers, aux connexions réseaux et aux bases de données). S'en remettre au code client pour libérer les ressources attribuées par un objet est courant dans certains cercles, faute de pouvoir mieux faire, mais il s'agit de bien piètre POO.

⁷² Une manière simple et amusante de vérifier ceci est d'insérer des écritures à l'écran (utilisations de `std::cout`) dans les constructeurs et destructeurs, ce qui permet de suivre le tout de façon visuelle. Évidemment, ceci ne vaut que pour la période d'apprentissage ou de déverminage; il est important de ne pas laisser de telles traces dans du code de production.

Puisque les constructeurs servent habituellement à préparer les états d'un objet et puisque le destructeur a souvent pour rôle de nettoyer les états dont l'objet est responsable, il est utile, pour comprendre les destructeurs, d'examiner en parallèle ces deux extrémités de la vie d'un objet.

Prenant un constructeur en particulier, prenant en paramètre le nombre de notes à supporter (qu'on supposera ici supérieur à zéro pour simplifier; le code complet, plus bas, est plus rigoureux), on verra quelque chose de semblable à ce qui est présenté à droite...

De même, le constructeur de copie allouera lui aussi de la mémoire de la même façon.

La méthode `GetNbNotes()` sert à garder trace du nombre de notes supportées par cette instance de `Notes`. Il ne semblait pas raisonnable de permettre la modification individuelle du nombre de notes une fois celle-ci déterminée, ce qui explique l'absence d'un `SetNbNotes()`.

Le destructeur, ici, sera plus occupé que dans les cas bénins couverts précédemment, servant de contrepois à l'emploi de `new` tel qu'il apparaît dans les constructeurs de cette classe.

```
Notes::Notes(unsigned int nbNotes)
: nbNotes{nbNotes} {
    notes_ = new float[GetNbNotes()];
    // initialisation des notes à 0.0f
    for (unsigned int i = 0; i < GetNbNotes(); ++i)
        SetNote(i, 0.0f);
}
```

```
Notes::Notes(const Notes &n)
: nbNotes{ n.GetNbNotes() } {
    notes_ = new float[GetNbNotes()];
    // copie des notes de « n »
    for (unsigned int i = 0; i < GetNbNotes(); ++i)
        SetNote(i, n.GetNote(i));
}
```

```
Notes::~Notes() {
    delete[] notes_;
}
```

Si vous êtes perverse ou pervers, vous aurez peut-être remarqué qu'il manque une opération clé à la classe que nous sommes en train d'implémenter. Laquelle?

L'interface complète de cette classe suit :

```

#ifndef NOTES_H
#define NOTES_H

// Fichier Notes.h
// Fait par votre professeur adoré
// Le 3 janvier 2001
// Interface de la classe Notes

//
// Cas d'exceptions possibles (nous reviendrons sur cette technique)
//
class HorsBornes {};
class NbNotesIllegal {};

//
// Classe Notes
// Représente un ensemble ordonné de résultats numériques (nombres à virgule flottante,
// simple précision). Ne valide pas les valeurs individuelles des notes. Le nombre de
// notes supporté est fixé à la construction et ne peut plus être modifié par la suite.
//
class Notes {
    // Mutateur SetNbNotes(const unsigned int)
    // Intrants: le nombre de notes à entreposer
    // Extrants: ---
    // Pour usage interne seulement. Si le nombre de notes reçu en paramètre est nul,
    // on prendra Notes::NB_NOTES_DEFAULT à la place
    void SetNbNotes(unsigned int);
public:
    // Constructeur par défaut / paramétrique
    // Intrants: le nombre de notes à supporter
    // Réserve l'espace requis pour entreposer le nombre de notes indiqué en paramètre
    Notes(unsigned int = NB_NOTES_DEFAULT);
    // Constructeur de copie. Duplique les notes de l'instance originale
    Notes(const Notes&);
    // Accesseur GetNbNotes()
    // Intrants: ---
    // Extrants: le nombre de notes supportées
    // Retourne le nombre de notes supportées par cette instance
    unsigned int GetNbNotes() const;
    // Accesseur GetNote (const unsigned int)
    // Intrants: indice de la note désirée
    // Extrants: la note demandée
    float GetNote(unsigned int laquelle) const;
    // Mutateur SetNote (const unsigned int, const float)
    // Intrants: indice de la note à modifier
    //          valeur à lui appliquer

```

```

// Extrants: ---
// Change la note à l'indice spécifié (seulement s'il est valide)
void SetNote(unsigned int laquelle, float valeur);
// ---- Destructeur
~Notes();
private:
// Le nombre de notes supporté par défaut
static const unsigned int NB_NOTES_DEFAULT;
// Le nombre de notes supporté par cette instance
unsigned int nbNotes_;
// Les notes à proprement dit
float *notes_;
};

#endif

```

Et pour ce qui est de son implémentation :

```

// Fichier Notes.cpp
// Fait par votre professeur adoré
// Le 3 janvier 2001
// Implémentation de la classe Notes

#include "Notes.h" // Interface de la classe Notes

// Le nombre de notes supporté par défaut
const unsigned int Notes::NB_NOTES_DEFAULT = 10; // arbitraire (mais > 0)

// Constructeur par défaut / paramétrique
// Intrants: le nombre de notes à supporter
// Réserve l'espace requis pour entreposer le nombre de notes indiqué en paramètre
Notes::Notes(unsigned int nbNotes) : nbNotes_{ nbNotes }{
    notes_ = new float[GetNbNotes()];
    // initialisation des notes à 0.0f
    for (unsigned int i = 0; i < GetNbNotes(); ++i)
        SetNote(i, 0.0f);
}
// Constructeur de copie
Notes::Notes(const Notes &n) : nbNotes_{ n.GetNbNotes() } {
    notes_ = new float[GetNbNotes()];
    // copie des notes de "n"
    for (unsigned int i = 0; i < GetNbNotes(); ++i)
        SetNote(i, n.GetNote(i));
}
// Accesseur GetNbNotes()
// Intrants: ---
// Extrants: le nombre de notes supportées
// Retourne le nombre de notes supportées par cette instance

```

```
unsigned int Notes::GetNbNotes() const {
    return nbNotes_;
}
// Accesseur GetNote(unsigned int)
// Intrants: indice de la note désirée
// Extrants: la note demandée
float Notes::GetNote(unsigned int laquelle) const {
    // valider que laquelle < GetNbNotes() (omis pour le moment)
    return notes_[laquelle];
}
// Mutateur SetNote(unsigned int, float)
// Intrants: indice de la note à modifier
//          valeur à lui appliquer
// Extrants: ---
// Change la note à l'indice spécifié
void Notes::SetNote(unsigned int laquelle, float valeur) {
    // valider que laquelle < GetNbNotes() (omis pour le moment)
    notes_[laquelle] = valeur;
}
// Destructeur
Notes::~Notes() {
    delete[] notes_;
}
```

Dans d'autres langages

La mécanique de destruction, ou son équivalent local, constitue l'un des points pour lesquels les divers langages OO commercialement répandus se distinguent les uns des autres.

En premier lieu, notons que ces trois langages bénéficient d'un moteur de collecte automatique des ordures. Cela signifie qu'un objet, créé avec `new`, sera éventuellement ramassé lorsqu'il ne sera plus référencé par une mécanique sur laquelle le programme *peut* avoir un certain contrôle mais qu'il ne voudra habituellement pas contrôler.

Notez que les tableaux en Java sont des objets, et connaissent leur propre taille. La méthode `getNbNotes()`, visible à droite, en tire profit.

Cela implique qu'on ne sait habituellement pas quand un objet sera effectivement détruit, et qu'on ne peut donc pas compter sur une mécanique symétrique de destruction pour assurer l'occurrence de certaines opérations de nettoyage lorsqu'un objet local cesse d'être référencé par qui que ce soit⁷³.

En Java, il est possible d'écrire un finisseur, soit une méthode d'instance qualifiée `protected`, nommée `finalize()` et de type `void`. Cette méthode *peut* être appelée par le moteur de collecte d'ordures lorsque l'objet auquel elle appartient est ramassé, mais le moteur de Java n'offre aucune garantie qu'elle sera appelée. Cela signifie qu'on ne peut pas, en pratique, compter sur cette méthode.

```
public class Notes {
    public Notes() {
        this(NB_NOTES_DÉFAUT);
    }
    public Notes(final int nbNotes) {
        notes_ = new float[nbNotes];
        for (int i = 0; i < getNbNotes(); i++)
            setNote (i, 0.0f);
    }
    public Notes(Notes n) {
        notes_ = new float[n.getNbNotes()];
        for (int i = 0; i < getNbNotes(); i++)
            setNote(i, n.getNote(i));
    }
    public int getNbNotes() {
        return notes_.length;
    }
    public float getNote(int i) {
        // valider que i < getNbNotes() (omis)
        return notes_[i];
    }
    public void setNote(int i, float val) {
        // valider que i < getNbNotes() (omis)
        notes_[i] = val;
    }
    private static final int NB_NOTES_DÉFAUT = 10;
    private float [] notes_;
}
```

Dans la mesure où un programme Java ne contient que des classes Java et des instances Java, le fait que `finalize()` ne soit pas nécessairement appelé pour un programme donné ne pose pas vraiment problème, du fait que le moteur lui-même libérera, éventuellement, les ressources du programme.

⁷³ La situation est complexe et difficile à traiter ici. Pour en savoir plus, voir [hdebSym].

C'est en milieu hétérogène, comme dans les systèmes client/ serveur ou les systèmes mêlant Java et d'autres langages, que cette *absence de garantie* devient un poids à porter et rend la programmation en Java beaucoup plus complexe qu'elle ne devrait l'être. En Java, la gestion de toute ressource externe (connexion à une base de données, objets écrits dans un autre langage, fichier, et ainsi de suite) doit être faite manuellement, souvent par le code client.

La version Java de la classe `Notes` que nous avons proposée en C++ (un peu plus haut) n'a pas besoin de finisseur puisque le tableau de `float`, servant à entreposer les notes, est un objet Java à part entière, qui sera éventuellement ramassé par le moteur de collecte automatique d'ordures. De même, `setNbNotes()` n'y a pas de sens du fait que les tableaux Java sont des objets et connaissent leur taille (donnée par l'attribut d'instance constant nommé `length`). Ces détails expliquent les différences entre les interfaces des versions C++ et Java de la classe.

Dans l'exemple `Notes` en Java, le fait que `float` soit un type primitif facilite le traitement au sens où le tableau `notes_` contient véritablement des `float`. Quand un tableau Java *contient* des objets, cela implique qu'il contienne en fait des références sur des objets et que ceux-ci doivent être instanciés individuellement. Nous verrons de tels cas plus loin dans ce document.

En C#, il est possible de rédiger un finisseur selon une notation suivant celle rencontrée en C++ (c'est-à-dire `~` suivi du nom de la classe dont le finisseur doit être appelé lors de la collecte d'une instance).

Cependant, cette méthode est d'une utilité limitée puisque, dans le cas où des objets se réfèrent l'un et l'autre mutuellement, il n'est pas possible de prédire dans quel ordre ils seront supprimés. Il est donc risqué de manipuler quelque objet `.NET` que ce soit dans un destructeur C#.

Le destructeur de C# est transformé par le compilateur en un appel implicite à une méthode `protected` nommée `Finalize()` et de type `void`, comme en Java, mais C# ne permet pas de définir `Finalize()` manuellement dans un objet.

Il est aussi recommandé, lorsqu'un objet C# manipule des ressources externes au monde `.NET`, d'implémenter la méthode `Dispose()` de l'interface `IDisposable`.

Cela dit, nous ne pouvons aborder ce sujet pour l'instant, n'ayant pas encore couvert l'héritage, encore moins l'héritage d'interfaces (ce que nous ferons dans [POOv01]).

```
namespace z
{
    public class X
    {
        private int valeur_;
        public X(int val)
        {
            valeur_ = val;
        }
        public int GetValeur()
        {
            return valeur_;
        }
        ~X()
        {
            System.Console.WriteLine
                ("Décès d'un X, valeur {0}",
                 GetValeur());
        }
        public static void Main()
        {
            X x = new X(3);
        }
    }
}
```

La classe `Notes` proposée en C++ plus haut se décline en C# à peu près comme en Java. Les différences entre Java et C# sont, ici, superficielles.

On aurait pu insérer un destructeur (méthode `~Notes()`) dans l'exemple à droite mais il aurait été vide, le tableau de `float` étant ramassé automatiquement par le moteur de collecte d'ordures.

On remarquera au passage que la plupart des différences syntaxiques entre Java et C# tiennent de l'emploi des majuscules et des minuscules.

Sous `.NET`, les types primitifs sont des alias pour des types `struct`, créés sur la pile plutôt que sur le tas et manipulés directement plutôt que de manière indirecte.

Il se trouve que `float` est un tel type, donc que le tableau `notes_` contient véritablement des `float`. S'il s'agissait d'un tableau d'objets, alors il faudrait les instancier à la pièce, comme en Java.

```
namespace z
{
    public class Notes
    {
        public int NbNotes
        {
            get { return Notes.Length; }
        }
        public int[] Note
        {
            get; private set;
        }
        public Notes()
            : this(NB_NOTES_DÉFAUT)
        {
        }
        public Notes(int NbNotes)
        {
            Note = new float[NbNotes];
            for (int i = 0; i < NbNotes; i++)
                Note[i] = 0.0f;
        }
        public Notes(Notes n)
        {
            Note = new float[n.NbNotes];
            for (int i = 0; i < NbNotes; i++)
                Note[i] = n.Note[i];
        }
        private const int NB_NOTES_DÉFAUT = 10;
    }
}
```

En VB . NET, la situation ressemble à celle présentée pour C# à ceci près que la syntaxe du destructeur à la sauce C++ ou C# n'est pas disponible (ce qui se tient considérant que la syntaxe du constructeur n'y est pas disponible non plus) et au sens où il faut rédiger la méthode `Finalize()` telle que proposée à droite.

```
Namespace z
  Public Class X
    Private valeur_ As Integer
    Public Function GetValeur() As Integer
      GetValeur = valeur_
    End Function
    Public Sub New(ByVal val As Integer)
      valeur_ = val
    End Sub
    Protected Overrides Sub Finalize()
      System.Console.WriteLine("Décès d'un X, valeur {0}", _
        GetValeur())
    End Sub
    Public Shared Sub Test(ByVal x As X)
      System.Console.WriteLine(x.GetValeur())
    End Sub
    Public Shared Sub Main()
      Test(New X(3))
    End Sub
  End Class
End Namespace
```

Encore une fois, dans le détail, VB.NET diffère un peu, syntaxiquement, de la plupart des autres langages décrits ici.

Tout d'abord, notons la syntaxe des tableaux. L'attribut `notes_` et VB.NET est déclaré avec les parenthèses avant le nom, un peu comme les tableaux de Java et de C# ont les crochets avant le nom de l'attribut.

L'instanciation d'un tableau VB.NET se fait avec `New`, mais les bornes sont indiquées entre parenthèses plutôt qu'entre crochets et la borne supérieure est incluse (prudence!).

Notez que, depuis VB.NET, les indices des tableaux commencent toujours à zéro. Dans un tableau VB.NET contenant des éléments dont le type est primitif (comme `Single`), il est nécessaire de suivre l'appel à `New` par une séquence de valeurs initiales placées entre accolades.

Ici, `{}` signifie vide et les valeurs par défaut des données sont alors l'équivalent de `0` pour le type manipulé.

```

Namespace z
  Public Class Notes
    Public Sub New()
      Me.New(NB_NOTES_DÉFAUT)
    End Sub
    Public Sub New(ByVal nbNotes As Integer)
      notes_ = New Single(nbNotes - 1) {} ' vide
      For i As Integer = 0 To GetNbNotes() - 1
        SetNote(i, 0.0F)
      Next
    End Sub
    Public Sub New(ByVal n As Notes)
      notes_ = New Single(n.GetNbNotes() - 1) {} ' vide
      For i As Integer = 0 To GetNbNotes() - 1
        SetNote(i, n.GetNote(i))
      Next
    End Sub
    Public Function GetNbNotes()
      Return notes_.Length
    End Function
    Public Function GetNote(ByVal i As Integer) As Single
      Dim Résultat As Single = 0.0F
      If (i < GetNbNotes()) Then
        Résultat = notes_(i)
        Return Résultat
      End If
    End Function
    Public Sub SetNote(ByVal i As Integer, _
      ByVal valeur As Single)
      If (i < GetNbNotes()) Then
        notes_(i) = valeur
      End If
    End Sub
    Private Const NB_NOTES_DÉFAUT As Integer = 10
    Private notes_() As Single
  End Class
End Namespace

```


Exercices – Série 04

EX00 – Ajoutez à la classe `Notes` une méthode de classe `est_nb_notes_valide()` qui prend un entier constant en paramètre et retourne `VRAI` si l'entier est un nombre de notes valide (un entier strictement positif), ou `FAUX` sinon;

EX01 – Ajoutez à la classe `Notes` une méthode d'instance `est_indice_valide()` qui prend un entier constant en paramètre et retourne `VRAI` si l'entier est un index de note valide pour cette instance (un entier strictement positif mais inférieur au nombre de notes de cette instance), ou `FAUX` sinon;

EX02 – Repérez tous les endroits, dans les différentes méthodes de la classe `Notes`, où vous pourriez utiliser l'une ou l'autre des deux méthodes ci-dessus, et procédez.

EX03 – Reprenez la classe `Rectangle` proposée plus haut et rédigez une fonction `pivoter()` prenant en paramètre un `Rectangle` et retournant un `Rectangle` équivalent au `Rectangle` original s'il était pivoté de 90° (ou $\frac{\pi}{2}$ radians). Le `Rectangle` original ne doit pas être modifié. Montrez par un petit programme de test que votre programme fonctionne (entre autres, `pivoter(pivoter(r)).dessiner()` et `r.dessiner()` doivent dessiner des formes identiques à l'écran).

Quelques questions, pour réfléchir un peu.

Q00 – Est-il possible de tirer profit de `est_nb_notes_valide()` dans l'écriture de `est_indice_valide()`? Si oui, comment? Sinon, pourquoi?

Q01 – Expliquez ce qui fait en sorte que `est_nb_notes_valide()` *peut* être une méthode de classe alors que `est_indice_valide()` *doit* être une méthode d'instance.

Q02 – La méthode `SetNbNotes()` a un (très!) gros défaut : elle ne rapporte pas les erreurs aux sous-programmes qui l'utilisent, corrigeant silencieusement les paramètres incorrects. Les programmes qui corrigent silencieusement les problèmes sont des programmes où les problèmes perdurent. Comment pourrions-nous procéder pour détecter et rapporter correctement les erreurs dans cette méthode⁷⁴?

Q03 – Faites la même réflexion avec la méthode `SetNote()` : quels sont les problèmes qui peuvent survenir? Comment pourrait-on les détecter et les gérer?

Q04 – Examinez EX03 de cette série d'exercices (plus haut). À votre avis, `pivoter()` devrait-elle être une méthode d'instance, une méthode de classe ou une fonction globale? Quels sont les avantages et les inconvénients dans chaque cas?

⁷⁴ Nous donnerons une stratégie dans la section *Traitement d'exceptions*, plus loin. Une discussion du problème est aussi proposée dans *Réflexion 00.3 : des correctifs silencieux?*

Approche OO et code efficace

Pour des raisons surtout historiques, le modèle OO a parfois mauvaise presse en raison de critiques selon lesquelles les programmes en résultant seraient plus lents que ceux faits selon le modèle structuré.

C'est faux, mais une chose reste vraie, pour le code OO comme pour le code structuré : une attention portée aux détails de saine hygiène du code mène, en pratique, à du code plus performant.

Sans donner dans des techniques d'optimisation pointues, nous examinerons ici quelques petits détails qui permettent d'avoir des programmes OO à la fois propres, élégants et rapides, et nous nous permettrons par la suite de les utiliser dans nos exemples.

Encapsulation et vitesse

Le professeur vous suggère de définir plusieurs petits mutateurs et plusieurs petits accesseurs pour obtenir un contrôle serré sur l'accès aux attributs de vos objets. N'est-ce pas abusif? En fait, pour poser la question formellement : *est-ce que passer systématiquement par des méthodes pour accéder aux attributs d'un objet implique un ralentissement par rapport à un accès direct aux attributs eux-mêmes?*

La réponse est simple : oui... et non.

Réaliser un appel de fonction⁷⁵ entraîne un coût à l'exécution, qu'il faut ajouter à celui, plus brut, résultant de l'accès à l'attribut encapsulé. Il y a une mécanique de bas niveau associée à l'appel d'une fonction qui implique plusieurs opérations supplémentaires si on la compare à une simple lecture de variable, et le coût de cette mécanique est vérifiable empiriquement.

Tuons l'idée dans l'œuf : aujourd'hui, les langages OO, en pratique, sont souvent plus rapides (quand ils sont bien utilisés) que leur contrepartie structurée. Les langages populaires comme Java et C# permettent de rédiger des programmes efficaces pour les applications de tous les jours, et un programme C++ *bien écrit* ne s'exécutera pas plus lentement qu'un programme C *bien écrit* réalisant la même tâche (au contraire). Ces deux programmes *bien écrits* seront toutefois très différents l'un de l'autre.

Pourquoi un détail aussi technique que la vitesse?

Certains abordent la POO dans l'optique de diriger un jour des projets en technologie de l'information. Pour ceux-ci, clairement, la question se pose: pourquoi donc un(e) chef de projet se préoccuperait-il/ elle d'un détail comme la vitesse d'exécution d'un sous-programme?

Tout d'abord, il faut prendre ce qui précède comme une indication que ***bien développer ne signifie pas nécessairement produire des objets lents et inefficaces***. Il est tout à fait possible de développer selon une approche OO et de prendre des décisions de design qui mènent à une solution à la fois solide, bien encapsulée, et aussi rapide qu'une solution équivalente mais réalisée selon le modèle structuré. Dans la majorité des cas, contrairement aux préjugés à cet effet, *beau* ne signifie pas *lent*.

Prendre une décision de design éclairée, par contre, implique aussi avoir une idée de la forme que peut prendre une solution efficace, et du prix conceptuel et technique à payer pour le gain de sécurité comme pour le gain de vitesse. Un bon général sait choisir ses combats, dit-on souvent.

Ensuite, il faut savoir qu'***une application robuste mais lente mène souvent vers un flop commercial***. La compétition est forte, et si vous et votre équipe n'êtes pas à l'affût des lieux dans un programme où un gain d'efficacité peut être fait, vos concurrents, eux, le seront assurément.

Remarquez que les plus grands gains de rapidité possibles sont en général beaucoup plus redevables aux avancées algorithmiques qu'à des détails techniques. N'empêche : les améliorations algorithmiques relèvent de code encapsulé, de détails d'implémentation, alors que le choix de révéler une méthode pour en accélérer l'appel relève plus de choix de votre ressort. *Le prix payé en révélant la méthode est-il compensé par l'amélioration obtenue lors de l'utilisation de l'objet dans un contexte réel?*

⁷⁵ Et une méthode comme `GetLargeur()`, c'est effectivement une fonction.

Pour de petites⁷⁶ méthodes, comme par exemple des accesseurs et des mutateurs, un bon compilateur peut souvent faire ce qu'on appelle du *Function Inlining* (ou, dans notre cas, du *Method Inlining*). Ceci correspond, *pour le compilateur*, à remplacer l'appel de la méthode par la *définition* de cette méthode, faisant en sorte que la perte de performance soit nulle ou à peu près nulle (selon les compilateurs et les circonstances).

Une méthode sur laquelle a été appliquée la technique du *Method Inlining* est une méthode `inline`, un mot qui porte un sens technique précis pour certains programmeurs et certains langages de programmation.

Attention : ici, c'est le compilateur qui procède à une optimisation, pas l'informaticien(ne) qui brise l'encapsulation de sa classe. Cette technique ne peut être appliquée à toutes les méthodes du modèle objet⁷⁷, mais pourrait s'appliquer systématiquement dans le cas des méthodes vues jusqu'ici.

On peut aider le compilateur à utiliser cette technique en indiquant la définition de la méthode à même la déclaration de la classe.

Dans notre cas, si on voulait s'assurer que le compilateur fasse du *inlining* avec nos méthodes `GetLargeur()` et `SetLargeur(int)`, les fichiers `Rectangle.cpp` et `Rectangle.h` (qui sont ici volontairement abrégés et simplifiés) deviendraient :

Rectangle.h	Rectangle.cpp
<pre> #ifndef RECTANGLE_H #define RECTANGLE_H #include <exception> class Rectangle { int largeur_; public: int GetLargeur() const { return largeur_; } void SetLargeur(int lar) { // valider que lar > 0 (omis) largeur_ = lar; } }; #endif </pre>	<pre> #include "Rectangle.h" // fichier vide, pourquoi pas? </pre>

⁷⁶ Il y a d'autres contraintes que celle de la taille des méthodes pour choisir de réaliser ou non cette optimisation. Par exemple, on évitera les méthodes récursives pour ce type de manœuvre car cela impliquerait une croissance à l'infini de la taille du code généré.

⁷⁷ Les méthodes d'une importante catégorie, celle des méthodes virtuelles, ne peuvent recevoir ce traitement pour des raisons techniques, ce qui pénalise beaucoup les programmes Java pour lesquels les méthodes sont virtuelles sauf indication contraire. Nous y reviendrons dans [POOv01] lorsque nous discuterons du polymorphisme.

Le gain de vitesse obtenu par cette technique peut être très significatif, mais le prix à payer est que l'implémentation devient visible aux *humain(e)s* examinant la déclaration de la classe. On choisira donc de s'en servir lorsque le code est banal (comme dans le cas ci-dessus), mais on l'évitera lorsque des secrets d'entreprise sont en jeu.

L'emploi de méthodes `inline` améliore le temps d'exécution mais tend à ralentir le temps de compilation. Sur de gros projets, il faut en être conscient(e). Le compilateur doit travailler plus fort et procéder à une série de remplacements de d'appels de sous-programmes par les sous-programmes appelés, et ce de manière récursive. Rien n'est gratuit.

Si les méthodes sur lesquelles le *Method Inlining* est appliqué sont plus complexes que deux ou trois instructions, le programme résultant grossira visiblement aussi, du fait que chaque appel à une telle méthode aura été remplacé par le corps de la méthode. *Si la technique est appliquée à l'excès, alors la taille du programme nuira au moteur d'optimisation du compilateur et le programme ralentira plutôt que d'accélérer.*

Il faut donc choisir les méthodes pour lesquelles on appliquera cette technique, et se limiter à celles qui sont vraiment critiques, compactes et utilisées massivement (les accesseurs et les mutateurs les plus directs, par exemple, de même que certains constructeurs).

Où poser son regard?

Bien choisir ses combats signifie aussi savoir investir sagement ses efforts. Certaines améliorations techniques apparentes n'apportent à peu près aucun gain d'ensemble à un système informatique.

Par exemple, appliquer du *Method Inlining* à une méthode qui n'est appelée que quelques fois lors du lancement d'un programme n'apporte essentiellement rien de positif à une application. Par contre, faire de même avec un accesseur inoffensif mais appelé à profusion pour des milliers d'objets aura un impact autrement plus visible.

De la même manière, lorsqu'un sous-programme utilise un objet de manière strictement sans danger de le modifier (n'utilisant que ses méthodes `const`), il est très avantageux de le passer par référence et `const` plutôt que par valeur, puisque ceci évite un appel au constructeur de copie de cet objet (et au destructeur de la copie, à la fin du sous-programme) à chaque utilisation du sous-programme en question. Évidemment, dans les langages où il n'est pas possible d'accéder directement à un objet (p. ex. : Java, C#, VB.NET), cette optimisation est implicite.

Le truc, donc, est d'identifier les méthodes dont il vaut vraiment la peine de chercher à améliorer le rendement. Insérer des compteurs pour vérifier le nombre d'appels à chacune est une option; utiliser des outils de profilage (*Profiling*) en est une autre. La décision en est une d'ensemble; elle est donc de votre ressort.

Déclarer avant d'utiliser?

S'il est vrai qu'il faut, en C++, déclarer toute chose avant de l'utiliser, cela signifie-t-il que, pour faire du *Method Inlining*, il devient nécessaire de déclarer les attributs ou les méthodes avant de les définir, donc qu'on ne peut plus déclarer, dans une classe, les membres dans l'ordre qui nous sied le mieux?

Non. La raison est subtile, par contre :

- le *Method Inlining* permet de remplacer les appels de méthodes par la définition des méthodes appelées. Ceci modifie, à chaque appel, *le code de l'appelant*;
- constatant ceci, on remarque que le code des méthodes définies à même la classe n'est pas compilé au moment de leur déclaration (et de leur définition, puisqu'il s'agit alors du même moment et du même lieu pour ces deux choses). Au contraire, le code est compilé à même chaque sous-programme appelant;
- lorsque l'appel d'une méthode est fait par un sous-programme, il se trouve que la classe entière d'où provient la définition de la méthode appelée a alors nécessairement déjà été déclarée en entier.

Une nuance s'impose ici : en C++, les types doivent être connus au moment où ils sont utilisés, dans une lecture du code de haut en bas. Pour cette raison, quand une classe contient d'autres classes ou définit des alias avec `typedef` (une technique plus avancée, couverte dans [POOv02]), les noms des types doivent être déclarés avant que les types en question ne soient utilisés.

Conclusion : nous demeurons libres, dans la déclaration d'une classe utilisant le *Method Inlining*, de déclarer ses membres dans l'ordre qui nous convient le mieux.

Constantes d'instance

Il peut arriver qu'on cherche à représenter un attribut d'instance tel que :

- la valeur de l'attribut ne puisse pas changer une fois celui-ci initialisé, donc qu'il s'agisse d'un attribut d'instance constant; et
- que chaque instance de la classe possède sa propre version de cet attribut, en particulier si la valeur de l'attribut peut varier d'une instance à l'autre.

On ne peut alors pas avoir recours à une constante de classe (`static const`), mais on ne peut pas non plus simplement insérer le mot clé `const` devant le type d'un attribut, puisque le constructeur ne pourrait pas l'initialiser, l'attribut étant constant!

Voilà donc un concept utile, mais qu'une notation naïve ne permet pas d'exprimer. Avec la préconstruction, par contre, il devient possible de représenter de manière élégante et utilisable l'idée de constante d'instance.

En effet, ici, lorsque l'accolade ouvrante du constructeur de l'objet est rencontrée, ses attributs sont déjà construits, et les constantes ont déjà une valeur immuable.

Tel que vu plus haut, il faut, pour avoir le droit de représenter une constante d'instance, intervenir avant même l'accolade ouvrante du constructeur.

La syntaxe demeure la même : insérer le symbole `:` suivi d'une série d'appels de constructeurs (par défaut, paramétriques, de copie, peu importe), un par attribut à initialiser avant la construction. Dans le cas où plusieurs attributs doivent être préconstruits, les invocations de constructeurs doivent être séparées par des virgules.

```
// illustration très simple
class EntierConstant {
    const int valeur_;
public:
    EntierConstant(int valeur) {
        // illégal!
        valeur_ = valeur;
    }
    int GetValeur() const {
        return valeur_;
    }
    bool EstPositif() const {
        return GetValeur() >= 0;
    }
    bool EstPair() const {
        return GetValeur()%2 == 0;
    }
    // etc.
};
```

```
// illustration très simple
class EntierConstant {
    const int valeur_;
public:
    EntierConstant(int valeur)
        : valeur_{valeur}
    {
    }
    // le reste ne change pas!
};
```

Écriture plus professionnelle

En pratique, une classe comme `EntierConstant` se décrirait de manière plus concise, un peu comme celle proposée à droite. Remarquez les éléments suivants :

- l'absence de mutateurs fait en sorte qu'une instance de cette classe n'offre que des accesseurs. Le préfixe `Get` devient redondant, et plusieurs l'omettront tout simplement;
- par conséquent, si `e` est un `EntierConstant`, alors écrire `e.valeur()` signifie consulter la valeur de `e`, alors que `e.pair()` est vrai si `e` est pair, justement;
- certains préféreront écrire `est_pair()` ou `EstPair()`, mais la différence est stylistique;
- le nom `valeur` attribué au paramètre du constructeur paramétrique est un nom local, qui cache le nom de la méthode d'instance `valeur()` pour la durée de l'exécution du constructeur. On pourrait toutefois accéder à cette méthode dans le constructeur en écrivant explicitement `this->valeur()`.

```
class EntierConstant {
    const int valeur_;
public:
    EntierConstant(int valeur)
        : valeur_{valeur}
    {
    }
    int valeur() const {
        return valeur_;
    }
    bool positif() const {
        return valeur() >= 0;
    }
    bool pair() const {
        return valeur() % 2 == 0;
    }
    // etc.
};
```

Question : étant donné la classe `EntierConstant` ci-dessus, le programme suivant est-il légal? Expliquez votre réponse.

```
#include "EntierConstant.h" // classe EntierConstant
#include <iostream>
int main() {
    using namespace std;
    EntierConstant e = 3;
    cout << e.valeur() << endl;
    e = 4;
    cout << e.valeur() << endl;
}
```

Instanciation tardive

Le langage C++, comme la plupart des langages OO, permet de déclarer des constantes et des variables à peu près n'importe où dans un programme. La portée d'une donnée en C++, que cette donnée soit une constante ou une variable, demeure délimitée inclusivement par le lieu de sa déclaration et la fin du bloc dans lequel elle a été déclarée. Pour une donnée globale, la fin du bloc est, implicitement, l'unité de traduction en cours de compilation⁷⁸.

La règle du langage C était de déclarer les variables au début d'un bloc, un bloc étant délimité par une paire d'accolades balancées, ouvrante et fermante dans l'ordre, et les blocs pouvant être inclus les uns dans les autres de façon récursive. Un bloc délimitant entre autres la définition d'un sous-programme, cette définition était à proprement dit suffisante. Les règles de la plupart des langages similaires ressemblaient à celle de C, étant parfois un peu plus contraignantes (forçant par exemple la déclaration des variables et des constantes locales au début d'un sous-programme plutôt qu'au début d'un bloc).

L'approche OO permet (et encourage) une stratégie différente soit déclarer chaque variable et chaque constante *le plus tard possible*, ou plus précisément *le plus près possible (dans la mesure où l'endroit est raisonnable) de son point d'utilisation*, ou du moins pas avant que le programme ait la certitude que la donnée en question sera requise, peut-être même pas avant de savoir avec quels paramètres elle sera construite, menant ainsi à une *instanciation tardive*.

À titre d'exemple, examinons le sophistiqué (!) code de ce `PetitJeuBanal()`.

Les deux exemples ont le même nombre de lignes de code et réalisent le même travail, mais celui à droite, qui applique l'instanciation tardive, est meilleur sous-programme que celui de gauche.

En effet, celui de gauche construit (et détruit) l'objet `mot` peu importe la réponse de l'utilisateur, alors que celui de droite ne réalisera ces opérations (plus complexes qu'il n'y paraît) que si cela s'avère véritablement nécessaire.

Façon traditionnelle	Instanciation tardive
<pre>// ... void PetitJeuBanal() { string reponse, mot; cout << "Jouer?"; if (cin >> reponse && reponse[0]!='0') { cout << "Un mot: "; if (cin >> mot && mot.size() > 10) cout << "Quel mot!"; else cout << "Bof"; } }</pre>	<pre>// ... void PetitJeuBanal() { string reponse; cout << "Jouer?"; if (cin >> reponse && reponse[0]!='0') { cout << "Un mot: "; string mot; if (cin >> mot && mot.size() > 10) cout << "Quel mot!"; else cout << "Bof"; } }</pre>

La version de droite est donc plus rapide et plus légère, en moyenne, que celle de gauche. Dans le pire cas, leur complexité est identique.

Comme tous les outils, la qualité des résultats de cette approche dépendra de la manière dont le programme s'en servira.

⁷⁸ L'unité de traduction dans laquelle apparaît une donnée globale, en C++, est le fichier source dans lequel cette donnée apparaît, chaque fichier source étant compilé individuellement.

On peut bien sûr faire encore mieux. Par exemple, par l'ajout de fonctions :

```
bool jouer(const string &invite) {
    string reponse;
    cout << invite;
    return cin >> reponse && reponse[0]!='O';
}

bool est_mot_chouette(const string &mot) {
    return mot.size() > 10;
}

void PetitJeuBanal() {
    if (jouer("Jouer? ")) {
        cout << "Un mot: ";
        string mot;
        if (cin >> mot && est_mot_chouette(mot))
            cout << "Quel mot!";
        else
            cout << "Bof";
    }
}
```

... ou encore, depuis C++ 17, en assurant une portée plus restreinte encore aux variables :

```
void PetitJeuBanal() {
    if (jouer("Jouer? ")) {
        cout << "Un mot: ";
        if (string mot; cin >> mot && est_mot_chouette(mot))
            cout << "Quel mot!";
        else
            cout << "Bof";
    }
}
```

Le principe de localité [hdLoc] nous invite à réduire au minimum la portée des variables. C'est souvent la meilleure chose à faire.

Un langage OO apporte, en comparaison avec un langage structuré, la possibilité de déclarer des classes et de les instancier, donc de solliciter les mécaniques de construction et de destruction des instances. Une déclaration de variable ou de constante n'est plus un événement banal, qu'on peut se permettre de positionner naïvement où bon nous semble, mais bien *une combinaison de deux appels de sous-programmes* (un pour la construction, un pour la destruction).

Avec des types primitifs, les deux sous-programmes ci-dessous⁷⁹ sont essentiellement équivalents en vitesse et en taille (en pratique, il est possible que celui de droite soit légèrement plus rapide que celui de gauche dû à de meilleures garanties offertes par le programme lors de la compilation). La différence entre les deux est esthétique :

<pre>float serie_harmonique(int n) { float somme {}, terme; for (int i = 1; i <= n; ++i) { terme = 1.0f / i; somme += terme; } return somme; }</pre>	<pre>float serie_harmonique(int n) { float somme {}; for (int i = 1; i <= n; ++i) { const float TERME = 1.0f / i; somme += TERME; } return somme; }</pre>
---	--

Avec des objets, les deux sous-programmes ci-dessous *peuvent* être très différents en vitesse et en taille, selon les implémentations de la classe `Reel`. Celui à gauche ne construit (et ne détruit) `terme` qu'une seule fois; celui à droite accomplit ces deux opérations pas moins de `n` fois (présument `n` positif).

<pre>Reel serie_harmonique(int n) { Reel somme{ 0.0f }, terme; for (int i = 1; i <= n; ++i) { terme.SetValeur(1.0f / i); somme += terme; } return somme; }</pre>	<pre>Reel serie_harmonique(int n) { Reel somme{ 0.0f }; for(int i = 1; i <= n; ++i) { const Reel TERME{1.0f / i}; somme += TERME; } return somme; }</pre>
---	--

Cet exemple laisse sous-entendre que l'instanciation tardive est coûteuse en temps; et dans le cas présenté ci-dessus, il est possible que ce soit effectivement le cas (en fonction de la qualité du constructeur, du destructeur, du mutateur, *etc.*). Il y a donc des occasions pour lesquelles une instanciation tardive est une stratégie qui nuit à l'efficacité d'un programme.

⁷⁹ ...qui calculent les n premiers termes d'une série harmonique : $\sum_{i=1}^n \frac{1}{i}$

Examinons maintenant ce qui suit :

<pre>int etrange_calcul(int n) { int resultat; if (n >= 0) { X x{ n }; x.OperationMystere(); resultat = x.GetValeur(); } else { Y y{ n }; y.OperationSurprise(); resultat = y.GetValeur(); } return resultat; }</pre>	<pre>int etrange_calcul(int n) { int resultat; X x; Y y; if (n >= 0) { x = X{ n }; x.OperationMystere(); resultat = x.GetValeur(); } else { y = Y{ n }; y.OperationSurprise(); resultat = y.GetValeur(); } return resultat; }</pre>
--	--

Ici, l'exemple à gauche applique l'instanciation tardive et est plus efficace que l'exemple à droite. En effet, notre exemple de gauche créera puis détruira une instance de `X` seulement si `n` est positif; de même, il créera puis détruira une instance de `Y` seulement dans le cas contraire.

De son côté, notre exemple de droite créera toujours (avec construction par défaut) et détruira toujours une instance de `X` de même qu'une instance de `Y`. De plus, il créera une instance temporaire, à l'aide d'une invocation de constructeur paramétrique, de l'une ou l'autre de ces classes selon la valeur de `n`⁸⁰. Évidemment, cette temporaire devra être détruite au plus tard à la fin de sa portée.

On le voit à l'aide de ces divers exemples : l'instanciation tardive a sa raison d'être, mais n'est pas une technique à appliquer aveuglément. Bien placée, utilisée avec discernement, il s'agit d'un outil permettant de rendre aussi efficace que possible un programme OO.

⁸⁰ Ceci aurait peut-être pu être évité par un mutateur bien placé.

Dans d'autres langages

Les constantes d'instance au sens de C++ n'existent pas vraiment en C# ou VB.NET. C# permet toutefois de simuler⁸¹ les constantes d'instance par le mot clé `readonly`, qui empêche à la compilation toute opération modifiant l'attribut ainsi qualifié, sauf dans le constructeur de la classe à laquelle il appartient, ou par des propriétés qui n'offriraient qu'un accès de type `get`. VB.NET fait de même mais utilise l'écriture `ReadOnly`.

En Java, il est possible de déclarer un attribut constant `static` ou non, et d'initialiser les deux à même la déclaration.

Une constante d'instance (`final` mais pas `static`) peut aussi ne pas être initialisée à la déclaration et recevoir sa valeur initiale dans le constructeur de l'instance à laquelle elle appartient. Une fois la première initialisation d'une constante d'instance complétée (dans le constructeur), celle-ci ne peut plus être modifiée.

Évidemment, les règles usuelles selon lesquelles une méthode de classe n'a directement accès qu'aux membres de classe s'appliquent ici aussi.

```
public class Z {
    private static final int M = 10;
    private final int N0 = 10;
    private final int N1;
    private int val_;
    public Z() {
        val_ = 3; // légal
        N1 = 4;   // légal
        N1 = 5;   // illégal: déjà initialisée
    }
    public static void main(String [] args) {
        System.out.println(M); // légal
        System.out.println(N0); // illégal
    }
}
```

Un compilateur Java cherchera à réaliser diverses optimisations telles que du *Method Inlining* mais ne le pourra pas toujours⁸². Cela dit, la machine virtuelle Java (la *Java Virtual Machine*, ou JVM) réalise certaines optimisations du code lors de l'exécution qu'un compilateur à vocation strictement statique comme C++ ne pourrait pas réaliser.

La machine virtuelle .NET (le *Common Language Runtime*, ou CLR) compile quant à elle le code .NET en code indigène lors de l'exécution du programme (typiquement, le premier appel d'une méthode provoque une compilation de cette méthode) et conserve le code ainsi compilé dans une antémémoire logique. Ceci ralentit la première invocation d'une méthode mais permet une performance s'apparentant à celle du code indigène lors des appels subséquents, du moins dans la mesure où le code ainsi compilé demeure dans l'antémémoire.

Les langages VB.NET et C# sont faits de manière telle qu'ils permettent au compilateur certaines optimisations que Java ne permet pas⁸³. Le prix à payer pour ce gain de savoir est une paire de langages nettement plus verbeux; nous pourrions mieux expliquer ces nuances quand nous aurons examiné le polymorphisme [POOv01].

⁸¹ La protection est déficiente, cela dit, alors utilisez `readonly` avec discrimination [ReadOnly].

⁸² Ceci tient au fait qu'en Java, hormis les constructeurs, toutes les méthodes d'instance sont polymorphiques (hormis les constructeurs et les méthodes qualifiées `final`), ce qui n'est pas un problème en soi mais empêche le compilateur de déduire certaines optimisations. Nous reviendrons sur ce sujet dans [POOv01].

⁸³ En particulier, ces langages permettent de mélanger des méthodes polymorphiques et d'autres qui ne le sont pas, ce qui augmente le savoir que possède le compilateur sur les éventuelles conditions d'exécution du code généré.

L'instanciation tardive, bien appliquée, est une stratégie gagnante dans tous les langages OO, incluant Java, C# et VB.NET. Considérant que le moment de construction d'un objet est une méthode à part entière et peut être d'une complexité arbitrairement grande, et considérant qu'il en est de même pour la destruction ou de la finalisation dans les langages offrant un support sérieux à ce volet de l'encapsulation, limiter l'instanciation au nécessaire et la réaliser au moment opportun tend à être sage.

Cette stratégie comporte des incarnations locales dans divers langages. Un exemple parmi plusieurs, décrit ici en Java mais ayant une contrepartie en C# et en VB.NET :

- en Java, le type `String` a droit à un support spécial du compilateur, mais reste un type immuable forçant les programmeurs à utiliser massivement, même en période de pointe, la mécanique de construction (et, Java étant Java, l'allocation dynamique de mémoire);
- une répétitive concaténant plusieurs fois successivement un suffixe à une instance de `String` générera de multiples instances temporaires de ce type;
- prenant ce détail en considération, travailler à partir d'une instance de `StringBuffer`, classe modifiable, puis instancier une `String` seulement une fois celle-ci véritablement prête à l'être, constitue une optimisation visible et non négligeable.

Ne vous méprenez pas : dans un langage OO, mal comprendre un système de types et mal saisir la mécanique de construction/ de destruction et ses impacts sur un programme sont d'excellentes manières de produire des programmes inefficaces et inutilisables. Prenez le temps de comprendre vos outils et d'intégrer les philosophies qu'ils sous-tendent.

Vie d'un objet et tableaux

La déclaration d'un tableau d'instances d'une classe donnée a pour effet d'appeler le constructeur par défaut pour chacune des instances qu'on retrouve au sein de ce tableau. De même, la destruction du tableau entraîne aussi l'appel du destructeur de chacune de ces instances.

Le petit programme ci-dessous illustre le phénomène :

```
#include "Rectangle.h"
int main() {
    // appel implicite du constructeur par défaut de Rectangle
    Rectangle r0,
    // appels implicites du constructeur paramétrique de Rectangle
        r1(5, 5),
        r2{ 6, 6 },
    // appel implicite du constructeur par défaut de Rectangle
    // pour chacune des sept instances de Rectangle du tableau
        r3[7];
    // ... on se sert de tout ce beau monde ...
} // destruction implicite de r0, r1 et de chaque Rectangle de r2
```

Vie d'un objet et allocation dynamique de mémoire

On peut (et, souvent, on doit) allouer dynamiquement des objets. L'allocation et la libération dynamique de mémoire se fait, pour les objets comme pour les types primitifs, à l'aide des opérateurs `new` et `delete`.

Notez que par défaut, un pointeur est nul (égal à `nullptr`). C'est ce qui explique l'initialisation de `r1`, `r2`, `r3` et `r4` à droite.

Il y a une différence dans le traitement au moment de la construction et de la destruction des objets. En effet, utiliser **new** pour allouer dynamiquement un objet appelle explicitement son constructeur, choisi à la compilation en fonction des paramètres. De même, appliquer **delete** pour libérer un objet appelle explicitement son destructeur.

Ainsi, le programme visible à droite présente un résumé de plusieurs variantes de construction et de destruction, implicite ou explicite selon les circonstances.

Subtilités syntaxiques : Java et C# obligent le recours à des parenthèses vides lors de la construction dynamique d'objets à l'aide de leur constructeur par défaut, mais le constructeur par défaut ne demande pas de parenthèses vides en C++.

Notez aussi que C++ demande de préfixer le nom d'un pointeur par un astérisque, pour distinguer objets accédés indirectement des objets accédés directement, alors que Java et C# ne le demandent pas pour les références puisque tous les objets sont accédés directement.

```
#include "Rectangle.h"
int main() {
    Rectangle *r1 = nullptr,
              *r2 = nullptr,
              *r3 = nullptr,
              *r4 = nullptr;

    // appel explicite du constructeur par défaut.
    // r1 pointe vers le nouveau Rectangle
    r1 = new Rectangle;

    // appel explicite du constructeur paramétrique
    r2 = new Rectangle(5, 7);

    // appel explicite du constructeur de copie
    r3 = new Rectangle(*r2);

    // appel explicite du constructeur par défaut
    // pour chacun des 10 instances construites.
    // r4 pointe vers le tableau résultant
    r4 = new Rectangle[10];
    // ... on se sert de tout ce beau monde ...
    // destruction explicite de r4. Le destructeur est
    // appelé pour chaque Rectangle du tableau
    delete[] r4;

    // destruction explicite de r1, r2, r3
    delete r3;
    delete r2;
    delete r1;
}
```

```
// Java ou C#
Rectangle r = new Rectangle();
// C++
Rectangle *r0 = new Rectangle;
auto r1 = new Rectangle;
// parenthèses et accolades
// permises, mais superflues
auto r2 = new Rectangle();
auto r3 = new Rectangle{};
```

Dans d'autres langages

Le cas des tableaux en Java, C# et VB.NET a été brièvement abordé dans la section *Dans d'autres langages* suivant la section *Destructeur*. Nous ferons donc ici un bref résumé explicatif et descriptif.

En Java, un tableau est un objet. Conséquemment, la création d'un tableau se fait à l'aide de l'opérateur `new` et le tableau est assujéti à une collecte automatique d'ordures.

La déclaration d'un tableau place les crochets entre le type et le nom de la variable. Chaque paire de crochets signifie une dimension. Dans l'exemple à droite, `tab_` est un tableau 1D de `int`, alors que `tabX_` est un tableau 2D (donc un tableau de tableaux) de `X`. Remarquez la notation de la construction de `tabX_` dans le constructeur de `Z` à droite.

Un tableau étant un objet, il a des attributs, dont sa longueur (une constante d'instance publique nommée `length`).

Les tableaux 2D (ou plus) n'ont pas à être rectangulaires. Dans l'exemple à droite, il serait légal que `tabX_[i].length` et `tabX_[j].length` soient différents

```
// X est une classe immuable
class X {
    private final int val_;
    public int getVal() { return val_; }
    public X(int Val) { val_ = Val; }
}

public class Z {
    private static final int NB_ENTIERS = 10;
    private static final int NB_LIGNES_X = 5;
    private static final int NB_COLONNES_X = 3;
    private int [] tab_;
    private X [][] tabX_;
    public Z() {
        tab_ = new int[NB_ENTIERS];
        for (int i = 0; i < tab_.length; i++)
            tab_[i] = i + 1;
        tabX_ = new X[NB_LIGNES_X] [];
        for (int i = 0; i < tabX_.length; i++) {
            tabX_[i] = new X [NB_COLONNES_X];
            for (int j = 0; j < tabX_[i].length; j++)
                tabX_[i][j] = new X(i * NB_COLONNES_X + j);
        }
    }
    public void afficher() {
        for (int i = 0; i < tabX_.length; i++) {
            for (int j = 0; j < tabX_[i].length; j++)
                System.out.print(tabX_[i][j].getVal () + "\t");
            System.out.println();
        }
    }
    public static void main(String [] args) {
        Z z = new Z();
        z.afficher();
    }
}
```

Un programme Java ne manipule pas d'objets mais bien des références sur des objets. Pour cette raison, les éléments d'un tableau contenant des objets contiennent des références qu'il faut prendre soin d'instancier individuellement.

Les tableaux Java sont immuables une fois construits (ce serait vrai même si leurs éléments ne l'étaient pas). On ne peut changer la dimension d'un tableau; si l'espace vient à manquer, il faut créer un nouveau tableau (plus grand) et y copier les éléments du tableau original.

En C#, la situation quant aux tableaux est identique à celle présentée pour Java, aux majuscules et aux minuscules près.

L'exemple à droite en fait d'ailleurs clairement la démonstration. Prenez note que la disposition des accolades a été adaptée par souci d'économie d'espace.

Les tableaux C# sont eux aussi immuables (bien qu'il n'en soit pas de même pour leurs éléments), et ne peuvent donc pas être redimensionnés en tant que tel.

En Java comme en C# (comme en C++ d'ailleurs), de nombreuses stratégies existent pour manipuler (copier, traverser, compacter, *etc.*) les tableaux autrement qu'à l'aide des accès à travers des indices placés entre crochets. Je vous invite à explorer par vous-mêmes pour en savoir plus à ce sujet.

```
namespace z
{
    // classe immuable
    class X
    {
        private int val_;
        public int GetVal() { return val_; }
        public X(int Val) { val_ = Val; }
    }
    public class Z
    {
        private const int NB_ENTIERS = 10;
        private const int NB_LIGNES_X = 5;
        private const int NB_COLONNES_X = 3;
        private int[] tab_;
        private X[][] tabX_;
        public Z()
        {
            tab_ = new int[NB_ENTIERS];
            for (int i = 0; i < tab_.Length; i++)
                tab_[i] = i + 1;
            tabX_ = new X[NB_LIGNES_X][] ;
            for (int i = 0; i < tabX_.Length; i++) {
                tabX_[i] = new X[NB_COLONNES_X];
                for (int j = 0; j < tabX_[i].Length; j++)
                    tabX_[i][j] = new X(i * NB_COLONNES_X + j);
            }
        }
        public void Afficher() {
            for (int i = 0; i < tabX_.Length; i++) {
                for (int j = 0; j < tabX_[i].Length; j++)
                    System.Console.Write
                        (tabX_[i][j].GetVal() + "\t");
                System.Console.WriteLine();
            }
        }
        public static void Main(string [] args) {
            Z z = new Z();
            z.Afficher();
        }
    }
}
```

En VB .NET, la situation est semblable à celle vécue en C# mais :

- la notation pour les tableaux repose sur des parenthèses plutôt que sur des crochets;
- l'opérateur `new` du tableau prend en paramètre la borne supérieure (inclusive) du tableau plutôt que le nombre de ses éléments (prudence!);
- les répétitives `For` de VB .NET considèrent la borne supérieure comme incluse (prudence là aussi!);
- l'utilisation des parenthèses comme notation d'indice font que l'acte d'instancier un tableau ne peut à lui seul être distingué par le compilateur VB .NET de l'acte d'instancier un seul élément. Ceci explique l'obligation d'insérer en suffixe une liste (potentiellement vide) de valeurs initiales, placées entre accolades.

Remarquez dans la méthode d'affichage que VB .NET ne permet pas de concaténer deux chaînes de caractères à l'aide de l'opérateur `+` et que le symbole `'\t'` n'y est pas reconnu comme une tabulation (il faut utiliser `vbTab` à la place).

Héritage de versions antérieures de la lignée VB, le langage VB .NET permet de redimensionner un tableau avec la commande `Redim`.

```

Namespace z
    Public Class X ' immutable
        Private val_ As Integer
        Public Function GetVal() As Integer
            Return val_
        End Function
        Public Sub New(ByVal Val As Integer)
            val_ = Val
        End Sub
    End Class

    Public Class Z
        Private Const NB_ENTIERS As Integer = 10
        Private Const NB_LIGNES_X As Integer = 5
        Private Const NB_COLONNES_X As Integer = 3
        Private tab_() As Integer
        Private tabX_()() As X
        Public Sub New()
            tab_ = New Integer(NB_ENTIERS - 1) {}
            For i As Integer = 0 To tab_.Length - 1
                tab_(i) = i + 1
            Next
            tabX_ = New X(NB_LIGNES_X - 1)() {}
            For i As Integer = 0 To tabX_.Length - 1
                tabX_(i) = New X(NB_COLONNES_X - 1) {}
                For j As Integer = 0 To tabX_(i).Length - 1
                    tabX_(i)(j) = New X(i * NB_COLONNES_X + j)
                Next
            Next
        End Sub

        Public Sub Afficher()
            For i As Integer = 0 To tabX_.Length - 1
                For j As Integer = 0 To tabX_(i).Length - 1
                    System.Console.Write _
                        ("{0}{1}", tabX_(i)(j).GetVal(), vbTab)
                Next
                System.Console.WriteLine()
            Next
        End Sub

        Public Shared Sub Main()
            Dim z As New Z
            z.Afficher()
        End Sub
    End Class
End Namespace

```

Exercices – Série 05

Rédigez la classe `Triangle` qui aura au moins :

- une hauteur (impair, inclusivement entre 3 et 19);
- un constructeur par défaut, un constructeur paramétrique et un constructeur par copie;
- des accesseurs et des mutateurs convenables;
- une méthode pour en connaître la surface;
- une méthode permettant de le dessiner.

À titre d'exemple :

un triangle de hauteur 5	un triangle de hauteur 9
<pre> * ** *** **** ***** </pre>	<pre> * ** *** **** ***** ***** ***** ***** ***** ***** ***** ***** </pre>

Testez votre classe à l'aide du programme ci-dessous, qui devrait afficher trois triangles de taille 3, 5 et 9 respectivement :

```

#include "Triangle.h"
int main() {
    Triangle t1,
              t2{5};
    Triangle t3{t2};
    t3.SetHauteur(t3.GetHauteur() + 4);
    t1.dessiner();
    t2.dessiner();
    t3.dessiner();
}

```

Objets et opérateurs

Rôle de cette section

Le modèle OO vise, entre autres choses, à une **abstraction complète de l'idée de type**, au sens où *tous les types, qu'ils soient primitifs ou non, s'y voient offrir le même niveau de support*. Ceci inclut, entre autres choses, la possibilité d'y définir les **opérateurs** sur nos propres types; de contrôler, au besoin, les règles de conversion explicites de types; et de réaliser des entrées/ sorties sur les types primitifs et sur les objets à l'aide des mêmes mécanismes.

⇒ En langage C++, un **opérateur** est un cas particulier de sous-programme, sous forme de fonction globale ou de méthode (selon le cas).

Pourquoi réfléchir à une question un peu technique et qui ne soit applicable que dans certaines implantations du modèle OO? Pour plusieurs raisons. La première est que la surcharge d'opérateurs permet, lorsque bien appliquée, une **utilisation plus naturelle** des objets pour lesquels elle a été définie.

C'est évident pour les objets représentant des entités mathématiques communes. Par exemple, un type numérique tel que `Complexe` se manipule plus intuitivement à l'aide des opérateurs usuels comme `*` qu'à l'aide d'un sous-programme tel qu'une fonction `multiplier(Complexe, Complexe)`; on peut en dire autant d'un type `Matrice`.

C'est aussi apparent pour toute famille d'objets qu'on peut comparer entre eux pour les mettre en ordre: les opérateurs relationnels `<`, `<=`, `>` et `>=` se prêtent vraiment mieux à ces comparaisons que des fonctions du genre `EstPlusPetitQue()`.

Thème OO ou non?

Ce ne sont pas tous les langages de programmation, OO ou non, qui poussent l'abstraction des types si loin qu'ils permettent de surcharger les opérateurs, donc de définir l'action d'un opérateur sur des types du choix des développeurs.

Le choix d'une syntaxe pour cette fonctionnalité est délicat :

- le concepteur de C++ a choisi de représenter un opérateur comme un cas particulier de fonction globale ou de méthode, supportant les deux avec des syntaxes légèrement différentes, et permet de les surcharger *presque* tous;
- les concepteurs de Java ont choisi de ne pas permettre la surcharge d'opérateurs du tout, pour des raisons philosophiques⁸⁴;
- les concepteurs de C# ne permettent la surcharge que de certains opérateurs seulement, et à travers une philosophie bien différente de celle appliquée en C++;
- ceux de VB.NET ont choisi de ne pas permettre la surcharge d'opérateurs du tout.

Pouvoir redéfinir les opérateurs constitue un aboutissement logique à la démarche d'abstraction des types qu'est, en grande partie, l'approche OO. En effet, l'approche objet vise, à la limite, un support équivalent pour tous les types, qu'il s'agisse de types primitifs ou non. C'est dans cette optique que s'inscrit la mécanique de surcharge des opérateurs en C++.

⁸⁴ Tiré de [JavaNoOpOv] : « **No More Operator Overloading**. *There are no means provided by which programmers can overload the standard arithmetic operators. Once again, the effects of operator overloading can be just as easily achieved by declaring a class, appropriate instance variables, and appropriate methods to manipulate those variables. Eliminating operator overloading leads to great simplification of code* ». Cela dit, il y a beaucoup de pression pour changer ceci chez Sun, et il semble probable que Java introduise sous peu (enfin) la possibilité de définir des opérateurs sur des types arbitraires. Le penchant pour éviter la question des opérateurs est lié de près au fait que Java ne permette que des manipulations indirectes des objets : une expression comme `c = a + b`; se pense plus aisément si `a`, `b` et `c` sont les objets en soi plutôt que des indirections vers des objets.

La possibilité d'affecter un objet à un autre à l'aide de l'opérateur `=` n'est pas non plus négligeable, du moins pour un langage qui permet d'accéder directement aux objets. Rares sont celles et ceux qui n'ont pas le réflexe d'utiliser `a = b`; pour affecter `b` à `a`, mais dans un contexte objet, où la structure de `b` et de `a` peut inclure des objets alloués dynamiquement, tableaux, pointeurs et autres connexions à des bases de données, qui sait ce que signifie *affecter* outre les objets eux-mêmes?

Pouvoir définir le sens des opérateurs sur les objets est une suite logique de la démarche par laquelle on contrôle leur construction et leur destruction. L'équivalence opératoire des types est un proche cousin de l'encapsulation; offrir le même support syntaxique à tous les types, du plus humble au plus riche, est un petit triomphe de l'approche OO.

Clairement, on n'a pas un besoin absolu de définir des opérateurs sur nos types (après tout, des langages comme Java et VB.NET ne le permettent pas et la vie continue). À l'utilisation, toutefois, c'est drôlement plus élégant et plus agréable d'avoir cet outil que de ne pas l'avoir.

Comme dans bien d'autres cas, simplifier la vie des gens à l'utilisation d'un outil, d'un objet, ne veut pas nécessairement dire simplifier la vie des développeurs. Un objet, relevons-le, passe beaucoup plus de temps à être utilisé qu'à être conçu, surtout s'il a été bien pensé à l'origine. Réfléchir, parmi les opérations possibles sur un objet, aux opérateurs à implanter (et à ceux qu'il ne fait pas de sens d'implanter) fait partie du design de l'application.

Utilisant notre exemple de `Rectangle`, on est en droit de se demander quels opérateurs pourraient y être utiles. Certains montrent un potentiel plus clair que d'autres.

Affectation On voudrait assurément pouvoir affecter un `Rectangle` à un autre à l'aide de l'opérateur `=` :

```
Rectangle r1, r2{3, 5};
// ...
r1 = r2; // r1 devient une copie de r2
```

Comparaison On voudrait sûrement pouvoir dire si deux instances de `Rectangle` sont équivalentes⁸⁵ ou non, à l'aide des opérateurs `==` et `!=`⁸⁶ :

```
Rectangle r1, r2{3, 5};
// ...
if (r1 == r2)
    cout << "Ces rectangles sont egaux!";
```

⁸⁵ Dans la plupart des langages de programmation contemporains, on dira de deux objets qu'ils sont **équivalents** si on ne peut pas les distinguer sur la base de leurs états manifestes, alors qu'on réservera le vocable **identiques** pour des références distinctes ou des pointeurs distincts sur un même objet.

⁸⁶ Pour les besoins de cette illustration, on dira que deux instances de `Rectangle` sont identiques si elles ont la même hauteur et la même largeur. Cela dit, remarquez que c'est aux instances de `Rectangle` elles-mêmes de convenir ce que signifie *être égal à*, *être différent de*, *être plus grand que*... l'emploi d'opérateurs, surtout sous forme de méthodes (voir plus loin), s'inscrit directement dans une démarche d'encapsulation telle que la nôtre.

Ordonnancement On peut imaginer des situations où on voudrait savoir si une instance de `Rectangle` est plus petite (ou plus grosse) qu'une autre, à l'aide des opérateurs `<`, `<=`, `>` et `>=`⁸⁷ :

```
Rectangle r1, r2{3, 5};
// ...
if (r1 < r2)
    cout << "r1 a une plus petite surface que r2!";
```

etc. On pourrait poursuivre ainsi pendant longtemps. Par exemple, on pourrait penser utiliser l'opérateur `*=` pour faire croître la surface d'un `Rectangle` par un certain facteur entier :

```
Rectangle r{3, 5};
// ...
r *= 2; // doubler la surface de r...
r.Dessiner();
```

Notez que dans le cas ci-dessus, utiliser `*=` pour accroître la surface d'une forme ne sera pas nécessairement un choix que toutes et tous qualifieront d'évident. Comme dans bien des domaines, la modération est une avenue pleine de sagesse pour ce qui est de la surcharge d'opérateurs, et mieux vaut s'en tenir à des usages qui sembleront intuitifs et évidents pour la vaste majorité des gens.

En C++, *presque* tous⁸⁸ les opérateurs peuvent être surchargés, modifiés, adaptés à de nouveaux types. Il y a certaines règles de bon comportement lors de la mise au point des différents opérateurs (par exemple, l'opérateur `==` devrait être une fonction booléenne) mais, pour le reste, les opérateurs peuvent être adaptés aux besoins des projets et des situations.

Certaines opérations naturelles impliquent même notre classe `Rectangle` et d'autres classes. Il serait naturel, par exemple, de pouvoir exprimer l'idée de dessiner un `Rectangle` à la console par un programme comme celui proposé à droite.

C'est parce que de tels programmes sont naturels à l'utilisation qu'il est intéressant de réfléchir à des moyens de permettre au code client de les exprimer. Un objet bien pensé s'utilise *bien* et s'utilise *naturellement*.

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    Rectangle r{5, 8};
    cout << r << endl;
}
```

⁸⁷ Il faut bien sûr décider ce que *plus petit* et *plus grand* veulent dire... prenons pour acquis, pour notre exemple, que *plus grand* signifie *occupe une plus grande surface*.

⁸⁸ Les rares qui ne peuvent être surchargés sont `.`, `::`, `?:` et `.*` (certains ajoutent `#` et `##` mais ce sont des opérateurs du préprocesseur, pas du langage C++). Il est possible que la surcharge de l'opérateur `.` soit permise à partir de C++ 17. Cela dit, voir *Opérateurs et pratiques*, plus bas, pour les nuances nécessaires entre ce qu'il est *possible* de faire et ce qu'il est *souhaitable* de faire.

Modalités générales

L'idée générale de la surcharge d'opérateurs est un mélange de sucre syntaxique et de fonctionnalité adaptée aux classes sur une base individuelle.

Le compilateur, en voyant la notation infixe $a \otimes b$ pour un opérateur \otimes donné (+, =, *=, peu importe) remplacera cette forme « opérateur » par une forme « fonction ». C'est là le volet « sucre syntaxique » de la surcharge, soit une transformation silencieuse du code source qui permet une écriture naturelle du code client.

Fonctions globales ou méthodes?

On trouve deux grandes formes pour modéliser les surcharges d'opérateurs, soit la forme méthode d'instance et la forme fonction globale. Pour les illustrer de manière générale ces deux formes, nous utiliserons l'opérateur +, mais sachez que nous examinerons plus en détail les ramifications de chacune (car il y en a, qui touchent à la nature-même de ce que sont les objets) et les bons usages dans les sections qui suivent.

La forme « méthode d'instance » s'exprime comme proposé à droite. Ce qu'il faut y lire est que l'opérateur + est une méthode de X qui prend en paramètre une référence const sur un X et qui ne modifie pas l'instance active. L'opérande de gauche est *this, et l'opérande de droite est le paramètre de la méthode.

```
class X {
    // ...
public:
    X operator+(const X &) const;
    // ...
};
```

Cette forme, en pratique, remplacera l'écriture suivante :

```
X x0, x1
// ...
x0 = x0 + x1;
```

...par celle-ci :

```
X x0, x1
// ...
x0 = x0.operator + (x1);
```

...mettant en relief que l'opérande de gauche est aussi *this alors que l'opérande de droite est passé en paramètre à la méthode X::operator+(const X&). Les deux formes (avec ou sans sucre syntaxique) sont équivalentes.

Ici, notez qu'aucune duplication d'objet n'est requise par la signature, et que ni l'opérande de gauche (car la méthode est const), ni l'opérande de droite (car le paramètre est const) ne seront modifiés par l'action de cet opérateur.

La forme fonction globale s'exprimera entre autres comme proposé à droite. Cette forme est une fonction globale déclarée hors de la classe `X`, et ne pourrait donc avoir accès qu'aux membres publics de `X`.

```
class X {
    // ...
};
X operator+(const X&, const X&);
```

Une autre écriture possible serait celle-ci, qui (de par sa déclaration interne à `X` et de par sa qualification `friend`) aurait accès aux membres protégés et privés de `X`.

```
class X {
    // ...
    friend X operator+(const X&, const X&);
};
```

Selon les classes et les domaines d'application, l'une ou l'autre de ces trois formes sera préférable aux autres.

Ces deux formes, en pratique, remplaceront l'écriture suivante :

```
X x0, x1
// ...
x0 = x0 + x1;
```

...par celle-ci :

```
X x0, x1
// ...
x0 = operator + (x0, x1);
```

Dans ces deux formes, l'opérateur est une fonction globale. Il n'y a donc pas de `*this`. En retour, les deux opérandes sont clairement `const` et aucun paramètre n'est dupliqué inutilement dû à une invocation de l'opérateur.

Typiquement, on utilisera la forme « méthode d'instance » quand l'opérande de gauche est nécessairement du type de la classe qui le définit (habituellement, l'opérande de gauche est modifié par ces opérateurs). Les opérateurs comme `=`, `+=` et `*=` tombent dans cette catégorie.

Une méthode d'instance a aussi accès à tous les membres (pas seulement les membres publics) de l'opérande de gauche; une fonction globale⁸⁹ doit se limiter à utiliser les membres publics des instances sur lesquels elle opère.

On utilisera par contre la forme « fonction globale » pour les opérateurs qui ne modifient pas l'opérande de gauche, comme `-` (à un ou deux opérandes), `+`, `*` ou `&`. En fait, de manière générale, il est sage de favoriser la forme « fonction globale » pour les opérateurs *infixes* comme l'opérateur de division (l'opérateur `/`, comme dans `a/b`) ou les comparaisons (comme par exemple l'opérateur `==`).

⁸⁹ ...à moins qu'elle ne soit qualifiée `friend`, mais nous éviterons ce sujet pour le moment.

Exemple concret : soit la classe `Entier` proposée à droite. On y trouve une l'implémentation de l'opérateur `+` sous forme de méthode d'instance.

Cette méthode est `const`, évidemment, et retourne (par copie) un `Entier` dont la valeur est la somme des valeurs des deux instances d'`Entier` impliquées dans le calcul.

```
class Entier {
    int valeur_;
public:
    Entier(int valeur) : valeur_{valeur} {
    }
    int valeur() const {
        return valeur_;
    }
    Entier operator+(const Entier &e) const {
        return { valeur() + e.valeur() };
    }
};
```

Reprenons le même exemple, mais cette fois avec une l'implémentation de l'opérateur `+` sous forme de fonction globale. L'implémentation doit être faite dans le fichier source pour éviter une violation de la règle ODR (voir ***ODR – la One Definition Rule***).

Ici encore, la fonction retourne (par copie) un `Entier` dont la valeur est la somme des valeurs des deux instances d'`Entier` impliquées dans le calcul. Clairement, les deux approches sont équivalentes sur le plan des caractéristiques du programme à l'exécution.

```
class Entier {
    int valeur_;
public:
    Entier(int valeur) : valeur_{valeur} {
    }
    int valeur() const {
        return valeur_;
    }
};

Entier operator+(const Entier&, const Entier&);

// dans le .cpp
Entier operator+(const Entier &a, const Entier &b) {
    return { a.valeur() + b.valeur() };
}
```

Enrichir l'interface de classes existantes

La forme « fonction globale » ainsi peut être particulièrement utile dans le cas où une classe existe déjà, et où le besoin se fait sentir d'implanter un opérateur mettant entre autres en jeu *en tant qu'opérande de gauche* une instance de cette classe. Rares sont les gens qui parviennent à tout prévoir à l'avance, après tout.

Une classe est une entité fermée, à l'intérieur de laquelle on ne peut faire d'ajouts une fois sa déclaration terminée. Les opérateurs sous forme de fonction globale sont une manière d'enrichir l'interface d'un type *a posteriori*.

Par exemple, présumant une classe `Nombre` pour laquelle on souhaite implémenter l'opération `*` avec un `int` (dans le cas où on voudrait éviter d'utiliser l'intervention du moteur d'inférence de types qui générerait un `Nombre` temporaire à partir d'un `int`), on pourrait procéder comme proposé à droite.

```
class Nombre {
    // ...
public:
    // ...
    // Nombre * Nombre, forme « méthode d'instance »
    Nombre operator*(const Nombre&) const;
    // ...
};
// int * Nombre, forme « Fonction globale »
Nombre operator*(int, const Nombre&);
// Nombre * int, forme « Fonction globale »
Nombre operator*(const Nombre&, int);
```

Avant C++ 11, il était vu comme sain de faire en sorte que les opérateurs binaires comme `*` retournent une valeur `const`. Cette recommandation de **Scott Meyers** dans [EffCpp] visait à rendre illégale une expression comme `(a * b) = c;`, en soi déraisonnable, du fait que le résultat de `a * b` serait constant.

Cependant, depuis C++ 11, la sémantique de mouvement [POOv02], opportunité d'optimisation très significative, fait en sorte qu'on privilégiera retourner des copies non-`const` qui, étant anonymes, permettront au compilateur d'accélérer encore plus l'exécution de nos programmes.

Implanter l'affectation

L'affectation est un cas très particulier d'opérateur. C'est en effet l'un des rares opérateurs qui ne peut s'exprimer légalement *que* sous forme de méthode, jamais sous forme de fonction globale.

La signature générale de l'opérateur d'affectation `=` sur deux instances de type `T` est celle visible à droite:

```
class T {
    // ...
    T& operator= (const T&);
    // ...
};
```

Ce qui signifie que l'opérateur `=` prend en paramètre une référence vers un `const T` et retourne une référence sur un `T`. Certains, comme **Dave Abrahams**, préconisent de passer le paramètre par valeur [AbrSpdVal], mais la pratique la plus répandue est celle présentée ici.

Visualiser l'affectation

Sur des entiers `i0` et `i1`, procéder à l'affectation `i0 = i1`; signifie *faire de i0 une copie (au sens du contenu) de i1*. Notons que l'expression `i0 = i1`; signifie la même chose que l'expression `i0.operator=(i1)`; et que c'est `i0`, l'opérande de gauche, qui est l'instance propriétaire de la méthode (`*this`).

On s'attend à ce qu'`i1`, le paramètre, ne soit pas modifié par cette opération. C'est pourquoi il devrait être `const`. On s'attend aussi à ce qu'`i0` *soit* modifié par cette opération; c'est la raison d'être de cette méthode que de modifier l'instance qui en est propriétaire. C'est pourquoi la méthode elle-même n'est pas une méthode `const`.

On s'attend enfin à ce que le résultat de l'affectation soit l'opérande de gauche, pour qu'on puisse enfiler les affectations (p. ex. : `a = b = c`;). C'est pourquoi le type de la méthode devrait être le type de l'opérande de gauche.

En pratique, l'affectation est une opération binaire, au sens où elle s'applique à deux opérandes. L'opérande de gauche est ce qui se verra affecter quelque chose, et le contenu de l'opérande de droite est ce qui lui sera affecté.

À l'utilisation, on verra donc :

```
T a, // a est une instance de T
    b; // b est une instance de T
// ...
// appel de la méthode operator=(const T&) de a, prenant en paramètre b
a = b;
```

Concrètement, on implantera cette opération dans la classe `Rectangle` en ajoutant la méthode suivante à la déclaration de la classe.

```
class Rectangle {
    // ...
    Rectangle& operator=(const Rectangle&);
    // ...
};
```

La définition de cette opération aura pour rôle de faire de l'instance propriétaire de la méthode une copie de celle passée en paramètre⁹⁰.

```
// ...
Rectangle& Rectangle::operator=(const Rectangle &r) {
    SetLargeur(r.GetLargeur());
    SetHauteur(r.GetHauteur());
    return *this;
}
// ...
```

Notez qu'en vertu de l'encapsulation, le paramètre `r` ici est présumé valide *a priori*, donc si `Rectangle` offre des mutateurs bruts, sans validation, c'est ceux-ci qu'il faudrait privilégier, dans un souci d'efficacité.

L'instruction `return *this` est importante puisque la valeur de retour d'une affectation est celle de son opérande de gauche, de sorte que ceci :

```
T a, b, c;
// ...
a = b = c; // équivalent à a = (b = c);
```

soit possible et fasse d'abord `b = c`⁹¹, puis affecte le résultat de cette opération (qui est `b`) à `a`. Pour que ceci soit possible, il faut que l'opérateur `=`, dans l'opération `b = c`, retourne `b` tel qu'il est suite à l'exécution de l'opérateur d'affectation.

Et c'est précisément ce que signifie `return *this` : **je retourne une référence sur moi-même**. C'est là l'une des principales raisons pour lesquelles on a besoin, parfois, de demander à une instance de pouvoir faire référence à elle-même.

Note sur la Sainte-Trinité

Notez que l'affectation fait partie de la Sainte-Trinité, avec le constructeur de copie (voir **Constructeur de copie**) et le destructeur (voir **Destructeur**). Il s'agit donc une opération générée automatiquement par le compilateur, à moins bien sûr que la programmeuse ou le programmeur ne la prenne en charge explicitement; dans ce cas, cette opération procédera à une affectation attribut par attribut.

Conséquemment, dans un cas comme celui de `Rectangle`, il est plus simple (et plus efficace) de laisser le compilateur faire son travail.

```
// ...
Rectangle&
    Rectangle::operator=(const Rectangle &) = default;
// ...
```

⁹⁰ C'est d'ailleurs bien ce que fait l'affectation.

⁹¹ Ce qui se produira effectivement, l'affectation étant associative de droite à gauche.

Empêcher la copie

Tout d'abord, il est possible d'empêcher la copie par affectation d'un objet en déclarant un opérateur d'affectation privé pour sa classe et en s'abstenant de le définir.

Si cet opérateur n'est pas utilisé par la classe elle-même, on n'a même pas à le définir. Le fait qu'il soit *déclaré* privé suffit à en interdire l'accès..

Référez-vous à la remarque technique sur les constructeurs par copie pour plus de détails.

Depuis C++ 11, il est possible d'explicitement l'intention de supprimer une opération telle que l'affectation en apposant à la méthode correspondante le suffixe `= delete`.

Cette façon de faire est préférable, car plus claire et plus explicite.

En utilisant la suppression par `=delete`, il est préférable de laisser à l'opération supprimée la qualification d'accès public, pour que le message d'erreur soit, le cas échéant, le plus clair possible. On veut en effet voir « fonction supprimée », pas « fonction inaccessible ».

```
class X {
    // déclaré (privé) , non défini:
    // affectation illégale!
    X& operator=(const X&);
    // ...
};
```

```
class X {
    // ...
public:
    X& operator=(const X&) = delete;
    // ...
};
```

Un cas douloureux

Imaginons une classe comme la suivante :

```
// Texte.h
class Texte {
    char *texte_;
    int longueur_;
    // ... méthodes diverses, attributs divers ...
    void SetLongueur(int);
    // ... méthodes diverses, attributs divers ...
public:
    Texte(const char*);
    // ... méthodes diverses, constructeurs, destructeur ...
    int GetLongueur() const;
    // ... méthodes diverses ...
    Texte& operator=(const Texte &t);
    // ... méthodes diverses ...
};
```

...pour laquelle on trouve la définition suivante pour l'opérateur d'affectation :

```
// Texte.cpp
#include "Texte.h"
// ...
Texte& Texte::operator=(const Texte &t) {
    // l'accès direct à texte_ ici sert à alléger le propos
    delete [] texte_;
    texte_ = new char[GetLongueur()+1]; // réserver de l'espace pour un délimiteur de fin
    SetLongueur(t.GetLongueur());
    for (int i = 0; i < GetLongueur(); i++)
        texte_[i] = t.texte_[i];
    texte_[GetLongueur()] = '\0';
    return *this;
}
// ...
```

Ceci semble tout à fait légal (bien qu'il soit perfectible, pour un certain nombre de raisons), et l'est dans presque tous les cas.

Par exemple, le programme ci-dessous est correct :

```
#include "Texte.h"
int main() {
    Texte t0{ "Coucou" }; // constructeur paramétrique
    Texte t1; // constructeur par défaut
    t1 = t0; // affectation: Ok
}
```

Cependant, le programme suivant, qui devrait être légal, est *extrêmement* dangereux :

```
#include "Texte.h"
int main() {
    Texte t0{ "Coucou" }; // constructeur paramétrique
    t0 = t0; // Ouch!
}
```

Pourquoi ceci est-il aussi dangereux? Si nous examinons brièvement l'opérateur d'affectation plus haut, nous verrons que l'opérande de gauche *détruit ses propres données* avant de les reconstruire de manière à devenir identique, en fait de contenu, à l'opérande de droite.

Dans ce cas bien précis, toutefois, l'opérande de droite est à son insu (par décision souveraine du code client) le même objet que l'opérande de gauche, ce qui fait que *l'objet à gauche détruit l'objet à droite puis tente de faire une copie de données devenues invalides!*

Quelles sont les solutions à ce problème? Essentiellement, il y en a deux :

- l'opérande de gauche peut copier tous les attributs de l'opérande de droite dans des variables temporaires avant de se saborder et de se reconstruire. Ceci fonctionne dans la plupart des cas⁹² mais peut être coûteux en ressources si le contenu est massif; et
- l'opérande de gauche peut d'abord prendre soin de vérifier s'il est aussi l'opérande de droite et escamoter la copie dans ce cas, tout simplement.

La seconde option est préférable. Reprenant l'exemple de la classe `Texte` plus haut, un opérateur d'affectation correct deviendrait tel que proposé à droite.

Il importe de comparer l'adresse des objets, pas leur contenu, car deux objets de même contenu mais différents ne constituent pas un problème ici (et parce que *comparer des contenus peut être arbitrairement long!*).

On peut toutefois faire encore mieux...

```
Texte & Texte::operator=(const Texte &t) {
    if (this != &t) {
        delete[] texte_;
        SetLongueur(t.GetLongueur());
        texte_ = new char[GetLongueur()+1];
        for (int i = 0; i < GetLongueur(); i++)
            texte_[i] = t.texte_[i];
        texte_[GetLongueur()] = '\0';
    }
    return *this;
}
```

⁹² Il peut y avoir des cas étranges où cette stratégie est risquée. Laissez courir votre imagination...

L’idiome d’affectation sécuritaire⁹³

Il existe une manière universelle d’implémenter, de manière sécuritaire et efficace, l’affectation. Elle repose toutefois sur une bonne implémentation :

- du constructeur de copie;
- du destructeur; et
- d’une méthode clé, soit la méthode `swap()`, qui aura pour rôle d’échanger les valeurs des états de deux objets.

Le constructeur de copie est le lieu où doit se situer la duplication de contenu des objets. Rien ne sert de répéter ce code ailleurs.

Le destructeur est le lieu où le contenu des objets est nettoyé. Lors d’une affectation, il faut nécessairement disposer du contenu de l’opérande de gauche, puisque (par définition) cet objet existait déjà au préalable.

La méthode `swap()` doit être très efficace (complexité constante) et, idéalement, ne lever aucune exception. Il est toujours possible de concevoir les classes pour atteindre ce résultat.

Reprenons la classe `Texte` sous cette lueur :

```
// Texte.h
#include <utility> // std::swap()
class Texte {
    char *texte;
    int longueur;
    // ... méthodes diverses, attributs divers ...
public:
    Texte(const char*);
    // ... méthodes diverses ...
    Texte(const Texte &);
    ~Texte();
    void swap(Texte &t) {
        using std::swap;
        swap(texte, t.texte);
        swap(longueur, t.longueur);
    }
    Texte & operator= (const Texte &t);
    // ... méthodes diverses ...
};
```

Si le constructeur de copie est correctement implémenté, et si la méthode `swap()` est convenablement écrite, l’affectation s’exprimera nécessairement comme ceci.

```
Texte& Texte::operator=(const Texte &t) {
    Texte{t}.swap(*this);
    return *this;
}
```

⁹³ Cette technique est due *Jon Kalb*, bien que je doive mon premier contact avec elle à *Herb Sutter*, dans [ExcCpp]. Rendons à César...

L'expression `Texte{t}` crée une copie de `t`, qui (étant anonyme) n'existe que pour l'expression en cours. L'invocation de `swap()` échange le contenu de cette temporaire avec le contenu de l'instance active.

Suite au `swap()`, l'instance active aura comme contenu une copie de ce que contenait le paramètre `t`, alors que la copie (générée de manière anonyme) contiendra ce que l'instance active contenait au préalable.

Enfin, le destructeur de l'instance anonyme de `Texte` éliminera le contenu de l'instance active telle qu'elle était avant l'invocation de la méthode. Le paramètre n'aura pas changé de contenu, l'instance active contiendra une copie du contenu du paramètre, et toutes les données intermédiaires auront été nettoyées par de le destructeur de la variable temporaire anonyme.

Si vous rejoignez [AbrSpdVal] et choisissez de privilégier le passage par valeur dans l'implémentation de l'opérateur d'affectation, alors l'écriture sera celle proposée à droite. C'est plus simple, mais moins idiomatique pour le moment.

```
Texte& Texte::operator=(Texte t) {
    swap(t);
    return *this;
}
```

Implanter les comparaisons

La signature générale de l'opérateur de comparaison `==` (et de son inverse, `!=`), sous forme « méthode d'instance » pour deux objets de type `T` est telle que visible à droite.

```
class T {
    // ...
    bool operator==(const T&) const;
    bool operator!=(const T&) const;
    // ...
};
```

Ceci signifie *opérateur == (ou !=) prend deux opérands constants de type T (pourquoi comparer deux objets modifierait-il l'un ou l'autre de quelque manière, après tout?), et dont la valeur de retour sera booléenne.*

La forme « fonction globale », quant à elle, est celle visible à droite. La nuance entre la version `friend` et celle qui ne l'est pas est indiquée plus haut.

```
class T {
    // ...
    friend bool operator==(const T&, const T&);
    friend bool operator!=(const T&, const T&);
    // ...
};
```

Pour `Rectangle`, une écriture convenable pour ces opérations serait celle proposée à droite. Remarquez l'expression de `!=` à partir de la définition de `==`.

```
class Rectangle {
    // ...
    bool operator==(const Rectangle &r) const {
        return GetLargeur() == r.GetLargeur() &&
            GetHauteur() == r.GetHauteur();
    }
    bool operator!=(const Rectangle &r) const {
        return !(*this == r);
    }
    // ...
};
```

Cette approche assure une cohérence entre ces deux opérations. On pourrait implémenter chacune individuellement, et la politique selon laquelle deux instances de `Rectangle` sont considérées équivalentes dépend du domaine d'application, mais peu importe les règles d'équivalence, il est souhaitable que $!(a == b) \Leftrightarrow a != b$.

Égalité ou équivalence

Deux idées distinctes, l'égalité et l'équivalence (certains disent aussi identité et égalité), se confondent parfois dans le discours mais méritent d'être examinées en tant qu'entités distinctes. En particulier, le sens que chaque langage OO donne à l'une ou l'autre de ces idées est un indicateur de certains choix philosophiques qui le sous-tendent.

Pour en savoir plus sur les sémantiques directes et indirectes d'accès aux objets, je vous invite à lire l'appendice OO de ce document.

Les langages OO pour lesquels la fin de la vie des objets repose sur les mécanismes d'un moteur de collecte automatique d'ordures, comme c'est le cas de Java et des langages .NET, font en sorte que les objets soient toujours manipulés indirectement. L'opérateur == dans ces langages exprime donc, à moins d'une surcharge (possible en C#), une relation d'identité : $a==b$ si et seulement si a et b sont un seul et même objet.

Typiquement, d'ailleurs, dans ces langages, l'affectation d'objets copie des références, pas des contenus. Ainsi, en Java, $a=b$ signifie *la référence a mène maintenant au même endroit que la référence b*, et les états de l'objet référé par a avant l'application de l'opérateur = ne sont pas modifiés.

Évidemment, si a et b mènent tous deux au même objet, alors modifier l'un modifie aussi l'autre, un problème rencontré sans arrêt en Java ou dans les langages .NET et qu'on nomme l'**aliasing**. Typiquement, dans ces langages, la meilleure solution est de ne pas permettre ces modifications, et de concevoir les classes de ces objets comme étant *immuables*.

L'illustration (en Java) à droite montre brièvement les ramifications de cette situation. Les instances sont créées par `new` et les entités $e0$, $e1$ et $e2$ sont des références (des indirections) sur les objets créés, pas des objets à proprement dit. Ainsi, deux objets seulement sont instanciés dans ce programme, même s'il existe trois références sur des objets.

L'alternative évaluant $e0 == e2$ en fin de programme teste l'identité, pas l'égalité. Cette condition sera vraie puisque $e0$ et $e2$ réfèrent au même objet. En retour, la condition évaluant $e0 == e1$ sera fausse même si les objets référés par ces deux références ont des états de même valeur.

Puisque == dans ces langages ne compare pas les contenus référés mais bien les références, ces langages proposent typiquement des idiomes de comparaison de contenu, reposant sur une méthode booléenne `equals()`.

Ces méthodes sont subtiles à implémenter, pour des raisons que nous ne pourrions couvrir en détail que dans [POOv01].

```
class Entier { // pas immuable
    private int valeur;
    public Entier(int valeur) {
        setValeur(valeur);
    }
    public int getValeur() {
        return valeur;
    }
    public void setValeur(int valeur) {
        this.valeur = valeur;
    }
}

class Test {
    public static void main(String [] args) {
        Entier e0 = new Entier(3);
        Entier e1 = e0;
        Entier e2 = new Entier(3);
        if (e0 == e2) {
            System.out.println("identité");
        }
        if (e0 == e1) {
            System.err.println("Oups!");
        }
    }
}
```

La surcharge de l'opérateur `==` en C++ porte quant à elle sur les objets comparés, pas sur des références. L'illustration à droite montre à la fois l'implémentation de l'équivalence, à travers `==` sur des objets, et un exemple de test d'identité, à travers `==` sur des pointeurs.

L'alternative évaluant `e0 == e1` en fin de programme teste l'égalité, invoquant l'opérateur `==` sur deux instances d'`Entier`. Cet opérateur réalise une comparaison de contenus. Les objets `e0` et `e1` ayant les mêmes états au sens de l'opérateur `==`, cette comparaison programmée retournera `true`.

L'alternative évaluant `p0 == p1` en fin de programme teste l'identité, invoquant l'opérateur `==` sur deux pointeurs d'`Entier`. Cet opérateur réalise une comparaison d'adresses, comme en Java; puisque `e0` et `e1` sont des objets distincts, ces deux objets occupent des espaces distincts en mémoire, et la comparaison de leurs adresses à l'aide de l'opérateur `==` aura pour résultat `false`.

```
#include <iostream>
class Entier { // pas immuable
    int valeur_;
public:
    Entier(int valeur) : valeur_{valeur} {
    }
    int valeur() const {
        return valeur_;
    }
    bool operator==(const Entier& e) const {
        return valeur() == e.valeur();
    }
}
int main() {
    using namespace std;
    Entier e0 = 3;
    Entier e1 = 3;
    Entier *p0 = &e0;
    Entier *p1 = &e1;
    if (e0 == e1)
        cout << "équivalence" << endl;
    if (p0 == p1)
        cerr << "Oups!" << endl;
}
```

Implanter l'ordonnement

La signature générale de l'opérateur de comparaison `<` sur deux objets de type `T` est celle visible à droite.

Ceci signifie *opérateur* `<` dont les opérandes sont toutes deux de type `T` et sont tous deux constants (il n'y a pas de raison pour que comparer deux objets modifie l'un ou l'autre de quelque manière), et dont la valeur de retour sera booléenne.

```
class T {
    // ...
    bool operator<(const T&) const;
    // ...
};
```

Concrètement, on implantera cette opération dans la classe `Rectangle` en ajoutant la méthode suivante à la déclaration de la classe :

```
bool Rectangle::operator<(const Rectangle &r) const {
    return GetAire() < r.GetAire();
}
```

Pour vous pratiquer, vous pouvez réaliser vous-mêmes l'implantation des opérateurs `<=`, `>` et `>=`. On ne parle (à l'évidence) pas d'une tâche très lourde, vous en conviendrez.

Note : plusieurs algorithmes standards de C++ pour lesquels une relation d'ordre est nécessaire (p. ex. : des algorithmes de tri) exigent qu'il soit possible de comparer deux instances des types auxquels on les applique à l'aide de l'opérateur `<`.

Dans la littérature, une classe pour laquelle les opérateurs `==` et `<` sont définis de manière conforme aux usages, l'opérateur `==` exprimant l'équivalence (`a == b` étant vrai si `a` et `b` sont équivalents, et ne peuvent être distingués du point de vue des états) et l'opérateur `<` exprimant une relation d'ordre (`a < b` signifiant `a` précède `b`), est dite capable d'ordonnement au sens strict (en anglais : *Strick Weak Ordering*). C'est typiquement l'exigence minimale pour l'application d'un algorithme de tri à un type donné⁹⁴.

⁹⁴ L'opérateur `!=` peut être exprimé à partir de `==` et de `!`, alors que les opérateurs `>`, `<=` et `>=` peuvent être exprimés à partir de `<` et de `!`.

Implanter l'autoincrémentation et l'autodécrémentation

Un cas pathologique pour l'expression des opérateurs sous forme de méthode est celui de l'opérateur d'autoincrémentation `++` (les mêmes remarques s'appliquent à l'opérateur d'autodécrémentation `--` bien entendu).

La raison de cette situation est que l'opérateur `++` se décline sous deux formes, soit la forme préfixée (p. ex. : `++i`) et la forme suffixée (p. ex. : `i++`). Ces deux formes sont semblables mais légèrement différentes.

Prises isolément, ces formes donnent en fin de compte toutes deux le même résultat (voir à droite). En effet, `++i` et `i++` sont toutes deux des opérations réalisant une autoincrémentation sur `i`, donc amenant `i` à la prochaine valeur pour son type.

Si l'examen de ces opérateurs s'arrêtait à ce niveau, alors il y aurait lieu de s'interroger sur la pertinence de supporter les deux formes.

```
int i;
i = 3,
++i;
cout << i; // 4
i = 3,
i++;
cout << i; // 4
```

La situation se corse (et devient plus intéressante) lorsque l'autoincrémentation est insérée dans une expression plus complexe.

Force est alors de constater que la version préfixée et la version suffixée sont toutes deux légèrement différentes :

- la version préfixée, symbolisée par `++i`, incrémente `i` puis retourne sa valeur *après* incrémentation; alors que
- la version suffixée, symbolisée par `i++`, incrémente `i` puis retourne sa valeur *avant* incrémentation.

```
int i;
i = 3,
cout << ++i; // 4
i = 3,
cout << i++; // 3
```

Sur le plan de la syntaxe, le problème est d'un autre ordre: voilà deux opérateurs unaires susceptibles d'être implémentés sur un même type. Les deux ont pratiquement la même syntaxe mais un sens légèrement différent. Comment les différencier?

Pour illustrer la procédure, imaginons une classe `Entier` représentant... un entier, et implémentons-y les deux versions de l'opérateur `++`.

La version préfixée s'exprime par un opérateur `++` sans paramètre et retournant une référence sur le type auquel s'applique l'opérateur.

La version suffixée s'exprime par un opérateur `++` ayant un paramètre de type `int` (qu'on ne nommera pas) et retournant une copie d'une instance du type auquel s'applique l'opérateur.

```
class Entier {
    int valeur_;
public:
    // ...
    Entier& operator++(); // forme préfixée
    Entier operator++(int); // forme suffixée
    // ...
};
```

Le compilateur parvient à différencier les formes l'une de l'autre à l'aide du paramètre anonyme de la forme suffixée.

Comment rédiger le code de chacune des deux versions de l'opérateur d'autoincrémentation? C'est chose relativement simple :

- la version préfixée modifie l'instance propriétaire de la méthode et retourne une référence sur cette instance une fois les modifications apportées; alors que
- la version suffixée crée une copie temporaire de l'instance propriétaire de la méthode, puis modifie l'instance propriétaire. Enfin, la copie temporaire est retournée à l'appelant (par valeur). Cette stratégie retourne la valeur de l'objet avant l'appel tout en modifiant l'objet.

Le résultat de ces séquences est conforme à l'intuition et aux attentes. Vous remarquerez que la version préfixée est plus légère que la version suffixée, épargnant un constructeur par copie et le destructeur correspondant.

Ceci explique que les fanatiques de performance, lorsque le choix se pose entre les deux formes sans que cela n'entraîne d'expressions ambiguës ou incorrectes, privilégieront systématiquement la forme préfixée.

Si vous trouvez déplaisante (et ça se comprendrait!) cette tare syntaxique qui impose d'insérer un `int` fantôme en tant que paramètre aux versions suffixes de ces opérateurs, vous pouvez vous inspirer de ce truc de **Richard Smith** (voir à droite) et définir des alias signifiants sur des types (en utilisant `void` pour exprimer l'absence de paramètre) pour clarifier l'écriture :

```
// forme préfixée
Entier& Entier::operator++() {
    ++valeur_;
    return *this;
}

// forme suffixée
Entier Entier::operator++(int) {
    Entier temp(*this);
    operator++();
    return temp;
}
```

```
using prefixe = void;
using suffixe = int;
class Entier {
    // ...
    Entier& operator++(prefixe);
    Entier operator++(suffixe);
    // ...
};
```

Opérateur d'écriture sur un flux

Serait-il raisonnable de prétendre que le programme à droite *devrait* dessiner un `Rectangle` de hauteur 5 et de largeur 3 à la console? Il est probable que votre réponse soit *oui*. Si c'est le cas, alors il vous faut vous demander comment arriver à écrire le code en conséquence.

```
#include "Rectangle.h"
#include <iostream>
int main() {
    using namespace std;
    Rectangle r{ 5, 3 };
    cout << r << endl;
}
```

Il se trouve que `std::cout` est une instance de la classe `std::ostream` (pour *Output Stream*, ou flux de sortie). Cela peut sembler étrange à première vue, mais il se trouve que `std::cout` est un objet global⁹⁵, représentant ce qu'on nomme conventionnellement la *sortie standard* (la sortie à l'écran en mode console, pour simplifier).

Il se trouve aussi que, quelque part dans la classe `std::ostream`⁹⁶, on retrouve l'opérateur `<<` défini pour les types primitifs du langage C++. Dans la définition de cet opérateur pour chaque type, on voit apparaître les opérations requises pour afficher une donnée du type en question à la console.

Afficher des données de type primitif avec `std::cout` est donc une tâche banale: tout le travail permettant d'y arriver est déjà fait à même les bibliothèques standard du langage.

Cela dit, il est possible de définir le comportement de l'opérateur `<<` de la classe `std::ostream` sur des types autres que les types primitifs du langage. On le constate entre autres du fait que, on l'a déjà vu sans vraiment le réaliser, les instances de `std::string` peuvent elles aussi être affichées à la console avec `std::cout` et l'opérateur `<<`.

```
#include <iostream>
#include <string>
void f() {
    using namespace std;
    string s = "Allo!"
    cout << s << endl;
}
```

Or, *on se souviendra que `std::string` est une classe, pas un type primitif du langage!*

La classe `std::ostream` étant d'ores et déjà définie (dans une bibliothèque standard, en plus), on ne peut y ajouter des méthodes : les classes sont des espaces fermés. Pour arriver à afficher un `Rectangle` avec `std::cout` et l'opérateur `<<`, nous devons donc décrire, *en tant que fonction globale*, l'opérateur `<<` appliqué à deux opérands.

En travaillant à enrichir les services applicables aux instances de `std::ostream` en général, nous obtiendrons par contre un bonus important, soit la capacité d'écrire un `Rectangle` sur n'importe quel flux en sortie standard, quel qu'il soit.

⁹⁵ ... global à l'espace nommé `std`, du moins.

⁹⁶ Attention : nous tournons un peu les coins ronds ici, faute d'avoir présenté le concept d'héritage...

Nature des opérandes

Examinons à nouveau l'expression (réduite à sa plus simple expression : on crée un `Rectangle` anonyme, on l'affiche et il s'autodétruit) que nous souhaitons rendre possible ici, et identifions-en les principaux éléments.

```
cout << Rectangle{5, 3};
```

L'opérande de gauche sera du type de `std::cout`, soit un `std::ostream&`. La référence tient au fait qu'écrire sur un flux modifie son état. L'opérande de droite, lui, sera de type `const Rectangle&`. On aurait pu choisir d'utiliser le type `Rectangle` (par valeur), mais cela impliquerait la construction et la destruction (inutiles) d'instances de `Rectangle`, ce qui ralentirait le code sans apporter quelque bénéfice que ce soit.

Cette fonction devra retourner un `std::ostream&` (qui sera le même que reçu en paramètre, donc probablement `std::cout` lui-même), ce qui permettra d'enchaîner les affichages de manière normale, comme dans l'exemple ci-dessous (le cas de `std::endl` est complexe et sera couvert dans un volume plus avancé) :

```
// ...
void f(const Rectangle &r) {
    cout << "Voici mon rectangle de hauteur " // operator<<(ostream&,const char*)
        << r.hauteur()                       // operator<<(ostream&,int)
        << " et de largeur "                 // operator<<(ostream&,const char*)
        << r.largeur()                      // operator<<(ostream&,int)
        << endl << r << endl;              // operator<<(ostream&,const Rectangle&)
}
```

Implémentation de l'opérateur

Ceux qui voudront cet opérateur voudront nécessairement projeter un `Rectangle` sur un flux. Conséquemment, nous placerons la déclaration de l'opérateur dans `Rectangle.h`.

Puisque le prototype de l'opérateur utilise un paramètre et une valeur de retour de type `std::ostream&`, il faudra inclure, avant le prototype de l'opérateur. On pourrait inclure `<iostream>`, mais ce fichier est très lourd et il est mal vu de l'inclure dans un fichier d'en-tête. Normalement, à moins de ne pas avoir d'autres choix, un fichier d'en-tête inclura plutôt `<iosfwd>`, qui est une version allégée de `<iostream>`, et les fichiers sources qui en ont vraiment besoin incluront `<iostream>` localement. Cela réduit le temps de compilation, et augmente la productivité.

Le fichier `Rectangle.h` devra donc être modifié pour y ajouter ce qui suit :

```
// En-tête d'usage et inclusions diverses (omises pour économie d'espace)
#include <iosfwd>
class Rectangle {
    // ...
};
std::ostream& operator<<(std::ostream&, const Rectangle&); // Prototype de l'opérateur
// ...
```


La définition de l'opérateur sera placée dans le fichier source `Rectangle.cpp`, qui inclura `<iostream>` puisqu'il en fera réellement usage (le `.h` n'avait, au fond, besoin que des déclarations de classes).

La définition de l'opérateur ressemblera à celle visible à droite. Remarquez qu'on ne peut utiliser *implicitement* les méthodes de `r`, notre instance de `Rectangle`, puisqu'il s'agit là d'une fonction globale, pas d'une méthode de `Rectangle`.

Le code proposé est un calque de celui de la méthode `dessiner()` d'un `Rectangle`, ce qui est raisonnable puisque les deux sous-programmes réalisent précisément la même tâche.

Ceci suggère que nous pouvons raffiner un peu notre démarche : rien de pire pour l'entretien des programmes que la duplication de code.

L'opérateur, en plus d'être une fonction globale plutôt qu'une méthode, a aussi la particularité d'être plus général que la méthode `dessiner()`; cette dernière ne peut dessiner qu'à l'écran en mode console, alors que l'opérateur que nous venons de rédiger peut dessiner dans n'importe quel flux de sortie, celui de l'écran en mode console inclus.

En fait, avec ce que nous venons d'ajouter la méthode `dessiner()` peut maintenant s'écrire tout simplement comme ci-contre, mais cette version n'est pas assez générale pour la plupart des gens.

```
#include "Rectangle.h"
#include <iostream>
using namespace std;
ostream& operator<<
    (ostream &flux, const Rectangle &r) {
    for (int i = 0; i < r.hauteur(); i++) {
        for (int j = 0; j < r.largeur(); j++)
            flux << '*';
        flux << '\n';
    }
    return flux;
}
// ...
```

```
void Rectangle::dessiner() const {
    cout << *this; // ok, mais bof
}
```

Procédons à une *refactorisation* de `Rectangle` pour tenir compte de cette généralisation que nous venons d'y introduire, soit la capacité de dessiner un `Rectangle` sur n'importe quel flux en sortie qui soit conforme au standard.

Une solution générale et élégante à ce problème irait comme suit :

- déclarer une méthode `dessiner()` générale, capable de projeter un `Rectangle` sur un flux quelconque. C'est elle qui dessinera véritablement un `Rectangle`;
- conserver la méthode `dessiner()` sans paramètres qui se limite à projeter un `Rectangle` sur la sortie standard, pour éviter de briser le code client existant⁹⁷; et
- conserver l'opérateur de projection d'un `Rectangle` sur un flux en sortie standard, pour faciliter l'intégration de notre classe à l'écosystème de C++.

L'implémentation de ces trois opérations se fait, quant à elle, de la manière annoncée plus haut :

- le `dessiner()` général, sur un flux quelconque, qui fait le travail;
- le `dessiner()` spécialisé pour la sortie standard, qui est un simple raccourci déléguant le travail au `dessiner()` général; et
- l'opérateur de projection sur un flux, qui délègue le travail à la version générale de la méthode `dessiner()`.

Le code n'est pas dupliqué, la fonctionnalité offerte par l'objet (son interface publique) est enrichie, l'objet est mieux intégré au système qu'auparavant, et aucun coût à l'exécution n'a été encouru. Surtout, nos efforts n'ont pas brisé le code client existant de notre classe.

```
// ...
#include <iosfwd>
class Rectangle {
    // ...
public:
    void dessiner(std::ostream& const; //
général
    void dessiner() const; // sortie standard
    // ...
};
std::ostream& operator<<
    (std::ostream&, const Rectangle&);
// ...
```

```
#include "Rectangle.h"
#include <iostream>
using std::ostream;
void Rectangle::dessiner(ostream &os) const {
    for (int i = 0; i < hauteur(); i++) {
        for (int j = 0; j < largeur(); j++)
            os << "*";
        os << '\n';
    }
}
void Rectangle::dessiner() const {
    dessiner(std::cout);
}
ostream& operator<<
    (ostream &flux, const Rectangle &r) {
    r.dessiner(flux);
    return flux;
}
// ...
```

⁹⁷ Nous aurions pu n'écrire qu'une seule version de la méthode `dessiner()` en utilisant la forme générale et en lui appliquant un paramètre par défaut (`std::cout`), mais cela nous aurait demandé d'inclure `<iostream>`, qui déclare `std::cout`, dans un fichier d'en-tête, ce qui aurait constitué un manque de bienséance.

Fruits de la généralisation

L'un des effets secondaires de cette généralisation, permettant de projeter un `Rectangle` sur n'importe quel flux de sortie conforme au standard, est qu'il existe plusieurs autres catégories de flux de sortie que celui de la console.

Sans vouloir entrer dans les détails, notons par exemple que le fichier d'en-tête standard `<fstream>` définit les classes `std::ifstream` (pour *Input File Stream*, ou flux d'entrée d'un fichier) et `std::ofstream` (pour *Output File Stream*, ou flux de sortie dans un fichier).

Bien que nous n'ayons pas encore couvert l'héritage, notons que la classe `std::ifstream` est une *spécialisation* de la classe `std::istream` (dont `std::cin` est une instance), et que la classe `std::ofstream` est une spécialisation de la classe `std::ostream` (dont `std::cout` est une instance).

Ainsi, sans entrer dans le détail de la mécanique, nous sommes en mesure de comprendre que *si on peut écrire dans un `std::ostream`, alors on devrait pouvoir écrire dans le cas particulier de `std::ostream` qu'est un `std::ofstream`.*

Je vous propose donc d'essayer le programme suivant, puis d'examiner le contenu du fichier `Surprise.txt` avec un éditeur de texte de votre choix. Si vous voulez en savoir plus, vous pouvez lire la documentation des classes en jeu.

```
#include "Rectangle.h"
#include <fstream>
int main() {
    using namespace std;
    Rectangle r{ 5, 3 };
    ofstream fichier{ "Surprise.txt" };
    fichier << "Voici mon rectangle de hauteur "
        << r.hauteur() << " et de largeur " << r.largeur() << '\n'
        << r // magie!
        << endl;
}
```

Opérateur de lecture sur un flux

Il est possible d'implanter la lecture sur un flux en entrée de manière semblable à celle selon laquelle on implante les opérations de projection sur un flux en sortie. Quelques nuances s'imposent toutefois :

- l'écriture sur un flux projette un objet présumé correct, en vertu de l'encapsulation. La lecture sur un flux, elle, ne peut présumer que les données en entrée sont convenables, et tend donc à être un peu plus complexe que l'écriture du fait qu'elle doit tenir compte des erreurs et de la corruption des données;
- on souhaite habituellement que, pour un objet qu'il est possible d'écrire sur un flux et de lire d'un flux, les opérations de lecture et d'écriture soient symétriques, donc que le programme bidon à droite affiche précisément la même chose à la console lors des deux projections sur `std::cout`;
- pour en arriver à cette symétrie, il faut que la projection sur un flux soit minimale et que la consommation à partir d'un flux consomme selon un format connexe à celui utilisé pour l'écriture. Si un programme souhaite mettre de l'enrobage cadeau autour d'un affichage (menus, texte explicatif ou autre), cet enrobage est la responsabilité du code client, pas celle des objets affichés.

Notez que les accès aux fichiers dans l'exemple à droite sont minimalistes : les fichiers en entrée et en sortie n'ont pas de noms et n'existent que pour l'instruction où ils sont définis. Si vous souhaitez les utiliser plus longtemps, donnez des noms à ces variables anonymes!

```
#include "X.h"
#include <fstream>
#include <iostream>
int main() {
    using namespace std;
    X x;
    // ...
    cout << x << endl;
    ofstream("fichier.txt") << x;
    ifstream("fichier.txt") >> x;
    cout << x << endl;
}
```

Puisque nous visons la symétrie dans les opérations d'entrée/ sortie pour un type donné, nous utiliserons ici un type plus simple que le type `Rectangle` (je doute qu'il soit pertinent de lire des séquences de `'*'` sur un flux).

Imaginons une classe `Taux` représentant un pourcentage. Nous tiendrons les détails de cette classe au minimum pour nous concentrer sur les éléments propres aux écritures et (surtout) aux lectures.

Nous souhaiterons que l'affichage d'un taux de 25,3% affiche `25.3%`, symbole de pourcentage inclus, et que la lecture d'un taux ne soit considérée valide que si le taux lu porte le suffixe `'%'`.

Notez qu'il est possible (mais pas du ressort de ce volume) d'adapter l'écriture des nombres aux standards locaux (donc, au Québec, de réaliser des entrées/ sorties sur des réels en utilisant la virgule plutôt que le point comme séparateur pour les décimales).

```
// ...
#include <iosfwd>
using namespace std;
class Taux {
    // ...
public:
    float valeur() const;
    Taux();
    explicit Taux(float);
};
ostream& operator<<(ostream&, const Taux&);
istream& operator>>(istream&, Taux&);
// ...
```

Remarquez la différence de signature entre les opérateurs de lecture sur un flux et d'écriture sur un flux. La lecture prend une référence non constante comme second paramètre puisqu'elle doit modifier l'objet dans lequel la lecture est réalisée. L'écriture, elle, prend plutôt une référence constante du fait qu'écrire un objet sur un flux ne devrait pas modifier l'objet projeté.

L'implémentation des opérateurs de lecture et d'écriture sur un flux pour `Taux` témoignent des différences de complexité entre ces deux tâches.

L'écriture est simple : projeter la valeur sur le flux, puis y écrire le suffixe `' % '`, tel que l'indiquait l'énoncé de mission plus haut.

La lecture est d'un tout autre ordre :

- il faut tout d'abord s'assurer que le flux en entrée soit en bon état. Si le flux est en mauvais état, tenter d'y lire ne fera pas planter le programme pourrait modifier la nature de l'erreur qui s'y trouve, alors mieux vaut ne pas perdre cette information;
- par la suite, un lire la valeur du `Taux` et le suffixe, et on validera ces lectures;
- seulement si tout cela s'est bien déroulé, on pourra modifier le `Taux` reçu en paramètre.

Notez que j'ai utilisé l'affectation et le constructeur paramétrique pour ce faire plutôt que d'implémenter un mutateur; c'est souvent une bonne stratégie.

```
#include "Taux.h"
#include <iostream>
using namespace std;
ostream& operator<<(ostream &os, const Taux &t) {
    return os << t.valeur() << '%';
}
istream& operator>>(istream &is, Taux &t) {
    if (is) {
        float valeur;
        char c;
        if (is >> valeur >> c && c == '%')
            t = Taux{valeur};
    }
    return is;
}
// ...
```

Tester l'état d'un flux en C++ chose est simple : il suffit de le considérer comme un booléen. Tester le résultat d'un accès au flux se fait de la même manière puisque les opérations sur les flux, pour qu'on puisse les enchaîner, retournent toutes le flux sur lequel l'opération a été réalisée.

Le programme proposé à droite lit un `taux` sur l'entrée standard (`std::cin`) et affiche ce `taux` seulement s'il est conforme (un réel suivi d'un symbole `' % '`). Dans le cas contraire, un message d'erreur est affiché.

```
#include "Taux.h"
#include <iostream>
int main() {
    using namespace std;
    Taux t;
    if (cin >> t)
        cout << t << endl;
    else
        cerr << "Erreur: pas un taux\n";
}
```

Brève parenthèse sur la sérialisation et la persistance

S'il est possible d'écrire un objet sur un flux et s'il est par la suite possible d'extraire ce même objet du flux, comme dans le cas du type `Taux` ci-dessus, alors on dira que cet objet peut être sérialisé, qu'il est *sérialisable*.

Certains parleront aussi d'**objets persistants** puisqu'ils pourront alors entreposer leurs états sur une mémoire de masse de manière à pouvoir récupérer ces états ultérieurement.

S'il n'est pas vraiment possible d'aborder la sérialisation en profondeur à ce stade-ci, indiquons tout de même que l'objet sérialisé (le `Taux`) n'est jamais écrit sur un flux ou lu d'un flux. Ce qui est écrit sur un flux et ce qui est lu d'un flux est dans chaque cas un *descriptif* de l'objet (ici, une simple suite d'attributs, écrite et lue dans le même ordre).

Les langages comme C# ou Java procèdent de la même manière, à notre insu. Un objet n'est jamais *sérialisable* au sens strict du terme, mais sa description peut l'être.

La sérialisation est une thématique importante dans les langages OO. La capacité de projeter un objet sur un flux et d'extraire un objet d'un flux comme on pourrait le faire pour toute donnée d'un type primitif s'inscrit dans une démarche de conformité opérationnelle des types.

Un langage OO de qualité devrait offrir un système de types homogène, au sens où tous les types du langage et tous les types créés à l'aide du langage devraient, du moins en théorie, avoir droit au même niveau de support de la part du compilateur. Nous y reviendrons dans [POOv03], section *Que signifie être OO?*

Sachant cela, il est sage de proposer des opérateurs de sérialisation pour tout type de votre cru, à moins d'avoir une excellente raison pour ne pas le faire. Ceci facilitera l'utilisation naturelle de vos propres objets.

Opérateurs de conversion et accesseurs

Les accesseurs sont des outils primitifs de consultation de l'état d'un objet. Certains objets ont la particularité d'être tellement proches, conceptuellement, d'un autre type qu'on voudrait parfois faire comme s'ils en étaient.

Pensons par exemple à une classe `Note`, qui serait proche d'un `float`, ou à une classe `Nom`, qui ressemblerait une `std::string` soumise à certaines contraintes (p. ex. : pas de chiffres, n'en déplaie à R2D2 ou à Henri VIII, ou simplement pas vide comme dans l'exemple proposé à droite).

Il est possible, en C++, d'indiquer et de contrôler les règles par lesquelles un type peut être converti en un autre type. L'opérateur `std::string` de `Nom` est une méthode d'instance qualifiée `const` parce qu'elle ne modifie en rien le `Nom` auquel il appartient, et montre comment on peut traiter une instance d'une classe donnée (`Nom`) comme un autre type (`std::string`).

Les conversions (du moins celles qui sont qualifiées `const`) sont une forme particulière d'accesseurs.

```
#include <string>
using std::string;
class Nom {
    string brut;
    static const string NOM_DEFAULT;
    static const string&
        valider(const string &valeur) {
        return valeur.empty()?
            NOM_DEFAULT : valeur;
    }
public:
    Nom(const string &s) : brut{ valider(s) }
    {
    }
    operator string() const {
        return brut;
    }
};
```

Ici, la conversion explicite d'une instance de `Nom` en `std::string` réalisée dans `main()` à droite sollicite, en arrière-plan, l'opérateur de conversion d'un `Nom` en `std::string`.

Petite remarque technique : les opérateurs de conversion doivent nécessairement être implémentés sous forme de méthodes d'instances.

Les opérateurs de transtypage comme `static_cast` sont décrits dans [POOv01], section *Conversions explicites de types*. Il y a un risque à l'utilisation d'opérateurs de transtypage; voir *Opérateurs et pratiques*, plus bas, pour des détails.

```
#include "Nom.h"
#include <string>
#include <iostream>
int main() {
    using namespace std;
    Nom n{"Gontran"};
    cout << static_cast<string>(n);
}
```

Opérateurs et pratiques

Quelques mots supplémentaires sur les opérateurs, leur surcharge et les bonnes pratiques de design comme de programmation. Parfois, les bonnes pratiques impliquent de ne pas réaliser certaines surcharges, ou du moins d'agir avec prudence.

Une saine maxime

Scott Meyers, un expert réputé du langage C++ et de l'approche OO, a lancé⁹⁸ cette saine et sage maxime au sujet de la surcharge d'opérateurs, du moins lorsqu'on l'applique aux types valeur : ***Do as the ints do*** (fais comme les entiers).

Règle générale, c'est sans doute la meilleure maxime dans le cas de la surcharge d'opérateurs. Une bonne surcharge d'opérateur est intuitive, prévisible et a un effet respectant la tradition. Un algorithme opérant sur des entiers devrait fonctionner de manière analogue et sans surprise en manipulant tout autre type pour lequel un ensemble d'opérateurs a été surchargé.

Le compilateur ne peut veiller à assurer le respect de la sémantique fine des opérations, du sens usuel des opérateurs dans leurs déclinaisons surchargées, un peu comme il lui est impossible de s'assurer qu'une fonction `RacineCarre(x)` calcule et retourne bel et bien la racine carrée de `x`. Seule la discipline de programmation et des programmeuses/ des programmeurs peut garantir le respect de la sémantique fine de comportement attendu d'un sous-programme.

Incarnation du mal : la surcharge de l'opérateur & unaire

Surcharger l'opérateur & unaire est possible en C++ mais équivaut *dans l'immense majorité des cas*⁹⁹ à vendre son âme au diable. Il faut comprendre que l'opérateur & unaire sert habituellement à obtenir l'adresse d'un objet. Le surcharger pour quoique ce soit d'autre que l'exemple à droite brise tout code désireux de manipuler des pointeurs sur le type pour lequel cet opérateur a été surchargé.

Le concept d'adresse est l'un des plus fondamentaux qui soient. Jouer avec le sens de cette opération est comme jouer avec des allumettes dans un entrepôt de poudre à canon.

La maxime *fais comme les entiers* est particulièrement recommandable ici. Rares sont les types pour lesquels l'opérateur & unaire a été surchargé et qui demeurent véritablement utilisables.

```
class X {
    // ...
public:
    // ...
    X* operator&() {
        return this;
    }
    // ...
};
```

⁹⁸ Je n'ai malheureusement pas la référence originale; je tiens ceci du très intéressant bouquin [ExcC++], p. 173.

⁹⁹ L'indication *dans l'immense majorité des cas* est importante : il existe ici et là des cas limites pour lesquels cette surcharge est convenable, mais ils sont *extrêmement* rares. Si vous envisagez de surcharger l'opérateur & unaire, posez vous plusieurs fois les questions *est-ce vraiment nécessaire?* et *est-ce vraiment la meilleure option?* avant de procéder... puis posez vous ces questions à nouveau, juste au cas où!

Mise en garde : surcharge de `&&` et de `||`

Une mise en garde s'impose pour ce qui est de la surcharge des opérateurs `&&` et `||`. En effet, il y a des risques très sérieux à implémenter une surcharge de ces opérateurs du fait que les surcharges résultantes sont des méthodes normales, qui ne respecteront pas les raccourcis des opérateurs logiques usuels, comme l'a entre autres fait remarquer *Scott Meyers* (encore lui).

Par exemple, en temps normal, l'expression `a && b` n'évaluera `b` que si `a` est vrai, ce qui est fortement utilisé pour valider des pointeurs avant d'accéder à ce vers quoi ils pointent. Si `&&` est surchargé, par contre, `a` et `b` seront tous deux évalués, quoiqu'il arrive (et on ne sait pas dans quel ordre puisque le standard est neutre en ce sens).

Implémenter une surcharge de ces deux opérateurs change donc le comportement usuel des programmes, peu importe la nature de la surcharge. Ne le faites qu'en connaissance de cause et en documentant très clairement votre œuvre.

Le cas de la méthode `operator bool()`

Il est possible depuis longtemps d'implémenter l'opérateur de conversion en `bool` sur un objet, ce qui le rend effectivement « testable » au sens de vérifiable en tant que condition dans une alternative ou dans une répétitive.

Avant C++ 11, plusieurs hésitaient à rendre leurs classes testables de cette manière, du fait que le `bool` ainsi obtenu pouvait être utilisé à des fins détournées (manipulation de bits, arithmétique ou autre).

Sachez que ce problème est désormais réglé, et qu'implémenter `operator bool()` sur un type est une manière correcte et légitime de faire des instances de ce type des objets testables. Il faut toutefois prendre soin de le qualifier `explicit` pour réduire les cas d'utilisation à ceux qui sont pertinents.

```
class texte {
    char *data = nullptr;
public:
    texte() = default;
    // ... plusieurs trucs ici ...
    // un texte est Ok si son
    // data_ est non-nul
    explicit operator bool() const {
        return data != nullptr;
    }
    // ...
};

texte generer_texte();

int main() {
    texte ze_texte = generer_texte();
    if (ze_texte) { // operator bool
        // ...
    }
}
```

Mise en garde : surcharge des opérateurs de conversion

Les opérateurs de conversion sont utilisés automatiquement et silencieusement par le moteur d'inférence de types de C++ lorsque celui-ci estime en avoir besoin. Il n'était pas possible en C++, avant C++ 11 du moins, de forcer un opérateur de conversion de types à n'être sollicité que de manière explicite. Heureusement, ce problème est chose du passé.

Retenez donc que les conversions implicites de types et les constructeurs à un seul paramètre peuvent jouer des tours aux programmes.

Prenons par exemple la classe `Entier` à droite. Elle expose un constructeur paramétrique à un seul paramètre, de type `int`, et un opérateur de conversion de type (`operator int()`).

Face à l'appel de la fonction `f()` dans `main()`, le compilateur a deux options, aussi valides l'une que l'autre : convertir `i` en `Entier` à l'aide du constructeur paramétrique, et convertir `e` en `int` à l'aide de l'opérateur de conversion de types. L'appel à `f()` est donc jugé ambigu et la compilation échoue.

La solution habituelle à ce problème est (a) de qualifier le constructeur paramétrique d'explicite, (b) de qualifier l'opérateur de conversion d'explicite, ou (c) de qualifier les deux d'explicites.

```
class Entier {
public:
    Entier(int);
    operator int() const;
};

void f(Entier, Entier);
void f(int, int);

int main() {
    Entier e = 3;
    int i = 4;
    f(e, i); // ambigu
}
```

Cependant, le moteur d'inférence de types continuera d'utiliser silencieusement l'opérateur de conversion de types, ce qui peut entraîner des coûts cachés à l'exécution. Utilisez, conséquemment, les opérateurs de conversion de types avec parcimonie, du moins dans leur acception implicite.

Dans d'autres langages

En Java, la surcharge d'opérateurs n'est pas permise bien que son éventuelle insertion semble faire son chemin, la pression étant forte, dans la tête des concepteurs... Peut-être y aurons-nous droit dans la version 1.8 du langage?

Java supporte les opérateurs `==` et `!=` pour comparer des références sur des objets, mais pour comparer le contenu de deux objets il est nécessaire d'utiliser une méthode d'instance, traditionnellement nommée `equals()` et qui prend un `Object` en paramètre.

L'exemple à droite utilise des `String`, mais convient tel quel à toute classe pour laquelle la méthode booléenne `equals()` aura été convenablement codée.

```
public class Z {
    public static void main(String [] args) {
        String s0 = "allo";
        String s1 = "all";
        s1 = s1 + "o";
        if (s0 == s1)
            System.out.println("Pas affiché");
        if (s0.equals (s1))
            System.out.println("Affiché");
    }
}
```

L'opérateur `=` permet quant à lui de copier des références et ne modifie donc pas l'objet servant à titre d'opérande de gauche. Java reposant sur un moteur de collecte d'ordures, les références oubliées sont éventuellement collectées.

Reprenant le cas de `String` ci-dessus, aucune instance de `String` n'est modifiée dans le programme donné en exemple :

- la déclaration de `s0` équivaut à `String s0 = new String("allo");`
- la déclaration de `s1` équivaut à `String s1 = new String("all");`
- la concaténation exprimée par `s1 = s1 + "o"` équivaut directement à l'expression plus complexe `s1 = new String(new StringBuffer("s1").append("o"))` ce qui, clairement, crée de nouveaux objets et fait pointer la référence `s1` vers une chaîne différente que celle vers laquelle elle pointait auparavant¹⁰⁰. La chaîne vers laquelle pointait `s1` avant cette opération flotte ensuite quelque part en mémoire et sera éventuellement ramassée par la collecte automatique d'ordures;
- la comparaison `s0 == s1` vérifie si `s0` et `s1` réfèrent au même objet en mémoire, pas si les objets auxquels réfèrent `s0` et `s1` ont le même contenu. La comparaison de références est prise en charge par le langage lui-même;
- en retour, la comparaison `s0.equals(s1)` compare le contenu de `s0` et de `s1` selon une stratégie préconisée par l'instance active qu'est alors `s0`.

En Java, la manière correcte de vérifier si une chaîne `s` donnée est vide est `if ("".equals(s))` car `""`, en Java, est une `String` (non nulle, alors que `s` pourrait être nulle) et car `equals()` retourne faux si son paramètre est une référence nulle. Les alternatives sont toutes plus complexes ou plus risquées.

¹⁰⁰ Cette notation est presque exacte. La classe `StringBuffer` est simplement une version modifiable de la classe `String` qui, elle, est immuable. Ces deux classes sont, en Java, cousines et on aura recours à `StringBuffer` principalement pour des raisons d'efficacité. Dans les langages .NET, on retrouve précisément le même concept mais implémenté à travers les classes `String` et `StringBuilder`.

En C#, la stratégie est plus nuancée et plus complexe. Un choix mitoyen a été fait¹⁰¹ de permettre la surcharge de certains opérateurs jugés pertinents par les concepteurs du langage.

Le langage C# implémente, comme Java, une stratégie de comparaison de contenu à l'aide d'une méthode de comparaison de contenu nommée `Equals()` et générera un avertissement à la compilation si un type expose `==` sans spécifier le sens de `Equals()`, car ceci pourrait mener à des schèmes de comparaison de contenu incohérents. Évidemment, on peut coder `==` et `Equals()` de manière incohérente à l'insu du compilateur même en programmant ces deux méthodes plutôt qu'une seule.

Autre aberration de C# : coder `Equals()` force par ricochet la programmation de `GetHashCode()`, au sens où deux objets égaux devraient avoir un code de hashage égal; malheureusement, il s'agit d'une autre faute de logique puisque, collisions aidant, deux objets ayant le même code de hashage peuvent être différents, provoquant une autre asymétrie problématique au cœur du langage.

Les opérateurs arithmétiques et logiques binaires (+, %, *, &&, ^, <<, etc.) peuvent être surchargés sous forme de méthodes de classe publiques.

```
using System;
namespace z
{
    public class Entier
    {
        public int Valeur { get; set; }
        public Entier(int val) { Valeur = val; }
        // Pour coder ==, il faut coder aussi !=
        // ... puis Equals() ...
        // ... puis GetHashCode() ...
        public static bool operator==(Entier e0, Entier e1)
        { return e0.Equals(e1); }
        public static bool operator!=(Entier e0, Entier e1)
        { return !e0.Equals(e1); }
        public override bool Equals(Object o)
        { return Valeur == ((Entier)o).Valeur; }
        public override int GetHashCode()
        { return Valeur.GetHashCode(); }
        // l'addition entre Entier et Entier et
        // entre Entier et int
        public static Entier operator+(Entier e0, Entier e1)
        { return new Entier(e0.Valeur + e1.Valeur); }
        public static Entier operator+(int i, Entier e)
        { return new Entier(e.Valeur + i); }
        public static Entier operator+(Entier e, int i)
        { return new Entier(e.Valeur + i); }
    }
}
```

¹⁰¹ Pour une liste des opérateurs possibles et des contraintes pour chacun, je vous invite à consulter le lien suivant : <http://msdn2.microsoft.com/en-US/library/8edha89s.aspx>

Les opérateurs relationnels `==`, `!=`, `<`, `>`, `<=` et `>=`, dans la mesure où les surcharges vont par paires. Par exemple, il est interdit de surcharger `==` si on ne surcharge pas aussi `!=`. Chacun de ces opérateurs doit être fait sous forme de méthode de classe publique; il est illégal de les exposer sous forme de méthode d'instance.

Étrangement, les paires d'opérateurs relationnels ne sont pas logiquement symétriques. Par exemple, qui veut surcharger `<` doit aussi surcharger `>` alors que le véritable inverse logique de cet opérateur est `>=`.

L'opérateur `[]` est supporté à travers une syntaxe spéciale sur laquelle nous reviendrons dans notre discussion des propriétés.

```
// pour coder <, il faut coder >
public static bool operator<(Entier e0, Entier e1)
    { return e0.Valeur < e1. Valeur; }
public static bool operator>(Entier e0, Entier e1)
    { return e0. Valeur > e1. Valeur; }
// conversion explicite d'un Entier en int
public static explicit operator int(Entier e)
    { return e. Valeur; }
// true (oblige qu'on programme false)
public static bool operator true(Entier e)
    { return e. Valeur != 0; }
public static bool operator false(Entier e)
    { return e. Valeur == 0; }
// autoincrément préfixé (le suffixé est
// géré par C#)
public static Entier operator++(Entier e)
    {
        e.Valeur++;
        return e;
    }
}
```

Il est possible de définir des opérateurs implicites (ne requérant pas de conversion explicite de types) et explicites (exigeant une conversion explicite de types) de conversion de type. Ce sont encore une fois obligatoirement des méthodes publiques de classe.

Les opérateurs comme `+=`, `*=`, `%=` ne peuvent être surchargés. Il faut les simuler à l'aide des opérateurs `+`, `*`, `%`... De même, en C#, les autres opérateurs (`=`, `new`, `()`, *etc.*) ne peuvent être surchargés.

On constate que l'affectation sous toutes ses formes est gérée par le langage et ne peut être prise en charge par le programmeur (ce qui, dans un langage où la collecte automatique d'ordures prime, a un certain sens).

En VB.NET, la surcharge d'opérateurs est interdite, et bien qu'un programme VB.NET puisse utiliser des classes C#, les opérateurs surchargés en C# restent inaccessibles à VB.NET.

Cela signifie qu'une entreprise développant dans un langage .NET devrait, du moins si elle ne se limite pas à C#, conserver une version sous forme de méthode plus classique de chacun des opérateurs surchargés ou sujets à l'être.

Exercices – Série 06

Reprenez la classe `Triangle` de *Exercices – Série 05*, et ajoutez-lui les opérateurs de comparaison, d’affectation et d’ordonnement.

Utilisez la surface (l’aire) des triangles comme critère pour examiner si un triangle est plus petit (ou plus grand) qu’un autre.

Pouvez-vous définir un opérateur d’écriture sur un flux en sortie pour `Triangle`? Est-ce qu’un opérateur de lecture sur un flux pour cette classe serait pertinent? Si oui, quelle forme prendrait-il? Sinon, pourquoi?

Justifiez les implémentations sous forme de méthodes ou de fonctions globales des opérateurs que vous implémenterez.

Exercices – Série 07

Ajoutez à `Rectangle` de **Exercices – Série 06** l'opérateur `*=` qui prend en paramètre un entier (strictement positif). Le rôle de cet opérateur est de modifier la hauteur et la largeur en les multipliant par l'entier reçu en paramètre.

Par exemple :

```
#include "Rectangle.h"
int main() {
    Rectangle r(3, 5); // r est un Rectangle 3 x 5
    r *= 2; // la hauteur et la largeur de r doublent
    r.dessiner(); // dessine un Rectangle 6 x 10
}
```

Ajoutez à `Triangle` l'opérateur `*=` qui prend en paramètre un entier (strictement positif). Le rôle de cet opérateur est de modifier la hauteur en la multipliant par l'entier reçu en paramètre.

Ajoutez à `Rectangle` l'opérateur `/=` qui prend en paramètre un entier (strictement positif). Le rôle de cet opérateur est de modifier la hauteur et la largeur du `Rectangle` en les divisant par l'entier reçu en paramètre.

Ajoutez à `Triangle` l'opérateur `/=` qui prend en paramètre un entier (strictement positif). Le rôle de cet opérateur est de modifier la hauteur du `Triangle` en la divisant par l'entier reçu en paramètre.

Plus difficile : ajoutez à `Triangle` l'opérateur `*` qui prend en paramètre un entier (strictement positif). En quoi cet opérateur sera-t-il différent de l'opérateur `*=`?

Réflexions

Quelques questions, pour réfléchir un peu.

Q00 – Le paramètre passé à l'opérateur `*=` peut-il être qualifié `const`? Pourquoi?

Q01 – La méthode qu'est l'opérateur `*=` peut-elle être qualifiée `const`? Pourquoi?

Q02 – Si nous transformons la méthode `operator*=()` en fonction globale, cette fonction pourra-t-elle alors être qualifiée `const`? Pourquoi?

Exercices – Série 08

Rédigez la classe `Point` afin que celle-ci offre au moins :

- deux membres entiers donnant les coordonnées `x` et `y` d'un point;
- des accesseurs et des mutateurs pour ses coordonnées `x` et `y`. Ces coordonnées doivent en tout temps être positives (0 étant considéré positif);
- constructeur par défaut (coordonnées par défaut : `{0, 0}`), un constructeur paramétrique et un constructeur par copie;
- un opérateur d'affectation et les opérateurs de comparaison `==` et `!=`;
- si vous avez envie de vous amuser, essayez aussi les opérateurs `+`, `+=`, `-` et `-=`.

Ajoutez à la classe `Rectangle` un `Point` d'origine, encapsulé derrière les méthodes `GetOrigine()` et `SetOrigine()`.

Ajoutez à la classe `Triangle` un `Point` d'origine, encapsulé derrière les méthodes `GetOrigine()` et `SetOrigine()`.

Traitement d'exceptions

Rôle de cette section

Le traitement d'erreurs et de cas exceptionnels fait partie intégrante de toute solution automatisée. L'approche OO apporte une lueur nouvelle à cette importante thématique.

Il y a des thèmes qui, sans être OO à proprement dit, demeurent incontournables pour toute approche moderne de développement de systèmes informatiques. Parmi ceux-ci, on retrouve le *traitement d'exceptions*.

Pourquoi le traitement d'exceptions?

On utilise les exceptions dans le cas où une erreur risque de survenir lors de l'exécution d'un sous-programme et où ce sous-programme n'est pas le mieux placé pour la gérer.

On a aussi recours au traitement d'exceptions lorsqu'on fait le constat qu'il faudrait *dénaturer* un sous-programme pour qu'il signale correctement une situation exceptionnelle (une *exception*) à son sous-programme appelant.

⇒ On nomme **exception** toute situation qui ne devrait pas se produire en temps normal, mais qui peut néanmoins se produire.

⇒ On représente habituellement une exception par un objet dont le type décrit bien la situation exceptionnelle rencontrée.

⇒ Le **traitement d'exceptions** permet aux programmes de faire pour l'essentiel comme si la situation ne courait pas de risque de se produire, agissant de manière fluide et naturelle, et de réagir convenablement et à l'endroit opportun au moment où, *exceptionnellement*, la situation en question survient.

Le traitement d'exceptions se retrouve maintenant dans presque tous les langages de programmation dignes de ce nom, qu'ils soient OO ou non. Cette thématique naît d'une réflexion quant à *comment* (et où) détecter et traiter les erreurs.

L'approche OO vise une utilisation toujours plus facile de ces entités complètes et opérationnelles qu'on nomme objets. Ce faisant, elle jette un regard neuf et perçant sur la nature-même de l'arrivée au monde, du devenir et de la disparition des entités logicielles.

Le traitement d'exceptions, lui, cherche :

- à *localiser* la détection des erreurs là où il est le plus approprié de la placer;
- à dissocier la localité de la *détection* d'erreurs de celle du *traitement* des erreurs; et
- à rendre le plus naturel possible l'emploi de méthodes (ou de sous-programmes en général) lorsqu'aucun pépin ne se produit, ramenant la gestion d'erreurs à ces cas exceptionnels, et entravant au minimum la fluidité des programmes.

Cette vocation tripartite va de concert avec l'approche objet, et s'inscrit dans une démarche générale de programmation moderne, élégante et efficace.

Cette section est la première de plusieurs, réparties sur l'ensemble des volumes de cette série, portant sur le traitement d'exceptions. Il s'agit d'un sujet important, aux ramifications nombreuses et profondes.

Un exemple très simple (et pas vraiment ☹) de fonction où une exception pourrait s'avérer être de mise serait :

```
int div_ent(int num, int denom) { // division entière
    int quotient;
    // Ligne suivante: si Denom == 0 ... Boum!
    quotient = num / denom;
    return quotient;
}
```

Procéder à une division entière devrait être une tâche fluide dans l'immense majorité des cas. Sur un peu plus de quatre milliards de possibilités¹⁰² pour le dénominateur `denom`, une seule (la valeur 0) risque de faire échouer l'exécution du sous-programme.

Et il est entendu que, *si les sous-programmes appelant `div_ent()` sont bien rédigés, cette situation ne devrait jamais se produire*. Cependant, la vie étant ce qu'elle est, les situations qui ne devraient jamais se produire, parfois, surviennent.

Les stratégies de réaction à l'erreur s'offrant à nous sont les suivantes.

Affichage d'un message d'erreur

Tout d'abord, on pourrait afficher un message d'erreur à même le sous-programme. C'est le réflexe qu'on tend à développer lorsqu'on ne produit que des programmes de type console.

Toutefois, s'il advient que ce sous-programme serve dans une application munie d'une interface personne/machine graphique, un affichage console¹⁰³ serait inapproprié¹⁰⁴.

Autre ennui significatif à cette stratégie: si on affiche un message d'erreur, même dans une application console, *l'utilisateur* saura qu'il y a eu un problème, mais *le programme*, lui, ne le saura pas.

Le dilemme est illustré dans ce cas par la clause `else`, zone ombragée : quelle sera la valeur à mettre dans `Quotient` si une erreur survient? *Tout entier peut être le résultat d'au moins une division entière!* Clairement, cette solution n'est d'aucune utilité systémique.

```
int div_ent(int num, int denom) {
    int quotient;
    if (denom != 0)
        quotient = num / denom;
    else {
        cerr << "ERREUR";
        // Que mettre dans quotient?
    }
    return quotient;
}
```

¹⁰² Si le type `int` est encodé sur 32 bits, bien sûr. Notons que, bien qu'un dénominateur de valeur 0 soit une possibilité sur 2^{32} pour ce paramètre, dans la réalité la probabilité de rencontrer la valeur 0 sur 32 bits est *beaucoup* plus grande que $\frac{1}{2^{32}}$.

¹⁰³ Le problème serait le même si on procédait à un affichage graphique.

¹⁰⁴ Évidemment, dans les grands systèmes, `std::cerr`, l'erreur standard, soit redirigée sur une sortie pertinente même dans les programmes à affichage graphique.

Ajouter un paramètre pour le code de succès ou d'échec

On pourrait choisir de passer un paramètre additionnel qui sera utilisé pour signifier s'il y a eu erreur ou non à l'exécution du sous-programme en question.

On pourrait déclarer ce paramètre comme étant booléen, prenant par exemple la valeur `true` si la division fut un succès et `false` sinon, ou encore comme étant un type énuméré dont `SUCCES` est l'une des valeurs possibles.

On privilégiera la deuxième option si plusieurs cas d'échecs sont possibles et s'il est pertinent de rapporter celui rencontré.

L'ennui de cette stratégie est qu'elle dénature (et complique) le travail à faire. Cette solution fonctionne, du moins pour les fonctions traditionnelles, mais souffre d'inélégance.

```
int div_ent(int num, int denom, bool &succes) {
    int quotient = 0;
    if (denom != 0) {
        quotient = num / denom;
        succes = true;
    } else {
        succes = false;
    }
    return quotient;
}
```

Note: avec cette stratégie, si `succes == false` après l'appel, il ne faut pas utiliser la valeur retournée par la fonction : elle sera invalide.

Le traitement d'erreur devient une préoccupation telle qu'elle altère l'interface (la convention d'appel) du sous-programme.

Ajouter des paramètres ainsi tend à mener les utilisateurs potentiels vers d'autres outils, qui permettent d'accomplir la même tâche, mais de manière plus naturelle.

Réflexion 00.7 – Cette approche est-elle applicable si un problème survient dans un constructeur? (voir *Réflexion 00.7 : erreurs à la construction*).

Ajouter un paramètre pour le résultat et retourner un code d'échec

Dans la même veine, on pourrait faire en sorte que le sous-programme retourne un code d'erreur (ou de succès)¹⁰⁵, et qu'il dépose le quotient (résultat de l'opération à effectuer) dans un 3^e paramètre.

Cette stratégie comporte les mêmes problèmes que la précédente. Elle rend l'utilisation moins confortable, moins naturelle.

Encore une fois, si le code retourné est un d'échec, il ne faut pas utiliser le quotient calculé par le sous-programme, cette variable `Quotient` n'ayant alors pas été remplie correctement.

```
enum class Resultat { SUCCES, ECHEC };
// ...
Resultat div_ent(int num, int denom, int &quotient) {
    Resultat resultat;
    if (denom != 0) {
        quotient = num / denom;
        resultat = Resultat::SUCCES;
    } else {
        resultat = Resultat::ECHEC;
    }
    return resultat;
}
```

Réflexion 00.7 – Cette approche est-elle applicable si un problème survient dans un constructeur? (voir *Réflexion 00.7 : erreurs à la construction*).

Vices du traitement d'erreur traditionnel

La raison pour laquelle chacune de ces solutions possibles comporte des problèmes est que, fondamentalement, traiter ses propres cas exceptionnels ne devrait pas faire partie du mandat normal d'un sous-programme.

Un sous-programme sujet à des erreurs d'exécution devrait *détecter* et *signaler* ces cas exceptionnels. Le sous-programme appelant, qui a une meilleure connaissance du contexte d'utilisation du sous-programme appelé, est normalement mieux équipé pour *gérer* l'exception en question¹⁰⁶. Le traitement des exceptions est une acceptation de la réalité que *celui qui détecte un problème n'est habituellement pas le mieux placé pour adresser le problème*.

¹⁰⁵ L'exemple utilise un `enum` par souci d'élégance, mais pourrait tout aussi bien utiliser un `int` et des constantes entières munies de valeurs explicites pour en arriver au même résultat.

¹⁰⁶ S'il n'est pas assez maître du contexte, il peut toujours lui-même relever l'exception en question (ou la laisser filtrer) pour en informer son propre sous-programme appelant, et ainsi de suite.

Finalement, pour que l'élégance naturelle à l'utilisation d'un sous-programme ne soit pas entravée, il importe que le support du traitement d'exceptions ne change en rien le prototype d'un sous-programme. *Les cas normaux ne devraient pas avoir à payer pour les cas atypiques*, exceptionnels, sinon, il s'agirait d'une démarche inutilement coûteuse et contre-productive.

Utiliser le traitement d'exceptions

Nous illustrerons l'emploi d'exceptions, avec le petit programme proposé à droite. Ce programme lit un numérateur et un dénominateur, calcule la division entière correspondante, et affiche le résultat de cette division. Il fonctionne très bien dans la mesure où personne n'a l'étrange idée de fournir un dénominateur de valeur zéro.

Notre tâche sera ici de le compléter en le protégeant de cette ignoble *exception*, tout en prenant soin de ne pas altérer l'interface de notre sous-programme (de ne pas le dénaturer inutilement).

Nous incluons tout d'abord la bibliothèque standard `<exception>`. Celle-ci contient les détails pertinents à l'utilisation de plusieurs exceptions standards C++.

Note importante

Le support aux exceptions de C++ est, à la base, celui du langage C. En langage C, on peut relever des cas d'exception avec à peu près n'importe quel type de données, quoique pas avec des objets, le langage C n'étant pas OO. Il est raisonnable d'utiliser, en C++, `std::exception`, même si on pourrait utiliser des `int` pour relever des exceptions. Cela rendra votre démarche plus réutilisable, étant plus proche de celles préconisées dans d'autres langages OO comme Java et C#. D'autres stratégies, plus intéressantes encore, seront examinées plus loin.

Notre programme est maintenant tel que visible à droite.

```
#include <iostream>
int div_ent(int, int);
int main() {
    using namespace std;
    int numer,
        denom;
    if (cin >> numer >> denom) {
        int res = div_ent(numer, denom);
        cout << res;
    }
}
int div_ent(int num, int denom) {
    int quotient = num / denom;
    return quotient;
}
```

```
#include <iostream>
#include <exception>
using std::exception;
int div_ent(int, int);
int main() {
    using namespace std;
    int numer,
        denom;
    if (cin >> numer >> denom) {
        int res = div_ent(numer, denom);
        cout << res;
    }
}
int div_ent(int num, int denom) {
    int quotient = num / denom;
    return quotient;
}
```

Reconnaître et lever une exception

Dans un traitement d'exceptions, c'est le sous-programme appelé qui est tenu responsable de *détecter* les cas d'exception. C'est lui le mieux placé pour les détecter, étant celui qui les rencontre effectivement.

Dépister les cas d'exception potentiels demande un peu d'analyse de l'équipe de développement, le même genre d'analyse qu'on demandera de toute façon pour tout traitement d'erreur.

Lorsque le sous-programme rencontrera un cas d'exception, son travail sera de le signaler au sous-programme l'ayant appelé. On dira alors que le sous-programme appelé *lève* (ou *lance*) une exception au sous-programme appelant.

⇒ On lève une exception à l'aide de l'instruction **throw** (certains langages, par exemple Ada, utiliseront *raise*)

Remarquez la syntaxe : *on lance une exception*, à laquelle on joint un bout de texte descriptif, le *message* de l'exception. On dit aussi *lever une exception* (de l'anglais *to raise an exception*)

```
#include <iostream>
#include <exception>
using std::exception;
int div_ent(int, int);
int main() {
    using namespace std;
    int numer,
        denom;
    if(cin >> numer >> denom) {
        int res = div_ent(numer, denom);
        cout << res;
    }
}
int div_ent(int num, int denom) {
    if (denom == 0)
        throw exception{ "Div. par zéro" };
    int quotient = num / denom;
    return quotient;
}
```

Note : certains langages, comme Java, demandent à tout sous-programme susceptible de lancer une exception de l'indiquer clairement, apposant un suffixe tel que `throws Exception`, et forcent les appelants à prendre cette notice en considération.

En langage C++, lever ou non une exception est un point d'implémentation, et un appelant n'est pas *forcé* d'attraper les exceptions lancées par l'appelé. Une mention `throw()` existe pour offrir des garanties de comportement de la part de sous-programmes, mais nous l'escamoterons pour le moment.

Gérer une exception

Le sous-programme appelant est responsable de capter l'exception et d'y réagir.

En langage Java, le sous-programme appelant *doit* mettre en place une procédure de traitement des exceptions, sinon il ne pourra être compilé. En C, C++ ou C#, le choix de mettre (ou non) sur pied une procédure de traitement des exceptions relève de l'équipe développant le sous-programme appelant. *Rien ne les y oblige.*

Si un sous-programme capte une exception mais n'est pas en mesure d'y réagir correctement, il peut la lancer au sous-programme l'ayant lui-même appelé, qui peut la lancer lui aussi, et ainsi de suite jusqu'à ce que quelqu'un finisse par la traiter ... ou que le programme échoue à cause d'une exception non captée (*Unhandled Exception*).

Appeler un sous-programme susceptible de lancer une exception est toujours un *essai*, puisqu'on ne sait pas si le sous-programme va compléter normalement ou s'il va lancer une exception avant de se terminer. Aussi faut-il envelopper l'appel dans ce qu'on appelle un bloc **try**.

Tout bloc **try** *doit* être suivi d'un bloc **catch**, qui sert à *attraper* (s'il y a lieu) l'exception lancée par le sous-programme appelé. C'est dans le bloc **catch** qu'on effectuera le traitement d'erreur.

```
#include <iostream>
#include <exception>
using std::exception;
int div_ent(int, int);
int main() {
    using namespace std;
    int numer,
        denom;
    if(cin >> numer >> denom)
        try {
            int res = div_ent(numer, denom);
            cout << res;
        } catch(exception e) {
            cerr << e.what();
        }
}
int div_ent(int num, int denom) {
    if (denom == 0)
        throw exception{ "Div. par zéro" };
    int quotient = num / denom;
    return quotient;
}
```

Note : pour attraper une exception sans s'occuper de son message, vous pouvez omettre le nom de la variable représentant l'exception plutôt que d'en faire un usage bidon : remplacer `catch(exception e)` par `catch(exception)` ou (mieux encore) par `catch(exception&)` dans la mesure où la variable `e` n'est pas utilisée dans le bloc `catch`.

La mécanique (en détail)

Un sous-programme reconnaissant un cas d'exception le lancera. Observez le sous-programme ci-dessous.

Dans le cas où `denom` vaut zéro, menant à la détection d'une exception, les lignes du sous-programme qui seront exécutées seront, dans l'ordre :

- la ligne (0) où on fait la vérification de la valeur de `denom`; et
- la ligne (1) qui lance l'exception ayant le message "Div. par zéro".

```
int div_ent(int num, int denom) {  
    if (denom == 0) // (0)  
        throw exception("Div. par zéro"); // (1)  
    int quotient = num / denom; // (2)  
    return quotient; // (3)  
}
```

Lancer une exception termine immédiatement l'exécution du sous-programme. Dans ce cas-ci, le `return` ne sera pas atteint. Il s'agit là d'un détail important: il n'est pas nécessaire de déposer une valeur bidon dans `quotient` puisque, *si une exception est levée, cette variable ne sera jamais retournée.*

Par contre, si le dénominateur diffère de zéro, les lignes du sous-programme qui seront exécutées seront, dans l'ordre :

- la ligne (0) (vérification de la valeur de `denom`);
- la ligne (2) qui procède, en toute sécurité, à la division demandée; et
- la ligne (3), qui retourne le `quotient` calculé.

Le cheminement normal des opérations, donc, ce qui tombe sous le sens. Au prix d'une alternative (un `if`), on obtient un sous-programme protégé contre la totalité des cas exceptionnels d'erreur qu'il risque de rencontrer.

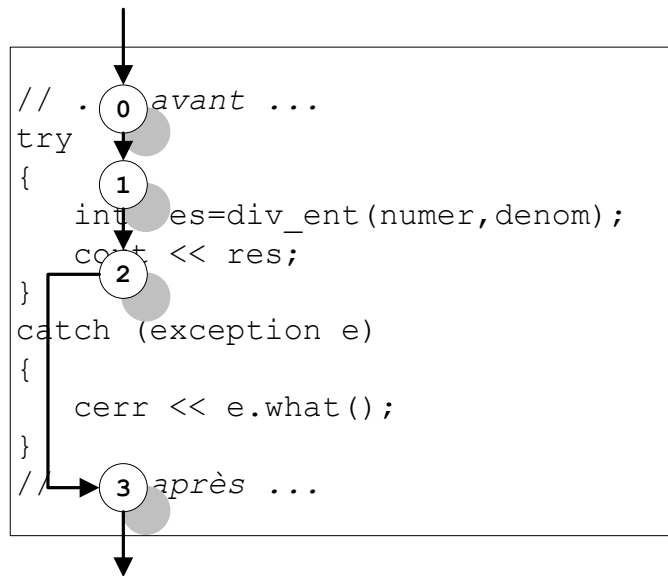
Maintenant, si on porte attention à l'appel¹⁰⁷, on voit ce qui suit (ci-dessous). Que `div_ent()` lance une exception ou non, les lignes suivantes seront exécutées :

- la ligne (0), début du bloc `try`; et
- la ligne (1), appel du sous-programme.

Il faut faire attention à la ligne (1), qui est en deux volets : l'appel du sous-programme, et l'affectation de sa valeur de retour à la variable `res`.

L'appel se fera en tout temps, mais l'affectation de sa valeur de retour, elle, ne se fera *que* si aucune exception n'est levée par le sous-programme.

Si aucune exception n'est levée, alors le sous-programme s'est exécuté normalement et a complété en retournant son résultat.



Dans ce cas, l'exécution se poursuivra à la ligne (2), qui affiche le résultat calculé, puis à la ligne identifiée par (... après ...) qui suit immédiatement le bloc `catch`.

En effet, si aucune exception n'est levée, le bloc `catch` est escamoté. C'est raisonnable, puisque ce bloc sert au traitement des cas d'exception : dans un tel cas, il n'y en a pas.

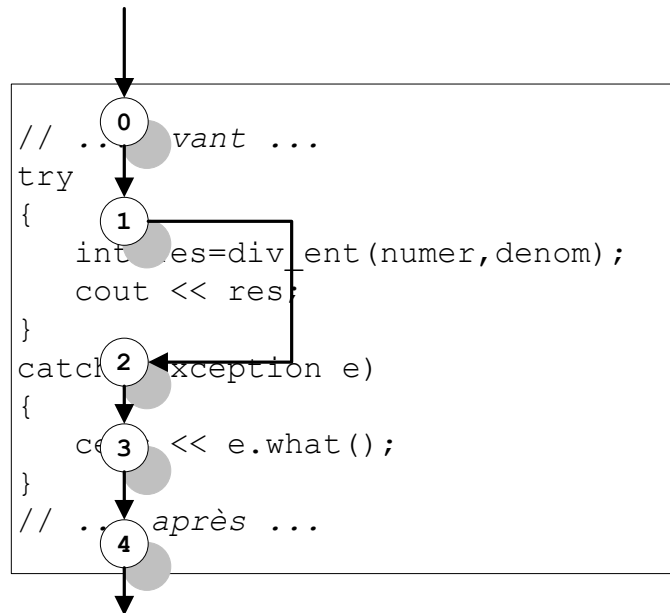
¹⁰⁷ Pour abrégier, seule la section portant sur l'appel du sous-programme a été conservée.

Si une exception est levée, par contre, l'appel à `div_ent()` échouera. Puisque l'exécution de `div_ent()` n'a pas été complétée, et que ce sous-programme n'a pas atteint l'opération `return`, on ne pourra compter sur la valeur de la variable `resultat`.

L'instruction à la ligne (1) ne sera donc pas complétée. La prochaine opération exécutée sera celle à la ligne (2), où commence le traitement d'erreur associé à l'exception levée.

Le mot clé `catch` est suivi de parenthèses à l'intérieur desquelles on voit `exception`, puis un nom de variable (ici : `e`). Cela signifie que l'exception lancée par le sous-programme appelé sera nommée `e`.

La variable nommée `e` est déclarée localement au bloc `catch`. Les accolades de ce bloc en délimitent totalement la portée.



Le traitement d'erreur est réalisé à la ligne (3). Notre exemple présente un traitement d'erreur des plus simples, qui consiste à afficher le message de l'exception ayant été attrapée¹⁰⁸. Une fois le bloc `catch` terminé, l'exécution se poursuivra à la ligne (4).

⇒ La méthode `what()` d'une `exception` retourne une chaîne de caractères contenant le *message* de l'exception¹⁰⁹.

Question : selon vous, après la fin du bloc `catch`, est-il sécuritaire d'utiliser la variable `res`? Expliquez votre position.

¹⁰⁸ Notre programme est un programme console, et choisit d'afficher un message d'erreur à l'écran. Il peut agir ainsi puisqu'il a une connaissance du contexte, connaissance qui échappe à `Div()`.

¹⁰⁹ Avec Java, on parle de la méthode `getMessage()` de la classe `Exception`. Sous C#, on parle de la propriété `Message`.

Lorsque le traitement d'exception est obligatoire : une technique

Tel qu'indiqué précédemment, certains langages *obligent* tout sous-programme appelant d'un sous-programme susceptible de lancer une exception (par exemple, en Java, toute méthode ayant une mention `throws`) à mettre en place une combinaison `try...catch` au cas où l'exception surviendrait.

Utiliser un couple `try...catch` à chaque appel d'un mutateur, pour prendre l'exemple le plus commun, peut alors devenir désagréable et fastidieux. Être obligé d'avoir recours au traitement d'exception à l'appel d'un mutateur par un constructeur par défaut est, en effet, un peu abusif. Les gens finissent souvent par ne pas utiliser de traitement d'exception, pour alléger le code, et perdent alors un outil privilégié de gestion des erreurs à l'exécution, ou encore finissent par négliger l'encapsulation et compliquer la gestion de l'évolution de leur code.

Une solution à ce problème, qui peut être appliquée dans d'autres langages (incluant bien sûr C++) et d'*utiliser deux méthodes plutôt qu'une seule* pour chaque cas où on peut avoir des appels garantis comme sécurisés et d'autres réclamant une forme de validation.

Prenons le cas de `SetHauteur()` pour la classe `Rectangle`, et présumons que les contraintes de validité d'une hauteur, quelles qu'elles soient, soient gérées par la méthode `est_hauteur_valide()`.

Ainsi, sans traitement d'exceptions, on aurait probablement quelque chose comme l'extrait proposé à droite.

```
void Rectangle::SetHauteur(int hau) {
    if (est_hauteur_valide(hau))
        hauteur_ = hau;
}
```

Le principal défaut de cet extrait, bien sûr, est que l'appelant de `SetHauteur()` ne sera jamais informé de la réussite ou de l'échec de sa tentative de modification de la hauteur du `Rectangle` propriétaire de cette méthode.

Les échecs silencieux lors d'invocations de méthodes, nous le savons, empêchent les systèmes de se diagnostiquer. Ce n'est donc pas, en pratique, une approche suffisante.

En ajoutant le traitement d'exceptions brut, on arrive à ceci (à droite).

```
void Rectangle::SetHauteur(int hau) {
    if (!est_hauteur_valide(hau))
        throw exception("Hauteur invalide");
    hauteur_ = hau;
}
```

En C++, c'est souvent suffisant. Si cette classe était implantée en Java, par contre, tout appel à `SetHauteur()` nécessiterait maintenant un couple `try...catch`.

Une solution intéressante est d'ajouter à la classe un mutateur privé, modifiant *sans validation* la valeur de l'attribut.

Dans l'exemple à droite, ce mutateur est nommé `SetHauteurBrut()`.

```
void Rectangle::SetHauteurBrut(int hau) {
    hauteur_ = hau;
}
```

Étant privée, cette méthode n'est accessible qu'à la classe `Rectangle` et aux instances de cette classe, ce qui restreint les possibilités d'appel à cette méthode à des intervenants qu'on est en droit de juger sécuritaires.

La méthode publique, `SetHauteur()`, profitera alors de `SetHauteurBrut()`, dans le but clair de limiter les lieux de modification potentielle de la hauteur du `Rectangle`.

```
void Rectangle::SetHauteur(int hau) {  
    if (!est_hauteur_valide(hau))  
        throw exception("Hauteur invalide");  
    SetHauteurBrut(hau);  
}
```

Tout accès externe à un `Rectangle` et voulant modifier sa hauteur devra alors passer par `SetHauteur()` (ou par une autre méthode publique, qui passera alors par `SetHauteur()`, ce qui est équivalent), alors qu'à l'interne, lorsque la hauteur candidate sera assurément correcte au préalable, on se permettra de passer par `SetHauteurBrut()`.

Cette technique est utile à plusieurs égards. Là où la surface publique d'un objet doit être blindée parce qu'elle est sujette à être utilisée par des tiers *a priori* inconnus, les méthodes privées ne sont accessibles que de l'intérieur, et ne peuvent donc être invoquées que de manière contrôlée (et, en théorie, sécuritaire).

La surface publique d'une classe est donc habituellement composée de méthodes validant systématiquement et rigoureusement leurs intrants, donc qui sont souvent plus lentes mais très sécuritaires, alors que les méthodes privées sont habituellement rapides, escamotant la validation et allant à l'essentiel. Évidemment, les méthodes privées sont utilisées par les méthodes publiques suite à la validation des intrants.

Signaler ses intentions en C++

Le langage C++ n'oblige en rien l'appelant d'un sous-programme susceptible de lever des exceptions à attraper toutes les exceptions pouvant en être lancées. Dans la philosophie C++, l'appelant est responsable de ses actes et *peut* payer le prix¹¹⁰ de la protection associée à des blocs `try...catch`, mais n'a pas l'*obligation* de le faire.

Selon la tradition C, qu'un sous-programme n'annonce rien de particulier quant aux exceptions possibles lorsqu'on l'invoque signifie non pas qu'il ne lèvera pas d'exceptions, mais plutôt qu'il ne s'engage à rien et peut lever n'importe quelle exception s'il le juge pertinent.

Toutefois, C++ permet aux sous-programmes levant des exceptions d'offrir certaines garanties aux sous-programmes qui les appelleront. Ces garanties sont de deux ordres :

- un sous-programme peut garantir qu'il ne lancera pas d'exceptions, quoiqu'il arrive; et
- si un sous-programme lance des exceptions, il peut garantir aux sous-programmes appelants que la liste d'exceptions possibles est finie et indiquer le type de toute exception pouvant être lancée par ce sous-programme (incluant celles potentiellement lancées par tous les sous-programmes qui peuvent y être appelés, récursivement).

La stratégie est simple, dans un cas comme dans l'autre, mais les avis sont partagés sur cette pratique (voir plus bas). Dans un langage OO, en particulier si ce langage est aussi polyvalent que C++, il est plus délicat qu'il n'y paraît d'être aussi strict.

La garantie no-throw

Un sous-programme peut indiquer qu'il ne lancera pas d'exceptions en ajoutant la mention `noexcept` à la fin de sa signature, dans son prototype comme dans sa définition.

Cela ne changera rien au fonctionnement de l'appelant ou de l'appelé.

La clause `noexcept` signifie en fait non pas que la fonction ne lèvera pas d'exceptions mais bien qu'il ne sert à rien de les attraper : si une fonction qualifiée `noexcept` lève une exception, il s'agit d'une situation tellement grave que le programme s'arrêtera, tout simplement.

```
int carre(int) noexcept;
class HautesMathematiques {
public:
    int cube(int nb) const noexcept {
        return carre(nb) * nb;
    }
};
int carre(int nb) noexcept {
    return nb * nb;
}
```

Si votre compilateur ne supporte pas au moins C++ 11, il est possible d'exprimer `throw()` (les parenthèses étant vides) pour signaler qu'une fonction ne lèvera pas d'exception, mais sachez que cette écriture est dépréciée; nous expliquerons pourquoi dans la section *Spécifications d'exceptions*, plus bas.

¹¹⁰ Ce prix est relativement élevé en fait de temps d'exécution. Pour certains programmes, par exemple ceux soumis à des contraintes de temps réel, le coût supplémentaire associé à la gestion des exceptions est parfois inacceptable.

Spécifications d'exceptions

En C++ 03, il était aussi possible avec ce qu'on nommait des **spécifications d'exceptions** de lister les types des exceptions susceptibles d'être levées dans un sous-programme et dans tous ceux auxquels il a affaire, en les séparant par des virgules. Cette pratique est dépréciée depuis C++ 11, la démonstration ayant été faite que cette pratique n'offrait pas de réels avantages en pratique (outre pour la spécification *no-throw*, autrefois exprimée sous la forme `throw()` et, depuis C++ 11, sous la forme `noexcept`).

Il est de bonne guerre de *documenter* les sous-programmes quant aux exceptions qu'ils sont susceptibles de lever, mais le compilateur ne peut pas réellement en tirer profit; ces mentions sont surtout utiles pour les programmeurs, et ont leur place dans les commentaires. Les spécifications d'exceptions étaient une intéressante idée sur le plan documentaire, mais elles ont des limites en C++, et ce pour plusieurs raisons.

Tout d'abord, les spécifications d'exceptions et les clauses *no-throw* décrivent les sous-programmes, les documentent, mais ne font pas partie de la signature des sous-programmes au sens strict comme le font par exemple les mentions `const` des méthodes d'instance. À ce titre, deux fonctions qui n'ont de distinct que leurs spécifications d'exceptions ne sont pas considérées distinctes l'une de l'autre par le compilateur (donc `void f();` et `void f() throw(A);` ne sont pas deux fonctions distinctes).

Ensuite, du fait que toute fonction n'offrant pas de spécifications d'exceptions peut lever n'importe quoi, les compilateurs ne peuvent pas vraiment se fier sur les spécifications d'exceptions lors de la génération du code.

```
int f();
int g() throw() { // est-ce vrai?
    return f();
}
```

Que se produira-t-il si une fonction ment à l'aide de ses spécifications d'exceptions? Par exemple, dans l'illustration ci-dessus, `g()` déclare ne rien lever, mais invoque `f()` qui, elle, pourrait lever quelque chose (on ne le sait pas). Pour des raisons historiques, ce code est légal à la compilation (sans quoi, toutes les fonctions C deviendraient inaccessibles aux fonctions munies de spécifications d'exceptions, ce qui réduirait fortement leur utilité), ce qui reporte la détection de la spécification erronée au moment de l'exécution du programme.

Que peut faire un compilateur dans de telles circonstances? Il se doit, en fait, de prendre des dispositions pour réduire l'ampleur des dégâts. En pratique, cela signifie que :

- du code doit être mis en place pour attraper les exceptions autres que celles n'ayant pas été annoncées (voir *Catch Any et Re-Throw*, plus bas), ce qui peut ralentir l'exécution du programme (mais cela variera selon la qualité des implémentations); puis
- la levée d'une exception non annoncée dénote un problème grave dans la sémantique du programme, qui doit se terminer brusquement (appel à la fonction `std::unexpected()` qui sert précisément à cette fin); ainsi
- les spécifications d'exceptions documentaient donc le contrat moral entre fonction appelante et fonction appelée, mais n'offraient pas de véritables garanties pouvant être validées de manière statique, malgré les apparences.

Clause *noexcept* en tant qu'opérateur

Il est possible d'écrire des fonctions qui sont conditionnellement `noexcept`. Ceci est particulièrement important dans la rédaction de code générique [POOv02]. Prenons à titre d'exemple le cas suivant :

```
int f(int n) noexcept {
    return g(n) + h(n);
}
```

Ici, le programmeur de la fonction `f()` affirme que `f()` ne lèvera pas d'exception, et cette affirmation est inconditionnelle. Pourtant, `f()` appelle `g()` et `h()`; qu'advient-il si l'un ou l'autre de ces fonctions lève une exception pour une valeur de `n` donnée?

En fait, si la promesse que `f()` a faite de ne pas lever d'exception est brisée, alors l'exécution du programme est terminée, et il n'est même pas garanti que les destructeurs des objets globaux seront appelés.

Si `f()` souhaite faire en sorte que sa qualification `noexcept` dépende de celles de `g(n)` et de `h(n)`, il est possible d'écrire ceci :

```
int f(int n) noexcept(g(n) + h(n)) {
    return g(n) + h(n);
}
```

Ainsi, la garantie offerte par `f(n)` sera aussi forte que celle des expressions dont elle dépend, sans plus.

Catch Any et Re-Throw

Il est possible d'attraper n'importe quoi, sans égard à la nature de ce qui a été levé, à l'aide d'un `catch(...)`. Ceci permet de réagir à une exception mais ne permet pas d'en connaître la nature exacte, mais le mécanisme est parfois utile.

On peut aussi lever une exception tout juste attrapée pour que le sous-programme appelant puisse la gérer. Ceci va souvent de concert avec un `catch(...)` : un sous-programme attrape quelque chose d'inattendu (pour lui), fait un peu de nettoyage, et lève la même exception (sans la connaître) pour donner une chance à d'autres de faire un traitement plus pointu. On nomme une telle opération un *re-throw*, et la syntaxe pour y arriver est `throw;` (sans parenthèses).

Le document [POOv01] va beaucoup plus en détail sur le mécanisme d'exceptions, en exploitant des concepts et des techniques que nous n'avons pas encore couvertes à ce stade-ci.

Dans d'autres langages

Pour concevoir des exceptions en Java, C# et VB.NET, il faut avoir couvert la notion d'héritage. Nous y reviendrons donc dans [POOv01]. Cela dit, dans ces trois langages, lever et gérer des exceptions est possible, et ce dans des termes semblables (souvent même identiques) à ceux vus plus haut pour C++.

En Java, le code susceptible de lever une exception *doit* être enrobé dans un bloc `try`¹¹¹. La gestion de chaque exception levée dans un bloc `try` est faite dans un bloc `catch` propre à cette exception.

On peut demander à une exception son message en invoquant sa méthode `getMessage()`.

Un bloc `finally` *peut* suivre la séquence de blocs `catch`. Ce bloc n'existe pas en C++ (et n'y serait pas nécessaire de toute manière).

```
public class Z {
    public static void main(String [] args) {
        int [] tab = new int[10];
        try {
            for(int i = 0; i <= tab.length; i++) {
                System.out.println("tab["+i+"]: "+tab[i]);
            }
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.err.println("Hors bornes " +
                ex.getMessage());
        } finally {
            System.out.println("Ceci s'affichera");
        }
    }
}
```

Le bloc `finally` sera toujours atteint, qu'une exception soit levée ou non, et sert habituellement de lieu pour nettoyer le code mis en place dans le bloc `try` (fermer des fichiers ou des connexions à des bases de données, par exemple). Sans destructeur déterministe, un langage comme Java doit s'en remettre au code client pour compléter le volet de l'encapsulation échappant aux objets.

Une méthode Java susceptible de lever une exception doit l'indiquer en ajoutant le suffixe `throws` suivi des types d'exception possibles (séparés par des virgules).

Si une méthode Java ne porte pas le suffixe `throws`, alors elle ne peut pas lever d'exceptions (ce privilège lui sera refusé par le compilateur).

```
class Test {
    void afficher(String s) throws NullPointerException {
        if (s == null) {
            throw new NullPointerException ();
        }
        System.out.println(s);
    }
}
```

Dans la méthode, l'exception est levée par `throw`. Étant un objet Java, l'exception est créée par `new` et ramassée par la collecte d'ordures. En C++, on préférera lever les exceptions par valeur plutôt qu'à travers de l'allocation dynamique. Les outils influencent les pratiques.

¹¹¹ Cette affirmation est insuffisamment nuancée puisqu'il existe deux familles d'exceptions en Java et que l'obligation de prudence ne s'applique que pour l'une des deux. Cela dit, l'illustration fait son travail.

En C#, on trouve les mêmes mots clés qu'en Java pour la gestion d'exceptions et ce, pour les mêmes usages.

De manière générale, C# est moins strict que Java pour la gestion des exceptions. Si une méthode est susceptible de lever une exception, il est possible mais pas obligatoire d'en enrober l'invocation d'une paire de blocs `try` et `catch`.

On peut obtenir le message d'une exception par sa propriété `Message`.

```
namespace z
{
    public class Z
    {
        public static void Main(string [] args)
        {
            int [] tab = new int[10];
            try
            {
                for(int i = 0; i <= tab.Length; i++)
                    System.Console.WriteLine("tab["+i+"]: "+tab[i]);
            }
            catch (IndexOutOfRangeException ex)
            {
                System.Console.WriteLine("Hors bornes "+ex.Message);
            }
            finally
            {
                System.Console.WriteLine("Ceci s'affichera");
            }
        }
    }
}
```

C# ne permet pas d'annoncer les exceptions qui seront levées par une méthode, et ne contient donc pas d'équivalent de la clause `throws` de Java. Outre ce détail, et ne considérant pas les variantes taxonomiques locales (noms des types d'exception standard), la mécanique de C# pour signaler un cas d'exception est précisément la même que celle de Java.

```
class Test
{
    void Afficher(string s)
    {
        if (s == null)
            throw new NullReferenceException();
        System.Console.WriteLine(s);
    }
}
```

En VB.NET, la situation est semblable malgré une forme syntaxique générale plus proche des coutumes VB. Dans ce cas-ci, ces règles donnent un code légèrement plus compact, ce qui est plutôt amusant.

Remarquez, dans l'exemple, le recours à la concaténation de valeurs à des chaînes de caractères à l'aide de l'opérateur &.

```
Namespace z
  Public Class Z
    Public Shared Sub Main()
      Dim tab() As Integer = New Integer(10 - 1) {}
      Try
        For i As Integer = 0 To Tab.Length
          System.Console.WriteLine("tab(" & i & "): " & tab(i))
        Next
      Catch ex As IndexOutOfRangeException
        System.Console.WriteLine("Hors bornes." & ex.Message)
      Finally
        System.Console.WriteLine("Ceci s'affichera")
      End Try
    End Sub
  End Class
End Namespace
```

Évidemment, la mécanique de VB.NET est semblable à celle de C#.

Remarquez que la constante du système pour signifier une référence illégale est `Nothing` plutôt que `null` et que la comparaison vérifiant qu'une référence soit illégale se fait avec l'opérateur `Is`.

```
Public Class Test
  Public Sub Afficher(ByVal s As String)
    If (s Is Nothing) Then
      Throw New NullReferenceException
    End If
    System.Console.WriteLine(s)
  End Sub
End Class
```

Pour une réflexion plus en profondeur sur les exceptions, voir [hdExcep].

Exercices – Série 09

EX00 – Implantez, s’il y a lieu, le traitement d’exceptions pour les méthodes des classes `Triangle`, `Rectangle` et `Point`, développées dans les séries d’exercices précédentes. *Justifiez chacun de vos choix.*

EX01 – On veut pouvoir diminuer la taille d’un triangle à l’aide de l’opérateur `/=`, qui prend en paramètre un entier. Expliquez où et comment vous gérerez les erreurs potentielles liées à la division de la taille courante du `Triangle`. Y a-t-il d’autres risques d’erreur ici que ceux liés à une division par zéro de la taille du `Triangle`? Si oui, indiquez lesquels, et expliquez comment ils seront gérés.

EX02 – Implantez, si ce n’est fait, la classe `Note`, qui permet de représenter une note entre 0 et 100 inclusivement. Protégez-la rigoureusement contre les intrants invalides. Expliquez pourquoi et en quoi votre solution protège intégralement toute instance de `Note` contre la corruption de ses attributs.

Réflexions

Quelques questions, pour réfléchir un peu.

Q00 – Que se produira-t-il, selon vous, si une exception se produit lors de la construction d’un objet? Y aura-t-il une différence si la construction est automatique ou si elle est dynamique (faite via un appel à `new`)?

Q01 – Que se produira-t-il, selon vous, si une exception se produit lors de la destruction d’un objet? Y aura-t-il une différence si la destruction est automatique ou si elle est dynamique (faite via un appel à `delete`)?

Q02 – Un constructeur par défaut devrait-il être en droit d’escamoter un `try...catch` lorsqu’il appelle un mutateur pour initialiser un de ses attributs? Pourquoi?

Q03 – Un constructeur paramétrique devrait-il être en droit d’escamoter un `try...catch` lorsqu’il appelle un mutateur pour initialiser un de ses attributs? Pourquoi?

Q04 – Un constructeur par copie devrait-il être en droit d’escamoter un `try...catch` lorsqu’il appelle un mutateur pour initialiser un de ses attributs? Pourquoi?

<p>Note : quand nous aurons couvert les mécanismes d’héritage et de polymorphisme, des détails seront donnés quant aux effets d’une exception dans un destructeur ou dans un constructeur. Vous pourrez alors comparer le fruit de vos réflexions avec la mécanique effectivement en place.</p>
--

Exceptions et classes : portrait plus riche

Examinons maintenant comment on pourrait créer un type complet, le type **TableauEntiers**, qui utiliserait le concept d'exception à son avantage. Nous profiterons de cet examen pour montrer plus en détail comment les exceptions s'intègrent au langage C++.

Notre classe allouera un tableau d'entiers dont la taille sera connue à la construction, et libérera ce tableau lors de sa destruction.

Nous ne montrerons pas comment créer une classe tableau raffinée, mais vous pouvez tout à fait extrapoler sur ce qui est présenté ici et compléter le portrait. Les *templates* [POOv02] pourront alors vous être d'un grand secours.

```
class TableauEntiers {
    int *tab_;
    int taille_;
public:
    // bloquer la copie
    TableauEntiers(const TableauEntiers&) = delete;
    TableauEntiers&
        operator=(const TableauEntiers&) = delete;
    int taille() const {
        return taille_;
    }
    int operator[](int n) const {
        return tab_[n];
    }
    int& operator[](int n) {
        return tab_[n];
    }
    explicit TableauEntiers(int n)
        : tab_{new int[n]}, taille_{n} {
    }
    ~TableauEntiers() {
        delete [] tab_;
    }
};
```

La classe élémentaire pourrait être telle que proposée à droite. On remarque :

- un constructeur paramétrique;
- un destructeur;
- une méthode permettant de connaître la taille du tableau; et
- des opérateurs permettant respectivement un accès en lecture et un accès en écriture sur un élément du tableau.

Il aurait été pertinent d'ajouter un à cette classe un constructeur de copie et un opérateur d'affectation. Signe qu'ils auraient été utiles : nous avons dû implémenter le troisième membre de la Sainte-Trinité, le destructeur. Pour que l'exemple demeure simple j'ai simplement bloqué la copie.

En exercice, je vous invite par contre à les implémenter ces deux opérations et à les tester rigoureusement.

Sachant ceci, demandons-nous quels sont les cas d'exception possibles lors d'opérations sur une instance de **TableauEntiers**?

En premier lieu, identifions les cas de méthodes pour lesquelles une clause *no-throw* serait indiquée. Souvenons-nous que la garantie *no-throw* signifie qu'une méthode ne lèvera aucune exception ou que l'exception levée, s'il y a lieu, sera si sérieuse que le programme se trouvera alors dans un état irrécupérable.

```
class TableauEntiers {
    int *tab_;
    int taille_;
public:
    // bloquer la copie
    TableauEntiers(const TableauEntiers&) = delete;
    TableauEntiers&
        operator=(const TableauEntiers&) = delete;
    int taille() const noexcept {
        return taille_;
    }
    int operator[](int n) const {
        return tab_[n];
    }
    int& operator[](int n) {
        return tab_[n];
    }
    explicit TableauEntiers(int n)
        : tab_{ new int[n] }, taille_{ n } {
    }
    ~TableauEntiers() { // noexcept implicite
        delete [] tab_;
    }
};
```

Dans cette classe, deux méthodes sont candidates à une clause *no-throw* : le destructeur et l'accessor `taille()`.

Dans le cas de `taille()`, rien de mal ne peut se produire à l'invocation de la méthode. Elle ne reçoit aucun intrant, ne gère aucune ressource externe, et a pour seule tâche de retourner une copie d'un type primitif, sans effet secondaire sur l'instance qui en est propriétaire (la méthode étant qualifiée `const`).

Dans le cas du destructeur, c'est plus subtil, et nous manquons de moyens à ce stade-ci pour expliquer les raisons détaillées de la clause *no-throw*. Allons-y donc pour des recettes :

- les opérateurs `delete` et `delete[]` (pour les tableaux) ne lèvent jamais d'exception. Notez que `delete nullptr;` et `delete[] nullptr;` sont toutes deux des opérations sans danger;
- un destructeur convenable ne devrait jamais, *jamais* lever d'exceptions. Un destructeur laissant filtrer une exception jette le système dans un état irrécupérable. **Le destructeur d'une classe C++ bien écrite devrait toujours être `noexcept`.** Conséquemment, par défaut, les destructeurs sont qualifiés ainsi et nous n'avons pas à l'expliciter.

```

class TableauEntiers {
    int *tab_;
    int taille_;
public:
    // bloquer la copie
    TableauEntiers(const TableauEntiers&) = delete;
    TableauEntiers&
        operator=(const TableauEntiers&) = delete;
    class TailleInvalide {};
    int taille() const noexcept {
        return taille_;
    }
    int operator[](int n) const {
        return tab_[n];
    }
    int& operator[](int n) {
        return tab_[n];
    }
    explicit TableauEntiers(int n)
        : tab_ {}, taille_{n} {
        if (taille() < 0) throw TailleInvalide{};
        tab_ = new int[taille()];
    }
    ~TableauEntiers() {
        delete [] tab_;
    }
};

```

Le constructeur peut lever au moins deux cas d'exception : une taille invalide (disons une taille inférieure à zéro¹¹²) et une erreur lors de l'allocation dynamique de mémoire.

Les opérateurs `new` et `new[]`, lorsqu'ils rencontrent un problème, lèvent une exception de type `std::bad_alloc`, un type déclaré dans le fichier d'en-tête standard `<new>`.

Pour une taille invalide, on pourrait utiliser l'exception `std::out_of_range` de l'en-tête standard `<stdexcept>`, mais nous explorerons plutôt une technique fort utile : lever des idées pures – des instances de classes vides.

La classe interne `TailleInvalide`¹¹³ sera une classe sans attributs ni méthodes, dont le seul rôle sera d'exister et de représenter un cas de taille invalide pour un `TableauEntiers`.

La déclaration de la classe `TailleInvalide` localement à `TableauEntiers` dénote une appartenance conceptuelle de la première à la seconde. S'il est décidé que `TailleInvalide` est une idée globale, alors on déclarera simplement `TailleInvalide` au niveau global, hors de `TableauEntiers`. Nous reviendrons, plus loin, sur le sujet des classes imbriquées. Il y a beaucoup à dire.

Si la taille est invalide, une instance par défaut (et anonyme) de `TailleInvalide` est créée et levée. Le code appelant pourra utiliser un bloc `catch` sur une instance de `TableauEntiers::TailleInvalide` pour détecter et gérer le problème.

¹¹² Car eh oui, un tableau de taille nulle est légal en C++ s'il a été alloué dynamiquement!

¹¹³ ...ou, pour utiliser son nom visible du monde extérieur, classe `TableauEntiers::TailleInvalide`

```

class HorsBornes {};
class TableauEntiers {
    int *tab_;
    int taille_;
public:
    // bloquer la copie
    TableauEntiers(const TableauEntiers&) = delete;
    TableauEntiers&
        operator=(const TableauEntiers&) = delete;
    class TailleInvalide {};
    int taille() const noexcept {
        return taille_;
    }
    int operator[](int n) const {
        if (0 > n || n >= taille()) throw HorsBornes{};
        return tab_[n];
    }
    int& operator[](int n) {
        if (0 > n || n >= taille()) throw HorsBornes{};
        return tab_[n];
    }
    explicit TableauEntiers(int n)
        : tab_({}, taille_{n}) {
        if (taille() < 0) throw TailleInvalide{};
        tab_ = new int[taille()];
    }
    ~TableauEntiers() {
        delete[] tab_;
    }
};

```

Un autre cas exceptionnel serait celui où un indice invalide serait passé à l'un ou à l'autre des opérateurs `[]`. Dans ce cas, l'exception pourrait se nommer `HorsBornes`. Pour les fins de l'illustration, nous présumerons que la classe `HorsBornes` est un problème d'ordre global, donc déclaré hors de la classe `TableauEntiers`, alors que nous avons défini `TailleInvalide` (plus haut) comme étant un problème d'ordre interne. Remarquez les positions relatives de la spécification `const` des méthodes et celle des clauses `noexcept`.

Cette distinction, pour un problème comme celui-ci, est un peu artificielle mais nous permet de montrer deux syntaxes possibles.

L'utilisation d'exceptions dans les méthodes des opérateurs `[]` peut sembler abusive, mais souvenez-vous que, contrairement à Java et à C#, C++ n'oblige pas l'appelant à envelopper chaque appel à un sous-programme sujet à lancer une exception d'un bloc `try`. Ainsi, un appelant bien élevé n'aura pas à être alourdi par cet ajout.

L'avantage de classes différentes selon les types d'exceptions

Quelle est l'utilité d'avoir recours à des types différents pour chaque catégorie d'exception? On agira ainsi parce que cela permet de déduire la nature d'un problème à l'aide d'une séquence de blocs `catch`, ce qui est très simple et très élégant (sans compter que c'est plus rapide que la plupart des alternatives possibles).

À titre d'exemple, examinez le programme suivant, qui instancie un `TableauEntiers` à partir d'une taille arbitraire et non validée. On décode la nature (et la sévérité) de l'exception (si exception il y a) selon le bloc `catch` qui est entré.

```
// ... inclusions et using...
#include "TableauEntiers.h"
bool poursuivre() {
    cout << "Poursuivre? (O/N)";
    char c;
    return cin >> c && (c == 'o' || c == 'O');
}
int main() {
    int taille = 0;
    do {
        try {
            cin >> taille;
            TableauEntiers tab{ taille }; // risques d'exceptions
            // utiliser tab
        } catch (TableauEntiers::TailleInvalide &) {
            cerr << "Taille invalide (" << Taille << "), recommencez : ";
        } catch (bad_alloc &) {
            cerr << "Mémoire insuffisante. Fin du programme" << endl;
            throw; // rien à faire avec ça
        } catch (...) {
            cerr << "Erreur inattendue. Fin du programme" << endl;
            throw; // rien à faire avec ça
        }
    } while (poursuivre());
}
```

Ce petit programme a plusieurs caractéristiques intéressantes :

- il distingue les exceptions par leur type, utilisant au passage plusieurs blocs `catch` pour un même bloc `try`;
- le fait que chaque type d'exception soit distinct des autres permet à la mécanique des blocs `catch` de détecter le type d'exception levée. Si nous avions utilisé un seul type général d'exception (comme par exemple `std::exception`), nous aurions dû analyser le message des exceptions pour découvrir la nature du problème, ce qui aurait impliqué de l'analyse de chaînes de caractères, des considérations d'internationalisation (langue des messages) et un tas de détails techniques (gestion des accents, par exemple, ou des majuscules et des minuscules);

- la déclaration de `Tab` à l'intérieur du bloc `try`, où il est utilisé, a plusieurs mérites : le destructeur de `TableauEntiers` libérera automatiquement les ressources de cet objet une fois la fin du bloc `try` atteint, et l'intérieur du bloc `try` montre la lignée normale d'opérations sur l'objet, sans que des préoccupations autres ne servent de distractions;
- évidemment, si le constructeur de `tab` lève une exception, alors `tab` n'aura jamais été construit et sa destruction ne se produira pas;
- le code distingue aussi entre une erreur récupérable (une taille invalide) et une erreur dont il n'est pas raisonnable d'essayer de récupérer (un manque de mémoire vive);
- dans le cas irrécupérable, nous aurions aussi pu simplement ne pas attraper l'exception et la laisser filtrer hors de `main()`. Le programme aurait planté, comme il se doit. La mention `throw;` sans parenthèses est ce qu'on nomme un *re-throw*, par quoi le sous-programme relance ce qu'il a attrapé à son propre sous-programme appelant, quoi que ce soit;
- nous avons choisi de traiter explicitement le cas du manque de mémoire avec un message spécifique, par souci de convivialité, et le cas des exceptions autres (`catch(...)`, qui attrape tout ce qui aurait pu nous échapper) par un message générique.

Remarquez que les exceptions ont été **levées par valeur** (dans le code de `TableauEntiers`, plus haut) mais **attrapées par référence**. Sans être nécessaire en soi, ceci évite de provoquer des copies inutiles et améliore la qualité du code sans que cela ne coûte quoi que ce soit.

Exceptions, constructeurs et destructeurs

En C++, si un objet lève une exception dans un constructeur, alors cet objet n'a jamais été créé. Par conséquent, dans l'exemple proposé à droite, le message "Coucou!" s'affichera, mais le message "Urg!", lui, ne s'affichera pas.

Lever des exceptions dans un constructeur est la manière privilégiée de signaler une défectuosité lors de la construction d'un objet. L'alternative serait de laisser un objet incorrect exister, ce qui serait contraire au principe d'encapsulation (un objet construit doit être correct; un objet incorrect ne doit pas être construit).

Après tout, il est impossible pour un constructeur de retourner un code d'erreur, puisqu'un constructeur n'a pas de valeur de retour. Le rôle d'un constructeur est d'initialiser un objet, pas de le retourner.

```
#include <iostream>
using namespace std;
class Zut {};
class X {
    ostream & os_;
public:
    X(ostream &os) : os_(os) {
        os_ << "Coucou!" << endl;
        throw Zut{};
    }
    ~X() {
        os_ << "Urg!" << endl;
    }
};
int main() {
    X x{cout};
}
```

Nous couvrirons les méandres du cycle de construction d'un objet dans [POOv03], section *Gestion avancée de la mémoire*. De même, nous examinerons plus en détail les ramifications de l'idiome RAII [hdIdiom], des constructeurs et des exceptions dans [POOv01].

Notez qu'il ne faut pas lever d'exceptions dans un destructeur en C++. Ceci mettrait fin au programme de manière irrécupérable. Heureusement, les opérateurs `delete` et `delete[]` ne lèvent jamais d'exceptions.

Dans [SutterExCtor], **Herb Sutter** étend cette vision des constructeurs dans lesquels une exception est levée à C# et à Java, mais en pratique, en C#, un objet dans lequel une exception est levée *mais attrapée* voit son code de finalisation invoqué (mais l'objet dont la construction a échoué demeure inutilisable en pratique, évidemment), contrairement à ce qui se produit en C++ où le destructeur d'un tel objet ne le serait pas. Notez toutefois que, si l'exception n'est pas attrapée et si le programme échoue rapidement, le code de finalisation n'est pas nécessairement invoqué.

Cela entraîne des adaptations locales aux langages des pratiques quant aux objets dont les constructeurs échouent. Entre autres, en C#, si du code susceptible de lever une exception apparaît dans un constructeur, il est préférable de s'assurer au préalable que les attributs susceptibles d'être manipulés dans le code de finalisation soient initialisés avec des valeurs qui sont clairement identifiables au préalable. Ce faisant, ces attributs pourront être testés avant finalisation.

```
using System;
namespace Z
{
    class Test
    {
        public Test()
        {
            Console.WriteLine("Début");
            throw new Exception();
        }
        ~Test()
        {
            Console.WriteLine("Fin");
        }
        public static void Main(string[] args)
        {
            try
            {
                Test t = new Test();
            }
            catch (Exception)
            {
            }
        }
    }
}
```

Le fait que le code de finalisation en Java ne soit pas invoqué de manière prévisible complique la mise en place de code de test et de comparatifs.

Exercices – Série 10

Rédigez un programme console interagissant avec un usager, que nous prénommerons le *Merveilleux Gestionnaire de Pool*, version 0.1 (ou, en écriture abrégée: MGP0.1).

Ce programme utilisera la classe **Joueur**, servant à représenter un joueur de hockey (pour simplifier les choses, nous ne considérerons pas les gardiens de buts pour le moment). Une grande partie de votre travail sera de la rédiger en fonction des critères donnés plus loin.

Ce programme devra aussi utiliser un tableau de MAX_JOUEURS instances de Joueur (peu importe la valeur de cette constante; vous pouvez utiliser 5, pour la période de débogage du programme). *Ce tableau représentera une équipe d'un pool de hockey.*

Le rôle du programme sera de calculer certaines informations pertinentes quant à une équipe (donc quant aux données du tableau). Il devra donc :

- lire le nom, le nombre de buts et le nombre de passes de chaque joueur; puis
- effectuer un certain nombre de traitements sur ceux-ci, pour déterminer par exemple le nombre de points de l'équipe en entier, la moyenne des points, le meilleur pointeur de l'équipe, etc. La liste des traitements exigés est donnée plus loin.

Qu'est-ce qu'un pool de hockey¹¹⁴?

Voici un très bref (et très incomplet) descriptif de ce qu'est un *pool de hockey*.

Un **pool de hockey** est une activité sociale par laquelle des individus s'approprient les performances d'un certain nombre de joueurs de hockey. Une ronde de sélection a lieu à un certain moment, où chaque personne choisit un certain nombre de joueurs¹¹⁵, puis chacun(e) suit les performances des joueurs qu'il/ elle a sélectionné (*son équipe*).

Dans bien des cas, chaque personne contribuera un (petit) montant à une cagnotte commune, qui sera offerte à la fin de la saison à la personne dont l'équipe aura amassé le plus de points au total.

Chaque pool a ses propres règles internes et sa propre manière de calculer les points.

Notre programme servira donc d'utilitaire très simplifié pour permettre le calcul de certaines statistiques élémentaires dans ce contexte.

¹¹⁴ Le concept, évidemment, peut s'étendre à d'autres sports, et à d'autres domaines que le sport professionnel, avec un peu d'imagination.

¹¹⁵ En général de la *Ligue Nationale de Hockey*, la LNH.

La classe *Joueur*

Chaque *Joueur* aura comme attributs son nom (une chaîne de caractères), son nombre de buts et son nombre de passes (des entiers).

On voudra retrouver, parmi les opérations possibles sur un joueur, *que vous utilisiez ou non chacune d'entre elles pour ce programme*¹¹⁶ :

- un constructeur par défaut;
- un constructeur paramétrique (recevant en paramètre une chaîne de caractères et deux entiers);
- un constructeur de copie;
- un destructeur;
- des mutateurs pour modifier le nom, le nombre de buts et le nombre de passes d'un *Joueur*. Un nombre de buts (ou un nombre de passes) négatif est considéré invalide;
- des accesseurs permettant de connaître le nom, le nombre de buts et le nombre de passes du *Joueur*, *de même que le nombre de points de ce Joueur*. Pour nous le nombre de points d'un *Joueur* sera son nombre de passes, auquel on additionne deux points par but marqué;
- on offrira l'opérateur = pour permettre l'affectation d'un *Joueur* à un autre *Joueur*;
- on offrira les opérateurs == et != pour déterminer si deux instances de *Joueur* représentent (ou non) le même *Joueur*. On dira que deux instances de *Joueur* représentent le même *Joueur* si celles-ci ont le même nom;
- on offrira les opérateurs <, <=, > et >= pour permettre de comparer deux instances de *Joueur* selon leur nombre de points respectifs. Si *j1* et *j2* sont deux instances de *Joueur*, on dira que *j1* < *j2* si *j1* a moins de points que *j2* (vous pouvez déduire les autres par vous-mêmes).

¹¹⁶ On ne développe pas une classe juste pour les besoins du moment, bien que ceux-ci orientent nécessairement notre réflexion en période de design ou de développement.

L'exécution du programme

Le programme devra s'exécuter en deux temps :

- lecture des données propres aux `MAX_JOUEURS` joueurs; et
- traitement et affichage des résultats (pour chaque traitement demandé).

Ceci signifie que vous devrez lire les données pour tous les joueurs avant de commencer la phase de traitement; ceci vous obligera à utiliser un tableau de `Joueur`.

Plus difficile : faites en sorte qu'il soit impossible d'avoir deux joueurs identiques (prenant *identique* au sens de l'opérateur `==` défini plus haut).

On veut que le programme calcule et affiche :

- la somme des points des joueurs de l'équipe;
- la moyenne des points des joueurs de l'équipe;
- le nom et le pointage du meilleur pointeur de l'équipe; et
- le nom et le pointage du pire pointeur de l'équipe.

Plus difficile : présentez aussi à l'écran les joueurs en ordre décroissant de pointage (du meilleur pointeur au pire). Vous pouvez examiner la bibliothèque `<algorithm>` de même que la classe `std::vector` de la bibliothèque `<vector>` pour vous aider. Il vous faudra lire un peu par vous-mêmes pour comprendre, mais ça en vaudra la peine.

Vous pouvez utiliser les sous-programmes et modules de votre choix, dans le respect des règles programmatiques du cours (et dans la mesure où votre programme fonctionne).

Dans tous les cas, il faut que votre programme fonctionne peu importe la valeur de la constante `MAX_JOUEURS`¹¹⁷.

¹¹⁷ Dans la mesure, bien sûr, où celle-ci est un entier strictement positif.

Raffinements conceptuels et technique

Cette courte section exposera quelques éléments qui sont de l'ordre du raffinement conceptuel et technique. Ces éléments, bien qu'importants, sont difficiles à placer à l'intérieur d'un enchaînement dans l'une ou l'autre des autres sections du document.

Déclarations a priori

Un compilateur C++, à la manière d'un compilateur C, s'adonne à de la compilation séparée, au sens où chaque fichier source y est compilé séparément, isolé des autres fichiers sources. Le lien entre les fichiers objets résultants est tissé par l'éditeur de liens, suite au passage du compilateur. Ce choix philosophique a pour conséquence que chaque fichier source doit être indépendant des autres au sens de la compilation.

D'autres langages ont faits d'autres choix. Java et les langages .NET ont des modèles de compilation très différents de C++. La simplicité du modèle de C et de C++ fait d'eux des langages souvent utilisés pour interfacer avec d'autres langages : le modèle de compilation et d'édition des liens qu'ils appliquent est compréhensible sans trop d'efforts.

Certains langages OO mettent beaucoup l'accent sur le dynamisme. Ces langages sont très souples, très flexibles, mais le dynamisme a un prix (en gros : moins le compilateur en sait sur le programme, moins il sera capable d'optimiser le code), allant parfois jusqu'au point de ne pas demander aux programmeurs d'indiquer les types des données sur lesquelles ils opèrent.

Les langages OO qui se destinent à un marché plus vaste cherchent habituellement un équilibre entre le dynamisme et le savoir statique et le dynamisme; ces langages placent le seuil de l'équilibre à divers endroits pour des raisons philosophiques, toutes bonnes à leur manière. Java et les langages .NET représentent les objets à même leurs bibliothèques de classes standards, ce qui leur permet d'implémenter la réflexivité [POOv03]. C++ met l'accent sur la généricité statique avec les *templates* [POOv02]. Tout langage OO qui se respecte supporte l'abstraction et le polymorphisme [POOv01].

Le compilateur C++ résout un maximum de problèmes dès la compilation. De plus, ce langage permet un usage naturel de la sémantique directe d'accès aux objets (*Appendice 00 – Sémantiques directes et indirectes*), alors que .NET et Java ne permettent d'accéder aux objets que par des indirections.

Un effet secondaire de cette préférence qu'a C++ pour le traitement statique face au traitement dynamique est que *la taille de tout objet doit être connue lors de son instanciation*. Sachant que la taille d'un objet est directement liée à la somme des tailles de ses attributs, et sachant que C++ permet les instanciations automatiques et statiques, il s'ensuit que toute déclaration de classe en C++ doit proposer la *totalité* de l'information nécessaire pour déterminer la taille et la structure interne de toute instance de cette classe.

En effet, utiliser les objets directement demande au compilateur de générer du code dans lequel ces objets occupent un espace connu à la compilation.

Ainsi, en C++, pour une classe `Personne` possédant un attribut d'instance de type `std::string`, l'espace occupé dans chaque instance par cet attribut ne sera pas celui d'une référence ou d'un pointeur, comme en Java ou en C#, mais bien celui d'une `std::string` entière.

En Java par exemple, si chaque instance de `Personne` a un attribut `String`, l'espace occupé par cet attribut dans la `Personne` est un espace de taille fixe, celui d'une référence, puisque seuls les accès indirects sont permis. La `String` vers laquelle cette référence mène sera allouée dynamiquement ailleurs dans l'espace adressable du programme.

Pour connaître l'espace occupé par un type donné, le compilateur doit en voir la déclaration. Dans notre exemple, `Personne.h` inclura `<string>` avant la déclaration de la classe `Personne`. La taille de `std::string` influence la taille de `Personne`, et la déclaration de `std::string` doit être connue du compilateur pour que celui-ci comprenne la déclaration de `Personne`.

Ceci mène parfois à un problème de poule et d'œuf :

- imaginons un `Bonhomme` vivant dans un `Monde`;
- chaque `Monde` contient (au pire) BEAUCOUP d'instances de `Bonhomme`;
- chaque `Bonhomme` sait dans quel `Monde` il est.

En Java comme en C#, un tel problème est simple à modéliser. En Java, à titre d'exemple, nous aurions quelque chose comme ceci.

Bonhomme.java	Monde.java
<pre>class Bonhomme { private Monde monde; private setMonde(Monde m) throws NullPointerException { if (m == null) throw new NullPointerException(); monde = m; } public Bonhomme(Monde m) { setMonde(m); } // ... }</pre>	<pre>class Monde { private static final int BEAUCOUP = 10000; private Bonhomme[] habitants; public Monde() { habitants = new Bonhomme[BEAUCOUP]; for (int i=0; i < habitants.length; i++) { habitants[i] = null; } // ... } }</pre>

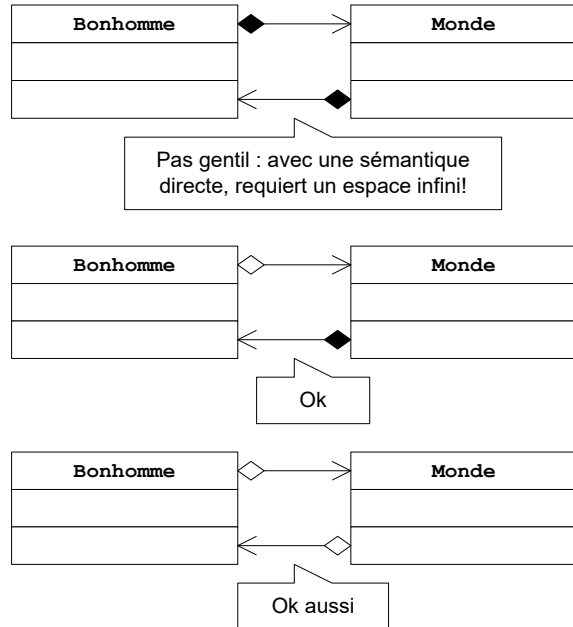
Cette structure est simple et fonctionne naturellement pour deux raisons :

- le langage repose sur un modèle de compilation différent, qui ne demande pas d'inclure lexicalement (avec `#include`) les déclarations des classes avant de les utiliser; et
- une instance de `Bonhomme` ne contient pas de `Monde`, pas plus qu'une instance de `Monde` ne contient des instances de `Bonhomme`. Chacune de ces classes contient des *références* vers des instances de l'autre, sans plus.

L'approche indirecte est d'ailleurs raisonnable ici : s'il est plausible (sans être nécessairement la meilleure approche) de dire que chaque Monde contient des instances de Bonhomme, il est beaucoup moins plausible de prétendre que chaque Bonhomme contient un Monde¹¹⁸.

Il serait plus raisonnable de dire que chaque Bonhomme connaît son Monde; une relation indirecte d'agrégation, pas de composition.

Ce faisant, les deux schémas à droite sont raisonnables. Celui du dessus fait en sorte que Monde contienne ses instances de Bonhomme et que chaque Bonhomme connaisse son Monde, alors que celui du dessous fait en sorte que Monde connaisse indirectement (pointeurs ou références) ses instances de Bonhomme.



Les relations indirectes en C++ sont communes : les pointeurs et les références y font partie du quotidien de toutes les programmeuses et de tous les programmeurs. Cependant, il y a un hic : *en C++, un nom doit être déclaré avant d'être utilisé*. Nous voilà donc (en apparence) revenus à la case départ, comme le montre l'illustration suivante.

Bonhomme . h	Monde . h
<pre> #ifndef BONHOMME_H #define BONHOMME_H #include "Monde.h" // hum // autres inclusions class MondeNul {}; class Bonhomme { Monde *monde_; void habiter(Monde *m) { if (!m) throw MondeNul{}; monde_ = m; } public: Bonhomme(Monde *m) { habiter(m); } // ... }; #endif </pre>	<pre> #ifndef MONDE_H #define MONDE_H #include "Bonhomme.h" // hum // autres inclusions class Monde { static const int BEAUCOUP = 10'000; Bonhomme *habitants_[BEAUCOUP] {}; public: Monde() = default; // ... }; #endif </pre>

¹¹⁸ Si l'une contient l'autre et l'autre contient l'une, la taille de chacun sera infinie!

Cette illustration est toutefois flouée, et on peut faire beaucoup mieux. En effet, trop de gens pensent que cette illustration est la seule approche possible en C++. Ceci amène les gens à émettre des critiques parfois acerbes envers le langage, pensant que :

- chaque classe se doit de présenter la totalité de sa structure interne à tout module en incluant la déclaration; et que
- certaines structures en apparence circulaires ne peuvent y être représentées.

La première critique signifie qu'une classe `Contenant` ne pourrait utiliser une classe `Contenu` que si la déclaration de `Contenu` est connue avant celle de `Contenant`. Dans notre cas, remplacez `Contenant` par `Monde` et `Contenu` par `Bonhomme` et vous aurez le portrait du problème sur lequel nous travaillons.

Sous cette optique, on serait porté à penser qu'inclure `Contenant.h` devient une sorte de bris d'encapsulation de `Contenu`, et que pour utiliser `Contenant`, on se voit nécessairement forcé de connaître le détail de l'interface et de la structure interne de `Contenu`.

Forcer un module utilisant un `Contenant` à connaître le détail d'un `Contenu` est, de toute manière, un triste (mais sérieux) bris d'encapsulation.

La seconde critique signifie que, suivant le même exemple, il est impossible pour un `Contenu` de connaître son `Contenant` dans avoir au préalable inclus la déclaration de `Contenant`, qui nécessite quant à elle la déclaration de `Contenu`, créant un inextricable cercle vicieux.

```
#include "Contenant.h"
class Contenu {
    Contenant *contenant;
public:
    Contenu(Contenant *enveloppe);
    // ...
};
```

Heureusement, on peut répondre à chacune de ces critiques avec une certaine élégance et avec une certaine facilité, grâce à une mécanique qu'on nomme les **déclarations a priori** (en anglais : *Forward Class Declarations*). Mêler déclarations *a priori* et compilation séparée des fichiers sources permet de résoudre la plupart des situations de circularité, et permet aussi d'augmenter singulièrement la qualité de l'encapsulation dans les programmes.

Rôle d'une déclaration a priori

On voit une analogie directe entre déclarations *a priori* et prototypes de sous-programmes :

⇒ Une **déclaration a priori** sert à introduire un nom, sans toutefois le définir.

Bien que ceci puisse sembler étrange, connaître l'existence d'un nom suffit pour pouvoir déclarer et utiliser des pointeurs vers ce nom et des références sur ce nom.

⇒ Pour déclarer *a priori* la classe `C`, il suffit d'écrire `class C;`.

⇒ De même, pour déclarer *a priori* le `struct S`, il suffit d'écrire `struct S;`.

Bien entendu, toute utilisation du type déclaré *a priori* qui sollicite une connaissance de sa taille ou de sa structure (toute instanciation, toute copie, ou toute autre utilisation d'un de ses membres) doit être précédée de près ou de loin par la déclaration pleine et entière de ce type.

On utilisera les déclarations *a priori* pour réduire le couplage (donc les dépendances) envers des classes aux endroits où seules des indirections sont requises. Entre autres avantages, les déclarations *a priori* brisent la circularité dans des cas comme ceux que nous examinons ici, et permettent de faciliter le développement de classes de manière indépendante et réduisant le savoir que chacune doit posséder sur l'autre.

Utilisation de déclarations *a priori*

Comment utilise-t-on les déclarations *a priori* pour résoudre des problèmes comme celui des classes `Contenant` et `Contenu` proposé ci-dessus?

En premier lieu, dans `Contenant.h`, on pourrait supprimer entièrement la directive d'inclusion lexicale de `Contenu.h` et la remplacer par une déclaration *a priori* de la classe `Contenu`.

Ceci nous contraint à déclarer non pas un `Contenu`, mais bien un pointeur (ou une référence) sur un `Contenu`.

```
class Contenu;
class Contenant {
    Contenu *contenu;
    // ...
};
// Contenu.h sera inclus
// par Contenant.cpp
```

Il est clair que `Contenant` voudra éventuellement utiliser des méthodes de `Contenu`. Il le pourra, dans la mesure où il inclut `Contenu.h` au préalable. Inclure dans un `.cpp` n'introduit pas de dépendances circulaires car on n'inclut pas les `.cpp`; toute dépendance introduite sera univoque.

Remarquez qu'en limitant les mentions de `Contenu` dans `Contenant.h` à des déclarations de pointeurs et de références, il suffit à `Contenant.h` de faire usage d'une déclaration *a priori* de `Contenu`.

L'inclusion de `Contenu.h` n'aura donc plus à se faire dans `Contenant.h` mais bien dans le fichier source `Contenant.cpp`, là où cela ne posera plus problème.

Ensuite, la classe `Contenu`, qui désire utiliser des pointeurs de `Contenant` dans sa déclaration n'a pas à inclure `Contenant.h` pour y arriver. La déclaration *a priori* de la classe `Contenant` suffit.

```
class Contenant;
class Contenu {
    Contenant *contenant;
public:
    Contenu(Contenant *enveloppe);
};
// Contenant.h sera inclus
// par Contenu.cpp
```

Notez encore une fois que pour pouvoir utiliser les méthodes de `Contenant`, il faudra ici aussi inclure `Contenant.h`, mais on peut maintenant n'inclure ce fichier d'en-tête que là où on en a vraiment besoin : dans `Contenu.cpp`, pour lequel cela ne pose pas le moindre problème.

Revenant à notre exemple, une solution possible serait celle-ci :

Bonhomme . h	Monde . h
<pre> #ifndef BONHOMME_H #define BONHOMME_H class Monde; // autres inclusions class MondeNul {}; class Bonhomme { Monde *monde_; void habiter(Monde *m) { if (!m) throw MondeNul{}; monde_ = m; } public: Bonhomme(Monde *m) { habiter(m); } // ... }; #endif </pre>	<pre> #ifndef MONDE_H #define MONDE_H #include "Bonhomme.h" // autres inclusions class Monde { static const int BEAUCOUP = 10'000; Bonhomme habitants_[BEAUCOUP]; public: // ... plus besoin de créer des // instances de Bonhomme avec new, // mais chaque Monde contient un // tas (BEAUCOUP) d'instances // typiques de Bonhomme }; #endif </pre>

Une autre serait celle-là :

Bonhomme . h	Monde . h
<pre> #ifndef BONHOMME_H #define BONHOMME_H class Monde; // autres inclusions class MondeNul {}; class Bonhomme { Monde *monde_; void habiter(Monde *m) { if (!m) throw MondeNul{}; monde_ = m; } public: Bonhomme(Monde *m) { habiter(m); } // ... }; #endif </pre>	<pre> #ifndef MONDE_H #define MONDE_H class Bonhomme; // autres inclusions class Monde { static const int BEAUCOUP = 10'000; Bonhomme *habitants_[BEAUCOUP] {}; public: Monde() = default; // ... }; #endif </pre>

Les deux ont leurs avantages et leurs inconvénients, mais les examiner en détail demande au moins d'avoir compris la matière de [POOv01]. Notons tout de même que les déclarations a priori ont permis un découplage du code, ont accru l'encapsulation en faisant disparaître des détails techniques des fichiers d'en-tête, et ont permis de briser une circularité structurelle apparente. Pas si mal, somme toute.

Dans d'autres langages

La déclaration *a priori* n'a de sens que dans une situation de compilation séparée comme en C ou en C++, du fait que le compilateur traverse le texte du code de haut en bas pour générer le code objet correspondant au fur et à mesure et que l'éditeur de liens (du langage C, donc qui se veut extrêmement simple) passe par la suite pour intégrer les fichiers objet en une forme complète (que ce soit une bibliothèque, un exécutable ou toute autre variante).

En Java et dans les langages .NET, les modules sont compilés dans une norme intermédiaire puis liés par un éditeur de liens plus moderne. L'information disponible peut donc être plus riche et l'éditeur de liens travaille à partir de contraintes différentes.

Cela dit, il y a un côté obscur à ce gain apparent de puissance. L'une des raisons pour lesquelles le compilateur peut intégrer des classes compilées ici et là sans nécessairement dépendre de l'apparition au préalable de leurs noms est qu'il est en mesure de découvrir les classes, leur structure et leur comportement de manière dynamique.

Un mécanisme OO important et surtout utilisé dans les langages plus dynamiques comme Java et les langages .NET, mécanisme qu'on nomme la **réflexivité** et sur lequel nous reviendrons dans [POOv03], permet à un objet de se représenter lui-même et de se décrire sous forme d'objets, ce qui permet entre autres une reconstitution et une instanciation dynamique des objets. Évidemment, la présence de mécanismes de réflexivité implique un effet secondaire : l'incapacité de véritablement empêcher le code client d'investiguer structurellement les sources d'un programme, à moins peut-être que ces sources aient été altérées de manière à les rendre difficile à reconstruire ou à comprendre (ce qu'on nomme le *Code Obfuscation*).

Simuler un passage par nom¹¹⁹

Un langage OO permet d'exprimer avec élégance des idées fondamentales sur la nature-même de l'existence des entités programmées, sur leur arrivée au monde et sur leur rôle quand elles y sont. Nous verrons ici un petit idiome de programmation qui permet, lorsqu'on le considère utile, d'offrir au compilateur les outils pour assister les programmeuses et les programmeurs dans la clarté de l'expression de leurs idées.

Ajouter de petites classes au rôle pointu, comme nous le ferons ici, permet :

- de lever des ambiguïtés;
- de remplacer la discipline de programmation (commentaires et bonnes pratiques), qui relèvent du bon vouloir et de la rigueur des humains, par des indications structurantes à même le programme, réduisant ainsi fortement les risques d'erreurs;
- d'enrichir, au passage, la syntaxe. En effet, un type bien écrit et dont le nom reflète la vocation est porteur de sens, et guide la lecture des programmes.

L'idiome ci-dessous se situe à un niveau de granularité très fine, et ne convient pas à toutes les applications¹²⁰. Cependant, il donne un (petit) aperçu de ce que la POO offre aux gens capables de penser hors des cadres établis.

La problématique qui nous intéressera ici se présente comme suit : C++ est un langage dans lequel les paramètres sont passés sur une base positionnelle, en ce sens que le premier paramètre à l'invocation correspond au premier dans le prototype du sous-programme appelé, le second correspond au second, *etc.*

Ceci explique que, lorsqu'un sous-programme appose des valeurs par défaut pour certains paramètres, ceux-ci doivent être placés à la fin de la liste des paramètres pour éviter les ambiguïtés.

Les noms des paramètres lors de l'appel ne correspondent en rien, sinon de manière accidentelle, aux noms des paramètres dans le sous-programme appelé.

Ceci peut mener à de la confusion lorsque la documentation d'un sous-programme est déficiente. Imaginons par exemple la classe `Rectangle` proposée à droite.

Faute de documentation adéquate ou de noms de paramètres dans le constructeur paramétrique, comment savoir si les paramètres *sont Largeur, Hauteur* ou *Hauteur, Largeur* (ou autre chose)?

```
class Rectangle {
    int largeur_,
        hauteur_;
public:
    Rectangle();
    // Mal documenté... nous voilà mal pris!
    Rectangle(int, int);
    // ... etc.
};
```

¹¹⁹ J'ai été inspiré par <http://www.artima.com/cppsource/typesafety3.html> pour l'écriture de cette petite section. Le début est en partie inspiré des réflexions sur les métalangages des très brillants *Dave Abrahams* et *Aleksey Gurtovoy* dans [CppMeta].

¹²⁰ On pourrait par contre l'enrichir avec des techniques de programmation générique [POOv02] pour qu'il devienne beaucoup plus intéressant en pratique.

Une saine solution à un tel problème est de clarifier la documentation de l'interface visible de `Rectangle` pour lever toute ambiguïté. Nommer les paramètres dans la signature d'un sous-programme est un pas en avant dans les cas où une ambiguïté est possible.

Un idiome de programmation, solutionnant de manière formelle ce problème (un peu lourd à écrire mais rapide à l'exécution), est de remplacer les types des paramètres par des classes aux noms clairs et encapsulant, à la limite, leurs propres politiques de validation.

Cet idiome se nomme **Idiome de la valeur entière** (en anglais : *Whole Value Pattern*).

Dans ce cas-ci, la manœuvre consiste à définir un type au constructeur paramétrique explicite pour les paramètres qu'on souhaite clairement nommés lors de l'instanciation de l'objet. Étant explicites, ils doivent être clairement identifiés comme tels dans le code client, donc lors de l'invocation des méthodes qui les utilisent.

Le code client proposé à droite montre un exemple clair d'utilisation. Ici, si le code client se trompe dans l'ordre, il aura une erreur à la compilation, ce qui lui évitera beaucoup de bogues à l'exécution.

```
class Hauteur {
    int valeur_;
    static int valider(int);
public:
    explicit Hauteur(int valeur)
        : valeur_{ valide(valeur) } {
    }
    int valeur() const {
        return valeur_;
    }
};

// même idée pour Largeur
class Rectangle {
    Hauteur hauteur_;
    Largeur largeur_;
public:
    Rectangle();
    Rectangle(const Largeur&, const Hauteur&);
    // ... etc.
};
```

```
int main() {
    Rectangle r{ Hauteur{3}, Largeur{5} };
    // utiliser r
}
```


Liée au traitement d'exceptions et à la préconstruction, cette technique montre comment une approche OO peut solidifier les programmes :

- si la classe `Hauteur` assure l'encapsulation et connaît les règles de validité pour une `Hauteur`, alors il sera impossible de construire une instance de `Hauteur` invalide (une instance de `Hauteur` lèvera une exception à la construction si on tente de lui imposer une valeur non respectueuse de ses règles de validité). Il en va de même pour une `Largeur`;
- les classes `Hauteur` et `Largeur` sont des types valeurs, donc la construction par copie, l'affectation et la destruction y sont banales et implicites. Le constructeur paramétrique explicite fait en sorte de cacher le constructeur par défaut. La seule manière par laquelle le code client pourrait tenter de construire une instance corrompue de ces classes est donc la construction paramétrique, qui sera blindée;
- un `Rectangle` devra instancier sa `Hauteur` et sa `Largeur` en utilisant leurs constructeurs paramétriques, et devra donc utiliser la préconstruction (ces classes n'ayant pas de constructeurs par défaut). Conséquemment, si une valeur invalide est passée à la construction d'un des attributs, cette construction échouera, et il en sera de même pour la construction du `Rectangle` dans son ensemble;
- conséquemment, construire un `Rectangle` invalide sera impossible;
- les politiques de validité de `Hauteur` et de `Largeur` ont été déplacées vers ces classes. `Rectangle` utilise des instances de ces classes en leur faisant confiance. Chaque entité est pleinement responsabilisée face à elle-même.

Notez que, si un `Rectangle` n'a comme attribut que des types valeurs, alors `Rectangle` est aussi, de manière transitive, un type valeur, et ses opérations de copie, tout comme sa destruction, sont aussi banales et implicites.

Thématiques diverses

Plusieurs thématiques ○○ ne sont pas du tout d'ordre technique. On pense par exemple aux choix de nomenclature; au concept même d'objet, pris dans un sens historique; aux relations entre objets, *etc.*

Dans cette section, nous explorerons des thématiques qui ont comme double particularité d'être d'ordre conceptuel et d'être compréhensibles même avec le mince bagage accumulé jusqu'ici.

Sens historique du mot objet

La lectrice attentive et le lecteur attentif auront remarqué la réticence de l'auteur de ces lignes à employer le mot *objet* pour parler d'une instance d'une classe donnée. Cela tient à son éducation et à sa perspective, toutes deux un peu particulières.

Lorsque le modèle objet a commencé à prendre de l'essor, avant que C++ et Java ne prennent le gros du plancher et que UML ne fasse son apparition comme canevas de langage descriptif, les débats faisaient rage entre les *aficionados* d'une approche objet puriste, selon laquelle il ne devrait y avoir aucun type primitif, que des objets, et ceux d'une approche hybride, les derniers visant des résultats plus immédiats alors que les premiers tendaient à faire primer leurs standards d'esthétique et de qualité générale sur la réalisation d'un produit final efficace. C'est une généralisation grossière, mais l'idée est là. De tels débats à teneur philosophique sont fréquents en période de fébrilité créatrice, peu importe le domaine, et nous sommes tous tributaires aujourd'hui des discussions qui eurent lieu alors¹²¹.

L'une des idées maîtresses du modèle objet, dès le début, était la jonction en une même unité nommée et sémantiquement cohérente de données et des opérations s'appliquant à celles-ci. L'*encapsulation* est une extension de ce regroupement, une fois les contraintes de protection des données ajoutées à l'idée de *composition* (quand un objet en contient d'autres).

Au sens initial, donc, *une instance d'une classe donnée est un objet...* mais toujours au sens initial, *une classe*, avec ses attributs de classe et ses méthodes de classe, *est aussi un objet*. Ramener le vocable objet au sens simplifié d'instance d'une classe est donc une restriction légèrement dommageable du concept, du moins pour ceux ayant adopté une pensée objet depuis suffisamment longtemps.

C++ et UML ont fait beaucoup pour ce glissement : les deux notations véhiculent des préoccupations pragmatiques, et le sens concret d'objet se rattache plus facilement à l'idée d'instance qu'à celui de classe, l'effort d'abstraction étant moins grand. Pour le commun des gens gravitant dans le monde objet, associer objet et instance seulement suffit pour exprimer la plupart des idées. C'est seulement lorsqu'on essaie de s'interroger sur le fondement d'un langage que la jonction des idées d'objet et de classe devient nécessaire; les gens ayant besoin de réfléchir à ces choses-là sont, effectivement, peu nombreux.

¹²¹ Voir à ce sujet le [WorseBetter] de **Richard Gabriel**. On y fait état de LISP, très élégant, et de C, correct du point de vue de l'esthétique mais très pragmatique dans ses prises de position et dans son approche au développement. On sait quelles parts de marché ces deux langages ont pris avec le temps; il y a clairement des leçons à en tirer.

Java et C# sont, à certains égards, des langages plus OO que C++ (et à d'autres égards, ils le sont un peu moins). Ce qui est énoncé ici sur Java tient aussi pour C#. Avec Java¹²², toutes les classes sont des spécialisations plus ou moins lointaines d'une classe assez abstraite nommée `Object`; on retrouve dans la classe `Object` tout ce que tous les objets ont en commun. Vous lirez par vous-mêmes à ce sujet si cela vous intéresse; à mon sens, c'est passionnant.

En Java toujours, il existe une classe `Class` à partir de laquelle sont représentées les classes du langage. Une classe est un objet car la classe `Class` est une spécialisation de la classe `Object`... mais la classe `Object` est représentée par une instance de la classe `Class`.

Quand on atteint un niveau d'abstraction suffisant, les idées de classe et d'instance se confondent dans le concept d'objet. Toutes les hiérarchies d'objet suffisamment abstraites finissent par demander une réflexion de ce genre (le mot *réflexion* ici porte plus d'un sens, mais nous y reviendrons un peu plus loin).

Pour moi, donc, le mot *objet* représente autant l'idée de classe que l'idée d'instance. C'est pourquoi je n'utilise qu'avec prudence ce mot lorsque je discute d'instances : je veux éviter une confusion des genres dans laquelle, à mon avis, nous perdons quelque chose de profond sur le plan de la sémantique.

Cela dit, les textes discutant de C++ utilisent presque exclusivement le vocable objet au sens d'instance, et les textes traitant de Java essaient de se rapprocher au maximum de la nomenclature C++, pour garder une base commune de développeurs qui soit la plus vaste et la plus mobile possible. Dans les deux cas, ça se comprend.

La norme UML, qui vise à atteindre une forme de leadership dans la description graphique d'objets et de systèmes composés d'objets, décrit quant à elle le mot objet comme étant *l'instanciation d'une abstraction*. La prise de position est on ne peut plus claire : on cherche à oublier la racine conceptuelle commune des instances et des classes.

Notez par contre qu'avec les métaclases, possibles dans plusieurs langages (même en C++, dans une certaine mesure : voir [POOv03]), une classe est aussi l'instanciation d'une abstraction d'un ordre supérieur.

Le modèle objet est un modèle en santé, qui évolue. Ce n'est pas votre humble serviteur qui dicte les directions que peut prendre le vocabulaire objet; le fait que la pensée pragmatique ait pris le dessus sur la pensée esthétique, si je peux encore une fois généraliser ainsi, est un signe que l'ensemble de la communauté s'est appropriée la plupart des idées du modèle, et que ce modèle est appliqué à des problèmes concrets et mène à des solutions qui sont appréciées et utilisées. La perversion (à mon sens) du vocabulaire est un contrecoup d'une démocratisation des idées qui, elle, ne peut être vue que comme une forme de triomphe.

L'avenir appartient à la majorité... et c'est tant pis pour les dinosaures tel l'auteur de ces lignes.

¹²² Comme en *Smalltalk* et dans d'autres langages se voulant *plus purs*...

Reconnaître la présence d'objets et décrire les relations entre objets

Le développement d'une solution OO nous mène naturellement à chercher quels sont les bons candidats à être représentés sous forme d'objets dans la solution d'un problème. On peut considérer certains critères généraux, comme entre autres :

- un objet regroupe des attributs (données) et des méthodes (sous-programmes) dans une même entité, typiquement nommée¹²³. Si on peut associer, autour d'une même idée, des données et des opérations, lui donner une vocation claire et délimitée, et qu'on peut donner un nom à cette idée, on tient peut-être là un bon candidat à une représentation objet;
- un objet est voué à une utilisation de longue durée, idéalement dans plusieurs programmes et dans des contextes variés. Si on peut modéliser quelque chose de manière à pouvoir l'utiliser dans plus d'un contexte, il s'agit peut-être d'un bon candidat à une représentation objet;
- un objet est quelque chose d'identifiable et de reconnaissable comme entité propre. On devrait normalement pouvoir le pointer du doigt et être en mesure de décrire son rôle, sa raison d'être;
- un objet doit être utile, étant voué à être utilisé. Ce qui constitue un bon objet dépendra conséquemment du contexte dans lequel il sera utilisé... et donc, de chaque entreprise, de sa culture et de ses projets;
- ce sera plus clair sous peu, mais si quelque chose peut être hiérarchisé ou catégorisé, côté structure comme côté opérations, en général on peut souhaiter le modéliser par objets;
- dans le même ordre d'idée, ce qui peut être abstrait et généralisé se prête en général bien à une approche objet. On verra plus tard les conteneurs standards¹²⁴, qui sont un exemple classique en ce sens.

Les exemples utilisés dans ce document sont délibérément simples, touchant à des structures de la vie courante (des formes géométriques simples; un compte bancaire; une personne). Ce sont des exemples à vocation pédagogique, et certains d'entre eux auraient une forme différente dans du code de production. Cette simplicité est délibérée : on veut se concentrer sur la matière et éviter les écueils de compréhension qui nous feraient nous attarder sur des détails accessoires.

Il ne faut pas prendre cette simplification comme une indication que seuls les objets de la vie courante sont voués à une modélisation par objets. On pourra modéliser par objet ce qui semble voué à une vie longue, à une utilisation massive, à des optimisations ultérieures, et à toute forme de généralisation, d'abstraction et de hiérarchisation.

¹²³ Il existe, après tout, des instances anonymes (temporaires, créées pour la durée d'une expression... Par exemple, l'expression `cout << string("allo")+"toi";` crée plusieurs objets anonymes) et des classes anonymes, dans divers langages, ce sur quoi nous reviendrons dans des documents ultérieurs.

¹²⁴ Un ensemble est un conteneur. Il en va de même pour une *pile*, comme une pile d'assiettes, ou une *file*, par exemple la typique file d'attente. Peu importe une pile *de quoi*, l'idée de pile demeure et se modélise.

Pour bien le faire, maintenant, il faut un peu de pratique. Essayez de vous représenter chacune des idées ci-dessous sous une forme objet :

- une table, une chaise, une porte;
- un crayon, un stylo, une efface, une feuille;
- un marteau, un tournevis, une torche;
- un clou, une vis, un vase;
- une roue, un moteur, un pneu;
- un salon, une salle à manger, un atelier de bricolage;
- une voiture, une bicyclette, un autobus, un avion, un train;
- un ordinateur, une télévision, une radio;
- un programme informatique, un processus administratif, un langage;
- un nombre, un chiffre, une idée, un mot; *etc.*

Quels sont leurs attributs? Leurs méthodes? Comment pourrait-on les mettre en relation les uns avec les autres? Qu'ont-ils en commun? Certains peuvent-ils en contenir d'autres? Comment peut-on passer de celui qui *contient* à ce qui *est contenu*¹²⁵?

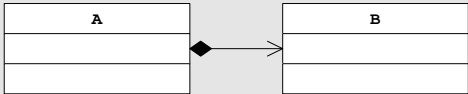
Si possible, essayez de décrire graphiquement les relations que vous aurez identifiées en appliquant la notation UML proposée à droite pour la composition, l'agrégation et l'association.

Ces réflexions vous serviront pour pratiquer. Vous pouvez même vous amuser à les coder en C++, pourquoi pas! Prenez ensuite la peine de discuter de vos solutions avec vos amis et collègues, question de comparer les points de vue et les approches. Cela ne peut être qu'enrichissant.

Plus amusant encore : frottez-vous à ces exercices maintenant et faites-le à nouveau suite à chacun des volumes de cette série de notes de cours. Cela vous permettra de jauger les changements dans vos schèmes de pensée.

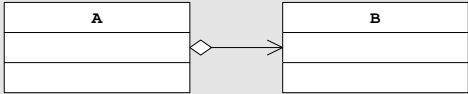
Composition, agrégation et association

La notation UML demande qu'une relation A contient B se dessine sous la forme suivante :

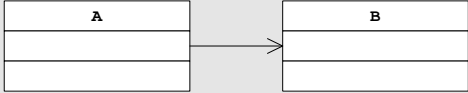


On parle alors de **composition**. Si A contient B par composition, alors la durée de vie de A délimite celle de B.

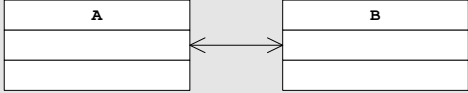
Lorsque A et B ont des existences indépendantes mais A contient B de manière indirecte, comme dans le cas d'une voiture avec lecteur CD amovible, on parle d'**agrégation**. La notation UML devient :



Enfin, lorsque A et B peuvent communiquer mais A ne contient pas B et B ne contient pas A, alors on parle d'**association**. La notation UML pour la relation A peut contacter B est :



ou encore, si la relation est bidirectionnelle (si A peut contacter B et B peut contacter A) :



¹²⁵ Pensez à l'élève et à ses notes, par exemple.

Manières de penser un objet

Lorsqu'on cherche à définir une classe, on peut s'y prendre de trois manières différentes. La première *serait de penser d'abord à ses attributs, puis d'écrire ses méthodes en conséquence*. Dans cette optique, on penserait à un `Rectangle` comme suit :

- on déciderait d'une représentation interne (un point $\{x, y\}$ pour le coin en haut à gauche; un autre pour le coin opposé; peut-être un attribut pour en connaître la couleur; *etc.*); puis
- on définirait les opérations possibles (p. ex. : connaître la largeur du `Rectangle`, peut-être à partir de la différence entre les x de ses deux coins; connaître sa hauteur, de manière analogue; en connaître le périmètre, par la somme de sa largeur et de sa hauteur, le tout multiplié par deux; *etc.*).

La seconde serait de penser d'abord à ses méthodes, puis réfléchir aux attributs requis pour être en mesure de les écrire. Dans ce cas, on penserait plutôt un `Rectangle` comme suit :

- que veut-on faire avec un `Rectangle`? Les réponses possibles seraient l'afficher à l'écran, le déplacer, en connaître la hauteur, la largeur, le périmètre, *etc.*; puis
- quels attributs nous permettraient le mieux d'y arriver? On prendrait à ce moment position entre une représentation utilisant deux points (coins haut/ gauche et bas/ droite) ou une autre utilisant un seul point (haut/ gauche), une largeur et une hauteur. Un facteur pouvant nous influencer serait la réalisation qu'on aurait souvent besoin de la hauteur du `Rectangle`; si c'est le cas, alors il pourrait être plus efficace de l'avoir déjà prête dans un attribut plutôt que d'avoir à la recalculer à chaque fois qu'on la désire.

Dans une optique mettant l'accent sur l'encapsulation et sur l'importance du visage public d'un objet, on comprendra que l'importance des méthodes d'un objet dépasse celle de ses attributs.

La pensée opératoire est difficile à développer, tout comme la pensée par abstraction et la pensée subjective. Ces modes de réflexion sont naturels dans une approche OO mais demandent des efforts particuliers dans une approche structurée plus traditionnelle. Cela explique en partie que tant de gens procèdent plutôt de manière hybride.

On pourra, finalement, y aller avec une approche hybride : *se laisser guider parfois par les données (les attributs), parfois par les méthodes (les opérations)*. Sans que ce soit là la meilleure option, c'est souvent celle qui finit par être suivie, facilitant aux informaticien(ne)s une réflexion libre sur le design en cours.

Remarquez la similitude avec les approches descendante (*Top-Down*) et ascendante (*Bottom-Up*) en programmation structurée. Chacune a ses *aficionados*, mais en pratique, les programmeurs expérimentés mêlent les deux, partant à la fois du haut et du bas, et intègrent au centre...

Ce qu'il est important de retenir, par contre, c'est qu'en général, la structure interne d'un objet, surtout ses données, mais en général tout ce qui en demeurera interne, caché du public, devrait être modifiable au besoin. La face d'un objet qui doit être stable et aussi fixée que possible est celle qui se révèle de manière manifeste : ce qui en est **public**; ce qui constitue son **interface**.

Ce qui est public d'un objet devrait donc presque toujours se limiter à des opérations, à des **méthodes**. C'est autour d'elles qu'un bon design devrait s'articuler. *Dans la plupart des cas*¹²⁶, *les attributs devraient être au service des méthodes*.

¹²⁶ Il y a bien sûr des exceptions à cette maxime. Parfois, un objet peut être conçu en vue de rendre accessible une structure de données particulière, et c'est celle-ci qui guidera alors la pensée. On visera quand même dans ces cas à se donner la latitude requise pour, si le besoin s'en fait sentir, pouvoir remplacer cette structure par une autre.

Penser subjectivement

Développer une classe et rédiger ses méthodes d'instance demande entre autres de *penser sur un mode subjectif*.

En effet, toute méthode d'instance appartient, évidemment, à une instance. Ainsi, dans le petit programme de démonstration ci-dessous, présumant que `uneMethode()` est une méthode d'instance de la classe `UneClasse` :

```

UneClasse instance0, // instance0 est une instance de UneClasse
    instance1; // instance1 est une instance de UneClasse
// On invoque la méthode UneMethode() de l'instance instance0
instance0.uneMethode();
// On invoque la méthode UneMethode() de l'instance instance1
instance1.uneMethode();
// Illégal! uneMethode() est une méthode d'instance, pas une procédure globale
uneMethode();
// Illégal! uneMethode() est une méthode d'instance, pas une méthode de classe!
UneClasse::uneMethode();

```

Sachant implicitement que toute méthode d'instance appartient à une instance en particulier, on adoptera un point de vue subjectif dans la rédaction de sa définition – lorsqu'on rédige la méthode `dessiner()` d'un `Rectangle`, on explique comment dessiner un `Rectangle` en particulier¹²⁷.

```

void p(Rectangle r0, Rectangle r1) {
    r0.dessiner(); // dessinera r0
    r1.dessiner(); // dessinera r1
}

```

Tout `Rectangle`, pour poursuivre avec notre exemple, aura sa propre méthode `dessiner()`. À l'utilisation, le sous-programme appelant cette méthode saura, lui, de quel `Rectangle` il s'agit.

Mais du point de vue de la méthode elle-même, le code rédigé doit fonctionner pour l'instance de `Rectangle` à laquelle appartient la méthode, quel qu'elle soit.

```

void Rectangle::dessiner() const {
    // code pour dessiner ce Rectangle-ci, avec
    // son GetHauteur() et son GetLargeur()
}

```

Dans la définition d'une méthode d'instance, on ne sait pas si le `Rectangle` à dessiner (si `*this` pour la méthode en question) est `r0` ou `r1`; on se met dans une position subjective et on se dessine soi-même¹²⁸.

¹²⁷ Celui qui est propriétaire de cette méthode lors de l'appel.

¹²⁸ C'est très zen de se prendre pour un `Rectangle`...

Choix de stratégie statique

On peut aborder un problème comme celui du choix entre calculer sur demande et calculer sur une base proactive de plusieurs manières. La première est de poser un jugement *a priori* (avant la compilation) sur la stratégie à privilégier :

- à l'aide d'une connaissance du contexte d'utilisation, on peut y aller d'un choix d'expert(e). Si on sait par exemple qu'un `Rectangle` change souvent de dimension, tout en sachant aussi ne pas vouloir souvent connaître sa surface, on prendra position en faveur du calcul de l'aire sur demande, qui sera l'option la moins coûteuse. Si la dimension d'un `Rectangle` tend à demeurer fixe de sa construction à sa destruction, alors calculer l'aire lors d'un changement de dimension devient la chose à faire;
- en retour, si on a des doutes quant à la meilleure des deux options, on pourra plutôt procéder à une batterie de tests dans un contexte le plus près possible de ce qu'on s'attendra à retrouver en réalité, puis tirer des statistiques d'utilisation de ces tests pour ensuite faire en sorte que l'opération la plus sollicitée soit la plus rapide. Ceci demandera l'ajout d'attributs servant à des fins statistiques, mais seulement pour la période de tests (on apportera une mise à jour à la classe une fois la stratégie à privilégier choisie).

Dans les deux cas, la stratégie est statique. À partir de connaissances sur le domaine d'utilisation ou à partir de tests et de simulations réalistes, on définit la forme probable que prendra l'utilisation de l'objet et on optimise les opérations en conséquence de ce savoir. L'encapsulation stricte permet de rédiger des tests *a priori* sans affecter l'interface des objets, donc sans nuire au code client.

Comme dans le cas de toute décision faite à la compilation, il importe qu'elle soit basée sur des données ou des estimés réalistes. Les stratégies statiques sont fixées à la compilation; cela rend leur exécution plus rapide mais réduit leur flexibilité.

Choix de stratégie dynamique

Les deux stratégies ci-dessus présument que le choix d'implantation fait sera valide pour l'ensemble des rectangles éventuellement utilisés, donc fait un postulat de stabilité quant à la nature de l'utilisation de tout `Rectangle`¹²⁹.

Une autre option s'offre dans une approche OO, permettant à tout `Rectangle` de s'adapter dynamiquement à des contextes variables. Cette option est de décider dynamiquement, donc à l'exécution, de la meilleure stratégie à adopter.

Comme pour toutes les options dynamiques, il y a un prix à payer pour implanter celle-ci. En effet, il faut :

- que tout `Rectangle` réserve à l'interne de l'espace pour entreposer l'aire, sans savoir si cet espace sera réellement utilisé ou non¹³⁰; et
- que tout `Rectangle` y aille de (courtes) opérations supplémentaires à chaque demande d'aire et à chaque modification faite à ses dimensions, pour inférer des constats quant à la manière dont il est utilisé.

Chaque méthode servant à modifier la dimension du `Rectangle` ou à en dévoiler l'aire utilisera, dans un contexte de stratégie dynamique, des compteurs internes pour se souvenir du nombre d'appels à chacune de ces méthodes, question de pouvoir éventuellement faire un choix éclairé quant à la stratégie à adopter.

À partir d'un certain seuil de sollicitation, un `Rectangle` pourra décider de tenir à jour l'aire lors de chaque changement de sa taille. Si les changements de taille deviennent plus fréquents, il pourra choisir de revenir à une stratégie de calcul de l'aire sur demande. L'approche objet à l'œuvre, visiblement : l'objet gagne en intelligence et devient responsable de sa propre efficacité.

Les changements dynamiques de stratégie impliquent une certaine surcharge de calcul, et ne sont donc pas raisonnables pour un cas aussi simple que celui du calcul de l'aire du `Rectangle` (qui ne demande, au fond, qu'une multiplication d'entiers; il coûte bien moins cher de calculer cette donnée à chaque fois qu'elle est sollicitée que de décider si un tel calcul serait optimal). En retour, si le calcul potentiellement requis est coûteux, une détermination dynamique de la meilleure option peut effectivement rapporter des dividendes.

¹²⁹ Du moins sous l'angle des moments où l'on interrogera un rectangle quant à son aire.

¹³⁰ L'espace en question sera petit, quelques octets au plus par objet, mais cela représente une augmentation de taille de l'ordre de 20% à 50% pour une classe aussi petite. Imaginez l'impact sur un programme comme une interface personne/ machine qui contient des dizaines de milliers d'instances de `Rectangle`...

Sur la piste des objets intelligents

Les objets deviennent de plus en plus intelligents, de plus en plus capables de réaliser avec sophistication leurs différents mandats. Dans le monde d'aujourd'hui, où on mêle approche OO et approche client/ serveur pour construire des réseaux de composants¹³¹, chaque objet tend à avoir une durée de vie importante, et à être sollicité concurremment par plusieurs entités clientes, chacune susceptible de l'utiliser différemment.

La répartition d'objets dans un réseau mène ces objets à devoir composer avec une plus grande diversité de contextes d'utilisation, ce qui diminue parfois la prévisibilité de ce que serait une bonne stratégie de gestion interne des données. C'est là que l'ajout d'intelligence stratégique à même les objets entre en jeu. L'exemple d'optimisation des accès aux bases de données, suggéré un peu plus haut, n'est pas innocent.

Un objet conçu avec sophistication peut choisir l'implémentation la plus appropriée pour ses propres membres en fonction de l'utilisation qu'on fait de lui. Un `Rectangle` dont on demande fréquemment l'aire ou dont on modifie fréquemment l'aire pourrait choisir dynamiquement, *alors qu'il s'exécute* de maintenir à jour un attribut la représentant directement.

Des objets intelligents sont de plus en plus utilisés dans des systèmes complexes¹³². Pour tirer profit d'un niveau de sophistication aussi élevé, il faut à tout prix restreindre l'accès aux détails d'implémentation, les attributs en particulier, à un passage par chemins contrôlés : des méthodes.

L'encapsulation rigoureuse est la clé derrière la mise au point d'objets intelligents. Un objet qui n'applique pas avec rigueur l'encapsulation ne peut pas prendre conscience de la manière dont on l'utilise, et ne peut conséquemment pas tirer profit de l'intelligence propre aux statistiques d'accès résultantes pour établir lui-même une stratégie de gestion de ses attributs.

Introduire dans un objet une intelligence stratégique peut être une tâche assez complexe; de tels efforts seraient de l'énergie perdue pour un cas comme celui des instances de la classe `Rectangle` et de leur aire. Il reste que des approches d'optimisation dynamique du comportement ne tiennent pas de la science-fiction, et font bel et bien partie de notre réalité contemporaine. Il existe des objets intelligents au sens entendu ici, et il y en aura de plus en plus.

Pour tenir des statistiques d'utilisation à jour dans un objet, on a souvent recours, en C++, à des attributs qualifiés `mutable`. Nous y reviendrons.

¹³¹ Pour en savoir plus, l'idéal est de suivre un cours sur les *Systèmes client/ serveur*.

¹³² Les intéressé(e)s pourront lire sur les `Container` intelligents du modèle CORBA.

Comparatif d'approches OO pragmatiques

Le langage C++ est un outil, un langage OO parmi plusieurs. Il a ses particularités, qu'on les aime ou non, et véhicule un certain nombre de points de vue philosophiques, comme le fait d'ailleurs tout outil, et comme le font implicitement et intrinsèquement tous les langages.

Dans le monde aujourd'hui, trois langages OO, similaires à plusieurs égards mais différents aussi sur quelques points, se partagent la faveur du public, soit C++, Java et C# (considérez VB.NET comme une variante syntaxique de C#, les deux ayant les mêmes limites dû à une infrastructure et à une bibliothèque de classes commune).

Ce ne sont pas les seuls langages OO pragmatiques, et ce ne sont pas nécessairement les meilleurs (peu importe ce que cela veut dire), mais ce sont clairement les plus populaires. En particulier, C# et Java se font compétition dans le même créneau. C# bénéficie de l'appui médiatique considérable de *Microsoft*, et se pose en compétiteur direct de Java, sur le plan philosophique comme sur le plan syntaxique.

La génération et l'exécution des programmes

Sur le plan de la mécanique, Java est un langage qu'on compile non pas pour un ordinateur physique donné mais bien dans un format intermédiaire destiné à être exécuté par un autre programme : une machine virtuelle (*Java Virtual Machine*, en abrégé JVM).

⇒ Exécuter une classe Java signifie lancer une JVM et lui dire d'exécuter la méthode `main()` de cette classe.

L'avantage de cette technique est que tout ordinateur muni d'une JVM peut lire les classes Java, peu importe la machine sur laquelle cette classe fut compilée, et qu'il est plus aisé d'écrire un programme Java de manière à ce qu'il s'exécute convenablement peu importe l'ordinateur et le système d'exploitation que d'écrire un programme C++ qui arrive à un tel résultat.

Le désavantage tient à la rapidité d'exécution : le code devant être lu et exécuté par un autre programme, on ajoute une strate de traitement au moment de l'exécution, ce qui finit évidemment par transparaître¹³³. Notons toutefois que la JVM est accompagnée d'un profileur qui réalise des optimisations supplémentaires pendant qu'un programme s'exécute, ce qui permet des optimisations basées sur la dynamique du programme, optimisations qu'un compilateur axé sur les optimisations statiques comme C++ ne pourrait faire.

¹³³ Les technologies de *Just-in-time Compiling* réduisent en partie ce problème, mais perdent en retour une partie du volet « optimisations dynamiques » résultant du profileur accompagnant la JVM. Les versions récentes de Java (JDK 1.7 au moment d'écrire ces lignes) procèdent aussi à des formes d'optimisation importantes à l'exécution d'une classe, ce qui améliore de beaucoup le portrait. Cela dit, il reste de l'intelligence à appliquer au code au moment où une classe est lancée, ce qui introduit un ralentissement de fait si on compare avec un outil compilé et prêt à l'exécution. Il est à noter que, pour beaucoup d'applications informatiques courantes en entreprise, la différence de vitesse à l'exécution n'est pas suffisante pour poser problème : dans un système interactif où l'interface personne/ machine prédomine, par exemple, le programme risque de passer presque tout son temps à attendre les actions de l'utilisateur de toute manière.

De manière très similaire, les langages .NET tels que C# ou VB.NET sont aussi compilés dans un format intermédiaire, format qui n'est pas destiné à être exécuté directement sur un ordinateur physique (à moins que cet ordinateur ne copie, dans son code machine, le code CIL utilisé par la machine virtuelle .NET) et est compilé à la pièce au moment de l'exécution. Ceci a pour impact que le premier appel à une méthode C# est habituellement lent, puis les appels subséquents sont plus rapides.

Le concept ressemble à celui de Java, à ceci près que la plateforme .NET utilise un standard binaire commun pour tous ses langages et offre à tous et chacun un même environnement protégé, le *Common Language Runtime* (CLR), une machine virtuelle équivalente en principe à une JVM.

⇒ Exécuter une classe C# signifie dire au moteur de la machine virtuelle de la plateforme .NET d'exécuter la méthode `Main()` (avec un `M` majuscule) de cette classe.

La philosophie .NET est de faciliter l'intégration directe de produits .NET entre eux, ce qui offre des avantages non négligeables. Par contre, la plateforme .NET est surtout liée aux produits *Microsoft*, comme *Windows* et *IIS*. Des versions à code ouvert de .NET progressent¹³⁴, incluant des CLR compatibles entre eux et des compilateurs C#, mais sont encore aujourd'hui des joueurs secondaires.

L'avantage principal d'utiliser C# est la facilité avec laquelle il s'intègre avec les modules C++ et VB *compilés pour l'environnement .NET* (le CLR), ce qui diffère par exemple de programmes C++ compilés au sens strict, donc en code indigène pour la plateforme. Notez que le dialecte de C++ qui s'intègre bien au CLR est un dialecte de C++, pas le langage couvert dans ces pages. Le dialecte n'est pas « C++ avec des extensions », comme le clame la publicité, mais bien un langage distinct qui utilise bien les classes .NET mais qui souffre des mêmes limites que C# (pas d'objets constants, pas d'héritage multiple [POOv01], support limité de la programmation générique, *etc.*). Le langage C++/ CLI, comme se nomme ce dialecte, est à bien des égards plus près de C# que de C++ ISO.

Le désavantage du côté rapidité d'exécution demeure le même que pour Java, une machine virtuelle devant procéder à une forme d'interprétation même pour du code presque compilé. Tel qu'indiqué plus haut, le prix à payer est plus grand lors du premier appel à une méthode donnée car le moteur du CLR compile chaque méthode nouvellement rencontrée dans une antémémoire en une version prête à être exécutée, mais C# offre des performances raisonnablement comparables à celles du code indigène produit par un bon compilateur pour les méthodes invoquées fréquemment, dû à son antémémoire.

Dans le cas de Java comme dans le cas de C#, les machines virtuelles sont de grande qualité. Les moteurs des deux plateformes procèdent à un nombre important d'optimisations sophistiquées lorsque les programmes s'exécutent, ce qui offre un certain contrepois au fait que ce modèle implique par définition une perte de temps à l'exécution.

¹³⁴ On pense en particulier au projet *Mono* (<http://go-mono.com/>), une implémentation multiplateforme de la partie standard de .NET, ou à *DotGNU* (<http://www.gnu.org/projects/dotgnu/>). On trouve même un compilateur Java pour Mono, nommé *IKVM* (<http://www.ikvm.net/>), ce qui permet la rédaction de code Java exécutable par un moteur .NET

De son côté, C++ met l'accent sur un fort travail à la compilation et des binaires natifs¹³⁵, bien que rien n'empêche la production d'un interpréteur C++ tel que Cling¹³⁶. Ceci signifie que le code source d'un programme, une fois traduit par le compilateur (et une fois l'édition des liens complétée), est prêt à être exécuté essentiellement sans intermédiaire, mais seulement sur la plateforme pour laquelle la compilation a été réalisée.

Le compilateur C++ peut réaliser beaucoup d'optimisations à même sa lecture des sources, du fait que le langage lui offre une quantité importante d'information à cet effet. On gagne alors nécessairement en rapidité, le programme étant d'ores et déjà prêt pour l'exécution.

Par contre, si le code source (bien écrit) d'un programme C++ peut être migré d'une plateforme à l'autre, le code déjà compilé, lui, est préparé pour un ordinateur et un système d'exploitation bien précis, et migre beaucoup moins aisément.

En retour, en C++, aucun profileur ne réorganise le code alors qu'il s'exécute, alors C++ profite essentiellement des optimisations statiques.

À code équivalent, ce langage demeure nettement plus rapide à l'exécution que les autres, mais pour le développement d'applications conventionnelles (dont font partie la grande majorité des applications), le temps de développement en Java et en C# tend à être plus court, s'appuyant sur la réutilisation de classes couvrant raisonnablement bien la plupart des problèmes de base.

Il y a beaucoup d'autres langages OO, surtout dans l'effervescent monde du développement Web, et la plupart des langages qui ne sont pas *a priori* des langages OO (pensez au très utilisé JavaScript, un langage fonctionnel, pas OO, qui n'a que peu à voir avec Java) ont un volet OO ou intègrent des pratiques facilitant une pensée OO à même leur corpus.

Il serait impossible de les examiner tous, évidemment; si vous devez travailler en Python, JavaScript, Ruby, *etc.*, prenez soin de remarquer les similitudes avec des langages que vous connaissez, mais prenez aussi soin de comprendre les pratiques de ce monde. On ne doit pas imposer les pratiques d'un langage dans le monde d'un autre langage; les résultats sont habituellement décevants.

Il est préférable d'assimiler la pensée de chacun et de réfléchir aux pratiques connues qui s'y intègrent, tout comme aux pratiques de ce monde qui sont mieux adaptées aux métaphores qu'il nous propose.

¹³⁵ Notez quand même que les programmes C++ peuvent être compilés pour des environnements virtuels comme le CLR de .NET. On gagne alors les privilèges de ces environnements, mais on perd la vitesse et la portabilité du produit. C'est un choix qui peut être fait, selon les besoins et les firmes.

¹³⁶ <http://root.cern.ch/drupal/content/cling>

Taille sur disque vs taille en mémoire

Dans le cas de Java comme dans le cas des programmes .NET, les binaires (le code compilé en un format propre à la machine virtuelle) ont tendance à être très petits. Ceci est un effet secondaire du fait que le seul programme s'exécutant en fin de compte est, dans chaque cas, la machine virtuelle. Les programmes Java et .NET, une fois compilés, sont plus des structures de données guidant l'exécution du code que des programmes à part entière.

Ainsi, étant donné les trois programmes ci-dessous, faisant presque exactement le même travail¹³⁷:

Version C++	Version Java	Version C#
<pre>#include <iostream> using namespace std; int main() { cout << "..."; cin.get(); }</pre>	<pre>import java.io.*; public class Z { public static void main (String [] args) { System.out.println("..."); try { System.in.read(); } catch(IOException ioe) { } } }</pre>	<pre>using System; namespace z { public class Z { public static void Main() { Console.WriteLine("..."); Console.ReadLine(); } } }</pre>

La taille des binaires générés va comme suit. Notez que ce comparatif a une valeur illustrative, sans plus, puisque les chiffres indiqués varient beaucoup selon les technologies et les options de compilation. Ici, les tailles ont été obtenues pour une compilation en mode `Debug`, sans aucune optimisation appliquée avec quelque compilateur que ce soit, avec les compilateurs de Microsoft pour C++ et C#¹³⁸ et avec le compilateur de Sun pour Java¹³⁹ :

Version C++	Version Java	Version C#
37376 <i>bytes</i> (36,5 Ko)	560 <i>bytes</i>	4608 <i>bytes</i> (4,50 Ko)

On remarque que le programme C++, une fois compilé, est beaucoup plus gros que les programmes Java et C#, tel que nous l'avons laissé entendre plus haut. La version C# est plus grosse que la version Java, probablement du fait que le binaire C# contient une petite strate véritablement exécutable pour lancer le CLR de .NET alors que le binaire Java procède différemment (la JVM doit être lancée explicitement).

¹³⁷ Java oblige les programmes à attraper les exceptions susceptibles d'être lancées, ce qui explique le bloc `try...catch` autour de la lecture au clavier.

¹³⁸ Les compilateurs livrés avec *Visual Studio .NET 2008 Français* dans les deux cas.

¹³⁹ La version 1.5 du JDK.

En mémoire, toutefois, on remarquera que la situation est bien différente :

Version C++	Version Java	Version C# ¹⁴⁰
352 Ko	6224 Ko	1084 Ko
		3800 Ko

Le chargement obligatoire en mémoire de la machine virtuelle apparaît beaucoup plus clairement dans un contexte d'exécution de programme que lorsque le code compilé repose inerte sur disque. Notez que les machines virtuelles de Java et de .NET sont chargées à nouveau pour chaque programme, ce qui multiplie le coût en mémoire plutôt que de l'amortir (de plus, pour C# avec *Visual Studio .NET 2008*, deux processus sont lancés plutôt qu'un seul).

Il faut remarquer que cette situation est un avantage pour les programmes à l'heure d'Internet : transférer un binaire Java ou C# sur un lien réseau est moins coûteux que ne le serait le transfert d'un binaire C++, et la plupart des ordinateurs clients ont beaucoup de mémoire à leur disposition ce qui diminue l'impact négatif du poids de la machine virtuelle sur les ressources de l'ordinateur sur lequel elle s'exécute.

Il faut aussi relever que les machines virtuelles ont un poids relatif qui diminue alors que les binaires deviennent complexes, du fait qu'elles contiennent au préalable beaucoup d'objets que les programmes sont susceptibles d'utiliser. La taille en mémoire du programme C++, elle, croîtra en proportion avec sa complexité.

Ressemblances et différences syntaxiques

Sur le plan syntaxique, les trois langages se ressemblent beaucoup, et pour cause! La syntaxe de Java étant inspirée en grande partie de celle de C++, ce qui fut fait consciemment pour aider la migration des programmeuses et des programmeurs de l'un vers l'autre, et la syntaxe de C# s'étant inspirée considérablement de celle de Java, les plateformes Java et .NET visant pratiquement la même clientèle.

Il y a des différences, bien sûr, quoique rien d'insurmontable. Quelques-unes sont listées dans le tableau ci-dessous (il y en a plus, mais la plupart des différences manquantes seront plus compréhensibles une fois que la matière des volumes subséquents de cette série aura été couverte et comprise).

¹⁴⁰ Enrober l'appel à `Console.ReadLine()` dans un bloc `try...catch` comme dans le cas du programme Java fait monter la taille du programme en mémoire à 4240 Ko.

C++	C#	Java
Les constantes sont qualifiées <code>const</code> .		Les constantes sont qualifiées <code>final</code> .
Une instance peut être constante, et il en va de même pour une donnée d'un type primitif.	Une instance ne peut être constante; seule une donnée d'un type primitif le peut. Java permet à une référence d'être <code>final</code> , mais pas au référé.	
Il existe des constantes de classe et des constantes d'instance. Les constantes d'instance sont initialisées en préconstruction.	Il existe des constantes de classe seulement.	Il existe des constantes de classe et des constantes d'instance. Les constantes d'instance sont initialisées pendant la construction.
Une méthode d'instance ne permettant pas la modification des membres d'une instance peut (et devrait) être spécifiée <code>const</code> .	Il n'existe pas de mécanisme à même le langage pour permettre à une méthode de garantir l'intégrité de l'objet auquel elle appartient.	
Dans la déclaration d'une classe, tout ce qui suit une spécification comme <code>public</code> est public, jusqu'à la rencontre d'une autre spécification. La même mécanique s'applique pour la spécification <code>private</code> et – même si nous ne l'avons pas encore vu – pour la spécification <code>protected</code> .	Chaque membre doit être spécifié <code>public</code> , <code>protected</code> ou <code>private</code> sur une base individuelle. Il est aussi possible de ne pas qualifier un membre (ci-dessous).	
Par défaut, un membre de classe est <code>private</code> .		Par défaut, un membre de classe est <i>Package Private</i> (une qualification qui ne peut être donnée explicitement).
Les <code>struct</code> sont équivalents aux <code>class</code> à la différence près ¹⁴¹ que la spécification de sécurité par défaut d'un <code>struct</code> est <code>public</code> , pas <code>private</code> .	Les <code>struct</code> sont l'équivalent, en C++, des types instanciés automatiquement, alors que les <code>class</code> sont instanciés dynamiquement (avec <code>new</code>). Le moteur de C# procède à des transformations entre <code>struct</code> en <code>class</code> (en inversement) de manière relativement transparente (cette mécanique s'appelle le <i>Boxing</i>).	Les <code>struct</code> n'existent pas. Depuis la version 5 du JDK, une forme de <i>Boxing</i> automatise le passage de types primitifs à équivalents classe et inversement (p. ex. : de <code>int</code> à <code>Integer</code> et <i>vice versa</i>).

¹⁴¹ Il y a au moins une autre nuance, liée à l'héritage. Voir [POOv01].

C++	C#	Java
Supporte des types primitifs, connus <i>a priori</i> du langage, et des types créés par programmation (<code>class</code> et <code>struct</code> principalement).	Les types « primitifs » sont des <i>alias</i> pour des <code>struct</code> du moteur .NET et ont des méthodes. Les classes ont un support de base équivalent aux classes Java.	Supporte des types primitifs, connus <i>a priori</i> du langage, et des types créés par programmation (<code>class</code> seulement).
Le support offert par le langage aux types primitifs est <i>inférieur</i> à celui offert aux types créés par programmation.	Le support offert par le langage aux types primitifs est <i>différent</i> de celui offert aux types créés par programmation. La différence est moins grande que pour Java, toutefois.	Le support offert par le langage aux types primitifs est <i>différent</i> de celui offert aux types créés par programmation.
On peut surcharger pratiquement tous les opérateurs sur tous les types de données.	On peut surcharger certains opérateurs sur les types de données créés par programmation, mais avec un nombre imposé de contraintes.	La surcharge d’opérateurs est implicite pour des <code>String</code> , et n’est pas supportée pour les autres classes.
L’unité de découpage modulaire de base est l’espace nommé (<code>namespace</code>). Toute déclaration est dans un espace nommé. Il existe un espace nommé par défaut qui est l’espace anonyme. On considère équivalent une déclaration globale et une déclaration dans l’espace anonyme.	L’unité de découpage modulaire de base est l’espace nommé (<code>namespace</code>). Toute déclaration doit être dans un espace explicitement nommé. Il n’existe pas d’espace nommé par défaut ou anonyme.	L’unité de découpage modulaire de base est le paquetage (<code>package</code>). Toute déclaration est dans un paquetage. La relation est directe entre la structure de répertoire dans laquelle se trouvent une classe et le nom de son paquetage.
Un sous-programme dans l’espace nommé anonyme, <code>main()</code> , unique au projet, constitue le point de départ de son exécution.	Il n’existe pas de sous-programme global; on n’a que des classes et des méthodes. Chaque classe peut par contre avoir une méthode de classe nommée... <code>Main()</code> (M majuscule). Si une classe possède une méthode <code>Main()</code> , alors elle peut être exécutée. Exécuter une classe signifie lancer sa méthode <code>Main()</code> .	Il n’existe pas de sous-programme global; on n’a que des classes et des méthodes. Chaque classe peut par contre avoir une méthode de classe nommée... <code>main()</code> . Si une classe possède une méthode <code>main()</code> , alors elle peut être exécutée. Exécuter une classe signifie lancer sa méthode <code>main()</code> .
On peut créer une instance <code>x</code> de la classe <code>X</code> de manière automatique : <code>X x;</code> ou de manière dynamique : <code>X *x=new X;</code>	On ne peut instancier une classe que de manière dynamique. Les instances sont nécessairement manipulées par référence (pas besoin de symbole particulier pour l’indiquer). Les références C# et Java sont analogues aux pointeurs C++ à ceci près qu’il est possible (mais moins idiomatique que par le passé) de réaliser de l’arithmétique sur les pointeurs de C++ (ce que plusieurs considèrent, à juste titre, dangereux). <code>X x=new X();</code>	

C++	C#	Java
Les paramètres peuvent être passés par valeur ou par référence. Le passage par adresse est une variante du passage par valeur (le code client obtient l'adresse d'un objet et en passe une copie au sous-programme appelé). La signature d'un sous-programme guide les modalités du passage de paramètres.	Trois qualifications de paramètres existent : le passage par valeur, qui est le mode par défaut; le passage par référence (<i>ref</i>); et le passage à titre d'extrait pur (<i>out</i>). Le code client et le code appelé doivent tous deux qualifier explicitement et de la même manière (<i>ref</i> , <i>out</i> ou rien du tout) chaque paramètre de chaque sous-programme.	Les paramètres ne peuvent être passés que par valeur. Par contre, les objets étant tous alloués par référence, il est possible de simuler des passages par référence en utilisant des instances de classes qui ne sont pas immuables.
Les paramètres peuvent être constants.	Les paramètres ne peuvent pas être constants.	Les paramètres peuvent être spécifiés <i>final</i> , mais ceci ne sert que très peu puisque tous les paramètres sont passés par valeur, et parce que lorsqu'on parle d'un objet, c'est la référence qui est constante (pas le référé).
La mémoire allouée dynamiquement doit être détruite manuellement. Cette gestion est typiquement automatisée par des pointeurs intelligents [POOv02].	Les plateformes Java et .NET reposent fortement sur la présence d'un moteur de collecte automatique d'ordures.	
La mémoire allouée automatiquement est détruite de manière déterministe, en fin de portée ou lorsqu'une exception sera levée.	Les <i>struct</i> déclarés localement à une méthode sont placés sur la pile et sont libérés automatiquement, mais ne peuvent avoir de destructeurs. Les instances de classes ne sont jamais allouées dynamiquement.	Les données de types primitifs déclarés localement à une méthode sont placées sur la pile et sont libérés automatiquement. Les instances de classes ne sont jamais allouées dynamiquement.
La saine gestion des ressources repose sur la présence de destructeurs déterministes. C++ offre des garanties particulières quant à l'exécution des destructeurs, même dans la plupart des circonstances exceptionnelles.	<p>Bien qu'un moteur de collecte d'ordures existe, il est possible d'écrire des destructeurs (des finisseurs). La syntaxe est la même qu'en C++. Le CLR appelle systématiquement les destructeurs, mais pas à des moments déterministes. Leur apport est donc limité.</p> <p>Les ressources externes au moteur .NET sont essentiellement la responsabilité du code client.</p>	<p>Bien qu'un moteur de collecte d'ordures existe, il est possible d'écrire une méthode <i>finalize()</i> (un finisseur) qui sera <i>peut-être</i> appelée lors de la destruction de l'objet auquel elle appartient. La JVM n'appelle pas rigoureusement <i>finalize()</i>, même quand le programme le lui demande; il n'est donc pas raisonnable de compter sur cette méthode pour des opérations de libération de ressources externes au programme.</p> <p>Les ressources externes à la JVM sont essentiellement la responsabilité du code client.</p>

C++	C#	Java
En C++, une classe est fermée pour modification. Une fois le ; apposé à sa déclaration, il n'est plus possible d'y ajouter des fonctionnalités de l'intérieur. Des opérateurs sous forme de fonctions globales peuvent toutefois continuer à enrichir son interface de l'extérieur.	C# permet ce qu'on y appelle des classes partielles, définies à la pièce dans plusieurs modules, et permet aussi (depuis la version 3.0) d'ajouter des méthodes externes ¹⁴² qui, avec un peu de sucre syntaxique, s'utilisent comme des méthodes d'instance dans le code client.	En Java, une classe est fermée pour modification. Une fois l'accolade fermante de sa déclaration atteinte, il n'est plus possible d'y ajouter des fonctionnalités de l'intérieur.
C++ supporte l'héritage privé, protégé et public [POOv01].	C# supporte l'héritage public. Il est possible d'y utiliser l'héritage protégé ou privé dans certaines circonstances seulement.	Java supporte l'héritage public.
C++ supporte l'héritage multiple, virtuel ou non [POOv01]. L'héritage d'interfaces est, en C++, un cas particulier d'héritage. Les concepts de classe et d'interface forment un tout homogène, et l'emploi d'interfaces strictes y est une technique de programmation, pas un concept.	Java et C# supportent l'héritage simple, et supportent les interfaces comme quelque chose de distinct de l'héritage. On peut implémenter plusieurs interfaces, mais on ne peut dériver que d'un seul parent.	
C++, dans sa forme contemporaine, fait un usage très poussé de la programmation générique [POOv02], applicable à tous les types.	C# (depuis la version 2.0) et Java (depuis la version 1.5) supportent la programmation générique mais seulement sur les classes.	

¹⁴² Des Extension Methods : <http://msdn.microsoft.com/en-us/library/bb383977.aspx>

Comparatifs sur la vie des objets

Quelques remarques comparatives entre C++ et Java sur la vie des objets et sur ce qui gravite autour. Notez que ce qui est indiqué ici pour Java tient aussi pour C#.

Instanciation et libération

En langage Java, toute instanciation doit se faire à l'aide de l'opérateur `new`. Ainsi, présumant l'existence d'une classe `Rectangle`, le code C++ suivant instancie et affiche un `Rectangle` de taille 3×5 :

```
// C++, légal
Rectangle r{3, 5}; // construction paramétrique d'un Rectangle 3x5
r.afficher(); // affichage de ce Rectangle
```

...ou simplement :

```
// C++, légal
Rectangle(3, 5).afficher(); // Rectangle jetable
```

En Java, un tel passage serait illégal. En effet, la déclaration suivante :

```
// Java, légal
Rectangle r; // utiliser r tel quel serait illégal
r.afficher(); // boum! Référence non instanciée!
```

...crée ce qu'on appelle une référence non instanciée, ce qui est l'équivalent d'un pointeur non initialisé en C++. Notons toutefois qu'un compilateur Java notera souvent que le code ci-dessus est illégal. On aurait, plus proprement, pu écrire :

```
// Java, légal
Rectangle r = null; // null en Java équivaut conceptuellement à l'adresse 0 en C++
```

Le code Java correct pour instancier un `Rectangle` de 3×5 puis l'afficher serait :

```
// Java, légal
Rectangle r = new Rectangle(3, 5); // construction paramétrique, Rectangle 3x5
r.afficher(); // affichage de ce Rectangle
```

...ou simplement :

```
// Java, légal
new Rectangle(3, 5).afficher(); // un Rectangle jetable
```

On remarquera la similitude avec le code C++ allouant dynamiquement un `Rectangle` de 3×5 . En C++, parce qu'on peut allouer automatiquement ou dynamiquement, au choix, l'allocation dynamique se fait à travers un pointeur :

```
// C++, légal
Rectangle *p = new Rectangle(3, 5); // construction paramétrique, Rectangle 3x5
p->afficher(); // affichage de ce Rectangle
// ne pas oublier de faire éventuellement delete p
```

C++ permet l'instanciation automatique d'un objet (`Rectangle r;` par exemple) de même que l'instanciation dynamique (p. ex. : `Rectangle *p= new Rectangle;`), alors que Java ne permet que des accès indirects (par référence) aux objets, et ne permet conséquemment que leur instanciation dynamique (`Rectangle r= new Rectangle();`).

En C++, on s'en souviendra, toute mémoire allouée dynamiquement doit être libérée de manière volontaire par le programme demandeur. Ceci complique légèrement la tâche de programmation, demandant une plus grande prudence de la part de l'équipe de développement.

Certains idiomes de programmation, dont RAII [hdIdiom], facilitent toutefois la tâche des programmeuses et des programmeurs.

En Java, un mécanisme de collecte automatique d'ordures (en anglais : *Garbage Collector*; certains disent aussi « ramasse-miettes ») libère périodiquement tout objet auquel plus personne ne réfère. Cela tend à simplifier les programmes (et à diminuer certains types d'erreur), mais peut ralentir les programmes à des moments inopportuns.

C++ 0x introduira un moteur optionnel de collecte automatique d'ordures que les programmeuses et les programmeurs pourront activer ou non s'ils ne veulent pas payer le prix pour l'indéterminisme et les ralentissements impromptus inhérents à cette mécanique. Ce moteur n'invoquera pas les destructeurs, et ne fera donc pas de finalisation; il ne fera que récupérer la mémoire oubliée, sans plus.

Tableaux

Le langage C++ s'inspire du langage C pour ce qui est de sa vision des tableaux en tant que suite contiguë d'objets de même type :

```
// C++, légal
Rectangle r[10]; // déclaration et construction automatique de 10 rectangles par défaut
```

Cette vision d'un tableau comme une séquence contiguë en mémoire d'éléments de même type est une vision axée sur la performance. Accéder à un élément d'un tableau C ou C++ exige une simple opération arithmétique (une addition d'un entier à une adresse) et un déréférencement de l'adresse résultante.

En Java, un tableau est un objet et doit être alloué avec `new`. Il est important de noter qu'instancier un tableau d'objets en Java ne mène pas à instancier les objets dans le tableau : au contraire, chaque entrée du tableau devient alors une référence non instanciée. En C#, la mécanique est la même à quelques nuances de syntaxe près.

```
// Java, légal
Rectangle [] r = null; // r est un objet de type Rectangle[] non instancié
// ...
r = new Rectangle[10]; // instanciation de r... mais prudence, car si le tableau
                       // r existe, aucune instance de Rectangle dans ce tableau
                       // n'a encore été construite!
for (int i = 0; i < 10; i++) {
    r[i] = new Rectangle(); // instanciation du i-ème Rectangle du tableau
}
```

Caractéristique utile des tableaux en Java : étant des objets, ils ont tous des attributs et des méthodes.

Entre autres choses, tout tableau connaît sa propre taille. Cela signifie que le code ci-dessus pourrait être réécrit comme suit.

```
// Java, légal
Rectangle[] r = new Rectangle[10];
for (int i = 0; i < r.length; i++) {
    r[i] = new Rectangle();
}
```

Le code C++ le plus proche de ceci serait sans doute celui proposé à droite. Prudence, par contre : en C++ ISO, il ne faudrait pas négliger de détruire le tableau `T` suite à son utilisation, du fait que ce langage ne repose pas sur un moteur de collecte automatique d'ordures.

```
// C++, légal
using pRect = Rectangle*;
// ...
pRect *r = new pRect[10];
for (int i = 0; i < 10; i++)
    r[i] = new Rectangle;
```

Gestion de la mémoire

Pour gérer la libération de la mémoire allouée dynamiquement, Java tient à jour, de manière transparente au programme, un compteur de références pour chaque objet. Ceci signifie que le moteur de Java sait en tout temps combien de références sont faites à toute chose, donc que chaque opération ajoutant ou retirant une référence à un objet (incluant de vulgaires affectations de références) est un peu plus complexe (et plus lente) qu'il n'y paraît. Le moteur CLR de .NET fait de même pour C#.

Tel que mentionné plus haut, une collecte de déchets périodique est faite par les moteurs de Java et de C#, collecte qui détruit chaque objet auquel plus personne ne réfère. Ceci simplifie la tâche de programmation, mais fait en sorte qu'à l'occasion, du traitement non sollicité (la collecte en question) s'ajoute à celui accompli par le programme en exécution¹⁴³.

Cette imprévisibilité fait en sorte que .NET soit peu utilisé pour les systèmes en temps réel. Java offre une norme temps réel, mais qui a entre autres caractéristiques celle de faire disparaître la collecte automatique d'ordures quand les programmes sont soumis à des contraintes strictes.

Comparatif explicite : une classe, trois langages

Il y a plus de différences que celles énumérées ici¹⁴⁴, mais cela devrait vous donner une bonne idée des principales différences conceptuelles à connaître pour s'y retrouver.

En général, on remarque que la migration des savoirs d'un langage à l'autre se fait relativement bien, quoiqu'il tende à être plus aisé de migrer de C++ à Java que l'inverse¹⁴⁵, Java ayant été conçu après C++ et avec l'idée d'attirer les développeurs C++. Lorsqu'on connaît Java ou C++, utiliser C# est banal à bien des égards.

Les acquis faits ici devraient donc, essentiellement, être transférables pour application dans d'autres langages OO que C++, avec une légère relecture conceptuelle de certaines idées et une adaptation syntaxique plus ou moins grande selon les transferts.

¹⁴³ Pour en savoir plus sur les subtilités de la gestion de la mémoire dans ces langages, voir [hdebSym].

¹⁴⁴ Et remarquez que nous n'avons pas listé ici les similitudes entre les deux langages, celles-ci étant très nombreuses.

¹⁴⁵ Ce qui explique en partie le choix de C++ comme outil dans ce cours.

À titre d'exemple, examinez les classes ci-dessous (notez qu'une classe Java s'écrit en entier dans un seul fichier, portant l'extension `.java`, et que la procédure est la même en C# où les fichiers portent l'extension `.cs`).

On y a enlevé les opérateurs, absents en Java et incomplets en C#, et l'idée de point d'origine pour simplifier le tout. Les similitudes devraient être très visibles à vos yeux.

Similitude ou justice?

Une version antérieure de ce document gardait le code en exemple aussi proche que possible d'un langage à l'autre, s'abstenant de se rapprocher des usages locaux dans chaque cas. Par souci de justice, j'ai décidé d'harmoniser le code Java aux usages sur cette plateforme (noms de méthodes qui débutent par une minuscule, appels croisés d'un constructeur à l'autre en utilisant `this` comme une méthode) et de faire de même avec C# en utilisant des propriétés plutôt que des paires accesseur/ mutateur explicites.

Déclaration de la classe `Rectangle`, version C++ (`Rectangle.h`)

```
#include <iosfwd>
class Rectangle {
    static bool est_hauteur_valide(int) noexcept;
    static bool est_largeur_valide(int) noexcept;
    static const int HAUTEUR_MIN, HAUTEUR_MAX, HAUTEUR_DEFAULT,
                    LARGEUR_MIN, LARGEUR_MAX, LARGEUR_DEFAULT;
    int hauteur_ = HAUTEUR_DEFAULT,
        largeur_ = LARGEUR_DEFAULT;
    static int valider_hauteur(int);
    static int valider_largeur(int);
public:
    class TailleInvalide {};
    int hauteur() const noexcept;
    int largeur() const noexcept;
    int aire() const noexcept;
    int perimetre() const noexcept;
    void dessiner() const noexcept;
    void dessiner(ostream &) const noexcept;
    Rectangle() = default;
    Rectangle(int hauteur, int largeur);
    // Rectangle est un type valeur; la Sainte-Trinité s'applique (les versions générées
    // automatiquement pour la construction par copie, l'affectation et la destruction
    // sont tout à fait convenables)
};
std::ostream & operator<<(std::ostream&, const Rectangle&);
```

Définition de la classe `Rectangle`, version C++ (`Rectangle.cpp`)

```
#include "Rectangle.h"
#include <iostream>
using namespace std;
const int Rectangle::HAUTEUR_MIN = 1, Rectangle::HAUTEUR_MAX = 20,
        Rectangle::HAUTEUR_DEFAULT = Rectangle::HAUTEUR_MIN,
        Rectangle::LARGEUR_MIN = 1, Rectangle::LARGEUR_MAX = 50,
        Rectangle::LARGEUR_DEFAULT = Rectangle::LARGEUR_MIN;
Rectangle::Rectangle(int hauteur, int largeur)
    : largeur_{valider_largeur(largeur)}, hauteur_{valider_hauteur(hauteur)}
{
}
bool Rectangle::est_hauteur_valide(int hauteur) noexcept {
    return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
}
bool Rectangle::est_largeur_valide(int largeur) noexcept {
    return LARGEUR_MIN <= largeur && largeur <= LARGEUR_MAX;
}
int Rectangle::valider_hauteur(int hauteur) {
```



```
    if (!est_hauteur_valide(hauteur)) throw TailleInvalide{};
    return hauteur;
}
int Rectangle::valider_largeur(int largeur) {
    if (!est_largeur_valide(largeur)) throw TailleInvalide{};
    return largeur;
}
int Rectangle::hauteur() const noexcept {
    return hauteur_;
}
int Rectangle::largeur() const noexcept {
    return largeur_;
}
int Rectangle::aire() const noexcept {
    return largeur() * hauteur();
}
int Rectangle::perimetre() const noexcept {
    return (largeur() + hauteur()) * 2;
}
void Rectangle::dessiner(ostream &os) const noexcept {
    for (int i = 0; i < GetHauteur(); i++) {
        for (int j = 0; j < GetLargeur(); j++)
            os << "*";
        os << '\n';
    }
}
void Rectangle::dessiner() const noexcept {
    dessiner(cout);
}
ostream& operator<<(ostream &os, const Rectangle &r) {
    r.dessiner(os);
    return os;
}
```

Déclaration et définition de la classe Rectangle, version Java (Rectangle.java)

```
public class Rectangle {
    public int getHauteur() {
        return hauteur;
    }
    public int getLargeur() {
        return largeur;
    }
    public int getAire() {
        return getHauteur() * getLargeur();
    }
    public int getPerimetre() {
        return (getHauteur() + getLargeur()) * 2;
    }
    private void setHauteur(int hauteur) throws Exception {
        hauteur = validerHauteur(hauteur);
    }
    private void setLargeur(int largeur) throws Exception {
        largeur = validerLargeur(largeur);
    }
    private static int validerHauteur(int hauteur) throws Exception {
        if (!estHauteurValide(hauteur)) {
            throw new Exception("Taille invalide");
        }
        this.hauteur=hauteur;
    }
    private static int validerLargeur(int largeur) throws Exception {
        if (!estLargeurValide(largeur)) {
            throw new Exception("Taille invalide");
        }
        this.largeur=largeur;
    }
    public void dessiner() {
        final int LARGEUR = getLargeur(),
                HAUTEUR = getHauteur();
        for (int i = 0; i < HAUTEUR; i++) {
            for (int j = 0; j < LARGEUR; j++)
                System.out.print("*");
            System.out.println("");
        }
    }
    public static boolean estHauteurValide(final int hauteur) {
        return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
    }
    public static boolean estLargeurValide(final int largeur) {
        return LARGEUR_MIN <= largeur && largeur <= LARGEUR_MAX;
    }
}
```

```

public Rectangle() throws Exception {
    this(LARGEUR_DEFAULT, HAUTEUR_DEFAULT);
}
public Rectangle(int hauteur, int largeur) throws Exception {
    setLargeur(largeur);
    setHauteur(hauteur);
}
protected Rectangle(Rectangle r) throws Exception {
    this(r.getLargeur(), r.getHauteur());
}
private static final int
    HAUTEUR_MIN = 1, HAUTEUR_MAX = 20, HAUTEUR_DEFAULT = HAUTEUR_MIN,
    LARGEUR_MIN = 1, LARGEUR_MAX = 50, LARGEUR_DEFAULT = LARGEUR_MIN;
private int hauteur,
        largeur;
}

```

La version Java n'aurait pas besoin de mention `throws Exception` dans le constructeur par défaut et dans le constructeur de copie (ou, du moins, dans ce qui y ressemble) si l'affectation des valeurs aux attributs y passait par des mutateurs bruts.

Déclaration et définition de la classe `Rectangle`, version C# (`Rectangle.cs`)

```

namespace Formes
{
    class Rectangle
    {
        private const int
            HAUTEUR_MIN = 1, HAUTEUR_MAX = 20, HAUTEUR_DEFAULT = HAUTEUR_MIN,
            LARGEUR_MIN = 1, LARGEUR_MAX = 50, LARGEUR_DEFAULT = LARGEUR_MIN;
        private int hauteur;
        private int largeur;
        public int Hauteur
        {
            get { return hauteur; }
            set { hauteur = ValiderHauteur(value); }
        }
        public int Largeur
        {
            get { return largeur; }
            set { largeur = ValiderLargeur(value); }
        }
        public int Aire
        {
            get { return Largeur * Hauteur; }
        }
        public int Périmètre
        {

```

```
        Get { return 2 * (Largeur + Hauteur); }
    }
    public Rectangle()
        : this(HAUTEUR_DEFAULT, LARGEUR_DEFAULT)
    {
    }
    public Rectangle(int hauteur, int largeur)
    {
        Hauteur = hauteur;
        Largeur = largeur;
    }
    protected Rectangle(Rectangle r)
        : this(r.Hauteur, r.Largeur)
    {
    }
    public static bool EstHauteurValide(int hauteur)
    {
        return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
    }
    public static bool EstLargeurValide(int largeur)
    {
        return LARGEUR_MIN <= largeur && largeur <= LARGEUR_MAX;
    }
    private static int ValiderHauteur(int hauteur)
    {
        if (!EstHauteurValide(hauteur))
            throw new Exception("Taille invalide");
        return hauteur;
    }
    private static int ValiderLargeur(int largeur)
    {
        if (!EstLargeurValide(largeur))
            throw new Exception("Taille invalide");
        return largeur;
    }
    public void Dessiner()
    {
        for (int i = 0; i < Hauteur; i++)
        {
            for (int j = 0; j < Largeur; j++)
                System.Console.Write('*');
            System.Console.WriteLine();
        }
    }
}
```

Visiblement, en C#, privilégier l'usage de propriétés à celui d'une paire accesseur/ mutateur entraîne un nombre important de changements dans le code. Tous les endroits où apparaissent les mots `Hauteur` ou `Largeur` sont en fait des appels d'accesseurs ou de mutateurs enrobés du sucre syntaxique des propriétés.

On peut aimer ou non ce que ça donne à l'usage. La transparence de l'accès à la donnée est plus grand (on ne voit pas dans le code qu'il s'agit d'un appel de méthode), mais cela a l'effet secondaire de ne pas rendre explicite l'application systématique du principe d'encapsulation.

Les programmes ci-dessous arriveront au même résultat en utilisant la classe `Rectangle` dans leur langage respectif.

En C++ (fichier `Demo.cpp`) :

```
#include "Rectangle.h"
int main() {
    Rectangle r{5, 3};
    r.dessiner();
    // ou simplement
    Rectangle{5,3}.dessiner();
}
```

En Java (fichier `Demo.java`) :

```
public class Demo {
    public static void main(String [] args) {
        Rectangle r = new Rectangle(5, 3);
        r.dessiner();
        // ou simplement
        new Rectangle(5,3).dessiner();
    }
}
```

En C# (fichier `Demo.cs`) :

```
using System;
namespace Formes
{
    public class Demo
    {
        public static void Main(string[] args)
        {
            Rectangle r = new Rectangle(5, 3);
            r.Dessiner();
            // ou simplement
            new Rectangle(5,3).Dessiner();
        }
    }
}
```

Appendice 00 – Sémantiques directes et indirectes

La sémantique d'accès aux objets est une chose fondamentale dans tout langage OO. De manière peut-être surprenante, c'est aussi l'un des points sur lesquels les langages OO diffèrent beaucoup les uns des autres.

En particulier, pour les objets, **Java** ne permet que des sémantiques indirectes : on n'y accède aux objets qu'à partir de références, qui sont analogues aux pointeurs du langage C mais auxquels on aurait retiré la possibilité de réaliser des opérations arithmétiques.

De leur côté, les langages **.NET** comme **C#** ou **VB.NET** font une différence entre les entités allouées sur la pile (des `struct` de **.NET**, ce qui inclut les types habituellement primitifs comme le `int` de **C#** qui y est un alias pour `System.Int32`) et pour lesquels plusieurs restrictions existent¹⁴⁶ et les entités allouées sur le tas (les classes), fortement analogues aux classes du langage Java.

La sémantique de valeur avec **C#** n'est typiquement pas l'option privilégiée. Voir [LippValSem] par *Eric Lippert* pour en savoir plus sur le sujet.

Pour un langage reposant sur un moteur de collecte automatique d'ordures, exploiter une sémantique indirecte facilite les choses : il faut nécessairement passer par un tiers pour accéder à un objet, ce qui signifie qu'il est plus facile pour le langage de prendre acte des opérations réalisées. Cela dit, il est possible d'implémenter un tel moteur en **C++** aussi, mais il n'aurait de sens que pour les objets alloués dynamiquement (avec `new`).

En **C++**, deux formes de sémantique indirecte et une forme de sémantique directe sont possibles. La voie royale est la **sémantique directe**, où un objet est défini par son type suivi de son nom et d'indications pour guider sa construction. L'objet existe alors jusqu'à la fin de sa portée (donc jusqu'à l'accolade fermante du bloc dans lequel il est déclaré), ou simplement pour la durée de l'opération en cours s'il s'agit d'un objet temporaire et anonyme (cas simple : pensez à un objet retourné par une fonction).

C++ offre un soutien de première classe à la sémantique d'accès direct aux objets. En ce sens, les objets ont droit au même support que tous les types primitifs du langage. Un objet peut être copié, construit, détruit, peut être constant ou volatile, peut exposer des opérateurs et ainsi de suite, le tout de manière contrôlable par programmation. De ce point de vue, **C++** supporte pleinement l'équivalence opérationnelle des types (en fait, les objets ont droit à un support légèrement supérieur à celui offert aux types primitifs).

On parlera parfois d'un **type valeur** (*Value Type*) lorsqu'on voudra mettre l'accent sur une sémantique directe¹⁴⁷, mais c'est un peu inapproprié comme terme du fait que le choix de la sémantique se fait¹⁴⁸ (en **C++**) sur une base instance, pas sur une base classe : pour une même classe, on peut trouver des instances manipulées directement et indirectement.

¹⁴⁶ En particulier, on a peu de latitude pour ce qui est des constructeurs avec ces types et il est interdit d'en hériter, comme dans le cas des classes qualifiées `final` en Java, `sealed` en **C#** et `NotInheritable` en **VB.NET**.

¹⁴⁷ Normalement, un type valeur offrira un plein support, implicite ou explicite, à la Sainte-Trinité : ses instances pourront être copiées, affectées et détruites « normalement ». Cela dit, le concept est difficile à définir formellement.

¹⁴⁸ Sauf exception, par exemple dans le cas des classes abstraites, mais il s'agit alors de choix de design faits par l'équipe de programmation, pas de contraintes du langage.

C++ supporte aussi l'**accès indirect par référence et par adresse** aux objets (comme à tout autre type). L'accès par adresse de C++ ressemble à l'accès par référence de Java et des langages .NET mais passe par des pointeurs comme en langage C, ce qui permet des manœuvres arithmétiques et demande un peu de prudence (sans parler de la syntaxe qui est un peu inhabituelle pour celles et ceux sans habitude avec C).

L'accès par référence permis par C++ est un accès simple et relativement sécuritaire, qui ne permet ni arithmétique de pointeurs, ni références nulles et qui offre une syntaxe très légère, semblable à la syntaxe des types valeur.

En C++, vous remarquerez que les syntaxes des références et des pointeurs sont similaires. Cette similitude reflète d'ailleurs des similitudes structurelles entre les deux concepts. Cela dit, l'utilisation d'une adresse et d'une référence diffère de plusieurs manières, alors il importe de bien comprendre les différences entre les deux.

Dans une sémantique indirecte (de référence au sens Java ou .NET, ou d'adresse au sens C++), l'affectation est une copie d'adresse. Dans une sémantique de valeur, l'affectation est une copie de contenu.

La sémantique indirecte est utile pour tout ce qui a trait aux prises de décision dynamiques (polymorphisme, abstraction, opacité), que nous examinerons dans [POOv01]. Contrairement à ce que plusieurs pourraient penser, la sémantique directe tend à donner des programmes plus rapides, même si l'objet est copié en entier (pas seulement son adresse). La raison est qu'un objet tend à être utilisé plus qu'il n'est copié et que les accès directs à ses méthodes sont plus rapides que ne le sont les accès indirects. Cependant, prudence : le tableau n'est pas monochrome, et la clé du succès est de choisir avec soin les endroits où une sémantique directe sont préférables et ceux où il est avantageux de procéder de manière indirecte.

Pour la même raison, un objet assemblé par composition plutôt que par agrégation tend à être plus efficace du fait que les accès à ses attributs sont plus directs (donc plus rapides parce que soumis à moins de niveaux d'indirection) que ne le sont ceux faits à travers des abstractions indirectes comme des pointeurs ou des références.

Ce document s'intéresse surtout aux bases syntaxiques et conceptuelles des sémantiques directes indirectes mises à votre disposition en C++. Les divers usages de chaque forme d'accès aux objets seront couverts en long et en large dans les divers volumes de cette série. L'idée ici est d'aider les gens ayant une connaissance de la POO dans d'autres modèles à se familiariser avec la gamme de sémantiques d'accès aux objets de C++.

Pour d'autres détails sur la sémantique de valeur et la sémantique de référence en C++, voir [ValVsRef]. Une discussion des particularités du modèle préconisé par .NET, qui offre des types valeurs sur la pile (bien que ce ne soit qu'un détail) et des types références sur le tas, est disponible sur [NetStack].

La sémantique directe : types valeurs et types concrets

Aussi étrange que cela puisse paraître, c'est la sémantique directe qui surprendra la plupart des programmeuses et des programmeurs ☹. Pourtant, il devrait s'agir du mode d'accès aux objets le plus naturel d'entre tous, mais pour des raisons de tradition (plusieurs langages ☹ visent une conformité conceptuelle avec le modèle mise de l'avant par *Smalltalk*) ou pour des raisons techniques (faciliter le travail de la collecte automatique d'ordures en réduisant la variété des modes d'accès aux objets) plusieurs langages ☹ font le choix conscient de l'escamoter.

La sémantique directe est celle qui s'apparente aux types primitifs :

- définir un objet invoque son constructeur;
- la classe d'une instance donnée est connue. Ceci implique que la sémantique directe n'est pas celle souhaitée pour les invocations polymorphiques [POOv01];
- l'objet a une portée délimitée par le bloc dans lequel il est déclaré; et, surtout
- l'objet expose une sémantique claire et opérationnellement homogène avec celle des types primitifs pour les opérations de copie. Décrit autrement : un objet peut être passé par valeur à un sous-programme, retourné par copie d'un sous-programme et copié de manière naturelle à l'aide des mêmes mécanismes que ceux appliqués aux `int`.

La maxime appliquée à la sémantique directe¹⁴⁹ est *Do as the ints do*. C'est probablement la manière la plus simple d'exprimer le concept de sémantique de valeur, soit la capacité *pour tout type concret* de se comporter, opérationnellement, comme un `int`.

Le type `int` est sans doute le type le plus fréquemment utilisé dans un programme C++, Java ou C# (*le type le plus typique*, pour faire un mauvais jeu de mots) et représente une base sémantique très claire pour exprimer le concept.

Le terme **type concret** est une suggestion de *Bjarne Stroustrup* pour décrire ces types qui jouent un rôle de type plutôt qu'un rôle d'abstraction.

Ce ne sont pas toutes les classes qui sont destinées à représenter des abstractions pour d'autres classes. Bien entendu, il existe une multitude de cas où nous voudrions du polymorphisme, des classes abstraites, des interfaces et autres mécanismes sophistiqués.

En retour, parfois, on veut représenter un nombre, une matrice, une chaîne de caractères, de la monnaie... et il est raisonnable de vouloir manipuler de telles entités avec la même aisance que celle avec laquelle on manipulerait un entier.

La définition de *Bjarne Stroustrup* pour l'idée de sémantique directe est aussi... directe : une sémantique de valeur signifie que l'affectation et la construction par copie font... des copies [StrouSlow].

¹⁴⁹ Cette maxime est attribuée à *Scott Meyers* [EffStl]. Voir aussi *Réflexion 00.1 : Machiavel ou Murphy?* et *Réflexion 00.2 : quelles méthodes exposer?*.

Sémantique indirecte : les adresses

Notez qu'en C++ contemporain, il est très rare en pratique que nous ayons à manipuler des adresses et des pointeurs directement. Un ensemble de classes de pointeurs intelligents sont mis à notre disposition pour formaliser la plupart des sémantiques d'utilisation pertinentes en pratique. Nous y reviendrons dans [POOv02], *Pointeurs intelligents et objets partageables*).

Tout objet est *quelque part*; tout objet se loge à une **adresse**. De même, les instances sont scalaires, donc chacune occupe un espace défini par son type, quel qu'il soit. Ce simple fait est porteur de sens en C++ et peut s'exprimer à même le langage.

Imaginons la variable `i` de type `int` et l'instance `r` de `Rectangle` dans l'exemple à droite. Pour que le programme puisse agir sur `i` ou sur `r`, il importe que chacun de ces objets soit quelque part et occupe un certain espace.

```
// ...
int i = 3;
Rectangle r{3, 5};
// ...
```

L'espace occupé par `i` est au moins gros comme un `int`. L'espace occupé par `r` est au moins gros comme un `Rectangle`. Le « au moins » tient à la latitude qu'a le compilateur d'ajouter des coussins (*Padding*) autour de certaines entités pour que l'alignement en mémoire soit plus efficace en pratique¹⁵⁰. Dans les deux cas, cette notion s'exprime dans le langage par l'opérateur `sizeof`.

Il se trouve que `sizeof(i)==sizeof(int)`, tout comme il se trouve que `sizeof(r)==sizeof(Rectangle)`. Ces réalités jouent plusieurs rôles dans le langage, même si elles ne sont pas importantes pour tous les programmeurs; sachez simplement (pour l'instant) qu'en C++, tout objet occupe un espace d'au moins un octet¹⁵¹, donc que pour tout type `T`, `sizeof(T) != 0`.

Quelle est l'adresse de `i`? Quelle est l'adresse de `r`? Ces questions ont un certain sens, mais elles trouvent leur réponse avec un opérateur du langage : l'**opérateur &**. En effet, l'adresse de `i` s'exprime sous la forme `&i` alors que l'adresse de `r` s'exprime sous la forme `&r`.

Si vous avez de l'expérience en Java ou en C#, il se peut que vous trouviez étrange qu'on puisse vouloir se saisir de l'adresse d'un objet, mais sachez que c'est précisément ce que fait tout programme Java lorsqu'il instancie un objet avec `new` ou lorsqu'il passe une référence sur un objet à une méthode.

Dans l'exemple à droite, `new X()` retourne une référence sur un `X` nouvellement créé, ce qui correspond à retourner son adresse (sans permettre de faire de l'arithmétique dessus); passer `p0` à la méthode `f()` correspond à copier une adresse; récupérer dans `p1` la référence retournée par `f()` correspond à copier une adresse; *etc.*

```
public class X {
    // ...
    public X f(X p) {
        // peu importe
    }
    public X g() {
        X p0 = new X();
        X p1 = f(p0);
        return p1;
    }
    // ...
}
```

¹⁵⁰ Ces coussins sont moins en vogue sur les architectures contemporaines qu'ils ne l'étaient pas le passé, mais ces choses changent si vite...

¹⁵¹ Voir *Pointeurs et tableaux*, plus loin.

En C++, l'héritage du langage C permet une vision des adresses plus près de la réalité effective du concept (plus *primitive*). **L'adresse est un entier correspondant à un lieu logique en mémoire**; par convention, cet entier sera exprimé sous forme hexadécimale, comme le montre le programme suivant :

```
#include <iostream>
int main() {
    using namespace std;
    int a = 3; // a est un entier signé sur 32 bits, de valeur 3
    cout << a << ' ' // contenu de a, donc 3
         << &a << endl; // adresse de a, p. ex. : 0068FDF4
}
```

Afficher l'adresse d'un objet présentera en général à l'écran un gros nombre quelconque. Ce nombre est important, puisqu'il dénote l'endroit en mémoire où se trouve la variable en question, mais ne fait pas beaucoup de sens pour un usager.

Adresses et pointeurs

Une adresse est un concept abstrait, un lieu. Le type le plus abstrait de C++ (et de C) représente d'ailleurs une adresse pure, sans sémantique autre que celle de lieu. Il s'agit du type **void***, qu'on évitera sauf pour des cas (souvent en programmation système) où il est nécessaire d'exprimer l'idée de lieu en la détachant de toute sémantique.

Ce passage par l'abstraction pure est dangereux : avec un **void***, le compilateur ne peut même plus savoir le type de la donnée à l'adresse exprimée et ne peut plus offrir le moindre soutien sémantique aux programmeuses et aux programmeurs.

On évitera donc en pratique de jouer à ce niveau à moins d'en avoir vraiment besoin (mais ça arrive).

L'analogue le plus près en Java et dans les langages .NET est le type `Object` (ou `object`) mais, dans ces langages, l'abstraction la plus haute comporte quand même une certaine dose de sémantique. Une adresse pure est une chose *beaucoup* plus abstraite (et plus *dangereuse*) qu'un objet abstrait, ce dernier étant quand même un objet.

Normalement, un programme n'a pas tant besoin d'adresses que de pointeurs. **Un pointeur est une adresse typée**, donc à laquelle se rattache une sémantique. L'adresse d'un `int` est de type *pointeur de int*; l'adresse d'un `Rectangle` est de type *pointeur de Rectangle*.

Le type pointeur de `int` s'écrit **int***. Le type pointeur de `Rectangle` s'écrit **Rectangle***. Cela explique que le type adresse pure s'écrit `void*` (ou adresse de *quelque chose, peu importe quoi*). Remarquez la présence d'astérisques et d'esperluettes dans la syntaxe des pointeurs et des références en C++ car c'est là que se trouve l'essentiel de la difficulté syntaxique; une fois ces éléments de syntaxe maîtrisés, le reste va habituellement de soi.

Adresses et indirections

Une adresse permet un accès indirect sur un objet. Évidemment, il est fréquent qu'on souhaite faire le chemin inverse et aller chercher le contenu pointé par une adresse. Évidemment, tout comme il est possible d'obtenir l'adresse d'un objet en lui appliquant l'opérateur `&`, il est possible de manipuler des pointeurs en déclarant des variables du type approprié.

Ainsi, examinez le programme à droite, qui utilise des entiers mais n'importe quel type (incluant des classes) aurait fait l'affaire :

- la variable `a` est de type `int` et est initialisée à la valeur 3;
- la variable `p` est de type `int*` (donc pointeur sur un `int`) et n'est pas initialisée. Ne pas initialiser un pointeur est une pratique peu recommandable.
- la variable `b` est de type `int` et n'est pas initialisée. On ne peut donc pas présumer de sa valeur initiale.

Le programme va comme suit :

- on fait pointer `p` vers `a` en déposant dans `p` l'adresse de `a`;
- on incrémente ce vers quoi `p` pointe (donc `a`). **Attention aux parenthèses qui sont importantes dans ce cas dû à la priorité des opérateurs;**
- on dépose dans `b` la valeur pointée par `p` (donc le contenu de `a`); et
- on affiche `b` (ce qui devrait afficher 4).

```
#include <iostream>
int main() {
    using namespace std;
    int a = 3,
        *p,
        b;
    p = &a;
    (*p)++;
    b = *p;
    cout << b << endl;
}
```

Les opérations `p = &a;` et `b = *p;` sont toutes deux représentatives de la relation entre pointeur, adresse et contenu pointé. Passer à travers un pointeur pour aller chercher ce vers quoi il pointe est ce qu'on nomme une **indirection**.

Évidemment, la sémantique est importante lorsqu'on écrit des programmes qui manipulent des concepts aussi fondamentaux que ceux d'adresse et de contenu. Ainsi, l'exemple qui suit montre quelques exemples d'opérations sémantiquement valides et sémantiquement invalides.

```
// a est un short; b est un pointeur de short
short a, *b; // b est un short*; a est un short
int i;
// ce qui suit est invalide. On ne peut déréférencer a qui n'est pas un pointeur
// *a = 5;
// ce qui suit est valide. b est un short*, a est un short donc &a est un short*
b = &a;
// ce qui suit est invalide. b est un pointeur de short et i est un int donc
// &i est l'adresse d'un int. Les pointeurs impliqués ne sont pas du même type
// b = &i;
// ce qui suit est valide. b est un pointeur de short donc *b est un short et
// la valeur 5 peut être affectée à un short (ceci écrit dans a)
*b = 5; // *b est ce vers quoi pointe b
```

Remarquez que les opérations sur des pointeurs doivent respecter les types impliqués. Déposer l'adresse d'un `Rectangle` dans un pointeur sur un `int` est une opération *extrêmement* dangereuse du fait que le compilateur considérera par la suite que ce qui se trouve au bout du pointeur est un `int` et le traitera comme tel.

Cela explique que les opérations pour faire taire le compilateur lors de manipulations risquées de pointeurs (les `reinterpret_cast` ; voir [POOv01]) doivent être réduites au minimum et doivent être documentées de manière rigoureuse et exhaustive.

À quoi s'associe l'astérisque?

Dans une déclaration de variable en C++, l'astérisque s'associe à droite, pas à gauche. Cela signifie que dans la déclaration suivante :

```
short a, *b, c;
```

les variables `a` et `b` ne sont pas du même type. En effet, `a` est un `short`, `b` est un `short*` (un pointeur de `short`) et `c` est un `short`.

Cette syntaxe tend à mélanger certaines programmeuses et certains programmeurs. Si c'est votre cas, alors une saine habitude est de déclarer les variables une ligne à la fois comme le montre l'exemple à droite. La confusion devrait alors disparaître d'elle-même.

```
short a;
short *b;
short c;
```

Comprendre l'astérisque

Lorsqu'un astérisque est placé devant une variable lors de sa déclaration (en général, on entend ici entre le type de la variable et son nom), cela signifie que la variable sera un pointeur.

Lorsque l'astérisque sert de préfixe à un pointeur une fois celui-ci déclaré, cela signifie qu'on parle du contenu pointé par ce pointeur. L'exemple à droite illustre ce fait.

```
// ...
int i,
    *p; // déclaration : p est un int*
p = &i;
// utilisation : on déréférence p
*p = 5;
// ...
```

Le pointeur nul

Puisqu'un pointeur peut pointer à n'importe quelle adresse, il importe d'avoir une convention pour signaler qu'un pointeur ne pointe pas encore vers un lieu légal. Sans une telle convention, il serait impossible de distinguer un pointeur prêt à être utilisé d'un pointeur illégal.

Comme Java et C# avec `null`, VB.NET avec `Nothing` et Pascal avec `Nil`, C++ a son concept de pointeur nul. On l'exprimera simplement par l'adresse zéro ou, depuis C++ 11, par `nullptr`. Il existe un symbole `NULL`, fortement répandu en C, mais il est déconseillé d'y avoir recours en C++ [`hdCppNull`].

Le bon côté de cette approche est que, suivant la tradition C, l'entier `0` est considéré comme faux pour les tests de condition, et il en va de même pour `nullptr` en C++. Ceci implique que tester pour la validité d'un pointeur se fait de manière toute naturelle dans un programme (comme le montre l'exemple à droite).

```
// ...
int *p = nullptr;
// ...
if (p) {
    // p est valide
}
// ...
```

Pointeurs et arithmétique

L'opération `(*p)++` est représentative des risques de la manipulation de pointeurs par des gens inexpérimentés. En effet, les parenthèses font en sorte de prioriser l'indirection puis d'appliquer l'opérateur `++` sur le contenu pointé par `p`. Si l'opération avait été exprimée sous la forme `*p++` (sans parenthèses), et présumant que `p` soit de type `int*`, alors son sens aurait été *expose le contenu pointé par p puis déplace p de l'équivalent de la taille d'un int en mémoire*.

Ainsi, après cette opération, `p` n'aurait plus pointé au même endroit qu'auparavant et tout accès à travers `p` aurait été à risque. Cette capacité de manipuler des adresses directement est ce qu'on nomme de l'**arithmétique de pointeurs**. C'est un outil très puissant, mais aussi très dangereux lorsque placé entre des mains un peu malhabiles.

L'arithmétique de pointeurs est une arithmétique typée. En effet, pour un pointeur `p` donné, l'expression arithmétique `p+i` signifie l'adresse `p` à laquelle on ajoute `i` fois la taille de ce vers quoi pointe `p`. Par exemple, si `p` est de type `int*` et si `sizeof(int)==4`, alors `p+2` signifie `p` plus 8 *bytes*.

L'arithmétique de pointeurs sert à plusieurs saucés. Il est possible d'aller à n'importe quelle adresse en mémoire accessible à un programme et, en s'appuyant sur une conversion explicite de types (un `reinterpret_cast`; voir [POOv01]), d'utiliser ce qui s'y trouve pour nos fins. Ceci permet de manipuler du matériel par programmation dans la mesure où l'adresse logique de ce matériel est connue et accessible au programme. Cela permet aussi d'écrire des outils logiciels très spécifiques comme des gestionnaires de mémoire propres à un programme donné.

Pointeurs et tableaux

Les programmeurs Java et les programmeurs .NET réagissent parfois vivement aux tableaux de C et de C++ du fait que ce sont des structures de données à la fois très efficaces et très primitives. C'est une réaction qui se comprend.

Les tableaux de C++ sont structurellement identiques à ceux du langage C. Par définition, dans ces langages, un tableau d'un type donné n'est qu'une suite contiguë en mémoire d'éléments du même type, et le tableau lui-même n'est qu'un pointeur sur le premier élément de cette séquence. Conséquence logique de cette affirmation : en C et en C++, tout pointeur peut se manipuler comme un tableau.

Un tableau de cinq `int` constitue un bloc de mémoire de `5*sizeof(int)` octets qui se suivent en mémoire sans que rien ne s'intercale entre eux; si la déclaration de ce tableau est exprimée sous la forme `int tab[5];`, alors `tab` est un pointeur sur le premier d'une séquence de cinq `int` contigus en mémoire.

On remarquera sans peine que les mots *bornes* ou *taille* n'apparaissent nulle part dans cette définition. Un tableau ne connaît ni sa taille, ni ses extrémités. En conséquence, l'utilisation imprudente de tableaux est une cause fréquente de dégâts et de bogues sournois. Écrire à l'extérieur d'un tableau (on dit *déborder* d'un tableau), que ce soit juste avant le tableau ou juste après, est une erreur dont les conséquences varient fortement selon le contexte.

Par exemple, si déborder d'un tableau signifie écrire par accident dans la variable située juste après lui en mémoire, il se peut que le programme soit livré sans que le bogue ne soit détecté et que ce bogue revienne hanter la compagnie plusieurs années plus tard.

Examinons le code proposé à droite. Ce petit programme insère des valeurs dans le tableau puis affiche chacune d'entre elles; il s'agit d'un programme tout ce qu'il y a de plus banal.

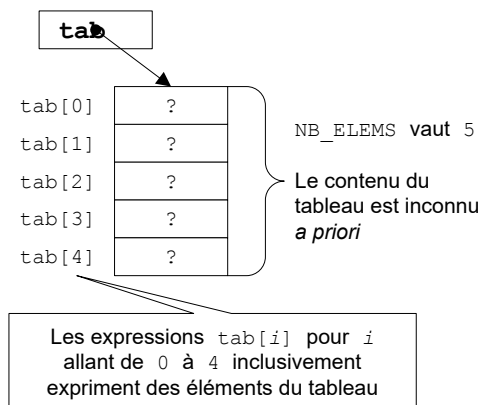
Si nous considérons que `tab` est un pointeur de `int`, alors il nous faut réfléchir un peu pour bien comprendre le sens d'une expression comme `tab[0]`, `tab[1]` ou, de manière plus générale, `tab[i]`.

```
#include <iostream>
int main() {
    using namespace std;
    const int NELEMS = 5;
    int tab[NELEMS];
    for (int i = 0; i < NELEMS; ++i)
        tab[i] = i + 1;
    for (int n : tab)
        cout << n << ' ';
    cout << endl;
}
```

Structurellement, la déclaration du tableau `tab`, `int tab[NELEMS];` résulte en quelque chose de semblable à ce qu'on peut voir ci-dessous.

Chacun des éléments de `tab`, exprimés sous la forme `tab[i]` où `i` est un indice légal pour `tab`, correspond à un endroit en mémoire calculé à partir de l'adresse que représente `tab`.

Concrètement, rien n'empêche un programme d'écrire une valeur dans `tab[NELEMS]`. Cet endroit est peut-être accessible au programme mais il est certain qu'il ne correspond pas à un élément du tableau. Y écrire en pensant écrire dans le tableau constitue donc une faute logique grave et dangereuse.



Remarquez que toutes les cases ont la même taille. Si `tab` est un tableau de `int`, alors chaque case de `tab` occupe un espace de `sizeof(int)` *bytes*. Ceci nous permet d'exprimer certaines relations mathématiques intéressantes :

- la case `tab[0]` se trouve en mémoire à l'adresse `tab` à laquelle on a ajouté `0*sizeof(int)` *bytes*, donc là où pointe `tab`;
- la case `tab[1]` se trouve en mémoire à l'adresse `tab` à laquelle on a ajouté `1*sizeof(int)` *bytes*;
- la case `tab[2]` se trouve en mémoire à l'adresse `tab` à laquelle on a ajouté `2*sizeof(int)` *bytes*; etc.

```
int tab[5];
int *p = tab;
// ici, *p==p[0] et *p==tab[0]
```

Ces relations nous montrent aisément pourquoi les indices de tableaux en C et en C++ débutent à zéro plutôt qu'ailleurs : l'opérateur `[]` appliqué à un pointeur en C et en C++ n'exprime rien de plus qu'une opération arithmétique sur un pointeur, opération suivie d'un *déréférencement* du pointeur résultant.

Souvenons-nous que l'arithmétique de pointeurs est une arithmétique typée. Sachant cela, de manière générale, `&tab[i] == tab+i` et `tab[i] == *(tab+i)`. Ce sont des conséquences directes à la fois de la notation des pointeurs en C et en C++ et de l'arithmétique de pointeurs dans ces deux langages.

Notez qu'un tableau à deux dimensions est un tableau de tableau, donc un pointeur de pointeurs. En pratique, si un tableau `tab` est déclaré `int tab[M][N]`, alors `tab` est de type `int**` (donc *pointeur de int**, ou *pointeur de pointeur de int*). L'idée s'étend à un nombre arbitrairement grand de dimensions.

Les pointeurs, un concept à bannir?

Alexander Stepanov, l'homme derrière STL qui est l'un des fleurons de la bibliothèque standard de C++, a exprimé ce qui suit dans une entrevue dans *Dr Dobb's Journal* [StepDobb]. À mon avis, cet extrait d'entrevue constitue une réflexion extrêmement pertinente quant à la nature des pointeurs et quant à leur relation à la pensée abstraite :

«I learned a lot at Bell Labs by talking to people like Andy Koenig and Bjarne Stroustrup about programming. I realized that C/C++ is an important programming language with some fundamental paradigms that cannot be ignored. In particular I learned that pointers are very good. I don't mean dangling pointers. I don't mean pointers to the stack. But I mean that the general notion of pointer is a powerful tool. The notion of address is universally used. It is incorrectly believed that pointers make our thinking sequential. That is not so. Without some kind of address we cannot describe any parallel algorithm. If you attempt to describe an addition of n numbers in parallel, you cannot do it unless you can talk about the first number being added to the second number, while the third number is added to the fourth number. You need some kind of indexing. You need some kind of address to describe any kind of algorithm, sequential or parallel. The notion of an address or a location is fundamental in our conceptualizing computational processes—algorithms.

Let's consider now why C is a great language. It is commonly believed that C is a hack which was successful because Unix was written in it. I disagree. Over a long period of time computer architectures evolved, not because of some clever people figuring how to evolve architectures—as a matter of fact, clever people were pushing tagged architectures during that period of time—but because of the demands of different programmers to solve real problems. Computers that were able to deal just with numbers evolved into computers with byte-addressable memory, flat address spaces, and pointers. This was a natural evolution reflecting the growing set of problems that people were solving. C, reflecting the genius of Dennis Ritchie, provided a minimal model of the computer that had evolved over 30 years. C was not a quick hack. As computers evolved to handle all kinds of problems, C, being the minimal model of such a computer, became a very powerful language to solve all kinds of problems in different domains very effectively. This is the secret of C's portability: it is the best representation of an abstract computer that we have. Of course, the abstraction is done over the set of real computers, not some imaginary computational devices. Moreover, people could understand the machine model behind C. It is much easier for an average engineer to understand the machine model behind C than the machine model behind Ada or even Scheme. C succeeded because it was doing the right thing, not because of AT&T promoting it or Unix being written with it.

C++ is successful because instead of trying to come up with some machine model invented by just contemplating one's navel, Bjarne started with C and tried to evolve C further, allowing more general programming techniques but within the framework of this machine model. The machine model of C is very simple. You have the memory where things reside. You have pointers to the consecutive elements of the memory. It's very easy to understand. C++ keeps this model, but makes things that reside in the memory more extensive than in the C machine, because C has a limited set of data types. It has structures that allow a sort of an extensible type system, but it does not allow you to define operations on structures. This limits the extensibility of the type system. C++ moved C's machine model much further toward a truly extensible type system. »

Plusieurs (moi le premier) ont grandi en apprenant que les pointeurs, parce qu'ils peuvent être dangereux quand on les utilise mal, représentent « le mal », une tare à bannir. En pratique, les pointeurs nous ont donné à la fois un modèle du concept de séquence (la très puissante théorie des itérateurs [POOv02]) et un modèle conceptuel brut d'un ordinateur. Le prix de la polyvalence est un devoir d'agir de manière responsable.

Visualiser les pointeurs

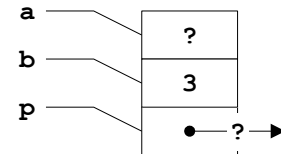
Les pointeurs peuvent être difficiles à visualiser de prime abord, du fait que leur contenu est en fait un lieu, une adresse, plutôt qu'une donnée conventionnelle. Pour s'y retrouver avec les pointeurs, on a souvent recours à des schémas.

Prenons le programme proposé en exemple à droite et examinons schématiquement l'impact de chacune des instructions qui y apparaissent.

```
int main() {
    int a,
        b= 3;
    int *p;
    p= &a;
    *p= 4;
    b+= a;
}
```

Initialement, les trois variables sont déclarées et sont de même taille¹⁵² : **a** et **b** sont tous deux des entiers signés, et **p** représente l'adresse d'un entier.

Les contenus de **a** et de **p** dans ce schéma sont tous deux inconnus (d'où les ?).



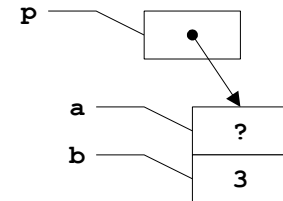
Remarquez la notation de **p**, avec une flèche : cela signifie *l'endroit pointé par p*.

Le fait que **p** soit un **int*** signifie qu'il ne pourra contenir que l'adresse d'objets reconnus comme des **int** par le compilateur.

Une fois que l'opération **p = &a** aura été traitée, nous obtiendrons le schéma suivant.

Bien sûr, **p** n'a pas bougé : cette variable est une variable comme les autres à ceci près que, conceptuellement, son rôle est de représenter l'adresse d'un **int**. En retour, le contenu de **p** est devenu l'adresse de **a** : on dit que **p** pointe vers **a**.

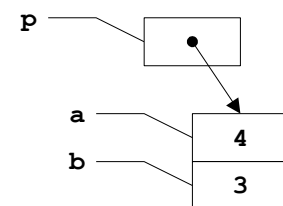
À partir de ce moment, modifier le contenu pointé par **p** signifie modifier le contenu de **a**.



Ainsi, suite à l'opération ***p = 4**, on obtiendra le schéma suivant.

En effet, modifier le contenu pointé par **p** est alors exactement la même chose que modifier **a**, donc ***p = 4** dépose **4** dans **a**.

Il y a de réels bénéfices à utiliser des pointeurs, de même que de considérables dangers. En fait, le mauvais usage de pointeurs a comme résultat les bogues les plus malins, les plus intermittents, les plus difficiles à éliminer.



¹⁵² ...présument que la taille d'un `int` soit égale à la taille d'un pointeur, ce qui est souvent (mais pas toujours) le cas. Le type `int` est réputé être celui sur lequel les opérations sont les plus rapides, donc la plupart des compilateurs utilisent pour le type `int` le type entier dont la taille correspond à celui d'un registre (et d'une adresse) sur l'ordinateur en fonction duquel le code est généré. Cette présomption d'égalité de taille n'est pas nécessaire pour le concept derrière notre discussion mais simplifie les schémas (toutes les petites boîtes ont la même taille) alors je me permets de la faire.

Sémantique indirecte : les références

Une référence est la cousine d'un pointeur. Tout comme le pointeur, la référence représente l'adresse d'un objet (le mot objet étant pris au sens large). Contrairement au pointeur, la référence ne référera qu'à une seule et même chose tout au long de son existence, et devra conséquemment référer à quelque chose dès sa déclaration.

En ceci, une référence C++ diffère des références Java, C# ou VB.NET. En C++, une référence est nécessairement valide; une référence nulle n'y a pas de sens. En Java et en C#, une référence est comme un pointeur en C++, sans la puissante mais dangereuse arithmétique de pointeurs.

Déclarer une référence

On déclare une référence en préfixant à la déclaration son nom d'une esperluette, symbole `&` avec lequel nous sommes familières et familiers depuis que nous avons vu comment obtenir l'adresse d'un objet. Une référence peut être considérée, pour les fins d'un programme, comme étant au même endroit, en mémoire, que ce à quoi elle réfère¹⁵³.

Examinez le code proposé à droite, qui montre comment il est possible de déclarer des références et quel impact celles-ci ont sur un programme.

Notez au passage que la référence `r` est liée à `i` dès sa déclaration. Déclarer une référence sans la lier immédiatement à un objet est illégal en C++. Ainsi, ceci :

```
// illégal: à quoi r réfère-t-elle?
int &r;
```

est absolument illégal.

```
int main() {
    int i = 5; // i est un int de valeur 5
    int *p = 0; // p est un pointeur nul
    int &r = i; // r est une référence a i
    // i devient 6, r aussi, car r réfère à i
    i++;
    // r devient 7, i aussi, car r réfère à i
    r++;
    p = &i; // p pointe maintenant vers i
    (*p)++; // i, r et *p deviennent 8
    // p pointe là où r réfère (p pointe vers i)
    p = &r;
}
```

Un avantage immédiat des références sur les pointeurs est donc qu'une référence est nécessairement valide, du moins au sens où elle réfère nécessairement à quelque chose d'existant. Nul besoin de la valider en tant que référence.

Un autre avantage important des références sur les pointeurs est la légèreté de la syntaxe. Une référence s'utilise comme un objet manipulé directement. En particulier, elle permet accès aux membres à travers l'opérateur `.` (le point), sans nécessiter de *déréférencement* explicite dans le programme. Une syntaxe plus légère signifie souvent moins de bogues.

Il faut toutefois retenir qu'une référence est une indirection. Ce détail aura un impact important en situation de polymorphisme [POOv01].

En retour, utiliser une référence en tant que variable locale, bien que légal, est chose plutôt rare. C'est surtout en tant que paramètres à des sous-programmes et en tant qu'attributs d'objets que les références trouvent leurs lettres de noblesse.

¹⁵³ C'est là un énoncé inexact, mais qui donne une image convenable.

Allocation dynamique de mémoire

L'allocation dynamique de mémoire en C++ requiert une sémantique indirecte (de pointeurs). L'objet alloué dynamiquement est construit sur demande et libéré sur demande (du fait que C++ ISO n'offre pas la collecte automatique d'ordures). Comme nous le verrons dans [POOv03], section *Gestion avancée de la mémoire*, même ces mécanismes peuvent être en grande partie contrôlés par programmation.

Les opérations d'allocation et de libération de la mémoire

Lorsqu'on déclare une variable, on doit en spécifier le **type**, le **nom** (et, dans le cas des tableaux, le **nombre d'éléments**). Le compilateur tire profit de toute cette information pour préparer l'espace en mémoire nécessaire à cette variable durant son utilisation.

Les variables dites **automatiques** sont construites sur la pile d'exécution et détruites à la fin de leur portée, sans que le programme n'ait à expliciter quoi que ce soit. Ainsi, en C++, le moment de l'invocation d'un destructeur est déterministe. On peut compter dessus pour réaliser une gamme impressionnante d'opérations, à un point tel qu'un idiome de programmation d'une grande utilité (RAII, discuté dans *Annexe 03 – Concepts et pratiques du langage C++*) repose sur ce mécanisme.

Lorsqu'on ne connaît pas *a priori* toute l'information sur un objet, comme par exemple dans le cas où un programme a besoin d'un tableau dont la taille dépendra d'une indication donnée par l'utilisateur, on aura recours à une technique nommée **allocation dynamique de mémoire** par laquelle on demandera au moment opportun l'espace mémoire nécessaire à nos fins.

L'opérateur C++ servant à allouer de la mémoire dynamiquement est **new**. C'est de cet opérateur que se sont inspirés Java et les langages .NET pour leurs propres mécanismes de construction.

On lit l'opération ci-dessus comme suit : *p pointe vers un nouveau int*. Une fois l'allocation effectuée, on peut référer à *p* comme on réfère à tout autre pointeur.

On peut aussi initialiser le `int` nouvellement alloué à une valeur (4, disons) lors de son allocation avec `new` en mettant cette valeur initiale entre parenthèses à la suite de la mention du type de données dynamiquement alloué, exactement comme lorsqu'on invoque le constructeur paramétrique d'un objet.

Dans le cas où l'on a recours à de la mémoire allouée dynamiquement, on devient aussi responsable de libérer cette mémoire lorsqu'elle n'est plus requise.

L'opérateur C++ servant à libérer de la mémoire allouée dynamiquement est **delete**. Un exemple d'utilisation serait celui proposé à droite.

```
// allocation dynamique d'un entier
// (valeur initiale inconnue)
int *p = new int;
// ...
```

```
int *p = new int;
*p = 4; // 4 déposé là où pointe p
// ...
```

```
// allocation dynamique d'un entier
// (valeur initiale: 4)
int *p = new int(4);
// ...
```

```
int *p = new int(4);
// ...
*p = 3;
// libération du int pointé par p
delete p;
```

Gestion de la mémoire allouée dynamiquement

D'autres langages (Java et les langages .NET font partie du groupe) ne demandent pas d'un programme qu'il gère lui-même la libération de la mémoire allouée dynamiquement, mais exige qu'il gère manuellement, dans le code client, les autres ressources du programme.

Ces langages utilisent de façon générale une mécanique de *collecte automatique d'ordures* ou *ramasse-miettes* (en anglais : *Garbage Collector*) qui tient le compte du nombre de références à chaque objet alloué dynamiquement. Lorsque le compte tombe à zéro pour un objet donné, celui-ci est identifié comme prêt à être ramassé. Périodiquement, la collecte se fait et nettoie la mémoire des rebuts qui y traînent.

Le concepteur de C++ ISO a fait un choix différent, et demande aux programmeuses et aux programmeurs de gérer elles-mêmes et eux-mêmes la libération de la mémoire allouée dynamiquement. Avoir plus de contrôle sur le mécanisme signifie imposer plus de travail (pour la mémoire) pour les gens qui programment, mais signifie aussi que le programme ne ralentira pas à un moment inopportun dû à l'action d'un nettoyeur non sollicité en arrière-plan.

Allocation dynamique et scalaires

On peut allouer dynamiquement des données de n'importe quel type, qu'il s'agisse d'un type primitif ou d'un type maison.

Notez que l'invocation d'un constructeur par défaut (donc sans paramètres) en C++ ne requiert pas de parenthèses, même avec `new` (ce qui diffère de la syntaxe de Java et de C#).

L'allocation dynamique en C++ est une action volontaire faite par le programme et souhaitée par la programmeuse ou par le programmeur. La portée de l'objet pointé suite à une allocation dynamique est pleinement sous la responsabilité du programme. Sachant cela, l'objet pointé existera jusqu'au moment où l'opérateur `delete` aura été invoqué sur lui.

L'opérateur `delete` est la contrepartie de l'opérateur `new` et sert à invoquer le destructeur de l'objet pointé puis à libérer la mémoire allouée pour l'objet.

```
#include <string>
enum Genre {
    Masculin, Feminin
};
struct Personne {
    std::string Nom;
    short Age;
    Genre Sexe;
};
int main() {
    short *pShort = new short;
    Genre *pSexe = new Genre(Feminin);
    Personne *pPersonne = new Personne;
    // ...
    delete pShort;
    delete pSexe;
    delete pPersonne;
}
```

Allocation dynamique et tableaux

On peut allouer dynamiquement un tableau à une ou à plusieurs dimensions. Du fait que l'allocation est faite pendant l'exécution du programme, la taille du tableau n'a dans ce cas pas à être une constante, bien qu'elle puisse l'être comme dans le cas de `t0` dans l'exemple à droite.

Prenez note qu'un tableau n'est, au fond, qu'un pointeur sur le premier élément d'une séquence contiguë en mémoire d'éléments du même type. Les divers exemples à droite en font foi.

```
#include <iostream>
int main() {
    using std::cin;
    // t0 est un tableau de dix int
    int *t0 = new int[10];
    int n;
    if (cin >> n) {
        // t1 est un tableau de n float
        float *t1 = new float[n];
        // t2 est un tableau de n par 8 short. Notez
        // l'emploi de parenthèses pour forcer t2 à
        // être un pointeur de short[8] plutôt qu'un
        // tableau de 8 short* (question de priorité
        // relative des opérateurs * et [])
        short (*t2)[8] = new short[n][8];
        // ...
    }
}
```

Une fois un tableau créé, le compilateur ne fait plus la différence entre un tableau et tout pointeur du même type. La responsabilité pleine et entière du bon usage d'un tableau repose en totalité sur les programmeuses et sur les programmeurs.

La libération de la mémoire allouée dynamiquement se fait avec `delete[]` pour les tableaux. Notez que `delete` et `delete[]` sont des opérateurs distincts (il en va d'ailleurs de même pour `new` et `new[]`): il faut indiquer au compilateur que c'est un tableau qu'il faut libérer pour assurer la bonne gestion de la mémoire.

```
#include <iostream>
int main() {
    using std::cin;
    int *t0 = new int[10];
    int n;
    if (cin >> n) {
        float *t1 = new float[n];
        short (*t2)[8] = new short[n][8];
        // ...
        delete[] t2;
        delete[] t1;
        delete[] t0;
    }
}
```

Allocation dynamique de mémoire et erreurs

Il peut s'avérer parfois que la mémoire disponible soit insuffisante pour répondre à une demande d'allocation dynamique de mémoire. Dans un tel cas, traditionnellement, `new` renvoyait `0` plutôt que l'adresse d'un bloc correctement alloué.

Un compilateur C++ conforme au standard devrait aujourd'hui lever une exception, la plupart du temps de type `std::bad_alloc` (du fichier d'en-tête `<new>`) dans le cas d'une erreur d'allocation dynamique de mémoire. Ces cas sont devenus tellement rares (et tellement graves) qu'on parle maintenant non pas d'erreurs mais bien d'*exceptions*.

La tradition demandait de valider l'adresse retournée par `new` avant de l'utiliser :

```
// ...
cout << "Combien d'éléments? ";
// pour ne pas essayer d'allouer un nombre négatif d'éléments
if (cin >> n && n > 0) {
    int *p = new int[n];
    if (p) { // si n est gros, des fois... CECI EST DÉSUET
        // on utilise p (de p[0] à p[n - 1])
        delete[] p;
    }
    // remarquez qu'on n'effectue le delete[] qu'à l'intérieur du bloc if; c'est en effet à
    // cet endroit qu'on peut garantir que l'allocation faite par new fut un succès...
}
// ...
```

Aujourd'hui, on ne s'en occuperait plus du tout ou, au mieux (ou au pire), on attraperait l'exception dans le cas où celle-ci serait levée :

<pre>// ... cout << "Combien d'éléments? "; if (cin >> n && n > 0) { int *p = new int[n]; // ... delete[] p; } // ...</pre>	<pre>// ... cout << "Combien d'éléments? "; if (cin >> n && n > 0) { try { int *p = new int[n]; // ... code ne levant pas d'exceptions delete[] p; } catch (bad_alloc &) { // ... } } // ...</pre>
--	---

En C++, appliquer `delete` ou `delete[]` sur un pointeur nul est sans conséquences, ce qui permet de simplifier le code de manière générale et d'en accroître la sécurité.

Examinons l'exemple suivant, où des opérations (ici, ramenées à une invocation de la fonction `f()`) sont exécutées dans le bloc `try`, entre les opérations d'allocation et de libération dynamiques de mémoire.

Ce code est-il sécuritaire lors d'une levée d'exceptions? Pour y répondre, il faut se questionner sur les sources possibles d'exceptions, et sur leurs conséquences dans le programme :

- l'opérateur `new[]` peut lever une exception. Dans ce cas, l'allocation ayant échoué, il n'y aura pas de pertes de ressources; mais
- que dire de la fonction `f()`? N'offrant aucune spécification d'exception, elle peut lever n'importe quoi. Ici, cela signifie que le bloc `catch` pourrait ne pas suffire, en particulier si `f()` lève autre chose qu'un `bad_alloc`;
- cela signifie aussi qu'en plus d'échouer, dû à une exception non gérée, la mémoire associée au tableau `p` sera perdue. Pire : si `p` contenait des objets, ceux-ci ne seraient pas finalisés.

```
// ...
void f(int[], int taille);
// ...
int main() {
    int n;
    if (cin >> n && n > 0)
        try {
            int *p = new int[n];
            f(p,n);
            delete[] p;
        } catch (bad_alloc &) {
            cerr << "Zut\n";
        }
}
```

Quelques ajustements seront donc requis, pour assurer le nettoyage de `p` en tout temps et pour tenir compte de l'étendue des possibles lors d'une invocation de `f()`.

La réorganisation du code mène :

- à la déclaration de `p` aussi localement que possible, mais pas plus (ici : dans le `if`, mais pas dans le `try`, pour que cette variable soit visible au bloc `try` comme aux blocs `catch`) et initialisation à l'adresse nulle;
- à l'ajout d'un bloc `catch-any` suite aux autres blocs `catch` (pour ne pas que le `catch-any`, plus général, ne cache les autres blocs et consomme les exceptions leur étant destinées); et
- au déplacement du `delete[]` à la fin du `if`, pour qu'il soit fait quoi qu'il advienne. Si le `new[]` échoue, alors `p` demeure nul et `delete[] p` sera inoffensif, alors que dans le cas contraire, nous atteindrons inévitablement ce point et le nettoyage sera fait.

```
// ...
void f(int [],int TAILLE);
// ...
int main() {
    int n;
    if (cin >> n && n > 0) {
        int *p = nullptr;
        try {
            p = new int[n];
            f(p,n);
        } catch (bad_alloc &) {
            cerr << "Zut\n";
        } catch (...) {
            cerr << "Flute\n";
        }
        delete [] p;
    }
}
```


Les aléas de la responsabilité

Étant responsables de la mémoire allouée dynamiquement, nous devons écrire nos programmes en conséquence. Le modèle OO emploie une méthode symétrique pour la création d'objets (des **constructeurs** et des **destructeurs**) et ce modèle nous aidera à bien gérer notre mémoire (et, de manière générale, nos ressources).

Cela dit, nous examinerons quand même tout de suite quelques éléments qu'il nous faudra prendre en considération.

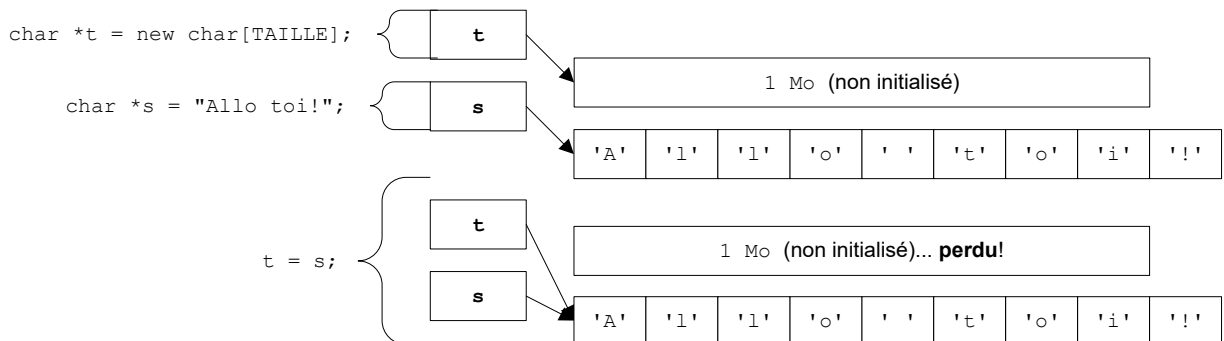
D'abord, **il ne faut jamais perdre de vue la mémoire allouée dynamiquement**. Si nous allouons dynamiquement de la mémoire et s'il nous arrive éventuellement de ne plus avoir de pointeur vers cette mémoire, alors cette mémoire est effectivement perdue. C'est un peu comme si nous avions jeté à la poubelle de la mémoire parfaitement utilisable.

Un exemple d'une telle erreur :

```
const int TAILLE = 1024 * 1024; // un Mo!
char *t = new char[TAILLE];
char *s = "Allo toi!";
t = s; // OUCH!
```

La ligne où l'on affecte `s` (un `char*`) à `t` (un autre `char*`) est valide au sens du compilateur, mais fait en sorte que notre programme ne sait plus du tout où se trouve en réalité le bloc alloué par `t = new char[TAILLE]`. D'une simple instruction anodine, ceci fait perdre à notre programme 1 Mo de mémoire vive!

En effet, schématiquement, ce bout de code nous donne :



Ensuite, il ne faut jamais essayer de libérer de la mémoire n'ayant pas été allouée dynamiquement. L'exemple à droite fera planter votre programme, tout simplement.

```
int *p = nullptr;
int i = 5;
p = &i;
// BOUM! i n'a pas été alloué dynamiquement
delete p;
```

Exemple de type concret : la monnaie

Ce programme montre un type `Monnaie` qui peut être utilisé comme tout type primitif. Le type peut être complété par une gamme d'opérations supplémentaires (en particulier, je vous laisse rédiger l'opérateur `==` qui est plus subtil qu'il n'y paraît).

Déclaration de la classe `Monnaie` (`Monnaie.h`)

```
#ifndef MONNAIE_H
#define MONNAIE_H

// Fichier : Monnaie.h
// Auteur : Patrice Roy
// Date : octobre 2006 (retouché en août 2017)
// Description : type concret simplifié représentant le concept de monnaie canadienne. Une
// version dépendant de la culture locale est possible mais dépasse le sujet de cet article

class Monnaie {
public:
    class SousInvalides {};
private:
    enum { SOUS_MIN = 0, SOUS_MAX = 99 };
    int dollars_ = 0;
    int sous_ = SOUS_MIN;
    static int valider_sous(int sous) {
        if (sous < SOUS_MIN || sous > SOUS_MAX)
            throw SousInvalides();
        return sous;
    }
public:
    Monnaie() = default;
    Monnaie(int dollars) noexcept : dollars_{dollars} {
    }
    Monnaie(int dollars, int sous) : dollars_{dollars}, sous_{valider_sous(sous)} {
    }

    //
    // La Sainte-Trinité est applicable
    //

    int dollars() const noexcept {
        return dollars_;
    }

    int sous() const noexcept {
        return sous_;
    }

    bool operator==(const Monnaie &m) const noexcept {
        return dollars() == m.dollars() && sous() == m.sous();
    }

    bool operator!=(const Monnaie &m) const noexcept {
        return !(*this == m);
    }
}
```

```

bool operator<(const Monnaie &m) const noexcept {
    return dollars() < m.dollars() ||
           dollars() == m.dollars() && sous() < m.sous();
}
bool operator>(const Monnaie &m) const noexcept {
    return m < *this;
}
bool operator<=(const Monnaie &m) const noexcept {
    return !(m < *this);
}
bool operator>=(const Monnaie &m) const noexcept {
    return !(*this < m);
}
Monnaie & operator+=(const Monnaie &m) noexcept {
    int cents = sous() + m.sous();
    if (cents >= SOUS_MAX) {
        cents %= SOUS_MAX;
        ++dollars_;
    }
    sous_ = cents;
    dollars_ += m.dollars();
    return *this;
}
Monnaie & operator==(const Monnaie &m) {
    //
    // Exercice: que faire ici? Comment gérer les différences négatives?
    //
    return *this;
}
};
Monnaie operator+(const Monnaie&, const Monnaie&) noexcept;
Monnaie operator-(const Monnaie&, const Monnaie&) noexcept;
#include <iosfwd>
std::istream& operator>>(std::istream&, Monnaie&);
std::ostream& operator<<(std::ostream&, const Monnaie&);
#endif

```

Définition de la classe **Monnaie** (**Monnaie.cpp**)

```

#include "Monnaie.h"
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
ostream & operator<<(ostream &os, const Monnaie &m) {
    return os << m.dollars() << "," << m.sous();
}
istream & operator>>(istream &is, Monnaie &m) {
    if (!is) return is;
    string montant;
    if (!(is >> montant)) return is;
    stringstream sstr;
    sstr << montant << noskipws;
    int dollars;
    if (!(sstr >> dollars)) return is;
    char c; // consommer la virgule
    if (sstr >> c && c == ',') {
        int sous;
        if (sstr >> sous)
            m = Monnaie{ dollars, sous };
        else
            m = Monnaie{ dollars };
    }
    else
        m = Monnaie{ dollars };
    return is;
}
Monnaie operator+(const Monnaie &m0, const Monnaie &m1) noexcept {
    Monnaie temp{ m0 };
    temp += m1;
    return temp;
}
Monnaie operator-(const Monnaie &m0, const Monnaie &m1) throw() {
    Monnaie temp{ m0 };
    temp -= m1;
    return temp;
}

```

Notez qu'implémenter un opérateur binaire générant une copie (comme l'opérateur `+`) à partir d'un opérateur modifiant l'opérande de gauche (comme l'opérateur `+=`) est un schème fréquemment rencontré. Dans la plupart des cas, cela donne à peu de choses près une implémentation optimale pour ces opérateurs.

Programme de test

```
#include "Monnaie.h"
#include <iostream>
using namespace std;
int main() {
    const Monnaie FORTUNE = 1'000'000; // constructeur paramétrique
    for (Monnaie m; cout << "Entrez un montant: ", cin >> m; ) {
        cout << "Vous avez entré: " << m << endl;
        if (m >= FORTUNE)
            cout << "Mais c'est une fortune!" << endl;
        Monnaie delta(1,1); // petit test
        cout << "En lui ajoutant $" << delta << ", on obtient " << m + delta << endl;
        cout << "En lui enlevant $" << delta << ", on obtient " << m - delta << endl;
    }
}
```

Tel que mentionné précédemment, un type valeur bien implémenté devrait se manipuler aussi simplement qu'on manipule un type primitif. Le type `Monnaie` ici se construit, se détruit, se lit, s'écrit et se manipule, sur le plan opératoire, comme on manipulerait un entier.

Exemple d'allocation dynamique : la pile

Ce programme, simple en soi, pourrait être encore simplifié. Il démontre rapidement une utilisation possible de pointeurs et d'allocation dynamique de mémoire à même le code client.

```
// Fichier : Pile-code-client.cpp
// Auteur : Patrice Roy
// Date : décembre 2003 (retouché en août 2017)
// Description : lit des nombres et les affiche dans l'ordre inverse de la lecture

struct Noeud {
    Noeud *suivant;
    int valeur;
};

#include <iostream>
int main() {
    using namespace std;
    Noeud *pile = nullptr;
    // on lit un entier et on l'ajoute à la pile jusqu'à ce que l'entier lu soit zéro
    cout << "Entrez des entiers, puis zéro (0) pour terminer l'insertion:\n"
    // tester pour les erreurs de lecture et pour la valeur 0
    for (int valeur; cin >> valeur && valeur; ) {
        auto p = new Noeud;
        p->valeur = valeur;
        p->suivant = pile;
        pile = p;
    }
    // On affiche les entiers lus en ordre inverse de celui de leur insertion
    cout << "\nNombres lus (du plus récent au plus ancien, excluant le zéro):\n";
    while (pile) {
        auto p = pile;
        pile = pile->suivant;
        cout << p->valeur << endl;
        delete p;
    }
}
```

Avec une pile présentée sous forme $\circ\circ$, on aurait plutôt ce qui suit. Remarquez principalement les modifications au code client.

Déclaration de la classe `Pile` (`Pile.h`)

```

#ifndef PILE_ENTIERS_H
#define PILE_ENTIERS_H
// Fichier : Pile.h
// Auteur : Patrice Roy
// Date : juillet 2008 (retouché en août 2017)
// Description : représentation OO simple d'une pile d'entiers
class Pile {
    struct Noeud { // classe interne, usage privé seulement
        Noeud *suivant = nullptr;
        int valeur;
        Noeud(int valeur) noexcept : valeur{valeur} {
        }
    };
    Noeud *tete = nullptr;
public:
    // bloquer la copie, par souci de simplicité
    Pile& operator=(const Pile&) = delete;
    Pile(const Pile&) = delete;
    class Vide {}; // exception possible
    Pile() = default; // pile par défaut: vide
    bool est_vide() const noexcept {
        return !tete; // ou return tete == nullptr
    }
    void push(int valeur) {
        auto p = new Noeud(valeur);
        p->suivant_ = tete;
        tete = p;
    }
    int pop() {
        int valeur = top();
        auto p = tete;
        tete = tete->suivant;
        delete p;
        return valeur;
    }
    int top() const {
        if (est_vide()) throw Vide{};
        return tete->valeur;
    }
    void vider() noexcept {
        while (!est_vide())
            pop();
    }
    ~Pile() {
        vider();
    }
}

```

```
};  
#endif
```

Cette classe est suffisamment simple pour ne pas avoir besoin d'un fichier source, bien que rien n'empêche de lui en adjoindre un si le cœur vous en dit. Il aurait été possible de rédiger le code de copie (affectation, construction) d'une `Pile` mais cela aurait alourdi notre exemple (cela dit, si vous en avez envie, ne vous gênez pas!), donc j'ai choisi de les bloquer.

Comme toujours, la Sainte-Trinité (voir *Sainte-Trinité*) s'applique; si une classe se préoccupe explicitement des copies, il est probable qu'elle doit aussi se préoccuper de la destruction.

Toute instance de `Pile` ici est responsable de sa gestion interne (basée ici sur de l'allocation dynamique de mémoire et sur la classe interne `Noeud`). Rien ne transparaît, de l'extérieur, quant à ces choix d'implémentation, mais en retour chaque instance de `Pile` doit s'assurer que ses propres opérations de copie et que sa propre destruction n'entraînent pas d'effets funestes sur le système qui l'aura accueillie.

Programme de test

```
// Fichier : Test-pile.cpp
// Auteur : Patrice Roy
// Date : juillet 2008
// Description : lit des nombres et les affiche dans l'ordre inverse de la lecture
#include "Pile.h"
#include <iostream>
int main() {
    using namespace std;
    Pile p;
    // on lit un entier et on l'ajoute à la pile jusqu'à ce que l'entier lu soit zéro
    cout << "Entrez des entiers, puis zéro (0) pour terminer l'insertion:"
        << endl;
    // tester pour les erreurs de lecture et pour la valeur 0
    for (int valeur; cin >> valeur && valeur; )
        p.push(valeur);
    // On affiche les entiers lus en ordre inverse de celui de leur insertion
    cout << endl << "Nombres lus (du plus récent au plus ancien, excluant le zéro):" << endl;
    while (!p.est_vide())
        cout << p.pop() << endl;
}
```

Toute la logique de gestion de la pile a été évacuée du code client. Seules les opérations pertinentes, soit ajouter (`push()`), retirer et consulter (`pop()`) et vérifier si la pile est vide (`est_vide()`) ont été conservées.

Notez que la `Pile`, telle que vue par le client, ne demande aucune gestion de ressources compliquée. Cette classe est responsable de ses ressources; ce faisant, le code client est simple et va droit au but, comme il se doit. Bien appliquée, visiblement, l'approche OO mène à de meilleurs programmes.

Exemple d'allocation dynamique : l'arbre binaire

Un autre exemple, plus raffiné cette fois, exploitant à la fois pointeurs et mécanismes d'allocation dynamique de mémoire.

```
// Fichier : Arbre-binaire-code-client.cpp
// Auteur : Vincent Echelard
// Date : juin 2004
// Description : lit des nombres et les affiche en ordre croissant.

struct Feuille { // contient l'info. et les liens vers les autres « feuilles » de l'arbre
    Feuille *gauche = nullptr;
    Feuille *droite = nullptr;
    int valeur;
    Feuille(int valeur) : valeur{valeur} {
    }
};

// création d'un synonyme pour des raisons syntaxiques
using PtrFeuille = Feuille*;

// Prototype des fonctions
void DeroulerArbre(PtrFeuille noeud);
void InsérerFeuille(PtrFeuille& noeud, int nombre);
#include <iostream>

int main() {
    using namespace std;
    PtrFeuille racine = {}; // initialisation (pointeur nul)
    // on lit un entier et on l'ajoute dans l'arbre jusqu'à ce que l'entier lu soit zéro
    cout << "Entrez des entiers, puis zéro (0) pour terminer l'insertion:\n";
    // tester pour les erreurs de lecture et pour la valeur 0
    for (int valeur; cin >> valeur && valeur; ) // insérer valeur dans l'arbre
        InsérerFeuille(racine, valeur);
    // On affiche les entiers lus en ordre croissant
    cout << endl << "Nombres lus (en ordre croissant, excluant le zéro):" << endl;
    DeroulerArbre(racine); // on appelle la fonction de fouille
}
```

```
// Rôle : déroule l'arbre à partir d'une « feuille » passée en paramètre.
//      Si on lui passe la « racine », on déroule donc l'arbre au complet.
//      Accessoirement, cette fonction libère la mémoire de l'arbre.
// Intranant : Une « feuille »
// Extranant : -

void DeroulerArbre(PtrFeuille noeud) {
    if (noeud) { // Si le noeud est utilisé
        DeroulerArbre(noeud->gauche);
        cout << noeud->valeur << endl;
        DeroulerArbre(noeud->droite);
        delete noeud;
    }
}

// Rôle : Insère une feuille dans l'arbre.
// Intranant : Une «feuille» à partir de laquelle on veut insérer.
//      La valeur à insérer.
// Extranant : -

void InsérerFeuille(PtrFeuille& noeud, int nombre) {
    if (noeud == nullptr) // Si le noeud est libre
        noeud = new Feuille(nombre);
    else { // Si le noeud est utilisé
        // Si le nombre à insérer est < que la valeur courante
        if (nombre < noeud->valeur) {
            // on insère dans la branche de gauche
            InsérerFeuille(noeud->gauche, nombre);
        } else { // Si le nombre à insérer est >= à la valeur courante
            // on insère dans la branche de droite
            InsérerFeuille(noeud->droite, nombre);
        }
    }
}
}
```

Avec un arbre binaire présenté sous forme $\circ\circ$, on aurait plutôt ce qui suit. Remarquez principalement les modifications au code client.

Déclaration de la classe `Arbre` (`Arbre.h`)

```
#ifndef ARBRE_BINAIRE_ENTIERS_H
#define ARBRE_BINAIRE_ENTIERS_H
// Fichier : Arbre.h
// Auteur : Patrice Roy
// Date : juillet 2008 (retouché en août 2017)
// Description : représentation OO simple d'un arbre binaire entreposant des entiers
#include <iosfwd>
class Arbre {
    struct Noeud { // classe interne, usage privé seulement
        Feuille *gauche = nullptr;
        Feuille *droite = nullptr;
        int valeur;
        Noeud(int valeur) noexcept : valeur{valeur} {
        }
    };
    using PtrNoeud = Noeud*;
    PtrNoeud racine = nullptr;
public:
    // bloquer la copie, par souci de simplicité
    Arbre& operator=(const Arbre&) = delete;
    Arbre(const Arbre&) = delete;
private:
    void DetruireBranche(PtrNoeud &p) noexcept {
        if (p) {
            DetruireBranche(p->gauche);
            DetruireBranche(p->droite);
            delete p;
            p = nullptr;
        }
    }
    // Rôle : déroule l'arbre à partir d'un Noeud passé en paramètre.
    // Si on lui passe la racine, on déroule donc l'arbre au complet.
    // Intrants : Un Noeud
    // Le flux sur lequel afficher
    // Extrait : -
    void DeroulerBranche(PtrNoeud, std::ostream&);
    InsérerNoeud(PtrNoeud &p, int valeur) {
        if (p == nullptr) // Si le noeud est libre
            p = new Noeud{valeur};
        else if (valeur < p->valeur)
            InsérerNoeud(p->gauche, valeur);
        else
            InsérerNoeud(p->droite, valeur);
    }
};
```

```

    }
public:
    bool est_vide() const noexcept {
        return !racine;
    }
    Arbre() = default; // par défaut, l'arbre est vide
    void Insérer(int valeur) {
        InsérerNoeud(racine, valeur);
    }
    void vider() noexcept {
        DétruireBranche(racine);
    }
    ~Arbre() {
        vider();
    }
    void dérouler(std::ostream &os) {
        DéroulerBranche(racine, os);
    }
};

```

Pour éviter d'imposer le code de `<iostream>` aux clients, la méthode `dérouler()` dans sa version la plus générale a été placée dans un fichier source (ci-dessous). Il aurait été possible de rédiger le code de copie (affectation, construction) d'une `Pile` mais cela aurait alourdi notre exemple (cela dit, si vous en avez envie, ne vous gênez pas!), donc j'ai choisi de les bloquer.

Comme toujours, la Sainte-Trinité s'applique; si une classe doit se préoccuper des copies, elle doit aussi se préoccuper de la destruction.

Toute instance d'`Arbre` ici est responsable de sa gestion interne (basée ici sur de l'allocation dynamique de mémoire et sur la classe interne `Noeud`). Rien ne transparaît, de l'extérieur, quant à ces choix d'implémentation.

Définition de la classe `Arbre` (`Arbre.cpp`)

```

// Fichier : Arbre.cpp
// Auteur : Patrice Roy
// Date : juillet 2008
// Description : représentation OO simple d'un arbre binaire entreposant des entiers
#include "Arbre.h"
#include <iostream>
using namespace std;

// Rôle : déroule l'arbre à partir d'un Noeud passé en paramètre.
//      Si on lui passe la racine, on déroule donc l'arbre au complet.
// Intrants : Un Noeud
//           Le flux sur lequel afficher
// Extrait : -

void Arbre::dérouler(PtrNoeud p, ostream &os) {
    if (p) {
        Dérouler(p->gauche, os);
        os << p->valeur << endl;
    }
}

```

```

    Derouler(p->droite, os);
}
}

```

Notez que, si nous la généralisons un peu, la méthode `Arbre::derouler()` serait une belle implémentation du schéma de conception *Visiteur*.

Programme de test

```

// Fichier : Test-arbre.cpp
// Auteur : Patrice Roy
// Date : juillet 2008
// Description : lit des nombres et les affiche en ordre croissant.
#include "Arbre.h"
#include <iostream>
int main() {
    using namespace std;
    Arbre a; // vide
    // on lit un entier et on l'ajoute dans l'arbre jusqu'à ce que l'entier lu soit zéro
    cout << "Entrez des entiers, puis zéro (0) pour terminer l'insertion:\n";
    // tester pour les erreurs de lecture et pour la valeur 0
    for (int valeur; cin >> valeur && valeur; ) // insérer valeur dans l'arbre
        a.Inserer(valeur);
    // On affiche les entiers lus en ordre croissant
    cout << "\nNombres lus (en ordre croissant, excluant le zéro):\n";
    a.derouler(cout); // on appelle la fonction de fouille
} // le destructeur de l'arbre nettoie le tout

```

Le code client bénéficie beaucoup de la simplification obtenue par une saine implémentation de l'arbre à l'aide d'une classe. Remarquez que le client n'a plus à manipuler le moindre pointeur, et n'a plus à gérer la mémoire associée à l'arbre par lui-même. Quoiqu'il arrive, ce programme ne laissera pas fuir de mémoire.

Appendice 01 – Penser un modèle OO

Il est facile de tomber dans les lieux communs, dans les recettes, quand on cherche à définir des composants logiciels selon une approche OO (ou autrement). Le réflexe de la plupart des informaticien(ne)s face à un problème est de produire rapidement quelque chose d'opérationnel, et ce réflexe est somme toute assez sain.

Pour la plupart des gens, produire suivant des recettes connues est une approche confortable et qui donne une certaine assurance de succès. Les recettes peuvent être sophistiquées (on pense aux schémas de conception, ou *Design Patterns*, qui documentent et guident les démarches de développement contemporaines) ou simples (encapsuler de manière primitive les attributs d'instance par une paire accesseur/ mutateur).

Cependant, un modèle OO est souvent sujet à survivre au passage du temps; il est risqué de se laisser aller à des recettes trop simples quand il y a une possibilité bien réelle qu'on doive par la suite entretenir le résultat sur une longue période. Le réflexe de la simplicité peut nous faire oublier qu'un objet est une entité basée sur les opérations, les services qu'elle offre au monde. Travailler systématiquement avec des triples {attribut, accesseur, mutateur} donne une impression de sécurité et de progrès mais, en nous menant à penser aux attributs plutôt qu'aux opérations, ne mène pas nécessairement à un bon design.

Cette section propose une démarche de rédaction de petit programme OO simple et mettant en relief certaines approches et certaines questions de design se situant à notre portée à ce stade de notre progression dans le monde OO.

Nous ne pourrions pas aborder ici des considérations de design trop complexes. Le chemin que nous avons parcouru n'est pas suffisant encore et notre coffre à outils n'est pas assez fourni pour nous permettre de construire quelque chose de véritablement riche. Nous pouvons toutefois créer un petit programme simple et en profiter pour soulever certaines questions déjà à notre portée.

Un exemple visuel : la fourmi affamée

Imaginons un jeu très (très!) simple où une fourmi affamée doit trouver un point de nourriture (de la *bouffe*). Notre fourmi se présentera à l'écran comme un petit symbole directionnel, soit > si elle fait face à l'est, A si elle va vers le nord, < si elle va vers l'ouest ou V si elle va au sud.

Nous ne ferons qu'aborder en surface ce petit jeu. Sentez-vous bien libres d'explorer par vous-mêmes et d'ajouter des éléments supplémentaires (niveaux de difficulté, obstacles, temps limite, etc.)

Les contrôles du jeu seront simples : la touche W permettra d'avancer, la touche A permettra de pivoter vers la gauche et la touche D permettra de pivoter vers la droite. La touche Q permettra de quitter. Ceci s'apparente aux contrôles qu'on peut rencontrer dans plusieurs jeux vidéo commerciaux.

Design par ébauche de code

Nous proposerons un cycle de jeu qui ira comme suit : afficher l'état du jeu et afficher un menu, puis lire la prochaine action de la fourmi jusqu'à ce que le jeu soit terminé. Le jeu se terminera lorsque l'utilisateur signalera son souhait de quitter ou lorsque la fourmi aura rejoint la nourriture tant convoitée.

Il y a déjà plusieurs approches possibles pour aborder ce problème¹⁵⁴. L'une d'elles est de penser d'abord à une ébauche de programme principal possible pour piloter le cycle de jeu. Comme toute approche au design, celle-ci donne de meilleurs résultats avec l'expérience.

Du point de vue d'un tel programme, les objets importants pourraient être le jeu, le menu et les commandes de l'utilisateur.

Une écriture possible pour ce programme est proposée à droite. Proposer du code client avant d'écrire les objets est une manière répandue de penser un design, qu'il soit OO ou non : après tout, si le code client jugé idéal est possible, le design risque d'être bon.

Déjà, un tel programme nous donne plusieurs indications quant à ce que nous devons faire (et ne pas faire) dans notre système.

Notez que nous aurions pu faire un peu mieux que ce qui est proposé ici, sur le plan de l'abstraction. Si vous êtes curieuse ou curieux, discutez-en avec votre chic prof.

```
#include "Commande.h"
#include "Jeu.h"
#include "Menu.h"
int main() {
    Jeu jeu;
    jeu.afficher_etat();
    Menu::afficher();
    for (Commande c = Menu::lire_commande();
        c != Menu::QUITTER && !jeu.fini();
        c = Menu::lire_commande()) {
        jeu.executer(c);
        jeu.afficher_etat();
        Menu::afficher();
    }
}
```

¹⁵⁴ Un programme, même un jeu, peut être vu comme une réponse possible à un problème concret.

- nous voudrions concevoir une représentation, peut-être `OO`, d'une commande. Nous pourrions évaluer nos besoins plus tard (au pire, le type `Commande` pourra être un simple alias pour un type entier);
- il faudra que la classe `Menu` ait une constante comparable, par les opérateurs `==` et `!=`, avec une `Commande`. Il se peut même que cette constante soit elle-même une instance de `Commande`;
- les méthodes de `Menu` ne semblent pas impliquer d'états particuliers; ce sont apparemment de purs services. Nous pourrions probablement les implémenter sous forme de méthodes de classe;
- les méthodes de `Jeu`, elles, risquent d'impliquer des états à tenir à jour (la fourmi se déplacera, il faudra savoir où se trouve la nourriture et il faudra savoir si la fourmi a atteint son but).

Dans le cas de la `Commande`, nous chercherons à identifier en cours de route l'étendue de nos besoins. En donnant un nom à ce type, nous nous permettons d'y réfléchir en soi lorsque le moment de le faire se présentera; nous évitons de nous peindre dans un coin.

Si nous constatons qu'il est important que chaque instance de `Menu` ait un savoir (un état) qui lui est propre, alors nous penserons à instancier cette classe et à lui définir des attributs d'instance.

Pour le moment, l'idée de `Menu` dans ce petit problème semble surtout être un cas de regroupement sous un même nom d'information et de comportements. Les besoins pourraient être différents pour d'autres programmes et une classe `Menu` bien différente pourrait être envisagée. En retour, les méthodes de `Jeu` impliqueront des changements d'états. Dans ce programme, nous voudrions donc clairement instancier `Jeu` sans nécessairement instancier `Menu`.

Remarquez que le programme principal ne parle pas du tout de fourmi ou de nourriture. C'est là une vertu, pas un défaut.

Autres constat : la première chose que fait vraiment le programme est d'afficher l'état du jeu, ce qui implique qu'une instance de `Jeu` doive, dès sa construction, être dans un état à la fois valide et présentable. Le principe même du constructeur est mis en relief ici : une instance donnée doit être dans un état connu (et correct!) dès la fin de sa construction.

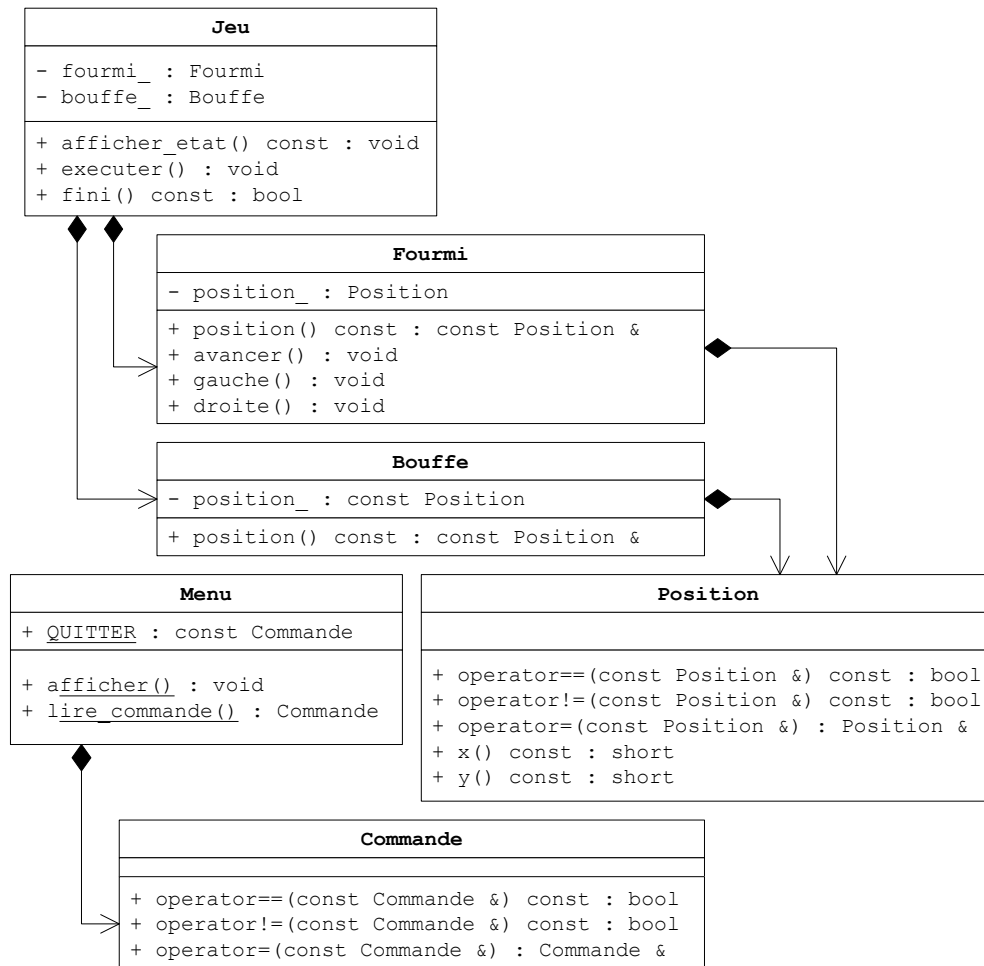
Design par diagrammes

Une autre stratégie pour penser le programme que nous nous apprêtons à rédiger est de concevoir un diagramme de classes montrant les relations entre les objets auxquels que nous planifions avoir recours.

Quelques relations sont évidentes :

- une instance de la classe `Jeu` devra connaître la nourriture et la fourmi;
- la fourmi aura une position;
- la nourriture aura aussi une position;
- il faudra pouvoir comparer deux positions pour savoir si elles sont identiques ou non (dans le but de savoir si la fourmi a atteint son but);
- il faudra pouvoir connaître la position de la fourmi;
- il faudra pouvoir connaître la position de la nourriture;
- la fourmi devra pouvoir se déplacer;
- la fourmi devra aussi pouvoir pivoter vers la gauche ou vers la droite;
- le jeu devra pouvoir nous indiquer s'il est terminé ou non;
- il pourra être utile de définir des accesseurs pour les composants `X` et `Y` d'une position pour faciliter l'affichage par le jeu.

Un diagramme de classes embryonnaire décrivant ces quelques relations pourrait ressembler à celui-ci-dessous.



Ce diagramme est incomplet mais il pourra nous guider dans la construction du code. Notez que nous nous sommes abstenus d'insérer toute proposition de code qui ne semble pas essentielle à notre propos : pas de mutateurs, quelques accesseurs mais pas de manière systématique, à peu près pas d'attributs... Un objet se pense mieux de manière opératoire (par ses méthodes, par ses services) que de manière structurelle.

Ébauches de classes

Nous pouvons d'ores et déjà concevoir quelques classes minimalistes. Les meilleures candidates à ce titre sont les classes qui ont le moins de dépendances envers d'autres classes. Notre diagramme de classes est éclairant en ce sens.

Classe *Position*

Ici, la meilleure candidate sera probablement la classe `Position`, très élémentaire, qui pourrait avoir l'air de celle proposée à droite.

Notez que les opérations de la Sainte-Trinité (voir *Annexe 03 – Concepts et pratiques du langage C++*) n'ont pas été explicitées puisque ceux générés automatiquement par le compilateur conviennent à nos fins.

Vous remarquerez sans doute que nous n'avons pas défini de mutateurs publics. Cela implique que modifier une instance de `Position` existante ne pourra se faire, avec le code proposé dans cet exemple, qu'à travers l'opérateur d'affectation. La copie d'une `Position` étant banale et à coût à peu près nul, cette stratégie est très acceptable.

Remarquez l'expression de `!=` à partir de `==`. Ceci permet de n'entretenir qu'une seule des deux méthodes et réduit les risques de dérive d'une définition par rapport à l'autre.

Pour réfléchir

Notre classe `Position`, telle qu'exposée, laisse transparaître un problème, soit le domaine de validité d'une `Position` : est-ce que toutes les positions envisageables sont valides?

Une réponse possible à cette question serait oui, bien entendu, et c'est ce que considère la classe `Position` présentement. Si nous souhaitons des contraintes, il nous faudra toutefois mettre en place un référentiel et des règles de validité. Nous y reviendrons plus tard, car d'autres questions accompagneront notre réflexion.

```
#ifndef POSITION_H
#define POSITION_H
class Position {
    short x_ = 0, y_ = 0;
public:
    Position() = default;
    Position(short x, short y) noexcept
        : x_{x}, y_{y} {
    }
    short x() const noexcept {
        return x_;
    }
    short y() const noexcept {
        return y_;
    }
    bool operator==(const Position &p)
        const noexcept {
        return x() == p.x() &&
            x() == p.y();
    }
    bool operator!=(const Position &p)
        const noexcept {
        return !(*this == p);
    }
};
#endif
```

Question typique : est-ce que la validité d'une `Position` dépend de `Position` ou du domaine dans lequel elle est utilisée? Plus clairement : peut-on exprimer une classe `Position` qui serait utilisable autant dans un écran console (de {0,0} à {79,24} inclusivement) que dans un plan cartésien tout en respectant les bornes de validité du domaine dans lequel elle sert?

Classe Bouffe

Une autre classe simple à définir étant donné notre ébauche de système serait la classe `Bouffe`. En effet, en un sens minimaliste, une instance de `Bouffe` est *quelque part, sans plus*, et on peut savoir où elle se trouve en le lui demandant.

Le design proposé ici conçoit une `Bouffe` comme étant placée à un endroit fixe dès sa construction. Si une `Bouffe` devait éventuellement être mobile, alors il faudrait repenser un peu cette classe.

Notez la notation du constructeur et de son paramètre : une instance de `Bouffe` par défaut se trouvera à la `Position` par défaut.

Pour réfléchir

La classe `Bouffe` pourrait être enrichie de plusieurs manières (un nom, une valeur nutritive, est-elle empoisonnée, *etc.*). Un peu plus loin, nous reviendrons sur cette classe pour faciliter l’affichage à l’écran d’une `Bouffe`.

```
#ifndef BOUFFE_H
#define BOUFFE_H
#include "Position.h"
class Bouffe {
    const Position position_;
public:
    Bouffe() = default;
    Bouffe(const Position &p) noexcept
        : position_{p} {
    }
    Position position() const noexcept {
        return position_;
    }
};
#endif
```

Représenter l'orientation

Le cas de la fourmi est complexifié du fait que, pour l'afficher, il nous faut connaître son orientation. Devoir connaître l'orientation implique qu'il nous faudra la représenter.

Ajoutons donc une représentation simple d'une orientation à l'aide de la rose des vents. Il aurait été possible de faire du type `Orientation` une classe en bonne et due forme; nous le ferons si le besoin s'en fait sentir.

```
#ifndef ORIENTATION_H
#define ORIENTATION_H
enum Orientation {
    Est, Nord, Ouest, Sud, NB_DIRECTIONS
};
Orientation pivoter_gauche(Orientation);
Orientation pivoter_droite(Orientation);
#endif
```

Remarquez le réflexe ici : une idée ou une entité semble s'annoncer utile dans un programme? Nous pouvons la nommer et lui donner un rôle? Alors il est probablement sage d'en faire un type (peut-être même une classe) à part entière. Plusieurs petites classes mènent habituellement à un programme plus clair, pas à un programme plus complexe.

Nous accompagnerons le type `Orientation` d'opérations (de petites fonctions, en fait) qui nous permettront de le manipuler convenablement.

Nous pourrions baser ces opérations sur la représentation choisie pour une `Orientation` du fait que nous avons fait le choix de les livrer ensemble.

Ici, nous profitons du fait que les constantes énumérées sont en fait des valeurs entières consécutives pour implémenter le pivot à l'aide de simples opérations arithmétiques. Il est possible de faire encore mieux [hdOptD].

```
#include "Orientation.h"
Orientation pivoter_gauche(Orientation o) {
    return static_cast<Orientation>(
        (o + 1) % NB_DIRECTIONS
    );
}
Orientation pivoter_droite(Orientation o) {
    return o == Est?
        Sud : static_cast<Orientation>(o - 1);
}
```

L'opérateur `static_cast` est décrit dans [POOv01].

Pour réfléchir

Aurait-il été préférable de représenter `Orientation` sous forme de classe avec des méthodes pour se pivoter plutôt qu'en tant qu'entité accompagnée de fonctions? Examinez les deux écritures possibles et réfléchissez à ce qu'elles entraînent comme programme.

Devrait-on qualifier ces fonctions de `noexcept`?

Classe *Fourmi*

Munis d'une implémentation minimale pour une position et une orientation, nous sommes désormais capables de définir un peu mieux ce que sera une fourmi.

Notez le choix, pour `Fourmi`, de ne pas offrir de mutateurs publics : la copie implicite nous suffira, à travers la Sainte-Trinité applicable aux types valeurs, et la copie des attributs d'une `Fourmi` est banale de toute manière. L'interface d'utilisation de notre `Fourmi`, outre les accesseurs dont nous pourrions avoir besoin, sera faite des méthodes `avancer()`, `gauche()` et `droite()`.

Nous laisserons ouverte, du moins pour le moment, la question de l'affichage d'une `Fourmi`. L'influence de l'orientation de la fourmi sur sa représentation à l'écran impliquera un algorithme, si simple soit-il.

En soi, la seule complexité évidente d'une `Fourmi` telle que nous avons conçu cette classe jusqu'ici sera l'algorithme requis pour la méthode `avancer()`.

Notez que celui proposé ici est à la fois lent et facile à optimiser, mais nous l'utiliserons parce qu'il est simple.

Notez aussi qu'il y a un vice de complexité de caché dans cette méthode. Dans chaque cas de la sélective se cache une fonction que nous avons négligé de programmer et qui alourdit inutilement notre écriture ici. Nous y reviendrons un peu plus bas.

```
#ifndef FOURMI_H
#define FOURMI_H
#include "Position.h"
#include "Orientation.h"
class Fourmi {
    Orientation direction_{ Nord };
    Position position_;
public:
    Fourmi() = default;
    Fourmi(const Position &p, Orientation o) noexcept
        : position_{p}, direction_{o}
    {
    }
    Orientation direction() const noexcept {
        return direction_;
    }
    Position position() const noexcept {
        return position_;
    }
    void gauche() noexcept {
        direction_ = pivoter_gauche(direction());
    }
    void droite() noexcept {
        direction_ = pivoter_droite(direction());
    }
    void avancer() noexcept;
};
#endif
```

```
#include "Fourmi.h"
void Fourmi::avancer() noexcept {
    const auto pos = position();
    switch (direction()) {
    case Est:
        position_ = Position{pos.x()+1, pos.y()};
        break;
    case Nord:
        position_ = Position{pos.x(), pos.y()-1};
        break;
    case Ouest:
        position_ = Position{pos.x()-1, pos.y()};
        break;
    case Sud:
        position_ = Position{pos.x(), pos.y()+1};
        break;
    }
}
```

Classe Commande

De manière minimaliste, une `Commande` pourrait ressembler à ce qui est proposé à droite.

Remarquez que nous faisons ici le choix d'inclure le fichier `<iosfwd>` plutôt que le fichier `<iostream>` pour déclarer le prototype de l'opérateur de projection sur un flux pour une `Commande`.

Inclure le minimum dans les fichiers d'en-tête permet de réduire le temps de compilation et les dépendances de manière générale.

Pour réfléchir

Nous aurons pu placer une mécanique de validation dans `Commande`, mais cela aurait impliqué que la validation d'une `Commande` soit possible *en soi*, de manière indépendante du contexte de la `Commande`.

Dans le programme qui nous intéresse, la question se pose : est-ce le travail du menu de valider les commandes? Est-ce le travail du jeu? Est-ce le travail des commandes elles-mêmes? Que signifie, pour une `Commande`, être invalide? Est-ce que `Commande` peut servir pour plusieurs jeux ou est-elle couplée à ce jeu-ci de manière spécifique? Ces questions méritent toutes une attention particulière, et nous reviendrons sur elles un peu plus loin.

L'implémentation de la projection d'une `Commande` sur un flux sera banale, mais l'extraction d'un flux est plus complexe (comme c'est souvent le cas). Notez les précautions prises avant de chercher à extraire une donnée du flux en entrée. Ces précautions servent à éviter de modifier l'état d'un flux corrompu par une lecture antérieure et, ainsi, d'éviter de perdre l'information quant à la nature de cette corruption.

```
#ifndef COMMANDE_H
#define COMMANDE_H
class Commande {
    char symbole_;
public:
    Commande(char symbole) noexcept : symbole_{symbole} {
    }
    char symbole() const noexcept {
        return symbole_;
    }
    bool operator==(const Commande &c) const noexcept {
        return symbole() == c.symbole();
    }
    bool operator!=(const Commande &c) const noexcept {
        return !(*this==c);
    }
};
#include <iosfwd>
std::ostream& operator<<(std::ostream&, const Commande&);
std::istream& operator>>(std::istream&, Commande&);
#endif
```

```
#include "Commande.h"
#include <iostream>
using std::istream;
using std::ostream;
ostream& operator<<
    (ostream &os, const Commande &c) {
    return os << c.symbole();
}
istream & operator>>
    (istream &is, Commande &c) {
    if (is) return is;
    char ch;
    if (is >> ch)
        c = Commande{ch};
    return is;
}
```

Classe Menu

La classe `Menu` devient facile à imaginer une fois l'ébauche de la classe `Commande` construite.

Comme il est généralement sage de le faire, nous réduirons au minimum ce qui, de cette classe, sera révélé au code client.

Le choix d'implémenter les concepts associés aux commandes sous forme de constantes se prête bien à une conceptualisation de `Commande` en tant que type valeur. Des alternatives à cette approche seront discutées plus loin.

Les méthodes de la classe `Menu` vont d'elles-mêmes, surtout dans une version aussi simple.

La tâche de cette classe `Menu` sera, en quelque sorte, d'organiser l'utilisation des instances de `Commande`. Peut-être un meilleur nom serait-il indiqué?

Remarquez la simplification importante de la classe `Menu` obtenue par l'implémentation des opérateurs permettant de projeter une `Commande` sur un flux et d'extraire une `Commande` d'un flux.

Si `Menu` gérait la lecture et l'écriture des instances de `Commande` de manière organique, alors la définition des méthodes de `Menu` serait beaucoup moins agréable à écrire et à entretenir.

Pour réfléchir

Nous ne faisons aucune validation de la `Commande` lue; lors d'une erreur de lecture, la `lire_commande()` retournera `QUITTER`. Nous profitons ici de notre connaissance organique de `Commande` car nous savons que la lecture ne modifiera pas le paramètre de droite si une erreur de lecture survient. Comment aurions-nous pu écrire `lire_commande()` de manière plus générale?

Retourner `QUITTER` lors d'une erreur de lecture est un choix arbitraire¹⁵⁵; ce qui n'est par contre *pas* arbitraire est la détermination d'une stratégie claire pour identifier (ou traiter) les erreurs de lecture. Cette stratégie doit être réfléchie et clairement documentée pour que le programme dans son ensemble demeure stable.

```
#ifndef MENU_H
#define MENU_H
#include "Commande.h"
struct Menu {
    static const Commande
        AVANCER, GAUCHE, DROITE, QUITTER;
    static void afficher();
    static Commande lire_commande();
};
#endif
```

```
#include "Menu.h"
#include "Commande.h"
#include <iostream>
using namespace std;
const Commande
    Menu::AVANCER = Commande{'W'},
    Menu::GAUCHE = Commande{'A'},
    Menu::DROITE = Commande{'D'},
    Menu::QUITTER = Commande{'Q'};
void Menu::afficher() {
    cout << "Avancer: " << AVANCER << "; "
        << "Gauche: " << GAUCHE << "; "
        << "Droite: " << DROITE << "; "
        << "Quitter: " << QUITTER << endl;
}
Commande Menu::lire_commande() {
    Commande c = QUITTER;
    cin >> c;
    return c;
}
```

¹⁵⁵ C'est aussi un choix lié de très près au contexte de ce jeu, ce qui appuie en quelque sorte le choix d'implémenter `lire_commande()` dans une classe comme `Menu` qui est très associée au contexte et sera peu susceptible d'être utilisée telle quelle dans un autre programme

Classe Jeu

Il nous faut maintenant examiner la question de l'instance `jeu` de la classe `Jeu`.

La déclaration de la classe `Jeu` nous indique qu'un `Jeu` contiendra, par composition, une `Fourmi` et une `Bouffe`. Ceci signifie en quelque sorte que la responsabilité de la portée de la `Fourmi` de même que celle de la portée de la `Bouffe` dans le `jeu` dépendent toutes deux de la portée du `Jeu` lui-même.

Les positions initiales de ces deux entités seront des constantes dont les valeurs seront fixées dans le fichier source définissant la classe.

L'existence de lieux par défaut sera donc connue des informaticien(ne)s, mais pas du programme (les constantes étant privées). Les valeurs de ces lieux initiaux, elles, resteront de purs détails d'implémentation¹⁵⁶.

```
#ifndef JEU_H
#define JEU_H
#include "Bouffe.h"
#include "Fourmi.h"
class Commande;
class Jeu {
    Fourmi avatar_{LIEU_FOURMI_DEFAULT};
    Bouffe objectif_{LIEU_BOUFFE_DEFAULT};
    static const Position
        LIEU_BOUFFE_DEFAULT, LIEU_FOURMI_DEFAULT;
public:
    Jeu() = default;
    bool fini() const noexcept;
    void afficher_etat() const;
    void executer(const Commande&);
};
#endif
```

Remarquez que, dans ce fichier, nous pouvons utiliser une déclaration *a priori* pour la classe `Commande` puisque nous ne dépendons pas ici du détail de sa définition. En effet, le seul recours à une instance de cette classe repose sur une référence. C'est une bonne nouvelle : il est en général sage de viser le plus faible couplage possible entre les objets d'un système, du fait que cela réduit les efforts d'entretien à y investir.

¹⁵⁶ Bien que les positions par défaut soient des constantes, il serait même possible de les générer au hasard. Le caractère constant des objets est fixé suite à leur construction, après tout, et notre programme contrôle pleinement le détail de la construction des objets qui l'habitent.

Petites fautes de design

L'implémentation de la classe `Jeu` nous montrera quant à elle que notre design initial est quelque peu fautif. Des retouches s'imposeront.

Tout d'abord, notons que la condition indiquant que le jeu est terminé (prédicat `fini()`) est convenable. Cette méthode détermine que le jeu est terminé quand la fourmi a atteint le lieu où se situe son objectif, ce qui est raisonnable considérant les règles que nous nous sommes fixées.

Si nous en venons à souhaiter des règles de fin de jeu plus riches, elles pourront sans peine être définies de manière encapsulée au même endroit.

L'affichage de l'état du jeu pose clairement problème¹⁵⁷, au sens où cette méthode empiète sur des zones de savoir qui ne lui appartiennent pas :

- elle décide du symbole à utiliser pour afficher la Fourmi et la Bouffe, ce qui ne la regarde pas. Implémenter nos règles d'affichage directionnel du symbole de la fourmi à cet endroit compliquerait à la fois la logique et l'entretien de la méthode

Le couplage entre `Jeu` et `Fourmi` est, actuellement, trop fort.

- elle détermine aussi les paramètres de l'espace dans lequel sera affiché le jeu. Il n'est pas clair qu'il soit sage d'entremêler règles du jeu et espace de jeu de manière aussi serrée.

Visiblement, il manque au moins une classe et quelques opérations à notre prototype de jeu.

```
#include "Jeu.h"
#include "Commande.h"
#include "Menu.h"
#include <ctsdlib>
#include <iostream>
using namespace std;
const Position
    Jeu::LIEU_BOUFFE_DEFAULT = Position{1,1},
    Jeu::LIEU_FOURMI_DEFAULT = Position{5,5};
bool Jeu::Fini() const noexcept {
    Return avatar_.position()==objectif_.position();
}
void Jeu::afficher_etat() const {
    enum { // l'écran
        X_MIN = 0, Y_MIN = 0,
        X_MAX = 80, Y_MAX = 20
    };
    system("cls");
    for (int i = Y_MIN; i < Y_MAX; ++i)
        for (int j = X_MIN; j < X_MAX; ++j) {
            const Position pos{j,i};
            if (pos == avatar_.position())
                cout << 'F'; // dessiner la fourmi
            else if (pos == objectif_.position())
                cout << 'B'; // dessiner la bouffe
            else
                cout << ' '; // rien ni personne!
        }
}
void Jeu::Executer(const Commande &c) {
    if (c == Menu::AVANCER)
        avatar_.avancer();
    else if (c == Menu::DROITE)
        avatar_.droite();
    else if (c == Menu::GAUCHE)
        avatar_.gauche();
}
```

¹⁵⁷ Petit détail : l'ordre des tests est important puisqu'il priorise l'affichage d'une fourmi sur l'affichage de la bouffe quand les deux se trouvent au même endroit. Cela permet à la fourmi de manger la bouffe plutôt que le contraire!

Un autre problème visible à qui testera le jeu maintenant est la structure de la répétitive principale que nous avons conçue à l'origine.

Mis à part l'absence de validation quant aux frontières de l'espace de jeu (présentement, il est possible de sortir de l'écran sans peine) et le manque de convivialité à l'affichage, deux irritants apparaîtront clairement à l'usage :

- nos commandes sont sensibles à la casse. Conséquence : la touche 'w' ne fera pas avancer la fourmi mais la touche 'W' le fera, ce qui est un peu déplaisant; et
- notre répétitive de contrôle est fondamentalement flouée.

```
// ...
for(Commande c = Menu::lire_commande();
    c != Menu::QUITTER && !jeu.fini();
    c = Menu::lire_commande()) {
    jeu.executer(c);
    jeu.afficher_etat();
}
// ...
```

Corriger le problème de la convivialité d'une Commande

Le problème de convivialité résultant des touches majuscules et minuscules pourra être résolu en ajustant la définition de l'opérateur == de Commande.

En effet, ceci permet de ne considérer que les comparaisons entre versions majuscules (ou minuscules, selon les préférences) des symboles encapsulés dans deux instances de Commande.

Notez que le choix d'implémenter != à partir de ==, fait plus tôt dans notre conception de Commande, réduit ici la complexité de la modification requise.

En effet, avec cette approche, la cohérence de l'opérateur != découle directement de la cohérence de l'opérateur ==.

```
#ifndef COMMANDE_H
#define COMMANDE_H
#include <locale>
class Commande {
    // ...
    bool operator==(const Commande &c) const noexcept {
        using namespace std;
        const locale loc("");
        return toupper(symbole(),loc) ==
            toupper(c.symbole(),loc);
    }
    bool operator!=(const Commande &c) const noexcept {
        return !(*this ==c);
    }
};
// ...
#endif
```

Solutions alternatives

Il existe d'autres stratégies légitimes pour résoudre le problème de convivialité dans les opérations de la classe Commande.

Nous aurions pour choisir de n'entreposer en tant que symbole d'une Commande que la valeur majuscule du symbole fourni à la construction. Nous aurions aussi pu réaliser la conversion à chaque invocation de la méthode symbole().

En pratique, le meilleur choix entre ces quelques options dépendra de l'utilisation qui sera faite du type Commande. Il n'y a pas de solution qui soit meilleure que les autres dans tous les cas.

Corriger le problème structurel de la répétitive principale

Le problème structurel de la répétitive du programme principal est lié au fait que, suite à un déplacement de la fourmi, déplacement qui peut mener à la fin du jeu, une `Commande` sera nécessairement lue avant de tester si le jeu est fini. Ceci force l'utilisateur à appuyer sur une touche bidon même si sa fourmi a atteint son objectif pour que l'invocation de la méthode `fini()` de `jeu` soit réalisée et pour que la fin du jeu soit déclarée.

Corriger convenablement ce problème est plus subtil qu'il n'y paraît. Il faut :

- tester la fin du jeu avant de lire une `Commande`; et
- tester la commande lue pour savoir s'il s'agit d'une demande de quitter le jeu avant de chercher à l'exécuter.

Souhaitant éviter la multiplication des alternatives redondantes, nous restructurerons la répétitive de manière à ce que sa condition comprenne *à la fois* le test pour la fin du jeu, la lecture d'une `Commande` et le test de cette `Commande`.

Combiner la lecture et le teste de la `Commande` dans l'ordre implique un recours à l'opérateur *virgule* (`,`) de C++. Nous ne surchargerons pas cet opérateur, nous ne ferons que l'utiliser.

Ici, nous lirons la `Commande` puis nous évaluerons la `Commande` lue, et le résultat sera celui de l'évaluation en question.

Pour réfléchir

Effet secondaire de cette restructuration : notre classe `Commande` requiert maintenant un constructeur par défaut, ce qui n'était pas le cas auparavant. Notre structure répétitive modifiée nous force à réfléchir à la question *Dans quel état une Commande par défaut sera-t-elle?*

Le choix d'une bonne valeur par défaut est une question à se poser. Ici, nous présumerons que `' '` fera le travail, mais cette situation pourra être réévaluée. Comme c'est souvent le cas, la clé ici est de faire un choix et de le documenter.

```
int main() {
    Jeu jeu;
    jeu.afficher_etat();
    Menu::afficher();
    Commande c;
    while (!jeu.fini() &&
           (c = Menu::lire_commande(),
            c != Menu::QUITTER)) {
        jeu.executer(c);
        jeu.afficher_etat();
        Menu::afficher();
    }
}
```

L'expression `a,b` signifie évaluer `a`, puis évaluer `b`, et le résultat sera `b`.

```
class Commande {
    char symbole_;
public:
    Commande(char symbole = ' ') noexcept
        : symbole_{symbole} {
    }
    // ...
};
```

Encapsuler l’affichage de la bouffe

Les prochains cas à examiner sont ceux de l’affichage de la bouffe et de l’affichage de la fourmi.

Dans le cas de la bouffe, pour le moment, rédiger un opérateur de projection sur un flux est une préoccupation esthétique puisque le symbole est fixé, mais il serait facile d’imaginer plusieurs catégories de bouffe, ce qui aurait pour effet d’entraîner une variation dans l’affichage.

Encapsuler l’affichage dans l’interface de la classe `Bouffe` est donc sage. L’implémentation de cette fonction sera banale.

Ici, nous n’avons pas vraiment besoin de consulter les états de la `Bouffe` à afficher pour le moment, toutes les instances de `Bouffe` étant équivalentes à nos yeux.

Nous pouvons donc omettre le nom (pas le type!) de ce paramètre (en gras).

Ajuster l’affichage du jeu

L’affichage de la bouffe dans le jeu, dans la méthode d’instance `afficher_etat()` de la classe `Jeu`, ne sera donc maintenant plus réalisé comme ceci...

... mais bien comme cela. Ceci libère `Jeu` d’un savoir qui ne lui appartenait pas de droit, et réduit le couplage de notre système.

```
// ...
#include <iosfwd>
class Bouffe {
    // ...
};
std::ostream& operator<<
    (std::ostream&, const Bouffe&);
// ...

#include "Bouffe.h"
#include <iostream>
using std::ostream;
ostream & operator<<
    (ostream &os, const Bouffe&) {
    return os << 'B';
}
```

```
// ...
else if (pos == objectif_.position())
    cout << 'B'; // dessiner la bouffe
// ...

// ...
else if (pos == objectif_.position())
    cout << objectif_;
// ...
```

Encapsuler l’affichage de la fourmi

Le même principe s’appliquera à la classe `Fourmi`, dont chaque instance devrait être responsable de son propre affichage, de sa propre projection sur un flux.

L’opérateur de projection sur un flux pour une instance de `Fourmi` sera défini de manière à examiner l’orientation de cette `Fourmi` puis de projeter le symbole approprié sur le flux visé.

Ce problème, sans être complexe, est plus riche que celui de l’affichage de la bouffe, ci-dessus.

L’implémentation de cette fonction à partir d’une sélective est une approche raisonnable mais fortement perfectible. Il existe plusieurs approches plus efficaces pour arriver au même résultat.

Nous utiliserons la solution par sélective par souci de simplicité. Cela importe peu pour le moment : en effet, toute optimisation à la stratégie de sélection du symbole à projeter sur un flux sera localisée et encapsulée. On peut donc se permettre de remettre ces raffinements à plus tard.

Si la vitesse d’affichage devient un irritant, une mise à jour de cette stratégie n’entraînera pas de modifications systémiques ou de dégâts indus dans le code client.

N’oubliez pas de modifier l’affichage de la `Fourmi` dans la méthode `afficher_etat()` de `Jeu` pour que cette méthode profite de la fonction projetant une `Fourmi` sur un flux plutôt que d’afficher bêtement le symbole `'F'`. Cet ajustement au code est banal et ne sera pas explicitement illustré ici.

Pour réfléchir

Considérant que le cas par défaut ne devrait, en théorie, jamais survenir, serait-il préférable de lever une exception plutôt que d’afficher un symbole comme `'??'`?

```
// ...
#include <iosfwd>
class Fourmi {
    // ...
    Orientation direction() const noexcept {
        return direction_;
    }
    // ...
};
std::ostream& operator<<
    (std::ostream&, const Fourmi&);
// ...
```

```
#include "Fourmi.h"
#include <iostream>
using namespace std;
ostream& operator<<
    (ostream &os, const Fourmi &f) {
    switch (f.direction()) {
    case Est:
        os << '>'; break;
    case Nord:
        os << 'A'; break;
    case Ouest:
        os << '<'; break;
    case Sud:
        os << 'V'; break;
    default:
        os << '??';
    }
    return os;
}
// ...
```

La classe manquante : l'espace de jeu

Il nous reste un problème de design à résoudre : l'espace de jeu n'a pas de définition propre. Les règles du jeu sont définies (l'utilisateur déplace un avatar, représenté par une fourmi, tant que la fourmi n'a pas atteint sa nourriture) mais l'espace de jeu, lui, ne l'est pas.

Nous avons déjà un embryon de définition d'espace dans la méthode `afficher_etat()` de la classe `Jeu`, pour que les répétitives affichant l'espace de jeu fonctionnent. Cependant, si nous souhaitons éviter que notre logique de jeu ne s'éparpille et se complique, nous voudrions ramener ensemble les idées :

- de dimension de l'espace de jeu (largeur, hauteur);
- de frontières de l'espace de jeu;
- de validation d'une position potentielle dans l'espace de jeu, pour contrôler les mouvements de la fourmi; et ainsi de suite.

Comme dans le reste de notre démarche, nous resterons minimalistes et nous viserons une solution à la fois concise, claire et élégante.

Nous chercherons à distinguer clairement ce qui tient des règles du jeu et ce qui relève plutôt de la gestion de l'espace. Ce faisant, certains éléments du pourraient être déplacés des règles (la classe `Jeu`), où ils se trouvent présentement, vers l'espace (la classe `Espace` que nous allons maintenant concevoir), comme par exemple la fourmi et la bouffe. Nous devons faire des choix.

Nous viserons aussi à réduire au minimum l'impact de nos changements sur le code client (ici : le programme principal).

Idéalement, en fait, le programme principal ne devrait pas changer, pas même d'une seule ligne.

Ajuster l’affichage de l’espace de jeu

Un premier ajustement à faire serait d’extraire de la fonction `afficher_etat()` de `Jeu` la définition des bornes en largeur et en hauteur de l’espace d’affichage pour les placer dans une classe `Espace` représentant l’espace de jeu.

Présumant que nous souhaitions conserver à peu près la même structure générale qu’auparavant, cette modification pourra se limiter à introduire quelques constantes de classe publiques dans la classe `Espace`.

Notez les noms des constantes, qui sont maintenant qualifiées par le nom de la classe à laquelle elles appartiennent (`Espace::`).

```
#ifndef ESPACE_H
#define ESPACE_H
struct Espace {
    enum {
        XMIN = 0,  YMIN = 0,
        XMAX = 80, YMAX = 20
    };
};
#endif
```

Les modifications au code des méthodes de `Jeu` qui résulteront de ce changement sont visibles à droite (en gras).

Plusieurs raffinements supplémentaires sont possibles ici à l’aide de techniques de POO et de programmation générique plus sophistiquées.

Nous examinerons un cas d’espèce plus riche dans [POOv03], section *Manières d’appliquer l’approche OO*.

Pour réfléchir

Aurait-on pu (et aurait-on dû) utiliser des instances de `Position` comme constantes de classe dans `Espace` pour déterminer les bornes de validité de cet espace? Quelles seraient les conséquences d’un tel choix de design?

```
// ...
#include "Espace.h"
// ...
void Jeu::afficher_etat() const {
    system("cls");
    for (int i = Espace::YMIN; i < Espace::YMAX; ++i)
        for (int j = Espace::XMIN; j < Espace::XMAX; ++j) {
            const Position pos{j,i};
            if (pos == avatar_.position())
                cout << avatar_;
            else if (pos == objectif_.position())
                cout << objectif_;
            else
                cout << ' ';
        }
}
// ...
```


Valider les déplacements et les positions dans l'espace de jeu

L'autre cas qui nous intéresse ici est celui de la validation des mouvements de la fourmi. En l'absence d'obstacles tangibles sur la surface de jeu, les règles de validité d'une position peuvent être résumées en quelques mots : une `Position` valide sera une `Position` se situant dans l'espace de jeu.

Une question fondamentale en POO naît naturellement de ce constat : la question de la **responsabilité**. Est-ce à une `Position` de déterminer si elle est valide dans un `Espace` ou est-ce à un `Espace` de valider une `Position`? Qui est responsable?

Dans une telle situation, un réflexe raisonnable est de *viser le choix qui entraînera la plus faible dépendance*.

Une position est un concept sujet à intervenir dans plusieurs projets, sans égard aux règles de l'espace, alors que l'espace de jeu est un concept très lié à un jeu spécifique.

Il est donc probablement préférable, dans un cas comme celui-ci, de confier à un `Espace` la responsabilité de valider une `Position`.

Les modifications à la méthode de `Jeu` chargée d'exécuter une `Commande` impliqueront de valider la destination prospective préalablement à l'actualisation d'une demande de déplacement.

Pour alléger à la fois la rédaction du code de validation des positions prospectives (dans la méthode `executer()` de `Jeu`) et le code de la méthode `avancer()` de `Fourmi`, nous ajouterons quelques méthodes d'instance à la classe `Position` pour générer les positions voisines d'une position donnée.

```
//...
#include "Position.h"
class Espace {
    // ...
    static bool est_valide(const Position &p) noexcept {
        return XMIN <= p.x() && p.x() < XMAX &&
            YMIN <= p.y() && p.y() < YMAX;
    }
};
// ...
```

```
//...
class Position {
    // ...
    Position voisine_est() const noexcept {
        return {x()+1, y()};
    }
    Position voisine_nord() const noexcept {
        return {x(), y()-1};
    }
    Position voisine_ouest() const noexcept {
        return {x()-1, y()};
    }
    Position voisine_sud() const noexcept {
        return {x(), y()+1};
    }
};
// ...
```

La classe `Fourmi` pourra maintenant offrir avec simplicité une méthode exposant la prochaine case à visiter pour une `Fourmi` désireuse d'avancer.

Par la suite, la méthode d'instance `avancer()` de la classe `Fourmi` pourra profiter de ce raffinement pour être fortement simplifiée.

Sans faire trop de bruit, en ajoutant quelques méthodes plutôt simples et ayant chacune une vocation claire, nous venons de clarifier le programme et d'en faciliter l'entretien sans faire quelque sacrifice que ce soit du côté vitesse.

Remarquez que le cas par défaut de cette implémentation, cas rencontré lorsqu'une instance de `Fourmi` a une `Orientacion` invalide, est de faire du sur-place. Une alternative serait de traiter un tel cas comme un problème et de lever une exception.

Enfin, la méthode `executer()` de la classe `Jeu` pourra prendre en charge, au prix d'une complexité somme toute très raisonnable¹⁵⁸, la validation de la destination prospective de la fourmi.

Nous n'avons pas introduit de validation lors des pivots de la fourmi, le problème en cours d'examen ne demandant pas de telle validation.

Le recours à une séquence d'alternatives (plutôt qu'à une sélective) tient au fait que `Commande` est une classe, pas un type entier.

```
// ...
class Fourmi {
    // ...
    Position destination() const noexcept {
        auto pos = position();
        switch (direction()) {
            case Est:
                pos = pos.voisine_est(); break;
            case Nord:
                pos = pos.voisine_nord(); break;
            case Ouest:
                pos = pos.voisine_ouest(); break;
            case Sud:
                pos = pos.voisine_sud(); break;
        }
        return pos;
    }
    void avancer() noexcept {
        position_ = destination();
    }
};
// ...
```

```
// ...
void Jeu::executer(const Commande &c) noexcept {
    if (c == Menu::AVANCER) {
        if (Espace::est_valide(avatar_.destination()))
            avatar_.avancer();
    } else if (c == Menu::DROITE) {
        avatar_.droite();
    } else if (c == Menu::GAUCHE) {
        avatar_.gauche();
    }
}
// ...
```

Nous pourrions écrire le même programme de manière plus simple et plus élégante quand nous aurons couvert la matière exposée dans [POOv01].

¹⁵⁸ On peut faire mieux, avis aux intéressé(e)s!

Remarques d'ensemble

Nous avons construit un petit jeu selon une approche ∞ limitée à ce que nous en connaissons jusqu'ici (donc à une infime fraction des possibles qui s'offriront sous peu à nous).

Le problème était petit et le système construire est plutôt simple, mais nous pouvons en tirer quelques questions et quelques réflexions susceptibles d'orienter notre pensée dans le futur.

À propos du design initial

Une première remarque à faire est que, pour notre design initial, nous avons utilisé une combinaison de code (une ébauche de code client) et de schémas (un diagramme de classe UML). Ne soyons pas dogmatiques et sachons utiliser à bon escient ce qui peut nous guider vers une solution de qualité.

Ici, une ébauche de code client permettait d'articuler la structure cyclique du jeu (l'algorithme général et aérien, disons) et de mettre en valeur un certain niveau d'abstraction souhaité et souhaitable (pas de fourmi ni de bouffe dans le code client). De son côté, le schéma a permis de placer quelques-uns des principaux composants logiciels du système et d'entrevoir lesquels entretiennent des relations entre eux. On aurait même pu décrire ces relations en qualifiant par un court texte les arcs reliant chaque paire de classes.

Il est sain de remarquer que ni l'ébauche de code client général, ni le diagramme de classes initial ne furent définitifs. Dans chaque cas, nous avons apporté des raffinements et des retouches dictées par l'usage. Nous avons cherché à garder le couplage entre les entités aussi bas que possible, et nous avons préféré introduire une nouvelle classe (`Espace`) plutôt que d'ajouter aux classes existantes des responsabilités qui ne leur reviennent pas de droit.

À propos de la responsabilité opératoire

L'une des questions les plus importantes auxquelles nous avons dû faire face fut la question de la responsabilité opératoire. Cette question est habituellement soulevée lorsqu'une opération implique au moins deux entités, souvent des instances de classes distinctes, et lorsque la responsabilité de l'opération pourrait revenir de droit à l'une comme à l'autre.

Dans les cas les plus simples, la réponse à la question de la responsabilité va de soi. Si un objet est propriétaire d'une donnée et si cette donnée est au cœur d'une opération, alors le propriétaire est presque toujours le responsable cherché. Qui se demandera, en effet, si une `Position` est vraiment responsable de dévoiler ses coordonnées en `X` et en `Y`? La `POO` relève souvent, par nature, d'opérations subjectives.

Évidemment, il existe des cas plus subtils, qui impliquent une responsabilité partagée ou discutable, comme celui de la position dans l'espace. Lorsque nous avons fait face à deux manières fonctionnellement équivalentes de mettre en place une relation, comme dans le cas de la relation entre une position et l'espace où elle se situe, nous avons choisi l'approche qui était la plus susceptible de survivre dans plusieurs projets et qui introduisait le moins de couplage. Pour nous, faire dépendre `Espace` de `Position` semblait moins coûteux à long terme que ne le serait le choix de faire dépendre `Position` de la classe `Espace`.

Deux autres approches sont fréquemment rencontrées et peuvent être à privilégier, selon le cas :

- introduire une fonction globale opérant sur les deux entités potentiellement responsables. C'est souvent l'approche choisie pour les opérateurs binaires lorsque les opérandes ne sont pas du même type; et
- introduire une classe supplémentaire, distincte de celles des deux entités en cause, qui sera responsable de définir les relations problématiques.

Dans un cas comme dans l'autre, la clé est de sortir un peu du cadre et des conventions. Quand un problème semble insoluble (ou n'avoir que des solutions inélégantes) tel qu'il est posé, la clé est souvent de le reformuler.

Ainsi, si les entités en cause ne peuvent être responsabilisées élégamment, on peut déléguer la responsabilité à un tiers (ajouter une classe) ou déterminer qu'il s'agit d'un cas sans responsable clair (ajouter une fonction globale).

Dans les deux cas, la clé est de sortir du cadre et de confier à un tiers la responsabilité lorsque ni l'une, ni l'autre des deux entités problématiques ne semble pouvoir correctement s'en charger.

À propos de la conception des classes

Remarquez ensuite que nous avons pensé les objets sur une base opératoire, donc au sens de ce qu'ils font, plutôt que sur une base structurelle. Nous avons pensé en termes de méthodes plutôt qu'en termes d'attributs. Ceci ne nous a pas mené à proposer des accesseurs et des mutateurs de manière systématique, puisque nous ne savions pas vraiment comment serait fait chaque objet.

Qu'on se comprenne bien ici : nous n'avons pas aboli ou omis les accesseurs et les mutateurs, mais nous les avons utilisés lorsqu'ils semblaient indiqués. Nous n'avons pas ajouté des accesseurs ou des mutateurs de manière automatique. Remarquez d'ailleurs l'absence de mutateurs dans cette ébauche de programme. Ce n'est pas un accident : les mutateurs, en pratique, sont utiles, évidemment, mais moins souvent qu'on pourrait le penser.

Nous avons tout de même cherché à minimiser à la fois la surface publique des objets et les points d'accès à leurs attributs. Comme nos objets étaient tous structurellement petits, munis de peu d'attributs, nous avons été en mesure dans la majorité des cas de restreindre les lieux de modification des attributs d'instances aux seuls constructeurs, offerts en nombre réduit, et aux opérateurs d'affectation, la plupart du temps générés de manière implicite. La simplicité au secours de l'efficacité, pourrait-on dire.

Plusieurs langages OO, nous l'avons vu, ont des systèmes de types incomplets et ne permettent pas de définir des objets constants une instance à la fois.

Dans ces langages, on a recours à la technique des objets immuables, dont la valeur ne peut changer suite à leur construction dû à une absence de mutateurs ou d'autres méthodes changeant la valeur de leurs états.

L'approche préconisée ici a le mérite de s'appliquer tout autant à ces langages qu'aux langages dont le système de types est plus riche et permet les instances constantes.

On remarquera l'accent mis sur la modification des objets à l'aide de l'opérateur d'affectation et de la comparaison à l'aide des opérateurs `==` et `!=`. La raison de ces choix est que nous avons travaillé essentiellement avec des types valeurs (voir *Appendice 00 – Sémantiques directes et indirectes*) et que, dans une telle situation, les informaticien(ne)s ont des attentes opérationnelles face aux objets.

Pour la majorité des informaticien(ne)s, affecter une `Commande` à une autre `Commande` devrait fonctionner; comparer deux instances de `Commande` à l'aide de l'opérateur `==` devrait retourner `VRAI` seulement si les deux instances de `Commande` en question sont pareilles au sens attendu dans le programme. Ce sont des comportements normaux pour des instances de types valeurs – des objets concrets.

À propos de la comparaison d'objets concrets

La comparaison d'instances de `Commande` à l'aide des opérateurs `==` et `!=` est une approche possible sans toutefois être la seule approche possible. Une autre option raisonnable aurait pu être d'exprimer l'idée sous une forme responsabilisant la classe `Menu` plutôt que la classe `Commande`.

Ceci aurait fait passer le programme de l'écriture

```
Commande c;
if ( ... && (c = Menu::lire_commande(), c!=Menu::QUITTER))
```

à l'écriture

```
Commande c;
if ( ... && (c = Menu::lire_commande(), !Menu::est_quitter(c)))
```

Remarquez les caractères gras, qui indiquent le glissement sémantique qui s'opère en passant d'une notation à l'autre. Les deux extraits font la même chose mais ne signifient pas la même chose. En particulier, un glissement s'est opéré entre les deux écritures quant à la responsabilité des diverses entités impliquées.

L'expression que nous avons choisie, la première de ces deux écritures, présume qu'il soit possible d'exprimer une constante `QUITTER` sous forme d'une `Commande` (ou sous une forme comparable avec une `Commande`) et de comparer `QUITTER` avec une instance quelconque de `Commande`. L'expression alternative indique qu'il est possible pour `Menu` d'examiner une instance de `Commande` et d'évaluer si cet objet représente une demande de quitter le programme. Elle indique aussi que `Menu` est responsable du sens opératoire à donner à l'idée de *cette commande signifie qu'il faut quitter*¹⁵⁹.

Les deux manières d'exprimer une demande pour quitter le programme sont valides. La seconde pourrait être préférable si plusieurs touches distinctes signifiaient qu'il faut quitter. Du fait que `Commande` soit un type valeur, la première, que nous avons choisie, convient mieux à un usage jugé normal pour la majorité des gens. La première écriture est probablement celle qui sera la plus intuitive pour le code client.

Notez que dans le cas de la première approche, il serait possible d'implémenter une logique sophistiquée à même la classe `Commande` pour atteindre un objectif semblable à ce que permet la deuxième approche.

¹⁵⁹ Notez qu'il existe des variantes intéressantes à ces interprétations. Entre autres, le type de `Menu::QUITTER` pourrait être un type spécial, par exemple une classe distincte de `Commande` mais pour laquelle il existerait un opérateur de comparaison avec une `Commande`. Ceci déplacerait la responsabilité de la comparaison d'une `Commande` avec la valeur du symbole `Menu::QUITTER` vers cet autre type. Certaines techniques OO intéressantes peuvent être construites à partir de cette tangente.

À propos de l'introduction d'une valeur dans un système

Extraire une instance d'un flux (en fait, reconstituer une instance à partir de données sérialisées sur un flux) et construire une instance en sollicitant son constructeur sont deux manières d'introduire un objet dans un système.

Lorsque nous rédigeons un constructeur par défaut ou un constructeur de copie, présumant un respect rigoureux du principe d'encapsulation, nous sommes en mesure de dire *a priori* que l'objet nouvellement construit sera dans un état valide (et que, s'il est impossible de construire un objet valide, une exception sera levée et la construction n'aura jamais eu lieu).

Avec les constructeurs paramétriques, nous chercherons habituellement à valider les paramètres reçus par le constructeur pour assurer la validité de l'objet construit et permettre son encapsulation pleine et entière.

Lorsque des paramètres invalides sont reçus par un constructeur paramétrique, deux grandes approches sont possibles :

- remplacer les valeurs illégales par des valeurs par défaut valides; et
- lever une exception pour signifier le problème au code client et faire en sorte que la construction n'ait pas lieu.

En fait, une troisième approche est possible, soit celle d'accepter l'état invalide et d'implémenter toutes les méthodes de l'objet de manière à ce qu'elles aient un comportement adéquat dans un tel cas. C'est ce que font les flux standards de C++.

Dans la majorité des cas, lever une exception sera privilégié pour éviter que le constructeur ne dissimule silencieusement une faute plus criante ailleurs dans le programme. Il faut alors que le code instanciant l'objet soit prudent : si une exception est levée par un constructeur, l'objet n'est pas considéré construit, ce qui influence grandement le code client.

Ceci nous amène à discuter de la question de la validité d'une `Commande`. Nous avons choisi dans cet exemple tout simple de ne pas valider les états d'une `Commande`. En procédant ainsi, tout symbole devient valide pour une `Commande` mais certaines instances de `Commande` auront des états plus pertinents que d'autres dans un contexte donné.

Dans l'optique développée ici, le contexte est responsable de valider une `Commande` même si la comparaison entre deux instances de `Commande`, qui est une opération plus primitive, demeure sous la responsabilité de la classe `Commande`.

Imaginons que nous ayons fait un choix de design différent, comme par exemple déterminer qu'une `Commande` valide contiendrait un symbole alphabétique. Quel aurait été l'impact de ce choix sur notre système?

Tout d'abord, nous aurions dû valider l'état initial d'une `Commande` à l'aide d'un mutateur ou d'un autre mécanisme, puis lever une exception convenable si l'état s'avérait invalide.

Notez qu'il nous faudrait une nouvelle valeur par défaut puisque ' ' ne convient plus, n'étant pas alphabétique. Ici, comment savoir si la valeur choisie est convenable?

```
// ...
#include <locale>

class Commande {
    char symbole_;
public:
    class Invalide {};
    Commande(char symb = 'z') : symbole_{symb} {
        using namespace std;
        if (!isalpha(symbole_), locale{ "" })
            throw Invalide{};
    }
    // ...
};
// ...
```

Une alternative est de prendre une valeur hors du domaine de validité (un '?', peut-être) et d'ajouter du code dans toutes les méthodes appropriées pour en tenir compte. Ce choix peut être transparent pour le code client mais accroîtra la complexité d'ensemble (et peut mener à du code lent).

L'opérateur d'extraction d'une `Commande` à partir d'un flux n'a pas à être modifié puisque son implémentation existante repose en partie sur la construction d'une `Commande` temporaire.

Le constructeur d'une `Commande` assurant la levée d'une exception lorsqu'une `Commande` entre dans un état invalide, le signalement d'un cas d'exception est d'ores et déjà implémenté.

Enfin, la méthode `lire_commande()` de la classe `Menu` devra tenir compte du risque de levée d'exceptions.

Ici, nous avons adapté la définition de la méthode de manière à réduire au minimum (en fait, de réduire à néant) l'impact sur le code client, mais nous aurions aussi pu rédiger une répétitive redemandant à l'utilisateur d'entrer une commande jusqu'à ce que la `Commande` lue soit jugée valide.

```
// ...
istream & operator>>
    (istream &is, Commande &c)
{
    if (!is) return is;
    char ch;
    if (is >> ch)
        c = Commande{ch}; // boum?
    return is;
}
// ...
```

```
// ...
Commande Menu::lire_commande() noexcept {
    Commande c = QUITTER;
    try {
        cin >> c;
    } catch (Commande::Invalide&) {
    }
    return c;
}
// ...
```


Remarques diverses quant au développement

Sur le plan du développement en tant que tel, enfin, nous pouvons faire quelques remarques.

- Nous avons évité d'implémenter explicitement la Sainte-Trinité (voir *Sainte-Trinité*) dans la mesure du possible¹⁶⁰. En effet, si un objet ne contient que des attributs de type valeur, les comportements par défaut sont tout à fait convenables. Cette approche allège le code et permet au compilateur de réaliser un certain nombre d'optimisations basées sur sa connaissance du problème.
- Presque toutes nos méthodes sont courtes et concises, n'étant composées que d'une ou deux opérations, hormis celles qui contiennent des sélectives (des `switch`) ou du traitement d'exceptions. Les sélectives pourraient, pour la plupart, être améliorées (et raccourcies) en utilisant des tableaux.
- Lorsque cela s'avérait possible, nous avons privilégié les méthodes de classe aux méthodes d'instance. Après tout, pourquoi forcer l'instanciation d'une classe qui nous n'avons pas besoin de ses attributs d'instance?
- Implémenter les opérateurs d'écriture sur un flux et d'extraction d'un flux pour un type valeur sont des gestes fréquemment utiles, pour ne pas dire salutaires, car ces gestes ramènent la responsabilité de la sérialisation d'un type sur les épaules (virtuelles) de ce type.
- Nous avons choisi de dégarnir les fichiers d'en-tête et d'utiliser les déclarations *a priori* lorsque cela s'avérait possible. Cela réduit le couplage entre les entités d'un programme et accélère la compilation. De même, lorsque cela s'y prêtait¹⁶¹, nous avons privilégié les entêtes légers aux entêtes lourds et les déclarations *a priori* aux inclusions de fichiers d'en-tête.

Notons enfin l'apport de l'encapsulation dans notre capacité de gérer les changements dans le code. Ceci fut mis en relief par l'irritant de la lecture des commandes, où 'w' et 'W' étaient considérés comme différents par le programme. En insérant la mise en majuscules des commandes au moment de la comparaison (ou ailleurs) mais de manière encapsulée dans les opérations de `Commande`, nous avons pu modifier la politique d'utilisation de cette classe sans forcer de modification au code client.

¹⁶⁰ En fait, ici, nous l'avons évitée dans tous les cas!

¹⁶¹ Pensez à `<iosfwd>` en comparaison avec `<iostream>`.

Appendice 02 – Nommer et documenter

Note : cette section utilisera des noms à consonance francophone. Libre à vous d’angliciser chacun si telle est la pratique de votre entreprise, vous souvenant ce faisant que la structure des phrases anglaises diffère de celles énoncées en français (p. ex. : `DEFAULT_NOTE` au lieu de `NOTE_DEFAULT`). Règle générale, les différences de nomenclature dues au choix de langue sont moindres qu’on ne le croirait : les termes techniques tendent à se ressembler d’une langue à l’autre.

Note : vous trouverez une version tenue plus à jour de cette section sur [hdNom]

Important : l’orthographe joue un rôle important dans le vécu informatique. Veillez à sauver temps et argent à votre firme, et prenez soin d’utiliser l’orthographe la plus juste possible, et d’insister auprès de vos employé(e)s pour qu’ils/ elles en fassent autant.

On demande souvent (en fait, toujours!) aux gens devant mettre la main à la pâte lors du développement informatique d’utiliser des *noms significatifs* pour leurs variables, constantes, sous-programmes, types, classes, instances, méthodes, modules, *etc.*

L’emploi de commentaires judicieux aux endroits appropriés, la rigueur quant à la nomenclature, et le respect des standards en place sont trois des éléments fondamentaux menant à une *autodocumentation* des programmes.

Bien que les commentaires soient importants dans les programmes, il est sage de rendre le code si clair que les commentaires deviennent, à toutes fins pratiques, inutiles. Voir *À propos des commentaires*, plus bas, pour des détails.

Certains doivent appliquer ces standards; il s’agit surtout des gens devant programmer, tester ou aider à la documentation d’un logiciel. D’autres doivent veiller à établir des standards, à les faire respecter, à vérifier occasionnellement leur application et (surtout, quoiqu’on en pense) à les vendre à leurs collègues et employés.

Mettre occasionnellement la main à la pâte dans le développement de systèmes implique faire partie du premier groupe. Votre lecture de cet appendice indique toutefois que vous risquez fort de vous retrouver aussi dans le second. Vous devrez donc connaître autant (sinon plus!) que vos développeurs les normes à appliquer dans votre entreprise, et les moteurs derrière les décisions menant à l’application de ces normes.

Généralités

Les principaux buts des choix de nomenclature dans les programmes sont d’instaurer une culture d’entreprise, par laquelle il sera plus facile pour un développeur de s’adapter au code des autres, et de réduire les cas d’ambiguïté possibles dans les programmes.

Ainsi, si vous examinez le standard appliqué dans ce document, vous remarquerez que déclarer un attribut `orientation_` de type `Orientation` et de lui affecter la valeur du paramètre `orientation` se fait sans heurts car :

- le nom du type débute par une majuscule;
- le nom de l’attribut débute par une minuscule et se termine par un `_`; et
- le nom du paramètre débute par une minuscule et ne se termine pas par un `_`.

Ne faites pas de guerres de religion basées sur des questions de standards de nomenclature. Parmi plusieurs standards possibles pour une entreprise donnée, le meilleur sera celui auquel tous souscriront, dans la mesure où celui-ci ne provoque pas d’écritures ambiguës.

Mieux vaut (nettement) un consensus imparfait aux yeux de chacune mais qui permette de travailler et de se comprendre qu’une tentative de standard que personne n’accepte de respecter.

Certaines firmes ont des standards particuliers et précis, alors que d’autres prennent une approche plus relaxe, mais donnent quand même des standards généraux auxquels les équipes de développement devront se conformer.

L’objectif derrière cet appendice est d’aider à la réflexion et de présenter quelques points de vue communs ou répandus. Il est clair que, si vous devez vous intégrer à une firme existante, les standards déjà en place seront probablement pour vous ceux à privilégier d’abord et avant tout; à moins que votre tâche ne soit de refondre les standards en place, il est probable que vous soyez d’abord appelé(e) à comprendre et à assimiler les façons de faire locales d’abord, puis (peut-être un jour) à les réexaminer si elles ne vous semblent plus adéquates.

Une étrange conséquence

Il est important de noter que l’orientation objet a, entre autres conséquences, celle de mener les langages à supporter plusieurs sous-programmes portant le même nom. Une raison pour ceci est la mécanique de construction des instances, qui demande généralement qu’on puisse avoir plusieurs constructeurs distincts portant le même nom.

Sur le plan du code généré à la compilation, les noms doivent toutefois être uniques pour qu’on puisse différencier les sous-programmes à l’édition des liens. Pour arriver à des noms effectifs distincts mais reconnaissables, les compilateurs génèrent dans le code objet des noms de sous-programmes qui incluent de l’information quant aux types de leurs paramètres – de la *décoration*.

Ceci ajoute une strate de complexité à l’édition des liens, surtout lorsque survient le besoin de lier entre eux des fichiers objets générés à partir de langages de programmation et de compilateurs différents, puisque les standards d’encodage sont locaux aux technologies individuelles.

Pour faciliter les choses dans de tels cas, il arrivera qu’on introduise des sous-programmes globaux qui seront compilés selon les standards C (nom unique, nom dans le code objet correspondant directement au nom dans le code source) et qui serviront de lien entre les modules de diverses sources. Ceci résulte à l’occasion, côté OO, en de curieux hybrides, mais tel est parfois le prix à payer pour assurer l’interopérabilité binaire.

Groupement et nomenclature

On cherchera à grouper les éléments d'un programme sur la base de leur vocation ou de leur appartenance logique dans la mesure du possible. Ceci permet à un module de n'inclure que les éléments dont il fait véritablement usage, réduisant du même coup le couplage dans le programme. Après tout, pourquoi ajouter des dépendances artificielles dans un programme?

Mis à part les programmes les plus simples, la conception d'un fichier global regroupant toutes les constantes d'un programme (`constantes.h`, par exemple) est une mauvaise idée : toute modification à ce fichier entraîne une recompilation de tous les modules qui y ont recours, ce qui peut ajouter des heures au temps de compilation des systèmes de taille moyenne¹⁶².

Regrouper les entités dans un même fichier ou dans un même module logique constitue une forme implicite de documentation, pour laquelle le compilateur offre un support direct.

Groupement par modules

Si les modules de votre système sont de simples fichiers (ou des paires `.h/ .cpp`), alors l'appartenance à un groupement logique se fera souvent en deux temps, soit logiquement (placer dans un même module les constantes vouées à une même tâche ou à une même structure de données) et par une forme d'homophonie.

On pourrait suffixer toutes les constantes d'un groupe de la même manière (par exemple, par `MENU_` pour des mnémoniques de menus), mais on tendra généralement plutôt à regrouper les constantes à l'aide de préfixes, pour simplifier la recherche et les tris lexicographiques.

Si votre projet utilise des espaces nommés ou des paquetages, le groupement sera structurel plutôt qu'homophonique, facilitant les directives `using` ou `import`. Dans la majorité des langages OO contemporains, ces modules sont des espaces logiques ouverts, et on peut leur ajouter des éléments à partir de plusieurs fichiers. C'est d'ailleurs vrai à la fois pour les paquetages Java et pour les espaces nommés de C++ et des langages `.NET`.

Groupement par classes

L'approche OO offre aussi un outil de classement intéressant pour le regroupement des éléments de code logiquement apparentés : les classes et les membres de classe. Cette stratégie est prise à plein par Java et par les langages `.NET`, qui ne permettent pas de définir des fonctions globales.

Selon cette stratégie, donc, on spécifiera des classes ayant entre autres pour rôle de regrouper des constantes de classe publiques de vocation connexe et, le plus souvent, des méthodes de classe servant à convertir des données, valider des données, et autrement manipuler à la fois les constantes en question et les variables qui leur sont apparentées.

¹⁶² Compiler un programme de type académique ne prend typiquement que quelques secondes. Compiler la totalité d'un système bancaire de moyenne envergure peut prendre une douzaine d'heures. Compiler un simulateur de vol militaire tout entier peut prendre une journée entière. Ajouter des dépendances inutiles à un module a pour effet de compiler des modules qui, au fond, n'en ont pas besoin. Pensez à votre entreprise et à vos collègues et visez un couplage minimal chaque fois que cela s'avère raisonnable.

Exemple concret : la classe *Integer* de Java

Par exemple, Java propose la classe `Integer`, dont chaque instance représente (par un objet immuable) un entier (un `int` du langage Java). Avoir une version objet de chaque type primitif est nécessaire, en Java, pour un ensemble de raisons techniques¹⁶³.

Dans la classe `Integer`, on trouve :

- les constantes de classe `MAX_VALUE` et `MIN_VALUE`, indiquant les bornes de validité d'un `int`;
- plusieurs méthodes de classe transformant une instance de `String` (chaînes de caractères Java) en instance de `Integer`; et
- plusieurs méthodes de classe transformant un `int` (type primitif) en instance de `String`.

On y trouve aussi des constructeurs et des méthodes d'instance, bien sûr. Cela n'est par contre pas une condition d'application de la stratégie de regroupement dont nous discutons ici : on pourrait imaginer sans peine une classe ne servant qu'à exposer, de manière publique, un ensemble de constantes logiquement associées, et qui ne serait pas vouée à être instanciée.

Évitez toutefois à tout prix de donner dans l'excès et de transformer ces classes de regroupement en l'équivalent d'un tas de constantes globales!

Les méthodes de classe sont l'alternative Java aux fonctions globales de C++. Les constantes groupées par classe y sont l'alternative plus localisée aux constantes globales de C++.

De son côté, C++ a choisi de regrouper d'abord les données qui étaient globales avant la norme ISO dans des espaces nommés¹⁶⁴, sans empêcher qui que ce soit d'utiliser plutôt des classes à des fins de regroupement. Avec le temps, cela dit, l'évolution des pratiques a mené les programmeurs C++ à utiliser des traits [POOv02] pour les constantes associées à des types.

¹⁶³ En particulier, Java ne propose que le passage de paramètres *par valeur*. Des procédures comme `permuter(x, y)` y sont donc *a priori* impossibles à implanter à l'aide de types primitifs. Par contre, en Java, tous les objets alloués sont gérés par référence; passer une copie d'une référence fait en sorte de donner au sous-programme appelé un accès à la donnée telle qu'elle est connue du sous-programme appelant, sauvant ainsi la mise. Évidemment, la classe `Integer` étant immuable, on ne pourrait pas non plus écrire une méthode comme `permuter(Integer, Integer)`...

¹⁶⁴ Toute donnée, classe, fonction, constante, *etc.* n'étant pas spécifiquement dans une espace nommé est, en C++ ISO, dans l'espace nommé anonyme (drôle de nom, soit, mais correct tout de même). Ainsi, la fonction `sqrt()` de `<math.h>` porte le nom effectif `::sqrt()` (le préfixe `::` seul signifie que `sqrt()` est dans l'espace anonyme) alors que celle de `<cmath>` se nomme `std::sqrt()`.

Noms significatifs de constantes

Sur le plan programmatique, plusieurs normes usuelles veulent que les constantes symboliques portent des noms écrits entièrement en majuscules. Ceci tient principalement d'une tradition liée aux usages en pseudocode, qui cousine celle des langages de *Niklaus Wirth*.

Personnellement, j'utilise en pratique des constantes en majuscules pour les entités sur lesquelles je peux compter dès la compilation des programmes. Pour les autres, je nuance.

En pseudocode, où on omet de typer les données dans le but de se détacher de la technique pour réfléchir en termes algorithmiques, l'emploi de lexèmes entièrement en majuscules simplifie l'identification de ces données invariantes et importantes que sont les constantes.

Les constantes ont, toujours selon la tradition, une valeur connue à la compilation, *brûlée* dans le code, ce qui permet de les considérer comme un savoir *a priori*. Un compilateur peut utiliser le caractère invariant et littéral de ces données pour optimiser certaines partie d'un programme, par exemple, ou pour réserver l'espace associé à des tableaux.

Avec l'approche objet, où une instance doit être construite et détruite mais peut quand même être constante, et où des constantes locales à un sous-programme peuvent parfois être spécifiées sans que leur valeur ne soit connue à la compilation, la définition *de bas niveau* d'une constante s'élargit un peu.

Noms historiques ou associés à un objet courant

Certaines constantes ont des noms à caractère historique. On pense par exemple à `PI`, qui est traditionnellement associé à l'usage fait en mathématiques de la lettre grecque du même nom; à `NULL` pour un pointeur vers l'adresse `0` en langage C; ou à des constantes comme `TPS` ou `TVQ` qui, contrairement à `PI`, ont une valeur précise délimitée dans le temps et sujette à des changements occasionnels.

La tradition C, en fait, est d'utiliser les majuscules pour les macros. Le langage C, à l'origine, ne permettait pas de déclarer des constantes, et comptait sur le préprocesseur pour remplacer certains symboles par leur valeur littérale dans le code source avant la compilation.

Les constantes *systèmes*, connues telles quelles des langages de programmation (plutôt que définies dans une bibliothèque), sont parfois en minuscules. On pense ici au `null` de Java (pour identifier une référence non instanciée) ou aux `true` et `false` de Java et de C++, qui sont les littéraux acceptables pour une donnée booléenne, pour ne donner que des exemples communs. D'autres langages, comme `VB.NET`, utilisent une forme plus littéraire et proposent des constantes systèmes comme `Nothing`, l'analogue au pointeur nul dans ce langage.

Remarquez que ces « constantes » ont fréquemment en commun le fait... de ne pas tant être des constantes symboliques que des littéraux d'un type donné. Un autre objectif visé par cet emploi des minuscules est de réduire les risques de conflits avec les noms utilisés dans les programmes écrits à l'aide de ces langages.

Dénombrement, bornes et énumération

D'autres constantes servent à des fins d'énumération, quand la cardinalité des énumérables est finie (et, idéalement, assez brève) et quand chaque élément énumérable peut se faire attribuer, en propre, un nom significatif. On peut penser ici à des constantes pour les jours de la semaine (LUNDI à DIMANCHE, par exemple), les mois de l'année, les signes du zodiaque, *etc.*

On séparera par des caractères de soulignement les mots distincts qui composeront un nom de constante (p. ex. : JOURS_FERIES); c'est là une coutume à peu près universelle pour les constantes nommées à l'aide de majuscules.

On visera habituellement à ce que les constantes portent un nom qui dénote bien ce qu'elles représentent. Pour les constantes de spécification de taille ou de quantité, par exemple celles exprimant la taille d'un tableau, on prefixera généralement le nom de la constante par un vocable comme MAX_ ou NB_. Pensez à NB_ELEVES pour les élèves d'une classe, ou à MAX_ELEMENTS pour une collection de taille bornée.

Dénoter une valeur maximale ou minimale, comme dans le cas où on voudrait spécifier les bornes de validité d'un type de données, l'usage veut qu'on utilise plutôt MAX et MIN comme suffixes.

Si on veut indiquer la note minimale et la note maximale acceptables pour un type Note, on pensera souvent à NOTE_MIN et NOTE_MAX.

Règle générale, une écriture comme NOTE_MIN sera préférable à une écriture comme MIN_NOTE du fait que NOTE_MIN et NOTE_MAX seront alors côte à côte en ordre lexicographique.

Dans le même ordre d'idées, on choisira souvent d'identifier la valeur par défaut pour un type de données par une constante dont le nom est suffixé par DEFAULT. Dans le cas d'une Note, on reconnaîtra immédiatement NOTE_DEFAULT comme la valeur par défaut utilisée (ou à utiliser) pour ce type.

L'emploi d'une stratégie OO permet de compartimenter les noms de manière à limiter le recours à de tels préfixes. En effet, si une classe Note contient des constantes MIN et MAX et s'il est clair qu'il s'agit des valeurs minimum et maximum pour une Note, alors la classe jouera souvent le rôle de qualificateur.

Après tout, Note::MAX est aussi clair que NOTE_MAX.

Noms significatifs de variables

Les problèmes informatiques ayant tendance à s'étendre sur des problèmes vastes et où apparaît la nécessité de gérer une masse très importante d'identifiants (constantes et variables, fonctions, types et classes, *etc.*), on évitera le plus possible les noms de variables d'une seule lettre ou les monosyllabiques, et dans les rares cas où on les utilisera, on fera en sorte que l'usage soit de portée *très* locale¹⁶⁵.

Dans les cas où la tradition les motive (on pense immédiatement au cas des boucles `for` avec un compteur d'itérations comme `i`, `j` ou `k`), on s'assurera de ne *jamais* utiliser des variables globales portant un nom non-significatif¹⁶⁶. En fait, n'utilisez pas de variables globales tout court, à moins d'être en sérieuse détresse, et donnez-vous plusieurs opportunités de réévaluer votre décision lorsque le stress aura diminué.

On suggère généralement aussi d'utiliser des noms qui seront familiers aux gens susceptibles d'œuvrer auprès du même code source, dans l'immédiat comme dans le futur envisageable. On voudra éviter l'emploi de terminologie échappant au commun des gens appelés à collaborer à un module, à moins bien sûr que ladite terminologie ne soit de nature technique et qu'elle soit intimement liée au projet en développement.

Cela dit, même dans un tel cas, il ne faut pas négliger l'importance des commentaires accompagnant la déclaration, qui devront donner une chance aux gens appelés à contribuer au projet de saisir le rôle de la variable malgré un bagage *a priori* différent.

Tel que mentionné dans la section *Instanciation tardive*, plus haut, une saine approche dans les langages OO est d'instancier les objets aussi près que raisonnable de leur point d'utilisation, et aussi localement que possible (en faisant attention, au passage, à la performance du programme si un objet doit être construit et détruit fréquemment). En plus de mener à des programmes dans lesquels les objets ne sont construits que si cela s'avère vraiment nécessaire (donc des programmes qui gèrent sainement leurs ressources), cette approche clarifie *de facto* le lien entre le nom d'une variable (ou d'une constante) et le code qui s'en sert.

¹⁶⁵ De manière complémentaire à la mathématique, où l'humain est penseur et acteur principal de la recherche d'une solution, d'une stratégie ou d'une démonstration, l'informatique vise souvent les tâches et problèmes où l'automatisation doit primer sur l'intervention humaine, incluant les situations où le volume de données ou la taille du problème dépassent les capacités des humains. On remarque aussi que les mathématiciennes et les mathématiciens sont souvent appelés à résoudre des problèmes d'ordre général, s'appliquant à une classe de situations très large, et à amener des solutions opérant de fait à un niveau d'abstraction élevé si on le compare aux situations réelles dans lesquelles elles seront, concrètement, appliquées. La signification des noms de variables est alors souvent détachée des contextes d'application, ce qui est raisonnable et compréhensible. Les problèmes auxquels s'attaquent l'informaticienne et l'informaticien sont parfois de cet ordre, mais peuvent plus souvent être liées à des situations pour lesquelles des variables aux noms monosyllabiques ou écrites d'un ou deux caractères seulement seraient moins appropriées. On recommande donc fortement, en informatique, de faire usage pour une variable de noms significatifs à son rôle dans le contexte de son utilisation.

¹⁶⁶ Cas documenté d'un tel problème : oublier de déclarer un compteur nommé `i`, ne pas s'en apercevoir puisque le programme compile (dû à l'existence d'une variable *globale* nommée `i`) donc, par inadvertance, modifier un compteur global utilisé *ailleurs*, ce qui introduit un bogue majeur et très, très difficile à identifier.

Composition de noms

Il arrive fréquemment qu'un nom de variable soit en fait un composé de plusieurs noms. On essaiera d'utiliser, pour nommer ces variables, un standard qui mette en relief les noms qui les composent. Ceci peut signifier séparer les mots par des caractères de soulignement, utiliser une majuscule pour débiter chaque mot, ou une combinaison des deux (voir l'encadré ci-dessus pour mes suggestions).

Selon les standards et les conventions choisies localement, on utilisera parfois une majuscule pour débiter un nom de variable, et parfois une minuscule. L'important ici est (a) de respecter les standards locaux, et (b) qu'il existe ou non des standards locaux, d'être rigoureux et conséquent(e) dans le respect de la démarche choisie.

On évitera bien sûr les noms de variables écrits strictement en majuscules, fuyant les ambiguïtés avec les conventions en place pour les constantes. Si un programme devait calculer une approximation de π plutôt que le la prendre pour acquis, utilisant ainsi une variable plutôt qu'une constante, on s'attendrait à voir cette variable nommée `Pi` ou `pi`, pas `PI`.

Plusieurs firmes suggèrent que, même dans les langages où les lettres majuscules et les minuscules sont considérées comme des symboles différents¹⁶⁷, on évite d'utiliser deux variables portant exactement le même nom aux majuscules et aux minuscules près (cette suggestion s'étend au couple variable/ constante aussi, de manière évidente).

En ce qui concerne les majuscules, les minuscules et les caractères de soulignement pour les variables représentant un nom composé comme *borne minimum*, certains recommandent `BorneMin`, d'autres `borneMin`, d'autres encore `borne_min` et d'autres, plus zélés, utiliseront `Borne_Min`. En général, la clé de la réussite est de se donner des standards d'entreprise et de les respecter.

Les noms et les verbes

Les variables tendent à désigner des états; de ce fait, on espère d'une variable que son nom donne une idée immédiate de son rôle dans le contexte où elle est destinée à servir. On évitera de donner à une variable un nom évoquant un verbe d'action ou un verbe d'état¹⁶⁸. Les verbes ayant un sens dynamique, on les réservera généralement pour les noms de sous-programmes.

Une exception importante, sur laquelle nous reviendrons dans [POOv02] : les foncteurs, qui sont à la fois des objets et des fonctions.

¹⁶⁷ Les langages C, C++, Java, mais pas VB6 ou Pascal par exemple.

¹⁶⁸ Des exceptions sont possibles, comme celui d'une variable booléenne nommée `doitPoursuivre` ou `poursuivre` et servant à contrôler la poursuite de l'itération d'une répétitive. Cela dit, on pourrait y aller par la négative et utiliser plutôt la négation d'une variable nommée `fini` ou `termine` (terminé si votre langage accepte les caractères accentués; *done* ou *completed* en anglais) pour signifier précisément la même chose.

Variables jetables

Certaines firmes recommandent aussi de définir des standards pour les variables dites *jetables* (en anglais : les *Throwaway Variables*). On verra alors des variables préfixées par le terme `dummy`, par exemple, ou par `spare` pour des variables servant à réserver de l'espace en vue d'ajouts à un système dont la taille ne doit pas, pour différentes raisons, varier trop souvent. Typiquement, les variables d'une seule lettre comme `i`, `j` ou `k` entrent dans cette catégorie.

Conserver des variables jetables est plus fréquent dans certains contextes spécialisés, par exemple les systèmes embarqués, où la mémoire qui sera utilisée dans le programme est calculée avec soin. Planifier quelques variables jetables clairement identifiées permet aux développeurs de mettre leurs systèmes au point et de les déverminer sans nuire à l'économie globale des ressources dans le programme sur lequel ils œuvrent.

Clarté ou concision?

L'emploi de noms significatifs doit être balancé par un souci de clarté et de concision. Entre `compteurDuNombreDEleves` et `compteurEleves` (ou, si la pratique locale le permet, `cptEleves`), on ne remarque pas de réelle perte de sens : il est clair dans chaque cas qu'on utilise cette variable pour compter ou énumérer des élèves.

Les formules brèves entraînent ici une présentation plus claire pour le code pris dans son ensemble, et sont plus directement accessibles à la lecture. En retour, les noms `cptE` ou `i` ne véhiculent pas du tout un sens aussi clair et complet; ne les utilisez que si leur sens est évident dans le contexte où ils apparaissent.

La modération et l'équilibre sont des qualités recherchées ici pour supporter les efforts de clarté, pas pour aller à leur rencontre.

Bien que plusieurs langages supportent des noms d'identifiants d'une longueur de 127 ou de 255 caractères, on voudra idéalement se limiter à moins de 30 caractères pour ménager ces pauvres humains qui les liront et qui en feront usage.

Un bon truc pour celles et ceux qui préfèrent des noms concis est d'écrire des fonctions courtes, dont le sens est immédiatement évident. Dans une fonction de deux ou trois lignes destinée à un usage général, des noms d'indices comme `i`, `j` et `k` ou des pointeurs nommés `p` et `q` sont tout à fait convenables et généraux, alors que dans une fonction d'un plus grand nombre de lignes ces noms de variables sont inadéquats.

Préfixes, suffixes et autres décorations

Lorsque le besoin de variables globales survient, ce qui survient à l'occasion pour des raisons techniques, plusieurs ont pris l'habitude de préfixer leur nom par `g_`. Les variables globales étant à éviter le plus possible, on vise à les repérer avec aisance dans un programme.

Conséquemment, on évitera pour les variables locales les préfixes comme `s_`, `m_` ou `g_`, du fait que plusieurs les réservent pour spécifier des attributs de classe, d'instance ou des variables globales. Cette mise en garde demeure, que votre firme fasse partie ou non des utilisateurs de ce standard : l'idée est de réduire les risques de conflits et d'ambiguïté.

La notation hongroise

Vous avez peut-être entendu parler d'un standard de nomenclature nommé **notation hongroise**, que certains appellent aussi *notation polonaise inversée* (voir l'encadré à droite), et qui a quelques dérivés¹⁶⁹. Bien qu'en disparition, il s'agit d'un standard qui a connu ses heures de gloire, en particulier pendant la transition des programmeurs C vers C++.

La confusion de nom avec celui de notation polonaise inversée tient d'un système mathématique du même nom et servant à faciliter, surtout dans des calculatrices, l'entrée d'expressions sans parenthèses ou ponctuation. Cette notation tend à donner des « mots » pleins de consonnes.

Il vaut la peine de faire ici une courte digression à ce sujet, pour expliquer :

- de quoi il s'agit;
- ses avantages; et
- ses inconvénients, le tout dans un contexte OO.

L'idée est de vous aider à vous forger un point de vue sur le sujet, pour éviter de tomber malgré vous dans des guerres de clocher à son sujet.

Les remarques qui suivent, exception faite bien sûr des notes historiques propres à la notation hongroise en tant que telle, s'appliquent de manière générale à toute notation avec laquelle on inscrit, à même les noms d'identificateurs, des données descriptives de type.

Ce que vous ne trouverez pas ici, c'est une position préfabriquée du genre *c'est la meilleure chose au monde* ou *c'est une tare à éviter*. Comme c'est souvent le cas, une position réaliste se situe quelque part entre les deux.

¹⁶⁹ On connaît une notation tchèque, ou *Czech Notation*, entre autres choses.

Bref historique

La notation hongroise est une manière d'identifier chaque donnée par un préfixe qui en spécifie le *type*, donc d'indiquer le type d'une donnée à même son nom. Clairement, on parle ici d'une pratique vouée à aider les développeurs à se souvenir des types en jeu lorsqu'ils manipulent des variables, et ce sans avoir à reculer jusqu'à leur déclaration.

Comme toute chose sans doute, cette notation est née en réponse à un besoin : dans un programme `OO` mettant l'accent sur l'instanciation tardive, elle serait redondante.

Cette notation a été développée par un hongrois (d'où le nom de la notation, on s'en doute) nommé **Charles Simonyi**. Elle est surtout connue aujourd'hui parce que son inventeur a œuvré une dizaine d'années comme programmeur senior à *Microsoft*, et que les programmes écrits en langage C¹⁷⁰ sous *Microsoft Windows* l'ont adopté comme standard¹⁷¹.

Par intérêt historique, notons l'un des premiers volumes expliquant comment programmer sous *Microsoft Windows*, titré *Charles Petzold's Programming Windows*, utilisait massivement un dialecte de la notation hongroise, ce qui a sûrement motivé beaucoup de gens à faire de même.

Le langage C est un langage d'une grande souplesse, qui estime les programmeuses et les programmeurs responsables de ce qu'ils écrivent. Ce faisant, ce langage permet plusieurs manœuvres dangereuses, comme traiter une donnée d'un type comme s'il s'agissait d'une donnée d'un autre type. Bien saisir le type des données y est donc crucial.

L'inventeur de la notation, donc, était d'avis que l'emploi systématique de noms de variables préfixés de leur type entraînerait une diminution des erreurs sur ces variables dues à des manipulations impropres. Il est en effet difficile de justifier devant un employeur d'avoir mal compris le rôle d'une variable contenant du texte si le nom de cette variable indique clairement qu'il s'agit de texte, et pas d'un entier par exemple.

¹⁷⁰ Le langage C impose la déclaration de variables au début de blocs seulement, contrairement à C++ qui permet la déclaration de variables à peu près n'importe où dans un programme, et il arrive que, dans un sous-programme de longueur importante, la déclaration d'une variable donnée soit loin de l'endroit où elle est utilisée.

¹⁷¹ La notation elle-même, par contre, n'a pas été mise au point durant le passage de son inventeur chez *Microsoft*, mais alors qu'il travaillait à l'université Berkeley; il la raffina ensuite alors qu'il œuvrait aux laboratoires de recherche PARC (de *Xerox*).

Problèmes de décision de nomenclature

Le poste de Simonyi chez *Microsoft* lui a permis de tester son modèle, l'imposant comme standard de travail pour son équipe¹⁷². Selon lui, le problème fréquent qu'est le choix d'un nom pertinent et utile pour une variable mène les développeurs à considérer les critères suivants :

- la *valeur mnémonique* du nom, à savoir *pourra-t-on se souvenir avec aisance du nom choisi?*
- la *valeur suggestive* du nom, à savoir *d'autres en tireront-ils suffisamment d'information immédiate à la lecture du code?*
- la *conséquence*, à savoir *si les noms de variables et de quantités apparentées seront-ils eux aussi apparentés?* et
- la *rapidité de décision* et d'écriture, à savoir *arrivera-t-on rapidement à un nom qui rencontre ces exigences?* et *les noms résultant seront-ils suffisamment brefs pour être rédigés sans erreur par des programmeurs?*

La notation hongroise se veut un standard permettant de répondre à l'ensemble de ces critères, mécanisant ainsi en grande partie la génération de noms et standardisant d'office les noms en regroupant les variables selon leur type.

¹⁷² Ses collègues, voyant les noms de variables préfixées par un tas de consonnes (comme par exemple `lpszFile` pour un nom de fichier) auraient lancé, en boutade: «*ça pourrait être du grec... ou du hongrois!*».

Aperçu de la notation

La notation hongroise telle qu'employée communément aujourd'hui ressemble à :

Préfixe	Type	Exemple
b	bool (ou, C, int servant de booléen)	bool bStillGoing;
c	char	char cLetterGrade;
str	string de C++	string strFirstName;
si	short int	short siChairs;
i	int	int iCars;
li	long int	long liStars;
f	float	float fPercent;
d	double	double dMiles;
ld	long double	long double ldLightYears;
sz	Chaîne ASCII brute	char szName[NAME_LEN];
if	Flot d'entrée d'un fichier	ifstream ifNameFile;
is	Flot d'entrée	void fct(istream &risIn);
of	Flot de sortie d'un fichier	ofstream ofNameFile;
os	Flot de sortie	void fct(ostream &rosIn);
h	HANDLE (pointeur opaque)	HANDLE hThreadLecture;
hwnd	HANDLE de fenêtre	HANDLE hwndMaFenetre;
S	Déclaration d'un struct	struct SPoint { ... };
C	Déclaration d'une class	class CPerson { ... };
Nom ou abréviation d'un struct ou d'une class	Déclaration d'une instance de...	SPoint pointLeft; ou SPoint ptLeft; CPerson personFound; ou CPerson perFound;

Il est commun de voir des ajouts locaux à la liste de préfixes¹⁷³.

On peut préfixer ces préfixes (eh oui!) des *pré-préfixes* suivants :

Pré préfixe	Type de données	Exemple
u	unsigned	unsigned short usiStudents;
k	Paramètre constant	void proc(const long kliGalaxies);
r	Paramètre par référence	void proc(long &rliGalaxies);
s	static	static char scChoice;
rg	Tableau (pour intervalle, range)	float rgfTemp[MAX_TEMP];
m_	Membre de ...	char m_cLetterGrade;
p	Pointeur de...	char *pcGrade;
prg	Tableau alloué dynamiquement (mélange de p et de rg)	char *prgcGrades;

¹⁷³ Par exemple *w* pour le type `WORD`, un entier 16 bits sous *Microsoft Windows*, ou *dw* pour `DWORD`, entier 32 bits sous cette même plateforme.

Raison d'être et avantages

L'avantage premier de l'emploi de la notation hongroise, comme de l'emploi de toute notation, est que son déploiement systématique introduit *de facto* une normalisation au niveau des règles grammaticales locales.

Une telle discipline tend à faciliter la formation des gens, ce qui explique que plusieurs enseignants l'imposent aux étudiant(e)s en début de formation, et à diminuer le temps requis pour former des individus familiers avec ces règles lorsque celles-ci ou ceux-ci doivent aborder un nouveau produit. On peut ne pas aimer un standard ou l'autre, pris individuellement, mais il est généralement bien, dans un domaine aussi vaste et mouvant que l'informatique, d'appliquer des standards. Même les plus individualistes et les plus rebelles du domaine reconnaissent les bienfaits d'une démarche de standardisation¹⁷⁴.

Les partisans et les partisanes de la notation hongroise apprécient le surcroît d'information immédiate que leur apporte l'ajout d'informations de type à chaque nom de variable. Particulièrement pour qui n'a pas intégré une discipline de nomenclature implicite à même ses méthodes de travail, un rappel continu, formel et explicite des choix d'implantation faits à même le nom de chaque variable peut accélérer le développement, en diminuant le temps de recherche et de débogage associé à cette tâche.

Une certaine simplification de la documentation d'un programme peut être apportée par l'emploi rigoureux d'une norme comme la notation hongroise, du fait que les types de données en jeu peuvent être considérés comme implicites (donc, à la limite, ne pas devoir être mentionnés en propre en support à la description des variables) ou encore être extraits avec facilité par des utilitaires de documentation automatique¹⁷⁵.

À ne pas négliger, parce qu'il faut être réaliste (même si c'est, moralement, très irritant) : le standard en question était véhiculé par une firme dont le poids, sur le marché, est considérable. Les idées de cette firme, qu'elles soient bonnes ou mauvaises, dangereuses comme avantageuses, tendent à se frayer un chemin dans le public, entre autres à travers les nombreuses PME qui dépendent d'elle pour leurs propres produits. L'impulsion apportée par l'adhésion de ce poids lourd à la notation hongroise a aidé à la dissémination et l'acceptation de cette dernière.

Les défenseurs de la notation hongroise (et de ses dérivés locaux) indiqueront que, dans la mesure où cette notation est utilisée de manière *conséquente* et *cohérente*, elle évitera aux développeurs de la rédaction de code inutilement long, les noms de variables devenant relativement compacts, et diminuera ainsi le nombre d'erreurs de frappe.

¹⁷⁴ C'est la raison d'être de cette section que de discuter standards et normes de notation, n'est-ce pas?

¹⁷⁵ Remarquez qu'un utilitaire de documentation automatique peut, s'il est capable de décoder la structure d'un programme (donc s'il connaît les règles grammaticales du langage de programmation en jeu) extraire l'information de type d'une variable de par sa déclaration, surtout si le langage est fortement typé. Par contre, certains peuvent trouver plus simple, à certains égards, de rédiger un utilitaire déduisant les types des variables de par leur nom que d'en écrire un autre qui doit comprendre plus en détail la structure d'un programme. Ce qui est sûr, c'est que de combiner la notation hongroise à des utilitaires de validation et de documentation automatiques peut aider au dépistage d'erreurs sémantiques dans un programme.

Évidemment, il faut veiller à ce que la correspondance entre nom et type demeure véridique en tout temps. Tout changement au type d'une variable *doit* entraîner une modification immédiate de son nom, et ceci partout où elle est utilisée, pour la notation hongroise demeure utile. C'est d'ailleurs l'une des raisons pour lesquelles cette norme est aujourd'hui en désuétude (voir *Inconvénients*, plus bas) : à long terme, assurer cette correspondance est un cauchemar.

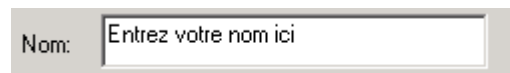
À ce sujet, mêmes les amateurs de la notation hongroise sont conscients que cette manière de nommer les variables est, essentiellement, un standard de documentation, équivalent fonctionnel des commentaires. Et comme c'est le cas pour les commentaires, la conformité du nom d'une variable à son type de donnée n'est pas validée par les compilateurs¹⁷⁶; conséquemment, les développeurs sont à 100% responsables d'assurer cette conformité, ce qui (surtout lorsque les échéances de livraison d'un produit approchent) peut être difficile à faire. Écrire du code si clair qu'il ne demande pas de commentaires est une vertu : voir *À propos des commentaires*, plus bas.

Le cas particulier des interfaces personne/ machine

Dans certains cas, comme par exemple celui du développement d'interfaces personne/ machine (IPM), l'emploi de standards de notation préfixée est presque une nécessité. On pensera en particulier aux outils RAD (pour *Rapid Application Development*) permettant d'appliquer des techniques de prototypage rapide, comme C# et VB.NET par exemple, pour lesquels existent souvent des normes locales pour les noms de contrôles.

Dans une IPM, on trouvera presque toujours plusieurs contrôles voués, chacun à sa façon, à un mandat commun. Ce problème est analogue à celui de la nomenclature des types, des attributs d'instance et des paramètres en POO.

Par exemple, si on joint une étiquette de texte statique¹⁷⁷ indiquant d'entrer un nom à une zone de texte¹⁷⁸ permettant d'entrer ce nom, on obtient deux contrôles voués à l'entrée d'un nom.



L'emploi de préfixes normalisés pour les contrôles¹⁷⁹ permet une différenciation immédiate de chacun d'eux. Il s'agit sans doute de la manière la plus simple pour arriver à une telle différenciation, du fait que des noms dépendant des rôles des contrôles ne suffirait pas.

¹⁷⁶ Ceci n'est pas incontournable, remarquez bien.

¹⁷⁷ Un Label en VB6, un JLabel en Java (avec Swing).

¹⁷⁸ Un JTextField en Java (avec Swing), un TextBox en VB6.

¹⁷⁹ Ceux de VB6 sont lbl pour une étiquette et txt pour une zone de texte. Ici, par exemple, on aurait lblNom et txtNom, ce qui donne immédiatement le rôle de chacun de ces contrôles.

Inconvénients

La lisibilité initiale de code rédigé selon les règles de la notation hongroise ou de toute autre stratégie de décoration manuelle des noms est ardue. Les préfixes sont faits d'une ou de plusieurs consonnes, et rebutent la plupart des gens qui, tout en gagnant l'information de types qu'elle véhicule, y perdent souvent le sens de la variable, du moins pendant leur période d'acclimatation. Cela dit, comme toute norme, on s'y habitue, et mieux vaut une forme rugueuse d'approche que pas de norme du tout.

Sur le plan programmatique, on peut mettre en doute l'utilité de cette notation dans les programmes rédigés dans un langage OO contemporain pour plusieurs raisons. Entre autres :

- on s'attend à ce que, dans un programme bien écrit, les sous-programmes tendent à être courts et à avoir une seule vocation, claire et précise. Dans un tel contexte, la déclaration d'une variable ne devrait jamais être très loin du lieu de son utilisation;
- même dans une longue méthode, il devrait être préférable de recourir à l'instanciation tardive plutôt que de déclarer des variables et de constantes loin de leur point d'utilisation;
- dans un contexte OO, l'accès direct aux attributs devrait se restreindre aux accesseurs et aux mutateurs, règle à laquelle on ne déroge habituellement que par paresse¹⁸⁰;
- modifier le type d'une variable signifie, dans un projet pour lequel on a choisi d'appliquer la notation hongroise, la renommer, et ce partout où elle est utilisée. L'utilisation de générateurs de code intelligents (et qui conviennent à nos standards locaux, ce qui est plus rare qu'il y paraît) ou de moteurs de refactorisation (*Eclipse*¹⁸¹, par exemple, ou certaines versions de *Visual Studio*) devient alors pratiquement une nécessité; ça, ou beaucoup de patience (pour procéder aux modifications rendues obligatoires) et de chance (pour espérer que le design initial tienne pratiquement en totalité);
- l'extensibilité du modèle laisse à désirer. La plupart des langages de programmation, à plus forte partie les langages OO, permettent aux développeurs de définir leurs propres types de données. L'une des raisons d'être de cette démarche est d'abstraire les types d'origine pour faciliter leur remplacement. Par exemple, l'emploi d'un type `Quantite` pour représenter une quantité, susceptible d'être un type entier mais dont la taille reste à négocier. Créera-t-on un nouveau préfixe local, ou apposera-t-on le standard hongrois, quitte à renommer toutes les variables de type `Quantite` à chaque modification du type lui-même? Ou évitera-t-on plutôt d'utiliser cet outil pourtant fort intéressant qu'est l'idée de type maison?

Irritant concret : la notation hongroise est une liée à un langage, le langage C, et utilise des préfixes qui ont du sens d'abord et avant tout pour ce langage. Des dérivés existent pour C++¹⁸², proche de C pour le nom de ses types primitifs, incluant l'apposition du préfixe `C` pour une classe (`CRectangle` pour une classe représentant un rectangle, par exemple).

¹⁸⁰ Prudence tout de même : la philosophie de certaines firmes fait qu'on y motive les bris d'encapsulation, prétendant ainsi simplifier ou accélérer le code. C'est souvent faux, de manière évidente et mesurable, mais il faut faire avec si on compte utiliser leurs produits.

¹⁸¹ Voir <http://www.eclipse.org/>

¹⁸² D'ailleurs, la version de la notation qui est proposée plus haut est, clairement, une version C++.

Conséquemment, la notation hongroise n'est pas directement transportable à d'autres langages, et on peut se retrouver, dans une firme où le développement se fait à l'aide de plusieurs langages distincts, avec une multiplicité de normes.

Microsoft, chez qui la notation hongroise est bien ancrée, a en partie contourné le problème dans son architecture .NET en uniformisant les types de données à même un standard binaire commun à tous leurs langages de programmation (le CLR, pour *Common Language Runtime*).

Plus irritant, d'un point de vue OO : l'emploi de noms variables incluant une information de type va à l'encontre de l'objectif d'abstraction des types. Les langages comme C++ visent à rendre *opérationnellement équivalents* les types complexes et les types simples. On l'a vu avec les opérateurs, la mécanique derrière les constructeurs par copie, le fait que l'allocation automatique de tableaux d'objets fait en sorte d'appeler spontanément le constructeur par défaut de chaque instance créée, et ainsi de suite.

L'emploi d'une notation stigmatisant les objets d'un préfixe (la lettre C) est, dans un contexte objet, très questionnable. Viser l'équivalence opérationnelle des types supprime le sens d'une notation comme la notation hongroise : sur le plan opérationnel, il n'existe pas de distinction entre les types valeurs *maison* de types primitifs. La démarche de programmation vise par définition à éliminer les distinctions opérationnelles entre les types; insérer artificiellement une distinction de ce genre à l'aide d'une convention de nomenclature est contradictoire.

Même avec les langages hybrides, utiliser la notation hongroise est un bris d'encapsulation. Plutôt que d'utiliser une entité pour ce qu'elle peut faire et ce qu'elle peut apporter à un programme, en pensant simplement aux opérations qu'on peut réaliser sur elle ou avec elle, il devient nécessaire de se préoccuper en tout temps de son type.

Même en langage C, dès les années '70, on avait perçu le besoin d'abstraction pour faciliter la maintenance du code à long terme. Les tailles standard du langage C sont de type `size_t`, un type non signé défini dans `<stddef>` (entre autres) et capable de représenter le nombre d'éléments dans un tableau. Rien n'indique s'il s'agit d'un `unsigned short`, d'un `unsigned int` ou d'un `unsigned long`. Le choix d'éviter de camper les tailles dans un type ou l'autre est délibéré, et de motiver les gens à manipuler les types à l'aide d'opérations (fonctions, en langage C) définies sur eux.

Même au cœur de *Microsoft Windows*, le maintien de la conformité des variables aux standards de la notation hongroise n'a pas passé la rampe de la migration du produit *Windows* de sa version 16 bits à sa version 32 bits.

En effet, le prototype de la fonction centrale de traitement des messages d'une fenêtre, `WinProc()`, reçoit entre autres choses deux paramètres, nommés `wParam` et `lParam`. Il se trouve que le préfixe `w` tient pour `WORD`, un entier 16 bits selon la tradition sur cette plateforme. Or ce paramètre est maintenant fait d'un entier encodé sur 32 bits, et aurait dû être renommé `dwParam`.

Ceci met en relief la difficulté d'adapter une notation comme la notation hongroise à un contexte évolutif de mise à jour de produit.

Pour réfléchir

À noter que, dans les livres de *Microsoft Press*[®] portant sur la stratégie .NET, on mentionne maintenant que la notation hongroise devrait être considérée obsolète.

Tristement, les raisons énoncées sont très mauvaises : la position de la firme qui a moussé la notation hongroise en premier lieu est que la technologie *IntelliSense*[®], qui met en relief des données sur chaque variable dans l'éditeur .NET alors que le pointeur de souris les survole, rendrait à leur avis l'emploi d'une notation préfixée redondante.

Pourquoi cette raison est-elle mauvaise? Entre autres :

- parce que les technologies de type *IntelliSense*[®] existent depuis longtemps. C'est l'une des nombreuses technologies qui était devenue, sous d'autres noms, un standard d'office un peu partout en informatique avant que l'environnement *Visual Studio* ne la propose. Pourquoi rendrait-elle *maintenant* redondante une approche de notation préfixée?
- parce que cette position présume que plus personne ne lira de code imprimé;
- parce que cette position présume qu'un seul environnement intégré d'édition soit officiellement viable.

La véritable raison pour rejeter une notation préfixée, si on la rejette effectivement, n'est pas une raison technologique, mais bien une raison de démarche :

- si l'on restreint les sous-programmes à des dimensions courtes (une page imprimée ou moins chacun, mis à part les sélectives – et encore!);
- si l'on respecte rigoureusement le principe de localité (dont nous discutons brièvement dans *Annexe 04 – Principe de localité et efficacité*), évitant au maximum les déclarations globales;
- si l'on applique avec rigueur une démarche par laquelle on choisit des noms de variables ou d'attributs significatifs;
- si l'on applique l'instanciation tardive;
- alors une notation préfixée devient redondante.

On peut comprendre l'emploi d'une notation préfixée dans un milieu d'apprentissage, où ces méthodes de travail ne sont pas encore bien ancrées (parce qu'on les développe). Et on doit aussi comprendre que l'ajout de technologies de support n'est pas un palliatif valable à une démarche rigoureuse de développement.

Pour en savoir plus, je vous suggère [SutterHung].

Noms significatifs d'attributs

Dans une approche objet, on vise à décourager l'accès direct aux attributs d'une instance. L'emploi de la spécification *privé* va bien sûr en ce sens; il se trouve que dans le passé pas si lointain, certaines expériences OO à vocation commerciale ne supportaient pas, à l'origine, l'encapsulation stricte à même le langage.

Pour motiver les programmeuses et les programmeurs à éviter l'accès direct aux attributs, les gens se sont alors mis à leur apposer systématiquement des préfixes et des suffixes, souvent des caractères de soulignement. L'idée était d'envoyer un message : *si tu leur touches directement, je vais leur ajouter ou leur enlever des symboles et ton code ne compilera plus; bien fait pour toi!* Selon les programmeuses et les programmeurs, on en trouvait un, puis deux, parfois même trois, toujours dans l'optique de mettre en relief que les accesseurs et les mutateurs, eux, portaient des noms stables et effectuaient les *bonnes* opérations de la *bonne* façon. Une forme d'encapsulation par la force brute, disons.

Avec la généralisation du support à l'encapsulation stricte à même les langages de programmation, on a développé l'habitude de standardiser les noms d'attributs de manière plus stable et plus légère.

Avec la popularité de la notation hongroise, il y eut un certain temps une tendance à préfixer les noms d'attributs d'instance de `m_`, au sens de *membre*, et cela même si le mot *membre* s'applique tout autant aux méthodes qu'aux attributs¹⁸³ (on ne préfixe pas les méthodes de cette façon). Aujourd'hui, en C++, l'usage est plutôt d'utiliser le suffixe `_`; la bibliothèque *Boost*¹⁸⁴, superbe projet de développement, est en partie responsable de ce nouveau standard.

Certains ont pris l'habitude de préfixer les membres de classe en Java et en C++ d'un `s_`, bien qu'il s'agisse là d'une habitude de travail beaucoup moins répandue. La plupart des attributs de classe tendent en effet à être des constantes, pour lesquelles un préfixe du genre n'a pas vraiment de pertinence.

Cette pratique, relativement rare, a un côté agaçant : le terme `static` pour un membre de classe est un terme technique qui n'évoque pas clairement le caractère *membre de classe* qui lui est associé. Le préfixe `s_` est donc très lié à une culture locale, et qui est difficilement transférable sans perte de sens.

Pour le reste, on essaiera de lier la nomenclature des attributs aux conventions de nomenclature en vogue dans l'entreprise.

¹⁸³ Historiquement, les présentations de C++ utilisent les vocables *Member Variable* et *Member Function* pour parler respectivement d'un attribut ou d'une méthode; ceci pour faciliter la transition vers C++ des programmeurs structurés, surtout ceux, nombreux, qui sont d'abord habitués au langage C.

¹⁸⁴ Voir <http://www.boost.org/>

Noms significatifs de sous-programmes globaux

Le premier critère de bonne nomenclature pour un sous-programme est de s'assurer que le nom soit un reflet fidèle de la tâche que ce sous-programme doit réaliser. Ceci peut sembler évident, mais un examen sommaire des pratiques en entreprise montre que la plupart des sous-programmes existants sont déficients en ce sens.

Les divergences entre nom et vocation pour un sous-programme tiennent surtout de deux facteurs, chacun important à sa façon :

- un découpage insuffisant du code, menant à des sous-programmes dont la tâche n'est pas claire et précise. De tels sous-programmes sont, conséquemment, difficiles à bien nommer;
- une maîtrise insuffisante de la langue d'usage, qu'il s'agisse du français, de l'anglais ou d'une autre langue. C'est là, tristement, un problème beaucoup trop commun.

On essaiera d'utiliser des verbes d'action pour débiter la plupart des noms de sous-programmes, ces entités étant pour la plupart des unités de code actives, quoiqu'on puisse imaginer des fonctions dont le nom est bien choisi mais débute par un verbe d'état (par exemple la fonction `est_pair(n)` qui retourne vrai si et seulement si `n` est un entier pair).

On a longtemps choisi des fonctions portant des préfixes similaires pour faciliter la documentation et l'organisation du code. Avec les langages OO, on tend à ne plus se préoccuper autant de considérations du genre du fait que, par nature, les méthodes sont naturellement regroupées sous l'égide de la classe à laquelle elles appartiennent.

Lorsqu'un projet est réfléchi selon l'approche OO mais à l'aide de langages qui, eux, ne sont pas des langages OO, on fera souvent attention de préfixer les fonctions destinées à devenir des méthodes par le nom de ce qui, selon la conception de l'équipe de développement, deviendra la classe à laquelle ces méthodes seront rattachées. Cela facilitera la migration vers un outil OO lorsque celle-ci sera entreprise.

Les fonctions globales et les méthodes publiques ont ceci en commun qu'une fois livrées, elles doivent être maintenues par la firme ou les individus qui en sont responsables. Ceci peut devenir une entrave au progrès : les changements philosophiques et technologiques peuvent amener les gens à réclamer des changements à des fonctions existantes alors que d'autres peuvent réclamer d'elles une stabilité à toute épreuve.

Certaines firmes motiveront alors l'emploi de suffixes dénotant clairement le passage à une nouvelle génération du sous-programme. Ce suffixe peut être numérique (p. ex. : la nouvelle version de `ChicFct()` devenant `ChicFct2()`) ou dénoter le caractère extensionnel du sous-programme le plus récent (p. ex. : la nouvelle version de `ChicFct()` devenant `ChicFctEx()`). Dans les cas où le nouveau sous-programme dépasse le seuil de la simple extension fonctionnelle à l'original, on cherchera alors un nouveau nom, tout simplement.

Les règles pour construire un nom de sous-programme à partir de plusieurs mots sont, en général, les mêmes que celles appliquées pour les noms de variables.

Noms significatifs de méthodes

Les règles de sous-programmes s'appliquent, dans l'ensemble, aux méthodes. Certains standards locaux existent (chez Sun puis chez Oracle, avec Java, on commencera le nom des méthodes par une minuscule, alors que chez *Microsoft*, pour la bibliothèque MFC ou dans les langages .NET, on commencera plutôt par une majuscule), qui se valent à peu près tous dans la mesure où ils sont respectés de manière rigoureuse, s'intègrent bien aux autres volets du standard choisi, et sont respectueux de la culture des gens en place comme de celle de leurs clients.

La grande particularité des méthodes, par opposition aux sous-programmes globaux, est leur caractère subjectif. En effet, un sous-programme global est une opération qui s'applique à des entités qui lui sont fournies de l'extérieur. C'est un agent externe. Ceci est aussi en grande partie vrai pour la plupart des méthodes de classe.

Une méthode d'instance, par contre, opère d'abord et avant tout sur l'instance qui en est propriétaire, l'instance active. Les `Get` et les `Set` réfèrent à des données propres à l'instance propriétaire de la méthode elle-même; cette méthode a ainsi un accès privilégié et implicite aux attributs (ou aux méthodes) en question.

Là où on nommera souvent un sous-programme global par une combinaison verbe/ mot, comme dans `AfficherDessin(d)` ou `AvancerVoiture(v)`, on simplifiera souvent le nom d'une méthode en se limitant à un verbe, l'objet de l'opération devenant un sujet, et préfixant lors de l'appel la méthode (`d.Afficher()` ou `v.Avancer()`, présumant que `d` soit un `Dessin` et `v` une `Voiture`).

Prenant le virage objet, cette simplification a l'avantage périphérique de ne pas lier le nom d'une méthode à un seul type de données mais plutôt à toute une hiérarchie d'objets¹⁸⁵. Ainsi, avec `v.Avancer()`, le nom de méthode `Avancer()` s'applique peut-être *subjectivement* à tout véhicule, qu'il s'agisse d'un `Avion`, d'une `Voiture`, d'un `Bateau`... la mécanique du polymorphisme fera en sorte de laisser le véhicule qu'est `v`, quel que soit son type réel, avancer de la manière la plus appropriée.

<p>Notez que la surcharge de sous-programmes permet aisément de procéder ainsi même pour les fonctions globales, permettant à la fonction <code>Avancer(Voiture)</code> et à la fonction <code>Avancer(Avion)</code> de coexister sans peine dans un même programme.</p>
--

¹⁸⁵ Nous le verrons clairement lorsque nous aborderons les thèmes de l'héritage et du polymorphisme.

Noms significatifs de types et de classes

Les types et les classes représentent des catégories, des familles d'entités plutôt que des entités en propre. Mis à part les types des bibliothèques standards (comme `std::string` par exemple), la plupart des noms de classes dans la plupart des langages OO débuteront par une lettre majuscule (pensez à `Rectangle` ou à `Note`, pour prendre des exemples tirés directement de ce document).

Un bon nom de type ou de classe sera apte à désigner une telle catégorie, une telle famille. En C++, et en partie en Java et dans les langages .NET, les types représentent de l'information statique, capable de guider le compilateur dans son travail. Face à plusieurs sous-programmes de même nom pouvant prendre un paramètre d'un type donné, un compilateur contemporain choisira la version pour laquelle le type respectera le mieux la signature du sous-programme (escamotant si possible les constructions de variables temporaires ou les conversions implicites de types). Les types sont donc un outil de travail précieux pour les programmeuses et les programmeurs OO.

En C++, en partie dû à l'héritage C, certains *alias* (autrefois créés par `typedef`, maintenant par `using`) se verront affublés du suffixe `_t` ou du suffixe `_type`. Si vos propres types doivent s'intégrer à des bibliothèques respectant cette convention, alors évaluez la pertinence de vous y conformer, dans le but de faciliter l'utilisation de vos types par vos collègues ou vos client(e)s.

À propos des commentaires

Les méthodologies agiles nous ont enseigné que les commentaires peinent, en général, à survivre à l'évolution du code qu'ils accompagnent. Bien qu'ils demeurent nécessaires, les commentaires devraient être ciblés et n'apparaître qu'en moment opportun.

Dans un monde idéal, chaque classe devrait avoir un but précis et évident; chaque sous-programme devrait être court et *autodocumenté*, clair à ce point que tout commentaire devrait être redondant. Entretenir à la fois les commentaires et le code mène à des divergences entre les deux, donc à une perte de pertinence des commentaires.

Un commentaire qui ment est nuisible.

Rédigez des classes pour lesquelles la raison d'être est claire et évidente. Rédigez des sous-programmes pour lesquels l'objectif est limpide. Nommez chaque entité de manière claire et sans la moindre ambiguïté. Vous n'aurez alors à peu près pas besoin de commentaires.

Certains commentaires sont inutiles. Par exemple, examinez ce qui suit :

```
bool Succes = false; // déclaration de la variable Succes
```

Dans un tel cas, le commentaire n'ajoute rien à la compréhension du code, et risque (dans le pire cas) d'induire en erreur, comme par exemple dans le cas où la variable changerait de nom sans que le commentaire ne soit mis à jour. Des cas plus extrêmes existent, évidemment¹⁸⁶.

Faire évoluer en parallèle les commentaires et le code implique une duplication des efforts, ce qui contrevient à ce que **Andrew Hunt** et **Dave Thomas** nomment le principe *DRY (Don't Repeat Yourself)* [hdPrin].

¹⁸⁶ Voir <http://thedailywtf.com/Articles/The-Road-to-Hell.aspx>

Il existe certains guides expliquant comment (ou quand), selon leurs auteurs, le code devrait être commenté. Quelques exemples :

- <http://msdn.microsoft.com/en-us/library/aa164797.aspx> de *Microsoft*, mettant l'accent sur le développement Office et Web;
- <http://particletree.com/features/successful-strategies-for-commenting-code/> qui met l'accent sur le code JavaScript;
- <http://www.icsharpcode.net/TechNotes/Commenting20020413.pdf> qui discute entre autres du coût qu'implique le choix de ne pas commenter;
- <http://www.cprogramming.com/tutorial/comments.html> qui discute à la fois de l'importance des commentaires et de l'importance de ne pas trop commenter;
- <http://www.perl.com/lpt/a/2005/07/14/bestpractices.html> qui discute des pratiques saines de développement avec Perl. L'élément 7 de cet article, en particulier, met l'accent sur les commentaires;
- <http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=290> met de l'avant que les commentaires sont, à bien des égards, plus importants que le code;
- <http://dkrukovsky.blogspot.com/2005/07/how-to-write-comments.html>, bien que bref, s'intéresse à la bonne écriture de commentaires.

Certains outils contemporains génèrent en partie la documentation des programmes à partir du code. C'est le cas de l'utilitaire Javadoc¹⁸⁷, pour lequel vous trouverez un guide sur <http://java.sun.com/j2se/javadoc/writingdoccomments/>, de même que de l'ensemble des programmes .NET¹⁸⁸.

Le blogueur *Jeff Atwood* discute de la distinction entre commenter le quoi et commenter le comment¹⁸⁹, et fait de même pour la question des commentaires abusifs ou insuffisants¹⁹⁰.

Comme pour bien d'autres choses, c'est dans l'équilibre qu'on trouve la clé.

Important : il en coûte plus cher d'entretenir un programme que de le développer. L'ensemble de vos décisions devraient être prises en connaissance de cause.

¹⁸⁷ Voir <http://java.sun.com/j2se/javadoc/>

¹⁸⁸ Voir [http://msdn.microsoft.com/fr-fr/magazine/cc302121\(en-us\).aspx](http://msdn.microsoft.com/fr-fr/magazine/cc302121(en-us).aspx)

¹⁸⁹ Voir <http://www.codinghorror.com/blog/archives/000749.html>

¹⁹⁰ Voir <http://www.codinghorror.com/blog/archives/001150.html>

Lectures complémentaires

Pour un autre point de vue, voir :

<http://www.objectmentor.com/resources/articles/naming.htm>

Pour une approche différente (par la dérision, rien de moins), voir le très amusant (bien que quelque peu déprimant d'un certain point de vue) :

<http://mindprod.com/unmain.html>

Les règles de programmation C++ chez Google sont :

<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>

Les règles de programmation Java chez Sun sont :

<http://gee.cs.oswego.edu/dl/html/javaCodingStd.html>

Appendice 03 – Dans l’objet ou hors de l’objet?

Dans la section *Accesseurs de second ordre* de ce document, il est proposé de rédiger des accesseurs comme `Rectangle::aire()` ou `Rectangle::perimetre()` même si ceux-ci reposent essentiellement sur des calculs faits à partir d’appels à d’autres accesseurs, plus primitifs (que nous qualifions d’accesseurs de premier ordre dans le texte).

En fait, le texte indique ceci :

« La raison est que l’objet lui-même tend à être le meilleur pour implanter correctement les opérations qui le concernent. L’objet seul sait comment il est constitué, et comment le calcul du périmètre pourrait être le plus efficace étant donné cette constitution. »

L’idée que promulgue ce passage est que parfois, il est clair qu’un objet pourra mieux faire certaines choses que ne le pourrait son code client. Ceci peut tenir à une meilleure connaissance de son organisation interne, par exemple.

Comme nous le verrons dans [POOv01], puisqu’il s’agit d’une thématique reposant que l’héritage, sur le **polymorphisme** et sur l’**abstraction**, il arrive aussi qu’on souhaite imposer à une classe d’offrir un service du fait que cela garantit qu’elle, mais aussi toute sa descendance, devra l’offrir. Par exemple, s’il est décidé que toute `Forme` se dessinera, et s’il est décidé qu’un `Rectangle` est une forme, tout comme un `Triangle`, alors il est possible d’imposer, à travers `Forme`, à tous les cas particulier de `Forme` d’exposer un service `dessiner()`. Ce faisant, il est possible d’écrire des programmes manipulant des entités qui soient *au moins* des cas particuliers de la classe `Forme` et de leur demander de se `dessiner()` en toute quiétude, le service étant nécessairement offert et implémenté dans chaque cas.

Cependant, d’autres enjeux, plus subtils, entrent en ligne de compte, en particulier si nous tenons compte du principe décrit dans la section *Le principe ouvert/ fermé (Open/ Closed Principle)* du présent document.

En effet, nous savons qu’une classe est généralement fermée pour modification. Cela signifie qu’une fois l’accolade fermante de sa déclaration apposée, il n’est plus possible de lui ajouter des services ou des états. Conséquemment, dans une optique où l’on souhaiterait qu’un objet offre tous les services possibles *sur lui*, nous faisons face à un dilemme : devons-nous être exhaustifs *a priori*, donc dès le design de la classe? Est-ce possible d’y arriver?

Est-ce même souhaitable? Après tout, qui peut prédire quels seront les services de cet objet dont aura vraiment besoin le code client? Est-ce sain, économiquement, d’insérer des centaines de méthodes dans une classe juste au cas où celles-ci, un jour, seraient utiles?

Qu'en est-il des services qui manipulent à la fois nos classes et d'autres classes? Supposons l'opérateur `==` sur une classe `entier`. Nous souhaitons que le code suivant soit correct, bien sûr, mais examinez bien la notation :

```
// ...
entier e(3);
int i;
cin >> i;
if (e == i) { // entier::operator==(int) const
    // ...
}
if (i == e) { // pas une méthode d'entier, clairement
    // ...
}
// ...
```

Il semble illégitime de prétendre que, dans une optique d'équivalence opérationnelle des types, la première alternative (`if (e==i)`) soit correcte mais que la seconde (`if (i==e)`) ne le soit pas. Ainsi, essayer d'offrir tous les services possibles à même une classe sous forme de méthodes n'est pas raisonnable en pratique.

Ce que la pratique, justement, nous apprend, c'est qu'il est sage de distinguer, dans une classe donnée, les services qu'on pourrait qualifier d'essentiels des services plus secondaires. Les services plus essentiels seraient ceux qui, par exemple, touchent directement aux attributs, à la structure interne de l'objet. Ceux-ci requièrent des privilèges d'accès particuliers et sont de bons candidats à devenir des méthodes.

D'autres services, quant à eux, peuvent être construits à partir de services essentiels, et ce sans perte de performance ou de généralité. Pensons par exemple à la distance entre deux points :

Sous forme de méthode	Sous forme de fonction
<pre>class Point { int x_, y_; public: Point(int x, int y); int x() const; int y() const; float distance_de(const Point&) const; }; int main() { Point p0{ 0,0 }, p1{ 1,1 }; auto dist = p0.distance_de(p1); // 2^(1/2) }</pre>	<pre>class Point { int x_, y_; public: Point(int x, int y); int x() const; int y() const; }; float distance(const Point&, const Point&); int main() { Point p0{ 0,0 }, p1{ 1,1 }; auto dist = distance(p0,p1); // 2^(1/2) }</pre>

L'écriture de gauche est-elle vraiment plus élégante que l'écriture de droite? On peut en douter, dans un cas comme celui-ci. D'ailleurs un indice que le service de calcul de la distance entre deux points n'a pas vraiment à être implémenté sous forme de méthode est qu'écrire `p0.distance_de(p1)` ou écrire `p1.distance_de(p0)` ne change pas grand-chose, pas plus d'ailleurs qu'écrire `distance(p0,p1)` ou `distance(p1,p0)`.

Les deux implémentations se ressembleront d'ailleurs sans doute beaucoup :

Sous forme de méthode	Sous forme de fonction
<pre>#include <cmath> float Point::distance_de(const Point &p) const { using namespace std; return sqrt(pow(x()-p.x(), 2) + pow(y()-p.y(), 2)); }</pre>	<pre>#include <cmath> float distance(const Point &p0, const Point &p1) { using namespace std; return sqrt(pow(p0.x()-p1.x(), 2) + pow(p0.y()-p1.y(), 2)); }</pre>

Lorsqu'il n'y a pas de gains de performance à faire, et lorsqu'il est possible d'offrir des services en périphérie d'un objet, il est souvent préférable de les offrir ainsi. Le résultat de cette pratique est une gamme d'objets plus légers, plus simples, auxquels se greffent un ensemble *extensible* d'outils et de services.

Dans le cas des services `aire()` et `perimetre()`, le choix de les implémenter sous forme de méthodes est :

- un peu esthétique (ces services semblent être des services de base);
- un peu pédagogique (cela montre qu'il est possible de faire des méthodes qui dépendent d'autres méthodes et ce, sans perte de performance); et
- un peu conceptuel (il serait plus subtil, mais pas impossible, d'établir un algorithme général de calcul de l'aire et du périmètre d'une forme 2D au sens large, mais il y aurait probablement un coût en termes de performance à l'implémentation générale en comparaison avec celle, simple et spécifique, du `Rectangle` de nos notes de cours).

Annexe 04 – Sémantique des constructeurs

Dans une entrevue [StepInf], **Alexander Stepanov** décrit ainsi sa conception de ce que signifie un constructeur par défaut :

« The role of a default constructor is that it constructs the object so it can be assigned to or destroyed. Nothing more. [My book] calls these “partially formed” objects. Default construction does not guarantee that the initial value is meaningful, or even the same across invocations. Of course, if you write your own default constructor for your own class, you may choose to initialize it with a useful value, but this is not part of the requirement »

L’acceptation de Stepanov est algorithmique, pas OO; on parle ici de l’un des plus grands experts qui soient, du moins pour ce qui a trait à la programmation générique [POOv02]. Laisser un objet « partiellement formé », à moins que ce ne soit dans un contexte contrôlé bien sûr, est (d’un point de vue OO rigoriste) un bris d’encapsulation : l’objet existe, mais est invalide et ne peut être utilisé. Ses invariants ne sont pas garantis... *Ce n’est pas véritablement un objet*, du moins pas au sens où nous l’entendons.

Cela dit, quel est le sens à donner à un constructeur, dans une optique de POO?

Le constructeur par défaut est utile quand il existe une chose telle qu’un objet par défaut. Cela s’avère fréquemment dans le cas d’entités mathématiques (qui sont probablement celles qui sont les plus près des préoccupations de gens comme Stepanov) : un `Point` par défaut peut représenter l’origine d’un système d’axes, par exemple. De même, un `Cercle` par défaut peut être un cercle unitaire (centre à l’origine, rayon unitaire).

Pour un concept plus complexe, par exemple une `Personne`, il peut ne pas être raisonnable d’offrir un constructeur par défaut. Que représente une `Personne` par défaut, en effet? Quel est son nom? Son numéro d’assurance sociale? Son sexe?

Notez qu’il se peut qu’un objet représentant un agrégat d’états (d’attributs) ayant chacun une valeur par défaut lorsque pris sur une base individuelle n’ait pas d’état par défaut lorsque pris sur une base composite. Un objet n’est pas que la somme de ses parties; il existe parfois des dépendances (des invariants) entre les éléments constitutifs d’un objet; il faut donc réfléchir la question de la pertinence, ou non, d’un constructeur par défaut en fonction de la classe et de ce qu’elle représente.

Un constructeur paramétrique représente le mécanisme par lequel le code client peut paramétrer l’état initial d’un objet. C’est ce qui permet d’ouvrir un fichier en donnant son nom au constructeur d’un objet, de même que certains paramètres guidant le fonctionnement de l’objet par la suite si cela s’avère pertinent (lecture binaire ou texte, par exemple). C’est aussi ce qui permet de créer un `Point` représentant un lieu précis, autre que l’origine.

On offrira des constructeurs paramétriques couvrant les cas pertinents à l’initialisation d’un objet. Pas plus, pas moins. Et on cherchera à n’offrir que les paramètres qui semblent raisonnables, dans un ordre cohérent (p. ex. : si un `Rectangle` peut être initialisé avec une hauteur, une largeur et, optionnellement, une couleur, on fera en sorte que l’ordre dans lequel la hauteur et la largeur sont passés soit le même chaque fois).

Le constructeur de copie est le lieu de duplication des états d'un objet. Mieux vaut le bloquer si la copie n'est pas raisonnable. Contrairement au constructeur par défaut, il est souvent raisonnable de ne pas implémenter explicitement le constructeur de copie, car le compilateur l'offrira alors pour nous : le comportement par défaut du constructeur de copie est de copier les attributs de l'objet un à un, à l'aide de leurs propres constructeurs de copie; conséquemment, si l'objet était cohérent *a priori*, ce que nous présumons en vertu de l'encapsulation, alors la copie implicite devrait aussi être cohérente.

On rédigera explicitement le constructeur de copie quand l'objet a des responsabilités qui échappent au compilateur, ou des « invariants silencieux » qui seraient brisés par une copie trop naïve. Par exemple, si l'objet original alloue dynamiquement un tableau (avec `new[]`) et conserve dans un pointeur brut l'adresse ainsi obtenue, alors la copie implicite copiera le pointeur, pas le pointé. Si l'objet est responsable de la mémoire ainsi allouée, alors l'original cherchera à libérer le tableau... et la copie en fera de même. Ce bris de responsabilité est un signe que le constructeur de copie implicite est, ici, trop naïf.

Même sans un tel risque de bris de responsabilité, il est possible qu'un objet soit brisé lors d'une copie implicite. Imaginons une classe comme la suivante :

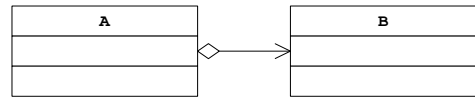
```
class TableauPrioritaire {
public:
    static const int TAILLE = 10;
private:
    int valeurs[TAILLE];
    int *prochain;
    // ...
};
```

Supposons ici que, dans un `TableauPrioritaire`, l'attribut `prochain` soit l'adresse du prochain élément à utiliser dans `valeurs`. Cet invariant est silencieux, au sens où le compilateur n'en connaît pas la nature.

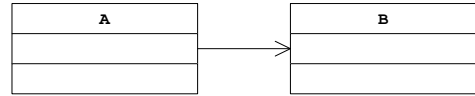
Ici, copier implicitement un `TableauPrioritaire` ferait en sorte que, suite à la copie, l'attribut `prochain` de la copie pointe dans l'attribut `valeurs...` de l'original. Ceci est probablement une erreur; si la copie doit être permise, il faudra sans doute la prendre en charge de manière explicite.

Annexe 00 – Résumé de la notation UML abordée dans ce document

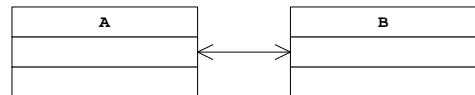
Agrégation. Dans le schéma à droite, un A est un agrégat comprenant un B, ce qui signifie que le B fait partie du A mais aurait pu commencer à exister avant A et pourrait continuer d'exister suite à la mort de A.



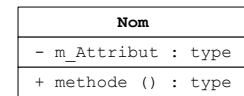
Association unidirectionnelle. Dans le schéma à droite, un A peut contacter un B, mais l'inverse n'est pas vrai.



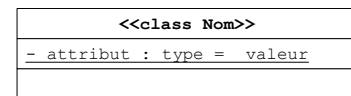
Association bidirectionnelle. Dans le schéma à droite, un A peut contacter un B, et un B peut contacter un A.



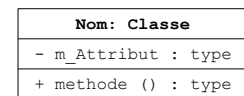
Classe. Dans le schéma à droite, le nom de la classe est Nom. Elle possède un attribut d'instance privé nommé m_Attribut de type type, et une méthode publique nommée methode() de type type.



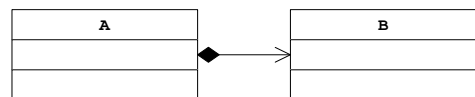
Classe (membre de). Dans le schéma à droite, la classe Nom possède un attribut de classe nommé attribut de type type et initialisé à la valeur valeur.



Instance. Dans le schéma à droite, Nom est une instance de la classe Classe et possède un attribut d'instance privé nommé m_Attribut de type type, et une méthode publique nommée methode() de type type.



Composition. Dans le schéma à droite, un A est un composé comprenant un B, ce qui signifie que B fait partie de A et que la vie de B est délimitée de manière stricte par celle de A.



Annexe 01 – Discussions sur quelques réflexions choisies

Cette annexe présente quelques discussions portant sur des questions de réflexion parsemées ici et là dans le document. Ces questions ont été choisies (et posées) délibérément parce que les réponses ne sont pas aussi évidentes et banales qu'il n'y paraît à première vue.

Réflexion 00.0 : tout est-il objet?

Une question de réflexion levée en début de document, et répétée à droite, pose en sourdine la question du découpage. Jusqu'où doit-on aller dans la décomposition d'un problème en objets?

Dans la classe `Client`, y aurait-il lieu d'utiliser une classe distincte pour représenter une adresse? Un numéro de téléphone? Expliquez votre position (la réponse est subtile).

Pour répondre convenablement, il convient tout d'abord de se remémorer la forme de la classe `Client` au moment où la question a été posée (voir à droite). L'adresse et le numéro de téléphone proposés étaient du texte, et la question à laquelle vous avez été convié(e)s était de réfléchir à l'utilisation de classes pour remplacer ces types plutôt simples.

Client	
- Nom:	texte
- Prénom:	texte
- Adresse:	texte
- NoTél:	texte
- SoldeImpayé:	réel

La littérature vous engagera dans une optique de découpage très fin. Par exemple, on vous conviera à représenter un numéro de téléphone sous forme d'une classe ayant ses propres règles de validation, et donc de produire des objets pleinement encapsulés et valides une fois construits. C'est d'ailleurs une excellente idée dans la mesure où les règles de validité sont claires :

- peut-on déterminer ce qu'est un numéro de téléphone valide? De prime abord, on aurait le réflexe de dire oui, et d'utiliser un indicatif régional entier (qu'on pourrait valider en lien avec une base de données) et de sept chiffres, mais il y a aussi des services particuliers comme le service 911 dont il faut tenir compte;
- existe-t-il des opérations propres à tous les numéros de téléphone? Peut-être (en particulier la saisie et l'affichage). Cependant, les règles propres aux numéros de téléphone sont fortement liées à des considérations de localisation, variant selon les pays, et de contexte (les cellulaires, les « services étoiles », les interurbains, etc.);
- ces considérations tiennent aussi pour les adresses postales. Les américains ont un ZIP code, les canadiens ont un code postal. Les règles pour l'un et pour l'autre diffèrent.

Les penseurs ○○ plus avancés, ou qui auront lus le prochain volume de cette série de notes de cours, envisageront peut-être une hiérarchie de classes dans un cas comme dans l'autre : le concept général de numéro de téléphone, par exemple, puis des déclinaisons locales selon les pays et les catégories de services téléphoniques. Cette avenue n'est pas sans intérêt, mais soulève des questions plus profondes qu'il n'y paraît : quels sont les services communs à tous les numéros de téléphone? Que signifie opérer sur un numéro de téléphone au sens large?

En pratique, si votre logiciel s'adresse à une clientèle locale, il est raisonnable et utile de définir des classes pour représenter des entités comme des numéros de téléphone et des adresses, du fait que les règles de validité seront stables. Si vous envisagez internationaliser votre produit, vous ferez face à une explosion de possibilités face auxquelles vous n'aurez que peu ou pas de contrôle. La conception de ces objets se transformera rapidement en cauchemar.

Les gens qui développent des systèmes à très grande échelle, par exemple des systèmes reposant sur des architectures orientées services (AOS, ou SOA en anglais), utiliseront des types de données très simples (texte, entiers, réels, tableaux) pour leurs échanges avec des tiers, et utiliseront des classes spécialisées à l'interne lorsqu'ils travailleront sur des entités moins variables. Ce faisant, les relations avec les autres entreprises (le *Business to Business*, ou B2B) ne seront pas affectées de manière aussi adverse si les règles structurelles d'une entité comme un numéro de téléphone doivent être adaptées à une nouvelle réalité, et le volet utilitaire des classes assurant, à l'interne, la validité des entités manipulées et facilitant à la fois les tests et les diagnostics sera conservée.

Un des pièges de l'approche OO est de vouloir l'appliquer partout et à outrance. Il n'y a pas d'outil qui soit une substitution pour l'intelligence, même un outil aussi puissant et polyvalent que l'approche OO.

Réflexion 00.1 : Machiavel ou Murphy?

Considérant l'importance du principe d'encapsulation, et de la capacité qu'ont les objets d'assurer leur propre intégrité, la question de savoir où s'arrêter se pose naturellement.

Jusqu'où devrait-on pousser la quête de la protection de l'intégrité des objets?

La question se pose dans un cadre plus large de sécurisation des programmes et des environnements informatiques : où tracer la ligne entre la sécurité et la convivialité?

Un adage fréquemment rencontré pour décrire les stratégies de validation dans un programme est celui-ci : *il faut choisir entre se protéger contre Machiavel et se protéger contre Murphy*. Le sens de cet énoncé est qu'il faut faire un choix entre se protéger contre les accidents et les imprudences et se protéger contre des tiers hostiles.

Le compilateur et la qualification de constance sont des plus utiles pour ce qui est de protéger contre Murphy. Dans la mesure où de saines pratiques de programmation sont mises en place et dans la mesure où le respect de l'encapsulation est strict, les risques de malchances sont nettement réduits.

D'autres pratiques aident à réduire les risques d'incursion de Murphy dans un système informatique (en particulier, ne pas tolérer qu'il reste des avertissements à la compilation d'un programme au moment de le livrer).

Se protéger contre Machiavel est un métier en soi pour des spécialistes très qualifiés et à l'esprit pervers. C'est beaucoup plus difficile. Une conceptrice ou un concepteur OO devra décider quels services exposer et lesquels cacher, et souvent, la réponse à cette question aura des ramifications à grande échelle.

La plupart des pratiques proposées dans ce document se veulent de saines pratiques de programmation et ont pour objectif d'accroître la lisibilité des programmes, de les rendre faciles à entretenir et de réduire les risques d'accident au minimum. Le compilateur est en général plus utile pour repousser Murphy que pour repousser Machiavel. Si un programme n'arrive pas à éviter Murphy, alors Machiavel est le dernier de ses soucis...

Une exception à cette règle générale se trouve à la frontière des systèmes : il faut envisager Machiavel dans les entrées/ sorties (usager malicieux, données corrompues, homologue pirate) et dans les logiciels permettant à d'autres de développer des ajouts à un système (pensez aux greffons de Firefox, par exemple). Sur ces frontières, Machiavel est très présent.

Scott Meyers résume l'idée comme suit : *il faut rendre simple la réalisation des opérations jugées saines, et rendre ardues les opérations jugées dangereuses.*

À titre d'exemple, dans un langage supportant la sémantique d'accès direct aux objets (voir *Appendice 00 – Sémantiques directes et indirectes*), un objet représentant un conteneur d'autres objets voudra donner accès aux éléments qu'il contient, mais pourra le faire en utilisant une métaphore de copie plutôt qu'une métaphore indirecte. Ceci réduira les risques que deux de ses clients ne modifient par inadvertance le même objet en ayant tous deux obtenus une indirection vers la même entité. Cependant, rien n'empêchera un code client d'y entreposer des indirections ou des objets cachant des indirections, peu importe ses raisons (car il y en a, et beaucoup), et un client désireux d'agir ainsi est responsable de ses actes.

Dans un système OO, il est possible de faire la vie dure à Machiavel, mais il est difficile de l'empêcher de procéder (entre autres parce qu'il a, avec les conversions explicites de types, le droit de mentir). Par contre, il est possible, en travaillant sainement (viser des objets aux états stables, idéalement constants; réduire la surface publique des objets et valider les données envers lesquelles on ne peut avoir confiance; offrir des services qui rendent agréables les opérations correctes sur l'objet; *etc.*) d'évacuer Murphy à toutes fins pratiques.

Réflexion 00.2 : quelles méthodes exposer?

Un objet peut exposer une gamme arbitrairement grande de services, des plus simples aux plus riches. Comment déterminer si la surface publique d'un objet est suffisante, si elle est bien définie?

Comment décider quels accesseurs et quels mutateurs (ou, de manière plus générale, quelles méthodes) une classe devrait exposer?

Aucune loi n'existe pour obliger une classe à exposer une paire accesseur/ mutateur pour chaque attribut. Bien que cette stratégie soit utilisée dans ce document d'introduction pour décrire l'encapsulation dans sa forme la plus primitive, et bien que plusieurs livres utilisent cette stratégie sur une base systématique, il arrive fréquemment en pratique que l'on ait recours à des stratégies plus fines.

Ainsi, un mutateur public n'a de sens que quand il est raisonnable de laisser les sous-programmes utilisant un objet en manipuler le contenu (même si l'objet, propriétaire de ses mutateurs, a le droit de refuser la demande de manipulation).

Si un objet ne souhaite utiliser des mutateurs que dans le but de contrôler les modifications qu'il souhaite apporter à son propre état, alors des mutateurs privés sont nettement plus indiqués.

Dans des langages qui ne supportent pas le concept d'objet constant, comme Java et les langages .NET, définir une classe sans mutateur public est même souvent une stratégie recommandable, au point de porter le nom de classe immuable et d'être érigé en schéma de conception (un idiome, en pratique, puisque les langages supportant les objets constants n'en ont pas besoin). Les langages fonctionnels et leurs *aficionados* ont en horreur les objets qui changent d'états pendant leur existence. Voir la section *Copies et classes d'objets immuables*, plus haut.

Ensuite, il faut retenir qu'un objet, en vertu du principe d'encapsulation, est plus utile pour ce qu'il sait faire que pour ce dont il est constitué. Ainsi, plutôt que d'exposer systématiquement une gamme d'accesseurs publics de premier ordre (donc un pour chaque attribut de l'objet) et de second ordre (un pour chaque opération de consultation composée et susceptible d'être exprimée à partir d'opérations de consultation plus primitives), il est préférable de s'interroger à savoir *quelles sont les opérations les plus naturelles sur cet objet*, sans égard aux attributs dont il est constitué, et d'exposer ces opérations sous forme de méthodes puisque c'est d'elles qu'auront besoin les utilisateurs de l'objet.

À titre d'exemple, la classe `std::list` de la bibliothèque standard de C++ [POOv02] représente une liste arbitrairement grande d'objets du même type, avec des facilités très efficaces d'insertion et de suppression d'éléments. Il se trouve que cette classe supporte la méthode `size()` qui en retourne le nombre d'éléments, mais que la majorité des implémentations commerciales de cette classe ne tiennent à jour aucun attribut dont le contenu est le nombre d'éléments de la liste. Ceci implique qu'appeler la méthode `size()` d'une instance de `std::list` est une opération relativement lente, la liste devant compter ses éléments un à un (une opération $O(n)$ plutôt que $O(1)$, pour celles et ceux qui savent un peu ce que cela signifie).

La raison derrière ce choix est que les concepteurs de la bibliothèque standard ont fait le constat qu'il est très rare qu'on ait besoin de connaître la taille d'une liste quand on choisit de manipuler plusieurs objets à l'aide de ce type de conteneur (d'autres conteneurs existent, comme le tableau par exemple, pour lesquels la taille est une donnée plus importante). Sachant que la taille n'interviendrait pas dans la majorité des opérations sur une `std::list`, ses concepteurs ont donc choisi d'économiser l'espace requis pour un tel attribut et d'offrir la méthode `size()` par souci de convivialité et d'uniformité avec d'autres conteneurs standards (sur lesquels nous reviendrons). Notez que la bibliothèque standard est documentée de manière à indiquer la complexité de chaque opération; la méthode `size()` de `std::list` ne garantit pas une complexité constante, ce qui invite à la prudence.

C'est là un choix de design qui ne correspond pas tant à une réalité structurelle (il n'y a pas d'attribut comme `size_` dans la majorité des implémentations de la classe standard `std::list`) qu'à un choix esthétique (tous les conteneurs standards offrent une méthode `size()`, alors offrons-la pour `std::list` aussi tout en recommandant aux gens de ne l'utiliser qu'en connaissance de cause).

Tel que mentionné en réponse à *Réflexion 00.1 : Machiavel ou Murphy?*, plus haut, **Scott Meyers** propose dans [EffCpp,§4] un adage clair et simple pour la conception d'interfaces, ce mot étant pris au sens large d'*ensemble d'opérations possibles sur un objet*, pas au sens restreint que nous examinerons dans la section sur l'héritage d'interfaces de [POOv01].

Selon lui, une bonne interface devrait montrer les deux qualités suivantes :

- être facile à utiliser correctement; et
- être difficile à utiliser incorrectement.

Cette écriture a la qualité de la simplicité et de l'efficacité. En effet, il serait sans doute contreproductif d'exiger d'une interface que la seule manière de l'utiliser soit celle qui ait été envisagée par les concepteurs d'une classe donnée. Ce serait dogmatique et cela brimerait l'imagination des gens; qui sait à quoi penseront les utilisateurs éventuels d'une classe donnée?

Meyers met entre autres de l'avant qu'une interface obligeant le code client d'un objet à retenir qu'il devra faire autre chose ultérieurement, par exemple une méthode ayant l'effet secondaire de créer un objet (avec `new`) et de le retourner de manière à ce que le code client doive le détruire ultérieurement, est une mauvaise interface.

Un objet responsable cherche à réduire son impact sur l'économie d'ensemble d'un programme, et un objet qui responsabilise trop son client sur la gestion de ses propres ressources mérite qu'on réfléchisse un peu plus longtemps sur son interface, par exemple en offrant des outils `RAII` (voir *Annexe 03 – Concepts et pratiques du langage C++*).

En retour, certains usages sont foncièrement incorrects, parce qu'ils donnent des résultats qu'on peut reconnaître *a priori* comme étant inefficaces (une opération possible mais qui ne peut être que lente) ou malsains (susceptibles de provoquer un bris d'encapsulation). Plutôt que de rendre impossibles les opérations peu recommandables (quoique, dans le cas d'opérations provoquant des bris d'encapsulation, rendre celles-ci impossibles est souvent la chose à faire), l'idée est de les rendre moins tentantes, ou d'offrir des alternatives préférables.

Le cas de la classe `std::list`, plus haut, est un exemple patent de cette approche : la méthode `size()` est susceptible d'y être lente mais la méthode `empty()`, retournant vrai seulement si la liste est vide, offre une garantie de performance plus radicale (temps constant, $O(1)$, ce qui est optimal). Considérant que la principale raison pour s'enquérir de la taille (exprimée en nombre d'éléments) d'une liste est, normalement, de savoir si elle est vide, la classe offre une alternative optimale en ce sens.

En même temps, la classe `std::vector` offre une méthode pour ajouter un élément à la fin (`push_back()`), qui est très efficace, mais l'ajout d'un élément au début est une opération très lente, pour laquelle il n'existe pas de raccourci (on peut le faire, mais en l'écrivant explicitement; ceci correspond à la recommandation de Meyers ci-dessus).

Le reste, soit choisir la bonne opération parmi une gamme d'opérations disponibles sur un objet, est histoire de documentation, de pédagogie et d'expérience.

Réflexion 00.3 : des correctifs silencieux?

La question de réflexion soulevée à droite réfère à un mutateur qui protège l'intégrité d'un objet (une instance de `Rectangle`) en empêchant les données invalides d'en polluer les états.

Le mutateur `SetHauteur()` de `Rectangle`, à droite, protège l'instance de la corruption, mais serait-il sage d'avoir recours à cette écriture en pratique?

Concrètement, l'algorithme de ce mutateur est *si la donnée est jugée valide, alors procéder à la modification demandée de l'état de l'attribut visé*. Ceci protège effectivement l'intégrité de l'objet en empêchant la corruption de sa propre intégrité.

En pratique, cette stratégie n'est qu'un début, et ne suffit pas sur le plan systémique. Du point de vue du code client, en effet, l'objet (l'instance de `Rectangle`) semble avoir accepté la demande de modification d'état puisqu'il ne donne aucun signe apparent à l'effet contraire. Le code client est donc peut-être erroné et n'a aucun moyen de diagnostiquer cette situation.

Protéger l'intégrité de l'objet est essentiel à la réalisation d'une encapsulation correcte. Cependant, un objet fait partie du système qui y a recours, et doit collaborer à la bonne marche de ce système. Tout comme un destructeur bien écrit doit s'assurer que l'objet laisse le système en aussi bon état que lorsque l'objet a été construit (dans la mesure de ses moyens), une méthode constatant un problème à l'exécution devrait à la fois protéger l'intégrité de l'objet et s'assurer de relever ce problème pour que des entités capables de le traiter puissent elles aussi constater le problème et, si possible, l'adresser.

La voie royale pour signaler une tentative de corruption ou un comportement anormal est, le nom le dit, le *traitement d'exceptions* (voir la section à cet effet, dans le présent document). Ne sachant pas dans quel contexte il est utilisé, l'objet ne peut habituellement pas traiter lui-même le problème qu'il a détecté. Il peut assurer sa propre intégrité, en vertu de sa responsabilité et des pouvoirs que son respect strict de l'encapsulation lui a conférés, et il peut signaler le caractère exceptionnel de la situation constatée à qui de droit.

Ainsi, dans les deux extraits à droite, celui de gauche protège silencieusement l'instance de `Rectangle`, mais celui de droite, qui signale le problème sur détection en plus de protéger l'objet, est *nettement* préférable.

```
void Rectangle::SetHauteur
(int hauteur) {
    if (hauteur > 0)
        hauteur_ = hauteur;
}
```

```
class HauteurIllegale {};
// ...
void Rectangle::SetHauteur
(int hauteur) {
    if (hauteur <= 0)
        throw HauteurIllegale{};
    hauteur_ = hauteur;
}
```

Il est important de considérer à la fois les objets par et pour eux-mêmes, responsables et actifs, et comme de bons citoyens du système qu'ils habitent.

Réflexion 00.4 : choix de représentation

Cette question de réflexion portait sur les choix de représentation et d'implémentation internes aux objets : que devrait-on calculer sur demande? Que devrait-on garder à jour en tout temps? Comment en arriver à un équilibre judicieux entre les coûts en temps et en espace

Est-il avantageux de garder à jour l'aire et le périmètre d'un `Rectangle` à l'aide d'attributs? Quel serait le prix à payer? Quel est le choix optimal? Expliquez votre position

Rappelons que le choix des types et structures de données servant d'attributs pour un objet sont des choix d'implantation, et relèvent de l'implémentation. On vise à forcer, par l'encapsulation, les intervenants externes à passer par des méthodes, opérations normalisées et sécurisées, pour interagir avec les détails d'implémentation. Ceci accroît le niveau d'abstraction requis du code client, rend le code plus général, réduit le couplage, facilite l'entretien des programmes... L'encapsulation est vraiment un principe précieux.

Trouver l'équilibre entre *ce qui doit être calculé* et *ce qui doit être tenu à jour en tant que donnée connue* dans un objet dépend du contexte d'utilisation. Pour prendre l'exemple simple de la classe `Rectangle`, décider si on devrait garder en note l'aire du rectangle ou s'il serait préférable de la calculer sur demande est une question qui en cache une autre : est-il plus efficace de calculer l'aire du rectangle à chaque changement de taille, ou à chaque fois qu'on voudra connaître l'aire du rectangle?

L'enjeu ici en est un de vitesse, d'efficacité. Si on choisit de calculer l'aire sur demande, alors chaque demande d'aire signifiera un calcul (somme de deux produits); si on choisit plutôt de calculer l'aire à chaque fois qu'elle change, pour l'avoir à jour en tout temps, alors chaque demande d'aire entraînera une réponse immédiate, mais chaque changement à la largeur ou à la hauteur du `Rectangle` impliquera un calcul d'aire.

Le problème du calcul de l'aire du `Rectangle` est un problème banal, que nous utilisons ici à cause de sa simplicité. Pour vraiment saisir en quoi et comment ce dont nous discutons ici pourrait être utile, remplacez mentalement la question de savoir si on devrait tenir à jour ou non l'aire d'un `Rectangle` par celle-ci : un programme communique avec une base de données et veut garder en mémoire les informations les plus susceptibles d'être sollicitées, mais ne veut pas garder toutes les données en mémoire, alors on veut savoir quelles sont les données qui risquent le plus d'être demandées à tout moment, dans le but d'être proactif dans les démarches d'interaction avec la base de données...

Réflexion 00.5 : clients hostiles

Le constructeur paramétrique prend les valeurs de ses paramètres du code client. En vertu du principe d'encapsulation, l'objet en cours de construction doit garantir son intégrité du début à la fin de son existence. En vertu du même principe, l'objet est le lieu où se gèrent ses propres politiques de validité.

Remarquez que notre constructeur paramétrique initialise ses attributs en passant par des mutateurs plutôt qu'en affectant directement aux attributs correspondants les valeurs reçues en paramètre. Pourquoi donc procédons-nous ainsi?

L'objet en cours de construction ne présume pas que le code client connaît la plage des valeurs acceptables à ses yeux. Il ne sait pas non plus si le client lui est hostile. Ce faisant, toute valeur reçue en paramètre doit être validée avant d'être affectée.

Ce qui est véritablement important pour un objet est de faire en sorte que son état initial soit connu et valide. Il faut éviter à tout prix une situation comme celle proposée à droite, qui ferait en sorte que l'instance `x` de classe `X` tout juste construite ait un état indéfini (tracez le code, vous verrez).

```
class X {
    int val_;
    void SetVal(int val) {
        if (val > 0)
            val_ = val;
    }
public :
    X(int val) {
        SetVal(val);
    }
};

int main() {
    X x{-3}; // oups!
}
```

Une approche évitant de tels ennuis consiste à initialiser systématiquement les attributs avec des valeurs par défaut convenables.

Une autre est de faire tel que recommandé dans la section **Réflexion 00.3 : des correctifs silencieux?** plus haut et d'éviter les correctifs silencieux. Ici, sur réception d'une valeur négative, la méthode `X::SetVal(const int)` lèverait une exception, l'instance `x` ne serait jamais construite, et le code client prendrait conscience du caractère erroné de sa tentative d'initialisation.

Peu importe l'approche choisie, un objet construit doit être valide; un objet invalide ne doit pas compléter sa construction.

Réflexion 00.6 : une saine gamme de constructeurs

Tout programme et toute bibliothèque devrait faire en sorte que ses objets soient systématiquement créés dans une forme **directement utilisable** et **sémantiquement viable**.

À quels signes reconnaît-on que les constructeurs exposés par un objet sont en nombre suffisant, sans plus?

Par exemple, si une position doit avoir un nombre de dimensions connu pour être utilisable, alors il devrait être impossible de construire une position sans spécifier, d'une manière ou d'une autre, ce nombre de dimensions, à moins bien sûr que l'idée d'un nombre de dimensions typique, par défaut, n'ait du sens.

Les stratégies de construction exposées par une classe devraient permettre en tout temps de définir un objet correct dans l'immédiat. Typiquement, cela implique une gamme complète (mais sans éléments superflus) de constructeurs ou une application du schéma de conception *Fabrique* (que nous aborderons dans d'autres volumes), par laquelle un sous-programme ou un objet spécialisé en fabrique d'autres en combinant construction et initialisation *a posteriori*. Un programme doit éviter de laisser circuler des objets inutilisables ou incohérents, puisque cela force le code client à prendre sur ses épaules une gestion qui ne devrait pas lui revenir.

Ceci implique quelques consignes :

- n'avoir recours à un constructeur par défaut que dans le cas où exprimer une instance par défaut a un sens complet et exprimable. Une coordonnée 3D par défaut dont les valeurs de x , y et z valent 0, 0.0 ou 0.0f dans un univers Euclidien est raisonnable, mais une coordonnée 3D par défaut sans système de référence (au moins implicite) n'est pas raisonnable. Une coordonnée à n dimensions n'est admissible que si n est connu à la construction, sinon toutes les méthodes de l'objet devront valider systématiquement si le nombre de dimensions est connu avant de procéder, ce qui introduira énormément de validation, de traitement d'erreurs et de risques d'oubli humain;
- si le problème tient de la sérialisation, envisager des fabriques accompagnant les objets à reconstituer et capables de désérialiser ses divers composants pour le reconstruire en sollicitant le constructeur paramétrique approprié (ces techniques sont abordées dans [POOv02] et [POOv03]). Une stratégie exprimée sous la forme d'une construction par défaut suivie d'une série d'invocations de divers mutateurs a le défaut d'introduire énormément de problèmes et de n'en régler que très peu;
- il est sage d'exposer un constructeur paramétrique pour chaque combinaison de paramètres à partir de laquelle il est acceptable de penser instancier un objet valide. Souvenez-vous toutefois que les constructeurs paramétriques publics reçoivent leurs intrants de sources potentiellement hostiles (ou à tout le moins imprudentes) et doivent donc prendre soin de valider systématiquement chacun de ces intrants pour des raisons de références ou de pointeurs nuls, de respect des bornes acceptables ou de toute autre forme de corruption, volontaire ou non.

Trop souvent, les constructeurs ne font pas leur travail convenablement. Le rôle d'un constructeur s'inscrit dans le principe d'encapsulation, énoncé comme suit : **tout objet est responsable de sa propre intégrité, du début à la fin de sa vie.**

Être responsable de son intégrité implique être en mesure d'offrir des garanties. Pour ce faire, il est important que l'état de tout objet soit connu de manière pleine et entière suite à la fin de l'exécution de son constructeur.

Cela implique initialiser explicitement tous les attributs de l'objet (au moins à `null` pour une référence en Java ou C#, à `Nothing` pour une référence en VB.NET et à `0` pour un pointeur en C++) dans chacun de ses constructeurs, à moins que les attributs en question aient eux-mêmes un constructeur par défaut valide et dont le fruit correspond aux attentes de l'objet qui les possède.

Une fois les états possible de l'objet connus, définis et énumérés pour chaque attribut, il devient possible d'écrire des méthodes d'instance sécurisées : si un objet risque de laisser certaines de ses références ou certains de ses pointeurs à une valeur nulle suite à un constructeur, alors il devient impératif de tester **systématiquement** pour cet état dans **chaque** méthode d'instance publique (et, selon les cas, dans certaines méthodes privées ou protégées [POOv01]).

Toute classe négligeant d'identifier clairement et explicitement les postconditions de tous ses constructeurs souffre d'une grande fragilité, que ce soit à cause de constructeurs incomplets ou que ce soit dû à des postconditions non identifiées. Il devient alors difficile d'écrire des objets robustes et capables d'auto validation.

Réflexion 00.7 : erreurs à la construction

Ajouter une référence à un booléen ou à un code d'erreur aux constructeurs est-il une option valide pour détecter et gérer les erreurs et les cas exceptionnels survenant pendant la phase de construction d'un objet?

Cette approche est-elle applicable si un problème survient dans un constructeur?
--

La réponse est un retentissant **non**, du moins pour qui souscrit au principe d'encapsulation. En effet, un objet est présumé responsable de son intégrité, du début à la fin de sa vie. Si un objet a été construit, le programme doit le présumer valide, en vertu de ce principe.

Conséquence logique : si un problème survient pendant la construction d'un objet, et si ce problème est tel que la construction ne sera pas complétée adéquatement, alors *il est important que la construction ne se termine pas*.

La beauté des exceptions est que, si un constructeur lève une exception, alors l'objet en cours de construction n'aura pas été construit. Seuls les objets valides intégreront le système. Lorsque nous aurons examiné l'héritage [POOv01], les vertus de cette façon de faire seront encore plus apparentes.

Si un objet n'a pu être construit, règle générale, il n'est pas en mesure de gérer les raisons du problème et de les mettre en contexte. Il peut reconnaître le problème et le signaler, toutefois, de manière à ce que le code client puisse faire un diagnostic convenable. Le traitement d'exceptions est nécessaire à la mise en place d'une approche OO saine et complète.

Annexe 03 – Concepts et pratiques du langage C++

Vous trouverez ci-dessous quelques informations sur des concepts associés à la programmation en C++ et sur certaines pratiques idiomatiques de ce langage.

Idiome RAII

Les langages reposant sur un moteur de collecte automatique d'ordures ont deux gros avantages : ils évitent les fuites de mémoire dans les programmes conventionnels¹⁹¹, ramassant les objets qui ne sont plus référencés au moment qui leur semble opportun, et facilitent le partage d'objets alloués dynamiquement¹⁹². Ils ont aussi un gros désavantage, qui limite la capacité des objets à implémenter une encapsulation complète : le moment de la collecte des objets est sous la gouverne du moteur de collecte d'ordures, ce qui rend cette mécanique indéterministe au sens du programme et réduit l'utilité de finisseurs.

Conséquence : dans un langage reposant sur un moteur de collecte d'ordures, le code client doit typiquement gérer le nettoyage des ressources des objets à la place de ceux-ci (fermer les fichiers, les connexions aux bases de données, les liens de communication, *etc.*). Rien n'est parfait : en réduisant les fuites de mémoire, ces approches forcent le code client à se mêler de dossiers qui ne le regardent pas. Les blocs `finally` sont un symptôme de ce choix philosophique.

De manière générale, la pratique en C++ est de faire en sorte que les ressources soient gérées de manière rigoureuse et déterministe dans la mesure du possible. En temps normal, dans ce langage, le code client ne devrait jamais avoir à fermer ou à nettoyer explicitement une ressource, outre pour des fins techniques (si, par exemple, un fichier doit être fermé puis ouvert à nouveau à travers un même objet sur un compilateur pour lequel ceci serait plus rapide que de laisser l'objet se détruire puis d'en reconstruire un nouveau).

Attribuer la responsabilité de la gestion d'une ressource à un objet, typiquement lors de sa construction, de sorte que cet objet puisse assurer le bon nettoyage de la ressource à la fin de sa vie, est un idiome de programmation, donc une pratique répandue pour un groupe de langages (les langages OO avec destruction déterministe) qu'on nomme RAII, pour *Resource Acquisition is Initialization*.

¹⁹¹ Oui, il est possible (et facile) d'épuiser la mémoire d'un programme Java et d'un programme .NET. Il ne faut pas abuser des ressources d'un programme, même si en théorie quelqu'un d'autre surveille et cherche à nous aider.

¹⁹² En effet, quand plusieurs entités (plusieurs *threads*, par exemple) partagent un même objet, il est parfois complexe de déterminer qui devra le libérer. Plusieurs solutions existent à ce problème (nommer un seul responsable et faire en sorte qu'il ne détruise pas l'objet avant qu'il ne soit certain que plus personne d'autre ne s'en sert, par exemple, ou compter les références sur l'objet pour qu'il ne décède que quand le compteur de références devient nul), mais elles peuvent être complexes à implémenter; il est clair qu'une solution gérée implicitement par le langage règle ce problème d'office.

Un exemple simple d'approche RAII associée à la libération automatique de ressources serait celui-ci-dessous (l'exemple à gauche n'applique pas l'idiome, alors que l'exemple à droite, quant à lui, le fait).

Sans RAII	Avec RAII
<pre>int connecter_BD(const string &); void fermer_connexion_BD(int); void f(int); // que fait-elle? void acceder_BD(const string &nom) { int id = connecter_BD(nom); // présumons que id est légal f(id); fermer_connexion_BD(id); // risqué }</pre>	<pre>int connecter_BD(const string &); void fermer_connexion_BD(int); void f(int); // que fait-elle? class AutoFermeur { int id_; public: AutoFermeur(int id) : id_{id} { } int id() const { return id_; } ~AutoFermeur() { fermer_connexion_BD(id()); } }; void acceder_BD(const string &nom) { AutoFermeur af{connecter_BD(nom)}; // présumons que id soit légal f(af.id()); }</pre>

L'exemple à droite semble plus long que l'exemple à gauche, mais c'est trompeur, et ce pour plusieurs raisons :

- la fonction `acceder_BD()`, qui fait le travail, est plus courte à droite qu'à gauche;
- la classe `AutoFermeur`, un objet RAII, est écrite une fois mais réutilisable plusieurs fois;
- plus important, la fonction `acceder_BD()` à droite est plus sécuritaire qu'à gauche. En effet, quoiqu'il arrive pendant l'appel de `f()` par `acceder_BD()`, **incluant des cas d'exceptions**, les variables locales à `acceder_BD()` seront détruites;
- ainsi, dans la version à gauche, il peut arriver que la connexion à la BD ne soit pas libérée correctement (si `f()` lève une exception) alors que dans la version à droite, la connexion sera fermée quoiqu'il arrive (hormis une panne de courant ou un bris matériel) par le destructeur de l'objet `af`.

Voir [hdIdiom] pour en savoir plus. Des exemples plus riches de cet idiome apparaissent dans les volumes ultérieurs de cette série.

ODR – la One Definition Rule

L'une des règles sémantiques fondamentales de C++ est la règle ODR, ou *One Definition Rule*. Cette règle stipule qu'il ne peut y avoir qu'une seule définition pour toute entité dans un programme C++ valide. Il peut toutefois y avoir plusieurs déclarations pour une même entité.

À titre d'exemple, si un programme est fait des trois fichiers ci-dessous, une erreur à l'édition des liens sera signalée dû à une violation de la règle ODR (l'opérateur de transtypage `static_cast` est décrit dans [POOv01]) :

a.h	b.cpp	c.cpp
<pre>#ifndef A_H #define A_H int f(double d) { return static_cast<int>(d)+1; } void g(); #endif</pre>	<pre>#include "a.h" void g() { }</pre>	<pre>#include "a.h" int main() { int x = f(3.5); }</pre>

La raison de cette violation est que `f()` y est non seulement déclarée mais aussi définie dans `a.h`. Puisque `b.cpp` et `c.cpp` incluent tous deux cette définition et puisque les deux sont compilés séparément, l'édition des liens trouvera deux définitions de `f()`, soit une dans `b.obj` et une autre dans `c.obj`.

En retour, si un programme est fait des trois fichiers ci-dessous, la compilation et l'édition des liens réussiront toutes deux :

a.h	b.cpp	c.cpp
<pre>#ifndef A_H #define A_H int f(double d); void g(); #endif</pre>	<pre>#include "a.h" int f(double d) { return static_cast<int>(d)+1; } void g() { }</pre>	<pre>#include "a.h" int main() { int x = f(3.5); }</pre>

En effet, `b.cpp` et `c.cpp` incluront tous deux la déclaration de `f()` mais une seule définition de cette fonction existera (celle dans `c.obj`, résultant de la compilation de `c.cpp`). Le code de `b.cpp` compilera car l'appel de `f()` est conforme à la déclaration de cette fonction (dans `a.h`) et l'édition des liens résoudra l'appel de `f()` dans `b.obj` vers la définition de `f()` dans `c.obj`.

La règle ODR a été héritée du langage C, et elle s'applique donc aux fonctions globales. Elle s'applique aussi aux attributs de classe, qu'ils soient constants ou non (avec une exception possible pour les constantes d'instance si celles-ci sont d'un type primitif entier) mais pas aux définitions de méthodes lorsque celles-ci se situent à même la déclaration de la classe. Elle ne s'applique pas non plus au code générique [POOv02].

Ainsi, le code ci-dessous n'est pas en violation de la règle ODR :

X.h	Test.cpp
<pre>#ifndef X_H #define X_H class X { public: // Ok (mène à du inlining) int f() const { return 3; } }; #endif</pre>	<pre>#include "X.h" int main() { X x; int i = x.f(); }</pre>

Le code ci-dessous n'est pas non plus en violation de la règle ODR :

X.h	X.cpp	Test.cpp
<pre>#ifndef X_H #define X_H class X { public: // Ok (déclaration seule) int f() const; }; #endif</pre>	<pre>#include "X.h" // Ok (definition seule) int X::f() const { return 3; }</pre>	<pre>#include "X.h" int main() { X x; int i = x.f(); }</pre>

Toutefois, le code ci-dessous est en violation de la règle ODR car la définition de `X::f()` est placée hors de la déclaration de la classe `X`, et sera traitée par le compilateur et l'éditeur de liens selon les règles applicables aux fonctions globales.

X.h	Test.cpp
<pre>#ifndef X_H #define X_H class X { public: int f() const { // Ok (déclaration) return 3; } }; // Pas Ok (mène à une violation // de la règle ODR dès que X.h sera // inclus par deux .cpp d'un projet int X::f() const { return 3; } #endif</pre>	<pre>#include "X.h" int main() { X x; int i = x.f(); }</pre>

Sainte-Trinité

C++ offre un support spécial à trois méthodes d'instance, qu'on nomme la Sainte-Trinité (ou la « règle de trois »). Ces opérations sont le constructeur de copie, l'affectation et la destruction.

Si un type programme ne déclare pas l'une ou l'autre de ces opérations, le compilateur C++ le fera pour lui, à partir des constructeurs de copie de ses attributs, de l'opérateur d'affectation de ses attributs et du destructeur de ses attributs.

Les opérations de la Sainte-Trinité vont de pair : en définir une implique *généralement* définir les trois, et en laisser une dans son état généré automatiquement par le compilateur implique *habituellement* n'en définir aucune. Les cas où on ne générera qu'une ou deux d'entre elles existent mais ils sont peu fréquents.

Y a-t-il un indice clair qu'il y ait lieu de définir de manière plus précise (et plus manuelle) ces opérations pour une classe donnée? *Bjarne Stroustrup* et *Gabriel Dos Reis*¹⁹³, proposent cette maxime toute simple : *les opérations de copie par défaut sont en général inadéquates lorsque la classe implémente un destructeur.*

Autres consignes : quand un objet a un attribut dont le type est une référence ou un attribut qui est une constante d'instance, alors il est possible que le constructeur par copie ait un sens mais il est *hautement improbable* que l'affectation doive être implémentée.

C'est effectivement une bonne maxime : en général, si on ressent le besoin de spécifier un destructeur, il faut aussi implémenter les opérations de copie. Si le destructeur est redondant, il est possible que la copie manuelle le soit aussi. Il y a évidemment des cas d'exception, mais la maxime demeure un guide correct dans la plupart des cas.

Pris sous un autre angle : les usages des programmeuses et des programmeurs ○○ se sont raffinés avec les années, et certaines pratiques se sont clarifiées avec l'expérience. Il en va de même pour le vocabulaire employé pour décrire ces pratiques. On n'a qu'à penser aux idiomes de programmation [hdIdiom], qui décrivent des façons de faire propres à un langage ou à un groupe de langages, et aux schémas de conception [hdPatt], qui décrivent des façons de faire applicables à l'aide des outils offerts par la grande majorité des langages de programmation.

Dans la majorité des cas, les problèmes associés au nettoyage des ressources ont un impact sur la mécanique de copie : si un objet responsable d'un pointeur sur une entité allouée dynamiquement est copié, il y a un risque que sa copie libère les ressources à sa place, entraînant une double libération de ressources (chose susceptible de briser un programme).

Nous ferons un retour sur la question de la Sainte-Trinité dans [POOv02] lorsque nous aborderons la question de la sémantique de mouvement.

¹⁹³ Voir <http://public.research.att.com/~bs/N1890-initialization.pdf>, p. 18.

Annexe 04 – Principe de localité et efficacité

Cette annexe est née d'une question reçue d'un de mes anciens étudiants du nom de **François Vaillancourt**, que je remercie au passage. François cherchait à définir un standard de programmation dans son entreprise et se questionnait sur le caractère pragmatique ou dogmatique du principe de localité.

Une question qui survient occasionnellement dans les discussions est l'opposition entre le **principe de localité**, selon lequel un nom (typiquement : une variable) devrait avoir la plus petite portée raisonnable lui permettant de jouer son rôle dans un programme, et l'efficacité du programme qui peut être affectée par le nombre d'appels aux constructeurs et aux destructeurs des objets dans ces portées.

À titre d'exemple, soit les quatre exemples suivants, et supposant que `X` soit un type dont le constructeur est complexe :

Exemple A	Exemple B
<pre>int i = 0; while(i < 100) { int var = 3; i++; }</pre>	<pre>int i = 0; while(i < 100) { X obj{3}; i++; }</pre>
Exemple C	Exemple D
<pre>int i = 0; int var; while(i < 100) { var = 3; i++; }</pre>	<pre>int i = 0; X obj; while(i < 100) { obj = X{3}; i++; }</pre>

À quoi peut-on s'attendre du code généré par un compilateur contemporain? Si les exemples A et B sont plus lents que les exemples C et D, cela signifie-t-il que le principe de localité s'oppose à l'efficacité du code généré?

Pour répondre à ces questions, il faut d'abord faire l'hypothèse que le code est bien écrit dans tous les cas pour l'affectation, la construction (par défaut, paramétrique et par copie, car les trois interviennent ici) et la destruction. Si on suppose que le programmeur n'a pas fait son travail dans l'un ou l'autre de ces cas, alors mieux vaut régler le « problème d'expertise » d'abord.

Remarques préliminaires

Quelques remarques préliminaires s'imposent :

- pour mener des tests sur la vitesse d'exécution d'un programme, il faut absolument utiliser la version optimisée du code généré. Cela fait un monde de différence, et les tests ne sont réalistes que dans ce cas;
- dans chaque cas, il est essentiel que les variables initialisées pour les tests soient utilisées en pratique par la suite, et que les résultats entraînent des conséquences dans le programme, par exemple en provoquant une entrée/ sortie (dont vous éviterez soigneusement de tenir compte du temps d'exécution pour vos tests, car ces opérations sont habituellement très lentes). Un compilateur C++ éliminera habituellement le code « inutile » dans un programme lorsqu'il optimisera le fruit de son labeur, et vos tests vous diront, de manière erronée, que le programme s'exécute de manière instantanée.

Primitifs

Pour les primitifs, par définition, nous sommes dépendants de la qualité de l'implémentation. Cela dit, en pratique, les exemples A et C devraient donner du code équivalent une fois compilé, du fait qu'il n'y a pas réellement de « construction » à réaliser (l'attribution d'une valeur dans une case mémoire implique habituellement le même code machine pour une construction ou pour une affectation).

Conséquemment, dans un cas comme celui présenté ici, j'appliquerais le principe de localité et je préférerais A à C, pour réduire à l'essentiel la visibilité de `var`.

Objets

Pour les objets, la question posée ici est essentiellement à savoir lequel de ceci...

```
X x;
for (int i = 0; i < BEAUCOUP; ++i) {
    x = 3; // ceci suppose X::operator=(int)
}
```

... et de cela...

```
for (int i = 0; i < BEAUCOUP; ++i) {
    X x = 3; // ceci suppose X::X(int)
}
```

... mènera au code compilé le plus rapide. À cette question, nous donnerons d'abord une réponse générale, puis nous examinerons quelques spécificités plus subtiles.

Réponse générale

Au risque de surprendre, notons d'office que de manière générale, à code équivalent, pour une classe qui est responsable de ses états et implémente la Sainte-Trinité (voir *Sainte-Trinité*), il devrait être plus rapide de construire dans la boucle que d'y affecter.

En effet, imaginons ceci :

```
X x = 3; // X::X(int)
X x2; // X::X()
for (int i = 0; i < BEAUCOUP; ++i) {
    x2 = x; // X::operator=(const X&)
}
```

L'opérateur d'affectation a pour tâche de remplacer le contenu de l'opérande de gauche par une copie du contenu de l'opérande de droite. Ainsi, l'instruction `x2 = x;` doit en fait réaliser :

- l'équivalent de `X::~~X()` pour le nettoyage de `x2`;
- l'équivalent de `X::X(const X&)` pour la copie du contenu de `x` dans `x2`; et
- un test pour éviter que l'opérande de gauche soit corrompu si quelqu'un écrivait `x = x;`, une opération légale mais qui peut prêter à conséquence si l'affectation est mal écrite.

L'idiome d'affectation sécuritaire (voir *On peut toutefois* faire encore mieux...

L'idiome d'affectation sécuritaire) résout ce dernier problème, combinant la construction par copie, la destruction et une opération `swap()` de complexité constante. Ce petit surcoût n'interviendra pas si l'objet est construit puis détruit à chaque itération.

Variations selon les types de constructeurs

Si nous utilisons un constructeur autre que le constructeur de copie, certaines considérations s'ajoutent. Imaginons un type `X` tel que :

```
class X {
    // ...
public:
    X(int);
    X &operator=(int);
    // ...
};
```

Alors le code suivant...

```
X x = 3; // X::X(int)
x = 4; // X::operator=(int)
```

...sollicitera directement deux méthodes sur lesquelles nous avons le plein contrôle.

Cela dit, imaginons maintenant que `X` soit :

```
class X {
    // ...
public:
    X(int);
    X &operator=(const X&);
    // ...
};
```

Alors le code suivant...

```
X x = 3; // X::X(int)
x = 4; // X::X(int) puis X::operator=(const X&)
```

...sollicitera trois méthodes, passant par le constructeur paramétrique de `X` pour créer un `X` à partir du `int` de valeur 4 avant de procéder à l'affectation. C'est ce que fait le moteur d'inférence de types du langage, sans grande surprise. Il y a un coût caché ici, qu'il est possible d'éviter (ou du moins, pour lequel il est possible de conscientiser le code client) en qualifiant le constructeur paramétrique du mot clé `explicit` (voir *Construction implicite ou construction explicite* pour plus de détails).

Par exemple, cet ajustement...

```
class X {
    // ...
public:
    explicit X(int);
    X &operator=(const X&);
    // ...
};
```

... aurait pour conséquence de forcer le code client à expliciter son intention :

```
X x = 3; // X::X(int)
// x = 4; // illégal!
x = X{4}; // X::X(int) puis X::operator=(const X&), mais c'est voulu
```

Implémentations par défaut

Quand tous les attributs d'instance d'une classe se comportent comme des valeurs, et quand leur Sainte-Trinité est bien codée, cette classe ne devrait pas implémenter ses propres versions des méthodes de la Sainte-Trinité. Il se trouve que le compilateur, par défaut, générera du meilleur code en supposant que tout est banal.

Individus

Les individus suivants sont mentionnés dans le présent document. Vous trouverez, en suivant les liens proposés à droite du nom de chacun, des compléments d'information à leur sujet et des suggestions de lectures complémentaires. Avis aux curieuses et aux curieux!

<i>Dave Abrahams</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#david_abrahams
<i>Andrei Alexandrescu</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrei_alexandrescu
<i>Jeff Atwood</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#jeff_atwood
<i>Confucius</i>	http://fr.wikipedia.org/wiki/Confucius
<i>Larry Constantine</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#larry_constantine
<i>Gabriel Dos Reis</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#gabriel_dos_reis
<i>Vincent Echelard</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#vincent_echelard
<i>Richard Gabriel</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#richard_gabriel
<i>Aleksey Gurtovoy</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#aleksey_gurtovoy
<i>Maurice P. Herlihy</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#maurice_herlihy
<i>Andrew Hunt</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrew_hunt
<i>Jon Kalb</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#jon_kalb
<i>Andrew Koenig</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrew_koenig
<i>Eric Lippert</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#eric_lippert
<i>Bertrand Meyer</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bertrand_meyer
<i>Scott Meyers</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#scott_meyers
<i>Charles Petzold</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#charles_petzold
<i>Pierre Prud'homme</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#pierre_prudhomme
<i>Dennis Ritchie</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#dennis_ritchie
<i>Charles Simonyi</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#charles_simonyi
<i>Richard Smith</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#richard_smith
<i>Alexander Stepanov</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#alexander_stepanov
<i>Bjarne Stroustrup</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bjarne_stroustrup
<i>Herb Sutter</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#herb_sutter
<i>Dave Thomas</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#dave_thomas
<i>Niklaus Wirth</i>	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#niklaus_wirth

Références

Les références qui suivent respectent un format quelque peu informel.

Elles vous mèneront soit à des notes de cours de votre humble serviteur, soit à des documents pour lesquels mes remarques sont proposées de manière électronique et à partir desquels vous pourrez accéder aux textes d'origine ou à des compléments d'information.

- [AbrSpdVal] <http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/>
- [ArtMPP] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#art_of_multiprocessor_programming
- [CppCodStd] <http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-coding-standards>
- [CppMeta] <http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-template-metaprogramming>
- [EffCpp] <http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-cpp>
- [EffStl] <http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-stl>
- [ExcCpp] <http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#exceptional-cpp>
- [hdCppNull] <http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/CPP--NULL.html>
- [hdExcep] <http://h-deb.clg.qc.ca/Sujets/Developpement/Exceptions.html>
- [hdIdiom] <http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html#idiome>
- [hdLoc] http://h-deb.clg.qc.ca/Sujets/Developpement/Lois-principes.html#principe_localite
- [hdNom] <http://h-deb.clg.qc.ca/Sujets/Developpement/Nomenclature.html>
- [hdOptD] <http://h-deb.clg.qc.ca/Sujets/Developpement/Technique-Optimisation-savoir-discret.html>
- [hdPatt] <http://h-deb.clg.qc.ca/Sujets/Developpement/Schemas-conception.html>
- [hdPrin] <http://h-deb.clg.qc.ca/Sujets/Developpement/Pratique-programmation.html#principes>
- [hdProp] <http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Proprietes.html>
- [hdSym] <http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/CPP--Exploiter-Symetrie.html>
- [hdTri] <http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Sainte-Trinite.html>
- [hdUB] http://h-deb.clg.qc.ca/Sujets/Developpement/Comportement_indefini.html
- [JavaOpOv] <http://java.sun.com/docs/white/langenv/Simple.doc2.html>
- [JavaStyle] <http://developers.sun.com/sunstudio/products/archive/whitepapers/java-style.pdf>
- [LippValSem] <http://blogs.msdn.com/b/ericlippert/archive/2010/10/11/debunking-another-myth-about-value-types.aspx>
- [NetStack] <http://blogs.msdn.com/ericlippert/archive/2009/04/27/the-stack-is-an-implementation-detail.aspx>
- [POOv01] POO – Volume 01, par Patrice Roy et Pierre Prud'homme.
- [POOv02] POO – Volume 02, par Patrice Roy et Pierre Prud'homme.
- [POOv03] POO – Volume 03, par Patrice Roy et Pierre Prud'homme.
- [ReadOnly] <http://www.bluebytesoftware.com/blog/2010/07/01/WhenIsAReadOnlyFieldNotReadOnly.aspx>
- [StepDobb] <http://www.sgi.com/tech/stl/drdoobbs-interview.html>
- [StepInf] <http://www.informit.com/articles/article.aspx?p=2314360>
- [StrouSlow] http://www.stroustrup.com/bs_faq2.html#slow-containers

- [SutterConst] <http://www.gotw.ca/publications/advice98.htm>
- [SutterExCtor] <http://herbsutter.wordpress.com/2008/07/25/constructor-exceptions-in-c-c-and-java/>
- [SutterHung] <http://herbsutter.wordpress.com/2008/07/15/hungarian-notation-is-clearly-goodbad/>
- [ValVsRef] <http://www.parashift.com/c++-faq-lite/value-vs-ref-semantics.html>
- [WorseBetter] <http://dreamsongs.com/WorseIsBetter.html>